THIS PAPER APPEARED IN: FRONTIERS'95: THE FIFTH SYMPOSIUM ON THE FRONTIERS OF MASSIVELY PARALLEL COMPUTATION, PAGES 350–357, MCLEAN, VIRGINIA, FEBRUARY 6–9, 1995

Automatic Synchronization Elimination in Synchronous FORALLs

Michael Philippsen and Ernst A. Heinz
IPD, University of Karlsruhe, Germany, email: (phlipp | heinze) @ ira.uka.de

Abstract

This paper investigates a promising optimization technique that automatically eliminates redundant synchronization barriers in synchronous FORALLs. We present complete algorithms for the necessary program restructurings and subsequent code generation. Furthermore, we discuss the correctness, complexity, and performance of our restructuring algorithm before we finally evaluate its practical usefulness by quantitative experimentation.

The experimental evaluation results are very encouraging. An implementation of the optimization algorithms in our Modula-2 compiler eliminated more than 50% of the originally present synchronization barriers in a set of seven parallel benchmarks. This barrier reduction improved the execution times of the generated programs by over 40% on a MasPar MP-1 with 16384 processors and by over 100% on a sequential workstation.*

1 Introduction

Data-parallel programs operate on all elements of a data structure simultaneously and are expressed with explicit or implicit FORALLs. During the compilation of FORALLs a synchronization barrier has to be implemented between potentially interfering data references if the compiler cannot assure the absence of data dependences. Hence, the primary optimization goal is to cover all detected dependences with as few synchronizations as possible.

We tackle this optimization problem by means of a restructuring technique based on source-to-source transformations in the framework of Modula-2* [18]. Our restructuring algorithm covers all language features of Modula-2*, including branches, loops, and procedure calls inside synchronous FORALLs as well as arbitrary nestings thereof. In general, reduction of synchronization barriers increases the amount of temporary storage. Thus, we face the secondary optimization problem of minimizing this increase.

The remainder of the paper is organized as follows. After discussing related work in section 2, we briefly introduce our notation of FORALLs. Section 4 formulates the restructuring algorithm and discusses its properties. Finally, section 5 describes the setup and results of the experiments evaluating the effectiveness of our techniques.

2 Related Work

Several researchers have studied different variations of the synchronization elimination problem in the context of compiling data-parallel programs [7, 10, 13, 14, 16, 17].

Our approach shares some similarities with the work of Hatcher and Quinn [10]. They use a data-parallel language that assigns a private address space to each virtual processor. Data from other virtual processors can only be accessed by explicit communication. Hence, synchronizations are only necessary where communications occur. The number of barriers is reduced by grouping communication operations together.

In languages with a shared address space the problem is more complicated. Here, synchronization barriers are not explicitly visible; compilers need sophisticated data dependence analysis to capture access interferences. Furthermore, our solution is more general than the work of Hatcher and Quinn because our restructuring algorithm works on (sub-) expressions and is extremely fine grained.

The article [7] by Chatterjee focuses on the compilation of VCODE for shared memory multiprocessors. VCODE is a low-level, data-parallel vector language intended to serve as the target for optimizing compilers of higher level languages. It is based on the shared address space paradigm and allows for nested parallelism.

VCODE programs do not contain any subscript expressions; this considerably simplifies the necessary data dependence analysis because the compiler needs no subscript tests. The VCODE compiler internally builds a so-called computation graph of the source program, which is similar to our graph representation of synchronous FORALLs. Their graph then serves as the basis for all optimizations, namely partitioning into clusters and epochs as well as runtime scheduling and storage minimization. Clustering simply amounts to the fusion of compatible vector operations which translates to a fusion of compatible FORALLs in our framework. Epoch formation resembles our restructuring translation from synchronous FORALLs into sequences of asynchronous FORALLs. As for this, Chatterjee's clusters and epochs closely correspond to our synchronous and asynchronous FORALLs, resp. His experiments and performance measurements – conducted for several parallel programs on 12 processors of a 16 processor shared memory Encore Multimax – further confirm the effectiveness of synchronization barrier elimination.

As for language framework and optimization goals, the recent work of Prakash et al. [17] closely follows our direction which originally stems from 1991 [16]. Prakash investigates synchronization elimination in the UC programming language, a data-parallel extension of C featuring a shared address space, built-in data-parallel operations like reductions, and two new statements that introduce parallelism. UC's `par` and `arb` statements are effectively equivalent to synchronous and asynchronous FORALLs, resp.

Prakash describes data dependence analysis and several possibilities of optimizing program transformations: array renaming, array alignment, barrier minimization, barrier weakening, definition variables, and fuzzy barriers plus non-blocking requests. Furthermore, a simple cost model which may serve as the foundation of future static performance estimation is introduced. But for none of the above [17] gives any concrete algorithms or implementation schemes. Hence, it remains unclear how to successfully automize the inter-play of the proposed optimizations. However, the implementation of an optimizing UC compiler is claimed to be in progress.

In contrast to Prakash, we focus on minimizing the number of synchronization barriers because this seems to be the most important optimization. Therefore we present and discuss complete algorithms automizing program restructuring and code generation. Moreover, we evaluate the practical usefulness of the proposed optimization techniques as implemented in our Modula-2* compiler [9, 14] by conducting quantitative experiments measuring the performance of Modula-2* programs on sequential workstations and a distributed memory MasPar MP-1.

3 Synchronous FORALLs

When speaking of *synchronous FORALLs* we mean high-level language constructs that allow for problem-oriented expression of synchronous parallelism. In Modula-2* the syntax of synchronous FORALLs is defined as follows.

```
FORALL <ident> ":" <SimpleType> IN SYNC
  <StatementSequence>
END
```

The `FORALL` creates as many (conceptual) processes as there are elements in the possibly non-static scalar range *SimpleType*. The identifier *ident* is local to the `FORALL` statement and serves as a runtime constant; it is initialized to a unique value of *SimpleType* for each process. The created processes execute *StatementSequence* in parallel synchrony. The `END` imposes an explicit synchronization barrier on the created processes. Termination of the whole `FORALL` statement is delayed until all processes have finished their execution of *StatementSequence*.

Synchronous FORALLs are especially good at implementing parallel modifications of overlapping data structures because the required synchronization need not be formulated explicitly. Even irregular data permutations are easy to understand and to program.

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

This `FORALL` permutes vector `X` according to permutation function `p()`. The synchronous semantics ensure that all RHS elements `X[p(i)]` are read and temporarily stored before any LHS variable `X[i]` is written.

The behavior of branches inside synchronous FORALLs is defined as follows: Modula-2* allows branches of synchronous `CASE` or `IF` statements to be executed concurrently without any synchronization. The exact synchronous semantics of nested statements are defined in [18].

4 Synchronization Barrier Elimination

4.1 Example

```
FORALL i : AnySimpleType IN SYNC
  A[i] := A[i+1] + A[i-1] + B[i];
  B[i] := B[i+1]
END
```

A number of parallel threads is created by this `FORALL` statement. Conceptually, the threads have to be synchronized after each individual subexpression, e.g. after the evaluation of `A[i+1]`, after the evaluation of `A[i-1]`, after the addition of both operands, after the evaluation of `B[i]`, and so on. Hence, a naive implementation would implement eight synchronization barriers.[1]

Changing the evaluation order is possible as long as certain constraints are obeyed. There are two types of such constraints. The first type reflects the semantics of individual statements, e.g. the RHS of a statement has to be completely evaluated before a store to the LHS can be performed. When representing constraints as edges in a program dependence graph, the above example will have two edges of this type; each connects the RHS with the LHS of one assignment. The other type of constraint is induced by data dependences as defined in [2, 3, 6, 19]. Data dependences may exist within one thread or between different threads; *intra-thread dependences* are similar to def-use chains (no synchronization necessary) whereas *inter-thread dependences* require synchronization between the threads.

In the example there are four data dependences: `A[i+1]` δ^a `A[i]`, `A[i-1]` δ^a `A[i]`, $\mathtt{B}_1$`[i]` δ^a $\mathtt{B}_2$`[i]` (intra), `B[i+1]` δ^a $\mathtt{B}_2$`[i]`.[2] All of them can be obeyed with a single synchronization barrier as shown in the following semantically equivalent code.

```
FORALL i : AnySimpleType IN PARALLEL
  H1[i] := A[i+1];
  H2[i] := A[i-1];
  H3[i] := B[i+1]
END
FORALL i : AnySimpleType IN PARALLEL
  A[i]  := H1[i] + H2[i] + B[i]
  B[i]  := H3[i]
END
```

Note that in contrast to the original code, two asynchronous FORALL statements are used. No implicit synchronization barriers remain inside the bodies of these asynchronous FORALLs. All aforementioned constraints are honored: the parts are evaluated in correct order,

[1]Because subexpressions may contain calls to functions with side-effects, barriers are required. For plain array references, however, some of the naive barriers can easily be removed.

[2]$\mathtt{B}_1$`[i]` stands for the first occurrence of `B[i]` in the first assignment. As usual δ^a denotes an anti dependence.

the three inter-thread dependences involve only data references that are in different asynchronous FORALLs and are therefore separated by a synchronization barrier.

Although the above code is semantically correct, there exists an even better solution which requires less memory (temporary variables) and access time.

```
FORALL i : AnySimpleType IN PARALLEL
  H1[i] := A[i+1] + A[i-1] + B[i];
  H3[i] := B[i+1]
END
FORALL i : AnySimpleType IN PARALLEL
  A[i]  := H1[i];
  B[i]  := H3[i]
END
```

4.2 Graph Representation

The basic data structure is a directed graph P representing a combination of a dependence graph and expression trees as they are commonly used for intermediate representation in compilers. For each nesting of FORALLs, a graph is constructed. Nodes of this graph are operands, e.g. designators, and operators. Nodes are connected by directed edges representing ordering constraints as explained in the following two subsections:[3]

4.2.1 Data Dependence Edges

By data dependence analysis the compiler tries to prove the absence of dependences in order to include as few edges as possible into the graph P. There are a few modifications of the usual data dependence analysis that are necessary to obtain the desired information in the context of synchronous parallelism.

- It is clear for sequences of assignment and branches that all data dependences run in a lexically positive direction since all threads execute all parts thereof in lock-step. In loops inside synchronous FORALLs we do not have to consider loop carried (i.e., lexically negative) dependences, since synchronous loop semantics [18] prescribe a barrier after each iteration.
- Hence, the resulting graph is acyclic.
- If two references to an array are both inside the same *asynchronous* FORALL, no data dependence edge is required, since the programmer explicitly allows the parallel threads to proceed with arbitrary speed.
- For branching statements, loops, and procedures inside the FORALL nesting, it must be detected which designators may cause data dependences. Our technique covers all these cases but due to space limitations we must refer the interested reader to [14].
- Prakash et al. [17] eliminate intra-thread dependences from their graphs. We show that keeping them in P leads to further optimization.

[3]The definition of P resembles both the program dependence graph PDG of [8] and the dependence flow graph DPG of [12]. Whereas nodes of PDG and DPG are complete statements, in P evaluation ordering is expressed on a subexpression basis.

4.2.2 Evaluation Ordering Edges

Although our restructuring and code generation techniques include the handling of branches, loops, and procedure calls [14], the current presentation is restricted to flat sequences of assignments for the sake of clarity.

To ensure the correct evaluation order inside of and between statements, additional edges are included into P: In case of assignments, operand nodes and operator nodes that occur on the RHS of the assignment are connected by a directed edge in P. The direction represents the required order of evaluation. The root of the expression tree for the RHS is then connected to the designator node of the LHS. For branching statements and loops, additional edges must be inserted.

4.3 Central Idea

The central idea of the restructuring optimization is to sort P topologically. The path with the maximal number l of inter-thread dependence edges determines the minimal number of synchronization barriers that are required; consequently l asynchronous FORALLs have to be generated.

With l known, we try to find l disjunctive subgraphs T_1, T_2, ..., T_l of P such that the following conditions hold:

- Each node N of P is mapped to exactly one T_i. We call i the *synchronization rank* of N. All nodes with synchronization rank i are in T_i.
- There is no forward path from a node $N \in T_j$ to a node $M \in T_i$ if $j > i$.
- Within one subgraph there are no two nodes that are connected by an inter-thread dependence edge. Thus, inter-thread dependences always connect nodes from different subgraphs.

Finding a subgraph partitioning is equivalent to computing appropriate synchronization ranks. For nodes that are on the path with the maximal number l of inter-thread dependence edges, the synchronization rank is fixed. For nodes on paths that have fewer inter-thread dependences, there is some freedom in assigning synchronization ranks.

This freedom can be used for a secondary goal of optimization: The number of evaluation ordering edges that link nodes in different subgraphs determines the number of variables necessary to store intermediate results. These edges prescribe that an intermediate result is used in another asynchronous FORALL than where it is computed. Usually, temporary variables are arrays with one element per thread.[4] Hence, when cutting P into subgraphs, the total number of evaluation ordering edges linking nodes with different synchronization ranks is to be minimized.

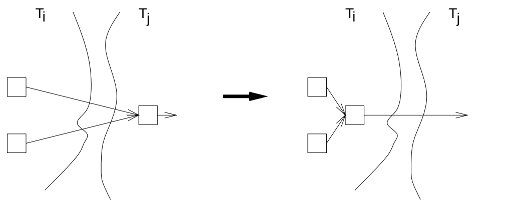
Since we suspect this problem to be NP-complete[5], we

[4]If no virtualization is necessary, i.e., if the number of threads does not exceed the number of available processors, registers can be used instead of arrays.

[5]We have not yet found a conclusive proof. For $l = 2$ the problem is NP-complete since it reduces to "minimum cut into bounded sets".

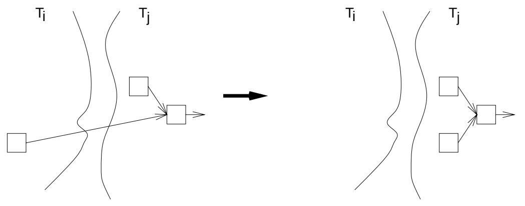
apply the following heuristics which are based on local information.

- **Successor Locality.** If there is a choice in mapping an operator node to the subgraphs, the best subgraph is the one in which the operands are evaluated.



  Instead of two intermediate results (left) only one (right) has to be stored in a temporary variable.

- **Predecessor Locality.** The same idea applies when mapping operand nodes. If there is a choice, the best selection puts the operand node in that subgraph in which the value is used.



  Instead of one intermediate result (left) no temporary storage is required (right).

4.4 Code Restructuring

Before the restructuring, edges are attributed with weights. Let $w_{P \to S}$ be the weight of the edge connecting P and S. Inter-thread dependence edges get a weight of 1, all other edges 0. The following algorithm subdivides a given graph P into subgraphs which fulfill the above conditions. For this purpose we compute a synchronization rank for every node. If there is a choice in mapping nodes to subgraphs, the algorithm uses an arbitrary selection strategy. In section 4.6 we add better heuristics.

Input. Graph P.

Output. Synchronization ranks for every node of P.

Data structures. Every node has two attributes: minimal synchronization rank $r \in \mathbb{N}$ and maximal synchronization rank $R \in \mathbb{N}$. The interval of possible synchronization ranks is $[r, R]$. The idea of the interval is that the node can be mapped to any of the subgraphs T_r, T_{r+1}, $\ldots T_R$ without violating any of the conditions for cutting P into subgraphs. After termination $r = R$ holds for every node, meaning that a synchronization rank is computed and P is cut into subgraphs.

Algorithm. The algorithm consists of two phases. During the first phase, the interval of possible synchronization ranks is computed for every node. A by-product is the number l of necessary synchronization barriers, which is the same as the number of subgraphs to be constructed.
The freezing phase handles the nodes with an unfixed synchronization rank ($r < R$) by selecting a $\rho \in [r, R]$ and then propagating this choice to adjust the intervals of possible synchronization ranks of neighboring nodes.

I. Computation of Intervals.

For each node, the interval of possible synchronization ranks is computed as follows.

I.1. Minimal Synchronization Rank. The graph P is sorted topologically. The minimal synchronization rank r of each node is initialized to 1. In topological order, the nodes update their values of r: The new value is the maximum of the old value and the values of the predecessors, incremented by the weights of the connecting edges. The maximum of all resulting r values is assigned to l.

I.2. Maximal Synchronization Rank. The maximal synchronization rank is computed with the dual algorithm. The direction of the edges is inverted, their weights are considered to be multiplyed by -1, and instead of maxima minima are computed. The initial value for each R is set to l.

II. Freezing of Ranks.

For each node of P, an interval of possible synchronization ranks $[r, R]$ is known. The final rank will be inside this interval but it depends on the synchronization ranks of neighboring nodes.

As long as there remain nodes with $r < R$ do:

II.1. Selection & Update. Select an arbitrary node K with $r < R$. For this node K choose any $\rho \in [r, R]$ and set the interval of synchronization ranks $[r, R] := [\rho, \rho]$.

II.2. Propagation. If an interval of possible synchronization ranks of a node K is updated, this may influence the intervals of neighboring nodes. In this case update their intervals as follows:

- The maximal synchronization rank of a predecessor V of K may not be larger than $R_K - w_{V \to K}$. If this condition does not hold after modifying the interval of K, set R_V accordingly. This update is propagated recursively.
- The minimal synchronization rank of a successor N of K may not be smaller than $r_K + w_{K \to N}$. If this condition does not hold after modifying the interval of K, set r_N accordingly. This update is propagated recursively.

After computing synchronization ranks, code is generated as follows. For each synchronization rank an asynchronous FORALL is generated in increasing order. In the body of FORALL i, code for all those nodes of P is generated that have synchronization rank i. The order of node implementation in the body, i.e. the resulting evaluation order, is determined by the edges with weight 0. Temporaries are used if evaluation ordering edges start at nodes with rank i and leave the subgraph T_i, or if nodes in T_i are destinations of evaluation ordering edges that start in T_j with $j < i$.[6]

4.5 Correctness, Minimality, Complexity

It is obvious that a transformation is semantically correct, if we can prove (a) that the necessary sequential intra-thread execution order is obeyed and (b) that a synchronization barrier is implemented between the source and destination of every inter-thread data dependence.

Based on the assumption that each evaluation ordering constraint and each intra-thread dependence is represented by an edge in P, condition (a) holds because for each synchronization rank the nodes are coded in the order determined by the edges with weight 0.

Thus, it is sufficient to show that the above restructuring algorithm will implement at least one synchronization barrier between source and destination of each inter-thread dependence $D_1 \, \delta \, D_2$. We assume that the data dependence analysis works correctly. For each inter-thread data dependence, P contains an edge with weight 1. All other edges have weight 0. The first phase of the algorithm ensures that for D_1 and D_2 the intervals of possible synchronization ranks $[r_1, R_1]$ and $[r_2, R_2]$ fulfill $r_1 + 1 \leq r_2$ and $R_1 + 1 \leq R_2$. This condition is an invariant of the freezing phase of the algorithm; it holds before selecting ρ. Since the new interval $[\rho, \rho]$ is inside the old one, the condition still holds. By propagating the update to neighboring nodes, their intervals may be altered without contradiction, since the weights of the edges are considered.

If there are only assignments in the body of the restructured synchronous FORALL, the algorithm finds the minimal number of synchronization barriers that are required to cover all corresponding dependences in P. This was motivated at the beginning of section 4.3. Depending on the quality and resolution of the initial dependence analysis, however, some semantically redundant barriers may still remain in the generated code.

With n edges and m nodes, the complexity of the topological sort is $O(n+m)$. In the worst case, n nodes remain with an unfixed synchronization rank after the first phase of the algorithm. The second phase fixes the synchronization rank and then propagates the update by means of a topological sort. Thus, the overall complexity amounts to $O(n \cdot (n+m))$.

[6] Code generation for nested synchronous FORALLs, branches, and loops is more complicated. Due to space limitations we kindly refer the interested reader to [14].

4.6 Heuristics

The number of temporary variables can be reduced if the freezing phase of the restructuring algorithms is more sophisticated. Instead of the "arbitrary selection" in step II.1 the successor and predecessor locality rules can be applied. Their application does not interfere with the above assertions concerning correctness, minimality, and complexity. The above algorithm has to be modified in step II.1:

II.1. Selection & Update. Apply the following rules with the given priority:

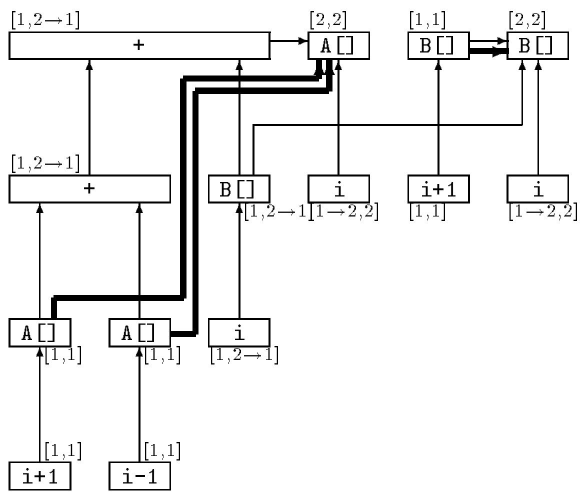
(a) **Successor Locality.** For each node K with $r < R$, consider all its predecessors on edges with weight 0. If some of these predecessors already have fixed synchronization ranks, let ρ be the maximum of these ranks. If $r \leq \rho \leq R$ holds for K, update the interval of K as $[r, R] := [\rho, \rho]$. If there is more than one node fulfilling the above precondition, use an arbitrary node with smallest $R - r$.

(b) **Predecessor Locality.** For each node K with $r < R$, consider all its successors on edges with weight 0. (As long as we deal only with assignments there is at most one successor.) If some of these successors already have fixed synchronization ranks, let ρ be the minimum of these ranks. If $r \leq \rho \leq R$ holds for K, update the interval of K as $[r, R] := [\rho, \rho]$. If there is more than one node fulfilling the above precondition, use an arbitrary node with smallest $R - r$.

(c) **Arbitrary.** Select an arbitrary node K with $r < R$. For this node K choose any $\rho \in [r, R]$ and set the interval of synchronization to $[\rho, \rho]$.

4.7 Example – Continued

The following graph results from the example of section 4.1. Thick lines have a weight of 1, thin lines indicate weights of 0. To simplify the presentation, expression trees of `i+1` and `i-1` have been condensed into one node each.



Directly attached to the nodes is the interval of possible synchronization ranks as computed after the first phase of the restructuring algorithm; the updates due to the freezing phase are given after small arrows. Note, that the intra-dependence edge which represents the data dependence `B`$_1$`[i]` δ^a `B`$_2$`[i]` has weight 0 since it does not require a synchronization barrier.

From this graph, the best/last code as given in section 4.1 is generated. Although there are only anti dependences in the example, the algorithm works for output and flow dependences as well.

5 Performance Results

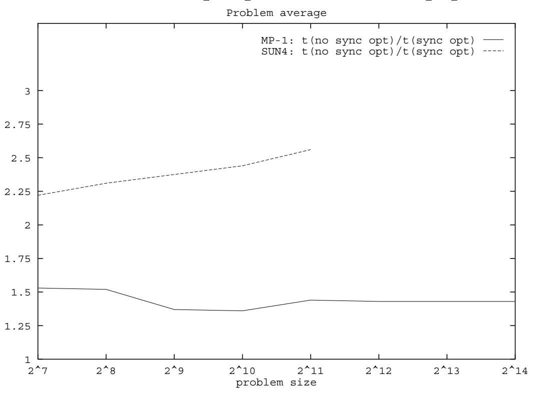
At the moment, our benchmark suite consists of 17 problems collected from literature [15]. Here, we only consider those seven problems whose Modula-2* solutions involve synchronous `FORALL` statements.

Using the PowerTest [20] to check subscripts during dependence analysis, the programs were compiled for a 16K processor MasPar MP-1 (SIMD) and a sequential SUN SparcStation-1 (SISD) by our Modula-2* compiler [9, 14]. Automatic application of the synchronization elimination scheme improved the execution times of the programs by over 40% for the MasPar and by over 100% for the Sparc on average. Note that even on an inherently synchronous, parallel SIMD machine the elimination of synchronization barriers clearly pays off due to the necessity of virtualization in the cases where problem size exceeds machine size. Because our work on Modula-2* compilers for MIMD machines, namely LANs of workstations and (virtual) shared memory multiprocessors, is still in progress, we cannot present any measurements therefor. But we expect even better results since each synchronization barrier causes high-latency delays on such machines.

For time measurements we used the high resolution DPU timer on the MasPar and the UNIX `clock` function on the Sparc (sum of user and system time). Below, $t_{sync-opt}$ and $t_{no-sync-opt}$ represent program execution times on either a 16K MasPar MP-1 or a SparcStation-1 (as appropriate) with the optimization techniques presented in the paper applied and not applied, respectively.

We define performance as work or problem size per time and focus on the following relative performances:[7] $\frac{size}{t_{sync-opt}} / \frac{size}{t_{no-sync-opt}} = t_{no-sync-opt}/t_{sync-opt}$. Thus, the diagrams show a ratio scale as the vertical axis. Good performance of the synchronization elimination technique is indicated by curves above unity, e.g. a curve around 2 shows that the elimination of redundant synchronization barriers halved the execution time.

For problem sizes ranging from 2^3 to 2^{21} we derived the relative performances from our execution time measurements. The resulting general, relative performances, averaged arithmetically over all test programs per problem size, are shown below. (Only results with at least three measurement points per problem size are included in this average graph.) Originally, the programs had 278 synchronization barriers with only 109 remaining after application of the elimination technique presented in the paper.

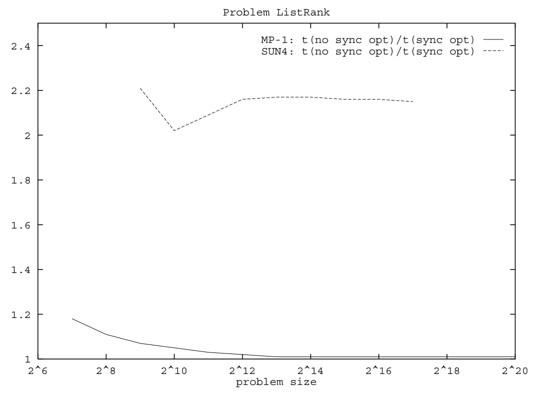


On sequential machines the asynchronous FORALLs are implemented as `for` loops. The number of synchronization barriers is equal to the number of loops. Hence, the performance gain is mainly due to reduced loop overhead and better register usage inside of larger loops.

5.1 Problems

5.1.1 List Rank

Problem: A linked list of n elements is given in an array $A[1..n]$. Compute for each element its rank in the list. **Approach:** This problem is solved by pointer jumping. **Note:** Ranking the elements of a list is one of the elementary list processing tasks [11]. **Comment:** This problem heavily relies on the general communication mechanism of the MasPar programming language (mpl). Since the cost of communication dominates the total work, the elimination of synchronization barriers can only be effective when just a few packets are sent, which is the case for smaller problem sizes.



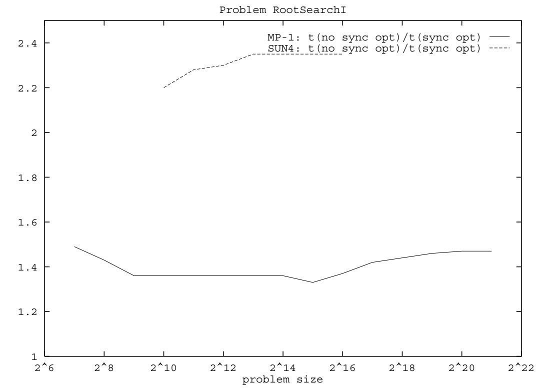
Reduction of synchronization barriers: 40 $\longrightarrow$ 16

5.1.2 Root Search

Problem: Determine the value of $x \in [a, b]$ such that $f(x) = 0$, given that f is monotone and continuously differentiable. **Approach:** The problem is solved with multisection. The interval $[a, b]$ is equally divided over n processes. If f has a root in $[a, b]$ then there is exactly one process p with $f(x_{p-1}) \cdot f(x_p) < 0$. Update the interval

[7]Comparisons with hand-coded programs are given in [15].

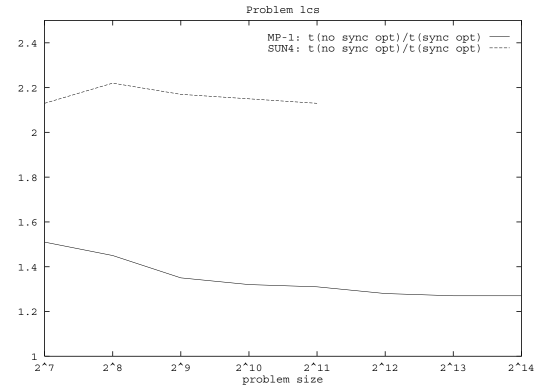
$[a', b'] := [x_{p-1}, x_p]$. Iterate until the error $b' - a' < \epsilon$. **Note:** This problem occurs in science and engineering [1]. **Comment:** The solution requires access to neighboring data elements. Currently this is implemented on the MasPar with global communication primitives. Since relative overhead of the work incurred by unnecessary virtualization loops will increase when faster grid communication can be used instead, we expect better results in future.



Reduction of synchronization barriers: 20 $\longrightarrow$ 7

5.1.3 Longest Common Subsequence

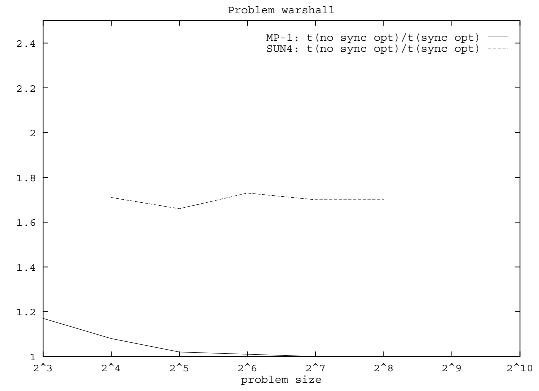
Problem: Two strings $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ are given. Find a string $C = c_1 c_2 \cdots c_p$ such that C is a longest common subsequence of A and B. (C is a subsequence of A if it can be constructed by removing elements from A without changing their order.) **Approach:** The solution uses a wave-front implementation of dynamic programming. It causes intensive access to neighboring data elements. **Note:** The parallel solution is based on [5]. **Comment:** see 5.1.2.



Reduction of synchronization barriers: 62 $\longrightarrow$ 26

5.1.4 Transitive Closure

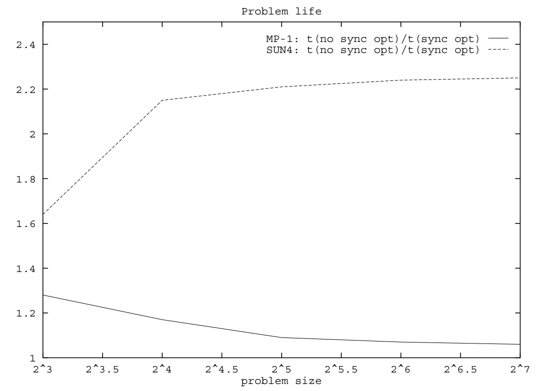
Problem: The adjacency matrix of a directed graph with n nodes is given. Find its transitive closure. **Approach:** Process the adjacency matrix according to the property that if nodes x and m as well as nodes m and y are (transitively) adjacent, then x and y are (transitively) adjacent. **Note:** The problem was suggested by Hatcher [10]. **Comment:** see 5.1.1.



Reduction of synchronization barriers: 12 $\longrightarrow$ 8

5.1.5 Game of Life

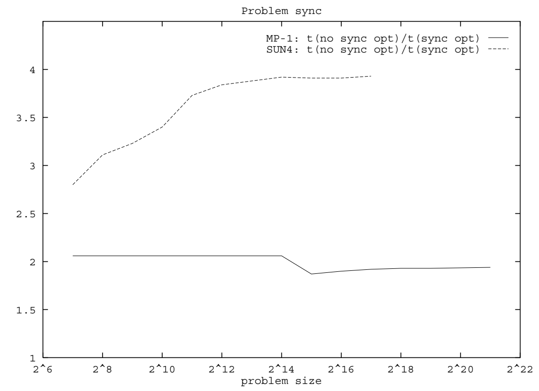
Problem: Apply Conway's rules of life to a given matrix. **Approach:** The value of a grid point depends on the sum of the values of its neighbors. **Comment:** see 5.1.2.



Reduction of synchronization barriers: 66 $\longrightarrow$ 30

5.1.6 Synchronous Example

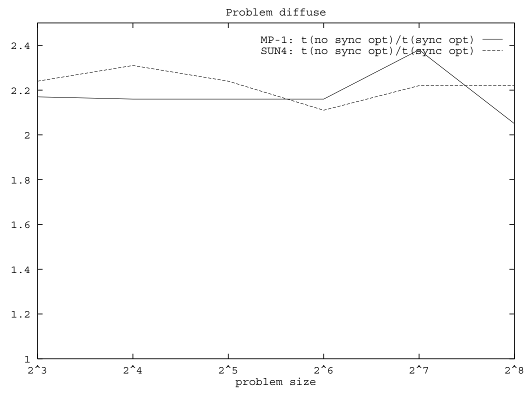
Problem: This is the example of a Modula-2* program containing one synchronous FORALL with a high potential of synchronization barriers that can be eliminated. **Note:** The graph has a differently scaled vertical axis.



Reduction of synchronization barriers: 44 $\longrightarrow$ 6

5.1.7 Heat Diffusion Kernel

Problem: The temperature on the edges of a square surface are given as constants, while those on the inside are to be calculated with a diffusion equation. **Approach:** The value of a grid point is iteratively computed based on the values of its neighbors.



Reduction of synchronization barriers: 34 $\longrightarrow$ 16

6 Conclusion

The encouraging performance results of the experiments assess the practical usefulness of our automatic elimination technique. Thus, we directly contribute to the field of compiler construction for parallel computers by showing how to handle synchronously parallel language constructs, e.g. synchronous FORALLs, Fortran90 vector operations, HPF FORALLs, and UC par statements, in order to generate efficient code for them.

Furthermore, Chatterjee's performance results and ours taken together provide good evidence that synchronously parallel language constructs can be compiled to efficient code for at least three different kinds of machine architectures: shared memory MIMD, SIMD, and SISD. Hence, we do not see any necessity to exclude synchronous parallelism from high-performance programming languages.

As for future work, we hope to be able to improve the overall optimization results by completely integrating processor virtualization, synchronization elimination, and data prefetching. Moreover, Allen's and Kennedy's work on vector register allocation [4] contains ideas on how to reduce temporary storage consumption that seem to apply and fit well into our framework, too.

The IPD Modula-2* system is freely available by anonymous ftp from ftp.ira.uka.de in pub/programming/modula2star.

References

[1] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[2] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proc. of ACM POPL'87*, pages 63–76, Munich, January 1987, ACM Press.

[3] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM TOPLAS*, 9(4):491–592, April 1987, ACM Press.

[4] R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

[5] A. Apostoli, M.J. Atallah, L.L. Larmore, and S. McFaddin. Efficient parallel algorithms for string editing and related problems. CSD-TR-724, Purdue University, Dept. of Computer Science, May 1990.

[6] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[7] S. Chatterjee. Compiling nested data-parallel programs for shared memory multiprocessors. *ACM TOPLAS*, 15(3):400–462, March 1993.

[8] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimizations. *ACM TOPLAS*, 9(3):319–349, March 1987.

[9] S. Hänßgen, E.A. Heinz, P. Lukowicz, M. Philippsen, and W.F. Tichy. The Modula-2* environment for parallel programming. In *Proc. of the Conf. on Programming Models for Massively Parallel Computers*, pages 43–52, Berlin, September 1993, IEEE Society Press.

[10] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1991.

[11] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[12] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proc. of ACM PLDI'93*, pages 78–89, June 1993, ACM Press.

[13] S.P. Midkiff and D.A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, December 1987.

[14] M. Philippsen. *Optimierungstechniken zur Übersetzung paralleler Programmiersprachen.* Ph.D. Thesis, University of Karlsruhe, Dept. of Informatics, 1993.

[15] M. Philippsen, E.A. Heinz, and P. Lukowicz. Compiling machine-independent parallel programs. *ACM SIGPLAN Notices*, 28(8):99–108, August 1993.

[16] M. Philippsen and W.F. Tichy. Modula-2* and its compilation. In *Proc. of the First Intl. Conf. of the Austrian Center for Parallel Computation*, pages 169–183, October 1991, LNCS 591, Springer Verlag.

[17] S. Prakash, M. Dhagat, and R. Bagrodia. Synchronization issues in data-parallel languages. In *Proc. of the 6th Intl. Workshop on Languages and Compilers for Parallel Computing*, pages 76–95, August 1993, LNCS 768, Springer Verlag.

[18] W.F. Tichy and C.G. Herter. Modula-2*: An extension of Modula-2 for highly parallel, portable programs. TR-4/90, University of Karlsruhe, Dept. of Informatics, January 1990.

[19] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, 1989.

[20] M. Wolfe and C.-W. Tseng. The power test for data dependence. TR-CSE-90-015, Oregon Graduate Institute, August 1990.