This paper appeard in: David J. Lilja and Peter L. Bird, editors, The Interaction of Compilation Technology and Computer Architecture. Kluwer Academic Publishers, Boston, Dordrecht, London, 1994.

10

PROJECT TRITON: TOWARDS IMPROVED PROGRAMMABILITY OF PARALLEL COMPUTERS

Michael Philippsen, Thomas M. Warschko, Walter F. Tichy, Christian G. Herter, Ernst A. Heinz, and Paul Lukowicz

University of Karlsruhe
Department of Informatics
Germany

ABSTRACT

The main objective of Project Triton is adequate programmability of massively parallel computers. This goal can be achieved by tightly coupling the design of programming languages and parallel hardware.

The approach taken in the Project Triton is to let high-level, machine independent parallel programming languages drive the design of parallel hardware. This approach permits machine-independent parallel programs to be compiled into efficient machine code. The main results are as follows:

Modula-2*.
This language extends Modula-2 with constructs for expressing a wide range of parallel algorithms in a portable, problem-oriented, and readable way.

Compilation Techniques.
We present techniques for the efficient translation of Modula-2* and similar imperative languages for several modern parallel machines and derive recommendations for future parallel architectures.

Triton/1 Parallel Architecture.
Triton/1 is a scalable mixed-mode SIMD/MIMD parallel computer with a highly efficient communications network. It overcomes several deficiencies of current parallel hardware and adequately supports high-level parallel languages.

1 INTRODUCTION

Project Triton focuses on the goal of adequate programmability of parallel machines. Adequate programmability means that programs can be formulated in a problem-oriented rather than a machine-oriented fashion and can be compiled to run efficiently on a wide range of parallel machines.

The development of today's commercially available parallel computers has been mainly driven by hardware considerations. Although users of the first massively parallel machines were willing to expend substantial programming effort to reach acceptable performance, the increasing availability of parallel machines calls for drastically lower program development costs. Given the rapid evolution of parallel architectures, few users can afford the luxury of non-portability of their programs. Besides portability, better programming languages are needed for problem-oriented formulation of programs with multiple levels of parallelism and multiple grain sizes of parallel operations.

One can observe that parallel computing is presently repeating some of the development steps that sequential computing underwent in the past four decades. Improvements in hardware architecture, operating systems, programming languages, and compiler technology should eventually render the current practice of machine-dependent parallel programming as obsolete as machine-dependent sequential programming.

As in sequential computing, the goal of adequate programmability of parallel machines can only be achieved by tightly coupling the design of programming languages, compilers, and parallel hardware. Triton couples the following subprojects:

Explicitly Parallel Language. We show simple language constructs that (a) avoid shortcomings of known parallel programming languages, (b) express parallel programs in a high-level, problem-oriented way, and (c) can be translated into efficient code for various parallel architectures. We use Modula-2* [31] as an example language (see section 2); similar extensions could easily be integrated into other imperative programming languages.

Optimizing Compilers. We found effective optimization techniques that improve runtime on parallel machines dramatically [11, 21, 22, 23]. In general, however, compilation and optimization are severely hampered by current parallel hardware. Our experience with writing compilers for parallel machines has led us to formulate several recommendations for future parallel

architectures. Optimization techniques, hardware recommendations, and performance results are given in section 3.

Parallel Machine Architectures. We explore novel architectural paradigms of parallel computers by building a massively parallel computer called Triton/1 [12] that is presented in section 4. Triton/1 is a mixed-mode SIMD/MIMD computer (called SAMD, for "synchronous or asynchronous instruction, multiple data") with a highly efficient communications network. Triton/1 overcomes various deficiencies of current parallel machines, supports high-level parallel languages such as Modula-2*, and implements many of our recommendations.

2 MODULA-2*

The majority of programming languages for parallel machines, including *LISP, C*, MPL,Fortran 90, Fortran D, HPF, Blaze, Dino, and Kali [7, 14, 17, 18, 25, 29, 30], suffer from some or all of the following problems:

- *Manual Virtualization.* The programmer must write explicit code for mapping processes, whose number is problem dependent, onto the available processors, whose number is fixed. This task is not only tedious and repetitive but also one that makes programs non-portable.
- *Manual Data Allocation.* Distribution of data over memory modules must be programmed explicitly for achieving adequate performance. Because of the tight coupling of data allocation with algorithms and the topology of interconnects, the programs are difficult to comprehend and not portable.
- *Manual Communication.* Inter-process communication must be implemented by means of low-level message passing primitives. This results in difficult code, especially if asynchrony and nondeterministic behavior is possible.
- *MIMD/SIMD Exclusiveness.* Parallel programming languages are either synchronous or asynchronous, reflecting whether the target machine is a SIMD or MIMD architecture. On SIMD machines, programs are restricted to total synchrony even if that causes poor machine utilization. On MIMD machines, tightly synchronous execution is expensive. Since the choice is dictated by the available hardware rather than the actual problem, the resulting programs are often distorted and not portable.

Modula-2* preserves the main advantages of data parallelism while avoiding the above drawbacks. The new language constructs allow for clear and portable parallel programs without intolerable loss of efficiency. Because of the compactness and simplicity of the extensions, they could easily be incorporated into other imperative programming languages, such as Fortran, C, or Ada. The following list describes the central advantages of our language approach:

- The programming model of Modula-2* is a superset of data parallelism.
- The language provides a single address space. Note that shared memory is not required; a single address space merely permits all memory to be addressed uniformly, but not necessarily at uniform speed. On machines without a shared address space, the compiler must insert communication instructions for all non-local accesses.
- Synchronous and asynchronous parallel computations as well as arbitrary nestings thereof can be formulated in a totally machine-independent way.
- Procedures may be called in any context (sequential and parallel) and at any nesting depth. Furthermore, additional parallel processes can be created inside procedures (recursive parallelism).

Overview of Language Extensions

Modula-2*extends Modula-2 with the following two language constructs.

1. The `FORALL` statement, which has a synchronous and an asynchronous version, is the only way to introduce parallelism into a Modula-2* program.
2. The layout of array data may optionally be specified per dimension by so-called *allocators*, e.g., `CYCLE`, `SPREAD`, and `LOCAL`. They do not have any semantic meaning but are merely data layout hints for the compiler.

The Modula-2* syntax of the `FORALL` statement is listed below:

```
FORALL  ident ":" SimpleType IN (PARALLEL|SYNC)
  [ VarDecl+
BEGIN]
   StatementSequence
END .
```

SimpleType is an enumeration or a possibly non-static subrange, i.e. the boundary expressions may contain variables. The `FORALL` creates as many (conceptual) processes as there are elements in *SimpleType*. The identifier introduced by the `FORALL` statement is local to it and serves as a runtime constant for every process created by the `FORALL`. The runtime constant of each process is initialized to a unique value of *SimpleType*. The `FORALL` statement provides an optional section for the declaration of variables local to each process. These local variables lead to better source code structuring, thus greatly increasing the readability and efficiency of parallel code.

Each process created by a `FORALL` executes the statements in *StatementSequence*. The `END` of a `FORALL` statement imposes a *synchronization barrier* on the participating processes; termination of the whole `FORALL` is delayed until all created processes have finished their execution of *StatementSequence*.

The version of the `FORALL` statement (asynchronous or synchronous) determines whether the created processes execute *StatementSequence* concurrently or in lock-step. Hence, for non-overlapping vectors `X`, `Y`, and `Z` the simple asynchronous `FORALL` statement on the lower left suffices to implement the vector addition `X := Y + Z`. In contrast, parallel modifications of overlapping data structures require synchronization provided by synchronous `FORALL`s. Thus, irregular data permutations can be implemented easily, as shown on the lower right. The effect of the second `FORALL` statement is to permute the vector `X` according to the permutation function `p`. Here, the synchronous semantics ensure that all right-hand side elements `X[p(i)]` are read and temporarily stored before any variable `X[i]` is written. This behavior stems from the *implicit* synchronization barrier introduced between the left and right hand side of any assignment in a synchronous context.

```
FORALL i:[1..N] IN PARALLEL
  Z[i] := X[i] + Y[i]
END
```

```
FORALL i:[1..N] IN SYNC
  X[i] := X[p(i)]
END
```

In synchronous branching statements such as `IF C THEN SS1 ELSE SS2 END`, the set of participating processes divides into *disjoint* and *independently operating* subsets, each of which executes one of the branches (`SS1` and `SS2` in the example) in unison. In contrast to other data-parallel languages, *no* assumption about the relative speeds or order of the branches may be made. The execution of the entire statement terminates when all processes of all subsets have completed their branches.

The semantics of other synchronous control statements are defined in a similar fashion. See the language definition [31] for more details.

The synchronous version of our `FORALL` operates much like the (more recent) HPF `FORALL`, except that ours is fully orthogonal to the rest of the language. Any statement, including conditionals, loops, other `FORALL`s, and subroutine calls, may be placed in its body. Thus, the language explicitly supports nested and recursive parallelism. An asynchronous `FORALL` is absent from HPF.

3 OPTIMIZATION TECHNIQUES AND HARDWARE RECOMMENDATIONS

The proposed language features can be translated to existing parallel machines. We have shown this for SIMD, MIMD, and SISD machines. For SIMD machines, we have implemented compilers targeting the Connection Machine CM-2 and the MasPar MP-1. For MIMD machines, we are currently targeting a network of workstations and the KSR-1. Additionally, Modula-2* programs can be translated for sequential UNIX workstations (SISD), which is appropriate for program development and teaching.

In this section we present basic translation and optimization techniques. We derive recommendations for hardware improvement by showing that the compiler is hindered by certain hardware characteristics. Finally, section 3.3 provides performance results for the Modula-2* compiler and for the optimization techniques presented.

3.1 Implementing `FORALL` Statements

Obviously, the most challenging task in translating Modula-2* for parallel machines is to generate efficient code for `FORALL` statements.

Basic Translation Scheme

A straightforward MIMD approach is to implement the processes created by a `FORALL` with threads, distributed over the p available processors. In our implementation, however, not more threads are spawned than processors are available. Each of these threads simulates a certain number of processes in

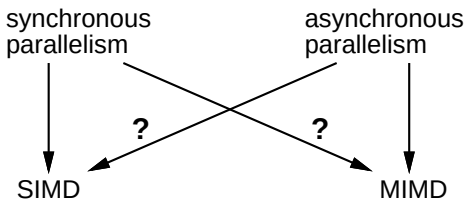
virtualization loops. This choice is necessary because thread switching is much more expensive than loop control.

Synchronization and termination at the end of a `FORALL` require a barrier. If all n threads of a `FORALL` terminate at about the same time, then the termination on a MIMD machine without synchronization hardware requires time proportional to $O(\log n)$. In our implementation, synchronization is done on the processor level instead of the process level, thus reducing the overhead to $O(\log p)$.

The synchronous nature of a SIMD machine, coupled with the broadcast bus from the (frontend) control processor, makes process creation, termination, and load balancing operate in constant time. However, if the number t of necessary processes for nested `FORALL`s cannot be evaluated statically during compilation, a $O(\log n)$ summation must be used. Once t is known, t stacks are created by assigning to each of processors a segment of $\lceil t/p \rceil$ stacks. This takes constant time and balances the load perfectly. Process termination also takes constant time, since there is no synchronization overhead. The problem here is to achieve a good load balance in case of deactivation of processes by cascaded `IF`-statements inside of `FORALL`s and in case of nested parallelism.

Because of the frequency of `FORALL` statements in parallel programs we need hardware support for fast process creation, termination, and context switching.

Obviously, the problem of translating `FORALL` statements into efficient code is more complex:



The problem of implementing a synchronous `FORALL` on a MIMD machine, or an asynchronous `FORALL` on a SIMD machine demands novel optimization techniques and requires hardware improvements. Both are presented in the next two sections.

Elimination of Synchronization Barriers

We have developed transformation schemes that map synchronous into (equivalent) asynchronous `FORALL` statements with temporary variables. These can be implemented on MIMD machines directly. Our transformations are more general than Hatcher's work [9] because they also account for nested parallelism and do not rely on having explicit communication instructions in the source program. Consider for example the following synchronous `FORALL` statement and its inefficient transformation into asynchronous `FORALL`s shown below.

```
FORALL i:  [1..N] IN SYNC
  Z[i] := Z[i+1];
  X[i] := X[2*i];
  Y[i] := Y[p(i)]
END
```

⇓

```
FORALL i:[1..N] IN PARALLEL   H1[i] :=  Z[i+1]   END;
FORALL i:[1..N] IN PARALLEL    Z[i] := H1[i]     END;
FORALL i:[1..N] IN PARALLEL   H2[i] :=  X[2*i]   END;
FORALL i:[1..N] IN PARALLEL    X[i] := H2[i]     END;
FORALL i:[1..N] IN PARALLEL   H3[i] :=  Y[p(i)]  END;
FORALL i:[1..N] IN PARALLEL    Y[i] := H3[i]     END
```

Although this transformation yields code that can be run on both SIMD and MIMD machines, it is not efficient because of the short virtualization loops that are a problem on SIMD machines [4], and the large number of synchronization messages induced on current MIMD hardware.

Our optimization is based on the insight that most of the synchronization barriers introduced by synchronous language constructs need not be implemented in real synchronous `FORALL`s to ensure the prescribed semantics [11]. To detect redundant synchronization barriers we apply data dependence analysis originally developed for parallelizing Fortran compilers.

In the above example there are only three data dependences, one per assignment. After restructuring the program, a single synchronization barrier which cuts all three dependences at once suffices to ensure correctness.

```
FORALL i:[1..N] IN PARALLEL
  H1[i] := Z[i+1];
```

```
  H2[i] := X[2*i];
  H3[i] := Y[p(i)]
END;
FORALL i:[1..N] IN PARALLEL
  Z[i] := H1[i];
  X[i] := H2[i]
  Y[i] := H3[i]
END;
```

The number of resulting asynchronous `FORALL`s and, correspondingly, the number of time consuming synchronization messages on MIMD machines is reduced. Furthermore, virtualization loops grow with the number of synchronization barriers that can be eliminated. Larger virtualization loops allow for traditional sequential optimizations, such as common subexpression elimination and strength reduction, which are an important source of performance improvement on SIMD machines.

We are presently exploring optimizations that reduce the amount of temporary storage. In the above example, the temporary array `H1` could be changed into a per-processor register by taking advantage of the direction of the virtualization loop. However, such optimizations usually require additional synchronization barriers, trading space for time.

Synchronization barrier elimination as described here has been implemented in our compiler suite targeting SIMD, MIMD, and SISD machines. Quantitative results are given in section 3.3.

Although this optimization technique is quite efficient, some synchronization barriers will always remain necessary to be implemented. On current parallel machines even these synchronization barriers can cause a significant loss of performance. The reason is that communication networks today are optimized for transporting data at high rates with use of latency hiding techniques. However, for synchronization only a few bits, i.e., small packets, have to be transmitted and latency hiding is impossible. Hence, we recommend

hardware support for fast barrier synchronization (on MIMD machines).

Branch Combining

Branch combining is a method that attempts to improve the efficiency of asynchronous `FORALL`s on SIMD machines. The goal is to avoid the idling of a large

fraction of the processors. Although the semantics of Modula-2* prescribe that branching statements create independent groups of threads, it is permissible to combine these groups when they execute the same code. Identical segments in different branches can be found statically by determining the *Longest Common Subsequence* [26] of these branches. Hanxleden [33] studies branch combining for the special case of parallel loops on SIMD machines.

3.2 Automatic Data and Process Distribution

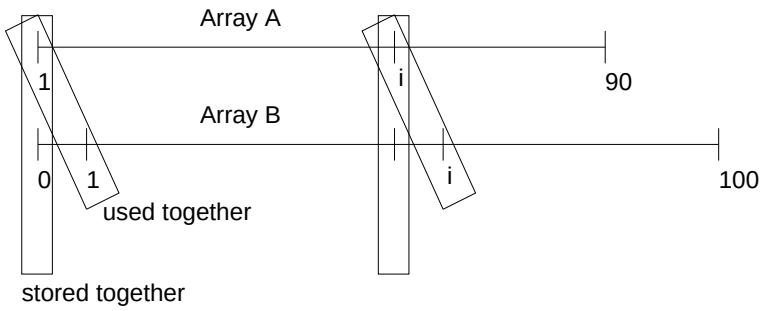
Because of slow and high latency communication networks on distributed memory machines, the distribution, i.e., alignment and layout, of data and processes over the available processors is a central problem.

Alignment is the task of finding an appropriate trade-off between the two conflicting goals of (1) data locality and (2) maximum degree of parallelism. Our automatic alignment algorithm is described in [22] and briefly sketched below by means of an example. *Layout* is the assignment of aligned data structures and processes to the available processors. Desirable goals are (3) the exploitation of special hardware supported communication patterns and (4) simple address calculations. We use an automatic mapping [20] of arbitrary multidimensional arrays to processors. Thus we exploit grid communication if available and achieve efficient address calculations.

To understand the techniques and advantages of automatic data and process distribution consider the following example.

```
VAR A: ARRAY [1..90]  SPREAD OF INTEGER;
    B: ARRAY [0..100] SPREAD OF INTEGER;
FORALL i:[1..90] IN PARALLEL
  A[i]   := B[i];
  B[i-1] := 0
END
```

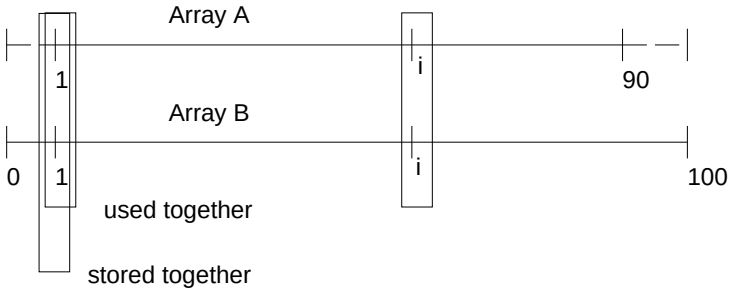
Without any distribution optimization `A[i]` and `B[i-1]` would reside in the same processor's local memory. In the `FORALL` statement, however, `A[i]` and `B[i]` are used together although they are located in different processors, at least at virtualization boundaries.



Enlarging `A`'s bounds will result in the same storage pattern for both `A` and `B`. Enlarging `A`'s upper bound will ensure this effect even if virtualization due to segmentation over a small number of processors is necessary. The enlargement decouples alignment and layout. Since the resulting arrays have the same size, the layout algorithm maps corresponding elements of the array to the same processor. We allow for moderate storage waste because the primary goal is execution speed. This transformation yields the following code fragment:

```
VAR A,B: ARRAY [0..100] SPREAD OF INTEGER;
FORALL i:[1..90] IN PARALLEL
  A[i]   := B[i];
  B[i-1] := 0
END
```

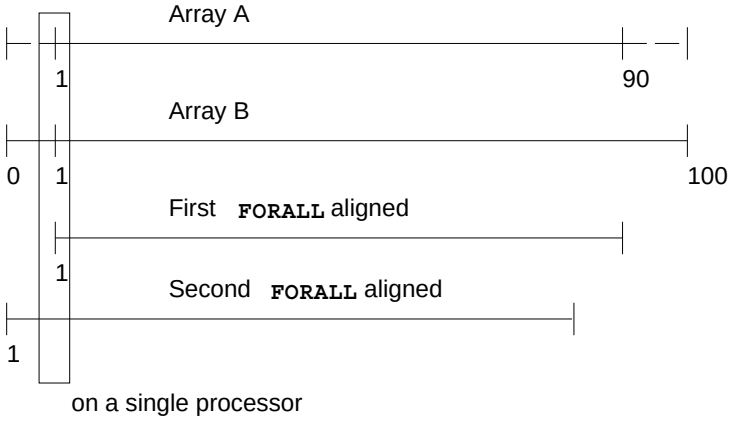
The corresponding storage pattern is illustrated below:



Up to now we have only dealt with the alignment of data. Process alignment is also achieved by means of source-to-source transformations. During the transformation, `FORALL`s are attributed with an `ALIGNED WITH` clause that directs the code generator to allocate each process exactly where the corresponding data element resides:

```
VAR A,B: ARRAY [0..100] SPREAD OF INTEGER;
FORALL i:[1..90] IN PARALLEL ALIGNED WITH A[i]
  A[i]   := B[i]
END;
FORALL i:[1..90] IN PARALLEL ALIGNED WITH B[i-1]
  B[i-1] := 0
END
```

The resulting patterns of date storage and process scheduling is shown below:



The original `FORALL` has been split into two. In the first `FORALL`, the process with index `i` will be executed where data element `A[i]` resides. In the second `FORALL`, the process with index `i` will be scheduled according to the distribution of `B[i-1]`. This results in local accesses that could not be achieved with a single `FORALL`. Cost estimation is necessary to trade-off the cost of splitting up of `FORALL`s and the cost of access to non-local data. See [22] for more details.

Automatic data and process distribution as described here have been implemented in our compiler suite; quantitative results are given in section 3.3.

Although the compiler can often find alignments and layouts that reduce the amount of non-local communication significantly, there will still be communication in the general case. Therefore, the basic performance of the network must be improved and we conclude that

MCPS measure must approach MIPS measure.

Present communication networks are far too slow compared to the speed of the processors. We measure the speed of the communication network in Million Connections Per Second (MCPS), i.e., the number of messages delivered per second. This measure is more accurate than bandwidth numbers because it does not hide latency and routing overhead. Unfortunately, typical MIMD machines can execute 100 to 1,000 arithmetic instructions in the time it takes to deliver a single small message. This disparity forces a distortion of parallel algorithms: reducing the number of data packets by reorganizing the algorithm and by combining packets becomes all-important.

With some SIMD machines, the ratio of arithmetic operation time to packet delivery time is somewhat better: approximately 1:10 for neighbor communication and 1:40 for random communication [24]. But even that is not sufficient: the programmer is still forced to find good mappings of the data structures onto the topology of the network to exploit the faster neighbor communication.

A second performance-oriented recommendation is

support for latency hiding

since it allows the delivery of packets concurrently with computation. That would enable the compiler to interleave computation and communication and, thus, to hide some of the communication latency. Support for latency hiding could be implemented by an independently operating network with asynchronous message delivery or a decoupled access processor prefetching data.

Furthermore, we recommend a

shared address space.

System wide addresses are especially important for the implementation of pointers because otherwise they would have to be simulated quite inefficiently in software. Therefore, all processors should be able to generate addresses for the entire memory of the system. Even the memory of the control processor, e.g., the frontend of a SIMD machine, should be part of that address space. As noted earlier, a shared address space does not imply shared memory.

The problems due to different types of addresses can be studied with C*. The old version has about ten different variants for each pointer type. The variants reflect whether the pointer itself is stored in a singular or a parallel variable and whether it actually points to a singular or a parallel variable, plus some additional variants. The result is that parallel pointers in old C* are exceedingly complicated to program with. It appears that the same complexities would arise in new C* and were omitted for this reason, resulting in severe non-orthogonalities and restrictions. See [32] for a more detailed critique of

C*. These difficulties in compiler and language design would vanish with a hardware-provided shared address space.

3.3 Benchmark Results

Presently, our benchmark suite consists of thirteen problems collected from literature [1, 6, 8, 9, 16]. For each problem we implemented the same algorithms in Modula-2*, in sequential C, and in MPL[1]. Then we measured the runtimes of our implementations on a 16K MasPar MP-1 (SIMD) and a SUN-4 (SISD) for widely ranging problem sizes. Measurements for LANs and KSR-1 are not yet available.

Modula-2* Programs. In Modula-2* we employ our libraries wherever possible. A technical deficiency in our current Modula-2* compiler forced us to manually "unroll" two-dimensional arrays into one-dimensional equivalents. This will no longer be necessary in the future.

MPL Programs. In MPL we have implemented the same algorithms as in Modula-2* and carefully hand-tuned them for the MasPar MP-1 architecture. The MPL programs make extensive use of registers, neighbor communication, standard library routines, and documented recommendations and programming tricks. We are quite certain that we have not overlooked possible optimizations because the MPL programs were reworked numerous times and the best implementation chosen by careful measurement. However, to ensure the fairness of the comparison, the MPL programs are as generally scalable as their Modula-2* counterparts. Since scalability is not restricted to multiples of the number of processors, boundary checks are required in every virtualization loop.

Sequential C Programs. The sequential C programs implement the parallel algorithms on a single processor. We use optimized libraries wherever possible.

In the following, we first compare the resource consumption of these three program classes. Secondly, we discuss their overall performance. The individual benchmark problems and their performance are presented in [21]. Finally, we show the quantitative effects of the optimization techniques.

[1] MPL [17] is a data-parallel extension of C designed for the MasPar MP series. In MPL, the number of available processors, the SIMD architecture of the machine, its 2D mesh-connected processor network, and the distributed memory are visible. The programmer writes a SIMD program and a sequential frontend program with explicit interactions between the two. MPL provides special commands for neighbor and general communication. Virtualization loops and distributed address computations must be implemented by hand.

Resource Consumption

The comparison is based on the criteria program space, data space, development time, and runtime performance.

Program Space. Our compiler translates Modula-2* programs to MPL or C. The resulting programs consume slightly more space than the hand-coded MPL or C programs. Regarding source code length, Modula-2* programs are typically half the size of their corresponding MPL or C programs.

Data Space. The memory requirements of the Modula-2* programs are typically similar to those of the MPL and C programs. Memory overhead is limited to variable replication into temporaries which occurs during synchronous assignments. This replication, however, is most often also required in hand-coded MPL. Furthermore, there is some additional overhead caused by controlling synchronous, nested, and recursive parallelism (16 bytes per `FORALL`).

Development Time. Due to compiler errors detected while implementing the benchmarks, we cannot give exact quantitative figures on implementation and debugging time. However, we estimate that the implementation effort in Modula-2* is not more than one fifth of the MPL effort.

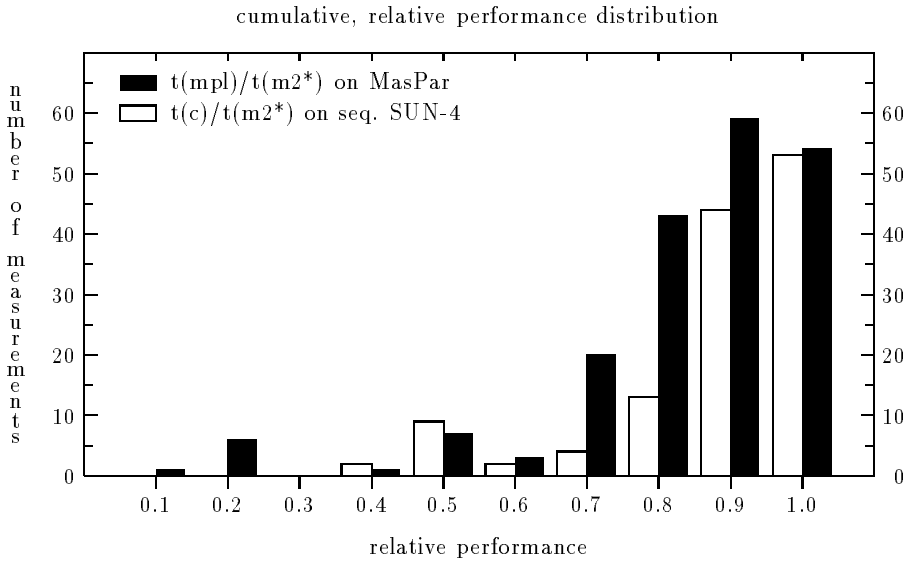
Runtime Performance

MPL versus Modula-2*. The general relative performance of Modula-2* is quite stable over all problem sizes and averages to 80%. of the MPL performance. Problems that can be implemented in MPL with a high amount of neighborship communication on arrays with multiple dimensions, currently perform quite bad in Modula-2* since the necessary optimization is not implemented yet.

Sequential C versus Modula-2*. The general relative performance of Modula-2* is again quite stable over all problem sizes and averages to 90% of the sequential C performance.

For widely varying problem sizes we measured the runtime of each test program on a 16K MasPar MP-1 and a SUN-4. We used the high-resolution DPU timer on the MasPar and the UNIX `clock` function on the SUN-4 (sum of user and system time). Below, t_{m2*} represents the Modula-2* runtime on either a 16K MasPar MP-1 or a SUN-4 (as appropriate); t_{mpl} gives the MPL runtime on a 16K MasPar MP-1; t_c stands for the sequential C runtime on a SUN-4.

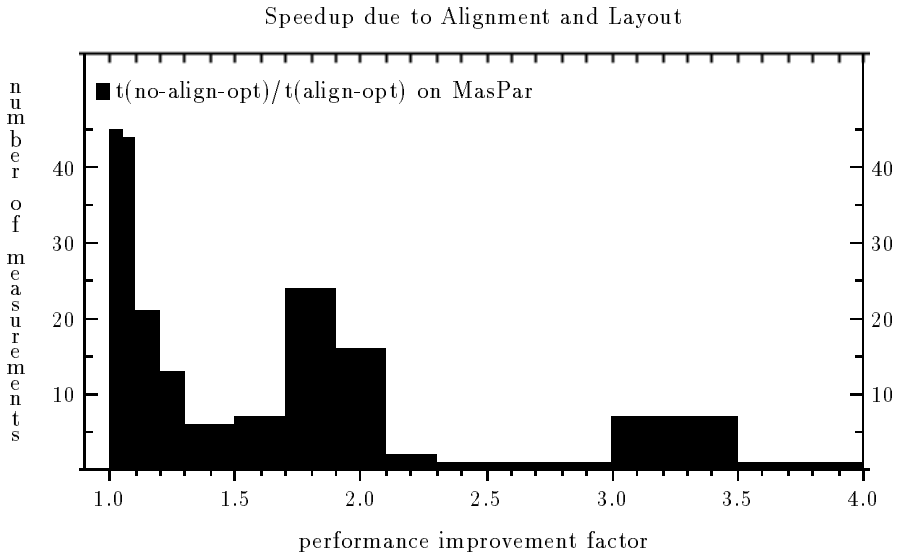
We define performance as problem size per time unit and focus on performances $\frac{size}{t_{m2*}}/\frac{size}{t_{mpl}} = t_{mpl}/t_{m2*}$ and $\frac{size}{t_{m2*}}/\frac{size}{t_c} = t_c/t_{m2*}$.



The overall distribution of relative performances proves to be encouraging. The above histogram provides the number of relative performance values falling into one of the classes [0%–5%), [5%–15%), ..., [95%–100%]. The numbers are the accumulated sums over all problems and problem sizes (all data points).

Effect of the Optimizations

Alignment and Layout. Data locality should pay off since data access involving communication is slower than access to local memory. The following diagram compares the runtimes of two versions. The first version has no `ALIGNED WITH` clause in the program text ($t_{no-align-opt}$). The compiler produces code that detects dynamically at runtime whether addresses are local or not. In the second version ($t_{align-opt}$), alignment optimization in the compiler has produced `ALIGNED WITH` information. Thus, the code generator statically knows about locality.

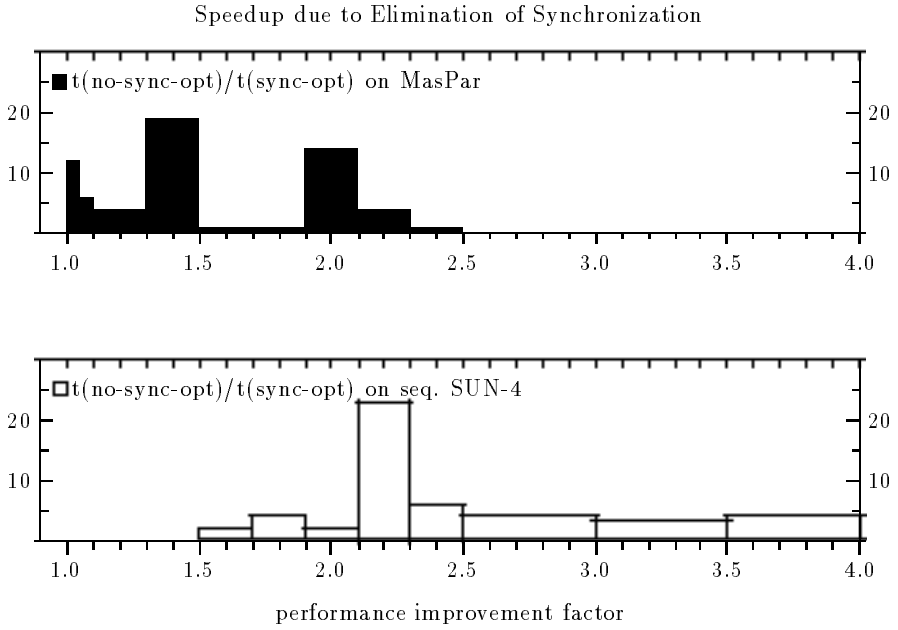


On the MasPar MP-1, this optimization improves runtime performance by 40% on average. The advantage of statically determined locality grows with the amount of data accessed. No differences could be measured on a sequential workstation since all accesses are local.

Elimination of Synchronization Barriers. The elimination should pay off for machines without synchronization hardware. Most MIMD machines, for example, synchronize by message passing which can be two or three orders of magnitude slower than instruction execution. However, synchronization barrier elimination is beneficial even on SIMD machines because it reduces virtualization overhead and the number of temporary variables needed. Furthermore, it may improve register usage.

The following diagrams show the performance ratio between runs without and with elimination of synchronization barriers ($t_{no-sync-opt}/t_{sync-opt}$) for the MasPar and the SUN-4.

Synchronization barrier elimination improves runtime by over 40% on a MasPar MP-1 and by over a factor of 2 on sequential workstations. Originally, the benchmark programs had a total of 278 synchronization barriers which were reduced to 109 by applying the optimization.



On SISD and MIMD machines, the performance improvement stems from the fact that fewer virtualization loops and fewer temporaries are needed. On a workstation, loop control and computation is done by the same processor. Without the elimination of synchronization barriers more than 50% of the runtime is used for loop control and memory access for additional temporaries. On the MasPar MP-1, loop control is performed by the fast frontend processor whereas the computation is done by the much slower parallel processors. Since the optimization technique only affects the frontend part, the relative performance gain is smaller for the MasPar MP-1 than that achieved on sequential workstations.

4 TRITON/1

The poor programmability of today's parallel machines is a consequence of the fact that the design of these machines has been driven mostly by hardware considerations. Programmability seems to have been a secondary issue, resulting in languages designed specifically for a particular machine model. Such

languages do not satisfy the needs of programmers who need to write machine-independent applications.

General Architecture. Triton/1 is a SAMD (synchronous/asynchronous instruction streams, multiple data streams) machine: it runs in SIMD mode where strict synchrony is necessary; it can switch to MIMD mode where concurrent execution of different tasks is beneficial. It is even possible to run a subset of the processors in SIMD mode and the other in MIMD. Thus, Triton/1 is truly SAMD, i.e., mixed-mode, not just switched-mode. Only a few research prototypes of mixed-mode machines have been built: OPSILA, TRAC, AP1000, and PASM [2, 13, 27]. Triton/1 provides support for switching rapidly between the two modes. With Modula-2* we have a high-level language to control both modes effectively.

Fast Barrier Synchronization. Fast barrier synchronization is supported by special synchronization hardware both in SIMD and MIMD mode. Synchronization with hardware support overcomes the necessity of coarse grained parallelism.

Network. We chose the De Bruijn network for Triton/1 because it has several desirable properties (logarithmic diameter, fixed degree, etc.), is cost-effective to build, and can be made to operate extremely fast and reliably. In section 4.3 we present performance figures.

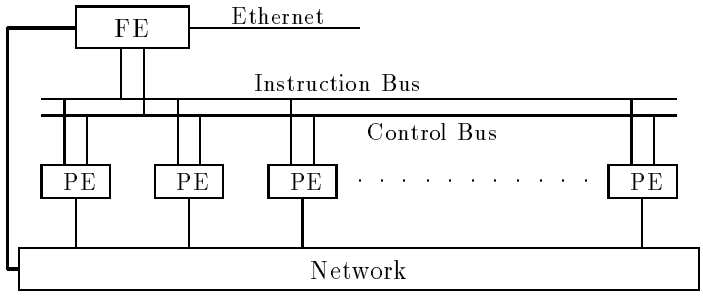
Scalability and Balance. Parallel machines should scale in performance by varying the number of processors and by adapting to progress in technology. Furthermore, the performance of the individual components (processor, memory, network, and I/O) should harmonize. Scalability in size is mainly a property of the network. Popular networks do not scale well: hypercubes are too expensive because they have variable degree. Grids cause high latency because of large diameter. Triton/1's De Bruijn net has none of these problems and, hence, scales well. It is also well matched to the speed of the processors. Section 4.4 comments on the scalability of Triton/1 in terms of technology.

I/O Capabilities. I/O must also scale with the number of processors. Few parallel machines provide for scalable I/O. Triton/1 implements a massively parallel I/O architecture: one disk per processor. For large sets of disks we have extended the traditional notion of a file to what we call a *vector file*. Massively parallel I/O also provides the basis for research in parallel operating systems, such as virtual memory, parallel paging strategies, and true multi-user environments. Results in these areas are required in order to bring parallel machines into wide-spread use.

4.1 Architecture of Triton/1

Triton/1 is divided into a frontend and a backend portion. The frontend is a UNIX workstation with an interface connected to the backend portion via the instruction and the control bus. The backend portion consists of the processing elements, the network, and the I/O system.

The Triton/1 prototype uses an Intel 486 based PC running BSD UNIX as its frontend. The prototype will contain 256 + 4 PEs, 72 of which are supplied with a disk. 256 of the PEs are provided for computation and 4 PEs are for hot stand-by. These PEs can be configured under software control into the network if other PEs fail. The reconfiguration involves changing the PE numbers consistently and recomputing the routing tables in the network processors. The 72 disks are logically organized in 8 groups of 9 disks where each group contains 8 data and one parity disk. RAID level 3 [19] is used for error handling. The logical organization of Triton/1 is presented in the following diagram:

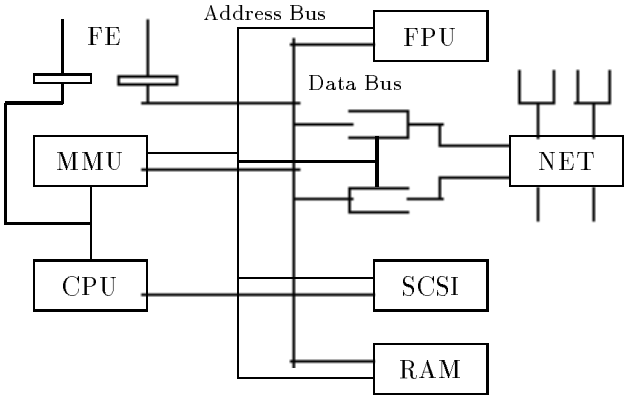


In SIMD mode, the frontend produces the instruction stream and controls the backend portion at instruction level. In MIMD mode, the frontend is responsible for downloading the code and the initiation of the program. The instruction bus is 16 bits wide. For reasons of decoupling frontend and backend in order to reduce the time of the frontend waiting for the backend to become ready, or vice versa, the instruction stream is sent through a fifo. The handshake signals necessary to control the instruction stream are part of the control bus.

The common address space of Triton/1 is needed by the so called *analyze mode*. In this mode the frontend has direct access to the local memory of all backend processing elements. It can be used for frontend controlled data transport between frontend and backend (both directions) and for reasons of debugging. To support the analyze mode the control bus includes 40 address

lines and several dedicated control signals; the instruction bus is used for data transport. While in analyze mode, all PEs release their local busses to enable direct memory access from the frontend.

The processing elements are designed as universal computing elements, capable of performing computation as well as service functions. Each PE consists of a Motorola MC68010 micro-processor, a memory management unit (MC68541), a numeric co-processor (MC68881 or MC68882), 2 MBytes of main memory, a SCSI interface, a network-processor, and the frontend bus. No extra controllers for mass storage access or any other I/O are necessary. The following figure shows the architecture of the processing elements of Triton/1 and emphasizes the novel aspects, i.e., the differences to a traditional sequential architecture.



The network of Triton/1 consists of the network-processors included in the PEs, the interconnection lines realized with flat cables, and fifo buffers for intermediate buffering of data packets. The network can route data packets from their source to their destination asynchronously, i.e., without interfering with the PEs. Non-interference permits latency hiding techniques to be applied in the compiler. The interface between a PE and its respective network-processor is also implemented with fifos because of decoupling reasons.

Parity checking of main memory, network links, and mass storage implements error detection. Periodic signature tests locate malfunctioning elements.

The architecture of Triton/1 matches the following recommendations derived in section 3 to support the translation of high-level languages:

- Hardware support for fast creation and termination of processes and for context switching. (The processor and the memory management unit provide hardware support for virtual memory and special instructions to build process management routines.)
- Hardware support for fast synchronization. (Several global-OR signals are provided both for SIMD and MIMD processing.)
- Hardware support for synchronous and asynchronous parallelism (i.e. fast switching between SIMD and MIMD mode).

4.2 Details of Selected Hardware Aspects

Instruction Bus and Control Bus

The instruction and control busses are implemented by a hierarchy of bus drivers for signals from the frontend to the backend. For the opposite direction, *global-OR* lines are emulated by explicit OR-combination of the signals from individual PEs. In SIMD mode all PEs read and execute the same instruction or idle. This is controlled with a three-way handshake protocol.

The general problem with handshake protocols on hierarchies of bus drivers is that these introduce a non-negligible amount of delay if the signals must traverse the complete hierarchy several times. In Triton/1 this delay is reduced by employing instruction buffers at each driver level. Thus, the handshake protocol is executed between every two hierarchy levels in pipeline fashion, rather than between the frontend and the PEs. This technique results in the same duration of instruction fetch in SIMD and MIMD mode.

Global Address Space

As mentioned above, 40 bit addresses are used to implement the global address space. The least significant 23 bits are used to select the memory and the memory mapped I/O in the PEs (up to 16 MByte). The next 16 bits are used to identify the PE to be accessed (up to 64k). The remaining bit distinguishes between frontend and backend addresses. The id of the PEs is twofold. Each PE has a hardware id which is selected by a switch setting. Additionally, each PE has a software identification which is used while computing. Initially, the software id is set to the same value as the hardware id; but it may change during operation because of reconfiguration.

As long as the total size of memory (over all processors) stays below 4 GBytes, the internal 32 Bit address arithmetic of the MC68010 is sufficient. For larger configurations, the 40 address bit arithmetic must be simulated in software.

SIMD/MIMD Mode-Switching

In SIMD mode the function codes of the processor are used to determine whether the processor accesses data or instructions. Accordingly, the processor bus is connected to the local memory or the instruction bus, respectively. The values of the program counters are completely ignored in SIMD mode. In order to switch to MIMD mode, a program must be downloaded to the memory of the PEs. Downloading is done via the instruction stream in SIMD mode. Thus, the distribution of code is, in contrast to many other MIMD machines, accomplished in a time proportional to the length of the code, independent of the size of the machine. The switch from SIMD to MIMD mode is performed by two instructions. With the first instruction, the program counter is set according to the location of the program to be executed in MIMD mode. With the second instruction, the MIMD request in the command register local to the PE is activated. The PE then switches to MIMD mode at the end of the current cycle and commences execution of the local code without delay. To switch from MIMD to SIMD mode, the MIMD request in the local command register is deactivated which causes the PE to switch to SIMD mode at the end of the current cycle. The next instruction is then expected form the instruction stream.

Data Transfer

There are several different data paths to consider in parallel computers. The most important one is the PE-network (see section 4.3). Another important path is the data transport from the frontend to the backend and vice versa. There are different possibilities for each direction: to transport data from the frontend to the backend, the easiest way is to send the data as *immediate data* via the instruction stream in SIMD mode. With that possibility any subset of the PEs can be the destination of the data. Unfortunately, only unidirectional access is possible. The second possibility is direct memory access in analyze mode. Here, data can be transferred in both directions. The drawback of the analyze mode is that no computation can take place and only one single PE can be accessed at any time. The third is to use the network. There is one dedicated network node connected to the frontend. This path is especially useful for transmitting more than a few bytes from different PEs to the frontend,

e.g. picture data. Another advantage of a network node included in the frontend is that parallel computation can commence while data is being transported.

Fast Barrier Synchronization in MIMD Mode

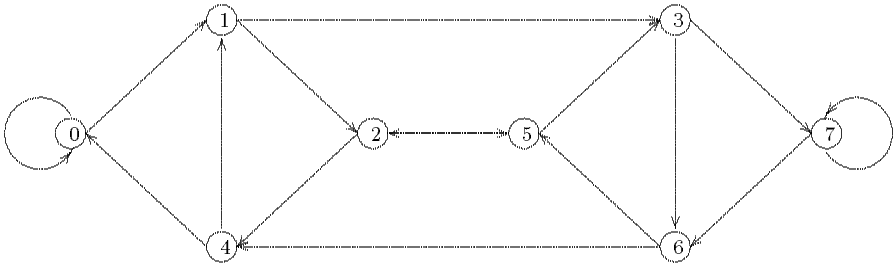
If all PEs are executing the same code, barrier synchronization is easily done by a global-OR line. Each PE sets its ready bit to true as soon as it reaches the synchronization barrier. Approximately one clock cycle after the last PE sets its bit, the frontend recognizes it and notifies the PEs on the result line.

In general MIMD case more than one group of processes exists. But the single global-OR line cannot be partitioned according to the process distribution. To implement barrier synchronization with several groups of processes the global-OR line is administrated by the frontend as a synchronization resource. Each group of processes is identified by a unique process group number. Initially, each PE is allowed to request the synchronization line on behalf of a group. The request is performed by the first PE reaching a barrier which interrupts the frontend and sends the group identification via the analyze circuits. If more than one PE reaches a barrier at once, the analyze circuits will select one randomly. The frontend then knows which group demands the synchronization line. Next, the frontend interrupts all PEs and forces them into SIMD mode to perform a barrier setup. The PEs not belonging to the requesting group are prohibited to request the sync line themselves. They also turn on their ready bits. The PEs belonging to the requesting group set their ready bit to true if they already reached the barrier, otherwise to false. After this setup phase, the PEs return to MIMD mode and continue computation, independently of their group membership. As soon as the last ready bit is turned on, the group owning the global-OR line synchronizes and releases the sync line. The frontend then releases the request prohibition in order to enable other groups to synchronize.

4.3 Communications Network

The Triton/1 network is based on the generalized De Bruijn net [3, 15] that can be characterized by the following interconnection rule: assuming all nodes are labeled 0 through $N-1$, a node with label X has direct connections to the nodes with labels $(2 * X + \alpha) \bmod N$ where $\alpha \in \{0..d-1\}$. The out-degree d equals the in-degree. The number N of nodes in the network is not limited to powers of two. The (maximum) diameter is $\lceil \log_d N \rceil$. The average diameter is well below $\log_d N$ and in practice quite close to the theoretical lower bound, the average diameter of directed Moore graphs. (Example: the average diameter of

the De Bruijn net with $N = 256$ and $d = 2$ is only 5% worse than the average diameter of the optimal Moore graph with same number of nodes. In general, Moore graphs cannot be realized in practice.)

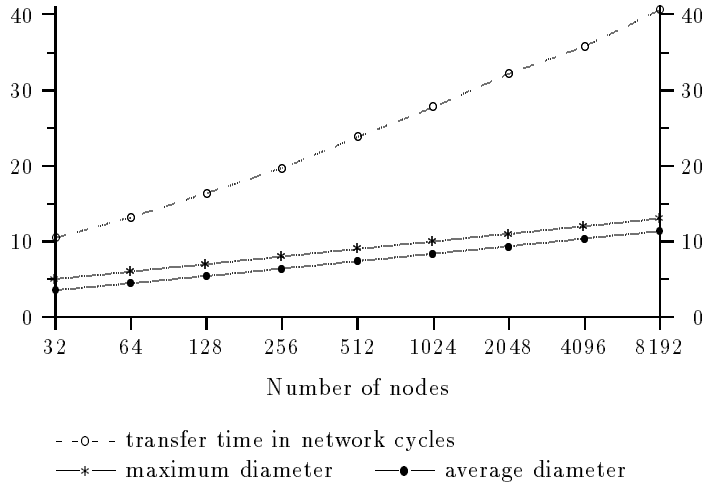


In our implementation we use degree $d = 2$, which makes our net a perfect shuffle. The above graph illustrates the structure of a De Bruijn net with 8 nodes. In comparison with other frequently used networks, this design has the benefits of a constant degree per node and a small average diameter. Data transport is done via a table-based, self-routing packet switching method which allows virtual-cut-through-routing and load dependent de-touring of packets. Every node has its own routing table and three input buffers: two for intermediate storage of data packets coming from other nodes and one to communicate with its associated PE. An output buffer is used to deliver data packets to the associated PE. Buffering temporally decouples the network from local processing. Packets contain the address of the target node, the length of the message, and the data itself. The size of the packets can range from 4 to 506 bytes.

To achieve low latency, we implemented the communications processor at the gate level with a programmable gate array. The throughput of one communications processor – measured on our prototype – is 800,000 packets per second (32 bit user data) which is equivalent to 0.8 MCPS. Thus, the communications processor is three to five times faster than the PEs can read or write packets. Therefore, we have adequate performance to build large networks and to operate them under heavy loads without suffering from high latency.

The communications processor routes the packets without interfering with the PE. Optimal routes are stored in a routing table per communications processor. Hence, the network can transport data concurrently to the operation of the processing elements. This feature can be used by the compiler to overlap communication and computation.

In order to analyze the behavior of the network, we built a simulator based on the measured performance of a single communications processor. We simulated the overall performance of the network in various modes. The number of nodes examined ranged from 32 to 8,192. The results of a series of experiments with a random communication pattern are given in the following diagram.



Both the sender and the receiver were chosen randomly, with the restriction that the number of data packets to be transported equals the number of nodes in the network. The simulation shows that the network scales well: the delay introduced by the network lies within $O(\log N)$, where N denotes the number of nodes and messages.

The robustness against overload is surprisingly good. Even if all processing elements send a large number of packets simultaneously, the overall throughput of the network does not decrease. Irregular permutations are performed especially fast. All "hard" patterns known from literature, e.g., transposition of a matrix, butterfly, and bit reversal are delivered at average speed or faster.

A severe disadvantage of De Bruijn networks is that they are not deadlock free. If used as a self-routing packet switching network with a limited buffer size and no possibility of re-arranging the packets in the buffers, deadlock can occur. Dally et al. [5] show that a network is deadlock free if its dependency graph is free of cycles. This condition is easy to prove for hypercubes; it does not hold for De Bruijn nets. Although several other methods for deadlock avoidance are known from literature, none of them apply to De Bruijn nets.

We have implemented a combination of three methods to cope with deadlocks, contention, and starvation. The first one uses a static priority scheme to reduce the likelihood of deadlocks. This gives us something similar to a priority-path through the network that has to be mapped to a hamiltonian cycle within the De Bruijn net. The second method is a static packet insertion rule which prevents overloading of the network. The third method uses de-touring of packets if it detects contention. In case of starvation, we use a routing similar to the one used in Denelcor HEP [28] although without explicit packet priorities.

The design of our network matches all the network related recommendations of section 3.

- The MCPS measure approaches the MIPS measure (0.8 MCPS vs. 0.8 MIPS).
- The network operates independently with asynchronous message delivery.
- Triton/1 provides uniform communication instructions. (There is no difference between neighbor, global, and frontend communication.)

4.4 Scalability

Scalability of parallel machines can be interpreted in two ways – scalability within the size of the machine and scalability in terms of technology. The Triton/1 architecture scales well in size. With our current hardware we are able to connect up to 4,096 PEs. The design allows for up to 64k PEs.

Scaling Triton/1 in technology raises some difficulties. The De Bruijn net and the I/O-System should scale well. Upgrading to a state-of-the-art microprocessor requires to change the 16 bit design to a 32 bit or even 64 bit design which is simple. Also the upgrade of our network processor, a FPGA with about 1,500 gates, is quite easy for an experienced VLSI-designer. However, the frontend instruction bus needed for SIMD processing is crucial, since the delay induced by the hierarchical bus architecture cannot be reduced in the same way that must be expected of the performance improvement of standard processor chips. Provided that we have hardware support for fast barrier synchronization, our experience in compiler technology shows how to produce efficient code. Since the synchronization hardware of Triton/1 – a reduction/broadcast tree – scales well, the Triton/1 architecture, given minor modifications, must be considered to scale well in terms of technology.

5 STATUS AND FUTURE

Compilers

A first non-optimizing Modula-2* compiler targeting the Connection Machine CM-2 was operational in 1991. Since spring 1992, optimizing Modula-2* compilers for the MasPar MP-1 and sequential machines are available. Modula-2* compilers for the KSR-1 and networks of workstations are under construction. Contact `msc@ira.uka.de` if interested. Further research will focus on Modula-3* [10], nested and recursive parallelism, as well as other optimizations for latency hiding and parallel expression evaluation.

The migration to a new compiler becomes necessary because our current implementation has already reached its limits of extensibility. This also offers the opportunity to move to a new language. Modula-3* clearly distinguishes between the orthogonal properties *range* and *mode* (synchronous or asynchronous) of parallel computations and supports more modern concepts than its predecessor, e.g. parallel object-oriented programming.

Triton/1

The prototype of the first node was completed in October 1991. The individual components (communication processor, PE, and control processor interface) are tested and running according to their specifications. Manufacturing of the printed circuit boards is in progress. The final assembly of Triton/1 will be completed in 1993.

One serious bottleneck in the current Triton/1 architecture is the missing network packaging unit. This unit should be capable of taking an address in the global address-space of Triton/1, recognize the remote processor number, and automatically send the corresponding packet through the network. Moreover, in case of a data-fetch request, the packaging-unit of the responding processor should be able to process data requests and send the requested data back to the originating processing element. Currently, this functionality is implemented with low-level libraries. We expect that hardware support for these features will improve performance of remote data access at least by an order of magnitude. Further research will focus on novel architectures to overcome the network latency problem.

6 CONCLUSION

Massively parallel machines are now beginning to be used routinely. With increased use, the programmability of these machines has become an important concern. We have proposed simple language extensions that allow for clear and portable expression of parallel algorithms in most imperative programming languages. We have demonstrated that these language constructs can be compiled effectively for parallel machines. Beside further research in optimization techniques, parallel architectures must be tuned to the needs of higher-level languages and the capabilities of compilers. We have proposed and are still evaluating promising hardware features that would support problem-oriented explicitly parallel programming languages.

Acknowledgements

We would like to thank our students, especially Boris Bialek, Udo Böhm, HaJo Brunne, Thomas Gauweiler, Stefan Hänßgen, Oliver Hauck, Ralf Kretzschmar, Michael Längle, Hendrik Mager, and Markus Mock for many valuable ideas and their implementation work.

REFERENCES

[1] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[2] M. Auguin and F. Boeri. The OPSILA computer. In M. Consard, editor, *Parallel Languages and Architectures*, pages 143–153. Elsevier Science Publishers, Holland, 1986.

[3] N. G. De Bruijn. A combinatorial problem. In *Proc. of the Sect. of Science Akademie van Wetenschappen*, pages 758–764, Amsterdam, June 29, 1946.

[4] Peter Christy. Virtual processors considered harmful. In *Proc. of the 6th Distributed Memory Computing Conference*, pages 99–103, Portland, Oregon, April 28 – May 2, 1991.

[5] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C–36(5):547–553, 1987.

[6] John T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems.* Elsevier Science Publishers, Holland, 1992.

[7] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[8] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[9] Philipp J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press Cambridge, Massachusetts, London, England, 1991.

[10] Ernst A. Heinz. Modula-3*: An efficiently compilable extension of Modula-3 for explicitly parallel problem-oriented programming. In *Joint Symposium on Parallel Processing*, pages 269–276, Waseda University, Tokyo, May 17–19, 1993.

[11] Ernst A. Heinz and Michael Philippsen. Synchronization barrier elimination in synchronous forall statements. Technical Report No. 13/93, University of Karlsruhe, Department of Informatics, April 1993.

[12] Christian G. Herter, Thomas M. Warschko, Walter F. Tichy, and Michael Philippsen. Triton/1: A massively-parallel mixed-mode computer designed to support high level languages. In *7th International Parallel Processing Symposium, Proc. of 2nd Workshop on Heterogeneous Processing*, pages 65–70, Newport Beach, CA, April 13–16, 1993.

[13] Takeshi Horie, Hiroaki Ishihata, Toshiyuki Shimizu, Sadayuki Kato, Satoshi Inano, and Morio Ikesaka. AP1000 architecture and performance of LU decomposition. In *Proc. of the 1991 International Conference on Parallel Processing*, volume I, pages 634–645, August 1991.

[14] High Performance Fortran (HPF): Language specification. Technical report, Center for Research on Parallel Computation, Rice University, 1992.

[15] Makoto Imase and Masaki Itoh. Design to minimize diameter on building-block network. *IEEE Transactions on Computers*, C–30(6):439–442, June 1981.

[16] Joseph JáJá. *An Introduction to Parallel Algorithhms.* Addison-Wesley, Reading, Mass., 1992.

[17] MasPar Computer Corporation. *MasPar Parallel Application Language (MPL) Reference Manual*, September 1990.

[18] Piyush Mehrotra and John Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, November 1987.

[19] David A. Patterson, Garth Gibons, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RIAD). In *Proc. of the 1988 ACM-SIGMOD Conference on Management of Data*, pages 109–116, Chicago, June 1–3, 1988.

[20] Michael Philippsen. Automatic data distribution for nearest neighbor networks. In *Frontiers '92:The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 178–185, Mc Lean, Virginia, October 19–21, 1992.

[21] Michael Philippsen, Ernst A. Heinz, and Paul Lukowicz. Compiling machine-independent parallel programs. *ACM SIGPLAN Notices*, 28(8):99–108, August 1993.

[22] Michael Philippsen and Markus U. Mock. Data and process alignment in Modula-2*. In Christoph W. Kessler, editor, *Automatic Parallelization – New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 171–191, AP'93 Saarbrücken, Germany, March 1–3, 1993, 1994. Verlag Vieweg, Wiesbaden, Germany, Advanced Studies in Computer Science.

[23] Michael Philippsen and Walter F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991*, pages 169–183. Springer Verlag, Lecture Notes in Computer Science 591, 1992.

[24] Lutz Prechelt. Measurements of MasPar MP-1216A communication operations. Technical Report No. 1/93, University of Karlsruhe, Department of Informatics, January 1993.

[25] M. Rosing, R. Schnabel, and R. Weaver. DINO: Summary and example. In *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 472–481, Pasadena, CA, 1988. ACM Press, New York.

[26] David Sankoff and Joseph B. Kruskal (eds). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Mass., 1983.

[27] H.J. Siegel, T. Schwederski, J.T. Kuehn, and N.J. Davis. An overview of the PASM parallel processing system. In D.D. Gajski, V.M.Milutinovic, H.J.Siegel, and B.P. Furht, editors, *Computer Architecture*, pages 387–407. IEEE Computer Society Press, Washington, DC, 1987.

[28] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Real Time Signal Processing IV, Proceedings of SPIE*, pages 241–248. International Society for Optical Engineering, 1981.

[29] Thinking Machines Corporation, Cambridge, Massachusetts. **Lisp Reference Manual, Version 5.0*, 1988.

[30] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Language Reference Manual*, April 1991.

[31] Walter F. Tichy and Christian G. Herter. Modula-2*: An extension of Modula-2 for highly parallel, portable programs. Technical Report No. 4/90, University of Karlsruhe, Department of Informatics, January 1990.

[32] Walter F. Tichy, Michael Philippsen, and Phil Hatcher. A critique of the programming language C*. *Communications of the ACM*, 35(6):21–24, June 1992.

[33] Reinhard v. Hanxleden and Ken Kennedy. Relaxing SIMD control flow constraints using loop transformations. Technical Report CRPC-TR92207, Center for Research on Parallel Computation, Rice University, April 1992.