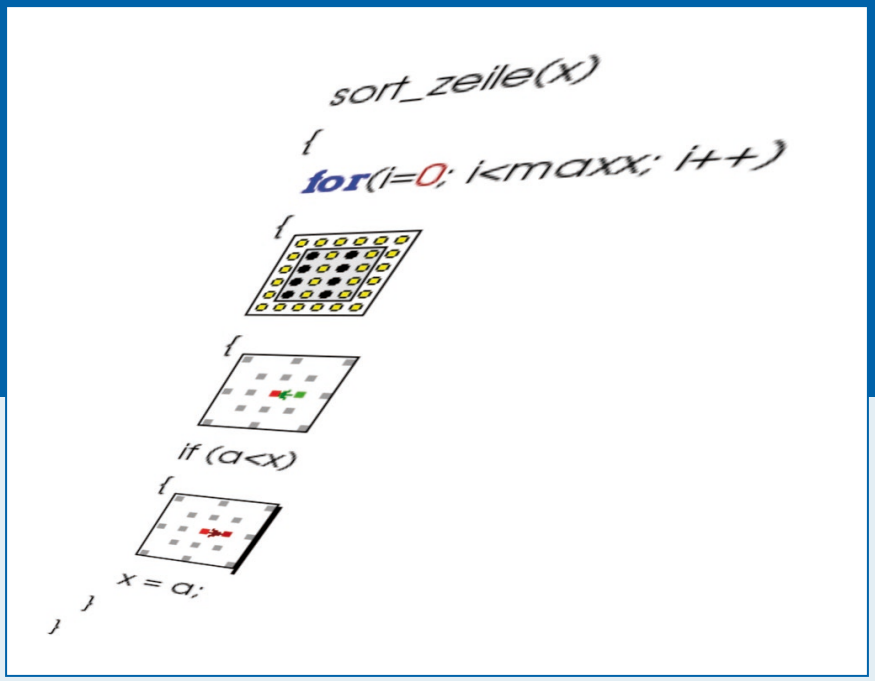


Frederic Toussaint

Grafische Benutzungsunterstützung auf Befehlsebene für die Entwicklung massivparalleler Programme

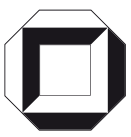
universitätsverlag karlsruhe

Frederic Toussaint

Grafische Benutzungsunterstützung auf Befehlsebene für die Entwicklung massivparalleler Programme

Grafische Benutzungsunterstützung auf Befehlsebene für die Entwicklung massivparalleler Programme

von
Frederic Toussaint



universitätsverlag karlsruhe

Dissertation, genehmigt von der Fakultät für Wirtschaftswissenschaften der
Universität Fridericiana zu Karlsruhe, 2007
Referenten: Prof. Dr. H. Schmeck, Prof. Dr. M. Schimmler

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Universitätsverlag Karlsruhe 2007
Print on Demand

ISBN: 978-3-86644-130-9

Inhaltsverzeichnis

II Abbildungsverzeichnis

III Tabellenverzeichnis

1 Einleitung

1.1 Motivation

„Nicht ein einziger Einkern-Chip wird präsentiert, alle Forscher arbeiten an Mehrkern-Designs“. So steht es im Februar 2006 in einem Online-Artikel zur Vortragsreihe über Prozessoren auf der IEEE Halbleiterkonferenz „International Solid-State Circuits Conference“ (ISSCC) [Hei06a]. Etwa zur gleichen Zeit stellt der Chief Technology Officer von Intel auf dem Intel Developer Forum seinen „Weg zu Hunderten von Prozessorkernen“ vor [Hei06b]. Auch die Firma Rapport zeigt auf der Embedded-Systems-Konferenz 2006 in San Jose ein Prozessordesign mit insgesamt 1025 Kernen [Hei06c].

Diese Auswahl an aktuellen Pressemitteilungen zeigt, dass zurzeit die Entwicklung in Richtung Mehrprozessortechnik sehr stark ist, da beim aktuellen Stand der Forschung keine Firma Einprozessorsysteme herstellen kann, welche die stark gestiegenen Anwenderanforderungen erfüllen. Vor einigen Jahren war dieser Trend zur Mehrprozessortechnik bereits bei sehr rechenintensiven Forschungsbereichen vorhanden, wodurch sehr leistungsstarke Computer mit mehreren Tausend Prozessoren entstanden sind. Diese sogenannten massivparallelen Rechner waren alle Eigenentwicklungen einzelner Firmen, so dass sie nicht kompatibel zueinander waren. Damit die Forscher mit den Computern arbeiten konnten, wurden proprietäre Entwicklungsumgebungen zur Verfügung gestellt, welche die jeweiligen Besonderheiten der Systeme unterstützen sollten. Neben einem meist rudimentären Editor zur Programmeingabe wurden häufig Tools zur Verfügung gestellt, welche ein fertiges Programm analysieren konnten, um dessen Performance zu testen. Teilweise wurden dabei bereits grafische Elemente wie Balken- und Flussdiagramme eingesetzt. Damals wie auch heute bietet allerdings keine der Entwicklungsumgebungen die Möglichkeit an, den Benutzer bei der Eingabe seiner

Befehlszeilen durch grafische Elemente zu unterstützen. Diese grafische Unterstützung bei der Eingabe eines einzelnen Befehls ist Ziel dieser Arbeit.

Motiviert wurde diese Arbeit durch Forschungsarbeiten im Bereich genetische Algorithmen am Lehrstuhl Effiziente Algorithmen des Instituts für Angewandte Informatik und Formale Beschreibungsverfahren an der Universität Karlsruhe (TH). Bei diesen Forschungstätigkeiten wurde sehr häufig der Massivparallelrechner MasPar eingesetzt, welcher mit 16.384 Prozessorelementen ausgerüstet war. Die erforderlichen Algorithmen wurden in den meisten Fällen entwickelt, indem Mitarbeiter auf Papier Skizzen zeichneten, auf denen Populationen bestimmten Prozessoren zugeordnet wurden. Gleichzeitig wurde skizziert, auf welchem Weg die einzelnen Populationen miteinander kommunizieren sollten. Dieses entstandene grafische Abbild eines Algorithmus musste anschließend in eine Quelltextform umgesetzt werden. Da der eingesetzte Editor bei der Entwicklung von parallelen Programmen nur sehr wenig Unterstützungsmöglichkeiten bot, wurde der Wunsch nach einem „besseren“ Editor laut. Als Vergleich konnten Entwicklungsumgebungen für PCs herangezogen werden, beispielsweise Turbo Pascal von Borland, welches eine integrierte Hilfe anbot, den direkten Aufruf des Compilers unterstützte und viele weitere Hilfestellungen zur Verfügung stellte. Somit war das Ziel der neuen Entwicklungsumgebung einerseits den „Komfort“ von PC-Entwicklungsumgebungen auf den Großrechner zu übertragen und – noch wichtiger – andererseits eine konkrete Hilfe anzubieten, welche den Programmierer bei der Erstellung der parallelen Programmteile unterstützt. Da diese parallelen Programmteile, wie oben gezeigt, häufig bereits mittels Skizzen vorbereitet wurden, sollte ein Konzept entwickelt werden, welches es erlaubt, mithilfe dieser Skizzen gezielt die jeweiligen Befehlszeilen zu entwickeln.

1.2 Abgrenzung

Ziel der Arbeit ist es Möglichkeiten vorzustellen, wie ein Entwickler von parallelen Programmen bei seinen täglichen Aufgaben durch Hilfestellungen unterstützt werden kann. Dabei wird ein besonderes Augenmerk auf die Unterstützung zur Erstellung von einzelnen Befehlen gelegt. Auf dieser Befehlsebene ist es wichtig, dass der Anwender auf einfache Weise Befehle in korrekter Syntax erzeugen kann. Neben der syntaktischen Korrektheit spielt auch eine korrekte Semantik eine wichtige Rolle. Damit hier der Benutzer eine bestmögliche Unterstützung findet, wird ein Verfahren vorgestellt, durch das parallele Programmteile mittels grafisch unterstützten Dialogboxen erzeugt werden können. Um anschließend den Überblick über das erzeugte Programm zu behalten, wird eine Darstellungsform entwickelt, die wichtige parallele Programmeigenschaften innerhalb des Quelltextes grafisch aufbereitet darstellt. Bei allen Arbeiten wird besonders darauf geachtet, dass der Benutzer mit ergonomisch sinnvollen Hilfsmitteln unterstützt wird.

Die Arbeit erhebt keinen Anspruch darauf, eine neue, vollständig funktionsfähige Entwicklungsumgebung zur Verfügung zu stellen, welche für alle modernen parallelen Rechner eingesetzt werden kann und dabei alle Benutzerunterstützungen moderner

Entwicklungsumgebungen bereit hält. Deshalb wird an einem Prototyp gezeigt, welche Möglichkeiten besonders bei der Unterstützung von parallelen Programmteilen auf Befehlebene nach Meinung des Autors sinnvoll eingesetzt werden können. Zusätzlich wird anhand von Designstudien und Skizzen gezeigt, wie ähnliche Hilfestellungen auf weitere Typen von Parallelrechnern angewandt werden können.

In weiten Teilen beschränkt sich die Arbeit auf Hilfestellungen, die bei der Befehlseingabe genutzt werden können. Andere Phasen der Programmentwicklung, wie die Problemanalyse, das Testen und Beschleunigen oder die Wartung, werden in der Arbeit nur eingeschränkt betrachtet. Außerdem wird der Schwerpunkt auf die Unterstützung von SIMD-Rechnern (Single Instruction – Multiple Data) gelegt, worunter man nach Waldschmidt Rechner mit einer „Architekturform versteht …, die als Kontrollstruktur nur einen Informationsstrom besitzen, aber mit einer Instruktion mehrere Datenelemente einer Datenstruktur verarbeiten können“ [Wal95]. Die Abgrenzung von SIMD-Rechnern zu MIMD, SISD und MISD hat Michael J. Flynn bereits 1966 durchgeführt [Fly66].

1.3 Aufbau

Diese Arbeit gliedert sich in insgesamt acht Kapitel. Begonnen wird sie mit einer Einleitung, welche aus einem Motivationsbereich besteht, der kurz Statements aktueller Konferenzen aufgreift und die Notwendigkeit einer geeigneten Entwicklungsumgebung schildert. Anschließend folgen eine kurze Abgrenzung und diese Beschreibung des Aufbaus der Arbeit.

Im zweiten Kapitel werden Grundlagen vorgestellt, welche die Voraussetzungen für ergonomisch gestaltete Programme bilden. Dabei wird noch nicht auf spezielle parallele Eigenschaften eingegangen. Vielmehr werden allgemeine Richtlinien beschrieben, angefangen von der Notwendigkeit von Benutzermodellen, über Modellen für Benutzungsschnittstellen, Normen für die Dialoggestaltung und Gestaltgesetze, sowie Werkzeugen, welche für die Programmentwicklung benötigt werden, bis hin zu einzusetzenden Entwurfsmustern und der Notwendigkeit einer Barrierefreiheit.

Im Anschluss folgt ein Kapitel, welches einige massivparallele Rechner vorstellt (MasPar, Systola, RMesh). Dabei wird ein Modell für einen parallelen Algorithmus erarbeitet, welches einen groben, aber allgemeingültigen Programmablauf auf allen Parallelrechnern beschreibt. Wichtig ist dabei die Herausstellung von Aktivierungs- und Kommunikationsphasen. Das Modell des parallelen Algorithmus wird an Beispielalgorithmen überprüft, welche auf den vorgestellten Parallelrechnern ablaufen.

Kapitel Vier ist dem Stand der Technik gewidmet. Dabei werden ausgewählte Entwicklungsumgebungen vorgestellt. Begonnen wird mit einem allgemeinen Teil, welcher Gemeinsamkeiten aller Entwicklungsumgebungen enthält. Anschließend folgt ein Abschnitt, der Entwicklungsumgebungen für Einprozessormaschinen mit besonderen Eigenschaften beschreibt. Parallele Entwicklungsumgebungen werden im Anschluss unterschieden nach der benutzten Plattform und welche Phasen der Programmentwicklung sie besonders unterstützen. Abgeschlossen wird das Kapitel mit zwei Vorstellungen

von Entwicklungsumgebungen, welche als Vergleich bzw. Vorlage für die in dieser Arbeit entwickelten grafischen Benutzungsunterstützungen dienen sollen.

Im darauf folgenden Kapitel Fünf werden vielfältige Hilfestellungen für Benutzer ausführlich vorgestellt und dabei neue Möglichkeiten erarbeitet. Begonnen wird mit einem Überblick bekannter Hilfestellungen. Anschließend folgen Betrachtungen zu Hilfestellungen, die den Benutzer besonders bei der Eingabe von Befehlen im Quelltext unterstützen, ohne dass dabei grafische Hilfsmittel notwendig sind. Grafische Komponenten werden anschließend zusätzlich untersucht, um eine noch bessere Unterstützung bieten zu können. Dabei wird auch gezeigt, wie eine Anreicherung des Quelltextes mit dynamisch erzeugten Icons zur Darstellung ausgewählten Quellcodes diesen übersichtlicher und dadurch auch besser auf semantische Fehler überprüfbar machen kann. Besonderes Augenmerk wird hierbei auf die Unterstützungsmöglichkeiten gelegt, welche die Aktivierungs- und Kommunikationsphasen auf Parallelrechnern abbilden und die bis jetzt auf diese Weise noch nicht bekannt waren.

Der Realisierbarkeit der vorgestellten Hilfestellungen ist Kapitel Sechs gewidmet. Hier wird ein Prototyp für einen Editor vorgestellt, welcher von Grund auf den Anforderungen angepasst ist und in dem die im vorherigen Kapitel vorgestellten Hilfestellungen implementiert wurden. Dabei wird gezeigt, welche Eigenschaften ein Editor besitzen muss, um sinnvoll grafische Elemente integrieren zu können.

Nach der ausführlichen Darstellung einer vollständigen Implementierung eines Editors für einen Parallelrechner werden in Kapitel Sieben eine Designstudie für einen zweiten und Ideenskizzen für weitere Typen von Parallelrechnern vorgestellt, wobei jeweils die besonderen Eigenschaften dieser Rechner bestmöglich unterstützt werden sollen. Abgeschlossen wird das Kapitel mit der Vorstellung einer Möglichkeit, wie man bestehende Editoren um grafische Hilfestellungen erweitern kann, obwohl diese bisher nicht unterstützt werden.

Die Arbeit schließt in Kapitel Acht mit einer Zusammenfassung und einem Ausblick, sowie einem Literaturverzeichnis.

2 Ergonomiegrundlagen für die Benutzung von Computerprogrammen

Bevor mit der Codierung von Programmen begonnen werden kann, müssen viele Vorbereitungen getroffen werden. Dabei muss man sich unter anderem über folgende Punkte Gedanken machen:

- Für wen soll das Programm entwickelt werden?
- Über welche Schnittstellen soll der Benutzer das Programm bedienen können?
- Welche allgemeinen ergonomischen Richtlinien sind zu berücksichtigen?
- Welche Software sollte eingesetzt werden?
- Gibt es Muster, wie bestimmte Aufgaben im Allgemeinen durchgeführt werden können?

Um diese Fragen zu klären, muss man sich mit verschiedenen theoretischen Grundlagen beschäftigen. Die Frage nach dem „Für wen“ kann durch den Einsatz eines Benutzermodells geklärt werden. Die Schnittstellenauswahl, bzw. die richtige Einbindung in den Programmcode wird durch unterschiedliche in der Literatur beschriebene Standards für Benutzungsschnittstellen definiert. Bei jeder Schnittstelle zu einem Benutzer ist die Ergonomie zu beachten, die unter anderem durch einige DIN- und ISO-Normen geregelt ist. Weiter bleibt die Frage, welche Software für die Umsetzung eingesetzt werden sollte. Hier kann man grob unterscheiden, wie hoch der Umfang der Benutzungsunterstützung ist und welche Methoden eingesetzt werden. Bevor man sich letztlich auf ein System festlegt, sollte man noch überprüfen, ob es für bestimmte Aufgabenstellungen nicht

bereits Standardlösungen bzw. Muster gibt, die man übernehmen kann, damit man nicht alles neu erfinden muss.

In den folgenden Unterkapiteln werden die einzelnen Fragen aufgegriffen und es wird ein kurzer Überblick über vorhandene Grundlagen gegeben.

2.1 Das Benutzermodell

Möchte man eine Entwicklungsumgebung zur Verfügung stellen, die einerseits Programmieranfänger bei einführenden Problemen unterstützt und andererseits professionellen Programmierern alle Funktionen zur Verfügung stellt, um komplexe, mehrere Mannjahre dauernde Projekte zu realisieren, so ist das zum gegenwärtigen Zeitpunkt der Softwareentwicklung mit einer einzelnen Software nur sehr schwer möglich.

Der Profi verlangt eine sehr große Vielfalt an Werkzeugen, die den Anfänger bei der richtigen Auswahl überfordert, so dass dieser die für ihn benötigten Elemente nicht mehr finden kann. Durch geeignete Maßnahmen wie Assistenten oder voreinstellbare Konfigurationen kann die Anspruchsvielfalt in bestimmten Bereichen unterstützt werden. Trotzdem ist es notwendig, dass man sich Gedanken macht, wer aktuell die Entwicklungsumgebung nutzen soll und wie der Wissensstand des Benutzers ist.

Wird eine Umgebung für „Laien“ zur Verfügung gestellt, so sollte sie sehr stark gegen Fehlbedienungen gesichert werden. Zusätzlich müssen an vielen Stellen Hilfestellungen vorgesehen werden. Hingegen erwartet man von „Experten“, dass sie mit der Thematik vertraut sind und mehr Wert auf einen schnellen Erfolg bei der Bearbeitung des zu implementierenden Problems legen.

Viele Hilfestellungen und schnelle Bedienung sind zwei Ansprüche, die sich häufig widersprechen. Hilfestellungen sollten so angezeigt werden, dass sie ins Auge springen, um sie einfach in Anspruch nehmen zu können. Durch eine eventuelle Ablenkung hemmen sie aber den Arbeitsfluss, wenn sie nicht notwendig sind.

Gut strukturierte Menüs können Anfängern wichtige Funktionen zur Verfügung stellen, müssen aber durch den Medienwechsel von der Tastatur zur Maus und einige Klicks erkauft werden. Der Experte wünscht sich stattdessen lieber Tastenkombinationen, um direkt die gewünschte Funktion anzuwählen. Das Nebeneinander von Menüs und Tastenkombinationen ist ein Beispiel dafür, wie für unterschiedliche Benutzergruppen jeweils geeignete Unterstützungen angeboten werden können, ohne dass sie sich gegenseitig behindern.

Da der Experte sehr häufig mit „seiner“ Entwicklungsumgebung arbeiten wird, ist er bereit sich etwas länger einzuarbeiten, erwartet dafür aber später ein Programm, das ihn bei seinem Arbeitsfluss unterstützt und die Realisierung der Projekte beschleunigt. Der Anfänger dagegen setzt seine Entwicklungsumgebung selten ein. Er benötigt zu Beginn sehr viel Unterstützung, um grundlegende Aufgaben sehr leicht erledigen zu können. Komplexe Projekte müssen seltener realisiert werden, so dass hier der Schwerpunkt der Unterstützung darauf konzentriert werden muss, „nur“ eine korrekte Syntax zu unterstützen und lauffähige Programme zu erzeugen.

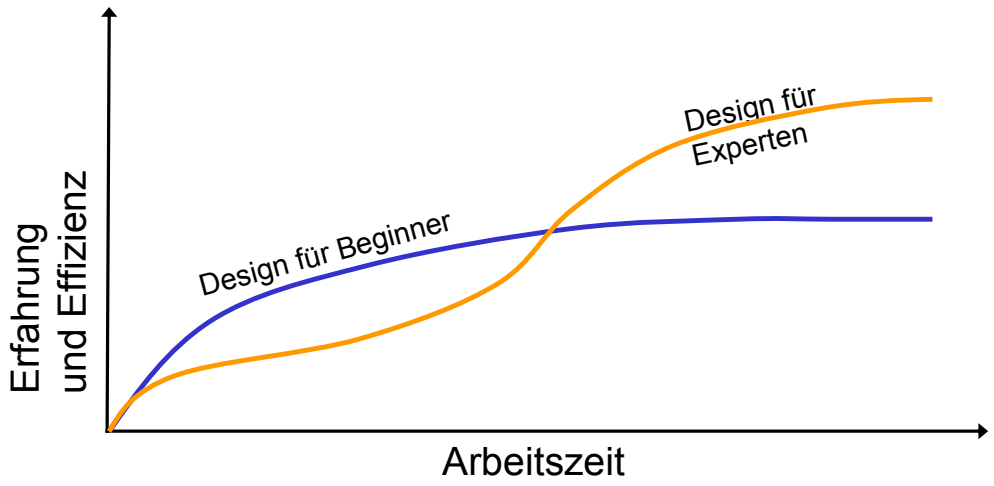


Abbildung 2-1: Lernkurven für hypothetische Systeme

In der Grafik nach Nielsen [Nie93] in Abbildung 2-1 werden zwei Lernkurven gezeigt, die unterschiedlich ausgelegte Entwicklungsumgebungen repräsentieren. Einmal kann ein Programm so gestaltet werden, dass die Benutzung schnell von Personen erlernt werden kann, eine komplexe Arbeit später aber viel Zeit kostet (dunkle Kurve). Somit ist das Programm gut für „Laien“ geeignet. Andererseits gibt es Programme für „Experten“, die zu Beginn schwerer zu verstehen sind, später die Entwicklung aber durch eine vielfältige Unterstützung beschleunigen und ein effizienteres Arbeiten erlauben (helle Kurve). So ist eine Entwicklungsumgebung wie Eclipse (siehe Kapitel 4.2.2) sehr mächtig, aber schon die Installation ist sehr aufwändig und das Schreiben eines kleinen Programms wie „Hello World“ ist für den Laien ohne das Studium von Hilfetexten schwer möglich. Dagegen können kleine Editoren wie Joe (siehe ebenfalls Kapitel 4.2.2) sehr schnell beherrscht werden. Das erste Programm ist mit Joe auch ohne vorherige Kenntnis der Umgebung in fünf Minuten geschrieben, aber bei Aufgaben wie dem Beschleunigen von Programmen erhält man keine Unterstützung mehr.

Um die zukünftigen Benutzer optimal unterstützen zu können, ist es wichtig sich bereits bei der Entwicklung Gedanken über deren Verhalten zu machen. Aus diesem Grund sollte der Entwickler einer Anwendung immer eine modellhafte Vorstellung der zu erwartenden Benutzergruppe zu Grunde legen. Diese Vorstellung wird in einem Benutzermodell zusammengefasst, in welchem folgende drei Kriterien enthalten sein müssen [SNI93, Kob85, Kob89]:

- Der allgemeine anwendungsbezogene Wissensstand des Benutzers
- Das Ziel des Benutzers, welches in einer jeweiligen Situation angestrebt wird
- Informationen, die der Benutzer benötigt, um die anstehenden Aufgaben ohne Probleme lösen zu können

In der Literatur wird unterschieden zwischen dem eigentlichen Wissen des Benutzers und dem, was er zu wissen glaubt [Kob85]. Dadurch kann noch besser auf die Anforderungen eingegangen werden. Allerdings wird diese Unterscheidung hier im Weiteren nicht verfolgt.

Damit ein Benutzermodell möglichst umfassend entwickelt werden kann, wird empfohlen in einem Entwicklungsprozess neben dem eigentlichen Entwickler auch Planer, Vertrieb, Kunde und Ergonom einzubeziehen. Dadurch wird ein Produkt unter verschiedenen Sichtweisen kritisch begutachtet und das spezielle Wissen jedes Einzelnen kann bei der Produktplanung einfließen. Allerdings müssen immer Kompromisse eingegangen werden, denn aus der Sicht des Einen kann eine Anforderung sehr wichtig sein, diese aber der Sicht eines Anderen völlig widersprechen. So wünscht sich zum Beispiel ein Ergonom eine aufwändige und intuitiv bedienbare Oberfläche, die somit zuerst teuer entwickelt werden muss, der Vertrieb möchte aber ein möglichst günstiges Produkt anbieten.

Da ein genaues Wissen über den jeweiligen Anwender nur in Ausnahmefällen bereits bei der Entwicklung eines Programms vorhanden ist, sollten Dialogsysteme benutzerorientiert gestaltet sein [Kob85]. Benutzerorientiert heißt hier, dass das System während des Dialogs selbstständig ein Modell des jeweiligen Benutzers aufbaut und entsprechend aktueller Beurteilungen die benötigten Informationen liefert.

In den meisten heutigen Programmen ist dieser Lernprozess nicht vorhanden. Allerdings kann der Benutzer die Reaktion bei einigen Anwendungen beeinflussen, indem er in den Einstellungen des Programms wählt, ob er im Experten-, Beginner- oder Normalmodus arbeiten möchte. Einen Weg, den Lernprozess zu berücksichtigen und ein benutzerangepasstes Verhalten zu erreichen, geht zum Beispiel die Firma Microsoft, indem lange nicht mehr benötigte Menüelemente nicht mehr angezeigt werden. Zusätzlich wird bei einer vermeintlichen Fehlbedienung durch so genannte Assistenten auf einfachere Lösungswege hingewiesen. Dass die individuelle Lernfähigkeit des Systems noch nicht ausgereift ist, kann man daran erkennen, dass viele Benutzer diesen „persönlichen Assistenten“ (Karl Klammer) nach einer gewissen Zeit genervt ausschalten, obwohl kaum jemand alle Optionen der Programme beherrscht. Aus diesem Grund werden die Assistenten in den neueren Programmversionen von Microsoft standardmäßig nicht mehr angeboten.

Für den weiteren Verlauf der Arbeit werden in Tabelle 2-1 beispielhaft Merkmale zusammengefasst, die den zukünftigen Benutzer einer Entwicklungsumgebung beschreiben sollen, der parallele Programme entwickeln möchte. Erarbeitet wurden die Merkmale anhand von Interviews mit Kollegen und Diplomanden. Diese Personengruppe hat sich als geeignet erwiesen, da teilweise jahrelange Erfahrung bei der Entwicklung paralleler Programme vorhanden war, anderseits immer wieder neue Personen mit der parallelen Programmierung vertraut gemacht werden mussten. Die erarbeiteten Merkmale können als Grundlage für ein späteres umfassenderes Benutzermodell dienen. Um die ergonomischen Besonderheiten der realisierten Entwicklungsumgebung aus Kapitel 6 beurteilen zu können, ist eine weitere Ausarbeitung eines Benutzermodells nicht notwendig.

Tabelle 2-1: Merkmale eines Benutzermodells angewandt auf Entwicklungsumgebungen für Parallelrechner

Allgemeiner Wissensstand
• Hat bereits viel Erfahrung bei der Programmierung linearer Programme gesammelt • Kennt die Möglichkeiten der Parallelrechner und deren Programmiersprachen • Benötigte Compiler, Debugger, Profiler und anderes werden individuell je nach Anforderung gewählt
Ziele in der jeweiligen Situation
• Einwicklung von komplexen Programmen ohne grafische Benutzungsoberfläche. Die zu implementierenden Algorithmen sind bekannt • Einfache, intuitive und sinnvolle Auswahl von Prozessoren • Möglichst effiziente Kommunikation zwischen den Prozessoren • Einfache Kommentierungsmöglichkeiten • Übersicht über den jeweils aktuellen Stand von Prozessorauswahl und Kommunikation
Fehlendes Situationswissen
• Wie werden die gewünschten Prozessoren ausgewählt? • Welche Arten der Kommunikation gibt es? • Welche Befehle werden zur Kommunikation mit dem Rechner benötigt? • …

2.2 Modelle für Benutzungsschnittstellen

Nicht nur die Benutzergruppe sollte modelliert werden, sondern auch die Schnittstellen, mit denen der Benutzer mit dem System kommunizieren kann. Dies ist nötig, um die komplexe Aufgabenstellung einer Programmerstellung zu strukturieren und in Einzelaufgaben zu zerlegen. Die Einzelaufgaben können anschließend leichter auf ihre Eignung überprüft und implementiert werden. In der Literatur sind verschiedene Modelle für Benutzungsschnittstellen zu finden, von denen die wichtigsten hier kurz vorgestellt werden sollen.

MVC-Modell

Das MVC-Modell (Model-View-Controller) oder im Deutschen teilweise auch Modell-Präsentations-Steuerungs-Modell (MPS) wurde 1978 von Trygve Reenskaug [Ree79] im Zusammenhang mit der objektorientierten Sprache Smalltalk bei der Firma Park Place entwickelt. Park Place bzw. die spätere Firma ObjectShare wurde 1999 von der Firma Cincom Systems aufgekauft (http://www.cincom.com).

Bei diesem ersten bekannten Modell in diesem Bereich sprach man noch nicht von einer Benutzungs-, sondern einer Benutzerschnittstelle [Lan94]. Heute verstehen Ergonomen unter einer Benutzerschnittstelle eher Hände und Augen, mit denen der Benutzer kommuniziert. Möchte man das Aussehen einer Programmoberfläche beschreiben, sprechen Ergonomen von der Benutzungsschnittstelle.

Ein Programm besteht nach dem MVC-Modell aus drei unterschiedlichen Arten von Modulen, den so genannten Models, Viewern und Controllern (siehe Abbildung 2-2). Dabei ist es die Hauptaufgabe des Modells, den Programmalgorithmus, die Schnittstellen zum Benutzer und die Darstellung auf den Ausgabemedien zu trennen.

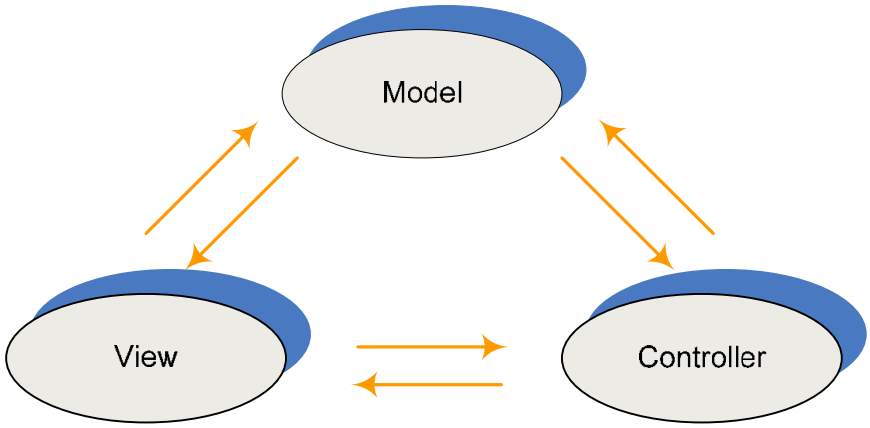


Abbildung 2-2: MVC-Modell (nach [Lan94])

Unter dem „Model“ versteht man den eigentlichen Algorithmus des Programms. Hier werden alle dauerhaften Daten und Algorithmen der Anwendung implementiert. Über geeignete Schnittstellen werden von und zu den anderen Modulen Werte entgegengenommen oder geliefert. Ob, wie und wie oft diese Werte in den anderen Modulen dargestellt werden, wird im „Model“ nicht festgelegt.

Mit einem „View“ wird dem Anwender eine Schnittstelle zur Verfügung gestellt, welche das Anwendungsobjekt repräsentiert. Hier wird das Layout der Programmoberfläche festgelegt und somit, wie Eingabemöglichkeiten oder Ergebnisse präsentiert werden sollen. Dabei ist es möglich, mehrere unterschiedliche Präsentationsschnittstellen zur Verfügung zu stellen. Wie weiter unten beschrieben wird, ist diese Vielfalt eine sinnvolle Voraussetzung, um aufgabenangemessene und individualisierbare Schnittstellen anzubieten (vgl. Kapitel 2.3).

Damit ein Anwendungsobjekt möglichst unabhängig entwickelt werden kann, sind noch Schnittstellen notwendig, welche die erforderlichen Daten liefern: Die Controller. Dabei fängt jeder Controller spezielle vom Benutzer oder sonstigen Umgebungseinflüssen kommende Ereignisse ab und leitet sie an die „Models“ und „Views“ weiter. Beispielsweise wird jeder Tastendruck vom Controller entgegengenommen und an die „Views“ gesendet, um visualisiert werden zu können und eventuell gleichzeitig an ein „Model“, um sofort auf die neue Umgebung mit der Änderung des Algorithmus reagieren zu können.

Ein Problem dieses Modells ist die notwendige Vielzahl an Controllern. In einer komplexen Anwendung gibt es mehrere hundert Controller, von denen viele gleichzeitig aktiviert werden könnten. Auf diese Vielfalt zu reagieren, erfordert eine sehr gute Planung, um die Übersicht über mögliche Aktionen zu behalten. Es hängt sehr stark von der eingesetzten Programmiersprache und Entwicklungsumgebung ab, wie gut und schnell auf Ereignisse reagiert werden kann. Nicht alle Sprachen stellen diesen Mechanismus zur Verfügung. Beispielsweise bieten die Standardversionen von C oder Pascal keine Möglichkeit der Ereigniskontrolle. Erst Erweiterungen durch einzelne Hersteller bieten mehr oder weniger komfortable Möglichkeiten der Ereigniskontrolle. Das in Kapitel 6.1 beschriebene Qt stellt beispielsweise eine Bibliothek von Benutzungsschnittstellen zur Verfügung, die innerhalb der Programmiersprache C++ genutzt werden können und eine MVC-konforme Programmierung ermöglichen.

IFIP-Modell

Während das MVC-Modell eine allgemeine Unterstützung bei der Implementierung eines Programms liefert, wird im IFIP-Modell (International Federation for Information Processing) ein Schwerpunkt auf die Kommunikation von Mensch und Maschine gelegt. Die komplette Benutzungsschnittstelle wird wie in Abbildung 2-3 dargestellt in drei Einzelschnittstellen aufgeteilt, die Ein-/Ausgabe-Schnittstelle, die Dialogschnittstelle und die Werkzeugschnittstelle [Lan94].

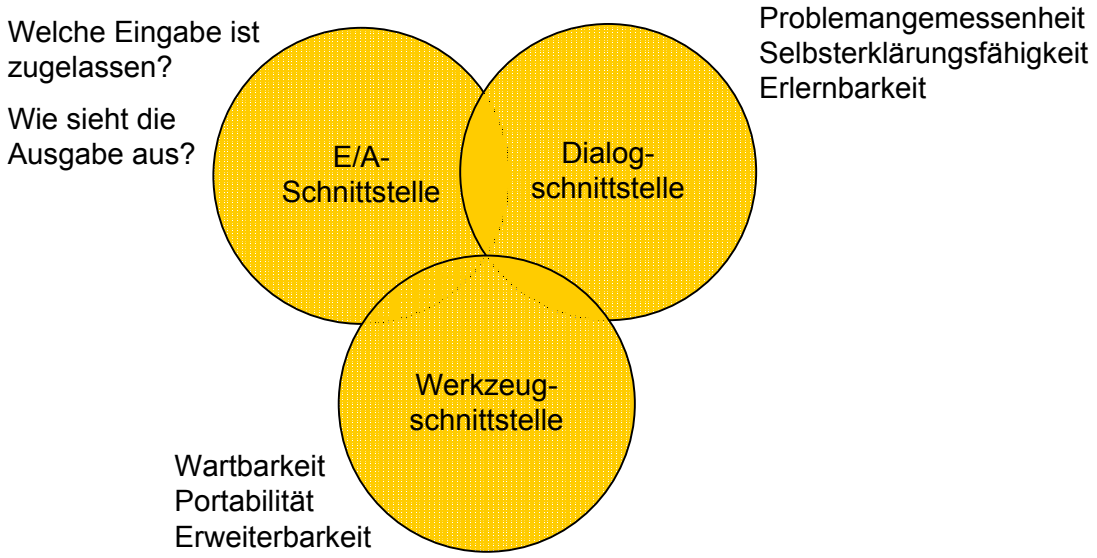


Abbildung 2-3: IFIP-Modell (nach [Lan94])

Über die Ein-/Ausgabeschnittstellen werden Regeln definiert, welche Eingabe ein System zulassen soll, bzw. wie ein Gerät seine Ausgaben zu präsentieren hat. In den Dialogschnittstellen findet man Regeln, wie eine Kommunikation aussehen soll. Diese Regeln sind die Grundlage für die in Kapitel 2.3 beschriebenen Normen. Die Werkzeugschnittstellen enthalten schließlich Regeln, die unter anderem über die Wiederverwendbarkeit, Zuverlässigkeit oder Erweiterbarkeit des Kommunikationssystems entscheiden.

Alle diese Regeln sollen dazu dienen, eine Kommunikation objektiv zu bewerten und somit vergleichbar zu machen [Her94]. Besonders die für die Dialogschnittstellen vorgestellten und in der DIN 66 234 Teil 8 (siehe Kapitel 2.3.1) festgehaltenen Regeln werden in der Arbeit öfter als Beurteilungskriterium zu Grunde gelegt.

Linguistisches Modell

Im linguistischen Modell wird eine Mensch-Maschine Kommunikation als Ebenenmodell aufgefasst (vgl. ISO/OSI-Schichtenmodell).

Unterschieden werden konzeptuelle, semantische, syntaktische und lexikalische Ebenen. Die Kommunikation vom Menschen zur Maschine nennt man Eingabestrom, umgekehrt heißt die Kommunikation von der Maschine zum Menschen Ausgabestrom (siehe Abbildung 2-4) [VoN98].

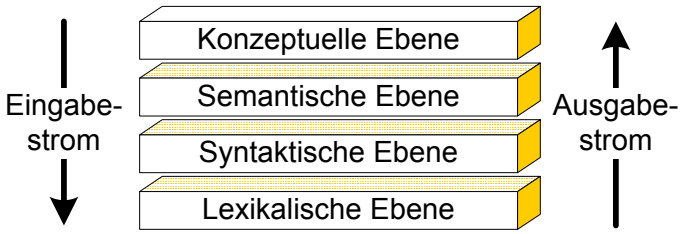


Abbildung 2-4: Linguistisches Modell

Auf konzeptueller Ebene werden allgemeine Objekte definiert und Beziehungen zwischen den Objekten aufgebaut.

Auf der semantischen Ebene werden die genauen Parameter festgelegt, die ein Objekt beschreiben, bzw. die zur Kommunikation zwischen den Objekten notwendig sind.

Auf syntaktischer Ebene wird das Layout der Benutzungsschnittstelle festgelegt. Dabei wird bestimmt, wo was angezeigt werden soll, bzw. in welcher Reihenfolge Informationen angeboten werden sollen.

Auf lexikalischer Ebene schließlich wird genau festgelegt, wie auf eine Dialogbox reagiert werden kann, ob man zum Beispiel Tatstatureingaben zulässt oder eine Mauseingabe unterstützt.

Die Überlegungen, die diesem Modell zu Grunde liegen, sollten bei jedem gut durchgeführten Programmierprojekt berücksichtigt werden. Bei der Realisierung wird das Linguistische Modell meist aber nicht die Hauptrolle spielen, sondern die einzelnen Aspekte spielen bei der Realisierung von Komponenten eine Rolle, die nach dem MVC-Modell konstruiert wurden.

Seeheim-Modell

1983 wurde das Seeheim-Modell auf einem Workshop für User Interface Management Systems (siehe Kapitel 2.5) in Seeheim entwickelt [Gre95]. Das Ziel des Modells ist es, eine klare Trennung zwischen der eigentlichen Anwendung und der Benutzungsschnitt-

stelle zu ziehen – genau wie das auch beim MVC-Modell der Fall ist. Innerhalb der Benutzungsschnittstelle wird beim Seeheim-Modell zwischen folgenden Komponenten unterschieden: Der Präsentationskomponente, der Dialogsteuerung und der Anwendungsschnittstelle, welche miteinander gekoppelt sind (siehe Abbildung 2-5) [VoN98].

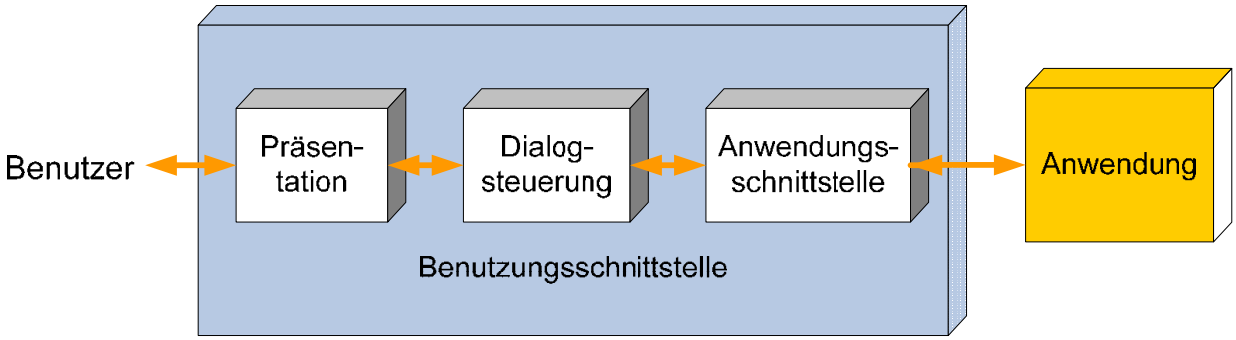


Abbildung 2-5: Seeheim-Modell (nach [VoN98])

Die Präsentationskomponente definiert die physische Darstellung (das Design) einer Anwendung, welche unter anderem folgende Fragen beantwortet: Wie wird eine Dialogbox dargestellt, wo sind Tastatureingaben möglich, wie wird an einer bestimmten Stelle auf eine Mauseingabe reagiert? Damit können relativ leicht für verschiedene nach dem Benutzermodell definierte Gruppen unterschiedliche Präsentationen zur Verfügung gestellt werden. Während der einen Gruppe mehr Hilfestellungen und evtl. eine Folge von Präsentationen geboten werden, kann eine zweite Gruppe innerhalb einer komplexeren Präsentation auf einmal wesentlich mehr Einstellungen vornehmen, ohne Platz durch nicht benötigte Hilfestellungen zu verlieren.

Durch die Abgrenzung der Präsentationsschicht können gezielt Präsentationen für unterschiedliche Geräte oder unterschiedliche Ansprüche zur Verfügung gestellt werden. Heute kennt man dieses Vorgehen beim Einsatz von unterschiedlichen Stylesheets für einen Webauftritt [ShH05] oder durch die Bereitstellung von unterschiedlichen Designs, auch „Themes“ genannt, bei vielen Programmen, beispielsweise dem Media Player von Microsoft [Mic06c]. Die unterschiedlichen Präsentationen müssen dabei alle die gleichen Parameter benutzen, die von der Dialogsteuerung ausgewertet und angepasst werden.

Die Anwendungsschnittstelle dient als Knotenpunkt zwischen der Dialogsteuerung (den gefilterten Ein- und Ausgaben) und den präsentationsunabhängigen Teilen des Programms.

Das Seeheim-Modell wird in vielen Publikationen zitiert [Kli96, Her94, Lan94, VN98]. Dabei wird es immer wieder mit den anderen Modellen verglichen. Je nach Gewichtung wird es als Schichtenmodell gesehen und neben das linguistische Modell gestellt, wobei beispielsweise die Präsentationskomponente und die lexikalischen Ebene gleich gesetzt werden. Herczeg sieht Ähnlichkeiten zum IFIP-Modell, wogegen das MVC-Modell oft als verfeinerbares Grundmodell des Seeheim-Modells gesehen wird.

Durch die Zusammenhänge der einzelnen Modelle sieht man, dass sich der Entwickler eines Programms zwar für ein Modell für Benutzungsschnittstellen entscheiden kann

und soll, die Überlegungen der anderen Modelle sollten allerdings immer an den geeigneten Stellen mit einfließen. Dieses Vorgehen erweist sich auch dadurch als sinnvoll, dass jedes Modell seinen Schwerpunkt auf unterschiedliche Aspekte des Entwurfs und Betriebs von Benutzungsschnittstellen legt. Somit kann ein Modell durch die Stärken der anderen erweitert werden.

Bei dem in Kapitel 6.3 vorgestellten Prototypen einer Entwicklungsumgebung wurde als Grundlage das MVC-Modell gewählt, da es zum einen bereits in der zu Grunde liegenden Bibliothek Qt genutzt wird, zum anderen als ausreichend angesehen wird, um den Quelltext mit einer Grundstruktur zu versehen. Zusätzlich werden in einigen Teilen, die eine besondere Benutzungsunterstützung erfordern, die Ideen der weiteren Modelle, besonders des IFIP-Modells, aufgegriffen.

2.3 Normen für die „Grundsätze der Dialoggestaltung“

Die vorgestellten Modelle zeigen Möglichkeiten, wie man Benutzungsschnittstellen sinnvoll aufbauen kann. Es wird aber nur unzureichend festgelegt, wie der Dialog selbst ablaufen soll, damit eine möglichst große Benutzungsfreundlichkeit erreicht wird. Unter benutzungsfreundlich versteht man dabei, dass der Anwender problemlos und effizient mit einem Programm umgehen kann [DIN98b].

Der Begriff der Benutzungsfreundlichkeit spielt besonders in Bezug auf die Interaktivität zwischen Benutzer und Computer eine Rolle, so dass hierfür zwei Normen aufgestellt wurden, die im Folgenden vorgestellt werden.

2.3.1 DIN 66 234 Teil 8

Die ergonomischen Gesichtspunkte der Dialoggestaltung wurden bereits frühzeitig in der Deutschen Industrienorm (DIN) 66 234 Teil 8 festgelegt. Die dabei beschriebenen Hauptkriterien sind Aufgabenangemessenheit, Selbstbeschreibungsfähigkeit, Steuerbarkeit, Erwartungskonformität und Fehlertoleranz.

Folgendermaßen werden die Begriffe in der Norm definiert [DIN98b]:

- Aufgabenangemessenheit: „Ein Dialog ist aufgabenangemessen, wenn er den Benutzer unterstützt, seine Arbeitsaufgabe effektiv und effizient zu erledigen.“
- Selbstbeschreibungsfähigkeit: „Ein Dialog ist selbstbeschreibungsfähig, wenn jeder einzelne Dialogschritt durch Rückmeldungen des Dialogsystems unmittelbar verständlich ist oder dem Benutzer auf Anfrage erklärt wird.“
- Steuerbarkeit: „Ein Dialog ist steuerbar, wenn der Benutzer in der Lage ist, den Dialogablauf zu starten sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist.“
- Erwartungskonformität: „Ein Dialog ist erwartungskonform, wenn er konsistent ist und den Merkmalen des Benutzers entspricht, zum Beispiel seinen Kenntnissen

aus dem Arbeitsgebiet, seiner Ausbildung und seiner Erfahrung sowie den allgemein anerkannten Konventionen.“

- Fehlertoleranz: „Ein Dialog ist fehlertolerant, wenn das beabsichtigte Arbeitsergebnis trotz erkennbar fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand seitens des Benutzers erreicht werden kann.“

Die ausführlichen Erklärungen und Beispiele der einzelnen Begriffe werden hier nicht wiedergegeben, können aber unter anderem in [DIN98b] nachgelesen werden. In den folgenden Kapiteln wird an vielen Stellen gezeigt, wie diese Anforderungen erfüllt werden können und wo durch das Zusammentreffen von unterschiedlichen Bedürfnissen Zielkonflikte entstehen (siehe unten).

2.3.2 EN ISO 9241-10

Die Norm DIN 66 234 Teil 8 wurde 1996 überarbeitet und zur europäischen Norm (EN) ISO 9241-10 portiert. Gleichzeitig wurde sie um die Punkte Individualisierbarkeit und Lernförderlichkeit erweitert [DIN98b]. Dabei werden Individualisierbarkeit und Lernförderlichkeit folgendermaßen definiert:

- Individualisierbarkeit: „Ein Dialog ist individualisierbar, wenn das Dialogsystem Anpassungen an die Erfordernisse der Arbeitsaufgabe sowie an die individuellen Fähigkeiten und Vorlieben des Benutzers zulässt.“
- Lernförderlichkeit: „Ein Dialog ist lernförderlich, wenn er den Benutzer beim Erlernen des Dialogsystems unterstützt und anleitet.“

Zusätzlich wird in der Norm verlangt, dass bei der Dialoggestaltung weitere Benutzermerkmale berücksichtigt werden. Dies sind zum Beispiel: Aufmerksamkeitsspanne, Grenzen des Kurzzeitgedächtnisses, Lerngewohnheiten und der Grad der Erfahrung im Umgang mit dem Dialogsystem.

Es ist leicht einzusehen, dass nicht immer alle Kriterien gleichzeitig erfüllt werden können. Beispielsweise verlangt die Aufgabenangemessenheit, dass man eine Aufgabe effizient lösen kann. Geht man davon aus, dass der Benutzer ein „Experte“ ist, kann er ein Programm am schnellsten durch eingeübte Tastenkombinationen bedienen. Das widerspricht allerdings teilweise der Selbstbeschreibungsfähigkeit, denn außer der direkten Ausführung des Befehls erhält man von dem Programm keine weiteren Erklärungen oder Rückmeldungen.

Beim Einsatz von Tastenkombinationen wird häufig ganz auf Dialogboxen verzichtet. Wenn man mit Dialogboxen arbeitet, kann es oft störend sein, ständig Bestätigungen beantworten zu müssen. Dies dient zwar der Selbstbeschreibungsfähigkeit, wirkt aber oft der Effizienz und somit der Aufgabenangemessenheit entgegen. Genauso wird mit Tastenkombinationen die Lernförderlichkeit nicht unterstützt, da man im Voraus keinen Hinweis erhält, welche Tastenkombination zum Ziel führen.

Somit kann es notwendig sein, die Vorteile des einen Grundsatzes gegenüber denen eines anderen abzuwägen. Dabei sind die Prioritäten fallweise festzulegen.

Shneiderman griff diese Normen auf und formte daraus „Acht goldene Regeln des Dialogdesigns“, welche als Richtlinie für die Dialoggestaltung genutzt werden können [Shn98]:

1. Versuche Konsistenz zu erreichen
2. Biete erfahrenen Benutzern Abkürzungen an
3. Biete informatives Feedback
4. Dialoge sollten abgeschlossen sein
5. Biete einfache Fehlerbehandlung
6. Biete einfache Rücksetzungsmöglichkeiten
7. Unterstütze benutzergesteuerten Dialog
8. Reduziere die Belastung des Kurzzeitgedächtnisses

Neben den Grundsätzen der Dialoggestaltung gibt es noch viele weitere Normen, die Richtlinien für ein problemloses Arbeiten am PC zur Verfügung stellen. Das fängt an bei der Farbdarstellung auf dem Bildschirm (EN ISO 9241-8), geht über das Aussehen einer Tastatur (DIN 9755) bis hin zur Höhe des Arbeitstisches (DIN 66 234-10) [DIN98a, DIN98b]. Da diese aber keinen direkten Einfluss auf die vorliegende Arbeit haben, werden sie hier nicht weiter aufgeführt.

2.4 Gestaltgesetze

Um die in den Normen beschriebenen Grundsätze der Dialoggestaltung einzuhalten, ist es sinnvoll einige gestalterische Regeln einzuhalten, welche die Psychologie des Menschen berücksichtigen. Dazu dienen die Gestaltgesetze von Max Wertheimer, die den Bezug und damit die unterschiedliche Bedeutung zwischen mehreren Elementen auf einem Bildschirm beschreiben:

1. Das Gesetz der Nähe
2. Das Gesetz der Ähnlichkeit
3. Das Gesetz der guten Gestalt
4. Das Gesetz der guten Fortsetzung
5. Das Gesetz der Geschlossenheit
6. Das Gesetz des gemeinsamen Schicksals

Für die genaue Beschreibung der Gesetze wird auf die Quellen verwiesen [Wer64].

Im Zusammenhang mit dieser Arbeit haben die Gesetze eine besondere Bedeutung beim Aufbau derjenigen grafischen Dialogboxen, die Kommunikationsmöglichkeiten und Aktivierung von Prozessorelementen bereitstellen sollen (vergleiche Kapitel 5.3.1, 5.3.2.).

2.5 Werkzeuge für die Programmentwicklung

Jedes Programm, das für einen „Normalanwender“ geschrieben wird, verfügt heute über eine grafische Fensteroberfläche. Programme, die von der Kommandozeile gestartet und mit vielen Parametern versehen sind, sind nur in Spezialfällen für kleinere Aufgaben und nur für eine kleine Anwendergruppe sinnvoll. Diese Kommandozeilenprogramme sollen in diesem Kapitel deshalb nicht weiter verfolgt werden.

Möchte man ein Programm mit grafischer Fensteroberfläche entwickeln, steht man vor der Frage, welche Hilfsmittel zur Verfügung stehen. Hersteller von Programmierumgebungen für Windowsprogramme erweitern die Grundsprachen fast immer um Bibliotheken, die beispielsweise die Entwicklung von Fenstern oder Dialogboxen erleichtern und so den direkten Einsatz der zugrunde liegenden „API-Funktionen“ (Application Programming Interface) unnötig machen. „API-Funktionen“ bieten Schnittstellen des Betriebssystems an, welche es ermöglichen, auf große Teile der Betriebssystemfunktionen zuzugreifen [Msdn06]. Allerdings sind sie meist nur mit sehr vielen Parametern und speziellen Variablentypen aufrufbar, so dass ihre Verwendung sehr fehleranfällig ist. Die Kapselung in spezialisierte Bibliotheken bietet einen wesentlich benutzungsfreundlicheren Zugriff. In den letzten Jahren wurde der Begriff der „API-Funktionen“ erweitert, so dass er nicht mehr nur beim Zugriff auf das Betriebssystem genutzt wird, sondern allgemeiner bei jedem Zugriff über eine definierte Funktionsbibliothek auf bestimmte Programme. Deshalb werden heute die Schnittstellen aller Programme im weiteren Sinne als „API“ bezeichnet.

Auf Linux/UNIX-Systemen benutzt man oft die originären Programmiersprachen C bzw. C++, die in ihrer Grundform keine Bibliotheken für grafische Programme enthalten. Alles selbst zu programmieren, ist bei den Anforderungen heutiger Programme nicht möglich, da alleine für die Programmierung typischer Fenstereigenschaften mehrere Mannjahre benötigt würden. Deshalb gibt es unabhängig von der Programmiersprache viele Bibliotheken, die die gewünschten Funktionen zur Verfügung stellen. Teilweise werden die Bibliotheken von freien Gruppen als so genannter „Open Source“ entwickelt, teilweise stehen auch genauso wie unter Windows von Unternehmen entwickelte kommerzielle Bibliothekssysteme, bzw. Entwicklungsumgebungen mit integrierten Spracherweiterungen zur Verfügung.

Brad Myers hat eine der ersten Einteilungen veröffentlicht, nach denen man Bibliotheken für Dialogsysteme unterscheiden kann [Mye95]. Kurze Zeit später hat sich eine darauf aufbauende Kategorisierung durchgesetzt, die unterscheidet, was und wie viele Komponenten in einer Bibliothek zur Verfügung gestellt werden:

- Graphical User Interface Systems (GUI-System)
- User Interface Toolkits (UI-Toolkit)
- Application Frameworks
- User Interface Builder (UI-Builder)
- User Interface Management Systems (UIMS)

- Model-Based- Interface Development Environments (MB-IDE)

Abbildung 2-6 verdeutlicht dabei, wie diese Entwicklungsumgebungen aufeinander aufbauen (nach [Hof98]):

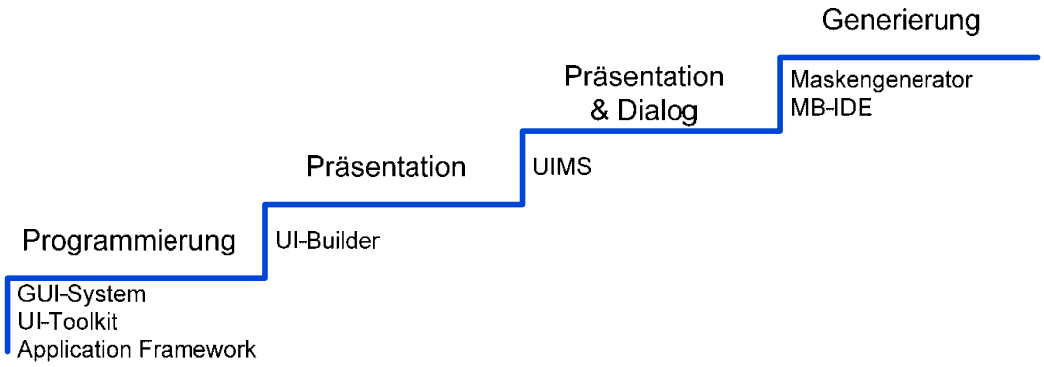


Abbildung 2-6: Entwicklungswerkzeuge

Um die Einteilung zu erklären und eine Auswahl des geeigneten Entwicklungswerkzeuges durchführen zu können, werden im Folgenden die einzelnen Stufen kurz vorgestellt.

Graphical User Interface System (GUI-System) / Windows System

Jedes moderne Betriebssystem hat eine grafische Oberfläche. Da heutige Betriebssysteme nicht mehr im Single-Task-Betrieb arbeiten, muss koordiniert werden, in welchem Programm ein gerade getätigter Tastendruck oder eine Mausbewegung eine Aktion auslösen soll. Das GUI- bzw. Windows-System stellt alle Funktionen zur Verfügung, um Fenster anzuzeigen, zu bewegen und zu manipulieren. Z.B. können Fenster verschoben oder vergrößert werden oder Linien, Punkte und Flächen können an beliebigen Positionen angezeigt werden. Beispiele für Windows-Systeme sind X-Windows oder MS Windows.

Alleine durch die Ausnutzung dieser Funktionen lässt sich jede Benutzungsoberfläche für Programme entwickeln. Allerdings sind Funktionen, die in das System integriert sind, meist nur mit sehr vielen Parametern aufzurufen und bieten nur die Grundfunktionalität. Wünschenswert ist dagegen, dass auch komplexe Funktionen zur Verfügung gestellt werden. Dazu werden oft UI-Toolkits benötigt.

User Interface Toolkit

Die Windows-Systeme bieten nur sehr elementare Schnittstellen. Um eine komfortablere Schnittstelle zu bieten, wird mit einem User Interface-Toolkit (UI-Toolkit) eine Bibliothek von wesentlich komplexeren Funktionen angeboten. So bieten die grundlegenden Windows-Systeme beispielsweise eine Klasse an, um ein leeres Programmfenster auf dem Bildschirm anzuzeigen. Zusätzlich kann man dieses Fenster bereits verschieben, seine Größe ändern, es aktivieren, deaktivieren und schließen. UI-Toolkits nehmen dieses Fenster als Grundlage und stellen darin zusätzlich Scrollbalken zur Verfügung, sobald diese benötigt werden. Daneben existieren bereits direkt unterhalb der Titelleiste Bereiche, welche das Menü und die Iconleiste aufnehmen können. Am unteren Rand

des Fensters ist eine Statuszeile vorgesehen. Somit wird dem Programmierer die Aufgabe abgenommen, diese unterschiedlichen Seitenbereiche zu erstellen und korrekt miteinander zu verknüpfen.

UI-Toolkits sind entweder als prozedurale oder objektorientierte Bibliotheken aufgebaut. Objektorientierte Toolkits können nur von objektorientierten Sprachen angesprochen werden. Prozedurale Toolkits dagegen können von allen Programmiersprachen aufgerufen werden. Um allerdings die Funktionalität objektorientierter Sprachen einsetzen zu können, sollten hier keine prozeduralen Toolkits benutzt werden.

Beispiele für UI-Toolkits sind Qt [Her01], gtk [Gri00], OSF/Motif [Pfo93], Fresco [Hir97], InterViews [LCV88, Pfo93], Khoros [Kho95] und ET++ [Gam92].

Application Framework

Application Frameworks bestehen im Normalfall aus einer kompletten Entwicklungsplattform. Dadurch werden dem Entwickler Editoren zur Verfügung gestellt, die vielfältige Hilfestellungen wie „automatisches Einrücken“, „Syntax-Highlighting“ oder „Autovervollständigung“ anbieten. Zusätzlich können eventuell Profiler angeboten werden, welche helfen, die Geschwindigkeit eines Programms zu optimieren oder ein Projektmanagement, das bei der Erstellung größerer Programme sehr hilfreich ist. Die einzelnen Hilfestellungen werden im Laufe der Arbeit noch genauer angesprochen.

UI-Toolkits sind genauso vorhanden wie weitere Bibliotheken, welche beispielsweise Methoden zur Verfügung stellen, um die Behandlung von Strings wesentlich zu vereinfachen. Auch Klassen, die dynamische Speicherstrukturen wie einfach und doppelt verkettete Listen, Stacks oder Queues zur Verfügung stellen, sind Beispiele für komplexere Aufgabenfelder, bei denen der Entwickler unterstützt wird.

Somit wird der Anwender nicht nur bei der Entwicklung der Programmoberfläche, sondern bei der Realisierung der kompletten Anwendung unterstützt.

Beispiele für Application Frameworks sind Mac-App oder Visual Works, außerdem bekannte Entwicklungsumgebungen wie Visual Studio, Eclipse, Delphi oder JBuilder.

User Interface Builder

Mit einem User Interface Builder (UI-Builder) können Oberflächen über einen grafischen Editor interaktiv zusammengestellt werden. Somit wird die aufwändige Programmierung einer Oberfläche weitestgehend ersetzt durch einfaches „Ziehen und Klicken“ mit der Maus. Für sehr viele der vorgestellten UI-Toolkits werden mittlerweile UI-Builder als Zusatztools angeboten. Auch Application Frameworks enthalten heute UI-Builder.

Für die Benutzungsoberfläche der zur erstellenden Anwendung wird durch die User Interface Builder der Quellcode automatisch erstellt. Somit steht die beim MVC-Modell „View“ genannte Benutzungsoberfläche automatisiert zur Verfügung (siehe Kapitel 2.2), der Entwickler muss den Quellcode meist nicht mehr ändern. Für die Interaktion mit dem Benutzer (den „Controllern“) werden häufig Dummy-Prozeduren bzw. -Methoden angelegt, die später an fest definierten Stellen von Hand angepasst werden müssen.

Diese Änderungen werden außerhalb der UI-Builder in normalen Quelltexteditoren vorgenommen.

User Interface Management Systems

Bei User Interface Management Systemen (UIMS) wird nicht nur die Oberfläche mit einem UI-Builder gestaltet, sondern die Dialogsteuerung kann zusätzlich mit einer Grammatik beschrieben werden. Dadurch können auch die Schnittstellen für die Programme ohne Programmierkenntnisse erstellt werden. Eine Möglichkeit, über ein UIMS-System eine Oberfläche zu gestalten, wird beispielsweise von IBM mit dem Abstract User Interface Markup Language (AUIML)-Toolkit angeboten, das in Eclipse eingebunden werden kann [IBM06].

Modellbasierte GUI-Entwicklungsumgebungen

Modellbasierte GUI-Entwicklungsumgebungen (Model-Based Interface Development Environments, MB–IDE) setzen sich aus verschiedenartigen Werkzeugen zusammen. Mit ihnen wird auf Modellen der Anwendung und der Anwendungsumgebung gearbeitet [Hof98]. Diese Werkzeuge befinden sich derzeit im Forschungsstadium, kommerzielle Produkte sind noch nicht in größeren Rahmen vorhanden. Denkbar ist es im Moment, dass Spezialprogramme auf diese Weise entwickelt werden können, zum Beispiel Ampelschaltungen an Kreuzungen, die anhand von Verkehrsflüssen automatisch generiert werden.

Auswahl der Bibliothek

Welche Werkzeuge man für die Entwicklung des eigenen Programms einsetzt, ist von vielen Faktoren abhängig. GUI-Systeme bieten alle Möglichkeiten, dabei muss der Entwickler aber sehr viel selbst programmieren. UI-Builder nehmen bereits sehr viel Programmierarbeit ab, allerdings muss man mit dem generierten Quellcode zufrieden sein, denn eine Änderung nach eigenen Bedürfnissen wird schnell so aufwändig, dass man auf den Einsatz der automatischen Quellcodegenerierung verzichten sollte. Application Frameworks bieten häufig viel Komfort, indem z.B. alle benötigten Tools auf Tastendruck erreichbar sind oder der Entwickler durch zusätzliche Informationen über z.B. Fehler, korrekte Syntax, vorhandene Funktionen/Methoden bei der Arbeit unterstützt wird und sich die Programmierung vereinfacht und somit beschleunigt.

Nicht jede Bibliothek unterstützt die gleichen Arbeitsschritte. Einige Bibliotheken sind spezialisiert z.B. auf den Aufbau von Dialogfenstern und unterstützen hier viele unterschiedliche Möglichkeiten. Andere gehen mit dem Angebot nicht so in die Tiefe, decken dafür aber eine breitere Basis ab. Teilweise werden auch unterschiedliche Grundfunktionen angeboten, durch die die Funktionsweise der Bibliothek geprägt wird. Ein Beispiel dafür ist das „signal-slot-Prinzip“ von Qt, welches in dieser Arbeit benötigt wird. Es wird in Kapitel 6.1 ausführlicher beschrieben.

Ein weiteres Unterscheidungskriterium für Bibliotheken ist deren zugrunde liegende Plattform. Es gibt viele Bibliotheken, die für Windows-, Mac- oder Linux-Systeme

angeboten werden. Einige unterstützen auch mehrere Plattformen, so dass man beispielsweise einen Quellcode erzeugen kann, der sowohl für Windows- als auch Linuxsysteme compiliert werden kann. In diesem Fall muss man allerdings darauf achten, dass man außerhalb der Bibliothekfunktionen keine plattformabhängigen Erweiterungen einsetzt.

Besonders bei Open Source Produkten muss man zusätzlich auf die Aktualität des Produktes und die Unterstützung durch eine große Entwicklergemeinde achten, damit im Problemfall jederzeit Unterstützung erhältlich ist und gefundene Fehler möglichst schnell beseitigt werden können. Im Laufe der Promotionsarbeit musste aus diesem Grund zweimal die Bibliothek gewechselt werden und viel Entwicklungsarbeit ging dadurch verloren.

Somit muss theoretisch vor jeder Entwicklungsarbeit entschieden werden, welche Bibliothek für das aktuelle Projekt am besten geeignet ist. Allerdings darf der Einarbeitungsaufwand nicht unterschätzt werden, so dass man sich häufig einmalig für ein System entscheidet und dieses dann immer einsetzt – mit allen daraus resultierenden Vor- und Nachteilen.

Leider haben Komfort und Vielfalt auch ihren Preis, so dass Applikation Frameworks schnell mehrere tausend Euro kosten. UI-Toolkits werden dagegen teilweise noch als Freeware angeboten.

Für die in Kapitel 6 beschriebene Implementierung einer Entwicklungsumgebung zur Unterstützung grafischer Programme wurde das UI-Toolkit Qt gewählt. Seit dem ersten Einsatz von Qt gab es bereits ca. ein Dutzend neue Versionen und viele weitere Updates, so dass der Quelltext immer wieder auf Kompatibilität überprüft und der Einsatz von neu zur Verfügung gestellten Funktionen getestet wurde. Seit einiger Zeit steht auch ein UI-Builder zur Verfügung.

2.6 Entwurfsmuster

Keine der vorgestellten Bibliotheken nimmt es dem Programmierer ab, ein Konzept für die Zusammenarbeit einzelner Komponenten zu entwickeln. Teilweise erhält man durch die in Kapitel 2.2 beschriebenen Modelle für Benutzungsschnittstellen eine Unterstützung. Die Modelle sind allerdings sehr allgemein gehalten und meist nur für Abgrenzung der Codezugehörigkeit und Schnittstellendefinition ausgelegt, so dass weitere, konkretere Überlegungen über den Programmaufbau folgen müssen.

Hier kann man „Entwurfsmuster“ einsetzen, die häufig auftretende Problemstellungen beschreiben und dafür Lösungsvorschläge bzw. Vorgehensweisen anbieten.

Die von Gamma et. al. vorgestellten Entwurfsmuster, allgemein „Design Pattern“ genannt, sind Vorlagen dafür, wie man bestimmte immer wiederkehrende Aufgaben lösen sollte. Dabei wird definiert, welche Klassen, Methoden und Objekte vorhanden sein sollen und welche Schnittstellen zur Interaktion zur Verfügung gestellt werden müssen [GHJV96]. Ein Entwurfsmuster wird dabei mit den vier Grundelementen Mustername, Problemabschnitt, Lösungsabschnitt und Konsequenzenabschnitt beschrieben.

Der Mustername ist als Stichwort gedacht, damit bei einer Kommunikation jedem bekannt ist, wovon man spricht. Wann das Muster angewandt werden soll, wird im Problemabschnitt beschrieben. Im Lösungsabschnitt wird eine Schablone (Template) vorgestellt, die Beziehungen, Zuständigkeiten und Interaktionen auf abstrakter Ebene darstellt. Der Konsequenzenabschnitt zählt die Vor- und Nachteile gegenüber anderen Entwurfsmustern auf. Da für eine Aufgabe häufig unterschiedliche Ansätze und somit auch unterschiedliche Entwurfsmuster gewählt werden können, soll diese Aufstellung helfen, bereits im Voraus das Richtige auszuwählen.

Für die eigene Entwicklung wurden nur einige der Entwurfsmuster benötigt, deren Aufgaben hier kurz dargestellt werden.

Singleton

Singleton ist ein Muster für eine Klasse, die in einem Projekt nur ein einziges Mal aktiv sein darf. Die Klasse soll selbst sicherstellen, dass keine zweite Instanz von ihr existiert. Benutzt wird die Klasse beispielsweise dafür, dass man nur eine Zwischenablage innerhalb des Programms benutzt.

Fliegengewichte

Ein Fliegengewichtsmuster hat die Aufgabe, innerhalb der eigenen Klasse möglichst wenige Daten speichern zu müssen. Der Grund liegt darin, dass die Klasse meist sehr häufig benötigt wird und die hohe Anzahl an umfangreichen Objekten schnell zu Performanceproblemen führen würde. Gelöst wird die Aufgabe, indem in der Klasse nur Zeiger auf Objekte gespeichert werden, die von vielen der Fliegengewichte gemeinsam genutzt werden können. So müssen die Daten nur einmal gespeichert werden, sind aber überall verfügbar. Eine klassische Aufgabe eines Fliegengewichtsmusters besteht in der Darstellung eines Buchstabens innerhalb eines Editors. Es wird nur das ASCII-Zeichen gespeichert. Schrift, Größe, Farbe, Ausprägung, … werden gesammelt in weiteren Klassen verwaltet.

Fassadenmuster

Das Fassadenmuster bildet eine einheitliche Schnittstelle, über die Methoden angesprochen werden können, die im Hintergrund auf mehrere Klassen verteilt sind. Der Grund ist, dass damit dem Anwender eine einheitlichere Schnittstelle angeboten werden kann, auch wenn bei der Realisierung viele verschiedene Komponenten benötigt werden.

Kompositionsmuster

Ein Kompositionsmuster besteht aus einer Reihe von abgeleiteten Klassen. Die höchste Klasse stellt dabei eine Grundfunktionalität zur Verfügung, welche durch die abgeleiteten Klassen jeweils verfeinert wird. Somit besteht das Konzept einer „Teil-Ganzes-Hierarchie“. Genutzt wird dieses Muster in der Arbeit für die Darstellung von Zeichen innerhalb des Editors, wobei einzelne abgeleitete Klassen beispielsweise Buchstaben, Zeilenumbrüche, Pixelgrafiken oder Vektorgrafiken darstellen können.

Neben den Entwurfsmustern, die von Gamma et. al. vorgestellt wurden, gibt es noch eine Reihe weiterer Entwurfsmuster [ShH03, Hor04, FFSB04]. Interessant sind dabei für Parallelrechner besondere „Design Pattern“ für parallele Programme [MSM04]. Da die Entwicklungsumgebung selbst allerdings ein lineares Programm ist, werden diese nicht direkt benötigt, können aber dazu dienen, sinnvolle Codefragmente beispielsweise über Skelettorientiertes Programmieren (siehe Kapitel 5.1.6) innerhalb einer Entwicklungsumgebung zur Verfügung zu stellen.

2.7 Barrierefreiheit

Zurzeit sind barrierefreie Webseiten ein großes Schlagwort [Hel05]. Seit Anfang 2006 sind alle öffentlichen Einrichtungen des Bundes verpflichtet, ihre Webseiten behindertengerecht bzw. barrierefrei darzustellen [BITV02]. Sinn dieser Verordnung ist, nicht nur Blinden oder körperbehinderten Personen den Zugang zu den Webseiten zu gewährleisten, sondern allen Personen eine einfachere, leichtere Sammlung von Informationen zu ermöglichen. Zudem gilt die Verordnung für alle grafischen Programmoberflächen, die öffentlich zugänglich sind.

Im Umfeld von ausführbaren Programmen wurde dieses Thema noch nicht eigenständig betrachtet, aber in den Ergonomierichtlinien, den einzelnen Styleguides der Hersteller und den beschriebenen Normen findet man Informationen zur Gestaltung von Benutzungsoberflächen, die indirekt einer behindertengerechten Gestaltung dienen.

Einige Eigenschaften, die für behindertengerechte Webseiten direkt angesprochen werden, sollten möglichst in allen Programmen benutzt werden. Hier eine kurze Aufzählung von ergonomisch wichtigen Erkenntnissen:

- Der Kontrast von Schrift und Hintergrund muss groß genug sein, damit jeder ermüdungsfrei arbeiten kann
- Schrift oder Abbildungen müssen in ihrer Größe an die Erfordernisse der Anwender anpassbar sein
- Notwendige Unterscheidungen von Elementen sollten möglichst in mehr als einer Dimension vorgenommen werden. Beispielweise können Schlüsselworte in der Farbe und Schriftdicke verändert werden.
- Die Navigation muss übersichtlich und leicht erreichbar sein

In den letzten Abschnitten wurden auf vielen unterschiedlichen Sichtweisen ergonomische Grundlagen für die Benutzung von Computerprogrammen vorgestellt. Alle sollten soweit möglich bei der Entwicklung von Programmen berücksichtigt werden. Auch wenn nicht immer alle Ergonomiegrundlagen eingehalten werden können, so muss bei jedem einzelnen Teilprogramm entschieden werden, welche Richtlinien jeweils am wichtigsten sind und welche eventuell vernachlässigt werden dürfen. Somit kann dies als multikriterielles Problem angesehen werden, zu dem es in den meisten Fällen keine eindeutige Lösung gibt. Nur wenn man eine Beziehung zwischen den Kriterien aufbauen und diese gegenseitig gewichten könnte, könnte das Problem auf ein Einkriterielles zurückgeführt

und eventuell eindeutig gelöst werden. Da das hier nicht der Fall ist, muss man bei der Entwicklung von Programmen immer mehrere Kriterien berücksichtigen und, soweit möglich, dem Endbenutzer die Möglichkeit bieten, dass dieser durch Parameter die für ihn am besten geeignete Variante selbst auswählen kann.

3 Massivparallele Rechner und Algorithmen

Bevor man Algorithmen auf massivparallelen Rechnern betrachtet, sollte erst definiert werden, was unter einem massivparallelen Rechner oder allgemeiner Parallelrechnern verstanden wird und welche unterschiedlichen Ausprägungen es gibt.

Unter massivparallelen Rechnern versteht man Computer, die aus sehr vielen Prozessoren bestehen. In der Regel sind dies einige Hundert bis mehrere 10.000 Prozessoren. Heute findet man meist Konstruktionen, in denen Standardprozessoren aus PCs mit einer geeigneten Kommunikationsstruktur vereint werden und dadurch Höchstleistungsrechner entstehen. Beispiele dafür sind aktuelle Entwicklungen von NEC oder IBM, die teilweise über 30.000 AMD- oder Intel-Prozessoren einsetzen. Die aktuell schnellsten und meist auch größten Parallelrechner werden auf der Webseite http://www.top500.org präsentiert und alle drei Monate aktualisiert.

Je nach Anzahl der Prozessoren spricht man von massivparallelen Rechnern oder „nur“ von Parallelrechnern, eine genaue Abgrenzung existiert allerdings nicht. Daneben gibt es weitere Kriterien, wie einzelne Parallelrechneranlagen unterschieden werden können:

- Anzahl der gleichzeitigen Befehlsströme (SIMD / MIMD)
- Ausstattung mit Hauptspeicher (Shared- / Distributed-Memory)
- Mächtigkeit einzelner Prozessoren
- Anordnung/Abhängigkeiten der Prozessoren (Vektorrechner, Cluster, Gitter)
- Größe des Rechners (PCI-Karte innerhalb eines PC / Platzbedarf einer Gebäude-Etage)

- …

Dabei kann eine Rechenanlage mehreren dieser Kategorien zugeordnet werden. Beispielsweise wird der in Kapitel 3.3 beschriebene MasPar-Rechner als massivparalleler Rechner, SIMD-Rechner und auch als Distributed-Memory Rechner beschrieben.

Die Mächtigkeit und Ausstattung eines einzelnen Rechenprozessors ist ein weiteres Kriterium, nach dem Parallelrechner unterschieden werden. Besitzt eine einzelne parallele Recheneinheit alle Merkmale, die für den Ablauf eines Programms notwendig sind, so wird die einzelne Recheneinheit im Allgemeinen Prozessor genannt. Fehlt dagegen ein Merkmal, spricht man von einem Prozessorelement (PE). Als mögliche Merkmale dienen beispielsweise Datenspeicher, Programmspeicher oder Registerspeicher. Diese Unterscheidung wird allerdings nicht von allen Parallelrechner-Herstellern unterstützt, so dass teilweise auch sehr kleine, zum Beispiel nicht mit eigenem Programmspeicher ausgestattete Prozessorelemente, Prozessor genannt werden. In dieser Arbeit wird jeweils die vom Hersteller vorgegebene Bezeichnung übernommen.

3.1 Speedup und Scaleup

Ziel des Einsatzes von massivparallelen Rechnern ist, die Abarbeitung eines sehr rechenintensiven Programms zu beschleunigen. Dazu muss das entsprechende Programm in geeigneter Weise in Teilprogramme zerlegt werden können. Jeder Prozessor sollte diese Teilprogramme möglichst mit lokal vorhandenen Informationen bearbeiten können, so dass das Gesamtergebnis nach dem Zusammenführen der Einzelergebnisse in wesentlich kürzerer Zeit vorliegt, als dies von einem einzelnen Prozessor berechnet werden könnte. Die erreichte Beschleunigung durch den Einsatz von mehr Prozessoren nennt man „Speedup“.

Idealerweise soll damit eine lineare Geschwindigkeitssteigerung erreicht werden, das heißt bei Verdoppelung der eingesetzten Prozessorzahl sollte das Ergebnis in der halben Zeit vorliegen.

In einigen wenigen Fällen ist sogar ein superlinearer Verlauf möglich, wobei dies nach Wiethoff auf „algorithmische Unzulänglichkeiten des seriellen Algorithmus“ hindeutet, „da in diesem Fall der serielle Algorithmus insgesamt mehr Teilprobleme untersucht als der parallele Algorithmus. Dies kann durch einen verbesserten seriellen Algorithmus vermieden werden.“ [Wie97].

In den allermeisten Fällen ist bereits eine lineare Geschwindigkeitssteigerung nicht möglich. Dies ist leicht damit zu erklären, dass der Kommunikationsaufwand mit der Anzahl der Prozessoren steigt. Zum einen müssen die Grunddaten verteilt und die Ergebnisse eingesammelt werden, zum anderen müssen die immer kleineren Teilprobleme häufiger auf Zwischenergebnisse anderer Prozessoren warten, um mit ihren eigenen Berechnungen fortfahren zu können. Bei bestimmten Problemfällen lässt sich sogar beobachten, dass durch die Erhöhung der Prozessorzahl die Erreichung der Gesamtlösung länger dauert, als beim Einsatz von weniger Prozessoren [Lin04].

Geht man nach den Leistungsbetrachtungen, die Amdahl bereits 1967 angestellt hat und die als „Amdahls Gesetz“ bekannt sind, so kann ein Speedup allein durch den Einsatz des notwendigen Datenaustauschs zu Beginn und am Ende eines Algorithmus drastisch eingeschränkt werden [Amd67].

Betrachtet man nach Amdahl ein festes Programm, das auf einem Rechner mit einem Prozessor eine Ausführungszeit T_1 benötigt, so benötigt es bei Einsatz von N Prozessoren ohne Berücksichtigung von Kommunikationskosten und voller Parallelisierbarkeit eine Ausführungszeit von

$$T_N = \frac{T_1}{N}$$

Berücksichtigt man, dass man einen Anteil von f für sequentielle, nicht parallelisierbare Programmteile benötigt, ändert sich die Formel auf

$$T_N = f * T_1 + (1-f) * \frac{T_1}{N} \qquad f \in [0,1]$$

Somit ergibt sich ein maximaler Speedup S_N von

$$\lim_{N\to\infty} S_N = \lim_{N\to\infty} \frac{T_1}{T_N} = \lim_{N\to\infty} \frac{T_1}{f * T_1 + (1-f) * \frac{T_1}{N}} = \frac{T_1}{f * T_1 + 0} = \frac{1}{f}$$

Das bedeutet, dass ein Programm, mit nur einem sequentiellen Anteil von 1% einen maximalen Speedup von 100 erreichen kann und somit kein Programm beliebig beschleunigt werden kann. Dadurch erscheint der Einsatz von Parallelrechnern mit mehreren 10.000 Prozessoren nicht sinnvoll. Allerdings wird bisher nicht berücksichtigt, dass auf größeren Rechnern im Normalfall größere Probleme bearbeitet werden. Bei diesen nimmt der relative Anteil an sequentiellem Programmcode ab, da die „Größe“ des Programms häufig dadurch bestimmt wird, dass mehr Stichproben gezogen werden oder mehr Referenzpunkte berücksichtigt werden müssen. Dieses mehr an Daten bezieht sich im Allgemeinen auf den zu parallelisierenden Teil des Gesamtprogramms. Somit wird neben dem Speedup der Scaleup interessant, der misst, wie schnell ein größeres Programm im Vergleich zu einem kleineren abgearbeitet wird.

Weiter muss berücksichtigt werden, welcher Programmteil als paralleler Programmteil angesehen wird. Hier muss stark nach den Rechnertypen unterschieden werden. Während ein SIMD-Rechner mit tausenden von Prozessorelementen darauf ausgelegt ist, dass immer einige deaktiviert werden, so ist es bei Clustersystemen sehr störend, wenn alle Prozessoren auf das Ergebnis eines einzelnen warten müssen. Nach Bräunl kann auf einem SIMD-Rechner auch mit einem „sequentiellen Programmteil“ von 20% noch ein guter Scaleup erreicht werden, dagegen können Message-Passing Systeme bereits bei 5% sequentiellem Programmanteil stark beeinträchtigt werden [Bra93].

3.2 Modell eines massivparallelen Algorithmus

Es gibt einige Modelle für parallele Rechner, die als Grundlage für die Entwicklung paralleler Programme dienen. Die bekanntesten sind dabei PRAM [FoW78], BSP [Val90] und LogP [CKPS+93].

PRAM (Parallel Random Access Machine) ist ein Modell, das davon ausgeht, dass nebeneinander viele Prozessoren zur Verfügung stehen, welche gleichzeitig ein Problem bearbeiten und den gleichen Speicher ansprechen können. Je nach Untertyp unterscheidet man, ob ein gleichzeitiger Lese- und/oder Schreibzugriff auf den Speicher möglich ist. Da bei dem Modell die Kommunikationszeit zwischen den Prozessoren vernachlässigt wird, ist es für die weiteren Überlegungen in dieser Arbeit nicht geeignet.

Das BSP-Modell (Bulk Synchronous Parallel Computers) verfügt über einen Parameter, um die Kommunikationszeit zwischen Prozessoren zu beschreiben. Dabei wird ein Synchronisationsmechanismus vorausgesetzt, welcher benötigt wird, um in bestimmten Zeitabständen, den sogenannten „supersteps“, zu überprüfen, ob die vorgegebenen Arbeiten erledigt wurden. Diese Synchronisation wird für die grundlegenden Überlegungen in diesem Kapitel nicht benötigt, so dass sie eine unnötige Einschränkung darstellt. Aus diesem Grund kann das BSP-Modell nicht als Grundlage der weiteren Überlegungen dienen.

Das LogP-Modell benutzt folgende Parameter, um einen Algorithmus zu beschreiben: Die Anzahl der Prozessoren (P, processor), die Kommunikationsbandbreite (1/g, gap), die Kommunikationsverzögerung (L, latency) und den Kommunikationsoverhead (o, overhead). Die Parameter sind notwendig, um die Laufzeit von Algorithmen berechnen zu können. Da im Folgenden keine Berechnungen auf Grundlage des Modells durchgeführt werden sollen, wird diese Einteilung nicht benötigt.

Bei den bisherigen Modellen steht die Ermittlung der Komplexität und Laufzeit von Algorithmen im Vordergrund. In dieser Arbeit wird die Abbildung des grundlegenden Aufbaus von Programmen für massivparallele Rechner benötigt, so dass diese aufwändigen Modelle nicht benötigt werden. Vielmehr genügt ein Modell, welches eine Beziehung zwischen wenigen definierten Begriffen aufbaut. Deshalb soll im Weiteren ein Modell vorgestellt werden, welches sehr allgemein den Ablauf paralleler Algorithmen beschreiben kann und welches durch weitere Parametrisierung in jeweils die oben genannten übergeführt werden kann.

Die Aufgabe der massivparallelen Rechner besteht in der Regel darin ein Problem zu lösen, für das sehr viel Rechenzeit benötigt wird. Meist müssen zu Beginn Grunddaten übergeben werden und evtl. Parameter, wie diese Daten zu verwerten sind. Anschließend benötigt der Rechner einige Sekunden bis mehrere Wochen, um die Problemstellung zu lösen. In dieser Zeit werden keine oder nur wenige Aktionen vom Benutzer erwartet. Am Ende werden die Daten zum Beispiel in Form einer Ergebnisdatei wieder zurückgegeben.

Im Folgenden soll nun ein Modell vorgestellt werden, welches die Begriffe „Aktivierung“, Kommunikation“ und „lokale Berechnung“ in den Mittelpunkt stellt und bei dem zusätz-

lich von einer „Initialisierung“ und einem „Rücktransfer“ gesprochen wird. Dabei wird gezeigt, wie diese Begriffe auf unterschiedliche massivparallele Rechnertypen angewandt werden können.

Aktivierung

Bevor auf einem „Parallelrechner“ ein Programm ausgeführt werden kann, muss zuerst bestimmt werden, welcher Prozessor/Prozessorelement die jeweilige Aufgabe übernehmen soll. Dieser Schritt wird im Modell „Aktivierung“ genannt.

Dies führt dazu, dass der aktivierte Prozessor entweder ein komplettes Programm zur Abarbeitung übertragen bekommt oder auch nur einen einzelnen Befehl, den ein einzelner Prozessor oder eine bestimmte Gruppe von Prozessoren ausführen soll. Bei weiteren Rechnertypen kann die Aktivierungsphase auch zur „Rekonfiguration“ genutzt werden. Das bedeutet, dass die Kommunikationsstruktur an aktuelle Gegebenheiten angepasst werden sollen.

Je nach Rechnertyp können durch die Aktivierung einzelne Prozessoren gruppiert werden und damit jeweils zeitgleich unterschiedliche Aufgaben übernehmen. Bei anderen Rechnertypen hat man lediglich die Möglichkeit zu entscheiden, ob ein Prozessor aktiviert oder deaktiviert werden soll. Diese Deaktivierung ist sinnvoll, denn dadurch können beispielsweise in den deaktivierten Prozessoren ältere Daten erhalten bleiben, die durch neue Berechnungen wieder verloren gehen würden. Weiterhin wird die Aktivierung/Deaktivierung eingesetzt, um bei einer Wenn-dann-Bedingung alle Zweige abzuarbeiten. Je nach dem, auf welchem Prozessor die Bedingung erfüllt ist, wird dort eine andere Berechnung durchgeführt. Ein weiteres Beispiel ist eine unterschiedliche Schleifenlänge auf verschiedenen Prozessoren.

Da die Ursache für eine Aktivierung auf einem Parallelrechner von Prozessor zu Prozessor unterschiedlich sein kann, ist diese nicht immer in einem Programmschritt durchführbar. Somit kann es vorkommen, dass der Aktivierungsschritt mehrfach ausgeführt werden muss, um unterschiedliche Kriterien für die Aktivierung berücksichtigen zu können.

Kommunikation

Nachdem in der Aktivierungsphase die benötigten Prozessoren bestimmt wurden, fällt der Schritt der „Kommunikation“ an.

Hier führen die vorher aktivierten Prozessoren einen Datenaustausch mit anderen, meist „benachbarten“ Prozessoren aus. „Benachbart“ bedeutet dabei, dass zwischen den Prozessoren eine Verbindungsstruktur besteht, die im Allgemeinen einen schnellen Datentransfer erlaubt. Dabei können meist zeitgleich viele Prozessoren mit jeweils ihren „Nachbarn“ eine schnelle Kommunikation durchführen, ohne dass dadurch die einzelne Übertragung verlangsamt wird.

Die Nachbarschaft hängt von der Hardware des jeweiligen Rechners ab. Es können beispielsweise Gitter- oder Torus-Strukturen aufgebaut werden, bei denen ein Prozessor

vier bis acht direkte Nachbarn hat. Andere Rechner können ihre Nachbarschaft zur Laufzeit ändern (siehe Kapitel 3.5) oder die Nachbarschaft kann durch geeignete Maßnahmen stark erweitert werden, indem eine Nachricht auf einen Bus gelegt wird und dort von mehreren Prozessoren „zeitgleich“ abgefragt werden kann (siehe Kapitel 3.3 und 3.5). In Rechenclustern werden normale Netzwerke als Kommunikationsmedium genutzt, während andere Rechnertypen spezielle schnelle Möglichkeiten zur Kommunikation einsetzen.

Bei der Kommunikation ist es bei einigen Rechnertypen nicht erforderlich, dass alle beteiligten Prozessoren vorher als aktiv markiert wurden. In diesem Fall kann ein aktiver Prozessor auch Daten von inaktiven Prozessoren abholen oder dorthin versenden. Ein inaktiver Prozessor kann aber keine Aufgabe anstoßen.

Da nicht immer alle Daten auf einmal verteilt werden können, muss auch der Schritt der Kommunikation eventuell mehrfach ausgeführt werden, bis alle benötigten Daten zu den gewünschten Prozessoren verteilt werden können. Dies kann daran liegen, dass benötigte Verbindungen von anderen Prozessoren belegt werden, ein Prozessor nicht zeitgleich in unterschiedliche Richtungen kommunizieren kann oder das Datenvolumen zu groß für eine einzige Übertragung ist.

Um die Verbindung in unterschiedliche Richtungen zu ermöglichen, kann es zusätzlich notwendig werden, dass zwischen zwei Kommunikationsschritten ein oder mehrere Schritte der Aktivierung notwendig werden.

Die hier vorgestellt Weise der Kommunikation wird auf SIMD-Systemen nur für den Austausch von Daten genutzt (SIMD-Systeme werden im nächsten Kapitel genauer beschrieben). Die Verteilung der auszuführenden Programme ist meist fest definiert. Entweder werden zu Beginn die Programme über feste Verbindungen an alle Prozessoren verteilt oder es erfolgt eine schrittweise Auslieferung der Befehle, die aber ebenfalls immer auf die gleiche Weise abläuft. Bei MIMD-Systemen können dagegen während der Laufzeit neben den Daten auch größere Programmteile verteilt werden, die von Prozessor zu Prozessor variieren können.

Lokale Berechnung

Bis zu diesem Zeitpunkt wurde auf den einzelnen Prozessoren noch keine Berechnung durchgeführt. Dies wird im Schritt der „lokalen Berechnung“ nachgeholt.

Durch die Kommunikation besitzen alle Prozessoren die benötigten Informationen. Die lokalen Berechnungen können je nach Rechnertyp aus einem einzelnen Schritt bestehen oder aus komplexen Algorithmen, die stundenlange Berechnungen erfordern.

Nachdem ein Ergebnis ermittelt wurde, werden entweder wieder Daten ausgetauscht oder neue aktive Prozessoren ausgewählt, welche die nächsten Aufgaben übernehmen. Durch diesen Mechanismus entsteht ein ständiger Wechsel zwischen Aktivierung, Kommunikation und lokaler Berechnung.

Initialisierung und Rücktransfer

Am Ende aller Berechnungen fallen Daten an, die entweder an ein weiteres Programm übergeben werden können oder einem Benutzer zur Verfügung gestellt werden sollen. Ein massivparalleler Rechner ist normalerweise nicht für interaktive Ein- und Ausgaben ausgelegt. Deshalb werden diese Aufgaben in der Regel von einem dafür vorgesehenen Frontend-Rechner bzw. Client vorgenommen.

Schließlich steht am Ende aller Berechnungen noch der Transfer der Daten zum Frontend-Rechner an.

Damit der Prozess beginnen kann, muss das Modell noch um einen Initialisierungsschritt erweitert werden, der zu Beginn Daten und Programme auf den massivparallelen Rechner überträgt. Jeder Algorithmus auf einem massivparallelen Rechner besteht also aus den Schritten Initialisierung, Aktivierung, Kommunikation, lokale Berechnung und Rücktransfer, wobei jeder einzelne Schritt null, ein- oder mehrmals ausgeführt werden kann. Zusätzlich besteht eine Schleife, die immer zwischen Aktivierung, Kommunikation und lokaler Berechnung wechselt.

Der beschriebene Programmablauf wird in der unten stehenden Abbildung 3-1 noch einmal grafisch dargestellt.

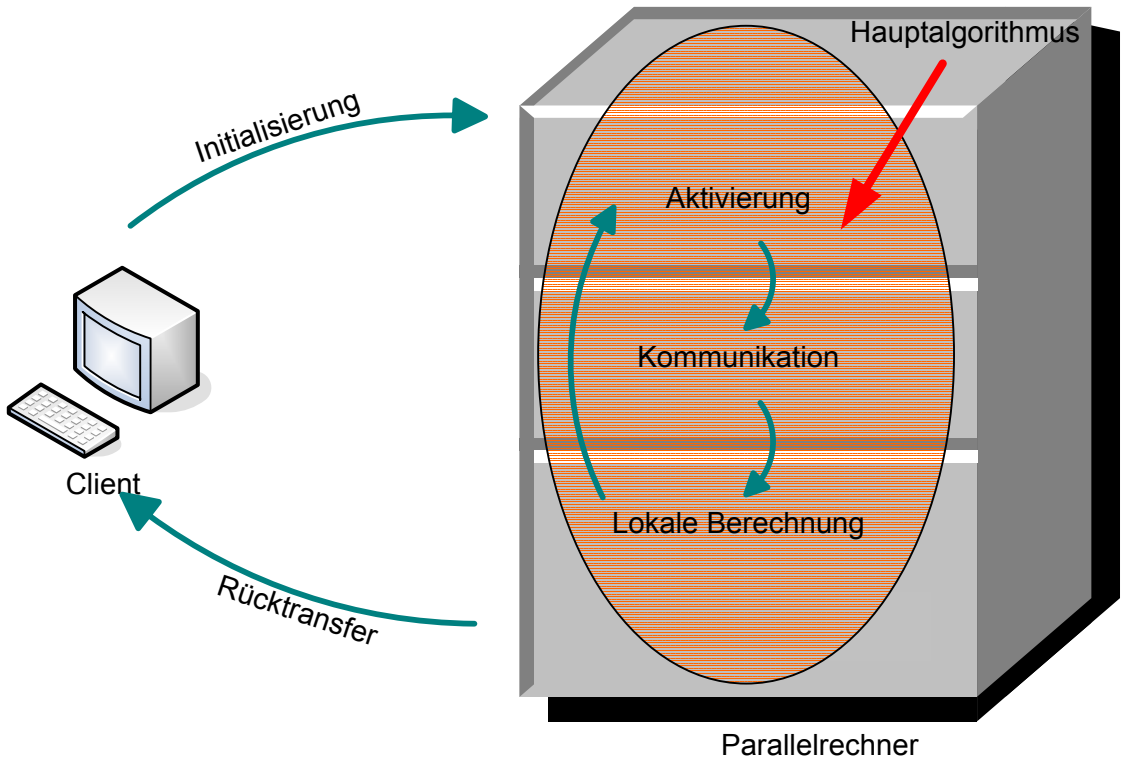


Abbildung 3-1: Modell eines massivparallelen Algorithmus

Das vorgestellte Modell soll nun anhand beispielhafter Algorithmen auf unterschiedlichen parallelen Systemen angewandt werden.

3.3 MasPar

Der MasPar-Rechner ist ein Rechner, der unter anderem am Rechenzentrum der Universität Karlsruhe (TH) eingesetzt wurde. Er besteht aus 16.384 Prozessorelementen, die in einer Matrix von 128 auf 128 angeordnet sind. Eine schnelle Kommunikation kann zwischen benachbarten Prozessorelementen stattfinden. Die Nachbarschaft ist beim MasPar-Rechner etwas weiter definiert, als bei vielen anderen Parallelrechnern. Alle Prozessorelemente, die in einer der acht Richtungen N, NW, W, SW, S, SO, O oder NO liegen, gelten als Nachbarn. Wie weit ein Prozessorelement in einer dieser Richtungen entfernt liegt, spielt (meist) keine Rolle. Diese Verbindungsstruktur wird auf der MasPar XNet genannt [Mas92a, Mas92b]. In Abbildung 3-2 sind die dunkelblauen Prozessorelemente Nachbarn des mittleren rot dargestellten Prozessorelementes.

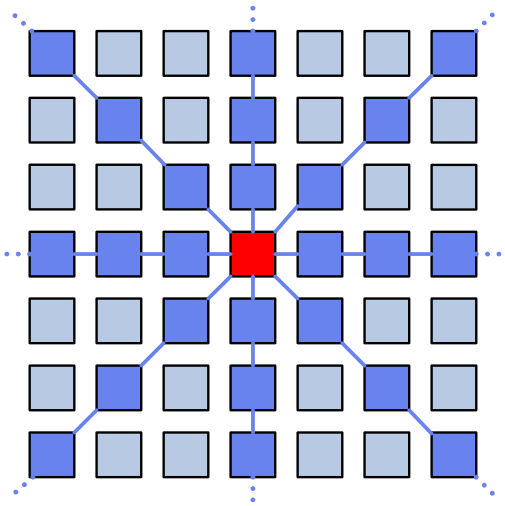


Abbildung 3-2: MasPar Nachbarschaft auf dem XNet

Über die Grenzen des Gitters bleibt die Kommunikationsstruktur erhalten, so dass als zugrunde liegende Struktur ein Torus vorliegt (siehe auch Abbildung 3-11).

Betrachtet man die Architektur der MasPar etwas genauer, so stellt man allerdings fest, dass die Prozessorelemente in 8x8-Blöcken aufgebaut sind. Überschreitet man diese Grenzen, so muss ein „Zwischentakt“ eingelegt werden und die Übertragung wird etwas langsamer. Die Dokumentation der Firma MasPar beschreibt diese Besonderheit nicht, allerdings konnten im Rahmen einer Dissertation [Koh99] bei Nichtbeachtung dieser Konstruktion geringe Geschwindigkeitseinbußen festgestellt werden. Für den grundsätzlichen Einsatz des XNet hat dieser Geschwindigkeitsunterschied aber keinen Einfluss.

Zusätzlich steht noch eine globale Kommunikation zur Verfügung, bei der jedes Prozessorelement mit jedem anderen kommunizieren kann. Allerdings ist diese Kommunikation bis zu einem Faktor 1000 langsamer als der Einsatz des XNet und sollte soweit möglich vermieden werden, um die Entwicklung effizienter Programme zu erreichen [AISW96].

Zur Befehlsverteilung benutzt die MasPar eine SIMD-Architektur. Das heißt, auf allen Prozessorelementen wird zu einem Zeitpunkt der gleiche Befehl bearbeitet, lediglich die Daten unterscheiden sich. Beispielsweise wird auf allen Prozessorelementen die

Berechnung a=2*b durchgeführt, der Wert für b (und somit nach der Berechnung auch für a) kann sich aber auf jedem Prozessorelement unterscheiden.

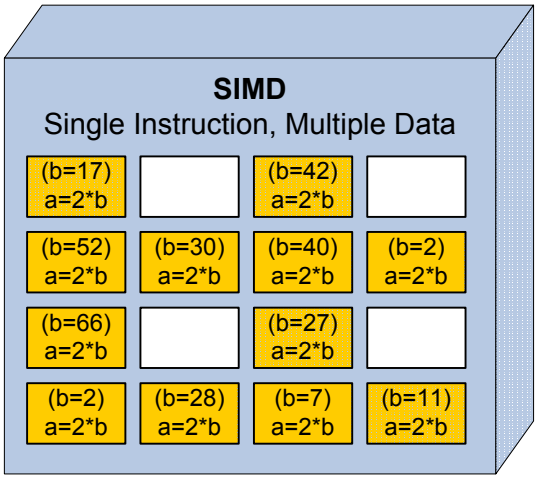


Abbildung 3-3: SIMD-Berechnung

Neben dem MasPar-Rechner setzen unter anderem folgende Großrechner die SIMD-Architektur ein: Die Connection Machine von Thinking Machines Corporation mit bis zu 65.536 Prozessorelementen und der Distributed Array Processor von Active Memory Technology mit 4.096 Prozessorelementen. Neue Entwicklungen sind wabenförmig aufgebaute, Honeycomb genannte, Architekturen. Auf diese Entwicklung wird in Kapitel 7.3 kurz eingegangen.

Die SIMD-Rechnerarchitektur wird heute neben den klassischen Großrechnern hauptsächlich bei spezialisierten Prozessoren bzw. Prozessorteilen eingesetzt, da der Aufwand, einen effizienten Algorithmus zu schreiben, indem die besonderen Vorteile der Architektur genutzt werden, relativ groß ist. Ein Beispiel für einen spezialisierten Prozessortyp, der die SIMD-Architektur ausnutzt, ist der Grafikprozessor. Grafikchips setzen intern mehrere parallele Pipelinearchitekturen ein, auf denen überall die gleichen Berechnungen für unterschiedliche Pixel durchgeführt werden müssen [LHKP+04]. Da nur eine einheitliche Befehlsfolge für alle Berechnungen bereitgehalten werden muss, können auf den Chips viele Schaltelemente eingespart und somit Kosten gesenkt werden. Da die spezialisierten Prozessoren der Grafikchips teilweise schneller sind als die eigentlichen CPUs der Rechner und zusätzlich über einen schnelleren Zugriff auf den Speicher verfügen, wird bereits versucht, aus Grafikkarten Parallelrechner zu bilden [FQKY04].

Im Folgenden wird ein Sortieralgorithmus („Odd-Even-Transposition-Sort“) auf einem SIMD-Rechner vorgestellt, der das typische Vorgehen bei der Programmierung auf einem SIMD-Rechner verdeutlicht und einzelne Schritte innerhalb des Modells für massivparallele Algorithmen aufzeigt [Tvr99]. Als Kommunikationsstruktur wird das XNet zugrunde gelegt.

Die Idee des Algorithmus ist, dass zwei nebeneinander liegende Prozessorelemente ihre Daten vergleichen und gegebenenfalls austauschen, so wie das auf sequentiellen Rechnern mit einem Bubble-Sort-Algorithmus durchgeführt wird. Wird dieser Prozess

mehrmals nacheinander durchgeführt, sind am Ende alle Elemente sortiert. Abbildung 3-4 zeigt den Algorithmus anhand der Sortierung von n unterschiedlichen Grauwerten, wobei maximal n-1 Schritte notwendig sind. Die Korrektheit der Sortierung kann mit Hilfe des 0-1-Lemmas bewiesen werden.

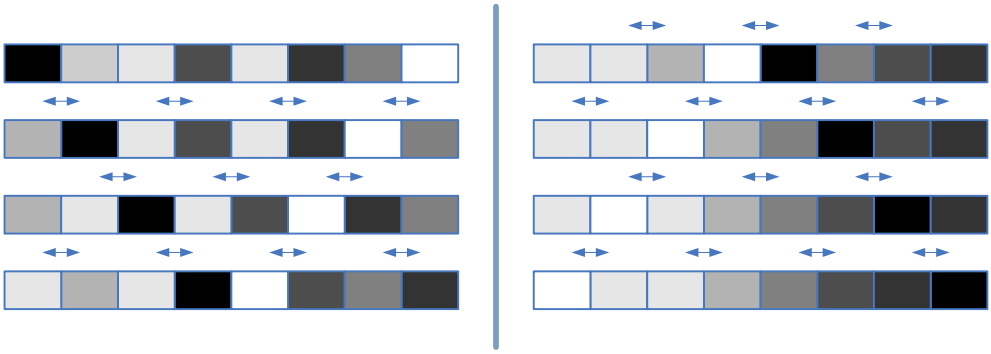


Abbildung 3-4: Odd-Even-Sort am Beispiel einer Graustufensortierung

Betrachtet man das Modell für massivparallele Rechner, muss zuerst eine Aktivierung von Prozessorelementen stattfinden. Dies ist notwendig, um zu klären, wer mit seinem linken bzw. rechten Nachbarn kommunizieren wird. Eine Lösung ist, dass jedes zweite Prozessorelement aktiviert wird. Im Schritt der Kommunikation müssen die aktiven Elemente die Werte ihrer rechten Nachbarn einholen. Anschließend kann in einer lokalen Berechnung entschieden werden, welches der beiden Elemente kleiner ist, um gegebenenfalls anschließend wieder eine Kommunikation aufzubauen und das größere Element dem rechten Nachbarn zurückzugeben. Anschließend werden die aktiven Prozessorelemente deaktiviert und die vorher deaktivierten Elemente aktiviert. Nun tauschen diese ihre Werte mit dem rechten Nachbarn aus. Somit kann sichergestellt werden, dass innerhalb von n-1 Schritten n Elemente sortiert werden können.

Abbildung 3-5 zeigt die abwechselnde Aktivierung mehrerer Spalten für die schnelle Zeilensortierung.

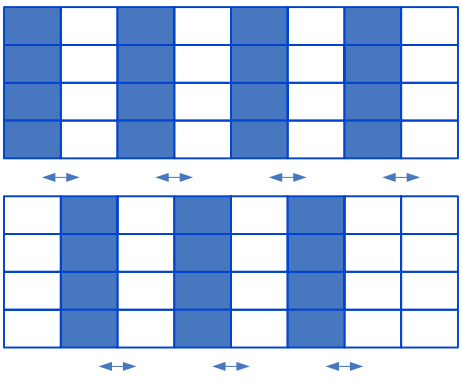


Abbildung 3-5: Wechselnde Aktivierung beim Odd-Even-Sort in mehreren Zeilen

In dem Beispiel wurde lediglich eine Nachbarschaft zum rechten Nachbarn ausgenutzt, so dass dieser Algorithmus auch auf allen SIMD-Rechnern mit einfacheren Nachbarschaften ohne Vorhandensein eines XNet realisiert werden kann.

Es wird deutlich, dass auf der MasPar sehr häufig zwischen Aktivierung, Kommunikation und lokaler Berechnung gewechselt wird. Durch die schnelle Kommunikationsstruktur und die dazu im Vergleich relativ langsamen und einfachen Prozessorelemente ist dieser Aufbau eines Algorithmus gerechtfertigt. Würde man zum Beispiel ein Cluster aus vernetzen PCs zu Grunde legen, müsste ein Sortieralgorithmus komplett anders realisiert werden.

Der Odd-Even-Sortieralgorithmus wird im nächsten Kapitel auch als Beispiel auf einem Befehls-Systolischen Array eingesetzt. Dabei wird eine grundlegend andere Architektur genutzt, so dass zur Aktivierung und Kommunikation auch andere Verfahren eingesetzt werden müssen.

Realisiert werden kann ein Algorithmus für den MasPar-Rechner in der Sprache MPL (MasPar Programming Language), die eine Erweiterung von C ist. Zusätzlich sind auch Programme in Fortran oder Parallaxis implementierbar. Im Folgenden wird allerdings für die Implementierung nur MPL betrachtet, da damit alle Funktionen der MasPar ausgenutzt werden können und Programmierumgebungen vorhanden sind.

Betrachtet man den Gesamtbefehlssatz des MasPar-Rechners genauer, so kann man feststellen, dass Befehle für die Parallelverarbeitung in vier Kategorien eingeteilt werden können:

- Kommunikation zwischen Frontend-System und Paralleleinheit
- Schnelle Kommunikation zwischen den Prozessorelementen
- Allgemeine Kommunikation zwischen Prozessorelementen
- Zugriff auf „Plural–Variablen”

Kommunikation zwischen Frontend-System und Paralleleinheit

Für die Kommunikation zwischen dem Frontend-System und dem eigentlichen Parallelrechner stehen, wie in Tabelle 3-1 dargestellt, zahlreiche Befehle zur Verfügung:

Tabelle 3-1: Befehle für die Kommunikation zwischen Frontend- und Parallelrechner

`blockIn()`	`callRequest()`	`awaitReply()`
`blockOut()`	`requestAsync()`	`awaitRequest()`
`copyIn()`	`setupAsync()`	`sendSync()`
`copyOut()`	`checkReply()`	`awaitSync()`

Bei genauerer Betrachtung stellt man allerdings fest, dass alle Befehle nur zum Austausch von Daten zwischen dem Clientrechner und dem In-/Output-Speicher auf der MasPar dienen und deshalb nicht zu den Befehlen zu rechnen sind, die für den eigentlich parallelen Betrieb genutzt werden. Aus diesem Grund werden sie im weiteren Verlauf der Arbeit nur noch vereinzelt als Beispiel herangezogen.

Schnelle Kommunikation zwischen den Prozessorelementen

Für die schnelle Kommunikation zwischen den Prozessorelementen stehen drei Befehlsgruppen zur Verfügung, die in den acht Kommunikationsrichtungen eingesetzt werden können. Sie werden in Tabelle 3-2 aufgezählt:

Tabelle 3-2: Befehle für das XNet

Plain Access	Pipe	Copy
xnetN	xnetpN	xnetcN
xnetNE	xnetpNE	xnetcNE
xnetE	xnetpE	xnetcE
xnetSE	xnetpSE	xnetcSE
xnetS	xnetpS	xnetcS
xnetSW	xnetpSW	xnetcSW
xnetW	xnetpW	xnetcW
xnetNW	xnetpNW	xnetcNW

In der Kategorie der „Plain Access-Befehle“ werden die Daten von einem Prozessorelement zu einem anderen transferiert, ohne dass die dazwischen liegenden Prozessorelemente davon beeinflusst werden.

Der „Pipe-Befehl“ hat das gleiche Ergebnis wie der „Plain“-Befehl, zusätzlich arbeitet er schneller. Allerdings wird der Datentransfer nur ausgeführt, wenn zwischen dem sendenden und empfangenden Prozessorelement kein aktives Prozessorelement vorhanden ist. Liegt ein aktives Element auf der Pipeline, wird der Befehl abgebrochen. Somit muss der Programmierer die jeweiligen Aktivierungen genau kennen, wenn er die gewünschten Ergebnisse erzielen möchte.

Die „Copy-Befehle“ kopieren die transferierten Daten in alle auf dem Kommunikationsweg liegenden Prozessorelemente. Liegt auf dem Weg ein weiteres aktives Element, wird vor diesem der Kopiervorgang abgebrochen.

Die hier vorgestellten Befehle werden bei fast allen Kommunikationen innerhalb des Parallelrechners eingesetzt. Auch im obigen Beispiel des Odd-Even-Sort-Algorithmus werden diese Befehle verwendet. Aus diesem Grund werden sie im weiteren Verlauf der Arbeit noch mehrfach angesprochen und es wird genauer untersucht, wie für den fehlerfreien Einsatz dieser Befehle eine grafische Unterstützung sinnvoll eingesetzt werden kann.

Allgemeine Kommunikation zwischen Prozessorelementen:

Sollte die schnelle Kommunikation nicht möglich sein, da einige oder alle Prozessorelemente „chaotisch“ über das Prozessorfeld kommunizieren müssen, steht ein Befehl zur

Verfügung, der beliebige Kommunikationsmuster zulässt: `router()`. Nachteil des Befehls ist, dass die Übertragungsgeschwindigkeit relativ gering ist. In Kapitel 5.3.3.1 wird der Befehl noch einmal genauer vorgestellt.

Zugriff auf „Plural–Variablen"

„Plural-Variablen" sind Variablen, welche die spezielle Eigenschaft eines SIMD-Rechners unterstützen, auf unterschiedlichen Prozessorelementen den gleichen Befehl auf unterschiedliche Daten anzuwenden. Eine Variable vom Type Plural steht auf allen Prozessorelementen zur Verfügung, der Wert kann aber auf jedem Prozessorelement individuell gesetzt werden.

Innerhalb der Entwicklungsumgebung ist es nicht einfach, eine Plural-Variable von einer singulären Array-Variablen zu unterscheiden. Sie werden gleich angesprochen, lediglich die Definition der Variablen findet mit unterschiedlichen Typen statt. Da die Variablendefinition auch außerhalb der aktuellen Methode bzw. Prozedur und sogar in anderen Dateien stattfinden kann, ist das exakte Erkennen des Variablentyps nur mit einem Parser möglich, der während der Entwicklung im Hintergrund arbeitet. Steht ein Parser nicht zur Verfügung, kann man allerdings in sehr vielen Fällen durch das Variablenargument erkennen, ob es sich um eine Plural-Variable handelt, denn diese haben sehr häufig eine Konstante `iproc`, `ixproc` oder `iyproc` als Parameter, die dazu dient, den Prozessor im Prozessorfeld einzuordnen.

Die Schwierigkeit eine Plural-Variable zu erkennen, spielt bei der Unterstützung durch grafische Komponenten eine große Rolle, da es vom Erkennen abhängt, welche unterstützenden Dialogboxen angeboten werden können bzw. wie der Befehl grafisch dargestellt wird. Die genauen Realisierungen sind in der Arbeit in den Kapiteln 5.3 und 6 zu finden.

3.4 Befehls-Systolisches Array (Systola)

Erstmals vorgestellt wurden Systolische Arrays von H. T. Kung und Leiserson 1979 [KuL79, Bra93]. Dabei werden Datenströme durch ein Netz von Prozessoren geleitet, welche von lokal vorhandenen Programmen bearbeitet werden. Im Gegensatz dazu wurden auch Netzwerke entwickelt, bei denen der Befehlsstrom durch die Prozessoren geleitet wird und die Daten lokal vorhanden sind [KLSS+88]. Eine Ausprägung davon ist das Befehls-Systolische Array (Instruction Systolic Array, ISA).

Die hier betrachtete Systola 1024 der Firma ISATEC, ist ein kleiner Vertreter eines Befehls-Systolischen Arrays, der in den 90er Jahren produziert wurde. Es ist realisiert als PCI-Einsteckkarte für einen PC [ISA98] (siehe Abbildung 3-6).

Abbildung 3-6: Systola 1024

Die Systola besitzt 1024 Prozessoren, die in einem Gitter von 32×32 Elementen angeordnet sind. Die Prozessoren können jeweils lesend auf Daten der vier Nachbarn zugreifen.

Um das Prozessorfeld mit Daten zu speisen, können diese über die nördlichen und westlichen Randprozessoren eingelesen werden. Anschließend besteht ausschließlich über lokale Operationen mit den Nachbarn die Möglichkeit die Daten weiter zu leiten. Somit werden für die Verteilung eines Wertes auf das gesamte Prozessorfeld $O\left(\sqrt{n}\right)$ Takte benötigt, wobei n die Anzahl der Prozessoren ist.

Diese Vorgabe eines globalen Taktes ist auch der größte Unterschied zu einer weiteren Gruppe von Parallelrechnern, den „Wavefront Array Processors“ (WAP), die 1982 von S. Y. Kung und anderen vorgestellt worden sind [KAGB82]. Diese besitzen einen Taktgeber, der jeweils nur für einen eingeschränkten Bereich gültig ist. Im Weiteren sollen WAP-Rechner nicht weiter betrachtet werden. Die Ähnlichkeit der Architektur führt aber dazu, dass die Ergebnisse dieser Arbeit mit relativ wenig Aufwand auch auf WAPs überführt werden können.

Beim Befehls-Systolischen Array werden Befehle im oberen westlichen Feld eingespeist und wandern bei jedem Takt ein Feld nach Süden und Osten. Dadurch erhält man eine Befehlsstruktur, bei der der gleiche Befehl immer in den diagonal angeordneten Prozessorelementen angewandt wird. Abbildung 3-7 zeigt, wie ein Befehl innerhalb von drei Takten durch einen Teil des Feldes wandert.

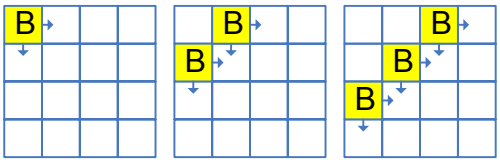


Abbildung 3-7: Diagonale Befehlsstruktur auf der Systola 1024

In der Realisierung der ISATEC-Karte wird der Befehl nur in der nördlichsten Zeile nach Süden und Osten verteilt, ansonsten nur nach Süden. Dies ergibt das gleiche Ergebnis, benötigt aber weniger Leitungen und erspart Synchronisationsaufwand.

Anders als in der vereinfachten Abbildung 3-7 gezeigt, kann nicht nur ein Befehl im Prozessorfeld aktiv sein, sondern es können im Pipelineverfahren ständig weitere Befehle nachgeschoben werden, so dass in den jeweiligen Diagonalen immer die

gleichen Befehle aktiv sind, außerhalb der Diagonalstruktur werden aber unterschiedliche Befehle ausgeführt. Berücksichtigt man nun, dass die Daten zeitgleich mit den Befehlen durch das Prozessorfeld geschoben werden können und vernachlässigt das Füllen und Leeren der Pipeline, so erkennt man, dass ein Befehl bzw. Wert in der Zeit von $O(1)$ verteilt und verarbeitet werden kann. Durch die spezielle Architektur kann die Systola nicht zu den reinen SIMD-Rechnern gezählt werden, obwohl diese Architekturbezeichnung bei Berücksichtigung der zeitlichen Verzögerung entlang der Diagonalen zutrifft.

Da es nicht sinnvoll ist, jeden Befehl in jedem Prozessor auszuführen, gibt es auch bei diesem Parallelrechner die Möglichkeit, die Aktivierung und Deaktivierung einzelner Prozessoren zu steuern. Dazu werden Aktivierungsvektoren eingesetzt, die durch die Prozessoren geschoben werden und jeweils nur die Werte 1 oder 0 bzw. „true“ oder „false“ haben können. Es gibt horizontale und vertikale Aktivierungsvektoren. Beide müssen „true“ sein, damit der Befehl ausgeführt wird. Da es z.B. k Takte Zeit benötigt, bis ein Befehl das oberste Prozessorelement der k-ten Spalten erreicht, werden die Aktivierungsvektoren mit einer Verzögerung von k eingespeist. Die Abbildung 3-8 verdeutlicht das Vorgehen. In der ersten und dritten Spalte wird der Befehl aktiviert, in der zweiten und vierten Spalte wird wegen der False-Werte der Spalten keine Aktion durchgeführt.

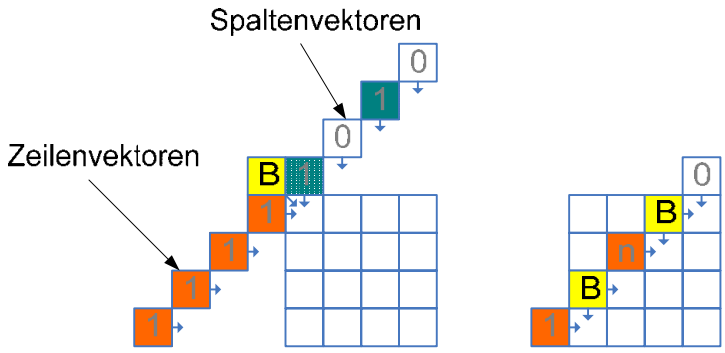


Abbildung 3-8: Aktivierung von Befehlen

Das Modell des massivparallelen Algorithmus kann auch auf diesen Rechnertyp angewandt werden. Die Aktivierung wird durch das Füllen der beiden Aktivierungsvektoren und des Befehlsvektors erreicht. Anschließend kann jedes aktive Element eine Kommunikation mit seinen vier Nachbarn ausführen. Zusätzlich sind lokale Operationen möglich. Die Initialisierung und der Rücktransfer erfolgt an den westlichen und nördlichen Randprozessoren.

Der Aufbau des Algorithmus lässt sich am Beispiel zweier kleiner Beispiele verdeutlichen:

Möchte man die Summe aller Felder ermitteln, so benötigt man dafür nur 2 Takte, wenn man das Pipelining nicht berücksichtigt. Zuerst muss der Wert des nördlichen Feldes zum eigenen addiert werden. Dies wird in allen Feldern außer der nördlichsten Zeile durchgeführt. Anschließend wird der Wert des westlichen Nachbarn zum eigenen

addiert. Dieser Befehl wird nur in der südlichsten Zeile und ohne das westlichste Element ausgeführt. In der rechten unteren Ecke steht zum Schluss die Summe. Abbildung 3-9 zeigt eine grafische Darstellung des Algorithmus.

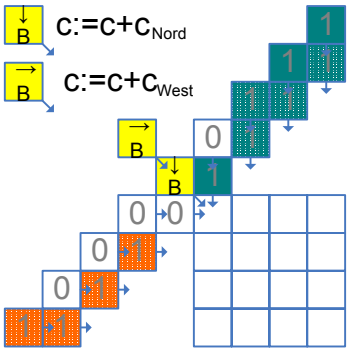


Abbildung 3-9: Summenbildung auf der Systola

Ein Sortieralgorithmus nach dem Odd-Even-Prinzip lässt sich ebenfalls auf der Systola sehr einfach realisieren. Wie auf der MasPar muss innerhalb einer Zeile verglichen werden, ob der lokale Wert oder der Wert des Nachbarn größer oder kleiner ist und gegebenenfalls die Daten getauscht werden müssen. Dazu werden nach Schmeck jeweils $n-1$ mal abwechselnd die Funktionen $C = \min(C, C_{Süd})$ und $C = \max(C, C_{Nord})$ ausgeführt, um die Sortierung innerhalb einer Spalte zu erreichen und anschließend jeweils $n-1$ mal abwechselnd $C = \min(C, C_{Ost})$ und $C = \max(C, C_{West})$, um die Zeilen zu sortieren. Dabei ist es nicht notwendig, dass zu Beginn und am Ende des Verfahrens alle Prozessorelemente aktiv sind, so dass sich die in Abbildung 3-10 abgebildete Struktur ergibt.

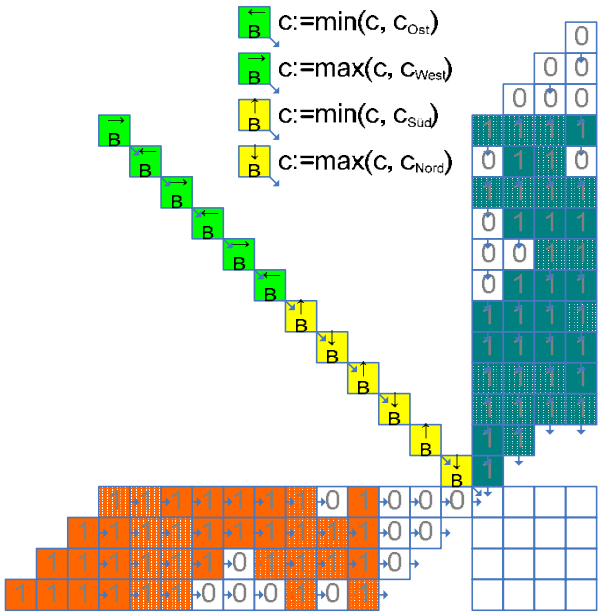


Abbildung 3-10: Sortieralgorithmus auf der Systola

Allgemein für ein Befehls-Systolisches Array wurde der Sortieralgorithmus bereits in [Sch86] beschrieben.

Weitere, komplexere Algorithmen werden in der Dissertation von B. Schmidt [Sch99] vorgestellt. Hier wird beispielsweise gezeigt, wie Fourier-Transformationen oder Bildmanipulationen durchgeführt werden können. In vielen Erklärungen wird dabei auf eine grafische Darstellung innerhalb einer Matrix zurückgegriffen, um die Arbeitsweise des Algorithmus erklären zu können.

Die Realisierung der parallelen Programme erfolgt in der Programmiersprache LAISA (LAnguage for Instruction Systolic Array), die zusammen mit der PCI-Karte ausgeliefert wird. Die Sprache erinnert stark an Assemblersprachen. Es stehen aber die aus Hochsprachen bekannten Schleifenkonstrukte und Bedingungen zur Verfügung. Das LAISA-Programm kann allerdings in Hochsprachen wie Pascal und C(++) von Borland eingebettet werden, so dass nur die parallelen Teile in LAISA programmiert werden müssen. Programmteile, die auf dem Hauptprozessor des Computers abgearbeitet werden sollen und die für die Kommunikation mit dem Prozessorfeld zuständig sind, können in der Hochsprache mithilfe spezieller Bibliotheken implementiert werden.

Die ISATEC-Karte steht in dieser Arbeit als Vertreter der „räumlich kleinen“ Parallelrechner. In noch kleinerem Maßstab wird die Parallelverarbeitung heute in fast allen Computerprozessoren eingesetzt. So benutzt Intel für die Befehlerweiterung der MMX-Befehle parallele Einheiten mit SIMD-Technik [CoR04]. Noch massiver wird die SIMD-Technik bei heutigen Prozessoren für Grafikkarten eingesetzt. Die Teile des Prozessors, die für Texturen und die Schattenbildung zuständig sind, werden alle parallel ausgelegt, so dass einzelne Pipelines nur einen bestimmten Bereich der Grafik bearbeiten müssen.

Dies zeigt, dass neben den parallelen Großrechnern auch kleine parallele Einheiten mit SIMD-Technik heute sehr weit verbreitet sind und ständig weiter entwickelt werden. Trotzdem wird im Folgenden nicht näher auf diese Typen von Rechnern eingegangen, da die Programmierung der Spezialprozessoren nur von sehr wenigen Personen durchgeführt wird. Das in Kapitel 2.1 zu Grunde gelegte Benutzermodell trifft auf diese Personengruppe nicht zu, da diese von ihren Firmen speziell ausgebildet werden und daher neben den sehr guten Kenntnissen der zugrunde liegenden Programmiersprache auch von sehr guten Kenntnissen der Spezialbefehle und deren Besonderheiten auszugehen ist. In der in Kapitel 2.1 eingeführten Lernkurve befindet sich der Personenkreis in der rechten, oberen Ecke (viel Erfahrung und Effizienz, lange Arbeitszeit). Wichtig ist deshalb die schnelle Arbeitsweise zu unterstützen, was nach heutigen Stand der Technik durch einige der Methoden, die in den Kapiteln 5.1 und 5.2 vorgestellt werden, erreicht werden kann. Eine grafische Unterstützung wird meist nur in Spezialfällen benötigt.

3.5 RMesh / Rekonfigurierbare Gitter

Der Begriff „rekonfigurierbare Gitter“ deutet bereits an, dass bei diesem Rechnertyp das „Gitter“, also die Verbindung zwischen den einzelnen Prozessorelementen, verändert werden kann. Dadurch wird es möglich, Daten auf sich dynamisch verändernden Wegen auszutauschen. Diese Möglichkeit nutzt man, um effiziente Algorithmen zu entwerfen.

Ein „rekonfigurierbares Gitter“ wird in dieser Arbeit als SIMD-Rechner angesehen, da zu einem Zeitpunkt nur ein Befehl auf dem Prozessorfeld ausgeführt werden kann. Die Prozessorelemente sind mit einem zweidimensionalen Gitter verbunden, das in einigen Modellen zu einem Torus erweitert wird. Man spricht von einem Torus, wenn ein äußerer Prozessor direkt mit seinem gegenüberliegenden Prozessor verbunden ist (siehe Abbildung 3-11). Jedes Prozessorelement hat dabei direkten Kontakt zu seinen vier Nachbarn (Nord, Süd, West, Ost).

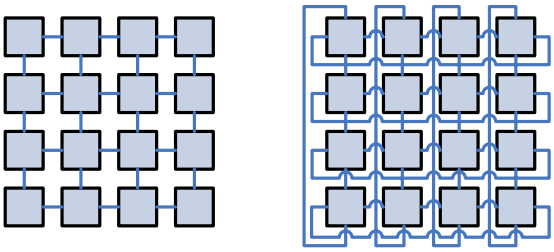


Abbildung 3-11: Zweidimensionales Gitter und Torus

Die Verbindungen der Prozessorelemente sind nicht fest verdrahtet, sondern lassen sich durch Software verändern. Werden beispielsweise auf allen Prozessorelementen die Nord- und Südanschlüsse miteinander verbunden, so können innerhalb der Spalten gleichzeitig die Daten von dieser Verbindung gelesen und an den Nachbarn weiter geleitet werden. Bei geeigneter Wahl der geschlossenen Verbindungen können dann je nach zu Grunde liegendem Modell Informationen in der Zeit O(1) oder O(log n) alle Prozessorelemente erreichen. Der Unterschied ist dadurch zu erklären, dass einerseits davon ausgegangen wird, Daten können in konstanter Zeit über alle Prozessorelemente verteilt werden, also unabhängig davon, wie weit diese voneinander entfernt sind. Andere Modelle berücksichtigen, dass bei getakteten Realisierungen Daten nur eine bestimmte Strecke zurücklegen können, so dass das komplette Gitter innerhalb einer Baumstruktur realisiert werden muss. Auf die Art der Algorithmen hat diese Unterscheidung meist keinen Einfluss.

Bei rekonfigurierbaren Gittern stehen die in Abbildung 3-12 dargestellten Verbindungen zur Verfügung.

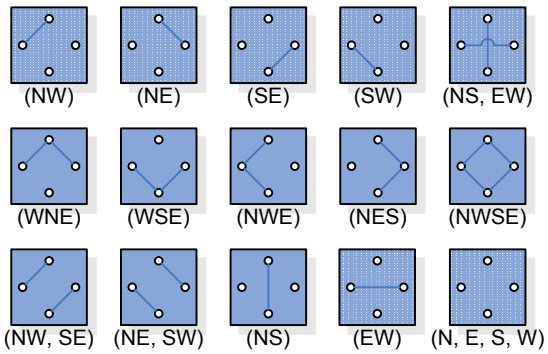


Abbildung 3-12: Verbindungen in rekonfigurierbaren Gittern

Für rekonfigurierbare Gitter existieren mehrere Modelle, welche unterschiedliche Anforderungen an die Hardware stellen und deshalb nicht immer alle Verbindungen

zulassen. Die einfachen Modelle erlauben nur maximal eine Verbindung je Prozessorelement, andere erlauben auch zwei Verbindungen, die sich aber nicht überschneiden dürfen, so dass die (NS, EW)-Verbindung nicht erlaubt ist. Im Weiteren wird davon ausgegangen, dass alle Verbindungen erlaubt sind. Diese Vielfalt an Möglichkeiten erschwert den Aufbau einer ergonomischen Benutzungsunterstützung, so dass alle einfacheren Modelle leicht abgeleitet werden können.

Wendet man das in Kapitel 3.2 vorgestellte Modell eines massivparallelen Algorithmus auf ein rekonfigurierbares Gitter an, so muss in einem ersten Schritt eine Aktivierung der Prozessorelemente stattfinden. Diese Aktivierung wird durch die Konfiguration der einzelnen Prozessorelemente und Busse übernommen. Der zweite Schritt der Kommunikation findet auf den vorher definierten Bussen statt. Im dritten Schritt können alle Prozessorelemente lokal die empfangenen Daten auswerten und damit erneut entscheiden, wie anhand des Algorithmus neue Schaltungen vorgenommen werden müssen.

Zur Verdeutlichung soll ein einfacher Algorithmus dienen, der die Anzahl von 0en und 1en zählen soll, die in einer bestimmten Zeile vorkommen. Dazu wird in einem ersten Schritt in allen Spalten eine (NS)-Verbindung aufgebaut und danach die 0en und 1en der Spalte in alle nördlichen und südlichen Nachbarn verteilt. Abbildung 3-13 zeigt den Aufbau der Schaltung, so dass die Daten in Nord-Süd-Richtung verteilt werden können.

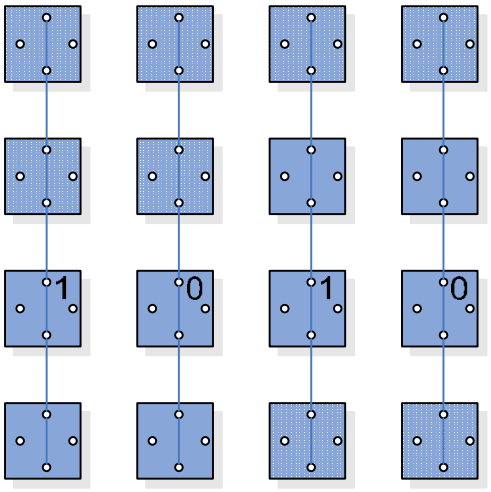


Abbildung 3-13: Verteilen von Daten in Nord-Süd-Richtung

Anschließend werden die Verbindungen neu aufgebaut. Dabei schalten alle Prozessorelemente mit einer lokal vorliegenden 0 eine (WE)-Verbindung und Prozessorelemente mit einer lokalen 1 eine (NW, SE)-Verbindung. Anschließend legt man an den Westeingang des linken unteren Prozessorelements eine 1 an. So erhält nur ein Ostausgang eines östlichen Prozessorelementes eine 1. Anhand der Zeilennummer des Prozessorelementes kann nun ermittelt werden, wie viele 1en zu Beginn in einer Zeile gesetzt waren. Man erhält einen Algorithmus, der in konstanter Zeit Bits zählen kann, allerdings quadratisch viele Prozessorelemente benötigt. Abbildung 3-14 zeigt die abschließende Schaltung.

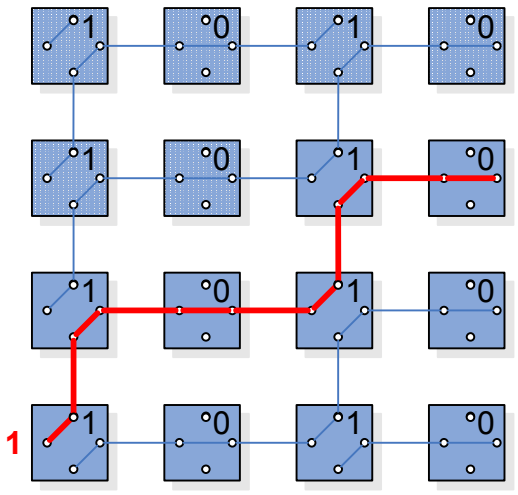


Abbildung 3-14: Algorithmus zum Zählen von Bits

Auch Algorithmen, welche das Maximum von n Zahlen ermitteln oder sogar n Zahlen sortieren können, können in konstanter Zeit das Ergebnis liefern [NiS92]. Aber auch hier werden n² Prozessorelemente benötigt.

Die Abfrage von n-Bit AND- oder OR-Verknüpfungen können bei konstanter Zeit bereits mit nur n Prozessorelementen durchgeführt werden. Alle Algorithmen benötigen bei ihrer Ausführung die Schritte Aktivierung, Kommunikation und lokale Berechnung

3.6 Diskussion des Modells für massivparallele Algorithmen

In den letzten drei Abschnitten wurde das Modell für massivparallele Algorithmen auf drei sehr unterschiedliche Rechnertypen angewandt. In allen drei Fällen konnte ein für die Architektur typischer Algorithmus mit dem Modell abgebildet werden, ohne dass dadurch Einschränkungen zu beachten gewesen wären. Das einfach gehaltene Modell kann besonders den Datenfluss abstrahieren, indem es diesen in die beiden Schritte „Aktivierung“ und „Kommunikation“ teilt und dadurch sehr gut an die unterschiedlichsten Architekturen anpassbar ist. Dabei fällt auf, dass beide Schritte visualisiert werden können, so dass der Benutzer auf einfache Weise eine Übersicht über die Architektur des Rechners und den möglichen Datenfluss erhält. Dies unterstützt besonders die später betrachtete ergonomische Sichtweise der Hilfestellungen.

Die Besonderheiten einzelner Rechner wie die SIMD-Architektur oder das Pipelining von Daten können jeweils berücksichtigt werden. Auch die Unterscheidung von parallelem und sequentiellem Code stören auf der abstrakten Ebene des Modells nicht, obwohl hier bei den unterschiedlichen Architekturen sehr große Unterschiede bestehen, da teilweise die Parallelisierung nur durch verschiedene Datentypen innerhalb eines einheitlichen Quellcodes dargestellt wird bzw. ganz unterschiedliche Programme in verschiedenen Programmiersprachen zu implementieren sind.

Im Folgenden sollen noch einmal kurz die Schritte des Modells und die Umsetzung auf den einzelnen Rechnersystemen betrachtet werden.

Bei allen Rechnertypen ist ein Schritt der Initialisierung bzw. am Ende der Schritt des Zurückschreibens der Daten notwendig. Da massivparallele Rechner im Allgemeinen über keine direkte Benutzerinteraktion verfügen, sind diese Schritte in jedem Fall notwendig.

Die Aktivierung von Prozessoren wird bei den jeweiligen Rechnertypen sehr unterschiedlich ausgeführt, ist aber trotzdem bei allen notwendig. Gleich ist bei allen bisher vorgestellten Rechnertypen, dass man eine Aktivierung beispielhaft an einem Teil des Gesamtrechners durchführen und das Ergebnis auf den ganzen Parallelrechner übernehmen kann. Die bisher nicht betrachteten Rechencluster, welche eine MIMD-Architektur (Multiple Instruction, Multiple Data) unterstützen, erfüllen diese Übertragbarkeit nicht, da jeder Prozessor unterschiedliche Programme bearbeitet. Eine Aktivierung einer bestimmten Prozessorgruppe und eine Zuordnung von zu bearbeitenden Daten sind aber auch bei MIMD-Architekturen notwendig und können von dem Modell unterstützt werden.

Die Kommunikation erfolgt je nach Rechnertyp ebenfalls sehr unterschiedlich. Einmal werden Daten auf einem vordefinierten Netz übertragen (beispielsweise das XNet der MasPar), dann können die Daten nur mit den unmittelbaren Nachbarn getauscht werden und der Datenfluss über den Gesamtrechner muss durch ein ständiges „Weiterschieben“ durchgeführt werden (siehe Systola) oder es besteht die Möglichkeit die Daten über einen vorher definierten Weg über weite Strecken auf viele verschiedene Prozessoren zu übertragen (RMesh).

Allen gemeinsam ist aber die zeitliche Reihenfolge, in der Aktivierung und Kommunikation stattfinden, so dass jeweils das Modell für massivparallele Rechner sinnvoll angewandt und damit der Aufbau des Algorithmus verdeutlicht dargestellt werden kann.

Die Darstellung der Kommunikation kann und muss jeweils dem Rechner angepasst werden. Bei den in den letzten Kapiteln vorgestellten Algorithmen wurden dabei Methoden kurz angedeutet, wie dies ergonomisch sinnvoll möglich ist.

Lokale Berechnungen werden auf allen Rechnern ausgeführt. Trotz großer Unterschiede bei der Programmierung können diese immer abgebildet werden. Dabei spielt es keine Rolle, ob parallele Programmeinheiten durch allein stehende Quelltexte mit speziellen Programmiersprachen entstehen oder die lokalen Berechnungen in den kompletten Programmquelltext integriert werden können.

Betrachtet man die bis jetzt nicht angesprochenen Clusterrechner, so lässt sich das Modell des Algorithmus ebenfalls darauf anwenden. Allerdings ist der „Rhythmus“, in dem die einzelnen Schritte abgearbeitet werden, ein anderer. Die Aktivierung wird nur selten durchgeführt und anschließend über relativ lange Zeit auf den jeweiligen Prozessoren ein komplexes Programm abgearbeitet, das wiederum in größeren Abständen eine Kommunikation zu anderen Prozessoren benötigt.

Genauso verhält es sich mit dem Jobverteilungssystem "JoSchKa", mit dem auf Basis von Web-Services Rechenjobs auf beliebigen Clients ausgeführt werden können. Derzeit werden dafür über 100 Clients in den Poolräumen der Fakultät für Wirtschafts-

wissenschaften eingesetzt [BTS05]. Mit Joschka können in einem ersten Schritt Daten und auszuführende Programme vom Arbeitsplatz auf einen zentralen Server geschoben werden. Beliebige Rechner können sich durch ein Hintergrundprogramm selbstständig aktivieren und ihre Rechenleistung zur Verfügung stellen. Sie erfragen beim Server eine Aufgabenbeschreibung, die von ihnen auszuführende Programme und notwendige Daten auflistet. Anschließend können sie die zugeteilten Jobs der Reihe nach auswählen, herunterladen und starten. Nach Beendigung der lokalen Berechnungen werden die Daten zum Server übertragen und können nun wieder vom Arbeitsplatzrechner abgeholt werden. In einem nächsten Ausbauschritt wird Joschka die Kommunikation zwischen den Clients unterstützen.

Trotz der möglichen Einbindung von Clustern oder Joschka in das Modell für massivparallele Rechner, sollen sie hier nicht weiter betrachtet werden, da der Schwerpunkt der Arbeit auf der grafischen Benutzungsunterstützung liegen soll. Hier wäre durch die unterschiedliche Architektur ein anderer Ansatz zu wählen, als das in der folgenden Arbeit gemacht wird.

Es konnte gezeigt werden, dass mit einem einheitlichen Modell für massivparallele Rechner unterschiedliche Typen von Parallelrechner unterstützt werden können. Die einheitliche Beschreibung der Schritte Aktivierung und Kommunikation soll im Weiteren genutzt und jeweils Benutzungsunterstützungen erarbeitet werden, welche diese Schritte auf ergonomische Weise unterstützen.

Bevor allerdings neue grafische Hilfestellungen auf Befehlsebene für die Erstellung paralleler Programme dargestellt werden, werden im nächsten Kapitel vorhandene Entwicklungsumgebungen und deren Benutzungsunterstützung vorgestellt.

4 Stand der Technik – Ausgewählte Entwicklungsumgebungen

Das Forschungsgebiet der Arbeit streift eine Vielzahl an Themen: Benutzungsunterstützung, Ergonomie, Parallelrechner und Entwicklungsumgebungen. Aktuelle Grundlagen der Ergonomie und Benutzungsunterstützung wurden bereits im letzten Kapitel angesprochen. Parallelrechner sind „nur ein Mittel zum Zweck“, so dass hier in den weiteren Kapiteln die notwendigen Rechner (Massivparallel, Cluster, Grid, …) an den jeweiligen Stellen nur kurz angesprochen werden. Die Entwicklungsumgebungen bilden die Grundlage für alle weiteren Arbeiten. Im Folgenden werden aus unterschiedlichen Bereichen verschiedene Vertreter vorgestellt und deren jeweiligen Einsatzgebiete angesprochen. Details, die eine besondere Benutzungsunterstützung beschreiben, werden ausführlich in Kapitel 5 behandelt und in bestimmten Bereichen durch neue Vorschläge erweitert.

Sucht man in der Literatur nach grafischer Benutzungsunterstützung oder Entwicklungsumgebungen, so findet man eine sehr große Anzahl an Artikeln, die sich mit diesen Themen beschäftigen.

Um einen Überblick zu gewinnen, wird zuerst eine grobe Gruppierung durchgeführt. Auf das Thema der Arbeit bezogen eignet sich dazu als erstes Kriterium die Unterscheidung zwischen Entwicklungsumgebungen bzw. Benutzungsunterstützungen für Einprozessorsysteme und Parallelrechner. Anschließend können für Einprozessorsysteme Kriterien wie Mächtigkeit, vorhandene Werkzeuge (siehe Kapitel 2.5) oder unterstützte Programmiersprachen herangezogen werden.

Bei Parallelrechnern fällt als erstes die große Unterstützung von Message-Passing Systemen auf. Deshalb ist es sinnvoll, zuerst nach Entwicklungsumgebungen für

Message-Passing Systeme und „Sonstige“ zu unterscheiden. Auf diese Weise kann der Block der „Spezialisten für bestimmte Großrechner“ von der Masse der Systeme abgetrennt werden, die Message-Passing unterstützen. Ein weiteres Unterscheidungskriterium ist die jeweilige Spezialisierung auf die Unterstützung wichtiger Eigenschaften von Parallelrechnern, wie die Geschwindigkeitsoptimierung, Minimierung von Platzbedarf oder Kommunikationsaufwand oder die Visualisierung von Zusammenhängen. Zusätzlich können ähnliche Kriterien wie bei Einprozessorsystemen herangezogen werden. In Kapitel 4.3 werden für die Arbeit wichtige Entwicklungsumgebungen genauer vorgestellt.

Trotz der Unterteilung nach mehreren Kriterien finden sich Gemeinsamkeiten, die alle heutigen Systeme enthalten müssen, wenn sie dem „Stand der Technik“ entsprechen wollen, indem sie die Benutzungsfreundlichkeit auf die eine oder andere Weise besonders unterstützen. Auf diese Gemeinsamkeiten wird zuerst eingegangen.

4.1 Allgemeines

Auf allen Computerplattformen hat sich heutzutage eine multiprogrammfähige grafische Benutzungsoberfläche durchgesetzt. Durch den „Multitasking-Betrieb“ wurde die Möglichkeit geschaffen, gleichzeitig mehrere laufende Programme auf dem Bildschirm anzuzeigen, so dass der interaktive Austausch oder Vergleich von Daten zwischen zwei Programmen vereinfacht wurde. Zusätzlich stellen fast alle aktuellen Programme ihre Informationen in mehr als einem Fenster dar. So werden zum Beispiel Hilfstexte in eigene Fenster ausgegliedert oder eine Sammlung von Schaltflächen frei platzierbar in einem eigenen Fenster angeordnet.

Dabei wird die „Mehrfenstertechnik“ nicht nur für die Unterstützung eines einzelnen Vorgangs eingesetzt, sondern sie erlaubt auch die Durchführung von unabhängigen Aufgaben nebeneinander innerhalb eines einzelnen Programms. Beispielweise können in einer Textverarbeitung mehrere Dokumente geöffnet und nebeneinander dargestellt werden. Genauso hat sich die Benutzung mehrerer Fenster innerhalb eines Programms bei den meisten der heutigen Editoren etabliert. Obwohl die einzelnen Texte unabhängig sind, ist ein gleichzeitiges Öffnen oft sehr hilfreich, da der Quelltext von größeren Projekten immer in mehrere Dateien geteilt wird und dadurch häufig Informationen aus mehreren Dateien gleichzeitig benötigt werden.

Durch die Fenstertechnik ist neben der Tastatur die Maus das wichtigste Hilfsmittel für die ständige Arbeit geworden. Durch die Eigenschaften einer Maus wird eine Positionierung des Cursors auf dem Bildschirm möglich, die wesentlich schneller und intuitiver vorgenommen werden kann, als dies über die Tastatur möglich wäre. Lediglich ein Tablet-PC oder ein Touchscreen erfüllen die Aufgaben der „Erwartungskonformität“ noch besser, als dies die Maus kann.

Fast alle modernen Editoren benutzen mehr oder weniger die Möglichkeit, den Benutzer mit grafischen Elementen zu unterstützen. In den meisten Fällen wird zumindest eine Icon-Leiste zur Verfügung gestellt, durch die auf schnelle Weise bestimmte Funktionen

aktiviert werden können. Icons und deren Eignung für die intuitive Nutzung bestimmter Aufgaben durch entsprechende Metaphern werden bereits seit Ende der 1980er Jahre untersucht und eingesetzt [CMK88, Roh90].

Eine weitere allgemeine Form die Benutzungsfreundlichkeit zu steigern, soll die „grafische Benutzungsunterstützung auf Befehlsebene“ werden, die in dieser Arbeit entwickelt wird. Dabei wird sie zuerst für die Entwicklung eines speziellen SIMD-Rechners entwickelt. Anschließend wird gezeigt, wie die Technik auch für andere Rechnersysteme sinnvoll eingesetzt werden kann.

Im Folgenden werden zuerst Entwicklungsumgebungen angesprochen, welche für die Entwicklung von Einprozessorprogrammen eingesetzt werden. Hier wird auf einen vollständigen Überblick der vorhandenen Systeme verzichtet, da durch entsprechende Toolsets die Erstellung von „einfachen“ Editoren heute sehr leicht möglich ist und somit Tausende von verschiedenen Ausprägungen existieren. Deshalb werden nur die interessantesten Aspekte einiger ausgewählter Entwicklungsumgebungen besprochen. Anschließend werden Entwicklungsumgebungen betrachtet, mit denen speziell die Entwicklung paralleler Programme unterstützt wird. Dabei wird ein kurzer Überblick über die Vielzahl vorhandener Entwicklungsumgebungen gezeigt. Anschließend werden zwei Umgebungen genauer vorgestellt, mit denen im späteren Verlauf der Arbeit die hier vorzustellenden grafischen Benutzungsunterstützungsmöglichkeiten verglichen werden, bzw. wie diese auf sinnvolle Weise erweitert werden könnten.

4.2 Entwicklungsumgebungen für Einprozessorprogramme

Hier ist der Hauptabsatzmarkt der heutigen Entwicklungsumgebungen zu sehen. Deshalb gibt es eine fast unüberschaubare Anzahl an verschiedenen Produkten. Nicht alle Produkte können und müssen hier vorgestellt werden, wobei die Entwicklungen in drei Kategorien eingeteilt werden können:

- Kleine, aber immer noch aktuelle konsolenbasierte Editoren
- aktuelle Open Source-Software in verschiedenen Ausprägungen
- extrem mächtige Neuentwicklungen der marktführenden Firmen

Stellvertretend für alle werden die jeweils aktuellen Techniken der typischen Vertreter vorgestellt.

4.2.1 Konsolenbasierte Editoren

Unter einem konsolenbasierten Editor versteht man ein Programm, das nicht in einem eigenen Fenster abläuft, sondern ein übergeordnetes Programm als „Wirt“ nutzt. Dieses Wirtsprogramm wird unter Linux „shell“ und unter Windows „Eingabeaufforderung“ genannt. Der Ursprung dieser Programme liegt in den Textkonsolen von Unix oder DOS. Grafische Oberflächen existierten noch nicht oder deren Start dauerte zu lange und der Betrieb der Oberfläche war nicht stabil genug.

Noch heute wichtige und viel benutzte Vertreter der konsolenbasierten Editoren sind unter Linux/UNIX `VI` und `Emacs`. Unter Windows/DOS ist `edit` ein Vertreter der konsolenbasierten Editoren. Dieser wurde aber weitgehend durch `notepad` abgelöst, das in einem eigenen Fenster läuft, sonst aber kaum Zusatzfunktionen hat.

VI

`VI` (steht für „Visual Interface“) ist der Klassiker unter den Editoren auf Unix-Systemen und löste den Zeileneditor `ed` ab. `VI` wurde 1976 von Bill Joy entwickelt, war schnell ein De-facto-Standard auf allen UNIX-Systemen und wird heute noch gepflegt [Rit05]. Seine Vorteile sind, dass er auf jedem Unix-System installiert ist, sehr schnell geladen werden kann, immer gleich zu bedienen ist, ohne grafische Oberfläche auf einer Textkonsole funktioniert und durch viele vordefinierte Tastenkombinationen sehr effektiv eingesetzt werden kann. Durch diese Eigenschaften wird er auch heute noch von sehr vielen Administratoren häufig eingesetzt, besonders bei der Änderung kleiner Skripte oder beim Durchsuchen von Logbüchern. Auch die Möglichkeit des Einsatzes von regulären Ausdrücken in Suchanfragen zeigt die besondere Unterstützung für entsprechende Experten. Diese erweiterte Such-Funktionalität wird von vielen „modernen“ Editoren nicht angeboten.

Die Vorteile von `VI` sind auch gleichzeitig seine Nachteile. So kann sich ein ungeübter Benutzer nur sehr schwer die wichtigsten Tastenkombinationen merken. Nur wenige andere Programme unterstützen noch diese Tastenkombinationen. Ohne diese kann der Editor allerdings nur als „Viewer“ eingesetzt werden, da vor einer Änderung erst mit einer der Tasten „`i`“, „`I`“, „`a`“, „`A`“, „`o`“ oder „`O`“ in einen „Edit-Modus“ gewechselt werden muss. Allerdings ist auch das Beenden des Editors ohne eine bestimmte Tastenkombination nicht möglich. Eine Hilfe kann ebenfalls nur durch Tastenkombinationen angezeigt werden. „Drag&Drop“ steht nicht zur Verfügung. Auch Syntax-Highlighting, Textfalten und viele weitere typische Editierfunktionen können nicht bzw. nur von der Weiterentwicklung `vim` (Vi IMproved, 1991) angeboten werden, die von Bram Moolenaar stammt und dessen letzte Version 7 Mitte 2006 freigegeben wurde [Moo06].

Emacs

`Emacs` ist ein „Nachfolger“ von `VI`. Er wurde 1976 am MIT als Makrosammlung des Editors TECO entwickelt, daher auch der Name „Editor MACroS“ [Sta81]. Bekannt wurde er, als er 1984 von Richard Stallman neu in C implementiert wurde und als erstes Programm des GNU-Projektes entstand. Zuerst konsolenbasiert, wurde 1991 von Lucid Inc. eine Version mit eigenem Fenster und Menüs entwickelt: `XEmacs`. Eine Besonderheit von `Emacs` bzw. `XEmacs` ist, dass er einfach durch Dritte mit so genannten „Plugins“ erweitert werden kann. Diese Erweiterungen müssen in der Sprache Lisp entwickelt werden. Durch die Vielzahl der vorhandenen „Plugins“ hat sich `XEmacs` nicht nur einen Namen unter den Programmieren erobert, sondern er wird auch als Editor für TeX-Texte oder HTML-Quellen eingesetzt. Leider sind seine Tastenkombinationen nicht zu denen unter den meisten grafischen Oberflächen bekannten kompatibel. Allerdings

können fremde Editoren teilweise an Emacs-Tastenkombinationen angepasst werden, wodurch ein Umstieg erleichtert wird.

Die Funktionalität von Plugins wurde in viele der heutigen Programme übernommen. Am bekanntesten ist der Einsatz in Browsern, die dadurch beispielsweise Filme, Ton, Animationen oder Formeln darstellen können. Auch Office-Pakete, Grafiksoftware und viele Programmierumgebungen haben diese Erweiterbarkeit übernommen.

4.2.2 Open Source-Software

Stellvertretend für die große Anzahl an frei verfügbaren Entwicklungsumgebungen sollen hier nur zwei Programme angesprochen werden: Joe als kleiner, spezialisierter Editor und Eclipse als universelle Entwicklungsumgebung, die in letzter Zeit als Grundlage für sehr viele Projekte genutzt wird.

Joe

Joe (Java Oriented Editor, http://www.javaeditor.de), ist ein Beispiel für einen frei erhältlichen, kleinen, aber sehr leistungsfähigen Editor, der von Timo Haberkern entwickelt wurde. Abbildung 4-1 zeigt einen Screenshot des unter Windows laufenden Editors, der für die Entwicklung von Java-Programmen vorgesehen ist.

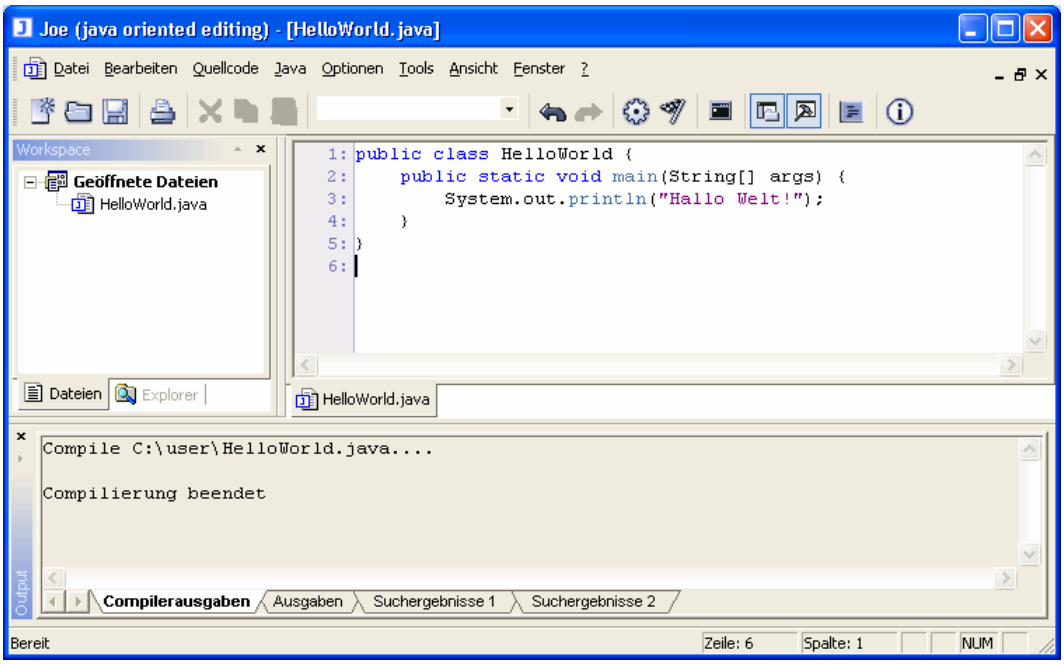


Abbildung 4-1: Screenshot JOE

Möchte man nur kleine Projekte entwickeln, stellt er eine sehr gute Alternative zu größeren Entwicklungsumgebungen dar. Nach dem ersten Starten kann man direkt mit der Eingabe des Quellcodes beginnen. Der Quelltext wird mit Syntax-Highlighting besser lesbar gestaltet, eine Projektübersicht steht zur Verfügung, die Icon-Leiste bietet leicht verständlich alle wichtigen Funktionen. Dabei ist sogar ein „Compile-Icon“ integriert, über das der Quelltext compiliert werden kann. Im unteren Bereich des Editors wird die Ausgabe des Compilers mit eventuellen Fehlermeldungen angezeigt. Außerdem

kann das entwickelte Programm direkt aus dem Editor gestartet werden. Bei bekannter Syntax ist ein Hello-World-Programm in drei Minuten geschrieben, compiliert und ausgeführt, ohne dass man vorher den Editor kennen lernen musste.

Eclipse

In den letzten Jahren hat sich Eclipse zum „wichtigsten“ Vertreter unter den frei verfügbaren Entwicklungsumgebungen entwickelt. Er ist der Nachfolger von IBMs Visual Age for Java 4.0 und wurde 2001 von IBM im Quelltext freigegeben. Zurzeit steht er in der Version 3.1 für die Betriebssysteme Windows, Linux, MAC OS X, HP-UX, AIX und Solaris zur Verfügung. Eclipse ist vollständig in Java geschrieben. Seine Besonderheit ist die sehr gut dokumentierte Erweiterbarkeit durch Plugins, für die eigene Bücher zur Verfügung stehen [GaB04]. Dies zeigt sich auch darin, dass Eclipse bis zur Version 2.1 als erweiterbare Entwicklungsumgebung konzipiert war, ab der Version 3.0 stellt es allerdings nur noch den eigentlichen Kern der Anwendung da, die durch mittlerweile über 1000 verfügbare Plugins individuell erweitert werden kann (http://www.eclipse-plugins.info). Die ursprünglich für Java zur Verfügung gestellte Entwicklungsumgebung kann dank der Plugins für fast alle Programmiersprachen eingesetzt werden und dabei die Besonderheiten der jeweiligen Sprache unterstützen. Zusätzlich hat sich Eclipse zur Plattform für Webentwicklungen entfaltet, indem es XML, Web-Services, Tomcat-Erweiterungen, CSS und vieles mehr unterstützt. Auch kommerzielle Anbindungen beispielsweise für SAP stehen zur Verfügung.

Je nach Plugin stehen viele Arten von Benutzungsunterstützungen zur Verfügung, angefangen von Assistenten bzw. Wizards, über Textfalten, Syntax-Highlighting oder Auto-Vervollständigung, bis hin zu Plugins, die den Quelltext von GUI-Interfaces erzeugen, indem man sie mit der Maus auf dem Bildschirm „zeichnet“ und alle Elemente wie gewünscht platziert.

Durch die Mächtigkeit von Eclipse ist es allerdings auch nicht mehr ganz so einfach wie bei JOE erste Projekte zu verwirklichen. So muss man einige Zeit suchen, bis man es schafft ein erstes „Hello World“ zu schreiben. Im Funktionsumfang steht Eclipse mittlerweile gleichauf mit kommerziellen Umgebungen und kann diese sogar bei Spezialanwendungen übertrumpfen.

4.2.3 Kommerzielle Produkte

Unter Markenprodukten versteht man große Softwarepakete, die eine Vielzahl an Programmen und Funktionalitäten zur Verfügung stellen, um den Entwickler zu unterstützen. Die bekanntesten Produkte stammen von Microsoft (Visual Studio 2005) oder Borland (Delphi, JBuilder).

Visual Studio

Ein vollständig installiertes Visual Studio mit allen Hilfetexten benötigt weit mehr als 1 GB Festplattenspeicher, alleine die Hilfetexte (MSDN Libraries) werden auf 3 CDs

geliefert. Dies alleine zeigt schon seinen Unterschied zu JOE, das ca. 1 MB Speicherplatz benötigt.

Visual Studio 2005 unterstützt die Programmiersprachen C++, C#, J++ und Visual Basic. Nach der Sprachauswahl kann man genauer definieren, was für eine Anwendung entwickelt werden soll (beispielsweise eine Windowsanwendung, eine Konsolenanwendung oder eine Bibliothek). Anschließend wird ein entsprechendes Projekt erzeugt, bei dem „nur“ noch der anwendungsspezifische Quellcode an den richtigen Stellen eingetragen werden muss. Bei Konsolenanwendungen wird ein fertiges „Hello World“ vorgegeben, das nur noch compiliert werden muss, bevor es gestartet werden kann. Bei Windowsanwendungen steht das „visuelle Programmieren“ im Vordergrund. Dabei muss man für ein lauffähiges Programm einzelne Bausteine von einer „Toolbox“ mit der Maus in das Arbeitsfenster ziehen. Ein Baustein kann dabei beispielsweise ein Button sein. Damit bei einem lauffähigen Programm anschließend die gewünschte Reaktion auf den Buttondruck entsteht, muss nun der erste Code geschrieben werden, indem man über einen Doppelklick auf eine Code-Ansicht gelangt. Dadurch wird jeweils nur der Code angezeigt, der für diese Komponente relevant ist.

Ist man mit dem Konzept der Entwicklungsumgebung nicht vertraut, so ist es einem Anwender nicht ohne weiteres möglich ein einfaches „Hello World“ in einem Fenster auszugeben, da die Entwicklung komplett auf die grafischen Komponenten ausgerichtet sind und die Eingabe von wenigen Codezeilen ohne den Einsatz von Fenstertechniken gar nicht vorgesehen ist. Dafür stehen dem Entwickler aber ca. 1 GB an Dokumentation zur Verfügung, in der alle Funktionen von Windows ausführlich mit Beispielen erklärt werden. Diese Fülle an Informationen ist bei keinem anderen Produkt verfügbar. Allerdings sind die meisten Informationen auch ohne Visual Studio über spezielle Webseiten von Microsoft zu erhalten. Außerdem kann die Beschreibung aller Windowsfunktionen unabhängig von Visual Studio erworben und als eigenständiges Programm auf dem lokalen Rechner gestartet werden, um auf diese Weise alle Quellen schnell zu durchsuchen.

Microsoft selbst spricht von zahlreichen neuen Merkmalen seines neuen Produktes, welche die Benutzungsunterstützung (auch im Team) bei der Produkterstellung unterstützen, wobei einige nur in der größten Programmversion zur Verfügung stehen [Mic06a]. Einige der aufgeführten Merkmale sind in Tabelle 4-1 zusammengefasst:

Tabelle 4-1: Merkmale der Benutzungsunterstützung bei Visual Studio 2005

- Automatische und manuelle Konfliktlösung
- Atomares Einchecken
- Bessere Bearbeitung von ASP.NET-Quellcode
- Eincheckrichtlinien
- Assistent bei Ausnahmefehlern
- Sperren von Dateien
- Zugriffssteuerung auf Dateiebene
- Suche nach ausgeblendetem Text
- HTML-Formatierungsoptionen

- Versionskontrolle
- Kommando-Dienstprogramm zum Testen
- DataTips
- Anzeigeattribute für den Debugger
- Verteiltes Testen
- Unterstützen des Bearbeitens und Fortsetzens während des Debuggens
- Debuggen nur für eigenen Code
- Lokalisierung
- XHTML-kompatibler Designer
- Health Monitoring und Nachverfolgung
- Personalisierung
- SmartTags für Steuerelemente
- Quellcodeverwaltungs-Explorer

Die Unterstützung mehrerer Programmiersprachen hat einerseits den Vorteil, dass man beim Wechsel der Sprache mit der gleichen Entwicklungsumgebung weiterarbeiten kann. Andererseits sind, wenn man immer nur eine Sprache benutzt, unnötige Komponenten vorhanden, bzw. diese durch die Universalität nicht optimal an die aktuelle Anforderung angepasst. Deshalb hat sich Microsoft in der neuesten Version von Visual Studio (VS 2005) wieder dazu entschlossen so genannte „Express Editions“ anzubieten, die jeweils nur eine Sprache unterstützen. In den Pressemitteilungen von Microsoft werden diese als „leistungsfähige, einfach zu verwendende und leicht erlernbare Tools für Einsteiger und Hobbyprogrammierer“ bezeichnet, für alle „die eine schlanke, schnelle Entwicklungsumgebung für die Web- oder Windows-Programmierung suchen“ [Mic06b]. Somit wird hier wieder der Weg weg von der Generalisierung hin zur Spezialisierung durchgeführt.

Microsoft hat es oft geschafft, mit seiner Marktstellung neue Trends aufzuzeigen, die dann von anderen Herstellern oder Open Source-Produkten übernommen werden. Dabei wird beobachtet werden müssen, ob das Konzept von Eclipse mit der freien Wahl der einzusetzenden Plugins oder die vom Hersteller vorgegebene Toolzusammenstellung zukunftsweisend sein wird.

Delphi

Borland bietet schon seit Jahren für jede Sprache eine eigene IDE an, wobei auch hier im Hintergrund jeweils gleiche Komponenten genutzt werden. Mit der Unterstützung der selbst entwickelten Sprache Delphi, die ein Nachfolger von Pascal ist, geht das Unternehmen einen Sonderweg. Dabei wurde die Syntax von Pascal beibehalten, zusätzlich aber alle notwendigen Komponenten und Konzepte integriert, um moderne Programme und Services unter Windows schreiben zu können. Durch die Beibehaltung der Pascal-Syntax kommen viele „ältere“ Programmierer sehr schnell mit dem System zurecht. Aus diesem Grund hat sich ein größerer Markt von kommerziellen Produkten entwickelt, die mit Delphi entwickelt wurden.

Die Benutzungsunterstützung bei Borland-Produkten unterscheidet sich nicht wesentlich von Microsoft-Produkten. Auch hier steht die Oberfläche und somit das „visuelle Programmieren“ mit dem Hineinziehen von Elementen in die Oberfläche im Vordergrund.

Da zum jetzigen Zeitpunkt Programme für Parallelrechner im Allgemeinen keine oder nur eine sehr eingeschränkte Benutzerinteraktivität besitzen, spielt hier die Entwicklung einer grafischen Benutzungsoberfläche eine untergeordnete Rolle. Dagegen wird der größte Wert auf möglichst effiziente Programme gelegt und dafür unterschiedliche Benutzungsunterstützungen angeboten.

4.3 Entwicklungsumgebungen für parallele Programmierung

Sucht man Entwicklungsumgebungen für Parallelrechner, so rechnet man alleine aufgrund der kleineren Nachfrage mit einem relativ geringen Angebot. Allerdings haben bereits Mattson, Sanders und Massingill in ihrem Buch „Patterns for Parallel Programming“ weit über 200 parallele Entwicklungsumgebungen zusammengetragen [MSM04]. Aber auch diese Liste ist nicht vollständig und kann durch Recherchen in entsprechenden Suchmaschinen leicht ergänzt werden.

Betrachtet man die Produkte etwas genauer, so erkennt man schnell, dass viele der genannten „Entwicklungsumgebungen“ parallele Programmiersprachen und Bibliotheken sind. Dadurch wird deutlich, dass speziell für Parallelrechner nicht nur das GUI-System als Entwicklungsumgebung angesehen werden kann, da in vielen Fällen die „Grundsprachen“ beispielsweise „nur“ mit Bibliotheken erweitert werden, um die speziellen Fähigkeiten bestimmter Rechner ausnutzen zu können. Diese Bibliotheken werden im parallelen Umfeld ebenfalls „Entwicklungsumgebungen“ genannt.

4.3.1 Übersicht

Man hat schnell erkannt, dass Parallelrechner nur verkauft werden können, wenn geeignete grafische Entwicklungsumgebungen zur Verfügung stehen. Deshalb werden von allen Parallelrechner-Herstellern entsprechende Tools zur Verfügung gestellt. Dritthersteller von Entwicklungsumgebungen haben dabei häufig das Problem, gegen den Hersteller des Parallelrechners auf dem Markt bestehen zu müssen. Dieser liefert seine Entwicklungsumgebung meist „kostenlos“ mit dem Parallelrechner mit und besitzt zusätzlich noch wesentlich mehr interne Kenntnisse. Aus diesem Grund ist der Markt an Entwicklungsumgebungen im engeren Sinne für proprietäre Parallelrechner häufig sehr klein.

Ähnlich sieht es bei den parallelen Programmiersprachen aus. Jeder Typ von Parallelrechner hat andere Eigenschaften, welche zur Beschleunigung des zu entwickelnden Programms beitragen. Deshalb werden fast immer unterschiedliche Spracherweiterungen zu bereits unter seriellen Rechnern laufenden Programmiersprachen angeboten. Zu diesen Grundsprachen gehören in erster Linie C und Fortran. Aber auch Java, Modula oder Lisp wurden um parallele Varianten und Bibliotheken erweitert, so dass Sprachen

wie C*, C**, Fortran 90, Fortran-M, JPVM [Fer97], javaPG, JAVAR, Modula-2* [Hae92], Modula-P oder *Lisp entstanden sind [MSM04, Bra93].

Zusätzlich entstanden eigenständige Sprachen, die sich nur an vorhandene Sprachen anlehnen. Hier sind beispielsweise Occam, Linda, Parallaxis oder MPL zu nennen [MSM04, Bra90].

Lediglich im Bereich offener Standards, in denen Parallelrechner durch geeignete Kommunikationsmethoden aus „einfachen Standardrechnern“ zusammengesetzt werden, hat sich ein großer Markt an grafischen Entwicklungsumgebungen gebildet, die meist auf den parallelen Spracherweiterungen MPI und PVM aufsetzen (siehe unten). Die Standardkomponenten sind den Softwareherstellern bekannt, so dass sie darauf aufbauend Konzepte entwickeln können, wie der Parallelrechner als „Cluster“ oder „Grid“ aufgebaut werden kann und wie diese Parallelisierung durch geeignete Tool unterstützt werden kann.

Klassifiziert man die Rechnertypen, so sind für MIMD-Systeme die meisten Entwicklungsumgebungen vorhanden. Dabei wird besonders das „Message-Passing“ unterstützt, welches auf verteilten, heterogenen und lose-gekoppelten Computersystemen ein paralleles Arbeiten ermöglicht. Daneben werden häufig noch Umgebungen betrachtet, die sich mit Shared-Memory-Systemen beschäftigen, die in letzter Zeit unter anderem durch Anstrengungen der führenden Prozessorhersteller wieder sehr stark an Bedeutung gewinnen. Die bekanntesten Vertreter für die Unterstützung von „Message-Passing“ sind die Erweiterungsbibliotheken PVM (Parallel Virtual Machine) [GBDJ+94] und MPI (Message-Passing Interface) [MPIF94], welche Methoden zur Verfügung stellen, um Daten zwischen den Rechnern auszutauschen. Beide Bibliotheken existieren mittlerweile für alle gängigen Programmiersprachen in unterschiedlichen Implementierungen, wobei die Realisierung der Kommunikation unterschiedlich durchgeführt wurde. So konnte im Rahmen der Untersuchungen festgestellt werden, dass je nach gewählter Bibliothek oder zu Grunde gelegtem Betriebssystem die Arbeitsgeschwindigkeit differiert, obwohl die Bibliotheken jeweils mit gleichem Quellcode angesprochen werden [Ric99].

Alleine durch den Austausch von bekannten Bibliotheken für Einprozessorsysteme durch Systeme, die PVM oder MPI unterstützen, können komplette Anwendungen zumindest teilweise parallelisiert werden. Ein Beispiel dafür sind die NAG Bibliotheken [McD95]. Weitere frei verfügbare Bibliotheken sind unter anderem p4 [BuL94], Nexus, oder Madeleine [HaR04]. Sie konnten sich aber nicht in der Breite durchsetzen. Auch für Distributed Memory Systeme stehen eine Reihe an Tools zur Verfügung, die aber in der Regel ohne grafische Plattformen angeboten werden [PaC04, RKN04].

Um Entwicklungsumgebungen zu beurteilen, welche grafische Elemente einsetzen, muss man sich im Klaren sein, welche Phasen der Programmentwicklung diese unterstützen sollen. Dabei zeigt sich schnell, dass sich viele grafische Entwicklungsumgebungen auf bestimmte Phasen spezialisiert haben. Deshalb werden im Folgenden kurz die drei wichtigsten Phasen vorgestellt.

4.3.1.1 Phasen der Programmentwicklung

Da Parallelrechner sehr teuer in der Anschaffung und im Unterhalt sind, werden sie häufig im „Batchmodus“ betrieben. Das bedeutet, die Rechenaufgaben werden in eine Warteschlange gelegt und nacheinander abgearbeitet. Somit kann man eine Rechnerauslastung „rund um die Uhr“ erreichen, verliert aber an Flexibilität und Interaktionsmöglichkeit. Da durch den Wegfall der Interaktivität nur ein geringer Bedarf an grafischen und somit benutzungsfreundlichen Bedienoberflächen besteht, wird das Hauptunterstützungsfeld der modernen Entwicklungsumgebungen für Einprozessorsysteme nur wenig benötigt. Somit entscheiden andere Kriterien bei der Auswahl von grafischen Entwicklungsumgebungen für Parallelrechner. Sie lassen sind darin vergleichen, welche Phase der Programmentwicklung sie besonders unterstützen.

Die drei wichtigsten Phasen sind:

- Programmentwicklung: Für die Entwicklung von parallelen Programmen werden häufig Tools angeboten, welche die logische Kommunikation oder die physikalische Verbindung zwischen Prozessoren darstellen. Besonders häufig sind diese Tools bei Message-Passing Umgebungen zu finden, wobei hier zum einen die Kommunikation per Drag&Drop zusammengestellt werden kann oder die Konfiguration von hybriden Systemen unterstützt wird.
  Die in dieser Arbeit entwickelte Programmierumgebung zeigt hier neue Ansätze, da für massivparallele Systeme in diesem Bereich nur sehr eingeschränkt grafische Tools zur Verfügung stehen.
- Fehlersuche: Das Finden von Fehlern ist bei parallelen Programmen wesentlich aufwändiger als bei sequentiellen Programmen. Dies liegt daran, dass durch den Protokollierungsvorgang in das eigentliche Programm eingegriffen werden muss und dadurch häufig Änderungen in der Abfolge von Kommunikationen stattfinden, so dass Fehler, die durch eine falsche Kommunikationsreihenfolge entstehen können, schwer zu entdecken sind. Die verfügbaren Tools unterscheiden sich in einem reinen Monitoring, welche „nur“ die Daten der einzelnen Prozessoren speichern und diese später über Visualisierungstools anzeigen und Debuggern, die es ermöglichen, parallele Programme Schritt für Schritt ablaufen zu lassen.
- Geschwindigkeitsverbesserung: Ein Programm schneller auszuführen oder mit einem bestehenden Programm ein größeres Problem lösen zu können, ist der Hauptgrund für den Einsatz von Parallelrechnern. Aus diesem Grund ist es sehr wichtig, dass Tools zur Verfügung stehen, welche den Programmierer bei der Optimierung seiner Programme unterstützen.

4.3.1.2 Grafische Entwicklungssysteme für Message-Passing Systeme

Im Folgenden werden einige grafische Entwicklungssysteme mit ihren jeweiligen Besonderheiten aufgezählt, wobei der Schwerpunkt auf die Unterstützung von Rechnersystemen gelegt wird, welche entweder innerhalb eines schnellen Spezialnetzes arbeiten oder zumindest auf ein schnelles LAN zugreifen können.

CODE

“An ideal system would let you program graphically, specify component in any language, and run it on any architecture. Such a prototype is already in use.” Mit diesen Sätzen stellen Browne, Azam und Sobek ihren Artikel über die Entwicklungsumgebung CODE vor [BAS89]. Umgesetzt wird diese Vision durch eine Visualisierung mit Hilfe von Abhängigkeitsgrafen, welche in einer abstrakten Sprache gespeichert wird und anschließend automatisch in die jeweilige hardwareabhängige Programmiersprache umgesetzt wird. Diese ersten Ansätze der visuellen Programmierung konnten generelle Ansätze der Parallelisierung bei Shared-Memory-Systemen realisieren. CODE 2.0 wurde erweitert, um auch Distributed-Memory-Systeme zu unterstützen [NeB92]. Der Programmansatz unterstützt in erster Linie die Programmentwicklung, wobei einerseits durch die Abstraktion mehrere Zielsysteme unterstützt werden, anderseits wird die Geschwindigkeitsoptimierung durch plattformspezifische Codefragmente realisiert.

TRAPPER

TRAPPER ist eine der bekanntesten deutschen Entwicklungsumgebungen, die auf grafischem Wege heterogene Programmentwicklung unterstützen. TRAPPER steht dabei für „TRAffonic Parallel Programming EnviRonment“, wobei Traffonic ein Begriff der Daimler-Benz Forschungsabteilung für Elektronik in Verkehrssystemen (electrONIC in TRAFFic systems) ist [SSK93, SSK94]. Die Entwicklungsumgebung ist sehr mächtig und unterstützt alle oben genannten Entwicklungsphasen mit grafischen Tools. Abbildung 4-2 zeigt einige Screenshots der Design-Tools.

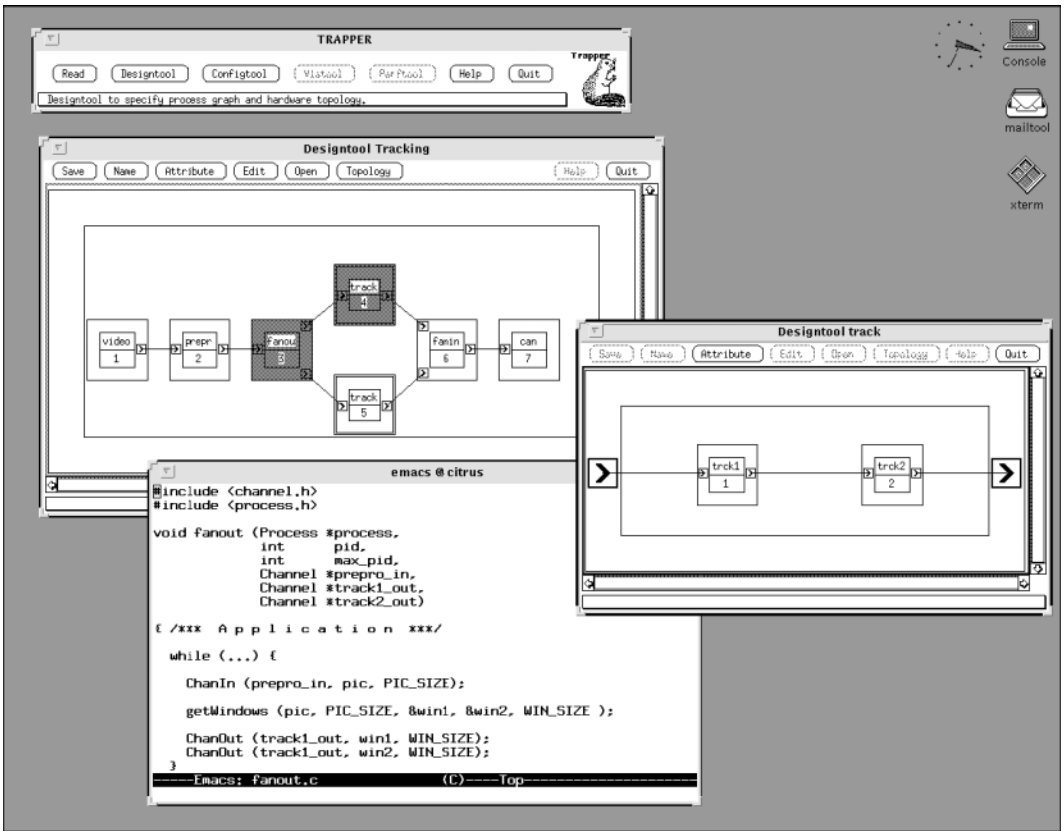


Abbildung 4-2: Screenshots von TRAPPER (aus [SSK93])

Daneben zeigt das Performancetool beispielsweise Informationen über den „kritischen Pfad“ eines parallelen Programms auf. Erste Untersuchungen, dass die „kritische Pfad Analyse“ signifikant für die Performancegewinnung von parallelen Programmen ist, wurden von Yang und Miller vorgenommen und unter anderem von Hollingsworth bestätigt [YaM88, Hol98]. TRAPPER nutzt als Grundlage die Grafikbibliothek InterViews und unterstützt sowohl PVM als auch MPI für die Zielsprachen ANSI C und Occam [SSK95].

GRIX

GRIX ist ein neuerer Vertreter grafischer Entwicklungsumgebungen. Es unterstützt SPMD (Single Programm Multiple Data) und benutzt als zugrunde liegende Sprache Java. Für die Kommunikation wird eine Java-Implementierung von MPI oder PVM genutzt. Ziel der Entwicklungsumgebung ist es, die bildlichen Vorstellungen des Benutzers über den zu implementierenden Algorithmus und insbesondere über die Verteilung der Anwendung auf die Prozessoren mit Hilfe von Methoden der visuellen Programmierung umzusetzen [SST02]. Dazu kann der Benutzer in einem grafischen Dialogfenster die einzelnen Prozessoren modellieren und per Drag&Drop Verbindungen einzeichnen. Abbildung 4-3 zeigt ein einfaches Beispiel dazu. Zu dem Prozessor und jeder Verbindung können notwendige Parameter angegeben werden, so dass der für die Ausführung benötigte Quelltext anschließend automatisch generiert werden kann.

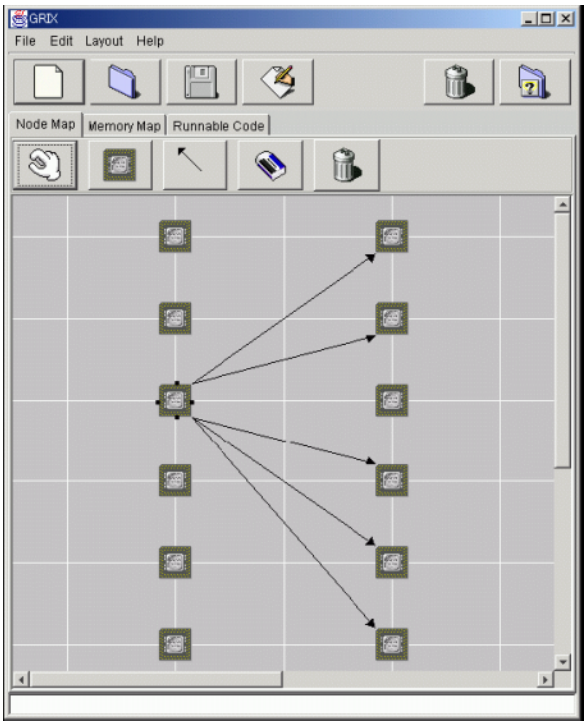


Abbildung 4-3: Screenshot GRIX (aus [SST02])

Leider besteht keine Möglichkeit innerhalb der grafischen Abbildungen den dazugehörigen Quelltext zu sehen. Dazu muss ein separater Editor geöffnet werden.

VISPER

VISPER geht über den reinen Ansatz einer Entwicklungsumgebung hinaus und bietet ein „Distributed Parallel Programming Framework“ mit einem objektorientierten Ansatz. Ziel ist es, nicht direkt die Kommunikationskanäle anzusprechen, sondern mittels visueller Programmierung einen Zwischenlayer zu integrieren, der flexibel, je nach zugrunde liegender Umgebung in die jeweils geeignete Kommunikationsstruktur automatisiert übersetzt werden kann [StZ02]. VISPER unterstützt visuell alle Phasen der Programmentwicklung, indem Design-, Konfigurations-, Debug- und Performance-Tools zur Verfügung gestellt werden.

Die von Stankovic und Zhang beschriebene Umgebung erzeugt Java-Code. Die MPI-Befehle unter Java nutzen die C++-Bibliotheken der MPI-Implementierung.

Weitere:

Es existieren noch viele weitere Tools, die alle eine grafische Unterstützung anbieten. Allerdings zeigen sie keine für die Arbeit relevanten Unterschiede zu den bereits vorgestellten Entwicklungsumgebungen auf und werden deshalb hier nur noch aufgezählt:

ParaGraph ist ein viel zitiertes Tool, welches die Visualisierung der Parallelisierung veranschaulicht [HeE91].

HeNCE ist ein Produkt von Beguelin, Dongarra und anderen, welches von vielen weiteren Produkten als Referenz genutzt wurde [BDGM+91]. Zur Visualisierung werden

Abhängigkeitsgrafen eingesetzt. Für die Performancemessung stehen Balkendiagramme zur Verfügung.

PARSE und PARSE-DAT verfolgen ähnliche Ansätze wie HeNCE [GJG93, LiG98].

Paralex ist ebenfalls eine Umgebung, welche durch Abhängigkeitsgrafen die Entwicklung paralleler Programme vereinfacht [BAAD+92].

Xab ist eines der ersten grafischen Tools für Parallelrechner, welche das Monitoring von PVM-Programmen unterstützt [Beg93]. Damit verfolgt es einen etwas anderen Ansatz, da es das eigentlich zu entwickelnde Programm „von außen“ betrachtet und dem Programmierer zusätzliche Informationen des Datenaustauschs zur Verfügung stellt.

Visputer unterstützt die Programmierung unter Occam. Dabei wird ein grafisches Entwicklungstool zur Verfügung gestellt, welches die Verbindungen zwischen den Prozessoren anzeigt. Zusätzlich wird ein Analysetool angeboten, um die zeitliche Visualisierung von parallelen Threads zu ermöglichen [MaZ94].

Annai ist eine Sammlung an grafischen Tools, welche schwerpunktmäßig das Debugging und die Geschwindigkeitssteigerung von High Performance Fortran (HPF)-Programmen unter Nutzung von MPI unterstützt [CEF96]. Dabei werden verschiedene grafische Tools angeboten, die unter anderem durch 3D-Grafen, Ansichten zur globalen Datenverteilung, Prozessorkommunikationen oder zeitabhängigen Speicherbelegungen den Programmierer bei der Programmentwicklung unterstützen.

Die Hauptaufgabe von Parsec ist die Optimierung von Message-Passing Programmen [FeW96]. Dabei wird die Konfiguration der Kommunikation abhängig vom jeweiligen Kontext erzeugt. Als grafisches Tool steht unter anderem die jeweilige Zusammenstellung der Prozessorfarm mit deren Aufgabeneinteilung zur Verfügung.

PATOP unterstützt ebenfalls hautsächlich den Bereich der Geschwindigkeitsoptimierung. Dabei wird mittels Matrizen die Speicherbelegung und Prozessorauslastung einer großen Anzahl an Rechner angezeigt, um auf diese Weise die „Bottlenecks“ einer Anwendung finden zu können.

Regis unterstützt das Softwaredesign mit Assistenten, automatisch erzeugtem Code und einer Entwicklungsumgebung, welche den Code nebeneinander grafisch und in Textform darstellen kann [NKMD96].

Einen etwas anderen Ansatz verfolgen Labarta u.a., die in der Programmierumgebung DiP zwei Tools vorstellen, mit denen Parallelrechner simuliert und visualisiert werden sollen: DIMEMAS und PARAVER [LGPC+96]. Ziel ist es Message-Passing Programme bereits vor dem Einsatz auf einem Parallelrechner soweit zu optimieren, dass die meist ausgelasteten Rechner möglichst wenig zu Testzwecken genutzt werden müssen.

In der folgenden Tabelle 4-2 werden die vorgestellten grafischen Entwicklungsumgebungen für Message-Passing Systeme zusammengefasst und dabei die wichtigsten Eigenschaften wiederholt.

Tabelle 4-2: Zusammenfassung grafischer Entwicklungsumgebungen für Message-Passing Systeme

	Sprachen	Typ	Programm-entwicklung	Fehlersuche	Geschwin-digkeitsver-besserung	Sonstiges
CODE	Me-tasprache	Shared-Memory, (Distributed Memory)	X		X	Abhängigkeitsgrafen
TRAP-PER	C, Occam	PVM, MPI	X	X	X	Unterstützt alle Bereiche der Programmentwick-lung
GRIX	Java	SPMD, PVM, MPI	X			Generierung des Quelltext durch Grafiken
VISPER	visuelle Zwischen-sprache, Java	MPI	X	X	X	Sprachenunabhängige Speicherung der Kommunikation
Para-Graph	C	Message-Passing			X	Einsatz graphischer Animationen
HeNCE	C, Fortran	PVM	X		X	Abhängigkeitsgrafen
PARSE	sprachen-unabhängig	Message-Passing, Shared Memory	X			Abhängigkeitsgrafen, Petri-Netze
Paralex	C	Distributed Memory, NFS-Datenaustausch	X	X		Abhängigkeitsgrafen, Schwerpunkt: Load Balancing
Xab	sprachen-unabhängig	PVM		X	X	Monitoringprogramm
Visputer	Occam	Message-Passing	X	X		
Annai	HPFortran	MPI		X	X	
Parsec		Message-Passing			X	
PATOP					X	
Regis			X			
DiP		Message-Passing				Simulation von Parallel-rechnern

4.3.1.3 Entwicklungsumgebungen für Grid-Computing

Grid-Computing ist eine Methode, um eine große Anzahl von heterogenen Rechnersys-temen über ein virtuelles Netzwerk miteinander zu verbinden und somit insgesamt eine

möglichst hohe Rechenleistung erzielen zu können. Durch die große Heterogenität der Recheneinheiten und die langsame Verbindung zwischen diesen werden Grids so eingesetzt, dass möglichst wenig Kommunikation durchgeführt werden muss. Durch diesen „anderen“ Ansatz liegt der Schwerpunkt der grafischen Entwicklungsumgebungen häufig nicht bei der Unterstützung der parallelen Anwendung, sondern bei der Verwaltung des „Gesamtrechners“. Li und Baker haben die Entwicklung des Grid-Computing in drei Generationen eingeteilt [LiB06].

In der ersten Phase wurden grundlegende Strukturen aufgebaut, um Authentifizierung, Autorisierung, Jobmanagement, Datentransfer und einen Informationsservice gewährleisten zu können. Dazu wurden als Oberflächen HTML-Seiten eingesetzt, welche durch JavaScript dynamisch angepasst werden konnten. Tools, welche diese Tätigkeiten unterstützen, sind beispielsweise GridPort, Grid Portal Development Kit, Ninf Portal oder XCAT. Wurden Grids mit diesen proprietären Tools aufgebaut, so konnten sie nicht ohne weiteres um neue Techniken erweitert werden.

Seit der zweiten Generation setzt man auf allgemeine Webportale, welche auf einfache Weise durch Portlets erweitert werden können. Als Portale werden unter anderem genannt: Microsoft SharePoint Server, IBM WebSphere Portal Server, Jetspeed oder GridSphere. Die Portlets erfüllen von internationalen Organisationen vorgegebene Standards, so dass sie einfach von einem Portal auf das nächste umgezogen werden können. Der bekannteste Standard für Portlets ist JSR168. Bei JSRs (Java Specification Request) handelt es sich um Spracherweiterungen der Programmiersprache Java, welche von einem Standardisierungskomitee durch den „Java Community Process“ (JCP) entwickelt werden. Aktuell existieren bereits über 300 standardisierte Spracherweiterungen von Java.

Die dritte Generation schließlich verfolgt einen serviceorientierten Ansatz, indem einzelne Knoten semantische Kennzeichen erhalten. Durch den Einsatz von Webservices, SOAP, WSDL oder UDDI sollen Eigenschaften wie Service Level Agreements, Sicherheit, Speichermanagement oder Kosten einzelner Grid-Teile berücksichtigt werden können. Als Beispielt für ein Portal der dritten Generation wird PortalLab genannt [LSWR+03].

Für Grid-Systeme wurden mittlerweile zahlreiche Design-Patterns entwickelt, welche beispielsweise vorgeben, wie man am besten einen Dienst von einem Portal auf ein Neues umziehen kann.

Da die Schwerpunkte der Tools für das Grid Computing sehr weit entfernt sind von den Anwendungen, die in dieser Arbeit erarbeitet werden sollen, werden sie hier nicht weiter betrachtet. Nichts desto trotz können die Ergebnisse dieser Arbeit auch in Umgebungen für das Grid Computing einfließen.

Im Folgenden werden noch zwei herstellereigene Entwicklungsumgebungen vorgestellt, die als Vergleich für die in dieser Arbeit entwickelten Erweiterungen einer grafischen Benutzungsunterstützung dienen sollen. Sie unterstützen keine Message-Passing Systeme, sondern Ansätze, die an die jeweiligen SIMD-Rechner angepasst sind.

4.3.2 MPPE

MPPE ist die “MasPar Programming Environment”, die speziell für Entwicklung von MasPar-Programmen zur Verfügung gestellt wird. Die Entwicklungsumgebung enthält eine Reihe von Tools und interaktiven Programmen, welche den Programmierer besonders bei den Phasen Fehlersuche und Geschwindigkeitsverbesserung unterstützen. Die eigentliche Programmentwicklung wird dagegen nicht mit geeigneten Tools vereinfacht.

Alle Tools verstehen die beiden Sprachen MasPar Fortran und MasPar Programming Language (MPL). Einzelheiten zu MPL wurden bereits in Kapitel 3.3 vorgestellt.

Aus visueller Sicht ist das interessanteste Tool ein Visualisierungsfenster, welches während eines Debugvorgangs den Zustand aller Prozessorelemente darstellt. Dabei wird in einem Fenster jedes Prozessorelement durch ein Pixel repräsentiert, so dass alle 16.384 Prozessorelemente auf einer Fläche von 128x128 Pixel dargestellt werden können. Aus Gründen der Übersicht ist allerdings zwischen allen Prozessorelementen noch ein „leeres“ Pixel vorgesehen.

Standardmäßig wird die Aktivierung der Prozessorelemente visualisiert. Dabei kann man Muster erkennen, die im Laufe dieser Arbeit dazu genutzt werden, um „Standardaktivierungen“ „mit einem Klick“ programmieren zu können (siehe Kapitel 5.3.1). Durch einen Parameter können aber statt der Aktivierung auch Eigenschaften wie „Prozessorelement empfängt Daten“, „Prozessorelement versendet Daten“ oder bestimmte Speicherzustände angezeigt werden.

Großer Wert wird außerdem auf einen Debugger gelegt, in welchem für den parallelen Code Breakpoints gesetzt werden können. Die Ausgabe von Zwischenergebnissen ist ebenfalls möglich, wobei die Ergebnisse der einzelnen Prozessorelemente in einer eindimensionalen Tabelle angezeigt werden. Alternativ kann auch hier das Visualisierungsfenster zur Verfügung gestellt werden. In diesem Fall können allerdings nur Pixel zurückgegeben werden, welche TRUE oder FALSE für eine definierte Bool’sche Operation repräsentieren. Ein Beispiel dafür zeigt Abbildung 4-4.

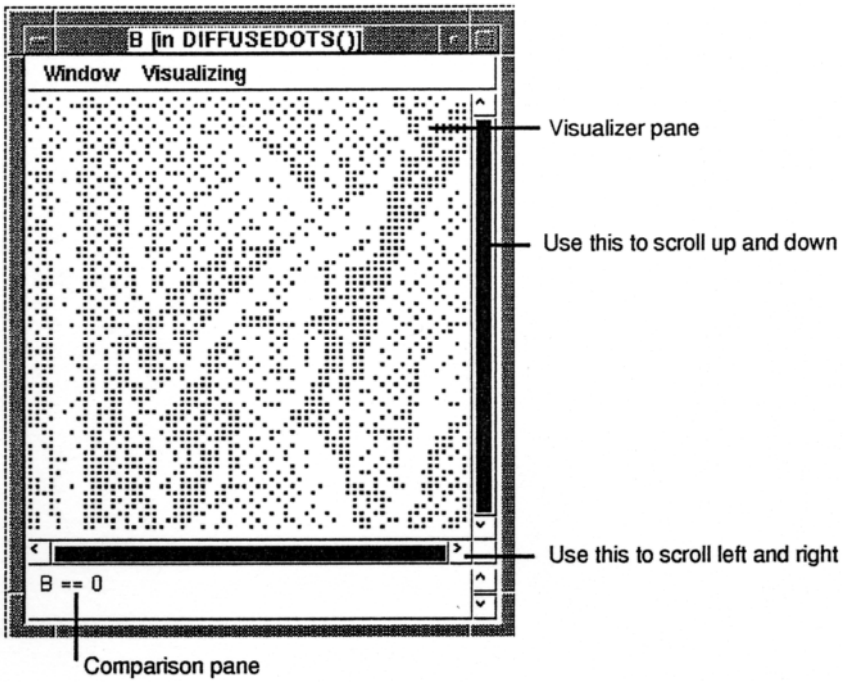


Abbildung 4-4: Datenvisualisierung von MPPE (aus [Mas92a])

Der Profiler zeigt, wie lange die Ausführung jeder Programmzeile gedauert hat. Dabei wird hinter jeder Quelltextzeile ein Balken angezeigt, welcher die jeweilige Ausführungszeit repräsentiert.

Die Beispiele dieser visuellen Tools zeigen, dass die Fehlersuche und Geschwindigkeitssteigerung im Vordergrund der Unterstützung steht. Leider wird der Anwender bei den ersten Phasen der Programmentwicklung überhaupt nicht unterstützt.

4.3.3 ISATOOLS

Die ISATOOLS stellen ein Programmpaket zur Verfügung, mit denen für das in Kapitel 3.4 vorgestellte befehlssystolische Array von ISATEC Programme entwickelt werden können. Im Einzelnen sind dies ein Editor für „Controller-Programme“ und ein „ISA-Programm-Editor“ [ISA98].

Anders als fast alle anderen Parallelrechner wird die ISATEC-Karte nicht unter UNIX oder Linux angesprochen, sondern unter Windows bzw. DOS. Deshalb stehen die Entwicklungstools nur unter Windows bzw. DOS zur Verfügung. Unterstützt werden die Sprachen Pascal und C++ von Borland. Diese Einschränkung auf zwei Sprachen und Windows/DOS hängt nicht mit dem Konzept des Parallelrechners zusammen, sondern mit den eingeschränkten Entwicklungskapazitäten und den Marktgegebenheiten, die zum Zeitpunkt der Entwicklung vorhanden waren. Die ISATOOLS entstanden in der ersten Hälfte der 1990er Jahre. Zu diesem Zeitpunkt spielte Linux noch eine untergeordnete Rolle auf PCs. Da bereits eine Bibliothek unter C++ besteht, sollte diese relativ leicht auch auf ein Linux-System konvertierbar sein, auch wenn die direkte Ansprache von PCI-Geräten unter Windows und Linux unterschiedlich durchgeführt wird.

Betrachtet man das in Kapitel 3.2 vorgestellte Programmiermodell für einen massivparallelen Rechner, so werden der erste und letzte Schritt des Modells, „Initialisierung und Rücktransfer“, nicht mit eigenen grafischen Programmen unterstützt. Dies hängt damit zusammen, dass beide Schritte besser in den „Wirtssystemen“ erledigt werden können. Dazu wird von der Firma ISATEC eine Bibliothek zur Verfügung gestellt, welche die notwendigen Befehle zur Verfügung stellt.

So können beispielsweise durch die Befehlszeilen

```
isa_board_reset
load_cfg(cfg_file)
load_block (WEST, 0, 1024, a)
run_iq (’START’, 1)
store_block (WEST, 0, 1024, a)
isa_board_down
```

der Parallelrechner von der Wirtssprache C oder Pascal gesteuert werden.

Im Einzelnen wird durch `isa_board_reset` das ISA-Board aktiviert, anschließend werden die benötigten Programme durch `load_cfg` in den Hauptspeicher geladen und direkt zum Parallelrechner transferiert. `load_block` kopiert alle benötigten Daten auf die Parallelrechnerkarte, wo durch `run_iq` das ausgewählte Programm gestartet wird. Nachdem die ISA-Karte mit ihren Berechnungen fertig ist, werden die Daten wieder mit `store_block` zurückgeschrieben und die Karte mit `isa_board_down` in einen Stromsparmodus versetzt.

Programme, die auf dem Parallelrechner ablaufen und durch `run_iq` initiiert werden, werden mit den ISATOOLS entwickelt.

Auf der ISATEC-Karte sind zwei verschiedene Arten von Programmen nötig, um einen Algorithmus ablaufen zu lassen. Zum einen sind das die „Controller-Programme“, welche die Daten von einem zentralen Bereich, an den die Daten vom Wirtssystem übertragen wurden, zu der „richtigen Stelle“ auf dem Prozessorfeld transferieren. Dies sind die westlichen und nördlichen Randprozessoren, von denen die Daten anschließend in das Prozessorfeld abgegeben werden. Zum anderen startet das „Controller-Programm“ den zweiten Programmtyp, das „ISA-Programm“, welches für die Kontrolle des Prozessorfeldes zuständig ist und dessen Editor weiter unten besprochen wird.

Der Editor für „Controller-Programme“ hat in seinem Hauptfenster eine Tabellenstruktur (siehe Abbildung 4-5)

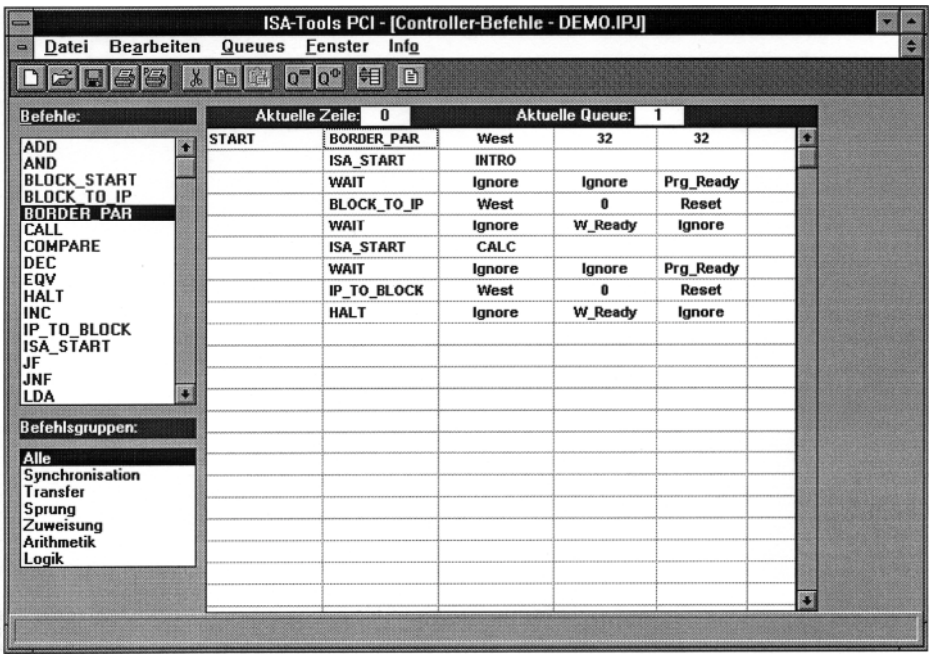


Abbildung 4-5: Fenster zur Erstellung von Controller-Programmen (aus [ISA98])

Diese Struktur unterstützt die Programmierung der Controller-Programme, deren Befehle an einen Pseudocode einer Assemblersprache erinnern. Die erste Spalte enthält Sprungmarken, die zweite Spalte den eigentlichen Befehl. Die weiteren Spalten sind für die Befehlsparameter reserviert. Das in Abbildung 4-5 vorgestellte Programm zeigt die Befehlsfolge für den Transfer der Daten zu den westlichen Randprozessoren, zum Start des ISA-Prozessorfeldes und dem Rücktransfer der Daten von den westlichen Randprozessoren.

Für die Sprungmarken können beliebige Namen vergeben werden, da allerdings eine Eindeutigkeit notwendig ist, können keine doppelten Begriffe vorkommen. Dies ist besonders bei Kopieraktionen zu beachten, bei denen der Editor mit einer entsprechenden Fehlermeldung darauf aufmerksam macht.

Die Eingabe der Controller-Befehle in der zweiten Spalte wird durch eine Auswahlliste erleichtert, die auf der linken Seite des Editors angebracht wurde. Im oberen Feld kann der gewünschte Befehl durch einen Doppelklick ausgewählt werden. Damit die Auswahl der Befehle nicht unübersichtlich wird, steht im unteren Feld eine Kategorisierung zur Verfügung, so dass auf Wunsch nur eine Teilmenge der Controller-Befehle zur Auswahl angeboten wird.

Wechselt man in ein Feld ab der dritten Spalte, so werden auf der linken Seite die hier möglichen Parameter aufgezeigt. Werden als Parameter ISA-Programmnamen oder Variablennamen erwartet, so kann durch ein Doppelklick auf das Eingabefeld eine Dialogbox geöffnet werden, welche vorhandene ISA-Programme mit kurzer Beschreibung oder Variablen anbieten.

Alternativ können alle Eingaben auch direkt per Tastatur eingegeben werden, ohne dass die Listen oder Dialogboxen benutzt werden müssen.

Im oberen Bereich des Editors werden die üblichen Funktionen wie speichern, öffnen, kopieren oder einfügen angeboten. Beim Speichern kann durch eine Dialogbox noch gewählt werden, ob das jeweilige Programm zusätzlich in ein maschinenlesbares Format codiert werden soll.

Durch die Schaltfläche ganz rechts kann man zum ISA-Programm-Editor gelangen. Dieser Editor ist recht einfach aufgebaut und ähnelt dem einfachen ASCII-Editor „notepad“ unter Windows (siehe Abbildung 4-6).

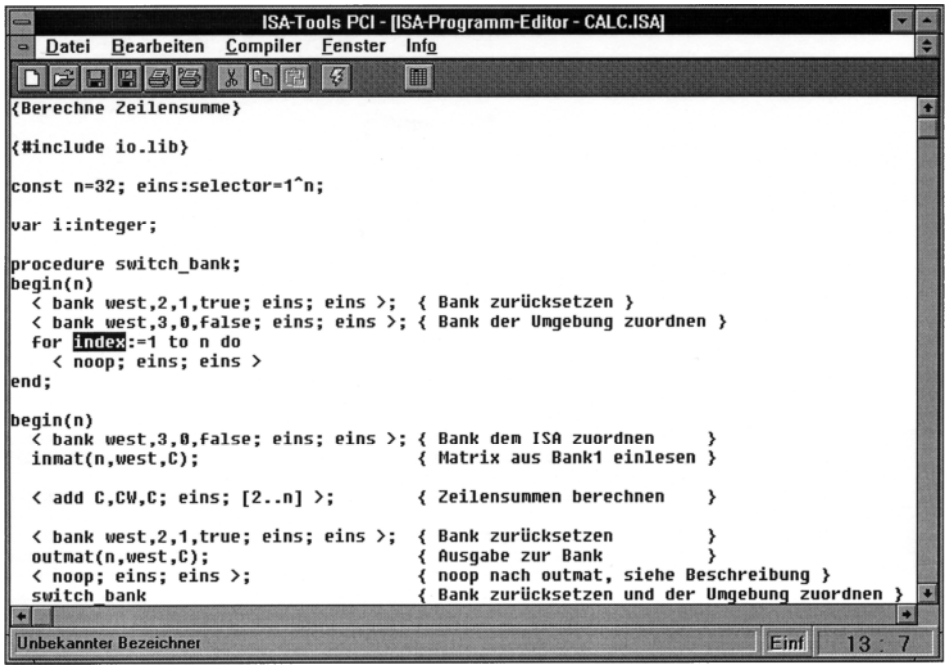


Abbildung 4-6: ISA-Programm-Editor (aus [ISA98])

Die einzige programmierrelevante Erweiterung ist die Möglichkeit, das aktuelle Programm zu compilieren. Dabei wird im Fehlerfall in der Statuszeile ein entsprechender Hinweis angezeigt und der fehlerhafte Quelltext markiert.

Weitere Unterstützung bereits bei der Erstellung der ISA-Programme wird leider nicht angeboten, allerdings kann der Entwickler in den Quellcode die Compiler-Anweisung `{#symboliccode}` integrieren, durch die die anschließende Fehlersuche unterstützt werden kann. Der Compiler erzeugt durch die Anweisung zusätzlich zur maschinenlesbaren Programmversion eine lesbare Textdatei, in der beispielsweise Schleifen aufgelöst werden und jeder Befehl mit Argumenten und Zeilen- und Spaltenselektoren angezeigt wird.

Da die Tools schon seit einiger Zeit existieren, befinden sie sich nicht auf dem aktuellen Stand der technischen Möglichkeiten. Die in Kapitel 1 vorgestellten ergonomischen Grundlagen werden weitgehend eingehalten, allerdings fehlen erweiterte Fähigkeiten der Selbstbeschreibungsfähigkeit und Erwartungskonformität. Die in der ISO 9241-10 geforderte Individualisierbarkeit ist nicht vorhanden, die Lernförderlichkeit dagegen ist beispielsweise durch die Auswahlmöglichkeit an Controllerbefehlen und der Anzeige einer Beschreibung berücksichtigt worden.

5 Hilfestellungen für den Benutzer

In der neueren Literatur wird im Zusammenhang mit den hier besprochenen Hilfestellungen nicht mehr von „Benutzerunterstützung“ gesprochen, sondern von „Benutzungsunterstützung“. Die Begründung ist, dass der Benutzer bei der Benutzung der Programme unterstützt wird. Eine Benutzerunterstützung kann viel weit reichender sein und beispielsweise auch die Ergonomie von Schreibtischen beschreiben, die für die Arbeit am Bildschirm geeignet sind [DIN98a, DIN98b]. Um die Einschränkung auf die unmittelbare Hilfestellung bei der Benutzung von Programmen zu verdeutlichen, wird deshalb im weiteren Verlauf dieser Arbeit nur noch der Begriff „Benutzungsunterstützung“ verwendet.

Der „ideale Programmierer“ kann ein Programm nach dem einfachen Wasserfallmodell entwickeln, das Winston Royce erstmals 1970 vorgeschlagen hat (siehe Abbildung 5-1).

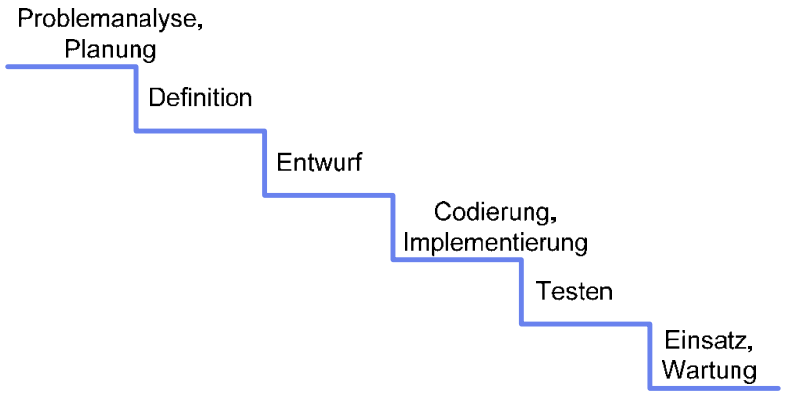


Abbildung 5-1: Wasserfallmodell

Es gelingt aber niemandem, ein etwas komplexeres Programm auf Anhieb korrekt zu entwickeln. Aus diesem Grund werden in allen gängigen Entwurfsmodellen Zyklen

eingebaut. Durch die Einbindung der Zyklen soll erreicht werden, dass vorher begangene Fehler durch die Rückkehr zu einer früheren Entwicklungsstufe wieder beseitigt werden können.

Jedes erweiterte Modell zeigt Wege auf, wie im Fall eines „Fehlers“ mit der Arbeit fort gefahren werden sollte. Teilweise muss zu früheren Entwicklungsstufen zurückgekehrt werden, teilweise muss innerhalb einer Entwicklungsstufe die Arbeit fortgesetzt werden.

Eine Schleife, die sehr häufig und in kurzen Zyklen durchgeführt wird, ist die zwischen der oben genannten vierten und fünften Stufe, der Codierung und dem Testen, da ein Programm häufig entweder aus syntaktischen oder semantischen Gründen nicht den Anforderungen entspricht. Diese Fehler treten bei ungeübten Programmierern besonders häufig auf und führen zu langen und frustrierenden Fehlersuchen, mit denen der Anfänger oft überfordert ist. Die in Kapitel 2.1 beschriebene Benutzergruppe ist im Bereich der parallelen Programmierung zu den Anfängern zu zählen, so dass auch bei diesen öfters mit Syntaxfehlern zu rechnen ist. Aber auch bei erfahrenen Programmieren und linearer Programmierung treten immer wieder Syntaxfehler auf, da durch eingesetzte Bibliotheken immer komplexere Befehle zur Verfügung stehen.

Im Vergleich zu semantischen Fehlern sind syntaktische relativ einfach zu beseitigen, trotzdem spielen sie bei der Entwicklungszeit eines Programms durch die Häufigkeit des Auftretens eine entscheidende Rolle. Deshalb ist es wichtig diese Fehlerquelle möglichst zu vermeiden, indem der Benutzer durch unterschiedliche Instrumente unterstützt wird.

Je nach Fehler erhält man von dem eingesetzten Editor bzw. der Entwicklungsumgebung keinen Hinweis darauf, wo der Fehler zu finden ist. In anderen Fällen wird zwar darauf hingewiesen, wo man den Fehler findet, aber es wird nicht gezeigt, wie man ihn beseitigen kann. Der beste Fall wäre, wenn der Entwickler überhaupt keine Fehler machen würde. Um dies zu ermöglichen, sollten ihm frühzeitig möglichst viele Hilfsmittel zur Auswahl gestellt werden.

5.1 Überblick der Möglichkeiten zur Benutzungsunterstützung

Im Folgenden wird eine Auswahl an Maßnahmen vorgestellt, die einem Programmierer bei der Codierung eines Programms helfen soll, so dass direkt ein möglichst fehlerfreier Quellcode entsteht. Dabei wird gezeigt, wie vielfältig Hilfe angeboten werden kann. Auf die Unterstützung, die direkt beim Eintippen bzw. Generieren der Befehle gegeben werden kann, wird genauer eingegangen.

5.1.1 Dialogboxen

Dialogboxen sind heute bei jedem Programm mit Bedienoberfläche anzutreffen. Normalerweise besteht ein Programm aus mindestens einem Hauptfenster. Sollen aktuelle Informationen, Warnhinweise oder Hilfen angezeigt werden, geschieht das, indem ein weiteres Fenster als Dialogbox geöffnet wird.

Die Dialogboxen dienen zur Ausführung in sich abgeschlossener Operationen, wobei sie häufig über Parameter Informationen aus der Umgebung aufnehmen und diese aufbereitet darstellen, um dem Benutzer die für die aktuelle Situation relevante Information zu präsentieren. Sie setzen sich im Allgemeinen aus mehreren User-Interface-Komponenten zusammen. Eine der genutzten Komponenten ist in der Regel ein OK-Button, der bei seiner Betätigung die in der Dialogbox eingegebenen Parameterwerte an das zugrunde liegende Programm zurückliefert, sowie ein Abbrechen-Button, bei dessen Betätigung die Eingaben ignoriert werden. In beiden Fällen wird die Dialogbox geschlossen und der Dialog beendet.

Eine Dialogbox kann „modal“ geöffnet werden, so dass keine Bedienhandlungen außerhalb des Dialogs möglich sind bis dieser geschlossen wird, oder „nichtmodal“, wenn parallel andere Aktionen ausgeführt und weitere Dialoge geöffnet werden können. Nach Schließen des Fensters gelangt man wieder in das Hauptfenster [Pre99].

Ohne Dialogboxen sind viele der weiter unten vorgestellten Hilfefunktionen für Entwickler nicht realisierbar, denn für die gezielte Benutzerunterstützung müssen zusätzliche Informationen am Bildschirm angezeigt werden, ohne den aktuellen Fensterinhalt und somit den zu programmierenden Quelltext zu verändern. Dies können beispielsweise Hinweise auf mögliche Fehler sein, Auskünfte, welche Parameter ein Befehl benötigt oder Informationen, dass bestimmte Bibliotheken fehlen. Fehlertoleranz und Erwartungskonformität (siehe Kapitel 2.3) verlangen dabei, dass auf die Ereignisse in geeigneter Weise hingewiesen wird.

5.1.2 Hilfesysteme

Unter einem Hilfesystem wird die allgemeine Möglichkeit verstanden, einem Anwender zusätzliche Angaben zur Unterstützung der jeweiligen Aufgabe anzubieten. Die einfachste Form eines Hilfesystems unter Unix/Linux sind „Man-Pages“, die in entsprechende Verzeichnisse kopiert werden müssen und anschließend über eine weitere Shell, die neben dem benutzten Programm geöffnet wird, angezeigt werden können. Diese Form wird heute noch bei allen konsolenbasierten Administrationsprogrammen angewandt.

Bei Programmen mit einer GUI-Umgebung sind heute Hilfen wesentlich verbreiteter, die in das Programm integriert sind. Unter Windows-Systemen wird in fast allen Programmen durch einen Druck auf die F1-Taste ein Dialogfenster mit Hilfeinformationen geöffnet. Diese Hilfe fällt wiederum sehr unterschiedlich aus. Im Allgemeinen wird nur eine statische Hilfe angeboten, die das Programm allgemein erklärt und diverse Suchfunktionen für weitere Hilfen zur Verfügung stellt. In Einzelfällen gibt es eine dynamische Hilfestellung, welche die aktuelle Arbeitssituation analysiert und versucht darauf zu reagieren. Die Reaktion einer dynamischen Hilfe kann darin bestehen, dass ein Assistent bei der weiteren Bearbeitung hilft (siehe Kapitel 5.1.4) oder direkt Hilfeseiten zur aktuellen Situation angeboten werden. Diese kontextsensitive Hilfe ist in Entwicklungsumgebungen mittlerweile vielfach anzutreffen und wird meist durch Drücken der Tasten Alt+F1 oder Strg+F1 aktiviert (siehe Kapitel 5.2.2). Daneben werden in Hilfesystemen

häufig Suchfunktionen und Indizierungen zur Verfügung gestellt, die zu einem schnellen Auffinden relevanter Informationen führen. Zusätzlich bieten immer mehr Systeme erweiterte Informationen durch direkten Zugriff auf Internetquellen.

5.1.3 Tool-Tipps

Die Eigenschaften von Tool-Tipps sind den meisten Nutzern durch die heutigen Internet-Browser bekannt geworden. Lässt man beispielsweise den Cursor über einer beliebigen Grafik für kurze Zeit ruhen, erscheint häufig ein (sehr) kleines, meist gelb hinterlegtes Fenster, in dem eine kurze Beschreibung angezeigt wird. Dieses Fenster nennt man Tool-Tipp.

In Entwicklungsumgebungen wird dieses Werkzeug schon etwas länger eingesetzt. 1996 wurde bereits vom Autor auf diese bzw. eine sehr ähnliche Möglichkeiten zur Unterstützung der Befehlseingabe hingewiesen [Tou96]. Seit 1997 findet man sie in Entwicklungsumgebungen von Borland oder Microsoft. Genauere Informationen werden in Kapitel 5.2.4 vorgestellt.

5.1.4 Assistenten

In den letzten Jahren findet man in Programmen immer mehr „Assistenten“, die bei der Erledigung einer komplexen Aufgabe helfen sollen, indem sie diese in kleinere Aufgaben teilen. Zu diesen kleineren Aufgaben werden dem Benutzer schrittweise Fragen gestellt und nach Beantwortung aller Fragen die Gesamtaufgabe vom System selbstständig gelöst.

Bekannt gemacht wurde der Begriff „Assistent“ von Microsoft in dessen MS Office-Paket. Aber auch vorher gab es schon Ansätze, wie man den Benutzer durch schwierige Aufgaben führen kann.

Eines der ersten Einsatzgebiete war die Installation von Programmen, auch wenn man hier nicht von „Assistenz“ sprach. Es werden nach und nach bestimmte Einstellungen abgefragt. Sind alle Informationen vorhanden, beginnt das System mit der Installation.

Betrachtet man Assistenten genauer, erkennt man, dass nichts anderes passiert, als dass immer wieder Dialogboxen eingeblendet werden, in denen Eingabefelder gefüllt werden müssen. Nach Abschluss aller Eingaben wird die angestrebte Aufgabe ausgeführt. Alle Informationen in einer Dialogbox abzufragen ist oft nicht sinnvoll, denn zum einen ist häufig die Anzahl der Abfragen zu hoch, um eine übersichtliche Gestaltung der Dialogbox zuzulassen, zum anderen können bestimmte Felder bereits ausgewertet und die anschließenden Abfragen darauf abgestimmt werden. Die Abfrage unnötiger, in der aktuellen Situation unverständlicher Fragen kann dadurch vermieden werden.

Der Abfolge der Dialogboxen liegt in vielen Fällen eine statische Baumstruktur zu Grunde, bei der je nach Auswertung der aktuellen Ergebnisse in den einen oder anderen Ast verzweigt wird.

Microsoft bezeichnet seinen Office-Assistenten als dynamischen oder intelligenten Assistenten, der auf Benutzungsaktionen reagieren kann. Die Dynamik besteht allerdings nur darin, dass der Assistent die Eingaben des Benutzers überwacht und an bestimmten (im Programm festgelegten) Stellen von sich aus eine Hilfestellung anbietet, ohne dass um eine Hilfe gebeten wurde. Diese Hilfen sind allerdings wiederum statisch und keineswegs intelligenter als Hilfen, die über andere Systeme angeboten werden. In vielen Fällen fühlen sich die Anwender nach einiger Zeit durch die Assistenten belästigt, da an immer den gleichen Stellen die gleichen Tipps gegeben werden. Aus diesem Grund wird der Hilfeassistent häufig abgeschaltet. Einige Gremien werfen Microsoft vor, dass das unerwartete Erscheinen eines Assistenten der Erwartungskonformität widerspricht und dadurch der Arbeitsfluss unterbrochen wird. An diesem Beispiel kann man erkennen, dass Hilfe zu einem falschen Zeitpunkt, an einem falsche Ort oder in falschem Umfang die eigentliche Arbeit eher stören als unterstützen kann. Deshalb ist bei einem guten Hilfesystem darauf zu achten, dass der Benutzer Art, Zeitpunkt und Umfang der Hilfe beeinflussen kann.

5.1.5 Bibliotheken

Die bisher vorgestellten Hilfen werden nicht nur für Programmiersprachen angeboten, sondern sind in allen Bereichen von Computerprogrammen zu finden.

Bibliotheken findet man im Wesentlichen nur bei Programmiersprachen oder bei großen Anwendungsprogrammen, die wiederum über Skript- und Programmierschnittstellen eine Anpassung durch den Benutzer zulassen. Sie sind nicht im direkten Sinne ein Hilfesystem, erleichtern die Arbeit aber immens.

Es gibt keine bessere Hilfe, als wenn eine bestimmte Aufgabe nicht selbst erledigt werden muss, sondern auf einen fertigen und funktionierenden Programmcode zurückgegriffen werden kann. Bibliotheken bieten diese Möglichkeit. Anstatt zum Beispiel selbst einen Sortieralgorithmus zu programmieren, greift man auf fertigen Quelltext zurück, der diese Aufgabe erledigt. Die Funktionen von Bibliotheken sind außerdem häufig auf Performance optimiert, so dass neben einer schnelleren und weniger fehleranfälligen Entwicklung auch ein schneller ablaufendes Programm entsteht.

In vielen Fällen ist ein Nachteil von Bibliotheken, dass die Funktionen, Prozeduren oder Methoden, die man benutzt eine Black-Box sind. Man kann zum Beispiel nur schreiben „BubbleSort(...)“, die konkrete Realisierung des Sortieralgorithmus bleibt unbekannt. Daraus ergibt sich auch der Nachteil, dass kleine Anpassungen nicht vorgenommen werden können. Unter Linux findet man auch Bibliotheken, zu denen der Quelltext verfügbar ist. Dadurch kann man zum einen auf (hoffentlich) ausgereifte Funktionen zurückgreifen, zum anderen kann man den Quelltext den aktuellen Anforderungen anpassen oder aus diesem Anregungen für eigene Entwicklungen entnehmen.

Wichtig ist, dass die zur Verfügung gestellten Funktionen sehr gut in die Entwicklungsumgebung integriert und beschrieben werden, so dass man gar nicht mehr unterschei-

den muss, welche Funktionen bereits von Beginn an zur Verfügung standen oder erst durch die eine oder andere Bibliothek zusätzlich zur Verfügung gestellt werden.

5.1.6 Skelettorientiertes Programmieren

Skelettorientiertes Programmieren ist vereinfacht gesagt die Zusammenführung von Bibliotheken und Assistenten. Für komplexe Algorithmen sind in einer Datenbank Mustervorlagen vorhanden. Um diese Vorlagen anwenden zu können, werden weitere Angaben durch Assistenten vom Entwickler abgefragt. Zum Schluss wird ein Quelltextskelett erstellt, in dem alle Parameter eingefügt wurden. Stellen, die später vom Entwickler noch angepasst werden müssen, sind mit entsprechenden Kommentaren versehen. Dadurch hat man den Vorteil, dass man zum einen sehr leistungsfähige Algorithmen einfach einsetzen, diese andererseits zusätzlich an die gegebenen Umstände ganz genau anpassen kann. Ein weiterer Vorteil besteht darin, dass durch die Anpassung noch effizientere Algorithmen angeboten werden können, da durch die Abfrage von Parametern im Voraus geklärt werden kann, ob Ausnahmen berücksichtigt werden müssen oder im aktuellen Fall beispielsweise ein Spezialfall vorliegt, der viele Abfragen unnötig macht, die ein Algorithmus in einer herkömmlichen Bibliothek berücksichtigen muss.

Entwickelt wurde skelettorientiertes Programmieren für Parallelrechner an der Universität Basel von Prof. Dr. H. Burkard im Projekt „Basler Algorithmen Klassifikationsschema“ (BAKS).

Skelettorientiertes Programmieren wird unter anderem Namen von Programmierumgebungen eingesetzt, indem vor Beginn eines Projektes einige grundsätzliche Fragen zum Programm gestellt werden und darauf zugeschnitten ein erster Quellcode zur Verfügung gestellt wird. Bei den aktuellen Einwicklungsumgebungen ist es allerdings nicht möglich, dass auf diese Weise während der Programmentwicklung einzelne Algorithmen durch dieses Verfahren eingefügt werden. Lediglich das Gerüst für einzelne Befehle kann teilweise durch eine „automatische Befehlsergänzung“ zur Verfügung gestellt werden (siehe Kapitel 5.2.5).

5.1.7 Weitere Möglichkeiten der Benutzungsunterstützung

Bis jetzt wurden Hilfen beschrieben, die allgemein bei der Codierung eines Programms den Entwickler unterstützen. Es gibt allerdings noch viele weitere „Kleinigkeiten“, die für sich gesehen keine neuen Konzepte aufzeigen, in ihrer Gesamtheit den Benutzer aber genauso unterstützen. Einige dieser Möglichkeiten sollen hier kurz angesprochen werden.

Tastenkombinationen

Alle wichtigen Funktionen der Entwicklungsumgebung müssen aufgrund der Erwartungskonformität per Menü erreichbar sein. Mit der Zeit kennt man „seine Entwicklungsumgebung“ sehr gut und es wird als zu zeitaufwändig angesehen, bestimmte Funktio-

nen immer wieder über Menüs abzurufen. Viel schneller geht es, wenn Funktionen alternativ über Tastenkombinationen erreicht werden können. Die Tastenkombinationen sollten allerdings nicht willkürlich gewählt werden. Tastenkombinationen, die weitergehende Aufgaben erfüllen, als nur das aufgedruckte Zeichen am Bildschirm anzuzeigen, nennt man auch „Hotkeys“. Zwar gibt es im Moment keine genormten Vorgaben, allerdings sollte man sich an Quasi-Standards halten, die durch Programme mit sehr hoher Marktpräsenz eingeführt wurden. Beispiele dafür sind die Tastenkombinationen F1 für „Hilfe“, Alt+F4 für „Programmende“ oder Strg+F für „suchen“ (find).

Drag&Drop

Drag&Drop wird mittlerweile von allen kommerziellen Programmen unterstützt. Verstanden wird darunter, dass man durch Drücken einer Maustaste das darunter liegende Objekt oder die aktive Markierung auf dem Bildschirm verschieben kann. Da die Funktion fast allen Computerbenutzern bekannt ist, wird sie hier nicht weiter beschrieben.

Bookmarks / Textmarken

Schreibt man an einer bestimmten Funktion, benötigt man häufig Informationen von einer anderen Stelle des Quelltextes. Nun muss man anfangen, diese andere Stelle zu suchen. Hat man die Information gefunden, sucht man wieder die Stelle, an der man vorher war. Leichter geht es, wenn man beliebige Stellen des Quelltextes markieren und diese anschließend direkt per Tastenkombination oder Menüauswahl anspringen kann. Zur Realisation kann man sich verschiedene Wege vorstellen, die zum einen die Textmarken in einem Fenster zur Auswahl anbieten oder zum anderen nur einen internen Vermerk verwalten, der gar nicht sichtbar ist.

Moderne Entwicklungsumgebungen übernehmen das Setzen von Bookmarks auf den Anfang von selbst geschriebenen Methoden und Funktionen automatisch. Allerdings hat man bei selbst gesetzten Bookmarks den Vorteil, dass die Sprungstellen an jeder beliebigen Stelle im Quelltext angelegt werden können. Somit kann man eine bestimmte Stelle auf Tastendruck exakt anspringen. Andererseits muss man aber selbst dafür sorgen, dass die Sprungstellen markiert werden.

Klammerüberwachung

Auf den ersten Blick ist die Klammerüberwachung nur eine kleine Erweiterung einer Entwicklungsumgebung. Schreibt man im Quelltext eine schließende Klammer, sollte gleichzeitig die dazu gehörende öffnende Klammer markiert werden. Dies kann zum Beispiel durch ein kurzes Aufblinken oder einen Farbwechsel geschehen. Da Klammern häufig geschachtelt werden müssen, verliert man schnell den Überblick, den man auf diese Weise leicht wieder gewinnen kann. Es wird schnell erkennbar, ob noch eine Klammer fehlt. Besonders interessant ist diese Funktion bei Programmiersprachen, die viele Klammern einsetzen, z.B. Lisp, aber auch alle anderen Programmiersprachen benötigen häufig die Funktionalität der Klammerung. Man muss den Begriff „Klammer“

auch nicht so eng fassen, dass man darunter nur die Zeichen „(){}[]“ versteht, genauso ist dies bei Begriffen wie „if – endif“, „while – wend“ oder „for – next“ möglich.

Syntax-Highlighting

Syntax-Highlighting ist eine Funktion, die heute fast alle Entwicklungsumgebungen zur Verfügung stellen. Dabei werden Schlüsselworte, Konstanten, Compileranweisungen, Kommentare, usw. mit unterschiedlichem Schriftbild dargestellt. Teilweise werden dabei unterschiedliche Farben oder Schriftausprägungen wie kursiv oder fett eingesetzt.

Wünschenswert ist Syntax-Highlighting zusätzlich für selbstdefinierte Worte bzw. für Begriffe aus Bibliotheken. Damit können zum Beispiel problemlos eigene Funktionen/Methoden oder Spezialbefehle zum parallelen Programmieren mit eigenem Schriftbild dargestellt werden. Dafür sollte die Art der Darstellung frei wählbar sein. Einige frei erhältliche Editoren verfügen über ein konfigurierbares Syntax-Highlighting. Allerdings können Änderungen nicht während der Arbeit für das jeweilige Projekt angepasst werden, sondern die zu unterstützenden Worte müssen über Konfigurationsdateien eingepflegt werden. Marktführer wie Microsoft haben eine Anpassung an eigene Bedürfnisse bis jetzt nicht vorgesehen.

Textfalten

Die Funktion des Textfaltens besteht darin, dass der Programmierer beliebige, meist fertige und funktionierende Teile des Quelltextes aus dem momentanen Blickfeld ausblenden kann und stattdessen eine Zusammenfassung bzw. ein Symbol für den ausgeblendeten Text angezeigt wird. Dadurch kann die Sicht auf den Quelltext auf das Wesentliche reduziert werden. Ein Algorithmus kann dadurch leichter lesbar und verständlicher werden. Ein ähnliches Konzept wurde bereits in einem Editor von Inmos für einen Transputercluster Anfang der 1990er Jahre realisiert. In [Tou96] wurde auf diese Methode als wichtige Benutzerunterstützung für Parallel-Editoren hingewiesen. Auch die neueren Entwicklungsumgebungen von Microsoft (Visual Studio) haben das Ein- und Ausblenden von Textteilen wiederentdeckt. Hier werden vor allen mehrzeiligen Befehlen durch ein Plus-Zeichen in der ersten Zeile, einem Minus-Zeichen in der letzten Zeile und einem dazwischen liegenden Strich die Zusammengehörigkeit angezeigt und der Bereich auf Wunsch per Klick „eingefaltet“. Auch die Firma Borland bietet diese Funktionalität in ihren neuen Versionen von JBuilder und Delphi an. Weitere Editoren, die diese Funktion in ihren neueren Versionen enthalten, sind: Eclipse, KDevelop, NetBeans, JCreator, UltraEdit, Kate, XEmacs und vim.

Projektmanagement, Makefile-Generierung und Funktionsübersicht

Bei größeren Projekten hat man häufig das Problem, dass man nicht mehr weiß, in welcher Datei die jeweilige Funktion implementiert wurde oder wo die Funktion innerhalb der aktuellen Datei zu finden ist. Aus diesem Grund sollte eine automatisch aktualisierende Liste angezeigt werden können, in der alle selbst entwickelten Funktionen des Projektes enthalten sind. Diese Liste kann in einem eigenen Fenster ständig oder nach

Anforderung dargestellt werden. Genauso ist eine Anzeige innerhalb des Programmmenüs denkbar. Durch entsprechende Auswahl einer Funktion kann man direkt zu dieser gelangen, um dort Änderungen durchzuführen und evtl. durch einen weiteren „Hotkey“ wieder an die Ursprungsstelle zurückkehren.

Aus den einzelnen Dateien soll letztlich ein lauffähiges Programm werden. Dazu müssen die einzelnen Dateien in der richtigen Reihenfolge übersetzt werden, damit alle Programmteile die benötigten Informationen erhalten können. Durch eine Analyse der Quelltexte kann ein Algorithmus ermitteln, welche Abhängigkeiten bestehen und damit die richtige Reihenfolge erstellen. Das Ergebnis wird bei C oder C++ Programmen in so genannten Makefiles gespeichert, welche wiederum genutzt werden, um anschließend die Compilation zu steuern.

Integrierter Compiler

Nachdem ein Entwickler neuen Quelltext in sein Programm eingefügt hat, möchte er diesen testen. Dazu muss für ein C/C++ Programm ein Compiler aufgerufen werden. Sinnvoll ist es nun, wenn die Entwicklungsumgebung nicht verlassen werden muss, sondern der Compiler direkt über das Menü oder eine Tastenkombination angesprochen wird. In einem eigenen Fenster sollten die Informationen des Compilers zur Verfügung gestellt werden. Werden Fehlermeldungen zurückgegeben, sollte durch Auswahl einer dieser Meldungen die Entwicklungsumgebung in der Lage sein, sofort den entsprechenden Quelltext zur Korrektur anbieten zu können.

Integrierter Interpreter

Da heute alle Sprachen, die auf Geschwindigkeit ausgelegt sind, Compiler benutzen, ist der Einsatz eines Interpreters auf den ersten Blick nicht einzusehen. Dieser soll auch nicht dazu benutzt werden, das spätere Programm zu erzeugen. Während der Entwicklung kann ein Interpreter aber sehr schnell auf eventuelle Fehler hinweisen, noch bevor ein Compiliervorgang von Hand gestartet wurde. Damit wird das Wechseln der Aufgaben des Entwicklers vermieden. Eventuelle Meldungen des Interpreters müssen so präsentiert werden, dass der Programmierer nicht behindert wird. Microsoft benutzt bei seiner Visual Basic Sprache keinen kompletten Interpreter, allerdings wird jede Zeile nach der Eingabe sofort auf syntaktische Fehler geprüft und dem Benutzer dies im Fehlerfall durch eine Farbveränderung der Schrift deutlich angezeigt. Borland bietet in seinen Entwicklungsumgebungen ähnliche Hilfen an.

Neben einem Interpreter ist auch ein intelligenter Compiler denkbar, der durch die Entwicklungsumgebung gesteuert wird. Da die Funktion nach jeder Eingabe, bzw. spätestens nach Verlassen einer veränderten Zeile aufgerufen werden sollte, sind die Änderungen am Quelltext nicht sehr groß. Compiler müssen im Normalfall nicht das komplette Projekt neu übersetzen, sondern können dies beispielsweise auf die aktuelle Datei oder Methode beschränken. Heutige leistungsstarke Rechner sind in der Lage diese Funktion im Hintergrund sehr schnell durchzuführen. Somit können syntaktische Fehler (fast) in Realzeit auf sinnvolle Weise angezeigt werden.

5.2 Befehlsorientierte Hilfe ohne grafische Unterstützung

Bisher wurden mehrere Unterstützungsmöglichkeiten vorgestellt, die jeweils Teile des gesamten Codierungsprozesses unterstützen. Im Folgenden sollen solche Hilfestellungen genauer untersucht werden, welche die direkte Eingabe eines Befehls unterstützen.

Beim Programmieren kommt es auf die korrekte Syntax jedes einzelnen Befehls an. Dabei ist es sehr unbefriedigend, wenn Programmtext fertig gestellt wird und der Compiler im Nachhinein Fehlermeldungen ausgibt, auch wenn dies durch integrierte Interpreter/Compiler sehr zeitnah geschehen kann. Viel angenehmer ist es, wenn Fehler bereits im Vorfeld vermieden werden können.

Fehler auf Befehlsebene treten zum einen auf semantischer Seite auf, wenn man den „am besten geeigneten“ Befehl nicht kennt. Ist er bekannt, kann es zum anderen immer noch zu syntaktischen Problemen kommen, da man nicht immer alle Parameter und deren richtige Reihenfolge kennt oder sich einfach bei der Eingabe einer dem System bekannten Konstanten verschrieben hat.

Aus diesem Grund ist es wichtig, dass man den Programmierer bereits bei der Befehlseingabe so weit wie möglich unterstützt. Bietet man diese Unterstützung an, ist darauf zu achten, dass die Grundsätze der Dialoggestaltung (siehe Kapitel 2.3) eingehalten werden. Insbesondere ist die Aufgabenangemessenheit zu berücksichtigen, denn unterstützt eine Hilfe die Programmerstellung nicht auf effiziente Weise, ist es besser, die Hilfe wird überhaupt nicht angeboten.

Da Programmierer unterschiedliche Kenntnisse besitzen, benötigen sie auch unterschiedliche Hilfestellungen. Der Eine benötigt nur einen kurzen Hinweis und möchte möglichst nicht im Programmierfluss unterbrochen werden, der Andere benötigt eine wesentlich umfangreichere Hilfe, da er eventuell einen sehr komplexen Befehl zum ersten Mal einsetzen möchte. Dies kann durch eine Individualisierung der Hilfe erreicht werden.

5.2.1 Statische Hilfefunktion

Die ersten Compiler wurden komplett ohne Programmierumgebung geliefert. War man auf die Hilfe für einen Befehl angewiesen, so musste man diesen in einem Buch nachschlagen.

Unter DOS wurden mit den ersten Entwicklungsumgebungen die integrierten Hilfefunktionen eingeführt. Dabei gibt es eine Tastenkombination (z.B. Strg+H oder F1) mit der man zu einer Start-Hilfeseite gelangen kann. Von dort aus muss man durch mehrere Seiten navigieren und gelangt damit zu den Befehlserklärungen.

Durch dieses umständliche Suchen in der Hilfe wird man jedes Mal im Programmierfluss unterbrochen. Mit Einführung größerer Bibliotheken ist die Anzahl der Befehle so umfangreich geworden, dass eine Suche nach dem richtigen Befehl zu aufwändig wird. Weiterhin ist es mit diesem Ansatz nicht möglich, die Hilfe für den jeweiligen Benutzer anzupassen, da nur ein starres Schema vorgesehen ist.

Aus diesem Grund wird diese statische Art der Hilfe heute kaum noch für die Unterstützung bei der Erklärung einer Befehlssyntax eingesetzt. Für einen ersten Überblick und allgemeine Erklärungen, wie die Programmierumgebung aufgebaut ist, welche Möglichkeiten sie bietet, welche Tastenkombinationen es gibt, ist sie immer noch sehr sinnvoll.

Neben einer Start-Hilfeseite wird nun häufig ein Suchfeld zur Verfügung gestellt. Damit ist es möglich, eine Volltextsuche über alle Hilfeseiten durchzuführen. Teilweise werden in der Ergebnisliste ähnliche Begriffe zur Auswahl angeboten, so dass mit dem richtigen Schlagwort eine gewünschte Hilfe schnell gefunden werden kann.

5.2.2 Kontextsensitive Hilfe

Wird an einer bestimmten Stelle im Programm Hilfe benötigt, ist es nur sehr selten sinnvoll, die Hilfe auf einer zentralen Startseite zu beginnen, sondern man möchte gerne automatisch eine Hilfe, die beim augenblicklichen Problem direkt weiterhilft. Auch die Eingabe des Befehls in der Suchen-Funktion einer Hilfe ist zu umständlich, denn die Umgebung besitzt bereits genügend Informationen, um „auf Anhieb“ die richtige Hilfe selbstständig anbieten zu können. Dies kann durch die so genannte kontextsensitive Hilfe ermöglicht werden.

Gibt man beispielsweise die Buchstaben „`for`“ ein, um in C++ eine Schleife zu programmieren, kennt aber die korrekte Syntax zum `for`-Befehl nicht, steht der Cursor normalerweise auf oder direkt hinter diesem Wort. Nach dem Druck einer bestimmten Tastenkombination (z.B. Alt+F1) kann von der Umgebung geprüft werden, wo der Cursor gerade steht. Ein Parser ermittelt anschließend, welches Wort in der Umgebung des Cursors steht. Somit kann direkt zu diesem Wort bzw. Befehl ein Hilfetext angeboten werden. Sollte der Parser keinen bekannten Befehl erkennen, so muss dem Anwender aus Gründen der Erwartungskonformität entweder eine Liste von „ähnlichen“ Befehlen zur Auswahl oder eine Fehlermeldung angezeigt werden.

Falls eine Hilfe angeboten werden kann, gibt es eine sehr große Bandbreite, wie dieses Angebot dargestellt werden kann.

Im einfachsten Fall wird eine Dialogbox geöffnet, die den Befehl mit seinen Argumenten anzeigt und eine kurze Erklärung gibt. Nach Beenden der Dialogbox kann diese Information genutzt werden, um die eigene Befehlsumsetzung korrekt zusammenzustellen.

Einige Systeme bieten zu dieser Kurzerklärung weitere Informationen an. So bekommt man eventuell als Option ein kleines Beispielprogramm eingeblendet, das zeigt, wie man den Befehl einsetzen kann und wie eine korrekte Syntax aussehen kann. Des Weiteren gibt es meist die Möglichkeit, dass man „ähnliche Befehle“ oder Befehle, die im Zusammenhang häufig benötigt werden, zur Auswahl gestellt und dadurch auch zu diesen die entsprechenden Informationen bereitgestellt bekommt.

Komplexe Programme benötigen im Normalfall für ihre volle Funktionalität Informationen aus ihrem Umfeld. So müssen beispielsweise Dateien aus bestimmten Verzeichnissen geladen, Fenster verschoben bzw. in der Größe geändert oder andere Programme mit

Informationen versorgt werden. Dazu werden so genannte API-Aufrufe (Application Programmers Interface) benutzt, die direkt auf Funktionen des zugrunde liegenden Betriebssystems zugreifen. Dies sind normalerweise Funktions- bzw. Methodenaufrufe von C/C++-Funktionen mit teilweise sehr komplexen Argumenten. Allerdings kennen selbst die Entwickler der einzelnen Programmierumgebungen nicht alle API-Aufrufe. Aus diesem Grund hat z.B. Microsoft für Windows-Umgebungen ein eigenes Programm entwickelt, das zusätzlich zur eigentlichen Programmierumgebung gestartet werden kann und alle benötigten Informationen zu den API-Befehlen zur Verfügung stellt. Der Informationsumfang ist riesig, so dass das Produkt auf mehreren CDs oder einer DVD ausgeliefert werden muss und über 1 GB an Informationen bereitstellt.

Durch diese Vielfalt und die auf den ersten Blick unübersichtliche Anzahl an vordefinierten Variablentypen, benötigt man allerdings einige Zeit, bis man sich in dieser „Hilfe“ zurechtfindet. Deshalb haben Firmen wie Borland zusätzlich eigene Hilfetexte zu den API-Funktionen in ihre Umgebung integriert, wobei diese nicht so ausführlich sind. Microsoft selbst integriert das normalerweise selbständige Produkt mit allen Hilfen zu den API-Funktionen in seine Entwicklungsumgebungen.

Die kontextsensitive Hilfe hat sehr dazu beigetragen, dass durch die gezielten Informationen der Programmiervorgang beschleunigt werden kann. Allerdings muss neben dem Programmierfenster prinzipbedingt immer ein weiteres Fenster für den Hilfetext geöffnet werden, was auf kleineren Bildschirmen schnell zu unübersichtlichen Situationen führen kann. Eine aktuelle Studie empfiehlt zurzeit Bildschirmarbeitsplätze mit sechs Bildschirmen, wobei jeweils drei nebeneinander und zwei übereinander angebracht sein sollen. Somit soll für alle Arbeiten genügend Sichtraum vorhanden sein, allerdings hat sich diese große Visualisierungsfläche (noch) nicht durchgesetzt. Aus diesem Grund ist es wünschenswert, dass man entweder, wenn man nur den Bedarf auf einen kleinen Hinweis hat, das Programmierfenster überhaupt nicht verlassen muss oder, wenn man eine umfangreichere Unterstützung zu der augenblicklichen Syntax wünscht, den Befehl im Dialog mit der Hilfestellung zusammenstellen kann und der komplette Befehl nachträglich vom System automatisch in das Programmierfenster übertragen werden kann.

5.2.3 Kontextsensitive Hilfe mit direkter Eingabemöglichkeit

Wie gerade erläutert kommt es bei komplexen Befehlen sehr häufig dazu, dass ein Blick in die Hilfe nicht ausreicht, um sich die komplette Syntax eines Befehls zu merken und diesen anschließend im Programmierfenster korrekt zusammensetzen zu können.

Aus diesem Grund wird in diesem Abschnitt eine erweiterte Hilfefunktion vorgestellt, welche die vorhandenen Befehlsinformationen direkt neben einer Eingabemöglichkeit präsentiert und somit dem Benutzer den ständigen Wechsel zwischen dem Hilfefenster und dem Editor abnimmt. Anders als bei der weiter unten vorgestellten „automatischen Befehlsergänzung“ besteht in dieser Ausprägung der Hilfestellung die Möglichkeit, wesentlich mehr Informationen zur Verfügung zu stellen, so dass auch komplexeste Befehle ausreichend für die Erstbenutzung erklärt werden können.

Eine erste Veröffentlichung zu dieser Hilfestellung mit direkter Eingabemöglichkeit erfolgte vom Autor bereits 1996 [Tou96]. Darin wird gezeigt, wie ein Befehl mit all seinen Argumenten komplett in einem Hilfefenster zusammengebaut werden kann und der fertige Befehl anschließend automatisch in das Programmierfenster übertragen wird.

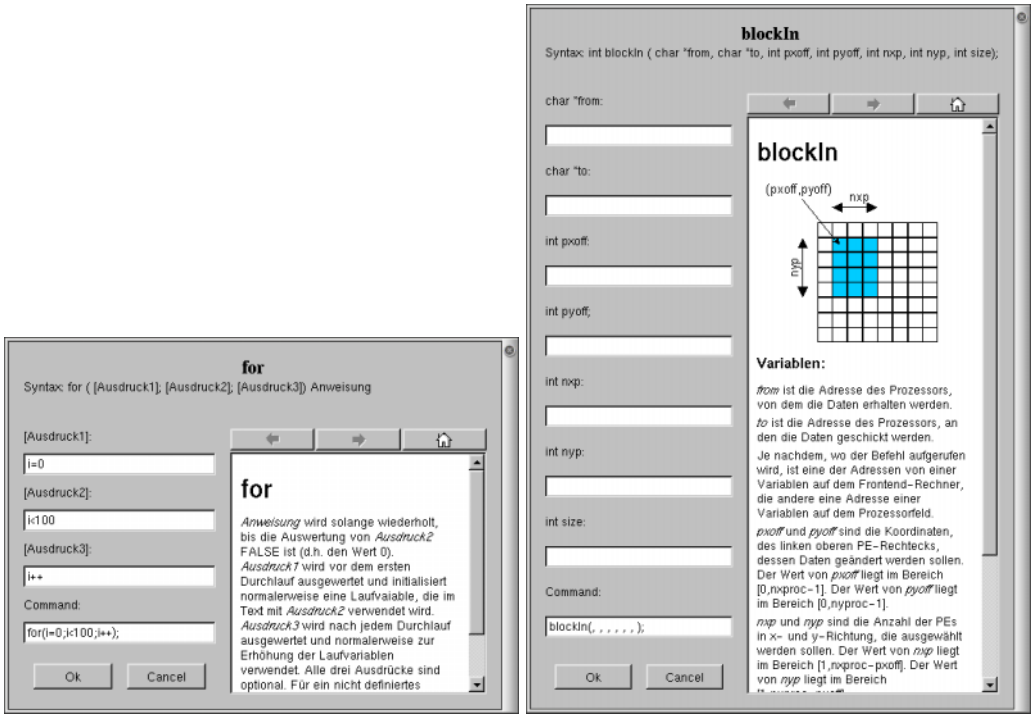


Abbildung 5-2: Screenshots von kontextsensitiver Hilfe mit direkter Eingabemöglichkeit

Abbildung 5-2 zeigt jeweils ein Hilfefenster, das als Überschrift den Befehlsnamen hat, der aktuell bearbeitet werden soll. Danach folgt eine Syntaxzeile, in der alle Argumente und deren Typen gezeigt werden. Diese Hilfe ist einerseits für Standardbefehle der Grundsprache, aber besonders auch für komplexe Befehle der parallelen Erweiterungssprache sinnvoll.

Auf der rechten Seite des Fensters findet man die ausführliche Hilfe. Sie wird in diesem Beispiel mittels einer HTML-Syntax aufgebaut, so dass auch ein Springen von einer Information zur nächsten möglich ist. Zu Beginn werden hier allgemeine Informationen zum jeweiligen Befehl angezeigt: Sinn und Zweck des Befehls oder benötigte Bibliotheken und Headerfiles. Anschließend folgen Informationen, die man zum Bearbeiten der einzelnen Argumente benötigt: Welche Aufgabe hat das Argument, welche Typen werden benötigt, wie setzt sich der Typ zusammen, muss eine Eingabe erfolgen, usw. Am Ende findet man evtl. einen Verweis auf ein Beispielprogramm oder auf ähnliche Befehle.

Zu den speziellen Informationen über die einzelnen Argumente gelangt man auch, indem der Cursor direkt in eines der Argumentfelder gesetzt wird, die im linken Teil der Dialogbox zu finden sind. Wird ein Eingabefeld aktiviert, so wird in der rechten Fensterhälfte die entsprechende Argumentbeschreibung automatisch an oberster Stelle angezeigt. Somit erhält der Anwender automatisch die notwendigen Informationen, um das Argumentfeld füllen zu können.

Im unteren Teil des Hilfefensters wird der komplette Befehl in korrekter Syntax zeitgleich zusammengestellt. Somit kann der Anwender jederzeit den kompletten Befehl überblicken und seine Kenntnisse erweitern. Mit der Zeit wird es nicht mehr notwendig sein, die ausführliche Hilfe mittels einer Dialogbox zu benutzen, sondern es reicht beispielsweise die unten beschriebene Unterstützung per Tool-Tipp. Nach dem Schließen des Fensters ersetzt der fertig gestellte Befehl automatisch den beim Aufruf unter dem Cursor stehenden Bereich im Editorfenster.

Weiterhin kommt es häufig vor, dass man einen Befehl bereits teilweise mit seinen Argumenten geschrieben hat, dann aber nicht mehr alle Argumente oder deren richtige Reihenfolge kennt. Möchte man nun nicht nur die kontextsensitive Hilfe benutzen, sondern, wie hier vorgestellt, in eine Hilfe mit Eingabemaske wechseln, wird der bisher unvollständige Befehlsteil in die Dialogbox übertragen und die bereits vorhandenen Argumente in entsprechenden Felder der Maske eingetragen. Dazu ist es notwendig, eine größere Umgebung um den Cursor auszuwerten, um die befehlsrelevanten Daten an die Dialogbox übertragen zu können.

Zusätzlich ist es denkbar, dass der Programmierer die Eingabe in der unten stehenden Ausgabezeile der Hilfebox vervollständigen möchte. Dies sollte aus Flexibilitätsgründen möglich sein, wurde in dem realisierten Prototyp allerdings nicht umgesetzt.

Betrachtet man diese Hilfe unter dem Gesichtspunkt der Aufgabenangemessenheit, so erkennt man, dass der alleinige Einsatz dieser Hilfestellung unangebracht ist, da in weniger aufwändigen Fällen die Umorientierung weg vom Editorfenster hin zur Eingabe in einer zusätzlichen Dialogbox einen zu großen Aufwand für das Auge darstellt. Es wird einige Zeit benötigt, bis man sich auf das neue Fenster einstellt. In Kombination mit den anderen in diesem Kapitel vorgestellten Unterstützungen erhält der Anwender allerdings ein an alle Anwendungsfälle angepasstes Hilfesystem.

Die kontextsensitive Hilfe mit direkter Eingabemöglichkeit kann noch ergonomischer gestaltet werden, indem Tool-Tipps innerhalb der Dialogbox eingesetzt werden und für die Eingabe der Argumente sinnvolle Werte vorgeschlagen werden. Aber auch ohne den Einsatz einer Dialogbox ist die Unterstützung durch Tool-Tipps heute zu einem sehr wichtigen Instrument geworden.

5.2.4 Hilfe durch Tool-Tipps

Wie zu Beginn des Kapitels erwähnt, ist es nicht immer notwendig, eine ausführliche Hilfe zu erhalten. Ganz ohne Hilfe kommt man aber häufig auch nicht weiter. Möchte man beispielsweise einen Kopierbefehl benutzen, weiß im Moment nur nicht, welches Argument Quelle und welches Ziel ist, so reicht eine kurze Information aus. Um möglichst wenig bei der weiteren Programmierung abgelenkt zu werden, sollte die Hilfe genau dort gezeigt werden, wo sie benötigt wird: Direkt neben dem Befehl, ohne großes Fenster und nur diejenige Information, die unmittelbar benötigt wird. Abbildung 5-3 zeigt einen Tool-Tipp zum Kopierbefehl eines Strings.

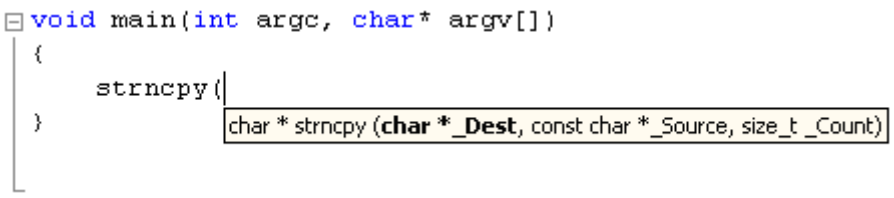


Abbildung 5-3: Screenshot eines Tool-Tipps

Damit man den Hilfetext vom Programmiertext unterscheiden kann, wird er in vielen Programmierumgebungen zur Abgrenzung mit einen transparent-gelben Hintergrund unterlegt oder bei größeren Tool-Tipps, die Alternativen anbieten, in Form eines Drop-Down Menüs angezeigt.

Beginnt man in modernen Entwicklungsumgebungen mit der Eingabe eines Befehls und wartet nach einigen Buchstaben eine bestimmte Zeit, werden durch einen Tool-Tipp mögliche Befehle angezeigt, die mit den entsprechenden Buchstaben beginnen. Nun kann man mit Maus, Cursortasten oder Tastenkombinationen einen Befehl auswählen oder ohne weitere Einschränkung die Eingabe per Tastatur vervollständigen.

Zur Eingabe angeboten werden dabei im Idealfall nicht nur die von der jeweiligen Sprache angebotenen Befehle, sondern es werden auch alle selbst geschriebenen Befehle automatisch berücksichtigt. Dass eine Umgebung alle selbst geschriebenen Funktionen übersichtlich anbieten soll, wurde ebenfalls bereits in [Tou96] erwähnt.

Besonders wichtig ist die Tool-Tipp-Funktion bei den heutigen objektorientierten Sprachen, da man dadurch bereits durch Eingabe des Klassennamens eine Übersicht über alle zur Verfügung stehenden Methoden erhält.

Wird nun ein Befehl ausgeschrieben oder durch einen Mausklick vervollständigt, wird automatisch eine Liste der benötigten Argumente angezeigt. Wenn eine Liste nicht eindeutig ist – was durch das Überladen von Methoden bei der objektorientierten Programmierung möglich ist – werden, solange keine Eindeutigkeit vorhanden ist, mehrere Listen angeboten. Für jedes einzelne Argument werden als Information ein Schlagwort, welches das Argument beschreiben soll und der benötigte Typ zur Verfügung gestellt. Wurde ein Argument eingegeben, verschwindet es aus der Liste und nur die jetzt noch benötigten Argumente werden vom Tool-Tipp angezeigt.

Mit diesen Tool-Tipps ist es möglich, dass der Anwender jederzeit eine Hilfestellung erhält, die ihm bei der Vervollständigung des aktuellen Befehls hilft, ohne dass er das Programmierfenster verlassen muss. Selbst der Ort der Texteingabe muss mit den Augen nicht verlassen werden, da direkt neben dem Eingabecursor die Hilfe erscheint.

Im Gegensatz zur Hilfe durch Dialogboxen kann durch Tool-Tipps nur eine sehr geringe, aber gezielte Hilfe angeboten werden. Im Vordergrund ist hier die flüssige Befehlseingabe zu sehen. Dieses System ergänzt die komplette Hilfestellung bei der Befehlseingabe.

Mit Tool-Tipps kann dynamisch Hilfe zu jedem neuen, auch selbst geschriebenen Befehl angeboten werden. Man benötigt lediglich einen guten Parser, der während der Laufzeit die Quelltexte durchsucht. Diese Dynamik ist auch innerhalb von Dialogboxen möglich.

Auch erfahrene Programmierer greifen gerne auf Tool-Tipps zurück, da sie Dank der automatischen Befehlsergänzung immer nur die ersten Buchstaben eingeben müssen und dann bereits „blind“ lange Befehle ergänzen lassen können. Somit können einerseits lange aussagefähige Befehlsnamen eingesetzt werden, andererseits können die Befehle trotzdem sehr schnell eingegeben werden. Im nächsten Abschnitt wird die automatische Befehlsergänzung genauer vorgestellt.

Nach dem Gesichtspunkt der Aufgabenangemessenheit sind Tool-Tipps zusätzlich zur Hilfe durch Dialogboxen sehr sinnvoll, ohne diese ersetzen zu können. Je nachdem, wie viel Hilfe gerade benötigt wird, kann eine der drei oben genannten Hilfestellungen gewählt werden. Die Erwartungskonformität wird in keiner Weise getrübt, da die jeweilige Hilfe vom Benutzer gewählt wird, die Individualisierbarkeit an die Bedürfnisse des jeweiligen Benutzers wird durch die Tool-Tipps weiter erhöht. Auch die Lernförderlichkeit wird gesteigert, da man die korrekte Syntax immer vor Augen hat. Die Fehlertoleranz des Systems ist gegeben, da die Eingabe ohne einschränkende Beeinflussung fortgesetzt werden kann. Wird ein Befehl so geschrieben, dass er mit der Hilfe nicht mehr übereinstimmt, wird die Hilfe automatisch ausgeblendet. Die Selbstbeschreibungsfähigkeit ist jederzeit gegeben, die Steuerbarkeit ist dadurch gewährleistet, dass zum einen in den allgemeinen Einstellungen einer Entwicklungsumgebung entschieden werden kann, ob man Tool-Tipps angezeigt bekommen möchte und man zum anderen einen Tool-Tipp jederzeit durch die ESC-Taste in der jeweiligen Situation ausblenden kann.

5.2.5 Automatische Befehlsergänzung

Ein weiterer Ansatz, die Eingabe eines Befehls zu vereinfachen, besteht darin, einen Befehl auf Tastendruck vom System ergänzen zu lassen.

Ausgehend vom aktuellen Wort unter dem Cursor kann ermittelt werden, welcher Befehl gerade eingegeben werden möchte. Dieser wird nun vervollständigt und die Argumente mit Schlagwort und Typ beschrieben. So kann beispielsweise aus einem „`rou`“ ein „`router(/* int Quelle*/, /*int Ziel*/)`“ werden. `router` ist ein Befehl, der für den MasPar-Parallelrechner benötigt wird.

Die Ergänzung kann direkt in der Befehlszeile ausgeführt werden, so dass sie direkt zum ausführbaren Code hinzugefügt wird. Möglich ist dies nur, wenn keine Alternativen in Hinsicht auf Befehlserweiterung, Argumentanzahl und –typ vorhanden sind. Ist eine Eindeutigkeit nicht gegeben, muss eine entsprechende Meldung (evtl. nur ein Piepston oder ein Tool-Tipp) dies melden. Gleichzeitig könnte der wahrscheinlichste Befehl vorgegeben werden, der mit einer weiteren Tastenkombination rollierend durch die anderen Alternativen ausgetauscht werden kann.

Im obigen Beispiel wird der komplette Befehl ergänzt und die Argumente durch Kommentare ersetzt. Dies muss nicht sinnvoll sein, da anschließend die Kommentare wieder vom Anwender durch korrekte Argumente ersetzt werden müssen. Deshalb wird im Normalfall nicht zum kompletten Befehl ergänzt, sondern immer nur so weit, wie der Befehl eindeutig ist. Da bei den einzufügenden Argumenten nur der Typ vorgegeben ist,

aber nicht der Variablenname oder -wert, kann hier bei einer reinen Befehlsergänzung innerhalb eines Texteditors keine Hilfe angeboten werden.

Der Vorteil der automatischen Befehlsergänzung gegenüber den Tool-Tipps liegt darin, dass ein rein textorientiertes Betriebssystem ausreicht. Es sind keine Fensterfunktionen nötig. So findet man die Befehlergänzung beispielsweise in UNIX-Shells wieder. Auch die Kommandozeile von Windows erlaubt teilweise eine Befehlsergänzung.

Bei Entwicklungsumgebungen steht im Normal eine grafische Umgebung zur Verfügung, die mehrere Fenster anzeigen kann. Somit kann die automatische Befehlsergänzung ideal um Tool-Tipps bereichert werden. Solange die Vorgaben nicht eindeutig sind, werden in den Tool-Tipps alle Alternativen vorgeschlagen. Der Anwender kann darin auswählen und der Befehl wird ergänzt. Für die Argumente können je nach Mächtigkeit des Systems entweder nur die notwendigen Variablentypen angezeigt werden oder sogar die aktuell zur Verfügung stehenden Variablen, die an dieser Stelle eingefügt werden könnten. Hat man sich an die notwendigen Tastenkombinationen gewöhnt, so wird die automatische Befehlsergänzung zu einem mächtigen Instrument, das von Profi-Programmieren sehr gerne benutzt wird, um die Eingabe zu beschleunigen, ohne Flüchtigkeitsfehler bei der Syntax befürchten zu müssen.

5.3 Befehlsorientierte Hilfe mit grafischer Unterstützung

Bei den bisherigen Hilfesystemen wurde ein Element bis jetzt nicht berücksichtigt, das Zusammenhänge häufig sehr gut verdeutlichen kann: Grafiken zur Veranschaulichung von Zusammenhängen und zur besseren Übersichtlichkeit.

Wird bereits eine HTML-basierte Hilfestellung angeboten, ist es kein Problem, auch Grafiken einzusetzen. Bis jetzt werden aber meist nur Strukturierungsmittel wie Aufzählungspunke oder Tabellen eingesetzt. Allerdings können selbst bei einer HTML-Hilfe Grafiken nur als passives Mittel benutzt werden. Das bedeutet, man kann durch eine Abbildung einen Zusammenhang besser verstehen, die Grafik kann aber nicht durch Eingaben verändert werden, um einen Vorgang zu verdeutlichen oder gar das Ergebnis eines Dialogs als neue Eingabeform für den eigenen Quelltext benutzen.

Eine Ausnahme dieser Regel sind bisher User Interface Builder (siehe Kapitel 2.5). Mit ihnen wird die Oberfläche eines Programms modelliert. Dies ist ähnlich wie in einem Zeichenprogramm möglich. Die fertige Zeichnung wird anschließend vom System in die entsprechende Syntax der Programmiersprache übertragen. Wie bereits beschrieben wird dadurch ein Quelltextgerüst zur Verfügung gestellt, im dem später die einzelnen Aktionen, die ausgelöst werden, von Hand programmiert werden müssen. UIMS bieten für diese Aktionen zusätzlich häufig eine eigene Syntax an, MD-IDE ebenfalls, allerdings müssen in jedem Fall die Aktionen textuell beschrieben werden.

Aber nicht nur beim Oberflächendesign ist der Einsatz grafischer Eingabehilfen sinnvoll. Gerade bei parallelen Programmen können durch Grafiken Zusammenhänge veranschaulicht werden: Welche Prozesse laufen parallel ab (z.B. bei MPI-Programmen), wo

kommunizieren Prozessorelemente miteinander, welche Prozessorelemente sind gerade aktiv, usw.?

In bisherigen parallelen Umgebungen wird das Element Grafik nur als Medium genutzt, um im Nachhinein eine bessere Übersicht über den Quelltext zu ermöglichen. So werden, wie in Kapitel 4.3 gezeigt, externe Fenster genutzt, um beispielsweise Kommunikationen oder Abhängigkeiten zu visualisieren. Bei der direkten Codierung werden Grafiken in keiner untersuchten Programmierumgebung als Unterstützungsmittel angeboten, es sei denn, die Programmierung geschieht komplett grafisch, indem Grafen mit bestimmten Eigenschaften aufgebaut werden.

In den folgenden Abschnitten soll am Beispiel des SIMD-Rechners MasPar gezeigt werden, wie für diesen Rechnertyp die grafische Befehlsunterstützung eingesetzt werden kann, damit auch Nicht-Experten auf einfache Weise die parallelen Besonderheiten des Rechners ausnutzen können und trotzdem die gewohnte Programmierung innerhalb eines Textfensters erhalten bleibt. Dabei wird sehr ausführlich erläutert, wie grafische Abbildungen den Programmierer besonders bei den parallelen Programmteilen der Aktivierung von Prozessorelementen bzw. der Kommunikation unterstützen können. Es wird zusätzlich gezeigt, dass durch eine geeignete Auswertung des vorhandenen Quelltextes eine korrekte Überführung von Quelltextteilen in grafische Abbildungen und von grafischen Abbildungen in Quelltext durchgeführt werden kann. Da die grafischen Abbildungen einen Kontext besitzen, gehen bei dieser Umwandlung keine Informationen verloren.

Bisher war es nicht möglich, die in diesem Kapitel vorgeschlagenen grafischen Unterstützungen direkt in einen Editor zu integrieren. Deshalb wird in Kapitel 1 eine neu entwickelte Programmierumgebung vorgestellt, welche diese und weitere Eigenschaften zur Verfügung stellt.

Um die Erkenntnisse aus den folgenden Abschnitten zu verallgemeinern, wird anschließend in Kapitel 7 gezeigt, wie grafische Benutzungsunterstützungen sinnvoll auf weitere Typen von massivparallelen Rechnern übertragen werden können.

5.3.1 Hilfe bei der Aktivierung von Prozessorelementen

Wie im Beispiel in Kapitel 3.3 vorgestellt, ist es in vielen Fällen notwendig, eine Berechnung nicht auf allen Prozessorelementen auszuführen, sondern nur auf ganz bestimmten, den so genannten aktiven Elementen. Diese Aktivierung von Prozessorelementen ist jederzeit möglich. Allerdings sind während dieser Zeit die anderen Prozessorelemente automatisch „deaktiviert“, führen also keine Befehle aus, so dass die Gesamtrechenleistung stark von der Zahl der aktivierten Prozessorelemente abhängig ist (siehe Kapitel 3.1). Zu beachten ist, dass es aktiven Prozessorelementen dabei jederzeit erlaubt ist, auch Daten von inaktiven Elementen anzufordern.

Um Aussagen darüber treffen zu können, welche Prozessorelemente typischerweise aktiviert bzw. deaktiviert werden, wurden Quelltexte analysiert, die im Rahmen einer Dissertation [Koh99] und mehrerer Diplomarbeiten angefertigt wurden [Tem95, Vei95,

Eic96]. Betrachtet man in bestehenden SIMD-Programmen, welche Prozessorelemente aktiv sind, erkennt man relativ schnell in fast allen Fällen symmetrische Muster.

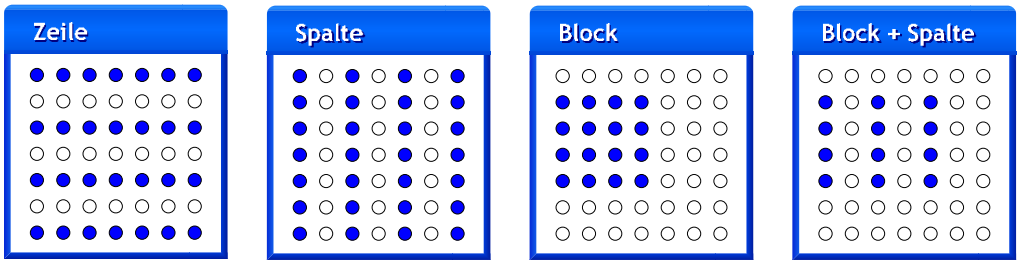


Abbildung 5-4: Einige häufige Muster bei Aktivierungen von Prozessorelementen

In Abbildung 5-4 werden einige Muster gezeigt, wobei aktive Prozessorelemente einen ausgefüllten Kreis und inaktive durch einen nicht ausgefüllt Kreis dargestellt werden. So sind häufig aktive Elemente zeilen- oder spaltenweise angeordnet, allerdings nicht immer über das ganze Gitter hinweg, sondern nur innerhalb eines bestimmten Bereichs. Des Weiteren sind in einer Zeile oder Spalte nicht alle Elemente aktiv, sondern zum Beispiel nur jedes Zweite. Auch der Abstand von aktiver Zeile zu aktiver Zeile bzw. von Spalte zu Spalte kann je nach Programm variieren.

Das Vorhandensein dieser Symmetrien kann man nutzen und gezielt mit Werkzeugen unterstützen. Möchte man einen Algorithmus für den SIMD-Rechner programmieren, so wird dieser im Allgemeinen in Gedanken grafisch auf das Prozessorfeld verteilt und die Zusammenarbeit der Prozessorelemente verbildlicht. In Kapitel 3 wurde bereits gezeigt, wie parallele Algorithmen grafisch leicht verständlich dargestellt werden können.

Eine Entwicklungsumgebung für parallele Programme sollte diese grafische Entwurfstechnik berücksichtigen und entsprechende Werkzeuge zur Verfügung stellen. Wichtig ist es dabei, weiter das grafische Abbild des Algorithmus im Gedächtnis halten zu können und eine Entwicklungsumgebung zu finden, welche diese grafischen Abbilder aufgreift und in maschinenlesbaren Code umsetzt.

Ermöglicht werden kann dies, indem für bestimmte Programmieraufgaben Dialogboxen mit grafischen Eingabemöglichkeiten zur Verfügung gestellt werden (vergleiche Kapitel 6.3.3). Die Dialogboxen müssen an die Eigenschaften eines MasPar-Rechners angepasst sein. So muss es möglich sein, einfach bestimmte Zeilen, Spalten oder Bereiche aktivieren zu können.

Beim Odd-Even-Sort-Algorithmus von Zeilen beispielsweise muss jede zweite Spalte aktiviert werden, damit sie gegebenenfalls ihre Daten mit den Nachbarn tauschen können. Allerdings sollte die Aktivierung nach jedem Schritt alternieren, damit der Algorithmus problemlos funktionieren kann. Möchte man nicht auf dem kompletten Prozessorfeld eine Sortierung durchführen, muss zusätzlich der aktive Bereich angegeben werden.

Somit müssen Felder vorhanden sein, in denen angegeben werden kann, wie groß der Abstand zwischen zwei zu aktivierenden Zeilen/Spalten ist und ab welcher Zeile/Spalte mit der Aktivierung begonnen werden soll. Jede Eingabe wird dabei aus Gründen der

Erwartungskonformität sofort in der entsprechenden Abbildung der Prozessorfelder grafisch umgesetzt. Dabei ist zu überlegen, ob man Zeilen, Spalten und Flächen in getrennten Feldern darstellt, um eine bessere Übersicht zu erhalten und/oder alle Informationen in ein grafisches Prozessorfeld einfliesen lässt. Die getrennte Darstellung hat den Vorteil, dass man Zeilen oder Spalten sehr einfach wieder erkennen kann, allerdings entspricht das jeweilige Ergebnis bei der Kombination von Aktivierungsmöglichkeiten nicht der Realität. Die Gesamtdarstellung kann allerdings unübersichtlich werden, so dass auch hier Nachteile bestehen.

Bei der Bereitstellung der grafischen Tools ist zu berücksichtigen, dass nicht jede Angabe von Abständen ohne weiteres grafisch dargestellt werden kann. Gibt man beispielsweise als Abstand die Variable `a` ein, so kann sich dieser zur Laufzeit des Programms beliebig ändern. In diesen Fällen sollte eine Grafik nicht einen festen Abstand andeuten, sondern muss diesen variablen Abstand wiedergeben, indem dieser mit einem symbolischen Wert in der Grafik dargestellt wird. Abbildung 5-5 zeigt, wie ein Prozessorfeld, in dem jede dritte Spalte aktiv ist korrekt dargestellt werden kann, ist dagegen der Abstand zwischen zwei Spalten über eine Variable `a` definiert, so wird dieser Abstand durch beschriftete Pfeile symbolisiert.

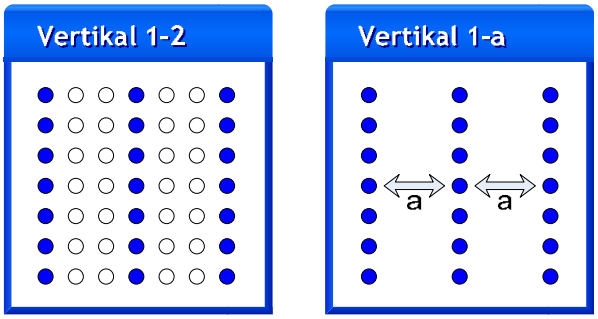


Abbildung 5-5: Unterschiedliche Darstellungen von Aktivierungen

Um dem Programmierer die volle Kontrolle über sein Programm zu lassen, ist die Anzeige des aktuell erzeugten Quelltextes jederzeit notwendig. Dabei sollte dem Benutzer auch die Freiheit gelassen werden, in dieser Quelltextanzeige Änderungen vornehmen zu können. Dabei muss darauf geachtet werden, dass jede Änderung ebenfalls in die grafische Darstellung der Aktivierung übernommen wird.

Bei der Analyse der am Institut für Angewandte Informatik und Formale Beschreibungsverfahren durchgeführten Diplomarbeiten konnte festgestellt werden, dass ein Programmierer an vielen verschiedenen Stellen des Algorithmus immer wieder die gleichen Muster nutzt, um Prozessorelemente zu aktivieren bzw. deaktivieren. Dabei muss es sich nicht immer um einfache Zeilen-, Spalten oder Flächenaktivierungen handeln, sondern es können beliebige Kombinationen sein. Aus diesem Grund ist es sinnvoll, dass der Benutzer nicht jedes mal alle notwendigen Werte eingeben muss, sondern bestimmte häufig definierte Kombinationen unter einem frei wählbaren Namen abspeichern kann. Anschließend kann diese Kombination schnell wieder aufgerufen werden. Unterstützt man den Anwender weiter, indem er noch Tastenkombinationen für seine Kombinationen vergeben kann, so kann eine spätere Wiederbenutzung einer Aktivie-

rung durch wenige Tastendrücke realisiert werden. Durch den Wiederaufruf selbst definierter Kombinationen kann dem Anwender viel „monotones Geklicke“ abgenommen werden, so dass auch bei professionellen Nutzern die Akzeptanz der Dialogboxen hoch bleibt, da die Arbeit nicht nur erleichtert, sondern auch beschleunigt werden kann.

Neben den selbst definierten Aktivierungskombinationen kann der Benutzer bereits von Beginn an durch besondere „Standardformen“ unterstützt werden, die sich normalerweise nur durch genauere Kenntnis von speziellen Rechnerkonstanten realisieren lassen. Dies sind zum Beispiel Diagonalen, mehrere gleichförmige Flächen gleichzeitig oder nicht ganz regelmäßig angeordnete Zeilen oder Spalten (siehe Abbildung 5-6).

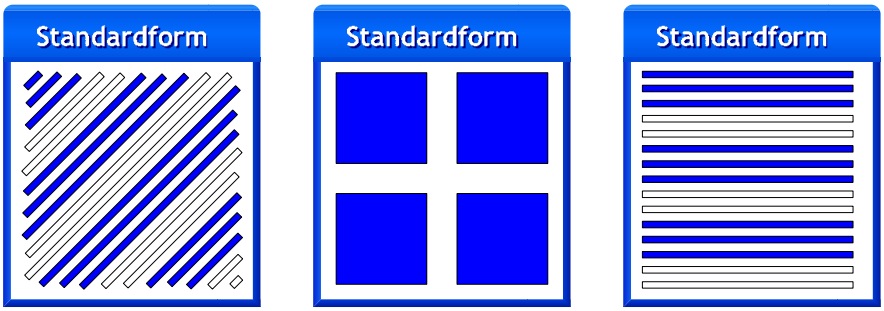


Abbildung 5-6: Standardformen zur Aktivierung von Prozessorelementen

Diese Formen können nicht ohne weiteres aus der allgemeinen Dialogbox durch Zeilen, Spalten und Flächenaktivierung kombiniert werden. Deshalb wird vorgeschlagen, für diese Standardformen speziell angepasste Eingabefelder anzubieten, welche die erforderlichen Werte abfragen. Bei einer Diagonalen kann zum Beispiel gewählt werden, wie breit der aktive Bereich sein soll und wie groß der Abstand zwischen zwei Streifen liegen soll.

Mit Hilfe dieser Dialogboxen lassen sich somit Prozessorelemente auf vielfältige Weise aktivieren, der benötigte Quelltext wird automatisch erzeugt und in das Programm eingefügt.

Dadurch ist es möglich, einen syntaktisch fehlerfreien Quellcode zu erhalten. Dieser kann zusätzlich dazu genutzt werden, die korrekte Syntax zu erlernen. Bei der weiteren Programmierung ist jedem freigestellt, ob er weiterhin mit den Dialogboxen arbeiten möchte oder ob die direkte Eingabe des Quelltextes gewünscht wird, welche durch Funktionen wie Tool-Tipps und Befehlsergänzung unterstützt werden kann.

5.3.1.1 Übernahme von Quelltext in grafische Dialogboxen

Bis jetzt wurde nur darauf eingegangen, wie man mit Hilfe von Dialogboxen neuen Quelltext erzeugen kann. Mindestens genauso wichtig ist es aber, dass bereits vorhandener Quelltext wieder in Dialogboxen eingelesen und anschließend weiterbearbeitet werden kann.

Am einfachsten kann man Daten in die Dialogbox importieren, wenn man diese Daten bereits genau kennt. Dies kann beispielsweise dadurch der Fall sein, dass der entsprechende Code bereits durch eine Dialogbox erzeugt wurde und bei der Übernahme in

den Quelltext durch geeignete Maßnahmen mit Zusatzinformationen versehen wurde, welche das spätere Einlesen wesentlich vereinfachen.

Diese Methode hat allerdings einige Nachteile. Zum Beispiel kann so nur Text in die Dialogbox geladen werden, der vorher auch damit erzeugt und anschließend nicht geändert wurde. Dies schränkt die Einsatzmöglichkeiten beträchtlich ein. Außerdem ist es immer notwendig, dass neben dem eigentlichen Quelltext weitere Daten entweder als Kommentar oder in zusätzlichen Dateien vorhanden sind, welche die Zusatzinformationen enthalten. Wird ein solcher Quelltext einmal mit einer beliebigen anderen Entwicklungsumgebung bearbeitet, werden diese zusätzlichen Daten nicht automatisch aktualisiert. Bei einem späteren Wechsel auf die ursprüngliche Umgebung mit paralleler Unterstützung ist es fraglich, ob die Zusatzdaten noch auf den veränderten Quelltext anwendbar sind. Würde man diesen Weg der Realisierung wählen, wäre man darauf beschränkt, dass der Quelltext immer nur mit dieser einen Entwicklungsumgebung bearbeitet werden darf.

Aus diesem Grund ist es sinnvoller, alleine aus dem vorhandenen Quelltext genügend Informationen herauszulesen, um eine Dialogbox bei Änderungen mit sinnvollen Anfangswerten zu versorgen. Das heißt, es muss sichergestellt werden, dass der vorhandene Quelltext korrekt in den zur Verfügung gestellten Dialogboxen wiedergegeben wird und nach Abschluss von dort wieder korrekt in den Quelltext übertragen wird.

Deshalb soll nun als nächstes die Konvertierung von Quelltext betrachtet werden, um Daten korrekt für die Aktivierung von Prozessorelementen vom Quelltext in Dialogboxen zu übertragen.

Folgende Fragen müssen dabei beachtet werden:

1. Welche Teile des Quelltextes sind für die Aktivierung von Prozessorelementen relevant?
2. Wie kann man diese Teile im Quelltext erkennen?
3. Können die Teile so ausgewertet werden, dass sie korrekt dargestellt werden?
4. Muss man Sonderfälle beachten?

Die erste Frage lässt sich relativ einfach beantworten: Sollen nicht alle Prozessorelemente arbeiten, muss an irgendeinem Punkt im Programm eine Abfrage gemacht werden, welche Prozessorelemente als nächstes arbeiten sollen. Abfragen lassen sich in C(++) leicht dadurch erkennen, dass Ausdrücke benutzt werden. Diese wiederum können nur nach den C-Befehlen `if`, `switch`, `do`, `while` und `for` stehen. Innerhalb der Ausdrücke müssen die eventuell relevanten Daten stehen.

Somit ist bekannt, welche Teile des Quelltextes für eine Aktivierung infrage kommen. Nun ist zu klären, welche Ausdrücke relevant sind, denn nicht jeder Ausdruck ist gleichzeitig eine Aktivierung von Prozessorelementen.

Diese Relevanz kann nur vorhanden sein, wenn sich der Wert des Ausdrucks auf den jeweiligen Prozessorelementen unterscheiden kann. Damit sich der Wert der Ausdrücke

unterscheiden kann, muss mindestens eine Konstante, Variable oder ein Funktionsergebnis auf unterschiedlichen Prozessorelementen unterschiedliche Werte annehmen können. Dazu muss diese vom so genannten „Plural-Typ“ sein. Auf der MasPar stehen unter anderem folgende „Plural-Typen“ zur Verfügung: `plural int`, `plural long`, `plural double` usw.

Jetzt hat man allerdings die Schwierigkeit, diese Plural–Typen zu erkennen, da die Typ-Definition der genutzten Konstanten oder Variablen an „beliebigen“ Stellen im Quelltext vorgenommen werden kann. Einfach ist es, wenn die Schlüsselkonstanten `iproc`, `ixproc` oder `iyproc` vorkommen, denn diese sind immer vom Typ Plural. In diesen Fällen muss auch der gesamte Ausdruck einen Plural-Wert annehmen, da sonst eine Typenunverträglichkeit vorläge. In fast allen untersuchten Fällen der vorliegenden Arbeiten sind in Ausdrücken, die für die Aktivierung wichtig sind, diese Schlüsselkonstanten vorhanden.

Sollten diese nicht vorkommen, muss zuerst die aktuelle Funktion, danach ein größerer Teil des Quelltextes abgesucht werden, um zu klären, von welchem Typ die einzelnen im zu untersuchenden Ausdruck vorhanden Konstanten, Variablen oder Funktionen sind. Dies kann teilweise sehr aufwändig werden und berücksichtigt man, dass auch Aliasnamen für Typen zulässig sind, ist diese Aufgabe nur mit einem sehr leistungsfähigen Parser zu lösen. Hat man dieses Werkzeug zur Hand, ist die Identifizierung eindeutig möglich.

Im realisierten Prototyp wird vorausgesetzt, dass anhand der oben genannten Schüsselkonstanten ein Aktivierungsprozess erkannt werden kann. Dies ist keine zu große Einschränkung, da Plural-Schlüsselkonstanten sehr häufig eingesetzt werden. In den seltenen Fällen, in denen keine Schlüsselkonstante vorhanden ist, wird eine Abfrage nicht als Aktivierungsabfrage erkannt und sie wird als normaler Text dargestellt. Auf die Lauffähigkeit des Programms hat dies keinen Einfluss.

Geht man davon aus, dass die Abfrage als Aktivierungsabfrage erkannt wurde, muss sie anschließend ausgewertet werden, um in der Dialogbox entsprechende Zuweisungen durchführen zu können. Die Quelltextzeile muss dabei so eindeutig zerlegt werden können, dass einzelne Teile den jeweiligen Eingabefeldern der Dialogbox zugeordnet werden können. Die Frage ist dann zum Beispiel: Welcher Teil des Ausdrucks ist dafür zuständig, dass die x–te Zeile oder y–Spalte aktiviert werden soll?

Aus diesem Grund muss die Reihenfolge genauer untersucht werden, in der Befehle und Ausdrücke in C(++) abgearbeitet werden. Tabelle 5-1 gibt einen Überblick über die Reihenfolge, in der die Operatoren innerhalb eines Ausdrucks ausgewertet werden.

Tabelle 5-1: Rang von Operatoren in C++

# Rang	Operator	Beschreibung	Ord.
1.	`()`	Funktionsaufruf	→
	`[]`	Array-Subskript	
	`->`	Indirekte Komponentenauswahl in C++	
	`::`	Gültigkeitsbereichszugriff und Zugriffsauflösung in	

# Rang	Operator	Beschreibung	Ord.
		C++	
	.	Direkte Komponentenauswahl in C++	
2. Unär	!	Logische Negation (NOT)	←
	~	Bitweises Komplement	
	+	Unäres Plus	
	-	Unäres Minus	
	++	Prä- oder Post-Inkrementierung	
	--	Prä- oder Post-Dekrementierung	
	&	Adresse	
	*	Umleitung	
	sizeof	Gibt die Größe des Operanden in Bytes zurück	
	new	Dynamische Speicherzuweisung in C++	
	delete	Dynamische Speicherfreigabe in C++	
3. Zugriff auf Klassenelemente	.*	C++ Dereferenzierung	→
	->*	C++ Dereferenzierung	
4. Multiplikativ	*	Multiplikation	→
	/	Division	
	%	Modulo Rest	
5. Additiv	+	Binäres Plus	→
	-	Binäres Minus	
6. Shift	<<	Shift links	→
	>>	Shift rechts	
7. Relational	<	Kleiner	→
	<=	Kleiner gleich	
	>	Größer	
	>=	Größer gleich	
8. Gleichheit	==	Gleich	→
	!=	Ungleich	
9.	&	Bitweises UND	→
10.	^	Bitweises XOR	→
11.	|	Bitweises ODER	→
12.	&&	Logisches UND	→
13.	||	Logisches ODER	→
14. Bedingung	?	(a ? x : y bedeutet ("wenn a dann x, sonst y")	←
15. Zuweisung	=	Einfache Zuweisung	←
	*=	Produkt zuweisen	
	/=	Quotient zuweisen	
	%=	Rest zuweisen (Modulo)	
	+=	Summe zuweisen	
	-=	Differenz zuweisen	
	&=	Bitweises UND zuweisen	
	^=	Bitweises XOR zuweisen	
	|=	Bitweises ODER zuweisen	
	<<=	Linksschieben zuweisen	

# Rang	Operator	Beschreibung	Ord.
	>>=	Rechtsschieben zuweisen	
16. Komma	,	Auswerten	→

In einem Ausdruck werden zuerst alle Komponenten vom Rang 1 gesucht und ausgewertet, das sind z.B. alle Funktionsaufrufe. Danach werden Ausdrücke vom Rang 2 ausgewertet, usw. Müssen mehrere Ausdrücke vom gleichen Rang ausgewertet werden, geschieht das im Normalfall von links nach rechts, wobei unterschiedliche Operatoren der gleichen Rangstufe gleich behandelt werden. Ausnahmen sind die Ränge 2, 14 und 15, dort wird innerhalb eines Ausdrucks von rechts nach links ausgewertet. Die vielfältige Unterscheidung in 16 verschiedene Ränge wird nur in C(++) durchgeführt. Andere Sprachen begnügen sich mit fünf bis acht Rängen.

Für die Auswertung, ob man einen entsprechenden Teilausdruck zur Zeilen–, Spalten- oder Flächenaktivierung benutzen kann, sind besonders die Ränge 12 und 13 interessant. Steht in einem Teilausdruck, der durch ein „logisches „`und`" oder „`oder`" von anderen Teilausdrücken getrennt ist, die Konstante „`ixproc`" und ein „==", werden Prozessorelemente in vertikaler Ordnung aktiviert. Heißt ein Teilausdruck beispielsweise „`ixproc == 1`" so bedeutet das, dass die erste Spalte aktiviert wird.

Auf die gleiche Weise kann man durch die Texte „`iyproc`" und „==" horizontal angeordnete Prozessorelemente aktivieren. Werden statt „==" Ungleichungen (<, >, <=, >=) abgefragt, bedeutet das, dass Flächen aktiviert werden sollen.

Wertet man die Teilausdrücke der Reihe nach aus, können jeweils Zuordnungen zu Zeilen, Spalten und Flächen vorgenommen werden.

Eine direkte Zuordnung zu Zeilen, Spalten oder Flächen ist nicht möglich, wenn in einem Teilausdruck „`ixproc`" und „`iyproc`" gleichzeitig vorkommen (bsw. `ixproc %a == iyproc %b`). Dieser Ausdruck aktiviert in Diagonalen (und Unterdiagonalen) angeordnete Prozessorelemente und kann somit weder Zeilen, Spalten noch Flächen zugeordnet werden. Einige dieser Fälle können durch Sonderabfragen erkannt und dementsprechend behandelt werden. Es bleibt allerdings noch eine Reihe an Möglichkeiten übrig, die nicht direkt den genannten Kategorien zugeordnet werden können. Diese Teilausdrücke müssen trotzdem zumindest in der Ergebniszeile korrekt wiedergegeben werden. Eine korrekte grafische Darstellung dieser Teilausdrücke ist allerdings ohne aufwändigere Parserfunktionen und Simulationen nicht möglich.

Trotz der genannten Ausnahmen können in fast allen Fällen die in vorhandenen Quelltexten erkannten Aktivierungsprozesse korrekt dargestellt werden, da zum einen nur wenige Aktivierungen ohne Schlüsselkonstanten vorhanden sind, zum anderen innerhalb einer logischen Verknüpfung fast immer nur eine Schüsselkonstante auftritt.

Die zu Beginn des Unterkapitels aufgeworfene Frage, ob und wie aktivierender Quellcode erkannt werden kann, konnte in den letzten Absätzen mittels einiger Ideenskizzen beantwortet werden.

5.3.1.2 Darstellung durch Icons im Quelltext und Abgrenzung durch Hilfslinien

Sinn der in dieser Arbeit behandelten Dialogboxen ist, dass man durch die Eingaben Quelltext generieren kann, der anschließend in das Programm übernommen wird. Für die Übernahme sind zwei Alternativen vorgesehen.

Zum einen kann im Programmfenster der korrekte und vollständige erzeugte Quelltext angezeigt werden. Er wird so angezeigt, als ob er von Hand eingegeben wurde. Diesen kann man analysieren und später in ähnlichen Fällen ohne Zuhilfenahme der Dialogbox die Prozessorelement-Aktivierung vornehmen, womit die Lernförderlichkeit gestärkt wird. Auch kleinere Änderungen lassen sich auf diese Weise leicht und schnell vornehmen.

Eine zweite Möglichkeit besteht darin, dass man den für eine Aktivierung erzeugten Quelltext als fertiges Ergebnis ansieht und deshalb eine symbolische Darstellung wählt, die klar anzeigt, dass hier eine Aktivierung stattfindet. Dadurch kann der Quelltext vor ungewollten Änderungen geschützt und zusätzlich durch die eingefügten grafischen Elemente klarer strukturiert werden. Möchte man später Änderungen an der Aktivierung vornehmen, genügt ein Doppelklick auf die Darstellung, um die notwendige Dialogbox zu öffnen.

Aus Gründen der Übersichtlichkeit und Sicherheit ist es deshalb häufig sinnvoll, wenn statt des Quelltextes nur ein Symbol bzw. Icon angezeigt wird. Dieses muss allerdings so aussagekräftig sein, dass erkennbar wird, was an der jeweiligen Stelle realisiert werden soll. Diese Aussagekraft kann dadurch erreicht werden, dass jedes Icon individuell durch die Daten aus der Dialogbox generiert wird. Ist eine Auswertung aus Gründen wie in Kapitel 5.3.1.1 beschrieben nicht vollständig möglich, wird ein Icon benutzt, das der Realität so weit wie möglich entspricht und evtl. zusätzliche Beschreibungen enthält.

Des Weiteren kann der Tool-Tipp so eingesetzt werden, dass, wenn die Maus über das Icon fährt, der vollständige Quelltext eingeblendet wird. Abbildung 5-7 zeigt zwei Beispiele möglicher Icondarstellungen im Quelltext.

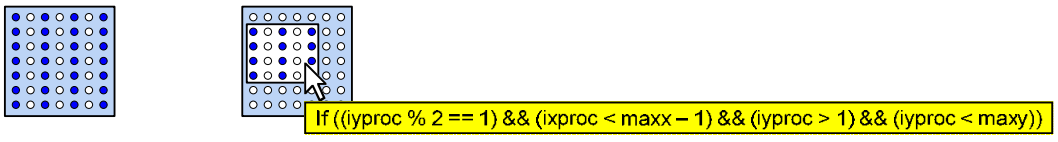


Abbildung 5-7: Icondarstellungen von Aktivierungen im Quelltext

Denkbar ist ebenfalls, dass eine voreingestellte Tastenkombination das Icon in eine Quelltextzeile umwandelt, damit ohne den Umweg über die Dialogbox Änderungen durchgeführt werden können.

Ein weiteres Hilfsmittel, das in diesem Zusammenhang eingeführt werden kann, sind Hilfslinien. Dies wurde bereits in [Tou96] vorgeschlagen. Damit kann zusätzlich visualisiert werden, in welchem Bereich eine Aktivierung gültig ist. Bei der konventionellen Programmierung werden zu diesem Zweck zusammengehörende Textteile entsprechend eingerückt. Diesen Effekt der Strukturierung kann man durch die umschließenden Hilfslinien verstärken. Sie sind nicht nur bei der Aktivierung sinnvoll, sondern immer

dann, wenn zusammenhängende Textteile über mehrere Zeilen programmiert werden. Programmtechnisch gesprochen sind diese Hilfslinien immer dann sinnvoll, wenn eine öffnende {-Klammer und eine schließende }-Klammer nicht in der gleichen Zeile stehen (was nach Richtlinien der übersichtlichen Programmierung der Regelfall ein sollte) [Str89]. Das gleiche Hilfsmittel wird von einigen Programmierumgebungen eingesetzt, um das Falten kompletter Programmteile zu unterstützen (siehe Kapitel 5.1.7).

5.3.2 Hilfe bei der parallelen Kommunikation

Die bisher vorgestellte Aktivierung von Prozessorelementen zeigt, wie man auf grafische Weise einfach bestimmte Prozessorelemente auswählen kann.

Nachdem die aktiven Prozessorelemente ausgewählt wurden, kann auf diesen eine Kommunikation initiiert werden. Jedes aktive Prozessorelement fordert dabei von einem anderen Prozessorelement Daten an oder verschickt es an dieses.

Die Kommunikation kann grafisch wie in Abbildung 5-8 dargestellt werden. Die linke Darstellung ist für eine Anwendung im Programmierumfeld weniger geeignet, da zu viel Platz auf dem Monitor benötigt wird und dieser Mehrverbrauch an Platz nicht durch eine Mehrinformation aufgewogen wird. Die rechte Darstellung dagegen symbolisiert die Darstellung der Prozessorelemente ausreichend und beschränkt die Darstellung auf das Wesentliche.

Abbildung 5-8: Darstellung der Kommunikation zwischen Prozessorelementen

Bei der herkömmlichen Programmierung hat man zwar dieses Bild der Kommunikation vor Augen, muss aber davon abstrahieren und zeilenweisen Quelltext schreiben. Viel angenehmer ist es, wenn man die grafische Abbildung, die man als Skizze des Algorithmus im Sinn hat, direkt am Bildschirm wieder finden und bearbeiten kann. Dazu ist es notwendig, dass es eine Eingabemöglichkeit gibt, welche das Kommunikationsnetz der Prozessorelemente darstellt. Dabei sollen durch einfache Mausklicks oder Tastenkombinationen gewünschte Kommunikationsstrukturen erstellt werden können.

Das Kommunikationsnetz eines massivparallelen Rechners kann beispielsweise dem XNet aus Abbildung 3-2 entsprechen. Dabei wird in der Mitte das aktive Prozessorelement farblich hervorgehoben dargestellt und strahlenförmig werden in allen Richtungen die Nachbarn platziert. Eine durchzuführende Kommunikation wird dabei immer von dem in der Mitte platzierten aktiven Prozessorelement initiiert. Durch die Eigenschaft

eines SIMD-Rechners kann durch die Darstellung eines einzigen aktiven Prozessorelementes die Kommunikation für den kompletten Parallelrechner symbolisiert werden, da alle anderen aktiven Elemente durch den gleichen Befehl angesprochen werden.

Um das aktive Element können die acht direkten Nachbarn dargestellt werden, mit denen eine Kommunikation stattfinden kann. Wie in Kapitel 3.3 beschrieben, kann aber nicht nur mit den acht unmittelbaren Nachbarn schnell kommuniziert werden, sondern auch mit allen weiter entfernt liegenden Prozessorelementen, solange sie über eine der Himmelsrichtungen N, NW, W, SW, S, SO, O, NO erreicht werden können. Diese weiter entfernt liegenden Prozessorelemente müssen somit auch dargestellt werden. Da aber auf einem Bildschirm aus Platzgründen nicht alle Prozessorelemente sinnvoll dargestellt werden können, muss man sich für eine geeignete Auswahl entscheiden.

Auswahl der dargestellten Nachbarschaft

Bei der Analyse von parallelen Quelltexten hat sich gezeigt, dass in den allermeisten Fällen entweder ein direkter Nachbar oder maximal ein weiter entfernt liegender Nachbar je Richtung zur Kommunikation genutzt wurde. Somit fällt die Auswahl auf jeweils zwei Prozessorelemente je Richtung. Um den größeren Abstand des jeweils zweiten Prozessorelementes darstellen zu können, nutzt man die in Kapitel 2.4 beschriebenen Gestaltgesetze, genauer die Gesetze der Nähe und der guten Fortsetzung. Die unmittelbaren Nachbarn werden direkt um das aktive Prozessorelement angezeigt, während zu den weiter entfernt liegenden Nachbarn absichtlich ein größerer Abstand vorgesehen wird.

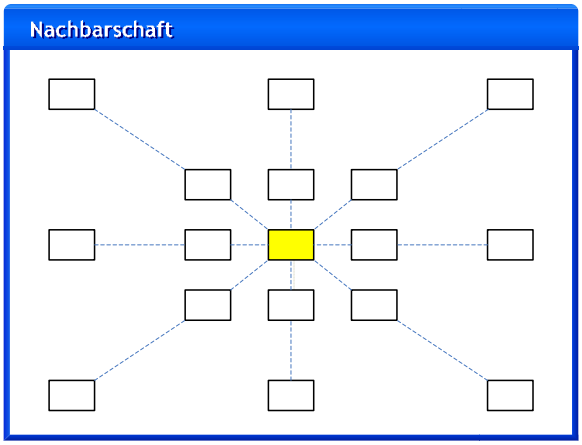


Abbildung 5-9: Auswahl an Prozessorelementen, die an einer schnellen Kommunikation teilnehmen können

Abbildung 5-9 zeigt eine Auswahl an Prozessorelementen, welche als grafische Darstellung für die Erstellung einer Kommunikation angeboten werden. Allerdings fehlen noch Informationen, mit welchen Prozessorelementen das aktive Element kommuniziert, bzw. welche Daten bei der Kommunikation ausgetauscht werden sollen.

Darstellung der Kommunikation

Ein Datenaustausch wird bei allen Modellen durch Pfeile symbolisiert. Dieses Mittel kann aufgegriffen werden. Um dem Anwender alle möglichen Kommunikationsformen anzeigen zu können, kann zwischen allen Prozessorelementen, die aktuell eine Kommunikation erlauben, ein schattierter Pfeil dargestellt werden. Beim eingesetzten XNet werden somit von allen 16 Nachbarn schattierte Pfeile in Richtung aktives Element gezogen. Zwischen den Nachbarn ist eine direkte Kommunikation nicht möglich.

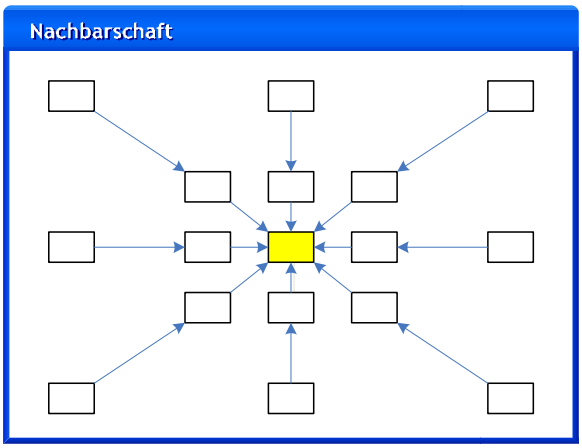


Abbildung 5-10: Darstellung der möglichen Kommunikationen

In Abbildung 5-10 führen die äußeren Pfeile jeweils zum inneren Nachbarn, obwohl diese korrekterweise direkt zum aktiven Prozessorelement in der Mitte führen müssten. Die direkte Verbindung könnte in der Darstellung leicht über gebogene Pfeile symbolisiert werden. Trotzdem hat sich der Autor für die indirekte Darstellung entschieden, da dadurch zum einen das „Gesetz der guten Fortsetzung“ besser zur Geltung kommt und zum anderen besser auf die weiter unten beschriebenen „`copy`“- und „`pipeline`“-Kommunikationen hingewiesen werden kann, bei denen die auf dem Weg befindlichen Prozessorelemente einen Einfluss auf das Ergebnis nehmen können.

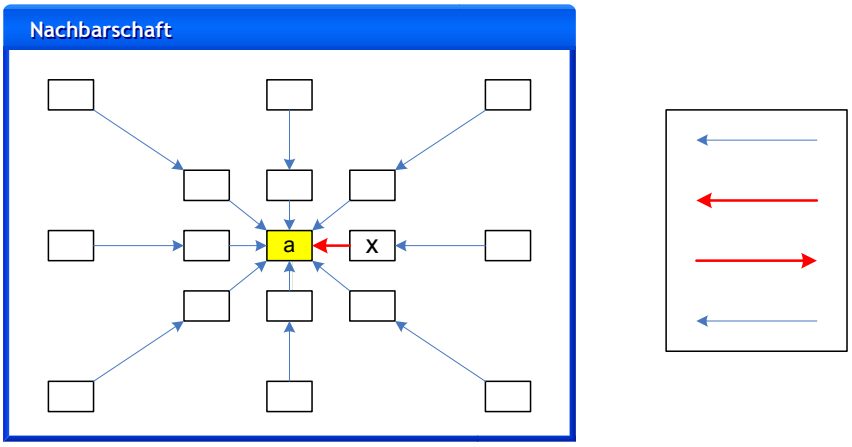


Abbildung 5-11: Darstellung einer aktiven Kommunikation

Um dem Anwender die Kommunikation intuitiv zu ermöglichen, sollten die einzelnen Pfeile leicht aktiviert werden können. Dies kann beispielsweise dadurch realisiert

werden, indem durch einen Klick auf einen Pfeil die Schattierung ersetzt wird durch einen kräftigen farbigen Pfeil. Ein weiterer Klick ändert die Richtung des Pfeils und ein dritter Klick deaktiviert den Pfeil wieder. Bei weiteren Klicks wiederholt sich das Vorgehen. Auf der rechten Seite von Abbildung 5-11 wird die Folge der Pfeilvarianten angezeigt, die linke Seite zeigt die Darstellung einer Kommunikation.

Achtet man auf eine barrierefreie Darstellung, so sollte neben der Farbunterscheidung der Pfeil zusätzlich bei einer Aktivierung dicker angezeigt werden. So können auch Personen, welche die verschiedenen Farben nicht erkennen können, einen Unterschied in der Darstellung feststellen.

Die Aktivierung und Deaktivierung kann für alle Pfeile einzeln durchgeführt werden, so dass dadurch fast alle Kommunikationen auf einem XNet dargestellt werden können. Zur Unterstützung sollte im Hintergrund zusätzlich ein Algorithmus prüfen, ob die aktuell beschriebene Kommunikation überhaupt zulässig ist. Dabei ist beispielsweise zu beachten, dass ein aktives Prozessorelement in einer Anweisung von sehr vielen Nachbarn Informationen erhalten kann, allerdings darf es maximal eine Information zu einem Nachbarn (oder in Spezialfällen zu einer bestimmten Folge von Nachbarn) verschicken.

Da nur ein „entfernter“ Nachbar je Richtung dargestellt wird, muss zusätzlich eine Möglichkeit zur Verfügung gestellt werden, durch die zu erkennen ist, wie weit dieser Nachbar entfernt sein soll. Dies kann dadurch geschehen, indem bei den äußeren Pfeilen bei der Aktivierung oder später bei einem Rechtsklick ein Eingabefeld erscheint, das eine Entfernungsangabe aufnimmt.

Da die Funktionsweise der Pfeile zwar bei entsprechender Kenntnis intuitiv ist, aber der Programmierer evtl. nicht auf die Idee kommt, auf die Pfeile mit der linken oder rechten Maustaste zu klicken, wird als Hilfsmittel ein Tool-Tipp eingesetzt. Er erscheint, wenn die Maus über einen Pfeil bewegt wird.

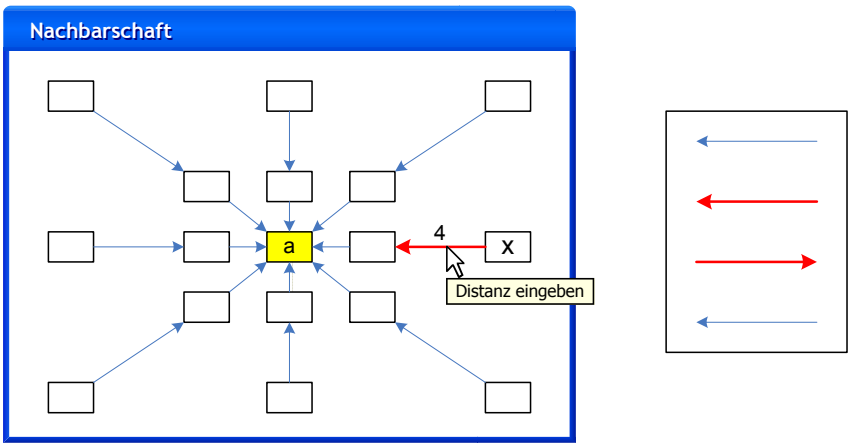


Abbildung 5-12: Kommunikation mit Entfernungsangabe und Tool-Tipp

Abbildung 5-12 zeigt, wie der Inhalt der Variablen `x` eines rechten Nachbars mit einem Abstand von 4 zum aktiven Element in die Variable `a` des aktiven Elementes übertragen wird.

Festlegung der zu übertragenden Variablen

Durch die Pfeile wird festgelegt, in welche Richtung kommuniziert werden kann, allerdings ist noch nicht klar, welche Daten übertragen werden sollen. Auch hier kann man eine direkte Eingabe in der Darstellung vorsehen, wie dies bereits in den beiden letzten Abbildungen geschehen ist. Dabei muss für das Feld des Empfängers aus syntaktischen Gründen eine Variable angegeben werden, bei den Absendern und Entfernungsangaben sind zusätzlich Konstanten denkbar. Das aktive Prozessorelement darf den Ausdruck unabhängig davon, ob es Sender oder Empfänger ist, neben Variablen und Konstanten auch um Funktionen erweitern.

Auswahl der Kommunikationsart

Betrachtet man das XNet des MasPar-Rechners, so bietet es drei unterschiedliche Arten der Kommunikation an. Da die Arten nur bei einer entfernten Kommunikation eine Rolle spielen und in der Abbildung nur jeweils ein entferntes Prozessorelement dargestellt wird, kann der unmittelbare Einfluss bei der komprimierten Darstellung schlecht dargestellt werden. Deshalb sollte eine Möglichkeit geboten werden, die unterschiedlichen Kommunikationsarten durch Drop-Down oder Radio-Button-Felder direkt neben der Abbildung anzuzeigen. Welche benutzt werden soll, kann durch Anklicken der entsprechenden Option entschieden werden.

Die erste Kommunikationsart ist der „normale“ Datenaustausch zwischen zwei Prozessorelementen. Die Daten werden wie eingetragen übertragen.

Bei der zweiten Art, der „Pipeline“, gibt es eine Einschränkung. Sollen Daten zwischen weiter entfernt liegenden Prozessorelementen übertragen werden, kann es vorkommen, dass zwischen Quelle und Ziel ein aktives Prozessorelement liegt. Ist so ein aktives Element auf dem Weg vorhanden, wird die Kommunikation nicht ausgeführt. Der Grund dafür liegt darin, dass versucht wird eine „Direktschaltung“ mittels Pipeline zwischen den kommunizierenden Prozessorelementen aufzubauen. Ist das nicht möglich, wird die Kommunikation gar nicht durchgeführt.

Bei der dritten Option „Kopieren bis maximal zum nächsten aktiven Prozessorelement“ werden die Daten nicht nur in die entsprechende Variable des Ziel-Prozessorelementes geschrieben, sondern in die Zielvariable auf jedem Prozessorelement, das zwischen Quelle und Ziel liegt. Hierbei ist zusätzlich zu beachten, dass das Kopieren des Wertes beim ersten aktiven Element endet, das auf der Verbindung liegt. Dadurch wird vermieden, dass einzelne Elemente mehrfach überschrieben werden. Allerdings kommt es dadurch auch vor, dass Daten nicht bis zur erwarteten Entfernung kopiert werden.

Um die unterschiedlichen Kommunikationsarten zu erklären, sollte jeweils ein Tool-Tipp eine Kurzinformation anzeigen, beispielsweise „Standardkommunikation“, „schnelle Pipeline, falls kein aktives PE im Weg“ oder „Kopie bis zum nächsten aktiven Prozessorelement“. Zusätzlich müssen auf einer extra Hilfeseite die Unterschiede der Kommunikationsarten symbolisiert und durch weiteren Text und Beispiele erläutert werden, da

ohne Beispiele und Visualisierungen diese Kommunikationsarten schwer zu verstehen sind.

Dynamische Darstellung der Nachbarschaft

Durch die starre Darstellung eines aktiven Prozessorelementes mit 16 Nachbarn benötigt man viel Platz für die Darstellung. Wählt man eine dynamische Darstellung der Nachbarn, so können beispielsweise in eine Richtung wesentlich mehr Nachbarn darstellt werden, indem auf der gegenüberliegenden Seite keine Nachbarn dargestellt werden. Der Platzbedarf bleibt dabei vergleichbar groß. Die Herausforderung besteht darin, einerseits den Anwender durch die unterschiedlichen Darstellungen nicht zu verwirren, andererseits eine für die durchzuführende Kommunikation sinnvolle Nachbarschaft anzuzeigen. Ein Vorschlag ist, dass zu Beginn eine Dialogbox mit 16 Nachbarn dargestellt wird. Zusätzlich befinden sich an den Außenrändern der Abbildung „Schiebepfeile“. Mit diesen kann die Ansicht auf das Prozessorfeld verschoben werden, wie man dies bei Darstellungen von Landkarten kennt. Dadurch kann das aktive Prozessorelement bis zum Rand der Abbildung geschoben werden und auf der entgegen gesetzten Seite steht genügend Platz zur Verfügung, um mehrere Prozessorelemente einer Kommunikationsrichtung darstellen zu können.

Nachteil der dynamischen Darstellung ist, dass die Erwartungskonformität nicht mehr voll erfüllt werden kann, da die Abbildung immer unterschiedlich aussehen kann. Dafür spricht aber, dass auch die seltenen Mehrfachkommunikationen in einer Richtung sehr gut dargestellt werden können.

Neben der Zuweisung von Daten kommt es auch vor, dass Daten nur verglichen werden sollen. Die Kommunikation ist dabei genauso notwendig, da nur ein Prozessorelement den Vergleich durchführen kann und somit der Transfer auf jeden Fall durchgeführt wird, auch wenn das Element anschließend verworfen werden muss. Somit kann die gleiche Eingabemaske wie beim Datenaustausch genutzt werden, wenn diese um ein Auswahlfeld erweitert wird, das entscheidet, ob ein Vergleich oder eine Zuweisung ausgeführt werden soll.

Mehrfachkommunikation

Aktiviert man mehrere Pfeile, so findet nicht nur eine Kommunikation statt, sondern mehrere. Das Ergebnis wird aber immer in einer Zielvariablen gespeichert, so dass Verknüpfungen bzw. Operatoren notwenig werden. Beispielsweise möchte man `a=ost(b)+west(c)-süd(d)*west(e)/nord(f)` schreiben. Die unterschiedlichen Operatoren könnten zwar bei der jeweiligen Kommunikation dargestellt werden, aber nicht bei allen Operatoren gilt das Kommutativgesetz, so dass die grafische Darstellung der Kommunikation nicht mehr eindeutig ist. Deshalb ist es sehr wichtig, dass zusätzlich zur symbolischen Darstellung gleichzeitig die fertig erzeugte Quelltextzeile angezeigt wird, in der die jeweiligen Operatoren, Klammerungen oder Konstanten zusätzlich eingetragen werden können. Somit erhält die Visualisierung die Aufgabe die Kommuni-

kationspartner darzustellen, kann in einigen Fällen die Kommunikation selbst allerdings nicht vollständig repräsentieren.

Wird für alle Kommunikationen ein einheitlicher Operator gewünscht, kann dafür ein Drop-Down-Feld oder ähnliches zur Verfügung gestellt werden. Somit muss der unerfahrene Benutzer keinerlei Änderungen im Quelltext durchführen.

Werden in der Quelltextzeile Daten geändert, die für die Abbildung der Kommunikation relevant sind, so muss die Abbildung sofort aktualisiert werden. Genauso müssen Änderungen in der Abbildung sofort auf die Quelltextzeile Einfluss haben, um inkonsistente Darstellungen zu vermeiden.

Komplexe Funktionen mit mehreren Kommunikationen innerhalb einer Anweisung, wie gerade beschrieben, sollten allerdings immer nur mit Bedacht eingesetzt werden, denn auch hier müssen alle Variablen nacheinander über das Bussystem übertragen werden. Sollte später die gleiche Variable noch einmal benötigt werden, ist eine zweite Kommunikation notwendig, da keine lokale Zwischenspeicherung stattfindet. Da aber lokale Operationen wesentlich schneller sind als die Kommunikation zwischen Prozessorelementen, sollten Werte in einer lokalen Variablen zwischengespeichert werden, sobald sie mindestens zweimal benötigt werden, um den erneuten Kommunikationsaufwand zu sparen. Die Konsequenz für den Programmierer sollte sein, dass er in den allermeisten Fällen innerhalb einer Zuweisung nur eine oder sehr wenige Kommunikationen nutzt und diese eventuell mit lokalen Operationen verknüpft werden. Die vorgestellte Dynamisierung der Darstellung ist daher zwar in einigen Situationen sinnvoll, aber nicht unbedingt notwendig, da selten mehrere Kommunikationen in eine Richtung ohne lokale Zwischenspeicherung benötigt werden. Somit ist ein weiterer Vorteil des konsequenten Einsatzes der Dialogbox, Mehrfachkommunikationen zu vermeiden und dadurch die Performance des Programms zu steigern.

Trotz einer überschaubaren Anzahl an Kommunikationen innerhalb einer Zuweisung findet man immer wieder Kommunikationsmuster, die häufig benötigt werden. Diese kann man über vorgefertigte Buttons aufrufen und anschließend die Variablen den aktuellen Erfordernissen anpassen. Das Vorgehen ist dabei das gleiche wie bei der Aktivierung von Prozessorelementen.

Durch den Charakter eines SIMD-Rechners wird diese Kommunikation, wie bereits weiter oben erwähnt, auf allen gerade aktiven Prozessorelementen angewandt. Die weiteren aktivierten Prozessorelemente könnten sich auch in der abgebildeten Umgebung befinden, so dass die Darstellung mit nur einem aktiven Prozessorelement nicht immer der Realität entspricht. Durch eine gezielte Auswertung aller Prozessoraktivierungen kann man diese weiteren aktiven Elemente erkennen. Deshalb sollten sie auch in der Dialogbox entsprechend aktiv gekennzeichnet und die Kommunikation auch für diese angezeigt werden.

5.3.2.1 Übernahme von Quelltext in grafischen Dialogboxen

Genau wie bei der Aktivierung von Prozessorelementen muss bei der Übernahme von Quelltext in die Dialogbox überprüft werden, welche Daten übertragen werden müssen, wie man sie erkennt und ob sie korrekt dargestellt werden können. Auch ist zu überprüfen, ob Sonderfälle vorliegen, die einer gesonderten Behandlung bedürfen.

Der einfachste Fall ist, wenn die Dialogbox ohne Kontext aufgerufen wird. Das heißt, der Cursor stand beim Aufruf auf einer leeren Pogrammzeile. In diesem Fall müssen keine Daten übernommen werden und es kann mit einer Dialogbox begonnen werden, deren Eingabefelder leer sind.

Für Kommunikation, wie sie hier vorgestellt wurde, sind die `xnet`-Befehle zuständig. Die Syntax für die `xnet`-Befehle lautet `xnet[entfernung].variable`, wobei `xnet` stellvertretend für einen der 24 Kommunikationsbefehle steht, die in Tabelle 3-2 aufgeführt wurden. Je nach Kommunikationsmethode und -richtung müssen im Quelltext die Befehle `xnetN`, `xnetNE`, `xnetE`, `xnetSE`, `xnetS`, `xnetSW`, `xnetW`, und `xnetNW` beziehungsweise die Varianten `xnetc` und `xnetp` mit den Ergänzungen für die acht Himmelsrichtungen vorkommen.

Da die Befehle die Abkürzung für die Himmelsrichtung enthalten, können sie direkt einem Textfeld zugeordnet werden. Auch die Entfernung kann direkt übertragen werden, da sie in eckiger Klammer angegeben werden muss. In jedem Fall kann durch einfache Quelltextuntersuchungen entschieden werden, welche Konstanten oder Variablen auf welchem Prozessorelement angesprochen werden und in welche Richtung die Kommunikation stattfindet. Wurden im Quelltext mehrere Kommunikationen genutzt, welche die gleiche Richtung besitzen, ist der Einsatz der „dynamischen Nachbarschaft“ am besten zur Darstellung geeignet. Ist diese nicht gewünscht oder diese Darstellung nicht vorgesehen, so kann der Anwender immer noch dadurch unterstützt werden, indem in dem Prozessorfeld, welches die entfernten Nachbarn repräsentiert, alle Variablen aufgelistet werden. Auf dem dazugehörigen Kommunikationspfeil kann in der gleichen Reihenfolge die entsprechende Entfernung dargestellt werden. Wie bereits weiter oben beschrieben, sollten solche nur unzureichend anzeigbaren Fälle nur selten vorkommen, da diese in den meisten Fällen mit einer schlechten Performanceausnutzung korrespondieren.

Wie im letzten Abschnitt beschrieben, kann es vorkommen, dass mehrere unterschiedliche Operatoren innerhalb einer Kommunikation genutzt werden. Dies ist leicht zu erkennen, kann aber nur mit Einschränkungen visualisiert werden.

5.3.2.2 Darstellung durch Icons im Quelltext

Wurden Daten zur Kommunikation mit Hilfe der Dialogbox erzeugt, sollen diese anschließend in das Quelltextprogramm übertragen werden. Dies muss automatisch erfolgen. Genau wie bei der Aktivierung von Prozessorelementen kann hier gewählt werden, wie das Ergebnis angezeigt werden soll: Entweder textuell mit korrekter Syntax oder mit Hilfe eines Symbols bzw. Icons, das den Kommunikationsvorgang veranschaulichen soll.

Die Übernahme in eine textbasierte Programmzeile ist einfach zu realisieren, da diese bereits in der Dialogbox als Ergebniszeile angezeigt wird. Sollte der Kommunikationsbefehl später noch einmal mit Hilfe der Dialogbox bearbeitet werden, können die bereits vorhandenen Daten genau wie im letzten Kapitel beschrieben, übernommen werden.

Aus Gründen der Übersicht ist statt der Textversion der Einsatz von Icons zur Kommunikationsdarstellung im Quelltext sinnvoll. Wird eine Kommunikation in der Dialogbox zusammengestellt, können alle beteiligten Nachbarn verkleinert als Icon im Quelltext wiedergegeben werden. Die vollständige Syntax wird im Hintergrund des Icons gesichert und kann jederzeit durch einen Tool-Tipp sichtbar gemacht werden. Auch hierbei ist die dynamische Darstellung der Icons interessant, da auf möglichst wenig Platz eine Kommunikation sinnvoll dargestellt werden soll. Zeigt man immer die 16-er Nachbarschaft an, werden die Symbole für die Prozessorelemente relativ klein und die wichtige Information geht verloren. Abbildung 5-13 zeigt zwei mögliche Darstellungen von Icons im Quelltext, wobei hier durch den Tool-Tipp gleichzeitig der vollständige Quelltext angezeigt wird.

Abbildung 5-13: Unterschiedliche Icondarstellungen einer Kommunikationen im Quelltext

Ein Doppelklick auf ein Icon bewirkt, dass man wieder in die Dialogbox gelangt, um dort Daten ändern zu können.

Setzt man konsequent Icons ein, hat man in seinem Programm einen sehr schnellen Überblick über Aktivierungen und Kommunikation und kann dadurch bereits in vielen Fällen während der Programmierung erkennen, wo eventuelle Performanceprobleme zu erwarten sind.

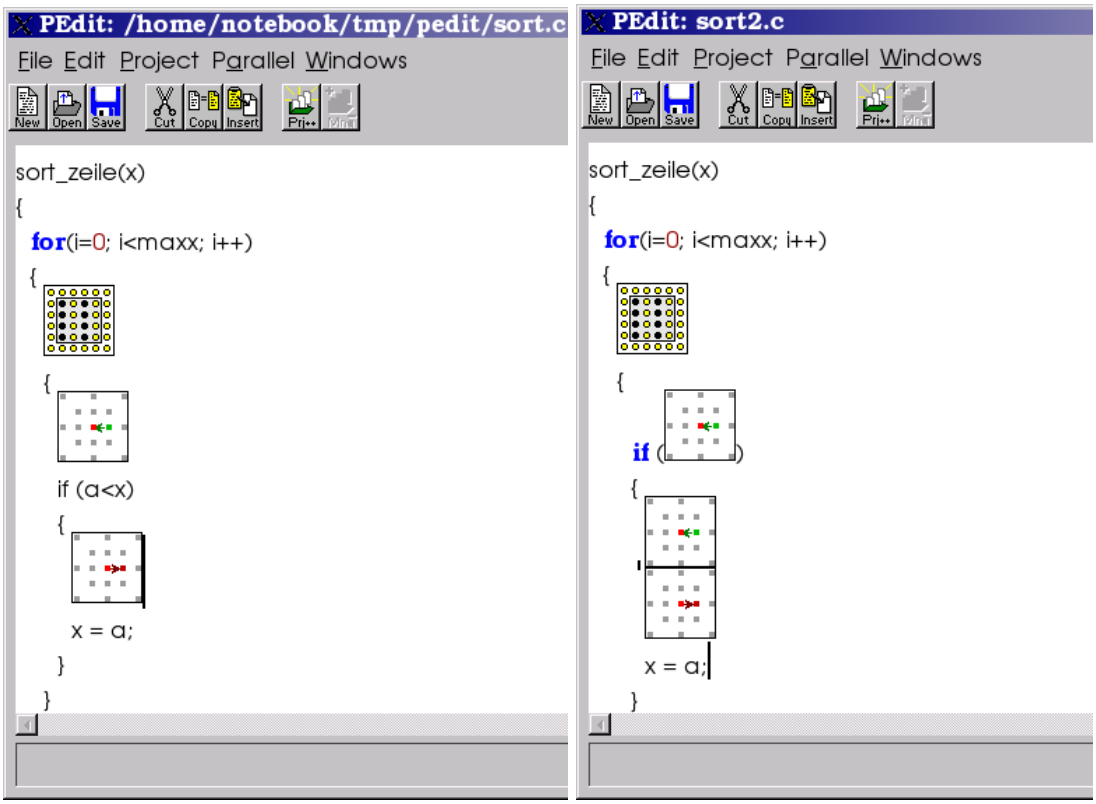


Abbildung 5-14: Zwei Implementierungen des Odd-Even-Sortieralgorithmus

Abbildung 5-14 zeigt zwei unterschiedliche Varianten, wie der Odd-Even-Sortieralgorithmus aus Kapitel 3.3 implementiert werden kann. In der Quelltextversion sehen beide Varianten sehr ähnlich aus. In der dargestellten Version mit Icons erkennt man sehr schnell, dass in der rechten Variante eine zusätzlich Kommunikation notwendig wird.

Die Darstellung durch Icons ersetzt sicher nicht die Analysetools, die von allen Parallelrechner-Herstellern angeboten werden. Diese können eine wesentlich genauere Performanceanalyse bieten - allerdings erst dann, wenn ein Programm fertig compiliert und durchgelaufen ist. Da parallele Programme teilweise tagelang laufen und eine Arbeitsstunde sehr teuer ist, wünscht man sich diese Ergebnisse allerdings oft früher, um ein möglichst effizientes Arbeiten unterstützen zu können.

5.3.3 Sonstige Hilfe mit grafischer Unterstützung

In den letzten Kapiteln wurden zwei Befehlstypen behandelt, die prädestiniert für eine grafische Präsentation sind. Denkbar ist die grafische Repräsentation zusätzlich für eine Reihe von weiteren Befehlen, nicht nur für parallele. So können zum Beispiel auch Schleifen grafisch unterstützt werden, um einen besseren Überblick zu gewährleisten.

Eine konsequente Weiterentwicklung der visuellen Darstellung könnte in einem Ansatz enden, die dem in der DIN 66 261 festgelegten Nassi-Shneiderman-Diagramm ähnelt. Damit bewegt man sich allerdings schnell weg von der gewohnten Darstellung von Quelltexten. Deshalb ist ein Benutzer oft nicht bereit, diese Hilfsmittel einzusetzen.

Aus diesem Grund ist immer abzuwägen, wie viele „Änderungen vom Standard“ ein Benutzer akzeptiert und willig einsetzt. Dies ist sicher von Benutzer zu Benutzer unterschiedlich, Verhaltensweisen kann man höchstens an entsprechenden Benutzermodellen festmachen. Trotzdem soll noch an einigen weiteren Beispielen veranschaulicht werden, wie der Benutzer zusätzlich unterstützt werden kann.

5.3.3.1 Komplexe Kommunikation (Router)

Die bisher vorgestellten parallelen Programmierelemente können alle sinnvoll grafisch dargestellt werden. Dies ist aber nicht immer der Fall. Der MasPar-Rechner bietet zum Beispiel den `router`-Befehl an. Mit ihm ist es möglich, dass quer über das Prozessorfeld „gleichzeitig“ jeweils zwei Prozessorelemente miteinander kommunizieren können. Dies ist grafisch nicht sinnvoll darzustellen, da nur ein „chaotisches Durcheinander“ angezeigt werden könnte. Da, wie in Kapitel 3.3 beschrieben, der `router`-Befehl ca. 1000-mal langsamer ist als die `xnet`-Befehle, könnte man argumentieren, diesen gar nicht zu unterstützen. Besser ist es allerdings, dem Benutzer die Entscheidung zu überlassen, wobei er entsprechend unterstützt und geeignet auf die Geschwindigkeit und Alternativen hingewiesen werden soll. Die Unterstützung kann wie in Kapitel 5.2.3 beschrieben durch eine kontextsensitive Hilfe mit direkter Eingabemöglichkeit stattfinden. Neben einer gleichzeitigen Erklärung des Befehls können die einzelnen Befehlsargumente in dafür vorgesehene Eingabefelder eingefügt werden (siehe Abbildung 5-15).

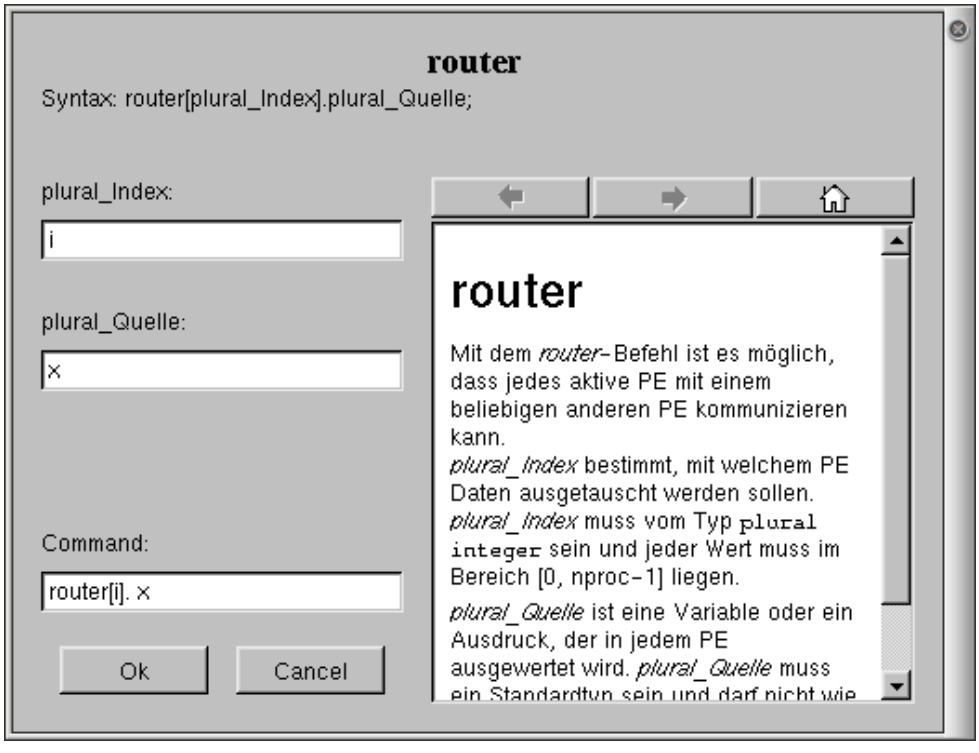


Abbildung 5-15: Dialogbox zur Unterstützung des `router`-Befehls

Der komplette Befehl wird in der Ausgabezeile angezeigt und später in Textform in das Quelltextprogramm übertragen. Die Darstellung als Icon kann im Allgemeinen nicht dynamisch angepasst werden, da in der Kommunikation keine Struktur erkennbar sein wird. Deshalb wären nur statische Icons einsetzbar, welche weniger aussagekräftig

wären, aber zumindest mit einer „Warnfarbe“ auf die langsame Kommunikation hinweisen könnten.

Genau wie bei allen anderen textuellen Hilfestellungen können Änderungen jederzeit in der Dialogbox bearbeitet werden, wobei vorherige Eingaben aus dem Quelltext in die Dialogbox übernommen werden können.

5.3.3.2 Spezialbefehle

Für häufig benötigte „Kleinstalgorithmen“ sind von den Programmiersprachen nicht immer entsprechende Befehle vorgesehen. Nun besteht die Möglichkeit, dem Anwender Bibliotheken (siehe Kapitel 5.1.5) oder skelettorientiertes Programmieren (siehe Kapitel 5.1.6) anzubieten. Bei komplexeren Aufgaben ist dies auch sinnvoll, bei kurzen Aufgaben behindern zusätzliche Funktionsaufrufe aber die Performance und sollten nicht eingesetzt werden.

Aber auch kurze Befehle können kompliziert sein. Als Beispiel sei hier das „Perfect Shuffle“ des gesamten Prozessorfeldes genannt, welches die angesprochenen Werte auf eine bestimmte Art und Weise mischt und das auf dem MasPar-Rechner in einer Programmzeile realisiert werden kann:

```
p = router[((iproc<<1) + (iproc>>lnproc-1)) & (nproc-1)].p
```

Mit dieser Syntax wird das gesamte Prozessorfeld auf effiziente Weise gemischt, auch wenn dabei der `router`-Befehl genutzt werden muss. Die `xnet`-Befehle könnten nur mit großem Aufwand eingesetzt werden, da bei einer Vermischung über das gesamte Prozessorfeld Quelle und Ziel bei fast allen Prozessorelementen nicht direkt über die Nachbarschaft erreicht werden können. Durch die Notwendigkeit die `xnet`-Befehl häufig aufrufen und zwischendurch die Aktivierungen ändern zu müssen, geht auch der Geschwindigkeitsvorteil der `xnet`-Befehle verloren.

Durch kleine Abänderungen der Befehlszeile könnten genauso „Perfect Shuffles“ von Zeilen oder Spalten durchgeführt werden. Allerdings sind hier alle Prozessorelemente über die `xnet`-Nachbarschaft erreichbar, so dass durch deren Ausnutzung ein wesentlich effizienteres Ergebnis erreicht werden kann. Das Ergebnis kann allerdings nicht mehr sinnvoll als einzelne Befehlszeile zurückgegeben werden, sondern es muss eine Folge von Befehlen genutzt werden, welche unterschiedliche Aktivierungen und Kommunikationen einsetzen.

Ein Beispiel für einen zeilenweisen „Perfect Shuffle“ lautet:

```
if (ixproc % 2 == 0) {
  x = xnetW[(nxproc<<1)].x}
else {
  x = xnetE[(nxproc – ixproc – 1)<<1].x
}
```

Nutzt der Programmierer keine Unterstützungen, so muss er bei jeder Anwendung entscheiden, welche der beiden Implementierungen er wählt. Zusätzlich sind beide Quelltexte so komplex, dass nur die allerwenigsten diese ohne größeres Nachdenken programmieren könnten. Um die Zeile bzw. Zeilen bei Bedarf jedes Mal neu zu schreiben, sind diese zu aufwändig, ein Funktionsaufruf dagegen müsste Parameter enthalten und könnte zu Performanceproblemen führen. Aus diesem Grund ist es das Beste, wenn „Perfect Shuffle“ „automatisch“ erzeugt werden könnte. Dies ist wiederum mit einer leicht von Kapitel 5.3.3.1 abgewandelten Form der Hilfestellung möglich (siehe Abbildung 5-16).

Abbildung 5-16: Dialogbox zur Unterstützung eines „Perfect Shuffle“

Dieses Mal wird nicht die Eingabe eines bestimmten Befehls unterstützt, sondern eine kurze und bestimmte Kombination von Befehlen. Da die Parameter in der Box abgefragt werden, kann für das Quellprogramm eine eindeutige Programmzeile zusammengestellt werden, die jeweils den schnellsten Algorithmus wählt.

Weitere für SIMD-Rechner typische Befehlsfolgen, die ebenso unterstützt werden können, sind das Ringschieben auf Zeilen oder Spalten oder das Spiegeln von Zeilen und Spalten.

6 PEdit, eine Entwicklungsumgebung für SIMD-Programme

Wie in der Motivation zu dieser Dissertation beschrieben, war der Grund für die Beschäftigung mit parallelen Entwicklungsumgebungen die Forschungstätigkeit am Lehrstuhl „Effiziente Algorithmen“. Hier wurden genetische Algorithmen entwickelt, welche durch die Leistungsfähigkeit des MasPar-Rechners neue und bessere Ergebnisse für große, komplexe Probleme finden sollten [Koh99]. Bedingt durch die Architektur des Rechners spielte die Kommunikation zwischen den Prozessorelementen eine sehr große Rolle. Um sich diese vorstellen zu können, wurden Skizzen erstellt, welche diese Kommunikation darstellten. Die in Kapitel 4.3.2 vorgestellte Entwicklungsumgebung MPPE des MasPar-Rechners unterstützt allerdings die Entwicklung und Verarbeitung solcher Skizzen nicht. Eine Erweiterung der Entwicklungsumgebung war auch nicht möglich, da keine Schnittstellen angeboten wurden, an denen man ansetzen konnte.

Durch Recherchen wurde herausgefunden, dass zu Beginn der Programmentwicklung nur Emacs Schnittstellen besaß, die es erlaubten, von dritter Seite die Entwicklungsumgebung mit eigenen Entwicklungen zu erweitern. Allerdings stellte sich heraus, dass mit der Erweiterungssprache Lisp nur wenige grafische Elemente und keine benötigten Bibliotheken zur Verfügung standen.

So wurde die Idee zu einer neuen Entwicklungsumgebung geboren, die den Programmierer bei der Entwicklung der Algorithmen durch grafische Elemente unterstützen sollte.

Im letzten Kapitel wurde herausgearbeitet, welche Merkmale für die Unterstützung eines Entwicklers sinnvoll sind. Diese Merkmale sollten von der zu entwickelnden Arbeitsumgebung möglichst gut unterstützt werden. Besonders wichtig war dabei die Hervorhebung paralleler Programmeigenschaften durch grafische Elemente.

In einem ersten Schritt wurde eine Unterstützung für die Programmierung von parallelen MasPar-Programmen angestrebt, die in diesem Kapitel besprochen wird. Im nächsten Kapitel wird gezeigt, wie diese Unterstützungsmöglichkeiten auf weitere Parallelrechnerumgebungen angepasst werden können.

Der Name PEdit steht für „Parallel-Editor“. Der Begriff „Parallel“ ist durch die zu unterstützende Rechnerarchitektur erklärbar. „Editor“ wurde gewählt, da im Gegensatz zu einer Entwicklungsumgebung der Schwerpunkt der Forschung auf das Eingabemedium, den Editor, gelegt werden sollte.

6.1 Zugrunde liegende Bibliotheken / UI-Toolkits

Bevor mit der Programmierung einer Entwicklungsumgebung begonnen werden kann, muss geklärt werden, welche speziellen Eigenschaften unterstützt werden sollen, welche Voraussetzungen durch die zu unterstützende Hardware gegeben sind und welche Werkzeuge eingesetzt werden sollen.

Die erwünschten Eigenschaften für parallele Entwicklungsumgebungen wurden bereits im letzten Kapitel beschrieben.

In Kapitel 3.3 wurde gezeigt, wie Programme für den MasPar-Rechner in Teile unterschieden werden, die auf der Paralleleinheit ablaufen und Teile, die auf dem Frontend-System benutzt werden. Als Frontend für die MasPar standen an der Universität Karlsruhe (TH) nur UNIX-Systeme zur Verfügung, die unter HP-UX, SunOS oder Solaris liefen. Windows-Systeme waren für die Zusammenarbeit nicht vorgesehen. Um für spätere Erweiterungen nicht von einem Betriebssystem abhängig zu sein, sollten Bibliotheken gefunden werden, die auf allen UNIX-Systemen und möglichst auch unter Windows lauffähig sind. Zusätzlich wurde darauf geachtet, dass die Bibliotheken frei verfügbar sind.

Um eine plattformunabhängige Lösung realisieren zu können, die grafische Systeme unterstützt, muss eine grafische Benutzungsschnittstelle (Graphical User Interface, GUI) genutzt werden, die vom darunter liegenden Fenstersystem abstrahiert. Auf diese Weise können auf allen Betriebssystemen die gleichen Befehle genutzt werden, die, von den Bibliotheken jeweils angepasst, umgesetzt werden. Zu beachten ist dabei, dass durch die Plattformunabhängigkeit besondere Eigenschaften einzelner Systeme nicht genutzt werden können, da diese auf anderen Systemen nicht abgebildet werden könnten.

Neben der Plattformunabhängigkeit und freien Verfügbarkeit waren bei der Auswahl einer geeigneten Bibliothek viele weitere Kriterien wichtig. So sollte die Bibliothek gut dokumentiert, logisch strukturiert und in sich homogen sein, damit ein schnelles Einarbeiten möglich ist. Aus Gründen der Wiederverwendung wurden objektorientierte Bibliotheken bevorzugt, allerdings wurden prozedurale Bibliotheken wegen ihrer Verar-

beitungsgeschwindigkeit nicht ganz außer Acht gelassen. Durch die angestrebte grafische Unterstützung mussten viele Instrumente angeboten werden, die grafische Elemente zur Verfügung stellen, trotzdem sollten alle Eigenschaften allgemeiner Bibliotheken enthalten sein. So war es wichtig, dass ein Fenstersystem, Menüs und Dialogboxen unterstützt werden, Klassen bzw. Funktionen bereit stehen, die Texte verwalten und manipulieren können, Suchfunktionen angeboten werden und möglichst weitgehende Unterstützung für Editorfenster vorgesehen sind. Booch hat als wichtige Kriterien in seinem Buch „Objektorientierte Analyse und Design“ [Boo94] Vollständigkeit, Anpassbarkeit, Effizienz, Sicherheit, Einfachheit und Erweiterbarkeit angegeben, die genauso berücksichtigt wurden, wie die Unterstützung der in Kapitel 2.6 kurz beschriebenen Entwurfsmuster und der Eingruppierung der Werkzeuge nach ihrer Arbeitsweise, wie sie in Kapitel 2.5 skizziert wurde.

Viele der Kriterien wurden in einer Studienarbeit ausgewertet und an Hand der vorausgewählten Bibliotheken Fresco, Interviews, Khoros und ET++ genauer untersucht [Hir97]. In einer nachfolgenden Diplomarbeit wurde noch die Bibliothek Qt, die Fresco sehr stark ähnelt, herangezogen [Hir98].

Fresco

Fresco, ein ehemaliges Forschungsprojekt unter der Schirmherrschaft des X Konsortiums (http://www.x.org/), stellte sich als sehr mächtiges objektorientiertes GUI-System heraus, das sehr viele Methoden für die Entwicklung von Benutzungsschnittstellen zur Verfügung stellt. Allerdings müssen zusätzlich weitere Bibliotheken für grundlegende Methoden wie Stapel, Stringfunktionen u.ä. herangezogen werden. Die Dokumentation ist nicht immer nachvollziehbar, so dass in vielen Fällen in den aussagefähigen Beispielprogrammen untersucht werden muss, wie eine bestimmte Klasse sinnvoll eingesetzt werden kann. Ansonsten werden die Kriterien Vollständigkeit (in Bezug auf die Unterstützung von Benutzungsschnittstellen), Anpassbarkeit, Sicherheit, Einfachheit und Erweiterbarkeit gut erfüllt, so dass mit Fresco ein erster Prototyp geschaffen wurde, welcher den grundlegenden Aufbau der zu entwickelnden Klassen überprüfen und das Zusammenspiel von Text- und Grafikelementen klären sollte.

Als erste Aufgabe wurde vorgegeben, einen in HTML-Syntax geschriebenen Text parsen zu können, wobei zumindest Texte und Grafiken der HTML-Syntax entsprechend dargestellt werden sollten. Im Rahmen der Studienarbeit konnte diese Umsetzung erfolgreich realisiert werden. Zum Zeitpunkt der Arbeit unterstützte noch kein GUI-System die direkte Unterstützung von HTML-Viewern, so dass diese Vorgehensweise sinnvoll erschien. Heute können bereits viele Systeme HTML-Viewer zur Verfügung stellen, allerdings ist meist die Erweiterung um Editierfunktionen nicht vorgesehen und nur schwer nachträglich zu integrieren, so dass diese Systeme nicht als Grundlage der Eigenentwicklung genutzt wurden.

Bei der Implementierung des HTML-Viewers stellte sich allerdings heraus, dass wesentliche Teile der Bibliothek nicht stabil arbeiten, da an einigen Stellen nicht klar erkennbar ist, ob mit Zeigern oder Referenzen gearbeitet werden muss und zum Abfangen der

Fehler eine Ausnahmebehandlung nicht zur Verfügung steht. Beim Erstellen der Bibliothek wurde konsequent auf ein objektorientiertes Design geachtet, leider mangelt es dadurch aber an der Performance und den hohen Speicheranforderungen des erzeugten Programms. Letztlich wurde aus den genannten Gründen nach einer weiteren geeigneten Grundlagenbibliothek gesucht.

InterViews

Das Ziel von InterViews, einem Projekt der Stanford University, war es, ein UI-Toolkit für grafische Benutzungsschnittstellen zu entwickeln. Es fällt sehr schnell auf, dass InterViews und Fresco viele Gemeinsamkeiten haben. Bei genaueren Recherchen konnte herausgefunden werden, dass Fresco auf InterViews basiert, InterViews selbst aber nicht mehr weiter entwickelt wurde, so dass es als Grundlagen für eigene Arbeiten nicht ausgewählt wurde.

Khoros

Khoros verfolgt den Ansatz, eine komplette Entwicklungsumgebung mit Elementen zur Datenexploration und –visualisierung bereit zu stellen. Es ist ein kommerzielles Produkt der Firma Khoral Research (http://www.khoral.com/) und wurde ursprünglich an der Universität von New Mexico begonnen. Khoros verfolgt einen prozeduralen Ansatz. Trotz vieler guter Eigenschaften wurde es deshalb nicht als Grundplattform benutzt.

ET++

ET++ steht für Editor-Toolkit und wurde von der ETH-Zürich entwickelt und verfolgt den Frameworkansatz [Gam92]. Realisiert wurden alle Interaktionselemente nach dem Modell-Viewer-Konzept. Grundlage aller Methoden sind die vier abstrakten Basisklassen `application`, `document`, `view` und `command`, von denen alle weiteren Objekte abgeleitet werden. ET++ verfolgte einen sehr guten Ansatz, allerdings wurde im Laufe der Zeit die Weiterentwicklung eingestellt, so dass es die heutigen Ansprüche an grafische Entwicklungsumgebungen nicht mehr unterstützt. Somit wurde der bereits entstandene Prototyp nicht weiter vervollständigt.

Qt

Als letzte Entwicklungsbibliothek wurde Qt betrachtet, die schließlich auch ausgewählt wurde [Her01]. Qt wird ständig von der Firma Trolltech weiter entwickelt. Im Jahr 2005 wurde die Hauptversion 4 veröffentlicht. Zu jeder Hauptversion wurden zusätzlich mehrere Unterversionen entwickelt, so dass mittlerweile über 20 „Final Releases“ erschienen sind. Es stehen unterschiedliche Lizenzmodelle zur Verfügung, so dass nicht kommerzielle Entwicklungen mit kostenlos zur Verfügung gestellten Bibliotheken entwickelt werden können. Neben mehreren Unix/Linux-Varianten werden auch Bibliotheken für Mac und Windows zur Verfügung gestellt. Dadurch können eigene Programme bei konsequenter Benutzung der Qt-Bibliotheken leicht für mehrere Betriebssysteme entwickelt werden. Die Online-Dokumentation ist mittlerweile einige hundert Seiten stark

und der Quelltext wird durch sehr viele gute Beispielprogramme aufgewertet. Qt ist die Grundlage der mittlerweile sehr erfolgreichen grafischen Benutzungsoberfläche KDE, die von vielen Linuxdistributionen standardmäßig genutzt wird. Große Unterstützung erhält man bei der Programmentwicklung durch sehr aktive deutsche und internationale Mailgruppen sowie eine Anzahl an Büchern, die in unterschiedlichen Verlagen erschienen sind.

Qt ist nach dem Model-View-Controller-Prinzip aufgebaut und kann als Werkzeugbibliothek zu den User Interface Toolkits gerechnet werden. Seit Version 3 steht zusätzlich ein User Interface Builder zur Verfügung.

Inhaltlich bietet Qt alle wichtigen Elemente, die für grafische Benutzungsoberflächen benötigt werden. Als wichtigste Elemente können Menüs, Schaltflächen, Kontrollkästchen, Optionsschalter, Eingabefelder, Listen, Rollbalken und Dialogfenster genannt werden. Um unabhängig vom zugrunde liegenden Betriebssystem programmieren zu können, werden zusätzlich allgemeine abstrakte Klassen für die Bearbeitung von Dateien und Verzeichnissen, Zugriffsrechten oder Datum und Uhrzeit zur Verfügung gestellt. Alle grafischen Elemente werden von einer abstrakten Klasse abgeleitet (`QWidget`), die auch dazu geeignet ist, eigene Weiterentwicklungen ansetzen zu können. Für die Verwaltung von kompletten Anwendungen steht die abstrakte Klasse `QApplication` zur Verfügung und Objekte, die im Hintergrund agieren, können mit allen notwendigen Eigenschaften von der Klasse `QObject` abgeleitet werden.

Eine als `QObject` realisierte Besonderheit von Qt ist das Signal/Slot-Prinzip. Über dieses Konzept ist es möglich, dass Objekte miteinander kommunizieren. Es können beliebige Methoden als `signals` deklariert werden, die Informationen verschicken. Gleichzeitig können ein oder mehrere `slots` eingesetzt werden, welche die gesendeten Signale empfangen. Dabei muss der Parameterblock des Slots genau dem empfangenen Signal entsprechen. Somit kann man im Voraus Methoden mit `signals` oder `slots` versehen, ohne bereits den genauen Empfänger oder Sender festlegen zu müssen – ein Vorteil gegenüber der klassischen „Event-gesteuerten“ Programmierung. Die Entwicklung der Methoden wird dadurch wesentlich erleichtert und eine spätere Erweiterung ist bei geschickter Planung sehr einfach möglich, da Nachrichten an bereits vollständig implementierte Methoden verschickt werden können, ohne dass diese nachträglich geändert werden müssen. Genauso können Nachrichten von fertigen Methoden in eigenen Entwicklungen genutzt werden. Um diesen Mechanismus nutzen zu können, ist allerdings ein Präprozessor notwendig, der die proprietären Quelltexterweiterungen in C++ konforme Daten umsetzt. Der bereitgestellte Präprozessor `moc` übernimmt diese Aufgabe und kann problemlos in ein „Makefile“ integriert werden.

6.2 Grundlegende Eigenschaften eines Editors für parallele Programme

Nachdem im letzten Kapitel die Grundlagen von Qt vorgestellt wurden, werden nun die grundlegenden Eigenschaften eines Editors für parallele Programme zusammengefasst.

Wie bereits erwähnt liegen SIMD–Programmen durch die Architektur der „single instruction“ lineare Befehlsabläufe zugrunde. Allerdings haben sie bedingt durch die Benutzung von „multiple data“ besondere Eigenschaften, die man beim Entwurf einer Entwicklungsumgebung bzw. eines Editors berücksichtigen sollte.

Grundsätzlich sollte der Editor alle Eigenschaften besitzen, die man von modernen Editoren für Einprozessorsysteme kennt (vgl. Kapitel 4.2). Dazu zählt zum Beispiel das Syntax–Highlighting (siehe Kapitel 5.1.7), sodass in C(++) alle existierenden 32 Schlüsselworte farbig dargestellt werden können. Zur Unterscheidung von Schlüsselwörtern, die zusätzlich für die parallele Plattform benötigt werden, können diese in einer zweiten Farbe dargestellt werden.

Die Benutzung von Farben zur Darstellung des Syntax-Highlighting war zu Beginn ihrer Einführung sehr umstritten, da man davon ausging, dass der Benutzer mehr verwirrt wird, als dass er dadurch eine Hilfe erhält. Mit der Zeit hat sich allerdings herausgestellt, dass Farben sehr sinnvoll eingesetzt werden können. Hat man beispielsweise ein Schlüsselwort falsch geschrieben oder vergessen, einen Kommentar zu schließen, sieht man das direkt an der entsprechenden Farbe. Den gleichen Vorteil kann man bei der Färbung von parallelen Befehlen benutzen.

Die Prozessorelemente von SIMD–Rechnern arbeiten aber nicht nur nebeneinander, sondern eine besondere Eigenschaft dieser Rechner ist, dass Daten zwischen den Prozessorelementen besonders schnell ausgetauscht werden können. Da SIMD–Rechner im Allgemeinen eine Gitterstruktur besitzen, kann man dem Benutzer diese Struktur als Grafik anbieten und ihn die Kommunikation „zeichnen“ lassen.

Ähnlich kann man den Benutzer grafisch auswählen lassen, welche Prozessorelemente aktiviert bzw. deaktiviert sein sollen, das heißt, auf welchen Prozessorelementen der nächste Befehl ausgeführt werden soll.

Komplexe Funktionen eines Parallelrechners, die nicht so einfach visualisiert werden können, müssen trotzdem auf sinnvolle Weise unterstützt werden. Das Vorgehen dazu wurde bereits in Kapitel 5 erklärt.

6.3 Implementierung

Für die Implementierung mit Hilfe von Qt wurde als Plattform ein Linux-System gewählt. Trotzdem soll die Entwicklung möglichst plattformunabhängig durchgeführt werden. Das MVC-Modell soll weitgehend umgesetzt werden, um weitere Möglichkeiten der Benutzungsunterstützung einfach integrieren zu können, ohne grundlegende Algorithmen ändern zu müssen. Qt unterstützt dieses Vorgehen, da es selbst das MVC-Modell als Grundlage nutzt.

Eine wichtige Vorbereitung für die Implementierung ist eine Übersicht über alle benötigten Klassen.

6.3.1 Klassenübersicht

Das Programm PEdit besteht aus über 50 Klassen, welche in Tabelle 6-1 aufgeführt sind.

Tabelle 6-1: Klassen von PEdit

Arrow	PEditProject	RTFGlyph
ArrowValue	PEditProjectFile	RTFChar
DialogAkt	PEditProjectNavigator	RTFDrawing
DialogKomm	PEditProjectTree	Folder
drawData	PEditWidget	Ppeadraw
FktScanner	PFileInfo	Ppekdraw
GlyphInfo	PPFile	RTFLine
help	PPTreeItem	RTFPicture
Init	ProjectDirectoryDialog	RTFReturn
initRec	ProjectFileSettings	RTFScrollWidget
Input	ProjectNameDialog	RTFStyle
LineInfo	PWinMenuInfo	RTFStyleContext
ParenthesisData	QktsItemSearchInfo	RTFStyleContextObject
PBookmark	QktsTreeList	RTFWidget
PEditBookmarks	QktsTreeListItem	ScanLine
PEditFile	RTFCHighlighter	TIndex
PEditFindDialog	RTFClipboard	yy_buffer_state
PEditMenu	RTFCursor	yy_trans_info

An dieser Stelle soll nicht auf alle Klassen und Methoden eingegangen, sondern nur eine kurze Übersicht gegeben werden.

Klassen, die mit „P“ beginnen, dienen der Gesamtverwaltung des Programms. Hier werden unter anderem die Verwaltung der Programmfenster, das Menü oder die Projektverwaltung gesteuert. Ein weiterer großer Teil an Klassen beginnt mit „RTF“. RTF soll auf Rich-Text-Format hinweisen und andeuten, dass durch diese Basisklassen Texte beliebig formatiert und Visualisierungen eingefügt werden können. Weitere Klassen enthalten parallele Dialogboxen, Hilfefenster, Initialisierungen, Syntax-Highlighting oder stellen Hilfsfunktionen zur Verfügung. Klassen, die mit „Qkts“ beginnen, stellen abstrakte Listen zur Verfügung, die für die Projektverwaltung genutzt werden. Sie wurden nicht wie die anderen Klassen selbst entwickelt, sondern als freie Quelle integriert.

Da fast alle Klassen Grundfunktionen zur Verfügung stellen, die in einer späteren Ausbaustufe erweitert werden können, wird die Vererbungsmöglichkeit noch sehr wenig eingesetzt. Lediglich bei der Darstellung von Elementen im Editorfenster, den so genannten Glyphs, und bei der Darstellung von Kommunikationspfeilen in Dialogboxen wurde die Vererbung genutzt, da hier die Elemente jeweils in unterschiedlichen Ausprägungen benötigt werden. Trotzdem besteht eine sehr hohe Abhängigkeit der Klassen untereinander, da sehr häufig Daten auf unterschiedlichen Abstraktionsschichten ausgetauscht werden müssen.

Auf eine Darstellung der Abhängigkeiten der Klassen untereinander wurde verzichtet, da sie nicht wesentlich zum Verständnis der einzelnen Funktionen beiträgt. Bei Bedarf kann man durch die automatisierte Dokumentation der Quelltexte einen Einblick gewinnen, die mit Hilfe des Programms „doxygen“ durchgeführt wurde [Pet03]. Auch in der Diplomarbeit von Eric Hirsch [Hir98] sind bereits die wichtigsten Abhängigkeiten ausführlich dargestellt worden.

6.3.2 Darstellung von parallelen Elementen im Editorfeld

Glyphs sind nach Gamma [GJH96] Objekte, die in einer Dokumentenstruktur erscheinen können. Sie haben die grundlegende Aufgabe zu wissen, wie sie sich darstellen müssen und wie viel Platz sie benötigen. Realisiert wurden in der Arbeit Glyphs durch eine abstrakte Klasse `RTFGlyph`, die nach dem Fliegengewichtsmuster erstellt wurde, da sie sehr häufig benötigt werden und deshalb wenig Platz in Anspruch nehmen sollen. Sie enthalten einen Konstruktor und Destruktor, zwei Methoden für die Text- und Grafikdarstellung, zwei Methoden zur Rückgabe der Abbildungsgröße und eine Kopiermethode. An Daten werden nur der zu repräsentierende Text, die Positionsdaten und ein Zeiger auf das Elternelement gespeichert. Weitere Funktionen, beispielsweise wie der Text dargestellt werden soll, werden in weitere Klassen ausgelagert und nur durch einen Pointer referenziert. Die Klasse `RTFStyleContext` enthält eine Liste der möglichen Textdarstellungen, so dass gleich aussehende Texte jeweils nur einmal beschrieben werden. Diese platzsparende Speichermethode kann effizient implementiert werden, da die Anzahl der möglichen Textausprägungen durch die Regeln des Syntax-Highlighting begrenzt sind und deshalb die Stileigenschaften (StyleContext) nicht als verkettete Liste, sondern als direkt zugreifbares Array implementiert werden konnten.

Von der abstrakten Klasse `RTFGlyph` werden die benötigten Klassen `RTFChar`, `RTFDrawing`, `RTFLine`, `RTFPicture` und `RTFReturn` abgeleitet. Sie benutzen in ihrer Implementierung Teile des Kompositionsmusters. Abbildung 6-1 zeigt einen Überblick über die Kindklassen von `RTFGlyph`.

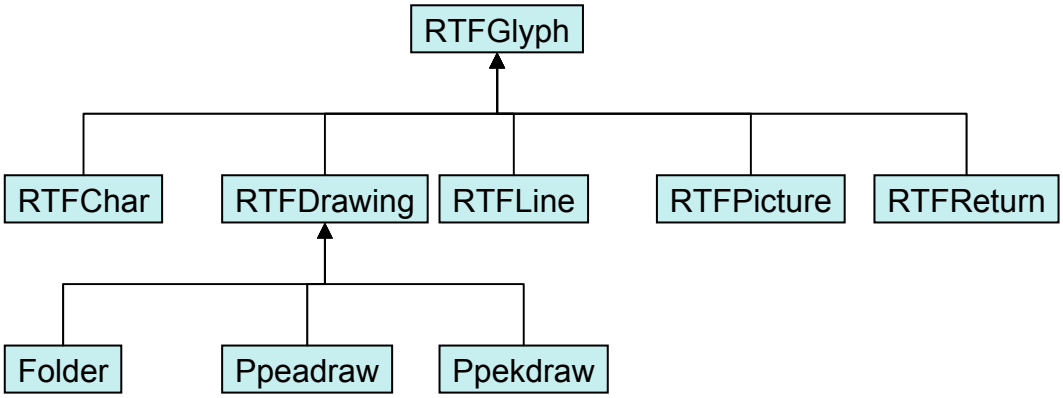


Abbildung 6-1: Abstrakte Klasse `RTFGlyph` und deren Kinder

`RTFChar` stellt einfache Buchstaben dar. `RTFDrawing` dient wiederum als Basisklasse für alle Klassen, die bestimmten Text durch grafische Symbole ersetzen sollen, indem Vektorgrafiken abhängig von mitgelieferten Parametern angezeigt werden. Die abgeleiteten Klassen `Ppeadraw`, `Ppekdraw` und `Folder` werden weiter unten genauer vorgestellt. Sie stellen die Hauptfunktionalität zur Darstellung von parallelen Programmteilen in einem Editorfeld zur Verfügung. `RTFPicture` kann beliebige Pixelgrafiken anzeigen. Die Klasse wurde integriert, um die Implementierung im Sinne des Ritch-Text-Formats zu vervollständigen. Allerdings wird die Klasse im Moment nicht benötigt. Die Klasse `RTFLine` dient als Containerklasse, damit komplette Zeilen schneller angesprochen werden können. `RTFReturn` stellt nur das Sonderzeichen für einen Zeilenumbruch dar. Die genannten Klassen zusammen enthalten alle Informationen, die innerhalb des Editors benötigt werden, um die Elemente darzustellen.

In Abbildung 6-2 ist links symbolisch ein Editorfeld mit möglichen Eingaben zu sehen. Auf der rechten Seite sieht man die zur Benutzung benötigten Klassen. Jeder Buchstabe wird durch die Klasse `RTFChar` dargestellt, die Kommunikation durch eine abgeleitete Klasse von `RTFDrawing` usw. Zusätzlich sieht man, wie eine Zeile nochmals durch die Klasse `RTFLine` beschrieben wird. Dort wird in einer Liste gespeichert, welche Elemente in der Zeile vorhanden sind, wie hoch und breit die Zeile ist, usw.

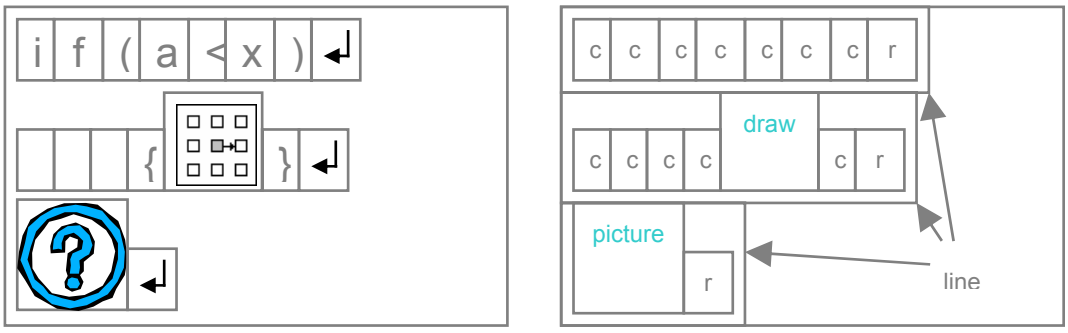


Abbildung 6-2: Darstellung von RTFGlyphs

Entgegen der Empfehlung von Gamma wurden alle Zeichen des Editors von RTFGlyph abgeleitet. Durch die konsequente Auslagerung von Zusatzinformationen konnten die Klassen so leichtgewichtig gestaltet werden, dass normale Quelltextdateien im Umfang bis 10.000 Zeichen schnell genug dargestellt werden. Gamma empfiehlt gleiche Ele-

mente durch „Oberklassen“ abzubilden. Der Vorteil wäre, dass die Anzahl der im Speicher zu haltenden Klassen wesentlich kleiner würde. Dies müsste man allerdings durch den Nachteil erkaufen, dass eine zusätzliche Navigationsmöglichkeit innerhalb von Elementen einer Klasse programmiert werden müsste. Auch Drag&Drop und weitere Funktionen müssten noch ein zweites Mal innerhalb der Oberklasse implementiert werden. Für ein professionelles Produkt wäre der höhere Aufwand aufgrund des Performancegewinns sinnvoll, für die Darstellung der Vorteile von parallelen Programmteilen durch entsprechende Symbole rechtfertigt sich der Mehraufwand allerdings nicht.

Soll eine Aktivierung von Prozessorelementen im Editorfeld angezeigt werden, so übernimmt dies die von `RTFDrawing` abgeleitete Klasse `Ppeadraw`. Das „`pea`“ steht dabei für Parallel-Editor-Aktivierung. In der momentanen Implementierung kann ein Feld von Prozessorelementen angezeigt werden, in dem aktive Zeilen, Spalten, Bereiche und beliebige Kombinationen davon dargestellt werden. Würde man versuchen alle Informationen in die Symbole zu integrieren, wie sie in den Dialogboxen möglich sind, so müssten die Symbole teilweise sehr groß werden. Dadurch würde allerdings der Vorteil der ins Auge fallenden Darstellung von Aktivierungen zunichte gemacht, indem sonst kaum noch Quelltext auf einer Bildschirmseite dargestellt werden könnte. Deshalb kann die jeweilige Aktivierungsart teilweise nur symbolisch gezeigt werden. Die Abbildung 6-3 zeigt einige Screenshots, wie die Aktivierung der aktiven Elemente im Editorfeld angezeigt wird.

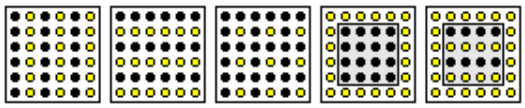


Abbildung 6-3: Darstellung von aktiven Elementen im Editorfeld

Die Darstellungen reichen aus, um eine schnelle Übersicht zu gewinnen, wie die Aktivierung aussieht. Benötigt man genauere Informationen über die aktivierten Felder, so kann man sich einen Tool-Tipp anzeigen lassen, indem man mit der Maus über die Abbildung fährt (siehe Abbildung 6-4).

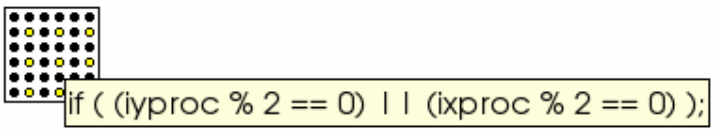


Abbildung 6-4: Darstellung von aktiven Elementen im Editorfeld mit Tool-Tipp

Alternativ führt ein Doppelklick auf die Abbildung zu einem ausführlichen Eingabe- und Informationsfenster, welches später vorgestellt wird.

Auch die Klasse `Ppekdraw` ist von `RTFDrawing` abgeleitet. Sie zeigt die Kommunikation von aktiven Prozessorelementen. Da auch hier nur wenig Platz für die Darstellung zur Verfügung steht, werden die Namen der zu übertragenen Variablen oder Konstanten nicht angezeigt, allerdings wird am Kommunikationspfeil die Entfernung dargestellt, wenn eine ferne Kommunikation vorliegt. Die Pfeile werden jeweils in der Kommunikationsrichtung dargestellt, so dass ein sehr guter Eindruck von der vorgenommenen Kommunikation wiedergegeben werden kann. Zusätzlich kann mit dem Tooltipp die

vollständige Kommunikation angezeigt werden. Die Abbildung 6-5 zeigt einige Darstellungen der Kommunikation.

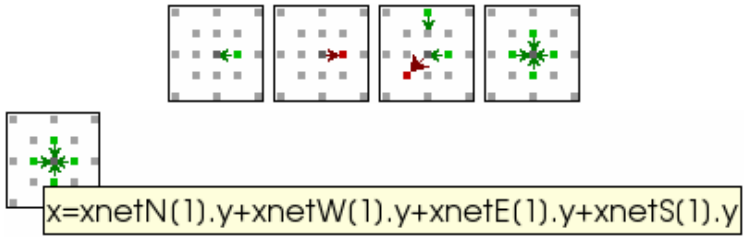


Abbildung 6-5: Darstellung von Kommunikation im Editorfeld

In der Abbildung ist zusätzlich zu sehen, dass für abgehende Kommunikationen eine andere Farbe und Pfeilgröße gewählt wurde als für eingehende. Dies soll zusätzlich der schnelleren Unterscheidbarkeit dienen.

Die Klasse `Folder` ist ein Beispiel dafür, wie durch eine sinnvolle Klassenplanung mit wenig Aufwand ganz neue Funktionen in die Programmierumgebung integriert werden können. Sie soll beliebig langen Quelltext verkürzt anzeigen, indem er durch ein entsprechendes Symbol angezeigt wird. Da `RTFDrawing` bereits alle notwendigen Methoden vorgibt, muss in der Klasse nur noch spezifiziert werden, wie sie sich darstellt und was gespeichert werden soll. Drag&Drop, Copy&Paste, löschen usw. funktionieren durch die Vererbung sofort. Die aktuelle Darstellung wird realisiert, indem die erste Zeile des zu faltenden Quelltextes in einer anderen Farbe dargestellt wird, der restliche Text wird nicht mehr dargestellt. Daneben ist (fast) jede andere Darstellung denkbar und leicht zu implementieren.

Innerhalb der Klassen wird eine Liste von Zeigern auf die vorhandenen Glyphs gespeichert. Dadurch ist es unwichtig, ob es sich um einfache Zeichen, Bilder oder Darstellung der Aktivierung oder Kommunikation handelt. Da der `Folder` selbst ein Glyph ist, kann die Faltung auch beliebig oft rekursiv genutzt werden, ohne dass dadurch mehr Aufwand bei der Implementierung entsteht.

Das einfache Beispiel des Faltens zeigt, dass durch die Klasse `RTFDrawing` mittels Vererbung sehr einfach neue Funktionalitäten in das Editorfeld integriert werden können, wie sie evtl. für andere parallele Rechnertypen benötigt würden.

Da die parallelen Eigenschaften der Rechner wie Aktivierung und Kommunikation als dynamische Eigenschaften direkt in den Editor integriert werden sollten und dabei grafische Elemente die erste Wahl waren, wurde die komplett neue Entwicklung eines Editors notwendig. Kein vorhandener und erweiterbarer Editor besaß die Eigenschaft, dynamische Grafiken in den Text zu integrieren und dabei einen auswertbaren Kontext zu integrieren. Der Ansatz, die Grafik genau wie jeden einzelnen Buchstaben als zu repräsentierendes Element anzusehen, hat bewirkt, dass alle weiteren Ergänzungen wesentlich einfacher vorgenommen werden konnten.

Die Gestaltgesetze wurden bei der Entwicklung der Symbole berücksichtigt. So kann beispielsweise jederzeit die Darstellung der Matrixorganisation von Prozessorelementen wieder gefunden werden, indem das Gesetz der Nähe berücksichtigt wurde. In Bezug

auf die DIN 66 234 Teil 8 können besonders die Elemente Aufgabenangemessenheit, Selbstbeschreibungsfähigkeit und Erwartungskonformität wieder gefunden werden.

6.3.3 Dialogboxen zur Darstellung von aktiven Prozessorelementen

Im letzten Abschnitt wurde gezeigt, wie Aktivierung und Kommunikation innerhalb des Editorfeldes dargestellt werden. Vor der Darstellung steht allerdings noch die Erzeugung, die durch spezialisierte Dialogboxen durchgeführt wird.

Das Prinzip der Darstellung von Aktivierungen wurde bereits in Kapitel 5.3.1 erläutert. Die Realisierung soll besonders zeigen, wie eine Dialogbox gestaltet werden kann, welche die Eigenschaften Erwartungskonformität, Fehlertoleranz, Lernförderlichkeit, Steuerbarkeit und Individualisierbarkeit berücksichtigt. Auch die Aufgabenangemessenheit und Selbstbeschreibungsfähigkeit sollen kurz angesprochen werden.

Die Erwartungskonformität ist für die Akzeptanz der grafischen Eingabehilfe sehr wichtig. Werden beispielsweise kleinere Zahlen als Abstandswerte zwischen Zeilen und Spalten eingegeben, so werden diese Abstände exakt in der Grafik wiedergegeben. Leider kann bei großen Abständen dieser aus Platzgründern nicht mehr dargestellt werden, so dass er mit einem Pfeil angezeigt werden muss. Der Pfeil wird mit der entsprechenden Zahl beschriftet. Damit der Abstand in einigen Grenzfällen trotzdem exakt visualisiert werden kann, wurde die Dialogbox sehr dynamisch aufgebaut. Zieht der Benutzer die Dialogbox größer, so werden in den Feldern mehr Prozessorelemente dargestellt. Sollte die Anzahl nun ausreichen, um einen exakten Abstand oder eine exakte Fläche darzustellen, so wird automatisch die Darstellung von Pfeilen auf exakte Anzahl an Prozessorelementen geändert, so wie dies in Abbildung 6-6 gezeigt wird.

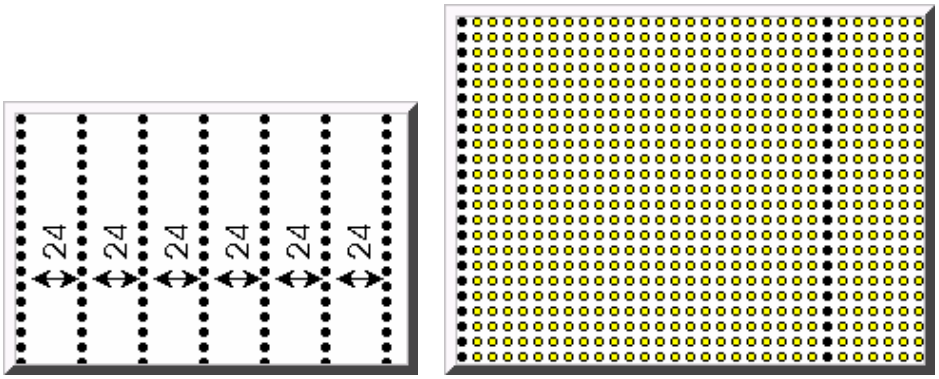


Abbildung 6-6: Darstellung einer Zeilenaktivierung innerhalb der Dialogbox

Wenn man als Abstandswerte Variablen oder komplette Funktionen einsetzt, können diese nur mit Pfeilen und Beschriftung wiedergegeben werden. Eine Darstellung durch eine genaue Anzahl an Prozessorelementen wäre in diesem Fällen nur innerhalb eines Debugmodus möglich (siehe Kapitel 6.3.6).

Trotz dieser kleinen Einschränkung des Modells auf dem Bildschirm gegenüber dem realen Prozessorfeld ist schnell zu erkennen, dass jeder ohne Kenntnisse von entsprechenden Befehlen bestimmte Zeilen oder Spalten von Prozessorfeldern aktivieren kann, da nur Abstandswerte eingegeben werden müssen.

Auf die gleiche Weise lassen sich rechteckige Bereiche von Prozessorelementen aktivieren, indem man die Koordinaten zweier gegenüberliegenden Eckpunkte angibt. Abbildung 6-7 zeigt eine Fläche von aktiven Prozessorelementen, die links einen Rand von `2` inaktiven Prozessorelementen besitzt, oben sind `4` Prozessorelemente inaktiv, rechts „`max_x`“ und unten „`y`“.

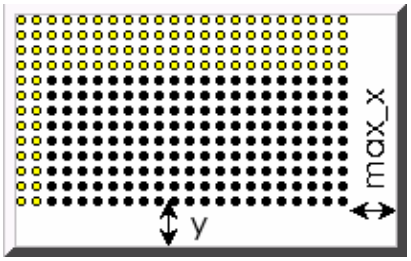


Abbildung 6-7: Darstellung einer Flächenaktivierung innerhalb der Dialogbox

Alle Eigenschaften lassen sich über logische „UND“- oder „ODER“-Verknüpfungen verbinden, so dass auf einer bestimmten Fläche bestimmte Zeilen und Spalten ausgewählt werden können.

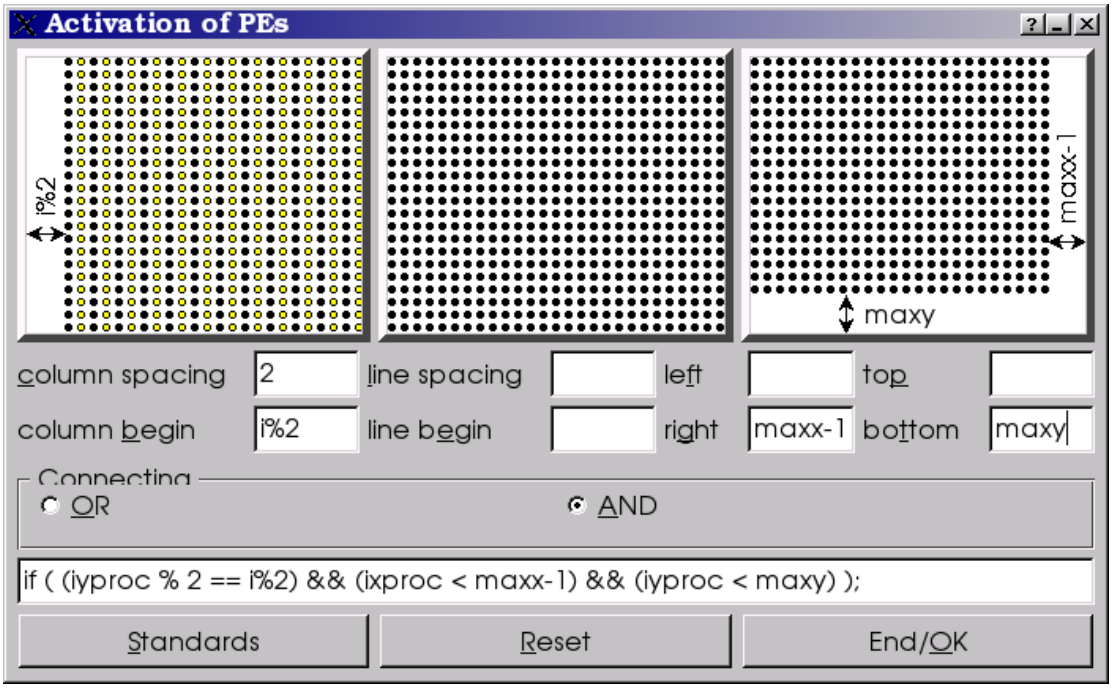


Abbildung 6-8: Dialogbox zur Aktivierung von Prozessorelementen

In der abgebildeten Dialogbox wird eine Aktivierung des Odd-Even-Sort Algorithmus gezeigt. Jede zweite Spalte ist aktiviert. Begonnen wird in der durch die Funktion i%2 definierten Spalte. Außerdem ist das Feld mit den Eckpunkten (0,0) und (maxx-1, maxy) ausgewählt. Der Eckpunkt (0,0) muss nicht angegeben werden, da dies die Voreinstellung ist. Damit Spaltenauswahl und Bereichsauswahl gleichzeitig benutzt werden können, ist die „AND“-Verbindung ausgewählt. Als Resultat werden nur die Prozessorelemente jeder zweiten Spalte im angegebenen Feld aktiviert.

Der Benutzer kann alle Programmiertätigkeiten ohne weitere Unterstützung direkt im Editorfeld durchführen. Nach dem in Kapitel 2.1 beschriebenen Benutzermodell ist

allerdings davon auszugehen, dass bei parallelen Programmteilen Unterstützungsbedarf besteht und deshalb die Dialogboxen eingesetzt werden, um die Lernförderlichkeit zu steigern. Gibt der Programmierer in den Feldern zur Aktivierung von Zeilen, Spalten und Flächen Werte ein, so werden diese nicht nur grafisch dargestellt, sondern zusätzlich der korrekte Programmcode angezeigt. Da jedes Feld der Aktivierung eine feste Aufgabe hat (beispielsweise „Abstand zwischen zwei aktiven Zeilen“), kann für jedes Feld einfach der dazugehörige Quellcode angegeben werden. Aufwändiger wird es allerdings bei der Benutzung mehrerer Eingabefelder, da dabei Verknüpfungen (meist in Form von AND oder OR) notwendig werden. Aus diesem Grund werden dem Benutzer auch Felder für die mögliche Verknüpfung angegeben. Trotzdem genügt das nicht, um jede mögliche Aktivierung darstellen zu können. Geht man davon aus, dass `A`, `B`, `C`, `D` und `E` Aktivierungen von Zeilen, Spalten und Flächen darstellen, kann ein Benutzer im Quelltext auch folgendes Konstrukt vorsehen:

```
(((A AND B) OR (C AND D)) AND NOT (E XOR A))
```

Die Klammern und die doppelte Auswertung von Ausdrücken können nicht immer sinnvoll dargestellt werden. In diesen Fällen werden trotzdem möglichst viele Einzelaktivierungen in den entsprechenden Grafiken dargestellt. Ganz wichtig ist allerdings, dass in diesen Fällen der Programmierer auf die nur teilweise Visualisierung hingewiesen wird und bei einer Änderung in den Aktivierungsfeldern der vorgegebene Quelltext nur an der richtigen Stelle geändert wird.

Beispiele, wie nicht mit einem von Benutzer erzeugten Quellcode umgegangen werden sollte, zeigen die grafischen HTML-Editoren, die um das Jahr 2000 erschienen sind. Sie änderten eigenmächtig die Groß-/Kleinschreibung, fügten Zeilenumbrüche hinzu oder löschten sie und fügten sehr viele meist unnötige zusätzliche Codefragmente ein. Ein Benutzer, der anschließend wieder mit seinem Quelltext arbeiten wollte, fand sich darin meist nicht mehr zurecht.

Um den Anspruch der vollständig erhaltenden Quellcodedarstellung zu erfüllen und trotzdem beliebig komplexe Quellcodeänderungen in den Aktivierungsfeldern zuzulassen, ist ein relativ aufwändiger Parser notwendig. Im Rahmen der Arbeit wurde hier eine etwas einfachere Lösung gewählt, die aus einem vorgegebenen Quelltext jeweils die erste Stelle ausfindig macht, die für ein Aktivierungsfeld notwendige Daten enthält. Diese Daten werden jeweils übernommen und bei einer Änderung an die gleiche Stelle zurück geschrieben. Alle nicht berücksichtigten Zeichen bleiben im Quelltext genau an der Stelle vorhanden, die der Benutzer vorgesehen hatte. Somit kann sichergestellt werden, dass der Quelltext durch die Dialogbox in seiner Semantik nicht verfälscht wird und die Erwartungskonformität möglichst hoch ist.

Steuerbarkeit und Individualisierbarkeit wurden zum einen dadurch realisiert, dass sich die Dialogbox beliebig vergrößern lässt und dabei die Anzeige der aktiven Prozessorelemente dynamisch angepasst wird. Jeder Benutzer kann die für seinen Bildschirm und seine Arbeitsweise ideale Größe der Felder wählen. Zum anderen wurde die Individualisierbarkeit berücksichtigt, indem alle Texte und Farben der Dialogbox persönlich

eingestellt werden können. Dazu wurde eine flexible Klasse `init` programmiert, die aus einer Textdatei beliebige Variablen und dazugehörige Werte auslesen kann. So werden `string`-, `double`-, `float`-, `int`-, `long`-, `short`-, `uint`-, `ulong`-, `ushort`- und `QColor`-Variablen berücksichtigt. QColor ist dafür zuständig, dass direkt Farbwerte eingelesen werden können. Durch diese Individualisierung ist es auch problemlos möglich, die Dialogbox in einer beliebigen Sprache darzustellen. Personen, die größere Kontraste oder dickere Pfeile benötigen, können dies ebenfalls beeinflussen, so dass in gewissem Rahmen auch die Barrierefreiheit eingehalten werden kann.

In Bezug auf das ergonomische Merkmal Aufgabenangemessenheit kann die Dialogbox direkt bei der anstehenden Aufgabe zur Lösung beitragen. Der Selbstbeschreibungsfähigkeit sollte durch die grafische Modellierung des Algorithmus im Kopf des Benutzers und deren Abbildung in der Dialogbox entsprochen werden. Zusätzlich sind alle Eingabefelder beschriftet und durch erklärende Tool-Tipps ergänzt.

6.3.4 Dialogboxen zur Darstellung der Kommunikation

Ähnlich wie bei der Aktivierung wird auch für die Eingabe von Kommunikationsvorgängen eine Dialogbox eingesetzt. Der Hauptteil der Dialogbox besteht aus der Darstellung des XNet. Dabei wird wie in Kapitel 5.3.2 beschrieben eine an das Aussehen einer 16er Nachbarschaft angelehnte Nachbarschaft dargestellt. Die äußeren Nachbarn erhalten einen größeren Abstand, um darzustellen, dass diese nicht unmittelbare Nachbarn sein müssen. Diese Berücksichtigung der Gestaltgesetze wird auch eingehalten, wenn die Größe der Dialogbox verändert wird. Dabei werden in der Implementierung so genannte „Stretchfunktionen“ eingesetzt, die wie Federn für die Ausrichtung der Felder sorgen. Bei der Realisierung wurde darauf geachtet, dass einerseits die Abstände der Felder untereinander proportional eingehalten werden, andererseits die Eingabefelder, die als Platzhalter für die einzelnen Prozessorelemente eingesetzt werden, möglichst groß werden, um breitere Variablennamen komfortabel eingeben zu können. Abbildung 6-9 zeigt den jeweiligen Ausschnitt des XNets bei unterschiedlich groß dargestellten Dialogboxen.

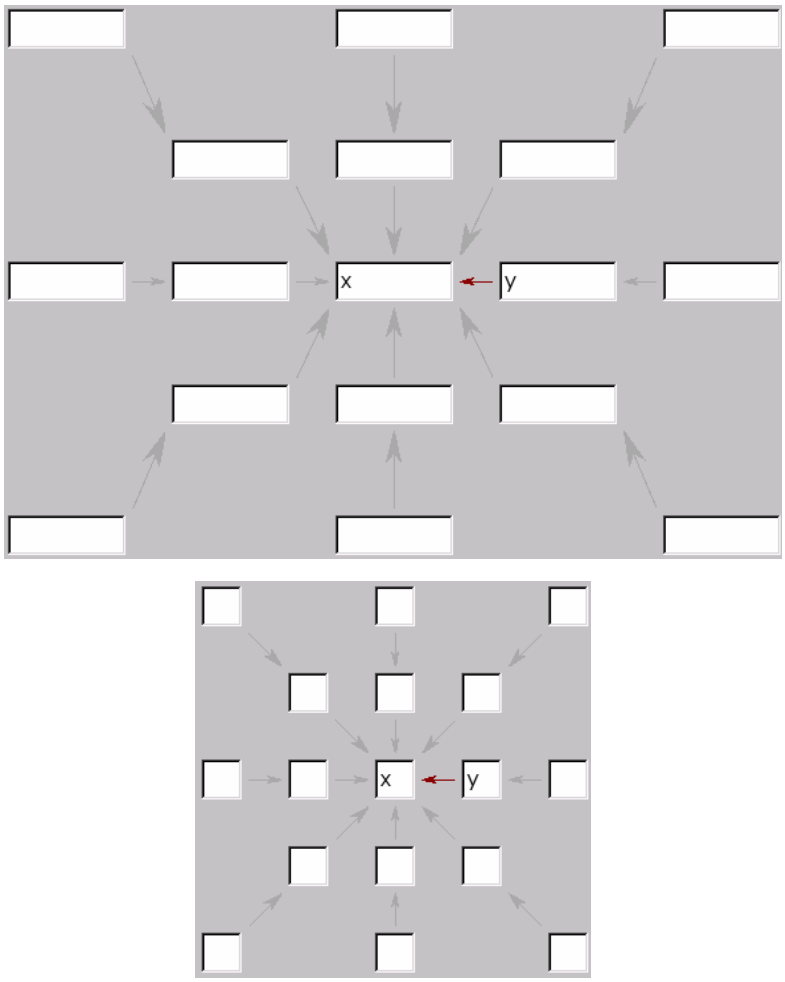


Abbildung 6-9: Unterschiedliche XNet-Darstellung je nach Dialogboxgröße

Die gesamte Dialogbox wurde wie in Kapitel 5.3.2 vorgeschlagen realisiert. Allerdings wurde die Dynamisierung des XNet nicht berücksichtigt, da die Notwendigkeit dazu nur selten vorhanden ist. Aus ergonomischen Gesichtspunkten ist die Anzahl der Eingabefelder auf den ersten Blick ein Problem. Laut Richtlinie sollten nicht mehr als fünf bis sieben Felder vorhanden sein, um nicht zu viele Eingaben innerhalb einer Dialogbox anfordern zu müssen. Da die Hauptaufgabe der Eingabefelder in der Kommunikationsbox die Darstellung des XNets ist, besteht hier allerdings eine Doppelfunktion. Die Repräsentation rückt in den Vordergrund, die Eingabefunktionalität in den Hintergrund. In den allermeisten Fällen werden für die Eingabe nur zwei bis drei Felder genutzt, so dass man innerhalb der ergonomischen Vorgaben bleibt.

Auch „Fitts Law“ wurde berücksichtigt. Darin wird ein Modell beschrieben, welches Vorraussagen über die Geschwindigkeit einer Interaktion zwischen Mensch und Maschine treffen kann. In dem 1954 vorgestellten Modell werden Formeln aufgestellt, mit denen man durch Übertragung auf aktuelle Gegebenheiten berechnen kann, wie schnell mit der Maus ein Wechsel zwischen zwei Elementen möglich ist [Mac91]. Eine Kernaussage ist, dass einerseits eine Navigation zwischen zwei Elementen immer länger dauert, je weiter diese auseinander liegen, andererseits können sie schneller angesteuert werden, je größer die Elemente werden. Somit bleibt bei der Dialogboxrealisierung die Geschwindigkeit, mit der innerhalb der Dialogbox navigiert werden kann, immer relativ

gleich groß, da sich einerseits die Abstände zwischen den Eingabefeldern, aber andererseits auch die jeweiligen Eingabefelder selbst in der gleichen Relation ändern, wenn die Größe der Dialogbox verändert wird.

Um die Flexibilität der Dialogbox zu erhalten, ist es wie beim Aktivierungsdialog sehr wichtig, dass die resultierende Ausgabezeile jederzeit sichtbar und aktuell ist. Sollen innerhalb einer Zuweisung eine größere Anzahl an Kommunikationen eingesetzt und dabei unterschiedliche Operatoren genutzt werden, so kann der entsprechende Operator innerhalb der Ausgabezeile beliebig geändert werden. Möchte der Benutzer die Ausgabezeile nicht ändern, da er sich zu unsicher fühlt, können Operatoren auch durch die Radio-Buttons am Rand der Dialogbox geändert werden.

In der Ausgabezeile können neben den Operatoren auch Klammern oder Konstanten eingefügt werden. Diese Änderungen werden bei allen weiteren Manipulationen berücksichtigt. Werden in der Zeile für eine Kommunikation relevante Daten geändert, so wird das sofort erkannt und die entsprechenden Felder aktualisiert. Dabei wird besonders darauf geachtet, dass die Ausgabezeile keine Inhalte verliert.

Die Reihenfolge, in der die Kommunikationsfelder gefüllt werden, ist Grundlage dafür, in welcher Reihenfolge sie in der Quelltextzeile eingefügt werden. Dies ist wichtig bei allen Operatoren, bei denen das Kommutativgesetz nicht gültig ist, also beispielsweise `x = west(b) - ost(c)` bzw. `x = ost(c) - west(b)`.

Die unterschiedlichen Kommunikationsarten werden wie in Kapitel 5.3.2 vorgeschlagen angeboten. Dabei wird darauf geachtet, dass jeweils erklärende Tool-Tipps zur Verfügung stehen.

Die Eingabefelder des Kommunikationsfeldes besitzen zusätzlich eine undokumentierte Funktion: Setzt man in ein Feld zwei oder mehr Variablen durch Leerzeichen getrennt, so wird dies so interpretiert, dass mehrere Kommunikationen durchgeführt werden. So entsteht beispielsweise die Zuweisung `x=ost(x)+ost(y)`. Dies ist durch den unnötigen doppelten Kommunikationsaufwand normalerweise allerdings nicht sinnvoll. Müssen mindestens zwei Variablen zwischen zwei Prozessorelementen ausgetauscht werden, so sollte vorher immer eine lokale Operation durchgeführt werden, um den erhöhten Kommunikationsaufwand zu vermeiden. Da aber beispielsweise für die weiter entfernten östlichen Prozessorelemente nur ein Feld zur Verfügung gestellt wird, können auf diese Weise mehrere östliche Nachbarn angesprochen werden. Dazu ist es notwendig, dass bei der Beschriftung der Aktivierungspfeile ebenfalls mehrere Abstände durch Leerzeichen getrennt eingegeben werden. Diese Kommunikationsmöglichkeit wurde bewusst nicht dokumentiert, um bei der nach dem Benutzermodell vorgesehenen Benutzergruppe die Benutzung mehrerer Kommunikationen in einer Zuweisung zu vermeiden. Die Benutzung mehrerer Kommunikationen innerhalb einer Zuweisung wurde bereits diskutiert und Vor- und Nachteile angesprochen. Da es auch sinnvolle Einsatzzwecke gibt, wurde die Funktion trotzdem implementiert. Abbildung 6-10 zeigt den vollständigen Screenshot einer Dialogbox zur Erzeugung und Darstellung einer Kommunikation.

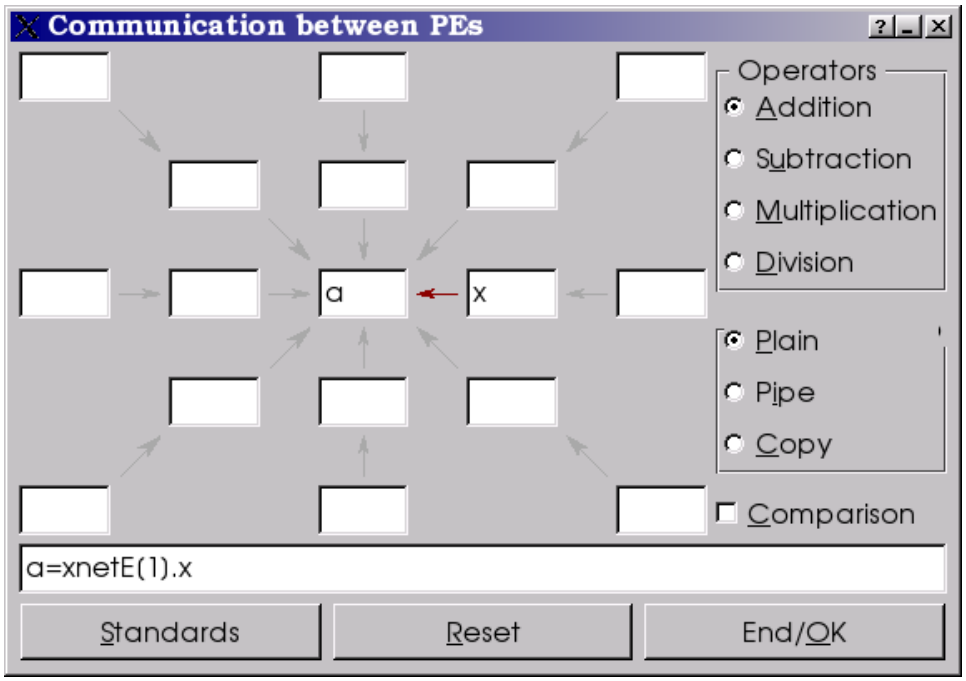


Abbildung 6-10: Screenshot einer Dialogbox zur Kommunikationszusammenstellung

Auf die Einhaltung der ergonomischen Richtlinien wurde in diesem Kapitel nicht im Detail eingegangen. Jedoch wurden sie bei der Kommunikations-Dialogbox in der gleichen Weise berücksichtigt, wie bei der Aktivierungs-Dialogbox.

6.3.5 Implementierung weiterer Eigenschaften

Viele der im Folgenden vorgestellten Mechanismen wurden bereits in [Tou96] besprochen. Zum damaligen Zeitpunkt waren diese nur teilweise und nur sehr vereinzelt in kleineren Entwicklungsumgebungen vorhanden. Unter Linux oder noch spezieller für Parallelrechner wurden fast nur Hilfestellungen in Form von Bedienungshandbüchern oder so genannten „Manpages“ angeboten.

Tool-Tipps

Tool-Tipps werden für alle Elemente angeboten, die von der Klasse `RTFDrawing` abgeleitet werden. Somit sind alle Icons, welche für die Aktivierung, Kommunikation oder das Falten von Text eingesetzt werden, mit dieser Funktion ausgestattet. Grundlage dafür ist die virtuelle Methode `text` der `RTFDrawing` –Klasse, in der jeweils der in einem Tool-Tipp anzuzeigende Text gespeichert wird. Der Text besteht bei der Aktivierung und Kommunikation jeweils aus dem Quellcode, der bei der Eingabe über die Dialogboxen erzeugt wird. Beim Falten wird der verborgene Text zurückgegeben.

Eingebettet ist die Funktion direkt auf der Oberfläche des Editors, dem `rtfwidget`. Dort wird entschieden, welches Zeichen bzw. Glyph an welche Position auf dem Bildschirm geschrieben wird. Ist als Glyph ein `RTFDrawing` zu zeichnen, wird zusätzlich ein Tool-Tipp initiiert, der über die von Qt zur Verfügung gestellten Funktionen integriert wird.

Falten

Das Falten von Text ließ sich durch den Aufbau des Klassenbaums relativ leicht implementieren. Die virtuelle Klasse `RTFDrawing` bietet alle benötigten Voraussetzungen. Statt wie ursprünglich geplant mit der Klasse nur die Icons für eine Aktivierung oder Kommunikation zu integrieren, können beliebige Klassen abgeleitet werden, welche ein beliebiges `Drawing`-Objekt im Editor abbilden und einen dazugehörigen Quelltext in der Methode `text` abspeichern. Als einfaches Implementierungsbeispiel wurde als `Drawing`-Objekt die erste Zeile des zu verbergenden Quelltextes mit einem gelben Hintergrund gewählt, der gesamte Quelltext wird über die Methode `text` gespeichert.

Projektverwaltung, Makefileerzeugung

Im Editor PEdit wurde auch eine Projektverwaltung integriert. Die Aufgabe der Projektverwaltung ist die Darstellung der Zusammenhänge zwischen einzelnen Projektdateien. Zusätzlich kann über diese Informationen die Erzeugung von so genannten Makefiles unterstützt werden, welche die Compilierung des Gesamtprojektes steuern.

Obwohl die Implementierung relativ rudimentär ausgefallen ist, ist der Entwicklungsaufwand groß. Abbildung 6-11 zeigt dazu die Abhängigkeiten der beteiligten Klassen.

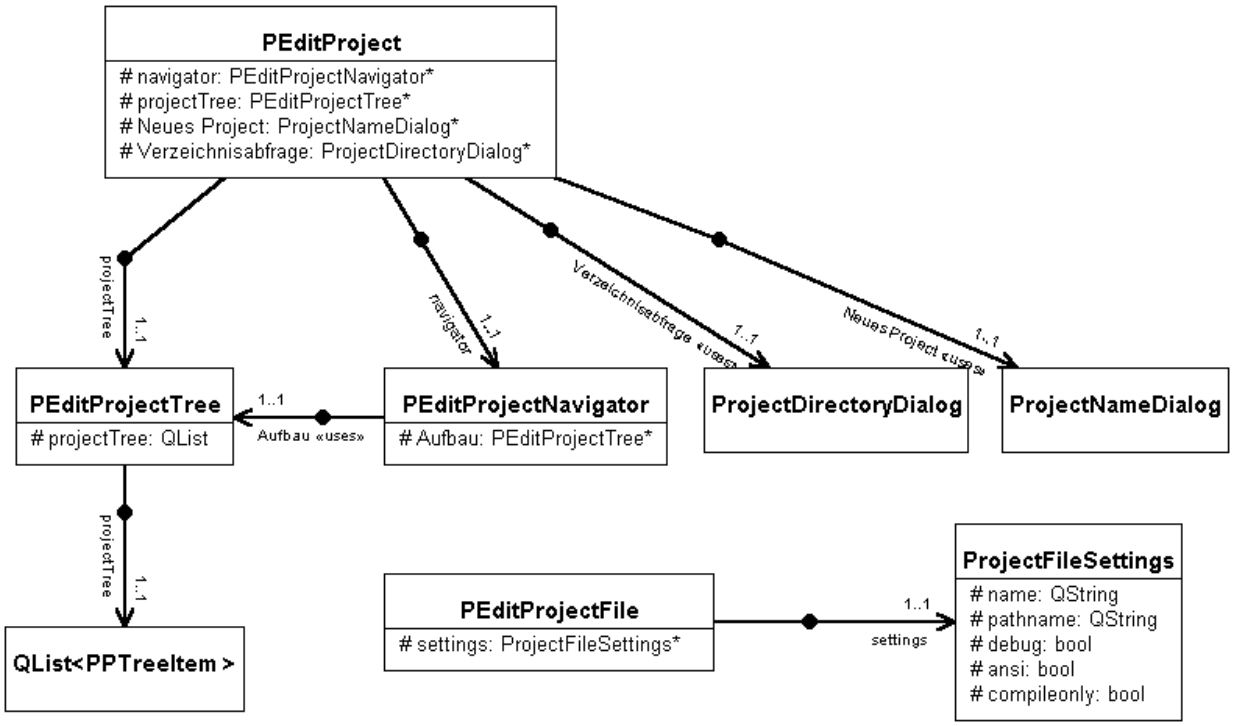


Abbildung 6-11: Klassen für die Projektverwaltung (aus [Hir98])

In der Implementierung soll zeitgleich nur ein Projekt geöffnet werden. Aus diesem Grund wurde das Entwurfsmuster „Singleton“ nach [GHJV96] eingesetzt, welches diese Prüfung durchführt. Die Klasse `PEditProject` kontrolliert das Projekt, `PEditProjectTree` kapselt den Abhängigkeitsbaum der Projektdateien und `PEditProjectNavigator` ist dafür zuständig, dass ein Navigationsmenü auf dem Bildschirm angezeigt werden kann, über das durch Doppelklick weitere Dateien des Projektes geöffnet

werden können bzw. bereits geöffnete Dateien in den Vordergrund geholt werden. `PEditProjectFile` steuert die Erzeugung des Makefiles, wobei die notwendigen Daten aus der Klasse `ProjectFileSettings` abgefragt werden. Die automatische Erzeugung des Makefiles wurde nur in Grundzügen implementiert und funktioniert nur bei einfachen Projekten. Komplexe Abhängigkeiten, beispielsweise mit der Integration unterschiedlicher Compiler können nicht erkannt werden. An dieser Stelle könnte das Programm eine wesentlich bessere Unterstützung liefern. Da aber die Projektverwaltung keine typisch parallele Anwendung ist, wurde hierauf kein Schwerpunkt gesetzt.

Syntax-Highlighting

Das Syntax-Highlighting ist mittlerweile aus jeder guten Entwicklungsumgebung bekannt. Beim vorliegenden Editor sollten die bekannten Schlüsselbefehle allerdings nicht nur hervorgehoben werden, sondern parallele Befehle sollten von sequentiellen unterscheidbar sein. Dazu musste die zugrunde liegende Grammatik erweitert werden. Zur Auswertung von Schlüsselbefehlen werden die Texte aus dem Editor an die Open Source Produkte lex und yacc [Her03] übergeben, die genauso wie die alternativen Produkte flex und bison eine Schnittstelle für Grammatiken und Suchfunktionen zur Verfügung stellen. In der Implementierung werden damit C-Schlüsselwörter, C++-Schlüsselwörter, parallele Schlüsselwörter, Präprozessorbefehle, Strings, Kommentare, Variablen und Konstanten unterschieden.

Eine Überprüfung findet zu Beginn beim Laden des Quelltextes und anschließend bei jeder Änderung im Editorfeld statt. Wird ein hervorzuhebendes Element gefunden, so wird in einer Tabelle nachgesehen, wie dieses darzustellen ist (beispielsweise grau, fett, kursiv, rot,…). Durch die Speicherung in einer Tabelle kann die Information, die bei einem einzelnen Zeichen gespeichert werden muss, sehr klein gehalten werden Das hier angewandte Konzept des Fliegengewichtsmusters wurde bereits in Kapitel 6.3.2 besprochen.

Kontextsensitive Hilfe / Unterstützung von Befehlsgruppen

Die kontextsensitive Hilfe wurde so umgesetzt, dass ein Benutzer diese beliebig erweitern kann. Über die Funktionstaste F1 wird das in Kapitel 5.2.3 beschriebene Hilfedialogfenster mit direkter Eingabefunktion aufgerufen. Um zu entscheiden, welche Hilfe für welchen Befehl aufgerufen werden muss, wird der Text unter der aktuellen Cursorposition im Editor analysiert, indem alle in der Umgebung liegenden, zusammenhängenden Buchstaben zu einem „Wort“ zusammengefasst werden. Für das „Wort“ sucht das Programm im Hilfeverzeichnis eine gleichnamige ASCII-Datei. In dieser ASCII-Datei wird in der ersten Zeile eine Kurzbeschreibung der Syntax zur Verfügung gestellt. In den folgenden Zeilen folgen die Parameter, die durch einen Doppelpunkt abgeschlossen werden. Folgt hinter dem Doppelpunkt noch Text, wird dieser als Standardwert für den Parameter angesehen. Für das Hilfe-Dialogfenster werden diese Informationen übernommen und die Kurzbeschreibung angezeigt. Für jedes einzelne Argument wird ein beschriftetes Eingabefeld zur Verfügung gestellt. Ist zusätzlich noch eine gleichnamige

HTML-Datei vorhanden, so wird diese in korrekter Syntax mit Überschriften, integrierten Bildern und Verweisen angezeigt. Hier ist der eigentliche Hilfetext zu finden, der für das Ausfüllen der Argumentfelder zur Verfügung gestellt werden soll. Bei jeder Änderung eines Argumentes wird aus Gründen der Lernförderlichkeit in einer Ausgabezeile der komplettierte Befehl angezeigt und zum Abschluss an den Editor zurückgegeben. Wurden beim Aufruf hinter dem „Wort“ unter dem Cursor noch Argumente erkannt, so werden deren Werte ebenfalls an die Dialogbox übergeben und in der Reihenfolge der auszufüllenden Argumente in die entsprechend identifizierten Eingabefelder übergeben.

Das gefundene „Wort“ entspricht im Normalfall einem Befehl, für den eine Hilfestellung gegeben werden soll. Da für jeden Befehl nur eine ASCII-Datei und eine HTML-Datei im Hilfeordner vorliegen müssen, kann der Entwickler selbst beliebige Hilfestellungen erweitern. So können beispielsweise für alle selbst geschriebenen Methoden Hilfen hinzugefügt werden, die teilweise sogar automatisiert durch Dokumentationstools wie doxygen [Pet03] erzeugt werden können. Durch eine konsequente Ausnutzung von automatisierten Dokumentationstools könnten für alle selbst geschriebenen Methoden Hilfe-Dialogboxen zur Verfügung gestellt werden, ohne dass ein Benutzer weitere Eingaben durchführen muss. Diese Automatisierung wurde allerdings nicht mehr implementiert, da dazu jeweils eine Anpassung an das vom Benutzer bevorzugte Dokumentationstool notwendig wird.

Statt der Nutzung einer ASCII-Datei, welche ein proprietäres Format besitzt, wäre der Einsatz einer XML-Datei möglich gewesen. Eine XML-Datei könnte flexibler an Erweiterungen angepasst werden. Allerdings ist der Aufbau der Datei so einfach (erste Zeile Syntaxbeschreibung, weitere Zeilen Argumente), dass der Einsatz von XML nicht zu einer Vereinfachung führen würde. Da in einem XML-Dokument die Reihenfolge der Einträge unberücksichtigt bleibt, wäre sogar ein zusätzlicher Parameter notwendig, welcher die Position der jeweiligen Argumente beschreibt. Außerdem wäre es für XML-unerfahrene Personen unübersichtlicher, neue Hilfedateien zu erstellen. Aus diesem Grund wurde bewusst die einfache ASCII-Datei eingesetzt.

Flexible Initialisierung

Die „flexible Initialisierung“ wurde nur teilweise implementiert. In einer Initialisierungsdatei können für vorgegebene Parameter der Entwicklungsumgebung entsprechende Werte über ASCII-Text übergeben werden. So sind Funktionen vorgesehen, über die Texte, beliebige Integer- oder Realzahlen oder Farbwerte übergeben werden können. Die Dialogboxen für die Kommunikation und Aktivierung wurden damit ausgestattet. Hier sind beispielhaft einige Werte wieder gegeben:

```
dkoLineTT = li. Taste: Aktivierung\nre. Taste: Entfernung
                          // Tool-Tipp für Pfeile
dkOpTitle = Operatoren    // Beschriftung eines Bereichs
dkOpAdd = &Addition       // Beschriftung eines Radio-Buttons
dkBtnEnd = End/&OK        // Beschriftung von Text-Buttons
```

```
ArrLenH = 0.4            // relative Länge von Pfeilen
ArrColAkt = darkRed      // Farbe von aktiven Pfeilen
```

In der ersten Zeile des Beispiels ist zusätzlich zu sehen, dass Tool-Tipp-Texte in gewissem Maße auch formatiert werden können (hier durch ein „`\n`“, welches einen Zeilenumbruch repräsentiert). Durch die Initialisierungsdatei können vom Benutzer selbst alle Beschriftungen, Farben, Längen der Pfeile, usw. selbst bestimmt werden. Bei einer konsequenten Umsetzung der Initialisierung kann ein sehr hoher Grad an Individualisierbarkeit erreicht werden. Allgemein können dadurch beispielsweise sehr einfach national angepasste Versionen des Editors zur Verfügung gestellt werden.

Auch bei der Initialisierungsdatei wurde auf eine XML-Codierung verzichtet. Der Grund liegt in dem einfachen Aufbau der Datei, der vorsieht, dass eine Anweisung immer innerhalb einer Zeile durchgeführt werden kann und keine Abhängigkeiten zwischen den Einträgen bestehen.

Klammerüberwachung, automatisches Einrücken, Drag&Drop, Fensterverwaltung, Bookmarks

Die meisten der weiteren zur Hilfestellung implementierten Funktionen wurden bereits in Kapitel 5.1.7 beschrieben.

Eine Klammerüberwachung ist relativ leicht zu implementieren. Wird beispielsweise eine schließende Klammer eingegeben, so wird im Text so lange nach vorne gesucht, bis eine dazu passende öffnende Klammer gefunden wird. Wurden bei der Rückwärtssuche weitere schließende Klammern gefunden, so muss zuerst deren öffnende Klammer gefunden werden, bis schließlich die zur aktuell schließenden Klammer gehörende öffnende gefunden wurde. Anschließend können durch einen Timer gesteuert beide Klammern kurzzeitig in einem anderen Stil visualisiert werden (z.B. fett), damit der Zusammenhang dargestellt werden kann. Zu beachten ist bei der Suche nach der dazugehörigen Klammer, dass Kommentare und Strings nicht berücksichtigt werden dürfen.

Ein automatisches Einrücken ist leicht zu realisieren, indem beim Einfügen des Zeilenumbruchs (Klasse `RTFReturn`) die vorherige Zeile analysiert wird und automatisch in der neuen Zeile die gleiche Anzahl an „Whitespaces“ (Leerzeichen oder Tabulatoren) eingefügt wird.

Drag&Drop ist relativ aufwändig zu implementieren. Hier müssen aktive Markierungen überprüft, sowie Mausbewegungen und –klicks interpretiert und anschließend die vorhanden Methoden für „Ausschneiden“ und Kopieren“, die für die Bearbeitung per Tastatur notwendig sind, korrekt angesprochen werden.

Eine Fensterverwaltung erlaubt es, dass ein Programm aus mehreren nahezu unabhängigen Fenstern besteht. Beschrieben wurde das Vorgehen in der Diplomarbeit von Eric Hirsch [Hir98]. Microsoft hat die Möglichkeit, mehrere Fenster zu benutzen, in seinem Office Paket 2000 eingeführt. Bei der Implementierung ist darauf zu achten, dass die

Fenster Informationen austauschen können, wer welche Datei offen hat, um jeweils komfortabel per Klick zwischen den Fenstern wechseln zu können. Zusätzlich müssen beim Öffnen und Schließen einzelner Fenster einige Überprüfungen stattfinden, beispielsweise ob Sprungmöglichkeiten zu bereits geöffneten Dateien angeboten werden müssen, Bookmarks übernommen werden können oder ein Gesamtprojekt gespeichert werden soll.

Die Bookmarks wurden nur sehr vereinfacht implementiert. Man kann einem markierten Text über eine Tastenkombination einen Namen zuweisen, der anschließend in ein Menü eingetragen wird. Durch die Auswahl des entsprechenden Menüeintrages kann man später jeweils aus einem beliebigen Fenster zur markierten Stelle zurückspringen.

Weitere Funktionen, wie das Drucken oder die automatische Weitergabe von Bookmarkinformationen zwischen mehreren Programminstanzen, sollen hier nicht besprochen werden, obwohl dies auch Bausteine eines „guten“ Editors sind.

6.3.6 Erweiterungsmöglichkeiten

Für den Editor PEdit sind noch sehr viele Erweiterungen denkbar. Viele wurden bereits in Kapitel 5 vorgestellt und sollen hier nur kurz zusammengefasst werden.

Dies beginnt bei „Kleinigkeiten“, wie dem Einbau eines Hilfe-Buttons bei den Dialogboxen, die durch reine Fleißarbeit zu erledigen sind. Zusätzlich kann in den Standardformen der Dialogboxen für Aktivierung und Kommunikation die Möglichkeit geboten werden, benutzerspezifische Konfigurationen zu speichern, damit bei einer mehrfachen Nutzung einer komplexen Anordnung diese nicht jedes Mal vollständig eingegeben werden muss.

Würde man einen Parser im Hintergrund während der Quellcodeentwicklung einsetzen, könnte man sofort auf Syntaxfehler hinweisen oder die Auswertung von parallelen und sequentiellen Variablen vereinfachen, so dass bessere Hilfestellungen gegeben werden können, bzw. die Dialogboxen in Fällen eingesetzt werden, für die im Moment Hintergrundinformationen fehlen. Der Einbau eines solchen Parsers benötigt allerdings eine enge Kooperation mit dem Compilerhersteller, um notwendige Informationen in geeigneter Weise austauschen zu können.

Durch einen integrierten Compiler wäre es unter Umständen auch möglich, einen Debugmodus einzuführen, so dass man den Quelltext an jeder beliebigen Stelle unterbrechen kann, um Zwischenergebnisse betrachten zu können. Obwohl der Debugmodus bei Entwicklungsumgebungen für sequentielle Rechner häufig zur Verfügung steht, muss man beachten, dass Parallelrechner im Allgemeinen nur im Batchbetrieb betrieben werden, um eine möglichst hohe Auslastung zu erreichen. Bei einem solchen Betrieb wäre der Debugmodus nicht sinnvoll einsetzbar, da hier immer eine Benutzerinteraktion notwendig wird. Könnte man aber zusätzlich eine Simulation des Parallelrechners zur Verfügung stellen, wäre einerseits der Debugmodus realisierbar und weiterhin könnten viele kleinere Testszenarien durchgespielt werden, um Fehler zu finden, ohne dass dadurch der leistungsfähige Parallelrechner benötigt wird.

7 Zusätzliche Anwendungsgebiete für grafische Hilfesysteme

Im letzten Kapitel wurde eine Implementierung vorgestellt, mit der grafische Hilfesysteme für einen klassischen SIMD-Rechner zur Verfügung gestellt werden können. In diesem Kapitel soll gezeigt werden, dass grafische Hilfen nicht nur für die klassischen SIMD-Rechner eingesetzt werden können, sondern auch bei vielen ähnlichen Rechnerarten. So wird eine Designstudie vorgestellt, welche für die Realisierung einer Entwicklungsumgebung für systolische Arrays eingesetzt werden kann. Dabei wird besonders das Pipelining des Rechners betrachtet. Anschließend wird kurz angesprochen, wie eine Unterstützung bei Rechnermodellen von rekonfigurierbaren Netzen (RMeshs) und so genannten Honeycombs aussehen könnte. FPGAs sind ebenfalls Rechenfelder, deren Programmierung durch grafische Unterstützung erleichtert werden könnte. Auch das Modell von parallelen optischen Bussen wird kurz angesprochen.

MPI und PVM-Systeme bilden eine ganz andere Klasse von Rechnern. Hier soll nur vorgestellt werden, welche weiteren Methoden der grafischen Befehlsunterstützung möglich wären. Damit die gezeigten Vorteile einer grafischen Befehlsunterstützung in bestehende Entwicklungsumgebungen integriert werden können, werden abschließend so genannte schwebende Werkzeuge vorgestellt. Dabei wird ein Vorschlag gemacht, wie die herausgearbeiteten Vorteile der grafischen Parallel-Unterstützung auf modernen Einprozessorentwicklungsumgebungen nutzbar gemacht bzw. die Umgebungen der Parallelrechnerhersteller erweitert werden können.

7.1 Systolische Arrays (Systola)

In Kapitel 3.4 wurde die Betriebsweise eines systolischen Arrays am Beispiel der Systola-1024 bereits vorgestellt. Dabei wurden die beiden präsentierten Algorithmen an Hand von Grafiken erklärt. Genau diese Art von Grafiken soll nun auch bei der Entwicklung der entsprechenden Programme zum Einsatz kommen.

Betrachtet man das Konzept der Parallelrechner-Karte genauer, so teilt sich ein Programm in drei Teile.

Der erste Teil dient zur Initialisierung und zum Aufruf der notwendigen Algorithmen und wird in der Wirtssprache des Rechners durchgeführt. Dementsprechend wird hier die Entwicklungsumgebung der Wirtssprache genutzt. Eine Benutzungsunterstützung in diesem Teil wäre durch so genannte „schwebende Werkzeuge“ möglich, wie sie später in Kapitel 7.4 vorgestellt werden. Hier soll aber darauf nicht näher eingegangen werden.

Der zweite Teil besteht aus einem Controllerprogramm. Es dient zur Kommunikation zwischen dem PC und der Rechenkarte. Dabei werden alle Daten auf die Rechenkarte geladen und an die Randprozessoren verteilt. Zusätzlich werden die zur Verarbeitung der Daten auf dem systolischen Array notwendigen Algorithmen geladen. Eine Parallelisierung liegt hier nicht vor, da die Kommunikation mit dem PC und den Randprozessoren von nur einem Controller gesteuert wird.

Der mitgelieferte Editor leistet dem Benutzer bei der Entwicklung der Controllerprogramme eine recht gute Unterstützung, auch wenn die Darstellung in einer an MS-Excel erinnernden Eingabematrix zuerst recht ungewohnt ist. Durch die vorgegebenen Felder kann jeweils schnell erkannt werden, wo welche Befehle genutzt und welche Parameter angehängt werden. Bei modernen Umgebungen würde man mehr Hilfsmöglichkeiten in der Form erwarten, wie sie in den letzten Kapiteln vorgestellt wurden. Diese Hilfen würden die ungewohnte Programmierung in vielen Fällen unterstützen. Bei der Realisierung von Projekten wird man allerdings nur wenige Controllerprogramme programmieren müssen, da sie nur beim Start bzw. Ende eines parallelen Programms zum Einsatz kommen. Zusätzlich ist der Aufbau aller Controllerprogramme sehr ähnlich, so dass häufig bestehende Programme kopiert und durch kleine Anpassungen in das neue Projekt übernommen werden können. Aus diesen Gründen wird hier die Entwicklungsumgebung für diese Programmteile nicht näher betrachtet, obwohl es auch in dieser Umgebung sinnvoll wäre, wenn Programmteile durch entsprechende Icons symbolisiert werden könnten, welche die Kommunikation betreffen.

Der dritte und größte Teil eines Systola-Programms besteht aus den eigentlichen Algorithmen, die auf dem Prozessorfeld abgearbeitet werden sollen. Hier werden, wie in Kapitel 3.4 gezeigt, die typischen SIMD-Operationen wie Aktivierung und Kommunikation eingesetzt. Als Programmiersprache wird das herstellereigene LAISA zur Verfügung gestellt.

Von den Entwicklern der Systola-1024 wird dazu ein Editor mitgeliefert, der in der Aufmachung dem unter Windows bekannten „notepad“ entspricht (ein Screenshot wurde bereits in Abbildung 4-6 gezeigt). Unter Linux kann er am ehesten mit einer stark

vereinfachten Version von XEmacs oder einem Editor wie „Kate“ ohne dessen Zusatzfunktionen verglichen werden. In einer Icon-Leiste stehen die Funktionen Öffnen, Speichern, Drucken, Ausschneiden, Kopieren und Einfügen zur Verfügung. Zusätzlich gibt es zwei Icons, welche das Compilieren aus dem Quelltext oder einen Wechsel zum Controller-Editor erlauben. Besonders die Möglichkeit des Compilierens mit anschließender Fehleranzeige hilft dem Anwender, Fehler schnell korrigieren zu können.

Die grafische Unterstützung dieser Arbeit soll im Wesentlichen schon bei der Befehlseingabe helfen, Fehler zu vermeiden. Deshalb wurde in einer Konzeptstudie das Aussehen des Editors wesentlich erweitert und um grafische Komponenten ergänzt. Auf eine komplette Ausarbeitung einer Entwicklungsumgebung wurde im Rahmen der Arbeit verzichtet.

Abbildung 7-1 zeigt einen Screenshot des neuen Eingabefensters. Die hier folgenden Beschreibungen sollen verdeutlichen, dass durch den Einsatz grafischer Komponenten die Programmierung von LAISA-Programmen erleichtert werden kann, indem zum einen syntaktische Fehler vermieden werden und zum anderen logische Fehler einfach erkannt werden können.

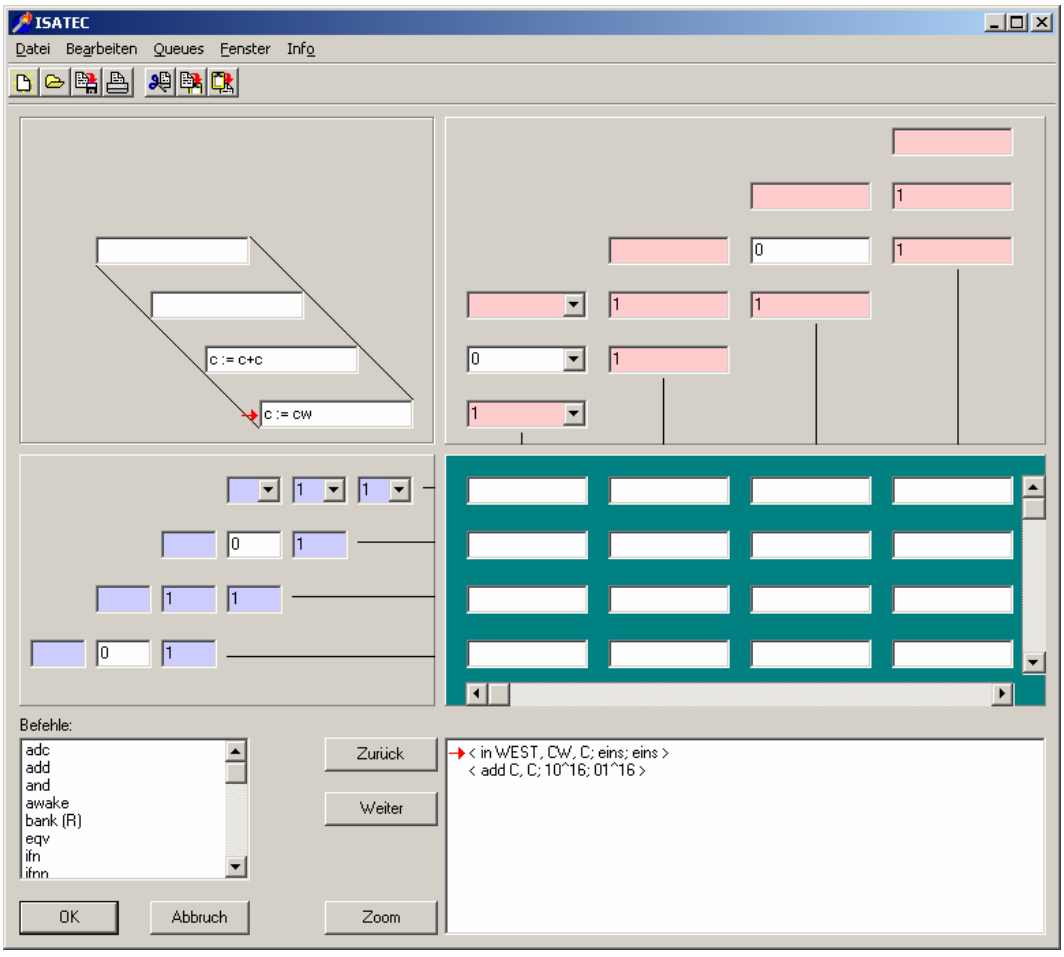


Abbildung 7-1: Screenshot eines erweiterten LAISA-Editors

Das ursprüngliche Editorfeld ist innerhalb des Screenshots im unteren rechten Feld wieder zu finden. Hier kann genau wie im Original-Editor der Quelltext vom Programmierer direkt eingegeben werden. Dadurch wird Benutzern, die bisher mit dem Originaledi-

tor gearbeitet haben, der Umstieg erleichtert. Alle Änderungen werden zeitgleich sowohl im Editorfeld als auch in den weiteren Eingabefeldern aktualisiert. Somit können einerseits bei Eingabe des Originalquelltextes die vom Programm neu angebotenen Funktionen erkannt und andererseits bei Eingabe in den neuen Feldern der resultierende Quelltext verfolgt werden.

Zusätzlich ist im Editorfeld in der ersten Zeile ein roter Pfeil zu sehen. Dieser erscheint automatisch, um den Befehl leichter als denjenigen erkennbar zu machen, der eine Kommunikation zwischen den Prozessoren auslöst. In diesem Fall wird verdeutlicht, dass eine Kommunikation von links nach rechts stattfindet. Genauso können weitere Kommunikationsrichtungen durch entsprechende Pfeile dargestellt werden.

Um die Eingabe des Quelltextes zu vereinfachen, werden links neben dem Editorfeld in einem Auswahl-Feld alle verfügbaren Befehle aufgelistet. Bei einem Einfachklick wird der Befehl in das Editorfeld übertragen, wobei alle Parameter bereits mit Bemerkungsfeldern gefüllt sind. An Stelle der Kommentare müssen die Konstanten oder Variablen eingetragen werden. Da die Anzahl der Parameter und deren Positionen vorgegeben werden, können Syntaxfehler vermieden werden.

Der Einsatz dieses Auswahl-Feldes mit allen verfügbaren Befehlen ist besonders sinnvoll, da die gleiche Möglichkeit auch beim herstellereigenen Controller-Editor besteht. Möchte man die Erwartungskonformität erfüllen, so sollte der gleiche Mechanismus auch hier verwendet werden.

Eine darüber hinaus gehende Unterstützung erhält man, wenn man statt des Einfachklicks einen Doppelklick auf einen Befehl im Auswahlfeld ausführt. Dann wird eine Hilfe-Dialogbox geöffnet, in der der Befehl und alle Parameter ausführlich erklärt werden. Alle Parameter werden über getrennte Eingabefelder eingefügt und zeitgleich der korrekt zusammengesetzte resultierende Befehl in einem weiteren Feld angezeigt. Der genaue Aufbau dieser Hilfefenster wurde bereits in Kapitel 5.2.3 vorgestellt.

Wünscht der Benutzer keine Hilfe durch die Auswahlbox an Befehlen, sollte das Editorfeld als weitere Unterstützung die automatische Befehlsergänzung beherrschen, wie sie in Kapitel 5.2.5 beschrieben wurde.

Um das Vorgehen auf dem Prozessorfeld zu veranschaulichen und semantische Fehler zu vermeiden, ist der obere Bereich des Eingabefensters vorgesehen. Hier wird ein Ausschnitt des Prozessorfelds gezeigt. Zusätzlich sind die Aktivierungsvektoren für die Zeilen und Spalten zu sehen, die versetzt abgebildet werden. Dadurch wird die Verzögerung dargestellt, mit der die Aktivierungen ausgeführt werden. Als letztes ist noch die Befehlspipeline zu sehen.

Die Befehle des Editorfeldes werden zeitgleich mit der Eingabe im Editorfeld in den entsprechenden Prozessor-, Befehls- und Aktivierungsfeldern angezeigt. Dabei kann der auszuführende Befehl in der Befehlspipeline und im Prozessorfeld wahlweise in der Syntax der Wirtssprache, also Pascal oder C(++) dargestellt werden oder in der original LAISA-Sprache wie im Editorfeld. Auch hier kann wieder vor dem Befehl der Pfeil dargestellt werden, um anzudeuten, dass durch diesen Befehl eine Kommunikation

ausgelöst wird. Um zu zeigen, dass Befehle durch Bedingungen oder Schleifen zu einem Block zusammengefasst sind, können Blöcke durch unterschiedliche Farben voneinander abgehoben werden. Dadurch ist schnell zu sehen, wie groß beispielsweise ein Bedingungsblock ist und man erhält einen Eindruck, wie lange Teile des Prozessorfeldes deaktiviert werden.

Neben der Darstellung der Befehle in der Pipeline bzw. auf dem Feld können diese auch dort direkt verändert werden. Die Änderungen werden sofort wieder in korrekter LAISA-Syntax im Editorfeld dargestellt. Die Nutzung der Wirtssprache im Prozessorfeld hat den Vorteil, dass hier bei den meisten Benutzern eine wesentlich größere Erfahrung besteht und deshalb syntaktische Fehler leichter vermieden werden. Die Konvertierung in LAISA-Befehle kann mit Hilfe eines Parsers relativ leicht durchgeführt werden, solange ein entsprechender LAISA-Befehl existiert. Ansonsten muss eine Fehlermeldung ausgegeben werden.

Standardmäßig wird der erste Befehl eines Programms in das am nächsten zum Prozessorfeld liegende Feld der Befehlspipeline eingetragen. Der nächste Befehl davor, usw., solange, bis alle sichtbaren Felder der Pipeline gefüllt sind. Zusätzlich kann man durch die Buttons „Weiter“ und „Zurück“ die Befehle auf den Feldern verschieben, so dass man erkennen kann, wie die Befehle bei der Programmabarbeitung durch das Prozessorfeld bewegt werden. Wo ein Programmausschnitt auf dem Prozessorfeld dargestellt werden soll, kann somit frei gewählt werden.

Durch die Charakteristik der Systola-1024 wird ein Befehl durch alle Prozessoren geschoben, allerdings soll meist nicht jeder Befehl auf jedem Prozessor ausgeführt werden. Die Aktivierung der Befehle geschieht in der Sprache LAISA in der Regel durch die letzten beiden Parameter eines Befehls. Hier werden die Daten für zu aktivierende Zeilen und Spalten mitgegeben. Diese Daten werden ausgewertet und in die entsprechenden Aktivierungsvektoren übernommen. Möchte man Daten direkt im Aktivierungsvektor ändern, so ist das in jedem Vektorfeld möglich. Das im Screenshot von Abbildung 7-1 dargestellte jeweils erste Feld der Aktivierungsparameter hat zusätzlich die Funktion, dass hier häufig genutzte Vektor-Varianten mit Hilfe eines Drop-Down-Feldes ausgewählt werden oder beliebige andere Werte für den Gesamtvektor in der LAISA-Syntax eingegeben werden können. So kann beispielsweise jede zweite Zeile oder Spalte durch folgende Anweisung aktiviert werden: `(10)^16`. Die binären Werte `1` und `0` sollen `16`-mal wiederholt werden. Somit können für alle 32 Zeilen bzw. Spalten in Kurzform Anweisungen eingetragen werden.

Aktive Zeilen und Spalten werden zusätzlich durch eine farbliche Unterstützung dargestellt. Da Prozessorfelder nur aktiv werden, wenn der zugehörige Zeilen- und Spaltenwert wahr ist, werden im Prozessorfeld vier Farben genutzt: das Feld ist deaktiviert, wenn das entsprechende Feld des Spalten- oder Zeilenvektors oder beide `false` sind oder das Feld ist aktiv, wenn beide Vektoren `1`-Werte besitzen. Ein kleines Beispiel wird in Abbildung 7-2 dargestellt. Hier sieht man im Ausschnitt das Prozessorfeld, wobei die in Abbildung 7-1 gezeigten Befehle bereits drei bzw. vier Schritte auf dem Prozessorfeld durchgeführt haben. Dabei wird deutlich, dass der zweite Befehl aus dem letzten

Beispiel in diesem Schritt überhaupt nicht ausgeführt wird, da immer entweder der Zeilen- oder der Spaltenvektor 0-Werte aufweisen, was durch die Farbcodierung schnell ersichtlich wird. Ändert man nun Werte für Aktivierungsspalten oder -zeilen, so kann man direkt das Ergebnis sehen und somit „am Beispiel“ die korrekten Werte für die Aktivierung ermitteln.

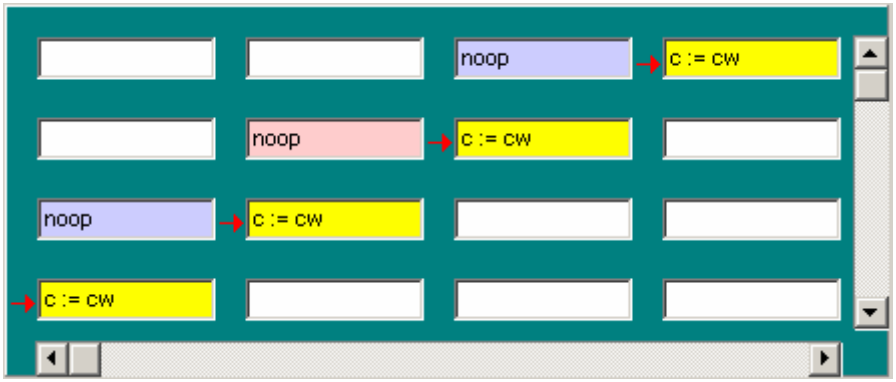


Abbildung 7-2: Aktivierung von Prozessorfeldern auf der Systola1024

Da durch die Befehlspipeline und die Aktivierungsvektoren der Platz für das Prozessorfeld eingeschränkt ist, existiert ein „Zoom“-Button. Wählt man diesen, so wird das gesamte 32x32 Prozessorfeld wahlweise auf zwei verschiedene Arten dargestellt. Dabei wird einmal gezeigt, wie sich zu einem bestimmten Zeitpunkt ein Programmteil auf einem Prozessorfeld verhält. Zum anderen wird abgebildet, wie ein einzelner Befehl in Laufe seiner Abarbeitung die einzelnen Felder aktiviert bzw. deaktiviert.

Die wichtigste Information in den Dialogboxen ist, welche Prozessorfelder aktiviert werden. Aus diesem Grund und aus Platzgründen wird der auf den Prozessoren anzuwendende Befehl nicht angezeigt. Möchte man den dahinter liegenden Befehl erfahren, so ist das über einen Tool-Tipp möglich, der aktiviert wird, wenn man mit der Maus über das entsprechende Feld fährt.

Abbildung 7-3 zeigt, wie sich die beiden Befehle des Beispiels zu einem bestimmten Zeitpunkt im Prozessorfeld verhalten.

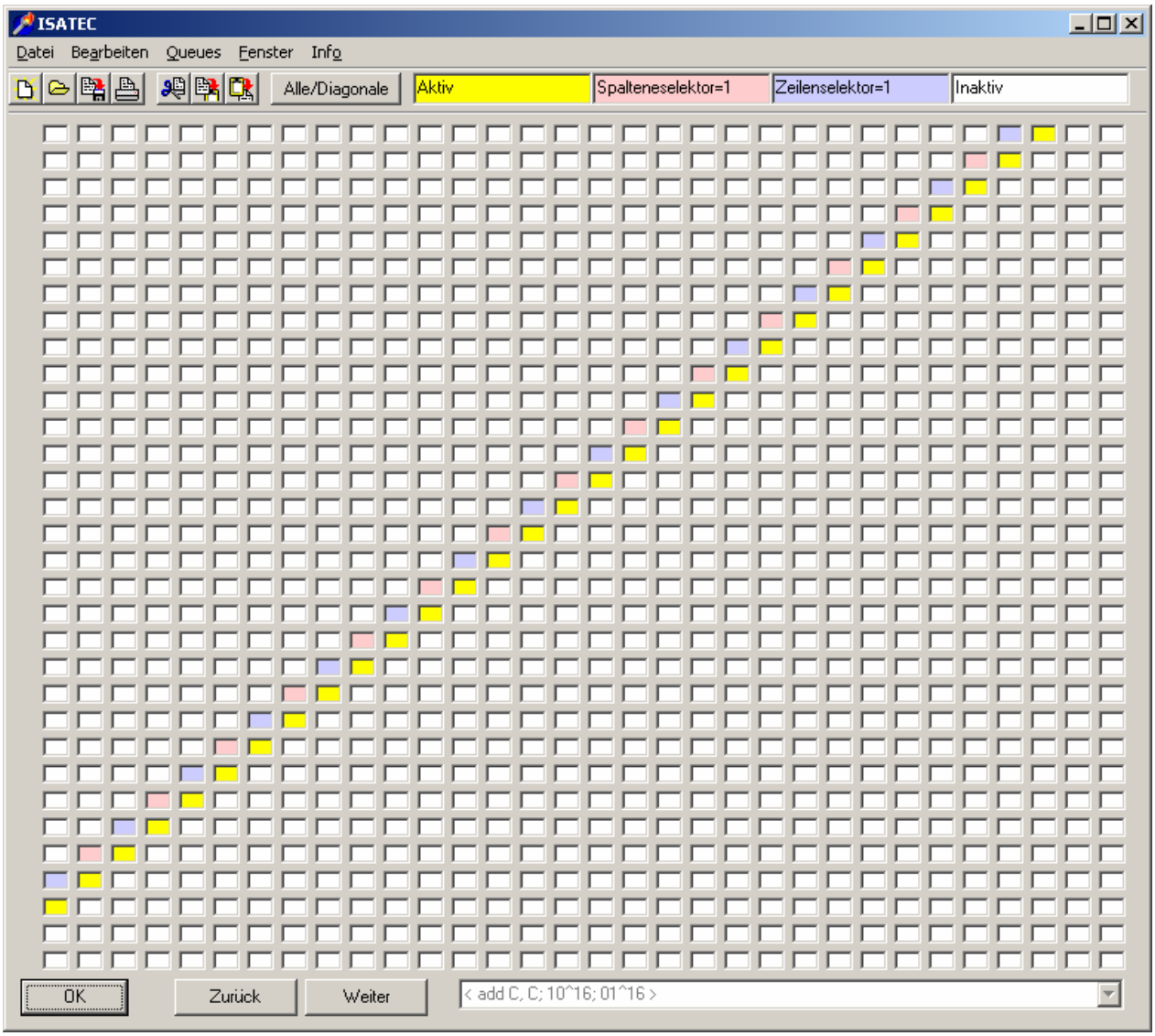


Abbildung 7-3: Darstellung der Aktivierung eines Programmteils

Die Abbildung zeigt, wie die Aktivierung für ein komplettes Programm oder einen vorher markierten Teilbereich aussehen würde. Durch die „Weiter“- und „Zurück“-Buttons kann die Aktivierung wie im Eingabefenster verändert werden, so dass die Aktivierung zu unterschiedlichen Zeiten betrachtet werden kann.

Interessiert sich der Benutzer weniger dafür, wie die Aktivierung eines kompletten Programmteils aussieht, sondern welche Auswirkungen ein einzelner Befehl beim Durchlaufen des Prozessorfeldes hat, kann man durch den Button „Alle/Diagonale“ auf eine weitere Darstellung umstellen, wie sie in Abbildung 7-4 dargestellt wird. Dabei wird automatisch ein weiteres Drop-Down-Feld am unteren Ende der Dialogbox aktiviert, in der der anzuzeigende Befehl ausgewählt werden kann. Die Auswahl an Befehlen entspricht dem Gesamtprogramm oder der vor Aktivierung der Dialogbox vorgenommenen Markierung.

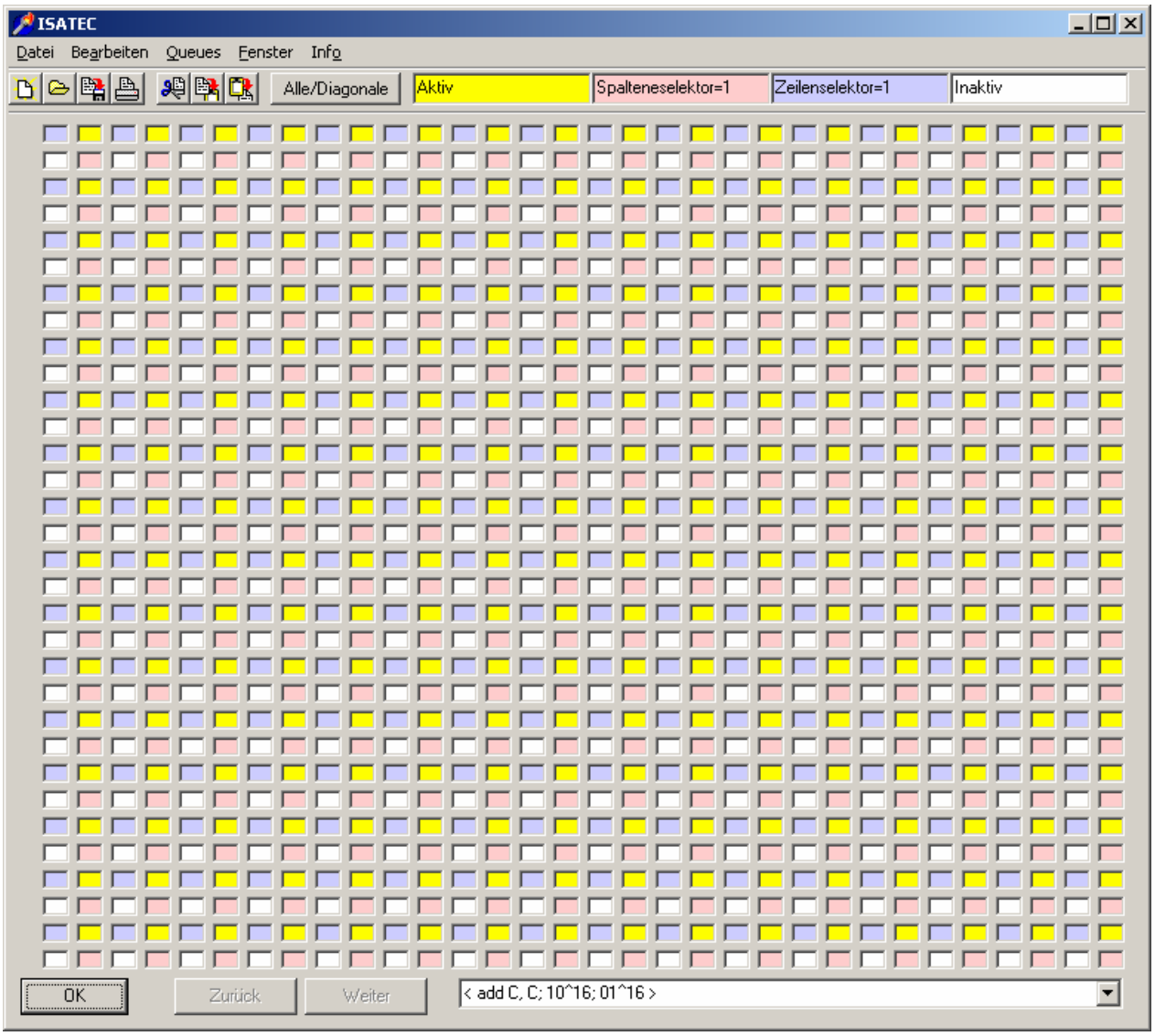


Abbildung 7-4: Darstellung der Aktivierungen eines einzelnen Befehls beim Durchlaufen des Prozessorfeldes

Die Dialogboxen zur Darstellung der Aktivierung und Deaktivierung können sehr gut dazu genutzt werden, um „Muster“ innerhalb eines Programms zu erkennen.

Durch die Übersicht können eventuelle falsche Aktivierungen oder Deaktivierungen leichter erkannt werden. Wird bei der Programmierung der Aktivierungen nicht nur mit Konstanten gearbeitet, sondern auch mit Variablen, so können nicht alle Felder korrekt dargestellt werden. Eine Lösung ist, dass die Werte von betroffenen Variablen vom Benutzer abgefragt werden, um konkrete Darstellungen realisieren zu können oder es muss durch einen leistungsstarken Parser versucht werden, möglichst viele relevante Daten zu ermitteln. Eine weitere Lösung kann sein, dass man das lauffähige Programm mit aktivierter Log-Funktion ablaufen lässt. Darin werden alle Aktivierungen protokolliert. Diese Datei muss von der Entwicklungsumgebung ausgewertet werden, und kann somit anschließend als Grundlage für die korrekte Darstellung dienen.

Durch die Einbindung der Log-Datei rückt an dieser Stelle die ursprüngliche Hauptaufgabe der Fehlervermeidung etwas in den Hintergrund, dafür wird die Fehlersuche wesentlich unterstützt.

Betrachtet man das neue Eingabefenster gegenüber dem Original, sieht man, dass das Original erhalten bleibt und durch Erweiterungen eine bessere Unterstützung für den Benutzer erreicht werden kann. Die Selbstbeschreibungsfähigkeit ist durch die möglichst realitätsnahe Abbildung des Prozessorfeldes mit seinen Eingabefunktionen vorhanden. Die einzusetzenden Unterstützungen können vom Benutzer selbst gewählt und gesteuert werden. Dies beginnt mit der Befehlsergänzung über das Drop-Down-Feld für die Befehle mit oder ohne komplexes Hilfefenster bis hin zur Darstellung der Befehle im Prozessorfeld oder der Darstellung der Aktivierung des kompletten Prozessorfeldes. Diese Unterstützungen sichern auch die Lernförderlichkeit, so dass die Vorgaben aus der Ergonomie erfüllt sind.

7.2 RMesh / Rekonfigurierbare Gitter

Da das RMesh ein allgemeines Modell für Rechner mit einem rekonfigurierbaren Gitter ist, soll hier keine realisierte Entwicklungsumgebung vorgestellt werden. Es werden aber Konzepte präsentiert, die auf den in den letzten Kapiteln vorgestellten Hilfestellungen basieren und innerhalb einer angenommenen Entwicklungsumgebung Aktivierung und Kommunikation für dieses Modell unterstützen.

In Kapitel 3.5 wurde bereits gezeigt, dass auch auf dem RMesh das Modell des massivparallelen Rechners mit Aktivierungen (hier Konfigurationen von Routen genannt), Kommunikation zwischen den Prozessoren und lokalen Operationen eingesetzt werden kann. Dabei können wie bei den bisherigen Parallelrechnertypen die Konfiguration und die Kommunikation sinnvoll durch grafische Elemente unterstützt werden, so dass der Benutzer einen Mehrwert für seine Entwicklungsumgebung erhält und wichtige Informationen schnell und übersichtlich dargestellt werden können. Hält man die „Grundsätze der Dialoggestaltung“ ein, so können besonders die Punkte Selbstbeschreibungsfähigkeit, Steuerbarkeit und Lernförderlichkeit gefördert werden.

Konfiguration des RMesh

Betrachtet man die Konfiguration eines RMesh genauer, so erkennt man, dass an dieser Stelle ähnliche Dialogboxen und Icons innerhalb des Quelltextes angeboten werden können, wie das bei der Aktivierung auf der MasPar vorgesehen ist.

Bei der Konfiguration müssen auf den einzelnen Prozessoren die Konnektoren gesetzt werden, damit anschließend über die geschalteten Wege die Kommunikation stattfinden kann. Die möglichen Konfigurationen (abhängig von der tatsächlichen Realisierung) wurden bereits in Abbildung 3-12 gezeigt.

Schreibt man gerade einen Algorithmus und möchte nur für einen Prozessor bzw. für eine bereits vorher ausgewählte Gruppe von Prozessoren eine Konfiguration durchführen, so sollte über eine Tastenkombination eine Dialogbox geöffnet werden können, in

der die gewünschte Konfiguration aus den 15 Möglichkeiten ausgewählt wird. Im Quelltext wird anschließend wahlweise symbolisch die Konfiguration durch ein dynamisch erzeugtes Icon dargestellt oder direkt der korrekte Quelltext erzeugt und eingefügt. Das Icon für eine gleichzeitige Nord-Ost und West-Süd Verbindung könnte beispielsweise wie in Abbildung 7-5 aussehen.

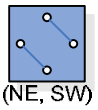


Abbildung 7-5: Dynamische erzeugtes Icon für eine Nord-Ost Süd-West Verbindung innerhalb eines Quelltexteditors

Genauere Informationen über die Befehle und deren Argumente werden wieder in dynamisch erstellten Tool-Tipps angeboten. Die Icons sind innerhalb des Quelltextes sehr schnell erkennbar, so dass auf einfache Weise ein Überblick über vorgesehene Konfigurationen gegeben werden kann. Außerdem werden Syntaxfehler vermieden, da ein Benutzer zwei Grafiken wie  und  leichter auswählen und unterscheiden kann, als dies beispielsweise durch die fehlerfreie Eingabe von zwei Befehlen wie `netNWES()` und `netNESW()` möglich wäre.

Neben der Konfiguration einzelner Prozessoren wird es häufig zu Konstellationen kommen, die ein größeres Prozessorfeld betreffen und Standardkonfigurationen erfordern, wie sie beispielsweise für den Algorithmus in Kapitel 3.5 benutzt werden. Dabei können Konfigurationen von kompletten oder teilweisen Zeilen- oder Spaltenbussen vorgesehen werden. Genauso können „Schlangen“ durch das komplette Prozessorfeld oder Diagonalen mit unterschiedlichen Abständen zwischen den Diagonalen angeboten werden. Je nach ausgewähltem Muster sollten teilweise spezialisierte Dialogboxen genauere Darstellungen anbieten oder weitere Eingaben ermöglichen. Dabei muss gewährleistet werden, dass neben der grafischen Darstellung immer gleichzeitig eine Text-Darstellung existiert und der Benutzer wahlweise eine der beiden Eingabemöglichkeiten nutzen kann und dabei die jeweils andere zeitgleich aktualisiert wird. Eine Begründung für die doppelte Bereitstellung von Informationen durch die grafische Auswahl und Quelltext wurde in Kapitel 5.3 gegeben.

Zusätzlich zu den Standardformen müssen für eine individualisierbare Umgebung häufig benötigte Muster selbst definiert und gespeichert werden können und so die Auswahlmöglichkeiten ergänzen. Dies kann in einer ähnlichen Dialogbox geschehen, wie dies für die Aktivierung von Prozessorelementen auf der MasPar vorgesehen ist. Allerdings ist zu berücksichtigen, dass eine konkrete Implementierung eines RMesh sicher Spezialisierungen vorsieht, die berücksichtigt werden müssen. In allen Fällen wird es so sein, dass die Konfiguration einzelner Prozessoren innerhalb eines Feldes individuell geändert werden können sollte. Um dies benutzungsfreundlich zu unterstützen, können an jeder Prozessordarstellung Drop-Down-Felder zur Verfügung gestellt werden, welche alle möglichen Verbindungen, wie in Abbildung 3-12 dargestellt, anbieten.

Aus Gründen der Übersichtlichkeit können nicht alle Prozessoren mit Eingabemöglichkeiten dargestellt werden. Deshalb wird wie bei der Entwicklungsumgebung für die Systola-1024 nur ein Ausschnitt des Prozessorfeldes angezeigt. Zusätzlich kann in einer weiteren Dialogbox das komplette Prozessorfeld mit allen darstellbaren Aktivierungen, aber eingeschränkten Konfigurationsmöglichkeiten, angeboten werden.

Durch die Konfiguration der Busse ist allerdings noch nicht festgelegt, welche Prozessoren anschließend Daten auf den Bus legen sollen. Deshalb muss bei der Unterstützung der Kommunikation auch eine Auswahl der Prozessoren berücksichtigt werden.

Kommunikation auf dem RMesh

Die Kommunikation auf dem RMesh läuft ab, indem an einer Stelle eine Information auf den Bus gegeben wird, die dann an allen angeschlossenen Stellen abrufbar ist.

Wer die Kommunikation anstößt, ist in den meisten Fällen entweder davon abhängig, welche Aufgabe ein bestimmter Prozessor aktuell innerhalb eines Algorithmus hat oder welchen Wert ein bestimmter Ausdruck oder eine Variable auf dem Prozessor hat. Dazu können dynamisch erzeugte Icons, wie in Abbildung 7-6 gezeigt, im Quelltext eingesetzt werden, die durch geeignete Dialogboxen erzeugt werden können.

Abbildung 7-6: Dynamisch erzeugtes Icon für eine Kommunikation innerhalb eines Quelltexteditors

Die linke Abbildung zeigt, dass der Wert der Variablen `a` auf den Bus gelegt werden soll (`a` steht über dem Pfeil). Dabei werden keine zusätzlichen Bedingungen an die Kommunikation gestellt, da diese häufig bereits durch die Programmumgebung vorgegeben ist und somit nicht erneut vorhanden sein muss. Auf der rechten Seite wird gezeigt, wie die zusätzliche Bedingung (hier `i>0`) unterhalb des Pfeils angezeigt wird.

Die Farbe für das Kommunikationsicon wurde bewusst anders gewählt, als die für die Konfiguration. Dies hat den Grund, dass beide Iconarten noch besser von einander unterschieden werden können und die gewünschten Informationen beim schnellen Scrollen durch den Text leichter gefunden werden.

Auch bei der Kommunikation können Standardformen zur Verfügung gestellt werden, die komplette Lösungen für Spezialaufgaben oder zumindest dokumentierte Quelltextskelette zur Verfügung stellen. So sind Formen wie „alle linken, Prozessoren sollen Daten verschicken“, genauso denkbar wie „alle unteren“, „alle Randprozessoren“ oder „alle Prozessoren entlang einer Diagonalen“. Zur Individualisierbarkeit tragen wieder selbst definierte Muster bei, die jeder Benutzer abspeichern kann.

Die vorgestellten Unterstützungen dienen besonders der einfachen und fehlerfreien Eingabe bestimmter Programmteile. Zusätzlich können die grafischen Darstellungen bei der Fehlersuche eingesetzt werden, besonders wenn Möglichkeiten vorhanden sind, leistungsstarke Parser oder Debugger einzusetzen und somit komplette Konfigurationen

oder Kommunikationen visualisiert werden können. Hilfreich kann dabei auch eine maschinenlesbare Logdatei sein, wie sie bei der Systola-1024 erzeugt wird.

7.3 Weitere mögliche Spezialanwendungen

Honeycomb vernetzte Systeme

Die „Honeycomb-Architektur“ wurde in den letzten Jahren entworfen. Einige grundlegende Arbeiten dazu wurden von Thomas und Becker [ThB05], Robič und Šilc [RoS93] oder Stojmenovic [Sto97] veröffentlicht. Dabei handelt es sich ähnlich wie beim RMesh um eine rekonfigurierbare Architektur. Ein großer Unterschied ist, dass die Prozessoren – hier Zellen genannt – nicht orthogonal, sondern hexagonal angeordnet sind und somit direkte Verbindungen zu sechs Nachbarn haben. Abbildung 7-7 zeigt das Vorbild des Aufbaus, die Honigwabe und die entsprechende Realisierung.

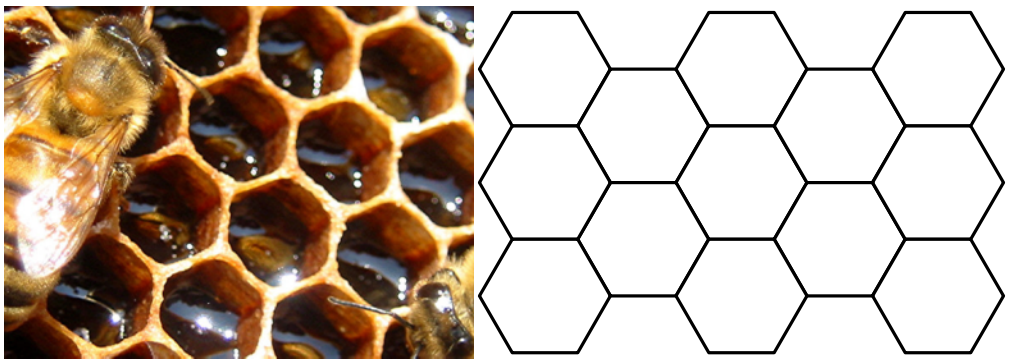


Abbildung 7-7: Aufbau der Honeycomb-Architektur

Der Rechner soll über eine „HoneyComb-Language“ (HCL) angesprochen werden. Dabei sind auch Debugging- und Simulationsumgebungen vorgesehen, die zurzeit entwickelt werden.

Da bei der Architektur während der Laufzeit des Programms Konfigurationen geändert werden sollen, ist es für den Anwender wünschenswert, wenn er diese visualisiert dargestellt bekommen kann. Zum jetzigen Zeitpunkt der Entwicklung von HCL-Programmen beschäftigt man sich noch weniger mit der Benutzungsunterstützung auf grafischer Basis, sondern es müssen erst grundlegende Tests der Programmierbarkeit geprüft werden.

Zu einem späteren Zeitpunkt sollten die in dieser Arbeit aufgezeigten Benutzungsunterstützungen sinnvoll einsetzbar werden, indem die Vorschläge, die für die Visualisierung von Aktivierung und Kommunikation paralleler Programmteile gemacht wurden, angepasst und in die Entwicklungstools eingebaut werden.

FPGA und weitere rekonfigurierbare Strukturen

Das Feld der „Field Programmable Gate Arrays“ (FPGAs) stellt ein weiteres sehr interessantes Anwendungsgebiet für grafische Benutzungsunterstützung dar. Auch in diesem Bereich können sicher viele der hier vorgestellten Mechanismen den Programmierer unterstützen. Allerdings wurde im Rahmen der Arbeit der Bereich der FPGAs

nicht näher untersucht, so dass dazu keine genaueren Vorschläge gemacht werden können.

Auch Spezialfälle von rekonfigurierbaren Systemen, wie beispielsweise rekonfigurierbare Busse werden häufig durch visuell dargestellte Algorithmen programmiert und sind somit geeignet, um die Erstellung des lauffähigen Quelltextes ebenfalls mit visuellen Mitteln zu unterstützen [ESSS+96]. Da aber jedes System spezielle Anforderungen stellt, soll die entsprechende Unterstützung hier nicht weiter verfolgt werden.

MPI/PVM

Grafische Entwicklungsumgebungen und Tools für Message-Passing Systeme wie MPI und PVM wurden bereits in Kapitel 4.3.1 besprochen. Dabei wurden sehr viele Möglichkeiten der grafischen Unterstützung vorgestellt. Allen gemeinsam ist allerdings, dass der Ansatz, parallele Programmteile im Quelltext durch grafische Komponenten hervorzuheben, nicht realisiert wird. Die Vorteile dieses Ansatzes wurden in den letzten Kapiteln ausführlich behandelt und würden auch bei Message-Passing Systemen für einen übersichtlicheren Quelltext sorgen. Nur ein Beispiel ist die grafische Darstellung von Kommunikation. Viele weitere der vorgestellten Techniken für Benutzungsunterstützung können ebenfalls eingesetzt werden. Zu berücksichtigen ist allerdings, dass Message-Passing Systeme MIMD-Systeme sind und somit auf allen Prozessoren unterschiedliche Programme laufen können. Somit ist eine Verallgemeinerung von „Einmal darstellen, überall anwenden“ nicht ohne weiteres übertragbar, so dass weitere Überlegungen angestellt werden müssen.

Dualcore und Mehrcore-Prozessoren

Ende des Jahres 2004 haben die großen Prozessorhersteller Intel und AMD erkannt, dass das Moore’sche Gesetz nicht mehr eingehalten werden kann, indem man einen einzelnen Prozessor immer weiter verkleinert und mit mehr Speicher ausstattet. Das „Gesetz“ besagt, dass alle 12-18 Monate die Leistung der Prozessoren verdoppelt werden kann [Moo65]. Stattdessen möchte man die auf einem Computer zu bearbeitenden Aufgaben teilen und auf mehrere Prozessoren verlagern, die zusammen auf einem Chip zur Verfügung gestellt werden. 2005 wurden dazu die ersten Dual-Prozessor Chips vorgestellt, die 2006 den Massenmarkt erreicht haben [Amd05, Int06]. Auch Spezialprozessoren für Grafikkarten nutzen mittlerweile diese Technik entweder intern, indem immer mehr parallele Pipelines zur Verfügung gestellt werden oder extern, indem mehrere Grafikkarten zusammen arbeiten, um die Gesamtleistungsfähigkeit zu erhöhen.

Obwohl für die meisten Anwendungen auf einem Bürorechner auch in Zukunft ein einzelner Prozessor ausreicht, so wird die Anzahl der gleichzeitig auf einem Rechner auszuführenden Programme stetig steigen, da beispielsweise der Bedarf nach „Presence und Awareness“ eines Benutzers ständig zunimmt und dazu bestimmte Anwendungen notwendig werden [Spa05]. Anderseits wird es komplexe Anwendungen geben, die auf sehr große Rechenleistungen angewiesen sind, und somit ein einzelnes Programm auf möglichst viele Prozessoren verteilt werden muss. Hier kann man an eine Textver-

arbeitung denken, die im Hintergrund die Rechtschreibung und Grammatik prüft, eventuell eine mündliche Eingabe entgegen nimmt und korrekt in Schrift umsetzt, Übersetzungen liefert oder Filme und Animationen in den Text integriert.

Durch diese kleinen Beispiele wird deutlich, dass auch Anwendungen für den Büroarbeitsplatz in Zukunft auf Mehrprozessormaschinen lauffähig sein müssen. Möchte man die Leistungsfähigkeit der jeweiligen Maschine ausnutzen, so sollten dem Programmierer ähnliche Methoden zur Verfügung gestellt werden, wie diese hier für MPI/PVM oder die klassischen Parallelrechner vorgestellt wurden.

Die genaue Darstellung einzelner Funktionen kann dazu hier noch nicht gezeigt werden, allerdings muss dem Programmierer bei den immer komplexer werdenden Programmiertätigkeiten auf unterschiedlichste Weise eine Unterstützung geboten werden. Ein Weg sind dabei die in dieser Arbeit vorgestellten grafischen Darstellungen innerhalb des Quelltextes, bzw. die Erarbeitung von Teilprogrammen durch die Unterstützung spezialisierter Dialogboxen. Am weitesten verbreitet ist diese Unterstützung im Moment bei der Entwicklung von Programmoberflächen. Diese werden in den meisten Fällen schon jetzt nicht mehr direkt in Quelltext geschrieben, sondern man kann diese mit geeigneten Werkzeugen zeichnen und den Quelltext automatisch erzeugen lassen.

Die bisher vorgestellten Werkzeuge zeigen aber, dass eine grafische Unterstützung auch bei wesentlich feingranulareren Aufgaben sinnvoll eingesetzt werden kann.

7.4 Plugins und „Schwebende Werkzeuge“

In den letzten Kapiteln wurden für unterschiedliche Parallelrechner unterschiedliche, angepasste Entwicklungsumgebungen vorgestellt. Diese optimierten Entwicklungsumgebungen zeigen jeweils, wie der Benutzer bestmöglich durch grafische Hilfen unterstützt werden kann. Trotz aller Vorteile, die diese Unterstützungen dem Benutzer bieten können, sollte in einem kommerziellen Umfeld berücksichtigt werden, dass in den fast immer vorhandenen herstellereigenen Entwicklungsumgebungen sehr viel Know How vorhanden ist, durch das ebenfalls die Erzeugung eigener Programme in vielfältiger Weise unterstützt wird. Deren Stärken liegen meist darin, dass weit mehr Informationen über den Zustand der Rechner und des aktuell laufenden Programms zur Verfügung gestellt werden können. Dazu werden dem Entwickler Tools wie ein Debugger und Auslastungs- oder Laufzeitanalysen zur Verfügung gestellt, die für die Erstellung von fehlerfreien und möglichst effizienten Programmen unerlässlich sind.

Durch die speziellen Debugger können im Falle eines Fehlers der Quellcode Schritt für Schritt abgearbeitet und gleichzeitig die jeweils aktuellen Werte von Variablen, Registern oder Aktivitätszuständen einzelner Rechnerteile angezeigt werden. Diese Debugger sind bei größeren Projekten unerlässlich, da nur damit Fehler gefunden werden, die über einfache Syntaxfehler hinausgehen.

Auslastungsanalysen zeigen, welche Teile des Rechners wann wie stark ausgelastet sind. Dadurch können Rückschlüsse gezogen werden, an welchen Stellen des Programms der Entwicklungsaufwand für eine weitere Parallelisierung lohnenswert sein

kann, bzw. warum es eventuell zu Skalierungsproblemen kommt. Wie wichtig diese Analyse ist, wurde durch die in Kapitel 3.1 vorgestellte Beziehung der Parallelisierung zu Speedup und Scaleup verdeutlicht.

Die Laufzeitanalysen können Hinweise darauf geben, welche Programmteile wie lange laufen und welche Ressourcen benötigt werden. Auch dadurch wird angezeigt, an welcher Stelle eine Weiterentwicklung für eine Laufzeitbeschleunigung sinnvoll ist.

Diese herstellereigenen Tools bieten einerseits Informationen, ohne die eine sinnvolle, effektive und effiziente Ausnutzung der Parallelrechnerressourcen nicht nutzbar ist. Andererseits werden dabei oft ergonomische Grundlagen nicht beachtet, die in dieser Arbeit als ein sehr wichtiger Grundstein für eine sinnvolle Benutzerunterstützung herausgearbeitet wurden.

Eine Möglichkeit wäre nun zu versuchen, die jeweiligen Tools nachzuprogrammieren und mit ergonomischeren Benutzungsschnittstellen auszustatten. Doch besteht diese Möglichkeit häufig nicht, da sie auf der einen Seite wirtschaftlich nicht sinnvoll ist, auf der anderen Seite stehen teilweise gar nicht die Informationen zur Verfügung, um solche Tools zu entwickeln.

Möchte man trotzdem ergonomische Benutzungsschnittstellen innerhalb der herstellereigenen Programme und Tools zur Verfügung stellen, so muss man einen anderen Weg wählen als die Neuentwicklung eigener Umgebungen.

Heutzutage bieten die großen Hersteller moderner Entwicklungsumgebungen spezielle Schnittstellen für diese Erweiterungen an, damit sie benutzer- und problembezogen angepasst werden können. Über diese Schnittstellen können mit so genannten „Plugins“ neue Programmteile hinzugefügt werden. Ein besonders gutes Beispiel ist dabei die in Kapitel 4.2.2 vorgestellte Entwicklungsumgebung Eclipse. Diese quelltextoffene Entwicklungsumgebung bietet vielfältige Schnittstellen, die von einer großen Anzahl an unabhängigen Programmierern genutzt wird, um Anpassungen durchzuführen [GaB04]. So ist Eclipse heute eine der führenden Entwicklungsumgebungen bei der Erstellung von java-Programmen. Aber auch für viele andere Programmiersprachen gibt es spezielle Erweiterungen. Zusätzlich werden sehr viele Tools angeboten, um Webseiten oder XML-Code zu erzeugen.

Mit Hilfe dieser Schnittstellen können viele der in dieser Arbeit vorgestellten grafischen Hilfestellungen innerhalb von Dialogboxen als Plugins zur Verfügung gestellt werden.

Trotz aller Vorteile wurde keine vorhandene Entwicklungsumgebung als Grundlage genommen, da bis jetzt eine der wichtigsten vorgestellten Funktionen einer grafischen Hilfestellung nicht unterstützt wird. Keine verfügbare Entwicklungsumgebung bietet die Möglichkeit an, den zugrunde liegenden Quelltexteditor um Grafiken zu erweitern. Diese Grafiken sind allerdings die Grundlage, um die in den Kapiteln 5.3.1.2 und 5.3.2.2 vorgestellten Darstellungen der Aktivierung und Kommunikation in den Quelltext integrieren zu können. Diese Darstellung wird aber als sehr wichtig angesehen, um dem Programmierer bereits bei der Codierung eine möglichst große Unterstützung für die Erstellung eines effizienten Codes anbieten zu können. Somit ist in einer wissenschaftli-

chen Umgebung die Realisierung eines kompletten Prototyps sinnvoll. Unter wirtschaftlichen Gesichtspunkten sollte allerdings überlegt werden, ob man die symbolische Darstellung im Quelltext bei dem jetzigen Stand der vorhandenen Entwicklungsumgebungen fordern kann oder darauf verzichtet und mit der Integration in marktführende Entwicklungsumgebungen ein Produkt anbietet, das am Markt schneller angeboten werden und zusätzlich von der Bekanntheit des zugrunde liegenden Editors profitieren kann.

Ein anderer Ansatz ist, nicht bei den Entwicklungsumgebungen von Programmiersprachen anzusetzen, sondern auf denen zur Webseitenentwicklung. Hier existieren mittlerweile sehr viele WYSIWYG-Programme, welche die Kombination von Text und Grafik im Editor beherrschen. Da aber ansonsten der Unterschied von Webseiten und Programmierquelltext relativ groß ist, muss hier an anderen Stellen viel Entwicklungsaufwand betrieben werden. Obwohl dieser Ansatz viel versprechend ist, wurden hier keine weiteren Untersuchungen gemacht.

Da die Bereitstellung der Plugin-Schnittstellen von kleineren Firmen häufig aus Kostengründen noch nicht standardmäßig angeboten wird, ist die Erweiterung von Entwicklungsumgebungen dieser Firmen wie oben vorgeschlagen nicht ohne weiteres möglich. Da die Hersteller von Parallelrechnern nur kleine Stückzahlen ihrer Rechner verkaufen und die Entwicklungsumgebung häufig eine „kostenlose“ Zugabe ist, sind hier Plugin-Schnittstellen meist nicht vorhanden. Aus diesem Grund soll hier noch eine andere Möglichkeit vorgestellt werden, wie vorhandene – auch ältere – Entwicklungsumgebungen mit eigenen Erweiterungen benutzungsfreundlicher ausgestattet werden können.

Moderne Betriebssysteme wie Windows oder Linux bieten mit ihren grafischen Oberflächen die Möglichkeit an, mehrere Programme gleichzeitig am Bildschirm anzuzeigen. Als Neuerung ist bei den letzten Betriebssystemversionen dazu gekommen, dass auch transparente Fenster angeboten werden können. D.h. man kann durch ein Fenster hindurch auf das darunter liegende Fenster blicken. Die Idee ist nun, dass man über die vorhandene Entwicklungsumgebung ohne geeignete Schnittstellen ein neues, transparentes Programm legt. Das transparente Programm ist aktiv und nimmt alle Benutzerinteraktionen entgegen. Mausbewegungen oder Tastatureingaben können über geeignete Betriebssystemschnittstellen an das darunter liegende Programm weiter gegeben werden, so dass der Benutzer und das Originalprogramm nicht merken, dass das transparente Programm vorhanden ist. Werden nun vom Benutzer spezielle Tastenkombinationen gedrückt, so kann das transparente Programm die in den letzten Kapiteln vorgestellten grafischen Hilfestellungen in (nicht transparenten) Dialogboxen anbieten. Dabei können die Befehle wie vorgestellt durch das Programm unterstützt erzeugt werden. Beim Schließen der Dialogbox wird das Ergebnis durch simulierte Tastendrücke an die „echte“ Entwicklungsumgebung weiter gegeben.

Etwas schwieriger ist es, wenn man die unterstützenden grafischen Dialogboxen bereits beim Öffnen mit Informationen aus der darunter liegenden Entwicklungsumgebung füllen möchte, um beispielsweise eine bereits im Quelltext vorhandene Aktivierung paralleler Prozessorelemente noch einmal anzuzeigen und eventuell ändern zu können. In diesem

Fall ist es notwendig, dass man Texte aus der Ursprungsentwicklungsumgebung ausliest. Dies sollte in den meisten Fällen durch spezielle Funktionen der Betriebssysteme bzw. deren Grafiktreiber möglich sein. Sollte das scheitern, müsste das transparente Programm die Bildschirmoberfläche auslesen und mit OCR-Methoden die relevanten Texte identifizieren.

Das transparente Programm nennt der Autor „schwebendes Werkzeug“, da es über dem eigentlichen Programm schwebt und dieses überwacht bzw. kontrolliert. Der Begriff „schwebendes Werkzeug“ wird heute in der Literatur hauptsächlich mit Werkzeugfenstern in Verbindung gebracht, die innerhalb großer Entwicklungsumgebungen von Benutzer frei platzierbar auf dem Bildschirm dargestellt werden. Dabei schweben die meist nur mit einigen Icons versehenen Fenster an beliebigen Stellen vor dem Eingabefenster, um beispielsweise die Farbe eines Pinsels ändern zu können oder von einem Rahmenwerkzeug zu einem Texteingabewerkzeug zu gelangen. Die Vielfalt dieser „schwebenden Fenster“ ist bei der auf dem Markt vorhandenen Software sehr groß. Im Gegensatz zu der hier vorgestellten Idee eines „schwebenden Werkzeugs“ sind alle bisherigen Werkzeuge Bestandteil des vorhandenen Programms oder zumindest über eine Plugin-Schnittstelle eingebunden, so dass die Zusammenarbeit zwischen Werkzeug und Programm wesentlich einfacher realisiert werden kann.

Der Idee des hier vorgestellten „schwebenden Werkzeugs“ ähnelt einem Programm der Firma Wacom, welche Smartboards herstellt. Diese aus der Konferenztechnik stammenden Geräte sehen wie große Bildschirme aus und haben die Besonderheit, dass man auf ihnen mit bestimmten Stiften „schreiben“ kann. Dabei wird das Signal des Stiftes vom Bildschirm bzw. einem zu installierenden Programm interpretiert und vor der eigentlichen Anwendung in einem ansonsten transparenten Fenster angezeigt. Damit ist es beispielsweise möglich, auf eine Powerpointpräsentation zu zeichnen. Die Zusatzinformationen werden dabei allerdings nicht in Powerpoint gespeichert, sondern müssen getrennt über das schwebende Programm verwaltet werden. Somit ist eine interaktive Präsentation auch möglich, wenn man nicht die dafür vorgesehene Betriebssystemversion „Windows XP Tablet Edition“ zur Verfügung hat.

Zusammenfassend kann gesagt werden, dass durch den Einsatz von „schwebenden Werkzeugen“ eine komplette Neuentwicklung einer Entwicklungsumgebung unnötig werden kann. Die herstellereigenen Tools können genutzt und durch den Einsatz der zusätzlichen Werkzeuge verbessert werden. Eine Umstellung auf neue Unterstützungsmöglichkeiten kann nach eigenem Ermessen und mit der eigenen Geschwindigkeit vorgenommen werden. Wichtig ist dabei aber auf jeden Fall, dass die „schwebenden Werkzeuge“ über die gleichen Tastenkombinationen wie das darunter liegende Programm zu bedienen sind. Genauso sollte das gesamte „Look and Feel“ dem Hauptprogramm entsprechen. Damit kann sichergestellt werden, dass die ergonomischen Richtlinien weitestgehend eingehalten werden können.

8 Zusammenfassung und Ausblick

Die Zusammenfassung der letzten Kapitel in einem Satz lautet: Aufbauend auf einem allgemeinen Modell für massivparallele Algorithmen und grundlegenden Arbeiten für den Entwurf einer grafischen Entwicklungsumgebung konnten in der Arbeit ergonomische Benutzungsunterstützungen erarbeitet werden, welche vorwiegend mit grafischen Mitteln den Anwender bei der Entwicklung paralleler Programme unterstützen, indem besonders die Erstellung von Befehlen, die parallele Programmteile wie Aktivierung von Prozessoren oder Kommunikation zwischen Prozessoren betreffen, interaktiv entwickelt werden kann.

Das in Kapitel 3.2 vorgestellte Modell eines massivparallelen Algorithmus ist eine wichtige Grundlage, um Entscheidungen treffen zu können, wie parallele Algorithmen sinnvoll gegliedert werden können. Dabei zeigt es sich, dass parallele Programmteile wie „Aktivierung“ und „Kommunikation“ vom übrigen Code getrennt betrachtet werden und somit auch getrennt unterstützt werden können.

Voraussetzung für eine grafische Benutzungsunterstützung ist, dass die Entwicklungsumgebung grafische Darstellungen ermöglicht. Deshalb wurde ein Vorschlag gemacht, wie grafische Elemente in einen vorwiegend für Quellcode bestimmten Editor eingebunden werden können. Dazu wurde aufbauend auf bekannten Entwurfsmustern ein Konzept mittels leichtgewichtiger Klassen entworfen, womit flexibel beliebige Pixel- und Vektorgrafiken in einen Fließtext integriert werden können und welches somit universell einsetzbar ist. Zuvor wurde der aktuelle Stand der Technik untersucht, wobei viele Einsatzgebiete von Entwicklungsumgebungen und spezielle Benutzungsunterstützungen betrachtet wurden. Der in der Arbeit entwickelte Ansatz einer grafischen Unterstützung auf Befehlsebene fehlt aber bei allen bisher bekannten Entwicklungsumgebungen.

Die Entscheidung, wie einzelne Entwicklungselemente grafisch unterstützt werden, wurde durch Anlehnung an die Designphase paralleler Programme getroffen, indem die dabei häufig benutzten Skizzen so aufbereitet wurden, dass damit grafisch unterstützte Dialogboxen und daraus resultierend Icons zur Darstellung des Quelltextes entwickelt werden konnten. Es hat sich gezeigt, dass besonders die zeitintensiven Programmteile der Aktivierung und Kommunikation hervorgehoben werden sollen. Um die Unterstützung den Benutzern ergonomisch sinnvoll anbieten zu können, wurden die Entwürfe anhand von DIN- und EN-ISO-Normen sowie den Gestaltgesetzen überprüft.

Mit den entworfenen grafischen Dialogboxen und den dynamisch erstellten Icons können neben der Vermeidung syntaktischer Fehler zusätzlich semantische Fehler aufgedeckt werden. Ein Beispiel dafür zeigt in Kapitel 5.3.2.2 die Darstellung zweier verschiedener Implementierungen von Odd-Even-Sortieralgorithmen. Dabei wird in beiden Versionen die gleiche Anzahl an Quellcodezeilen benötigt. Unter Nutzung der „Icon-Darstellung“ wird aber schnell deutlich, dass bei einer Variante eine Kommunikation mehr benötigt wird, diese aber leicht vermieden werden kann. Ein Programmierer mit weniger Erfahrung würde ohne diese grafische Unterstützung wahrscheinlich die „falsche“ Variante benutzen, da bei dieser eine Befehlzeile in eine Bedingung integriert werden kann und somit seltener aufgerufen werden muss. Dadurch würde der Alternativalgorithmus auf einer Einprozessormaschine die bessere Performance liefern, auf dem Parallelrechner wird diese Reduzierung um einen Befehl aber durch eine weitere Kommunikation teuer erkauft.

Anhand der prototypischen Implementierung konnte gezeigt werden, dass alle vorgestellten grafischen Benutzungsunterstützungen realisiert werden können. Durch die Anpassung an unterschiedliche Rechnerarchitekturen wurde gezeigt, dass die vorgestellten grafischen Benutzungsunterstützungen nicht nur auf einem Rechnertyp angewandt werden können, sondern vielfältig einsetzbar sind. Dabei wurde zu Beginn erörtert, wie eine Entwicklungsumgebung mit grafischer Benutzungsunterstützung sinnvoll von Grund auf aufgebaut werden sollte. Am Ende der Arbeit wurden Vorschläge gemacht, wie wichtige Teile der grafischen Benutzungsunterstützung durch Plugins in bestehende Entwicklungsumgebungen integriert werden können, bzw. wenn keine Plugins genutzt werden können, wie trotzdem durch sogenannte „schwebende Werkzeuge“ eine grafische Benutzungsunterstützung geboten werden kann.

Möchte man die in dieser Arbeit erarbeiteten Methoden konsequent weiterentwickeln und für eine breitere Benutzergruppe zugänglich machen, sollten als nächstes die Plugin-Möglichkeiten von Eclipse genauer untersucht werden. Die Einbindung eigener grafischer Dialogboxen stellt dabei kein großes Hindernis dar, allerdings muss für eine konsequente Unterstützung eine Lösung gefunden werden, wie dynamisch aufgebaute Icons im Quelltext zur Verfügung gestellt werden können, welche innerhalb des Quelltextes bestimmte Programmteile repräsentieren können. Die Kombination von Text und Grafik innerhalb des Eclipse-Editors stellt eine Herausforderung dar, deren Realisierung noch nicht geklärt ist.

Zurzeit arbeiten wesentlich weniger Menschen an SIMD-Parallelrechnern, als an Message-Passing-, insbesondere Clusterlösungen. Deshalb sollten die vorgestellten Überlegungen, welche für Message-Passing Systeme interessant sind, genauer ausgearbeitet werden. Dadurch hat man zusätzlich den Vorteil, dass eine Vielzahl einsetzbarer Systeme existiert und die „grafische Benutzungsunterstützung auf Befehlsebene für die Entwicklung paralleler Programme“ auf breiter Basis getestet werden kann.

Der aktuelle Trend hin zu Multi-Core Systems on Chip mit einer Vielzahl von Prozessoren (aktuell 2-8, absehbar sind bis zu 100 Prozessoren) eröffnet einen weiteren Bereich, in dem eine angemessene Unterstützung des Entwicklungsprozesses durch grafische Werkzeuge und Hilfesysteme unabdingbar ist.

9 Literaturverzeichnis

[Amd05] AMD, Multi-Core Processors – The Next Evolution in Computing, White Paper, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/Multi-Core_Processors_WhitePaper.pdf (zuletzt gesehen: 10.01.2006)

[AISW96] Amato, N. M.; Iyer, R.; Sundaresan, S.; Wu, Y.: A Comparison of Parallel Sorting Algorithms on Different Architectures, 1996

[Amd67] Amdahl, G.: Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities, AFIPS Conference Proceedings, vol. 30, Atlantic City NJ, S. 483-485, 1967

[BAAD+92] Babaoglu, O.; Alvisi, L.; Amoroso, A.; Davoli, R.; Giachini, L. A.: Paralex: An Environment for Parallel Programming in Distributed Systems, In Proceedings ACM international Conference on Supercomputing 1992, http://citeseer.ifi.unizh.ch/74890.html (zuletzt gesehen: 31.01.2006)

[BAS89] Browne, J.C.; Azam, M.; Sobek, S.: CODE: A Unified Approach to Parallel Programming, IEEE Software, Volume 6. No. 4, S. 10-18, 1989

[BDGM+91] Beguelin, A.; Dongarra, J.; Geist, G.A.; Manchek, R.; Sunderam, V.S.: Graphical development tools for network-based concurrent supercomputing, Proceedings of Supercomputing 91, S. 435-444, 1991, http://citeseer.ifi.unizh.ch/beguelin91graphical.html (zuletzt gesehen: 31.01.2006)

[Beg93] Beguelin, A. L.: Xab: A Tool for Monitoring PVM Programs, in Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences, S. 102 – 103, 1993

[Bem92] Bemmerl, T.: Programmierung skalierbarer Multiprozessoren, Wissenschaftsverlag, Mannheim, 1992

[BITV02] Barrierefreie Informationstechnik-Verordnung, Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behindertengleichstellungsgesetz (BITV), 27.04.2002 http://www.einfach-fuer-alle.de/artikel/bitv/ (zuletzt gesehen: 08.09.2006)

[Boo94] Booch, G.: Objektorientierte Analyse und Design, Addison Wesley, 1994

[Bra90] Bräunl, T.: Massiv parallele Programmierung mit dem Parallaxis–Modell, Springer, 1990

[Bra93] Bräunl, T.: Parallele Programmierung, Vieweg, Braunschweig Wiesbaden, 1993

[BTS05] Bonn, M.; Toussaint, F.; Schmeck, H.: JOSCHKA Job-Scheduling in heterogenen Systemen, PARS Mitteilungen 2005, 20. PARS Workshop, Gesellschaft für Informatik, Erik Maehle, Juni 2005.

[BuL94] Butler, R. M.; Lusk, E. E.: Monitors, Messages, and Clusters: the p4 Parallel Programming System, http://www-fp.mcs.anl.gov/~lusk/p4/ index. html (zuletzt gesehen: 07.03.2006)

[CEF96] Clémeçon, C.; Endo, A.; Fritscher, J.; Müller, A.; Rühl, R.; Wylie, B.: ANNAI: An Integrated Parallel Programming Environment for Multicomputers, in: Zaky, A., Lewis, T. (Editors), Tools and Environments for Parallel and Distributed Systems, Kluwer Academic Publishers, Boston, Dordrecht, London, S. 33-59, 1996

[CKPS+93] Culler,D.; Karp,R.; Patterson, D.; Sahay, A.; Schauser, K. E.; Santos, E.; Subramonian, R.; Eicken, T. v.: LogP: towards a realistic model of parallel computation, in: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, S. 1-12, 1993

[CMK88] Carroll, J. M.; Mack, R. L.; Kellogg, W. A.: Interface metaphors and user interface design, in: Helander, M. (Editor), Handbook of Human-Computer Interaction, Elsevier Science Publishers, S. 67–85., 1988

[CoR04] Cockshott, P; Renfrew, K.: SIMD Programming Manual for Linux and Windows, Springer professional computing, London, 2004

[DIN98a] DIN-Taschenbuch 194: Bildschirmarbeitsplätze 1, Arbeitsplatz und Lichttechnik, Beuth-Verlag, Berlin, 1998

[DIN98b] DIN-Taschenbuch 242: Bildschirmarbeitsplätze 2, Arbeitsumgebung und Ergonomie, Beuth-Verlag, Berlin, 1998

[Eic96] Eichberg, D.: Untersuchung des Insel-Modells genetischer Algorithmen auf einem massiv parallelen Rechner, Diplomarbeit, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, 1996

[ESSS+96] ElGindy, H., Somani, A. K., Schröder, H., Schmeck, H., Spray, A.: RMB - a reconfigurable multiple bus network, in Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA-2), IEEE Computer Society TCCA, San Jose, California, S. 108-117, 3-7. February 1996

[Fer97] Ferrari, A. J.: JPVM: Network parallel computing in java, Technical Report, CS-97-29, Department of Computer Science, University of Virginia, 1997, http://citeseer.ifi.unizh.ch/ferrari97jpvm.html (zuletzt gesehen: 31.01.2006)

[FeW96] Feldcamp, D; Wagner, A.: Performance and Scalability in the Design and Implementation, , in: Zaky, A., Lewis, T. (Editors), Tools and Environments für Parallel and Distributed Systems, Kluwer Academic Publishers, Boston, Dordrecht, London, S. 77-102, 1996

[FFSB04] Freeman, E.; Freeman, E,; Sierra, K.; Bates, B.: Head first design patterns, O'Reilly, 2004

[FoW78] Fortune, S.; Wyllie, J.: Parallelism in random access machines, in: Proceedings of the tenth annual ACM symposium on Theory of computing, S. 114-118, 1978

[FQKY04] Fan, Z.; Qiu, F.; Kaufman, A.; Yoakum-Stover, S.: GPU Cluster for High Performance Computing, in Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004, http://portal.acm.org/citation.cfm?id=1049991&coll=ACM&dl=ACM&CFID=67355605&CFTOKEN=37994987 (zuletzt gesehen: 15.03.2006)

[GaB04] Gamma, E.; Beck, K.: Eclipse erweitern, Prinzipien, Patterns und Plug-Ins, Addison-Wesley, 2004

[Gam92] Gamma, E.: Objektorientierte Software-Entwicklung am Beispiel von ET++, Springer Verlag Berlin Heidelberg, 1992

[GBDJ+94] Geist, A.; Beguelin, A.; Dongarra, J.; Jiang, W.; Manchek, R.; Sunderam, V.: PVM: Parallel Virtual Machine- A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994, Online-Version: http://www.netlib.org/pvm3/book/pvm-book.html (zuletzt gesehen: 08.03.2006)

[GBF94] Gentzsch, W.; Block, U.; Ferstl, F.: Software Tools for Parallel Computers and Workstation Clusters, in: H. Langendörfer (Hrsg.), Praxisorientierte Parallelverarbeitung, Carl Hansen Verlag, S. 80–98, 1994

[GHJV96] Gamma, E.; Helm, R.; Johnson, R.; Vlissides J.: Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software, Addison-Wesley GmbH, 1996

[GJG93] Gorton, I.; Jelly, I.; Gray, J.: Parallel Software Engineering with PARSE, in Proceedings of Seventeenth Annual International Computer Software and Applications Conference, COMPSAC 93, S. 124 – 130, 1993

[Gre95] Green, M.: Report on Dialogue Specification Tools, in Pfaff, G. E. (Editor): User Interface Management Systems: Proceedings of the Seeheim Workshop, Springer Verlag, S. 9-20, 1985

[Gri01] Griffith, A.: GNOME/GTK+ Programming Bible, IDG Books Worldwide, Inc., Foster City, CA, 2001

[Hae92] Hänßgen, S. U.: Ein symbolischer X Windows Debugger für Modula-2*, Diplomarbeit, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, 1992, http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=demo/haenssgen92 (zuletzt gesehen: 11.01.2006)

[Han96] Hansen, O: Performance Analysis of Large Scale Parallel Applications, in: Zaky, A., Lewis, T. (Editors), Tools and Environments für Parallel and Distributed Systems, Kluwer Academic Publishers, Boston, Dordrecht, London, S. 129-148, 1996

[HaR04] Hariri, S; Ra, I.: Message-Passing Tools, in: Hariri, S.; Parashar M. (Editors), Tools and Environments for Parallel and Distributed Computing, John Wiley & Sons, Inc., Hoboken, New Jersey, S. 11-56, 2004

[HeE91] Heath, M.T.; Etheridge, J.A.: Visualizing the performance of parallel programs, IEEE Software Volume 8, No 5, Sept. 1991 S. 29 – 39, 1991

[Hei06a] Heise Newsticker, ISSCC: Uniprozessoren auf dem Rückzug, 05.02.2006, http://www.heise.de/newsticker/meldung/69228 (zuletzt gesehen: 08.09.2006)

[Hei06b] Heise Newsticker, IDF: Intel auf dem Weg zu Hunderten von Prozessorkernen, 07.03.2006, http://www.heise.de/newsticker/meldung/70439 (zuletzt gesehen: 08.09.2006)

[Hei06c] Heise Newsticker, Sparsame CPU mit 1025 Kernen, 07.07.2006, http://www.heise.de/newsticker/meldung/71647, (zuletzt gesehen: 08.09.2006)

[Hel05] Hellbusch, J. E.: Barrierefreies Webdesign, dpunkt.verlag, 2005

[Her94] Herzeg, M.: Software-Ergonomie, Grundlagen der Mensch-Computer-Kommunikation, Addison-Wesley GmbH, 1994

[Her01] Herold, H.: Das Qt Buch – Portable GUI-Programmierung unter Linux/UNIX/Windows, SuSE PRESS, Nürnberg, 2001

[Her03] Herold, H.: Lex & yacc : die Profitools zur lexikalischen und syntaktischen Textanalyse, Addison-Wesley, 2003

[Hir97] Hirsch, E.: Untersuchung von Programmierschnittstellen für grafische Benutzeroberflächen unter UNIX, Studienarbeit, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, 1997

[Hir98] Hirsch, E.: Entwurf und Realisierung eines Editors zur grafisch unterstützen Programmierung, Diplomarbeit, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, 1998

[Hol98] Hollingsworth, J. K.: Critical Path Profiling of Message Passing and Shared-Memory Programs, in IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 10, 1998, http://www.cs.umd.edu/~hollings/papers/tpds98.pdf (zuletzt gesehen: 17.01.2006)

[Hor04] Horstmann, C. S.: Object oriented design and patterns, Wiley, 2004

[IBM06] IBM, 2006, Abstract User Interface Markup Language (AUIML) Toolkit, http://www.alphaworks.ibm.com/tech/auiml (zuletzt gesehen 21.03.2006)

[Int06] Intel: Next Leap in Microprocessor Architecture: Intel® Core™ Duo Processor, White Paper, http://www.intel.com/pressroom/kits/centrino/core%20duo%20white%20paper.pdf (zuletzt gesehen: 10.01.2006)

[ISA98] ISATEC Soft-und Hardware GmbH: The ISATEC Parallel Computer Systola 1024, Users Guide V. 3.0, 1998

[KAGB82] Kung, S.-Y.; Arun, K. S.; Gal-Ezer, R. J.; Bhaskar Rao, D. V.; Wavefront Array Processor: Language, Architecture, and Applications, IEEE Transactions on Computers, VOL. C-31, No. 11, 1982

[Kho95] Khoral Inc: Khoros visual programming manual, Khoros 2.0 Manual Set, 1995

[KLSS+88] Kunde, M., Lang, H.-W., Schimmler, M., Schmeck, H., Schröder, H.: The instruction systolic array and its relation to other models of parallel computers, in Parallel Computing 7, S.25-39, 1988.

[Kob85] Kobsa, A.: Benutzermodellierung in Dialogsystemen, Springer-Verlag, Berlin, 1985

[Kob89] Kobsa, A.; Wahlster, W.: User Models in Dialog Systems, Springer-Verlag, Berlin, 1989

[Koh99] Kohlmorgen, U.: Feinkörnige parallele genetische Algorithmen, Dissertation Universität Karlsruhe, 1999

[KuL79] Kung, H. T.; Leiserson, C.E.: Systolic Arrays (for VLSI), Sparse Matrix Proceedings ,78, Academic Press, Orlando FL, S.256-282, 1979

[Lan94] Langmann R.: Graphische Benutzerschnittstellen, Einführung und Praxis der Mensch-Prozeß-Kommunikation, VDI-Verlag, Düsseldorf, 1994

[LCV88] Linton, M. A.; Calder, P. R.; Vlissides, J. M.: InterViews: A C++ Graphical Interface Toolkit, Technical Report CSL-TR-88-358, Stanford University, 1988. http://citeseer.ifi.unizh.ch/255760.html (zuletzt gesehen: 08.03.2006)

[LiB06] Li, M; Baker, M.: A Review of Grid Portal Technology in Cunha, J. C.; Rana, O. F. (Editors), Grid Computing – Software Environments and Tools, Springer-Verlag London, 2006

[LiG98] Liu, A.; Gorton, I.: PARSE-DAT: An Integrated Environment for the Design and Analysis of Dynamic Software Architectures, in Proceedings if International Symposium on Software Engineering for Parallel and Distributed Systems, S. 146 – 154, 1998

[LGPC+96] Labarta, J.; Girona, S.; Pillet, V.; Cortes, T.; Gregoris, L.: DiP: A Parallel Program Development Environment, in Proceedings of the 2nd International Euro-Par Conference, S. II:665--674., 1996, http://citeseer.ifi.unizh.ch/labarta96dip.html (zuletzt gesehen: 31.01.2006)

[LHKP+04] Luebke, D.; Harris, M.; Krüger, J.; Purcell, T.; Govindaraju, N.; Buck, I.; Woolley, C.; Lefohn, A.: GPGPU: General Purpose Computation On Graphics Hardware, Proceedings of the conference on SIGGRAPH 2004 course notes GRAPH '04, 2004, http://portal.acm.org/citation.cfm?id=1103933&coll=ACM&dl=ACM&CFID=67355605&CFTOKEN=37994987 (zuletzt gesehen: 15.03.2006)

[Lin04] Lindner, D.: Skalierungsprobleme einer J2EE Applikation auf Mehrprozessorsystemen, Diplomarbeit, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, 2004

[LSWR+03] Lil, M.; van Santen, P.; Walker, D. W.; Rana, O. F.; Baker, M. A.: PortalLab: A Web Services Toolkit for Building Semantic Grid Portals, in Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003.

[Mac91] MacKenzie, I. S.: Fitts' law as a performance model in human-computer interaction, doctoral dissertation, University of Toronto, Canada, http://www.yorku.ca/mack/phd.html (zuletzt gesehen: 17.05.2006)

[Mas92a] MasPar Computer Corporation, MasPar Programming Environment (MPPE) User Guide, 749 North Mary Avenue, Sunnyvale, California 94086, 1992

[Mas92b] MasPar Computer Corporation, MasPar Programming Language, Reference Manual, 749 North Mary Avenue, Sunnyvale, California 94086, 1992

[MaZ94] Marwaha, G.; Zhang, K.: Parallel program visualisation for a message-passing system, in Proceedings of the 13th Annual IEEE International Conference on Computers and Communications, Phoenix, USA, S. 200 – 205, IEEE Press, 1994, http://citeseer.ifi.unizh.ch/marwaha94parallel.html (zuletzt gesehen: 31.01.2006)

[McD95] McDonald, K.: The NAG Numerical PVM Library, in: Dongarra, J.; Madsen, K.; Waśniewski, J. (Editors), LNCS 1041, Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, Springer-Verlag, Berlin Heidelberg, 1996

[Mic06a] Microsoft, Visual Studio 2005, Die neuen Features im Überblick, 2006, http://www.microsoft.com/germany/msdn/vstudio/products/features.mspx (zuletzt gesehen: 06.07.2006)

[Mic06b] Microsoft, Visual Studio 2005 Express Editions, 2006, http://www.microsoft.com/germany/msdn/vstudio/products/express/default.mspx (zuletzt gesehen: 21.03.2006)

[Mic06c] Microsoft, Windows Media Player 9 Series Designs, 2006, http://www.microsoft.com/windows/windowsmedia/9series/gettingstarted/personalization/Skins.asp?page=4&lookup=Skins (zuletzt gesehen: 17.07.2006)

[Moo65] Moore, G.: Cramming more components onto integrated circuits, Electronics, Volume 38, Number 8, April 19, 1965, ftp://download.intel.com/research/silicon/moorespaper.pdf (zuletzt gesehen: 10.01.2006)

[Moo06] Moolenaar, B.: Vim - Vi IMproved, http://www.moolenaar.net/vim.html (zuletzt gesehen: 08.02.2006)

[MPIF94] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, 1994, http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-1.0.ps (zuletzt gesehen: 08.03.2006)

[Msdn06] MSDN, Microsoft Development Network: Windows API Reference, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_reference.asp (zuletzt gesehen: 29.09.2006)

[MSM04] Mattson, T. G.; Sanders B. A.; Massingill, B. L.: Patterns for Parallel Programming, Addison-Wesley, 2004

[Mye95] Myers, B. A.: User Interface Software Tools, ACM Transactions on Computer-Human Interaction. Vol. 2, No. 1, S. 64-103, 1995, http://reports-archive.adm.cs.cmu.edu/anon/1994/CMU-CS-94-182.ps (zuletzt gesehen, 30.05.2006)

[NeB92] Newton, P.; Browne, J. C.: The code 2.0 graphical parallel programming language, in Proceedings of ACM International Conference on Supercomputing, 1992

[Nie93] Nielsen, J.: Usability Engineering, AP Professional, Cambridge, 1993

[NiS92] Nigam, M.; Sahni, S.: Sorting n Numbers On n × n Reconfigurable Meshes With Buses, 1992, http://citeseer.ist.psu.edu/23752.html (zuletzt gesehen: 10.04.2006)

[NKMD96] Ng, K.; Kramer, J.; Magee, J.; Dulay, N.: A Visual Approach to Distributed Programming, in: Zaky, A., Lewis, T. (Editors), Tools and Environments für Parallel and Distributed Systems, Kluwer Academic Publishers, Boston, Dordrecht, London, S. 7-31, 1996

[PaC04] Parashar, M; Chandra, S.: Distributed Shared Memory Tools, in: Hariri, S.; Parashar M. (Editors), Tools and Environments for Parallel and Distributed Computing, John Wiley & Sons, Inc., Hoboken, New Jersey, S. 57-78, 2004

[Pet03] Peterchen: 10 Minutes to document your code, http://www.codeproject.com/tips/doxysetup.asp (zuletzt gesehen, 15.03.2006)

[Pfo93] Pförtner, M.: Werkzeuge zur Entwicklung graphischer Benutzeroberflächen im Vergleich, Diplomarbeit, Institut für Betriebssysteme und Rechnerverbund, Technische Universität Braunschweig, 1993

[Pre99] Preim, B.: Entwicklung interaktiver Systeme – Grundlagen, Fallbeispiele und innovative Anwendungsfelder, Springer-Verlag, 1999

[Ree79] Reenskaug, T.: THING-MODEL-VIEW-EDITOR - an Example from a planning system, http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf (zuletzt gesehen: 10.01.2006)

[Rit05] Rittner, G.: The Traditional Vi, Source Code for the Modern Unix Systems, http://ex-vi.sourceforge.net/ (zuletzt gesehen: 08.02.2006)

[RKN04] Raje, R.; Kalaynaraman, A.; Nayani, N.: Distributed-Object Computing Tools, in: Hariri, S.; Parashar M. (Editors), Tools and Environments for Parallel and Distributed Computing, John Wiley & Sons, Inc., Hoboken, New Jersey, S. 79-148, 2004

[Roh90] Rohr, G.: Ikon-Techniken für Benutzerschnittstellen komplexer Anwendungssoftware, in: Reuter, A. (Hrsg.), GI-20. Jahrestagung I, Springer-Verlag, Stuttgart, S. 284-296, 1990

[RoS93] Robič, B.; Šilc, J.: High-performance computing on a honeycomb architecture, Technical Report, 1993

[Sch86] Schmeck, H.: A comparison based instruction systolic array, in: M. Cosnard et al. (eds.): Parallel Algorithms and Architectures, North-Holland S. 281-292, 1986

[Sch99] Schmidt, B.: Algorithm Design on the Instruction Systolic Array, dissertation, Parallel Algorithms and Architectures Research Centre, Department of Computer Science, Loughborough University, Loughborough, Leics., U.K., 1999

[ShH05] Shea, D.; Holzschlag, M. E.: Zen und die Kunst des CSS-Designs, Addison-Wesley, München, 2005

[Shn98] Shneiderman, B.: Designing the User Interface, Strategies for Effective Human-Computer Interaction, Addison Wesley Longman, Third Edition, 1998

[SJ03] Shalloway, A.; Trott, J. R.: Entwurfsmuster verstehen - eine neue Perspektive auf objektorientierte Software-Entwicklung, MITP-Verlag, 2003

[SNI93] Siemens Nixdorf: Grafik-Styleguide, Richtlinien zur Gestaltung grafischer Benutzeroberflächen, Siemens Nixdorf Informationssysteme AG, 1993

[Spa05] Spanier, M.: Geschäftsprozessoptimierung durch Echtzeitkommunikation, Diplomarbeit, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, 2005

[SSK93] Scheidler, C.; Schäfers, L.; Krämer-Fuhrmann, O.; TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications, in Proc. PARLE'93: Parallel Architectures and Languages Europe, München, 1993, http://citeseer.ifi.unizh.ch/scheidler93trapper.html (zuletzt gesehen: 29.03.2006)

[SSK94] Scheidler, C.; Schäfers, L.; Krämer–Fuhrmann, O.: TRAPPER: Eine graphische Software–Entwicklungsumgebung für MIMD–Parallelrechner, in: M. Baumann, R. Grebe (Hrsg.): Parallele Datenverarbeitung mit dem Transputer, Springer–Verlag, S. 112–121, 1994

[SSK95] Scheidler, C.; Schäfers, L.; Krämer-Fuhrmann, O.; Software Engineering for Parallel Systems: The TRAPPER Approach, in Proc. of the 28th Hawaiian International Conference on System Sciences, IEEE CS Press 1995, http://citeseer.ifi.unizh.ch/scheidler95software.html (zuletzt gesehen: 16.01.2006)

[SST02] Sakayori, Y.; Shizuki, B.; Tanaka, J.: Programming Environment Specified for Describing Interprocessor Communications based on the Operations on Graphical User Interface, 2002, http://citeseer.ifi.unizh.ch/570523.html, (zuletzt gesehen: 18.01.2006)

[Sta81] Stallman, R.M.: EMACS the extensible, customizable self-documenting display editor, in Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation Portland, Oregon, United States, S. 147 – 156, ACM Press New York, NY, USA, 1981

[Sto97] Stojmenovic, I.: Honeycomb networks: Topological properties and communication algorithms, in Proceedings of IEEE Transactions on Parallel and Distributed Systems, 1997

[Str89] Stroustrup, B.: Die C++ Programmiersprache, Addison-Wesley, 1989

[StZ02] Stankoviz, N.; Zhang, K.: A Distributed Parallel Programming Framework, IEEE Transactions on Software Engineering, Volume 28, Issue 5, S. 478 – 493, 2002

[Tem95] Tempel, U.: Vergleich lokaler Selektionsstrategien für feinkörnig parallele genetische Algorithmen zur Lösung von schweren Optimierungsproblemen, Diplomarbeit, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, 1995

[ThB05] Thomas, A.; Becker, J.: Online-adaptive Reconfigurable Hardware Architecture and Runtime Environment, in Proceedings of IEEE International SOC Conference, 2005

[Tou96] Toussaint, F.: PEdit, eine Programmierumgebung mit graphischer Unterstützung zur Entwicklung paralleler SIMD–Programme, PARS ’96, Gesellschaft für Informatik, S. 22-30, 1996

[Tvr99] Tvrdik, P.: Introduction to parallel sorting on mesh-based topologies, 1999, http://www.cs.wisc.edu/~tvrdik/15/html/Section15.html (zuletzt gesehen: 18.07.2006)

[Val90] Valiant, L. G.: A bridging model for parallel computation, Communications of the ACM archive, Volume 33 , Issue 8, S. 103 – 111, 1990

[Vei95] Veith, H.: Ein massiv paralleler Genetischer Algorithmus zur Lösung von Losgrößenproblemen, Diplomarbeit, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, 1995

[VoN98] Voss, J.; Nentwig, D.: Entwicklung von graphischen Benutzungsschnittstellen, Modelle, Techniken und Werkzeuge der User-Interface-Gestaltung, Carl Hanser Verlag München Wien, 1998, www.hanser.de

[Wal95] Waldschmidt, K. (Hrsg.): Parallelrechner, Architekturen – Systeme – Werkzeuge, B. G. Teubner Stuttgart, 1995

[Wer64] Wertheimer, M.: Drei Abhandlungen zur Gestalttheorie, Reprogr. d. Ausg. Erlangen 1925, Wiss. Buchges., Darmstadt, 1963

[Wie97] Wiethoff, A.: Verifizierte globale Optimierung auf Parallelrechnern, Dissertation Universität Karlsruhe, 1997

[YM88] Yang, C. Q.; Miller, B. P.: Critical path analysis for the execution of parallel and distributed programs, Proceedings of the 8th International Conference on Distributed Computing Systems, 1988, S. 366–375

Benutzungsunterstützung wird heute von jedem Programm verlangt. Aufbauend auf allgemeinen ergonomischen Grundlagen und einem allgemeinen Modell für parallele Algorithmen stellt die vorliegende Arbeit Methoden vor, wie Quelltext paralleler Programme durch den Einsatz dynamisch erzeugter Icons übersichtlicher gestaltet werden kann. Dazu wird eine neu entworfene Entwicklungsumgebung vorgestellt, die neben der Icondarstellung im Quelltext auch Dialogboxen bereitstellt, welche besonders für Aktivierungs- und Kommunikationsaufgaben auf Parallelrechnern angepasst sind. Im Weiteren beschäftigt sich die Arbeit mit einer Ideenskizze für „schwebende Werkzeuge“, durch die vorhandene Programme um diese grafischen Benutzungsunterstützungen erweitert werden könnten, ohne dass dazu die Ursprungsprogramme geändert werden müssen.

ISBN: 978-3-86644-130-9

www.uvka.de