Technical Report No. 13/93, Department of Informatics,
University of Karlsruhe, F.R. Germany, April 1993.

Synchronization Barrier Elimination in Synchronous FORALLs

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

Abstract

This paper presents and evaluates a restructuring optimization technique for the compilation of synchronous FORALLs. Because of its elegant machine-independence we choose Modula-2*, a problem-oriented explicitly parallel programming language, as our conceptual framework.

The presented optimization technique has improved runtime performance on Modula-2* benchmarks by over 40% on a MasPar MP-1 and by over 100% on sequential workstations.

We present our approach in terms of a source-to-source transformation. The compiler analyzes programs that contain synchronous FORALLs and, hence, vast numbers of synchronization barriers. These are transformed into semantically equivalent programs that only contain asynchronous FORALLs, thus greatly simplifying the task of mapping them onto any parallel machine. The transformed programs run significantly faster since redundant synchronization barriers are eliminated.

1 Introduction

Most parallel programmers consider the data parallel paradigm to be easier than programming with asynchronous control flow parallelism. The data-parallel approach has two major advantages: (1) It is a natural extension of sequential programming. The only parallel instruction, a synchronous FORALL statement, is a simple extension of the well known FOR statement and is easy to understand. (2) Debugging data-parallel programs is not much more difficult than debugging sequential programs. The reason is that there is only a single locus of control, which dramatically simplifies the state space of a program compared to that of a MIMD program with thousands of independent loci of control.

Understanding these advantages, designers of parallel programming languages included various types of a highly desirable synchronous FORALL statements into their languages. There are for example synchronous FORALLs in several parallel Fortran dialects, e.g. [2, 9, 15].

Although the synchronous FORALL is an appropriate means of abstraction from the programmer's point of view it induces several hard problems for compiler writers.

Many of the proposed parallel programming languages offering synchronous FORALLs have been developed with the main goals of high performance and feasibility of early compilers. Consequently, the originally high expressiveness of the data-parallel synchronous FORALL has been restricted dramatically. This is especially true for Fortran dialects. In the body of some proposed versions of synchronous FORALLs, for example, there may be only a single elementwise assignment; some other versions have copy-in-copy-out semantics or do not allow control flow statements in their bodies.

In this paper we deal with a synchronous FORALL statement that is orthogonally integrated into the language and suffers from no restrictions in its expressive power. We present an optimization technique to analyze the synchronous FORALLs and replace them by semantically equivalent asynchronous parallelism with often a minimal number of synchronization barriers. This technique has been implemented as part of a Modula-2* compiler. We evaluate the effectiveness of our technique by presenting some performance measurements comparing the runtime of translations with and without optimization.

2 Synchronous FORALLs

When speaking of *synchronous FORALLs* we mean high-level language constructs that allow for problem-oriented expression of synchronous parallelism.

2.1 Syntax and Semantics

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 and loops inside synchronous FORALLs is defined with MSIMD (multiple SIMD) execution in mind. Therefore, Modula-2* does not prescribe any synchronization between different branches of synchronous `CASE` or `IF` statements. The exact synchronous semantics of nested Modula-2* statements are defined in [23].

2.2 Compilation

Synchronous FORALLs cannot be executed directly on MIMD and sequential machines because of their special semantics. The same holds for SIMD machines as soon as you consider the necessity of automatic process virtualization (see [18, 14] for details). Therefore, we translate synchronous FORALLs into equivalent sequences of asynchronous FORALLs by applying the machine-independent source-to-source transformations of [13, 17].

Asynchronous FORALLs can directly be executed on any type of machine. Except for a synchronization barrier at the end they do not impose any synchronization constraints on the created processes. In Modula-2* the syntax of *asynchronous* FORALLs is defined as follows.

```
FORALL ident ":" SimpleType IN PARALLEL
  StatementSequence
END
```

The asynchronous semantics permit that the processes execute in any order, even strictly sequential one after another. Hence, asynchronous FORALLs may safely be compiled into sequential loops on every processor.

To illustrate the translation of synchronous FORALLs we proceed with an example that shows the source-to-source transformation in action. Note how the implicit synchronization barriers and hidden temporary variables of the synchronous FORALL are made explicitly visible by the transformation.

```
FORALL i : [1..N] IN SYNC
  A[i] := A[i+1];
  B[i] := B[i+1];
END

          ⇓

FORALL i : [1..N] IN PARALLEL
  H1[i] := A[i+1];
END;
FORALL i : [1..N] IN PARALLEL
  A[i] := H1[i];
END;
FORALL i : [1..N] IN PARALLEL
  H2[i] := B[i+1];
END;
FORALL i : [1..N] IN PARALLEL
  B[i] := H2[i];
END
```

First, the rhs `A[i+1]` of the assignment to `A[i]` is evaluated and temporarily stored in `H1[i]` for all i∈`[1..N]`. Second, the prescribed synchronization barrier between the rhs and the lhs of the original assignment is implemented by closing the asynchronous `FORALL` statement. Then the original lhs `A[i]` is written by assignment of the temporary `H1[i]` for all i∈`[1..N]`. Thereafter, the asynchronous FORALL is closed in order to implement the prescribed synchronization barrier between the two original assignments. The next two asynchronous `FORALL` statements perform the synchronous evaluation of the second original assignment in a similar way.

3 Synchronization Barrier Elimination

To be on the safe side a non-optimizing transformation of synchronous FORALLs generates vast numbers of synchronization barriers that lead to poor program performance. In this section we present an optimization technique that significantly reduces the overall synchronization costs of the transformed results. This is achieved by automatic elimination of redundant synchronization barriers. The optimization relies on fine-grain *dependence graph evaluation* and a sophisticated *program restructuring* method.

3.1 Example

We return to the sample transformation of section 2.2 to illustrate the central ideas of our elimination technique.

Closer examination reveals that two synchronization barriers of the transformed result are redundant. You only have to change the order of statement execution to eliminate them.

```
FORALL i : [1..N] IN SYNC
  A[i] := A[i+1];
  B[i] := B[i+1];
END
```

$$\Downarrow$$

```
FORALL i : [1..N] IN PARALLEL
  H1[i] := A[i+1];
  H2[i] := B[i+1];
END;
FORALL i : [1..N] IN PARALLEL
  A[i] := H1[i];
  B[i] := H2[i];
END
```

Of course, such restructurings must preserve the original program semantics. Assuming that `A[1:(N+1)]` and `B[1:(N+1)]` are disjunct, no data dependences exist between the first and the second assignment inside the original synchronous `FORALL` statement. Hence, these assignments may be executed in any order.

However, there do exist data dependences inside the assignments themselves, namely between `A[i+1]` and `A[i]` as well as between `B[i+1]` and `B[i]`. Therefore, the synchronization barrier after rhs evaluation cannot be eliminated, and the rhs values of both assignments must be stored in temporary variables.

As for synchronization barriers, the above restructuring transformation is *optimal*. In section 3.7 we show that our restructuring elimination technique always achieves optimality when being applied to synchronous FORALLs with an arbitrary flat statement sequence inside.

3.2 Program Restructuring

In contrast to other restructuring methods, e.g. for parallelizing Fortran compilers in [3, 7, 26, 27] to name but a few, we use (sub-) *expressions* as the basic units of program decomposition. Consequently, our restructuring method is of extremely *fine-grain* nature. Such fine-grain restructuring enables otherwise impossible eliminations of synchronization barriers that separate different parts of the same statement (intra-statement synchronization barriers). The automization of our fine-grain restructuring method requires additional information specifying the

- resulting execution order,
- necessity of temporary storage, and
- non-redundant synchronization barriers.

Therefore, we decorate statements and (sub-) expressions with two more attributes, a boolean *sync flag* and an integer *sync rank*.

Sync Rank. The absolute sync rank value determines the resulting execution order in ascending fashion. Statements and (sub-) expressions featuring equal absolute sync ranks may be executed/evaluated in any order. Runtime constants, e.g. FORALL variables, get zero sync ranks by definition. The sign of a sync rank specifies the necessity of temporary storage where positive means "yes" and negative means "no".

Sync Flag. If sync flags are set they characterize necessary synchronization barriers. Non-redundant synchronization barriers appear after each sequence of (sub-) expressions and statements with equal absolute sync ranks if one or more thereof have their sync flags set.

Below we present an algorithm for our fine-grain restructuring method that is based on sync rank and sync flag information.

Restructuring Algorithm

Definitions:
$E \in S$ denotes (sub-) expression E of statement S
$Rank()$ returns the sync rank of its argument
$Sync()$ returns **true** iff sync flag of argument is set

Input:
$(S_1, \ldots, S_k)$ = statement sequence decorated with sync ranks and sync flags

Output:
$(S'_1, \ldots, S'_l)$ = correct restructuring of $(S_1, \ldots, S_k)$ according to sync ranks and sync flags of each S_j

I. **if** $\forall j \in \{1..k\} \; \forall E \in S_j : \neg Sync(E)$ **then**
  "generate $(S_1, \ldots, S_k)$"
  stop
end if

II. $n := 1$
$m := \max\{\, |Rank(S_j)| \, : \, j \in \{1..k\} \,\}$

III. **while** $n \leq m$ **do**
1. $Flag :=$ **false**
2. **for all** $j \in \{1..k\}$ **do**
   if $|Rank(S_j)| = n$ **then**
   A. "generate the statement corresponding to S_j with correct substitutions for all pre-evaluated subexpressions"

```
            B. if Sync(S_j) then
                  Flag := true
               end if else
               for all E ∈ S_j : (Rank(E) = n) do
                  a. "generate statements for the
                      evaluation and temporary
                      storage of expression E"
                  b. if Sync(E) then
                        Flag := true
                     end if
               end for all
            end if
      end for all
   3. if Flag then
         "generate statements for
          synchronization barrier"
      end if
   4. n := n + 1
end while
```

Step (I) speeds up the algorithm for input sequences without any sync flags set. Step (II) initializes sync rank counter n and sync rank sentinel m. Step (III) repeatedly iterates over the whole input sequence and, generates all output statements of absolute sync rank n during each iteration. Step (III.3) checks for necessary synchronization barriers at the end of an iteration whereas step (III.4) increments the sync rank counter.

In the sequel we illustrate how the restructuring algorithm works. We study the application of our algorithm to the example of section 3.1. Both statements inside the synchronous FORALL serve as input to the algorithm. The correctly decorated input sequence is given below, with numbers representing sync ranks and black dots showing sync flags which are set. Note that all runtime constants get zero sync ranks.

```
FORALL i :  [1..N] IN SYNC
  -2 0    -2  1 000
   A[i]  :=  A[i+1];
             ●
  -2 0    -2  1 000
   B[i]  :=  B[i+1]
             ●
END
```

Let us now execute the restructuring algorithm for this input sequence by hand. Since there are some sync flags set in the given code fragment the **if** expression of step (I) evaluates to **false** and the conditional branch is not taken. Step (II) initializes $n := 1$ and calculates 2 as the value of m because the maximal absolute sync rank in the code fragment is |-2|. Therefore, step (III) exactly iterates twice over the input sequence.

Iteration 1.
Step (III.1) initializes *Flag* to **false**. Then, for j=1, step (III.2.a) generates `H1[i]:=A[i+1]` and for j=2 it generates `H2[i]:=B[i+1]`. These assignments evaluate and temporarily store the expressions `A[i+1]` and `B[i+1]` which have positive sync ranks equal to n. Furthermore, for $j=1,2$ the conditional branch of step (III.2.b) is taken setting *Flag* to **true**. Hence, step (III.3) closes the asynchronous FORALL around the assignments, the `END` thereof serving as the necessary synchronization barrier. Finally, the sync rank counter n is incremented, yielding 2 as its new value.

Iteration 2.
Step (III.1) initializes *Flag* to **false**. Then, for j=1, step (III.2.A) generates `A[i]:=H1[i]` and for j=2 it generates `B[i]:=H2[i]`. These assignments correspond to the two input statements with `H1[i]` and `H2[i]` substituted for the pre-evaluated expressions `A[i+1]` and `B[i+1]`. The *Flag* remains unset because the conditional branch of step (III.2.B) is neither taken for j=1 nor for j=2. Hence, step (III.3) does not generate any synchronization barrier statements. Finally, the sync rank counter n is incremented yielding 3 as its new value.

After loop termination, we close the last asynchronous FORALL and output the restructured result.

```
FORALL i : [1..N] IN PARALLEL
  H1[i] := A[i+1];              | generated by
  H2[i] := B[i+1];              | iteration 1
END;
FORALL i : [1..N] IN PARALLEL
  A[i] := H1[i];                | generated by
  B[i] := H2[i];                | iteration 2
END
```

3.3 Dependence Graph Evaluation

Automatic calculation of sync ranks and sync flags requires *three phases* of dependence graph evaluation. We use dependence graphs containing control dependences as well as data dependences of kind anti, intra, flow, and output. Input dependences are not considered. Intra dependences are data dependences with all equal direction vectors $(=,=,\ldots,=)$ only. In [14] we give an exact definition of our dependence graphs and thoroughly discuss dependence analysis for synchronous FORALLs. Standard concepts of dependence analysis are covered by numerous authors (see [5, 20, 26] for more).

The first phase of our evaluation scheme initializes the sync ranks and sync flags of all statements including their (sub-) expressions. According to the following rules the initialization treats dependence sources in a special way.

1. Set sync ranks of all (sub-) expressions to 0 and set their sync flags to **false**.

2. Set sync ranks of all non-intra anti dependence sources to $+1$.
3. Set sync ranks of all remaining data dependence sources to -1.
4. Set sync flags of all non-intra data dependence sources to **true**.

Rules (2) and (4) intentionally exclude intra dependences because these do not impose any synchronization constraints on their sources and sinks. They only affect the sequential order of statement execution in single processes. Therefore, the above four rules reduce to a single one when applied to intra dependences.

- Set sync ranks of intra dependence sources to -1 and set their sync flags to **false**.

To illustrate the initialization phase of dependence graph evaluation we employ our standard example. Below we show a snapshot thereof just after initialization. The arrows visualize data dependences. Control dependences are intentionally omitted because the two anti dependences dominate the rest.

```
FORALL i :  [1..N] IN SYNC
    0          1
     0    0      000
    A[i]   :=   A[i+1];
    0          1
     0    0      000
    B[i]   :=   B[i+1]
END
```

The second phase applies some kind of *topological sorting* by absolute sync ranks to the dependence graph. Thereafter, the sync ranks of statements and (sub-) expressions that are involved in data dependences specify the least constraining order of their correct evaluation. Topological sorting is possible because our synchronous dependence analysis ensures *acyclic* dependence graphs as shown in [14]. In case of synchronous loops we avoid dependence cycles by automatic loop unrolling and condition restructuring.

We must perform topological sorting sweeps for each root of the dependence graph because it need neither be closed nor connected. Let s denote the sync rank of a dependence source and d that of its associated sink. Then the following algorithm performs the sync rank modifications of our topological sorting.

if $d = 0$ **then**
  $d := -(|s| + 1)$
else if $|d| \leq |s|$ **then**
  $d := (|s| + 1) * sign(d)$
end if

This procedure always changes zero sync ranks of dependence sinks to $-(|s| + 1)$. For those with non-zero sync ranks, however, it increments the absolute values only while leaving their signs intact.

Applied to our example topological sorting changes the sync ranks of `A[i]` and `B[i]`. The snapshot below shows the synchronous `FORALL` statement right after topological sorting.

```
FORALL i :  [1..N] IN SYNC
   [-2]        1
     0    0      000
    A[i]   :=   A[i+1];
   [-2]        1
     0    0      000
    B[i]   :=   B[i+1]
END
```

Finally, the third phase recursively propagates sync ranks and sync flags of sub-expressions to their enclosing (sub-) expressions and statements. The propagation combines the sync ranks of the sub-expressions with regard to maximum absolute value. In case of absolute equality the combination prefers positive sync ranks over negative ones. Furthermore, the sync rank r of a whole statements is always set to the negative value $-|r|$ after sync rank combination. The result of sync rank combination guides the propagation of sync flags. The set sync flag of a subexpression E is propagated iff the absolute sync rank of E equals the absolute propagated sync rank.

As for our example, the propagation phase only affects the two assignments as whole statements. Their final sync ranks are calculated, thus yielding the completely decorated synchronous `FORALL` statement that we already know from Section 3.2.

```
FORALL i :  [1..N] IN SYNC
   -2 0    [-2]  1 000
    A[i]   :=   A[i+1];
   -2 0    [-2]  1 000
    B[i]   :=   B[i+1]
END
```

3.4 Control Dependences

In the previous sections we intentionally omitted control dependences from our considerations for the sake of simplicity. Now we show how to handle them because they are necessary for the correctness of our elimination technique.

In the following we subsume traditional operators, procedure calls with k arguments, and array designators with k subscripts under the notion of a k-ary operator. Obviously, such an operator cannot be evaluated *before* any of its k arguments. Hence, the absolute sync rank of a k-ary operator must at least equal the absolute maximum of the sync ranks of its k arguments. We ensure this sync rank constellation by including control dependences between arguments and operators in the dependence graph. During the topological sorting phase of dependence graph evaluation control dependences simply serve as absolute sync rank promoters. If the absolute sync rank $|s|$ of a control dependence source is greater than the absolute sync rank $|d|$ of its corresponding sink then the sync rank of the sink is set to $|s| * sign(d)$ by the topological sorting sweep. For examples and more details of control dependence handling we refer the interested reader to [14].

3.5 Correctness

In this section we prove the correctness of our restructuring elimination technique. We show the equivalence of original programs and restructured results with regard to operational semantics. As a natural pre-condition of our argumentation we assume that dependence analysis and code generation are correct. Therefore, semantic differences between original and transformed programs can only be due to *wrong execution order* or *missing synchronization barriers* in the restructured results. Below we prove that either is impossible. Thereby it follows that our restructuring elimination technique is correct.

Proof by Contradiction.

Execution Order. Suppose that in the restructured program a data element M is overwritten before its intended use. Then the original dependence graph contains a path from M's use to the overwriting access. Hence, the overwriting access is a direct or indirect successor of M's use in the dependence graph. Topological sorting and sync rank propagation ensure that, after dependence graph evaluation, the absolute sync rank of the overwriting access is greater than that of M's use.[1] Therefore, step (III) of our restructuring algorithm generates all statements of the overwriting access *after* those of M's use. From this follows that M's use *precedes* the overwriting access in the restructured program → Contradiction to initial assumption.

Synchronization. Suppose that in the restructured program a necessary synchronization barrier is missing. Necessary synchronization barriers prevent race conditions of overlapping read and write accesses in different processes. These synchronization barriers must always be placed between source and sink of a non-intra dependence from the original dependence graph. Therefore, the absolute sync ranks of the conflicting accesses will be different after dependence graph evaluation because of topological sortinging. The source of the mentioned dependence has its sync flag set because it is not an intra dependence. The restructuring algorithm recognizes this sync flag either in step (III.2.B) or in step (III.2.b) and generates the statements for a synchronization barrier in step (III.3) of the same loop iteration. As the absolute sync rank of the dependence sink is *greater* than that of the source the generated synchronization barrier lies exactly *between* the conflicting accesses → Contradiction to initial assumption.

[1]Here we need the control dependences of Section 3.4. This argument does not hold without them.

3.6 Nested Statements

In [14] we present detailed restructuring rules and examples for the complete set of Modula-2* compound statements including nested branches, loops, `FORALL`s, and procedure calls. Because of space limitations we can only briefly explain the fundamental ideas thereof in this paper.

All in all we *abstract* from nestings and restructure each flat statement sequence separately. We do so by *normalizing* dependences that cross nesting level boundaries to their nearest enclosing statement sequence. Essentially, this results in a *partitioning* of the dependence graph into a set of disjunct subgraphs, namely one for each flat statement sequence. The normalized dependences ensure the correct placement of compound statements with respect to their surrounding statements. The initialization rules for normalized dependences are the same as those for intra dependences stated in Section 3.3. As for control dependences, we treat compound statements like operators.

To maximize the chances of synchronization barrier elimination we unroll loops and pre-evaluate conditions of branches and loops as well as copy-in values of `VAR`-parameters (see [14] for details).

3.7 Optimality

In this section we analyze the theoretical capabilities of our restructuring elimination technique. Practical performance results follow in Section 4.

The automatic elimination of synchronization barriers trades space (additional temporary variables) for time (reduced synchronization costs). Although this may not be the right preference on special machines we believe that in the long run computation time but not memory consumption will be the limiting factor of parallel processing. Hence, we do not discuss the preference aspect of our elimination technique any further.

The question we are interested in runs "Which redundant synchronization barriers (if any) are *not* eliminated by our restructuring elimination technique?".

It is clear that every data dependence prescribes an evaluation order of source before sink. An equally obvious fact holds for synchronization constraints. Sources and sinks of non-intra data dependences need a necessary synchronization barrier between them because of otherwise conflicting data access. These are the minimal requirements of correct program execution.[2]

[2]According to the proof in Section 3.5 our elimination technique fulfills them.

Therefore, you can only reduce the number of synchronization barriers if you simultaneously fulfill the synchronization constraints of many data dependences by a barrier. Thus, the task of synchronization barrier elimination effectively reduces to a graph coverage problem. "Cover as many data dependences as possible by as few synchronization barriers as necessary". In the following we discuss our technique with regard to this task.

Flat Statement Sequences. Every statement and expression of a flat statement sequence is exactly evaluated once. Hence, an execution order with a *minimal* number of synchronization barriers covers all those data dependences by a single barrier whose sources are independent in the transitive hull of the dependence relation. Our restructuring elimination always produces such an *optimal* execution order because the topological sorting of the dependence graph embeds the partial ordering given by the transitive hull of the dependence relation in a linear ordering. Then the sync rank propagation ensures that all statements and (sub-) expressions which may be executed in any order receive equal absolute sync ranks. Finally, the restructuring algorithm of Section 3.2 covers the data dependences of all statements and (sub-) expressions with equal absolute sync ranks by a single synchronization barrier.

Nested Statement Sequences. Because of dependence normalization and dependence graph partitioning our elimination technique achieves optimal results for every flat statement sequence of the nest. For the nest as a whole the result need not necessarily be optimal as we do not optimize the combination of nestings. This is a theoretical disadvantage of our nesting abstraction. The measurements of Section 4, however, support our argument that ascribes only marginal practical relevance to such optimizations.

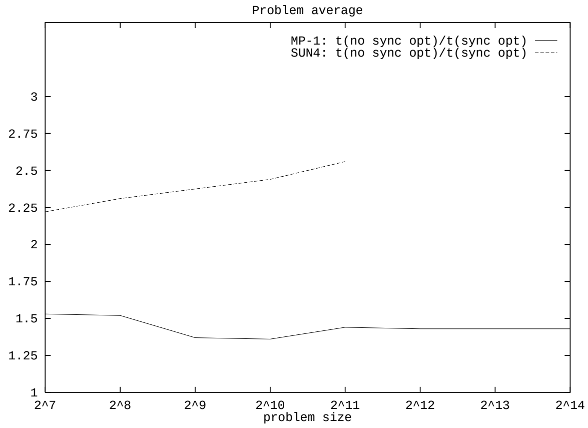
4 Performance Results

At the moment, our benchmark suite consists of 17 problems collected from literature [1, 10, 16, 11, 8]. In the implementation of seven problems we could take advantage of the synchronous `FORALL` statement of Modula-2*.

The programs have then been compiled by our Modula-2* compiler [24, 19] for a 16k MasPar MP-1 (SIMD) and for a sequential SparcStation-1 (SISD) for widely ranging problem sizes. The result of synchronization barrier elimination is an improvement of runtime performance by over 40% on the MasPar and by over a factor of 2 on the sequential workstation. Note that even on a synchronous parallel machine the runtime can be improved dramatically by the elimination of synchronization barriers. Although we cannot present performance results for MIMD machines, since work on Modula-2* compilers targeting KSR-1 and a LAN of workstation is still in progress, we expect even better results because on these machines each synchronization barrier must be implemented with high-latency message passing constructs.

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

For time measurement we used the high-resolution DPU timer on the MasPar and the UNIX `clock` function on the SparcStation (sum of user and system time). Below, $t_{no-sync-opt}$ represents the Modula-2* runtime on either a 16K MasPar MP-1 or a SparcStation-1 (as appropriate) when the optimization technique presented in this paper has not been used. Analogously, $t_{sync-opt}$ stands for the Modula-2* runtime with the elimination technique employed.

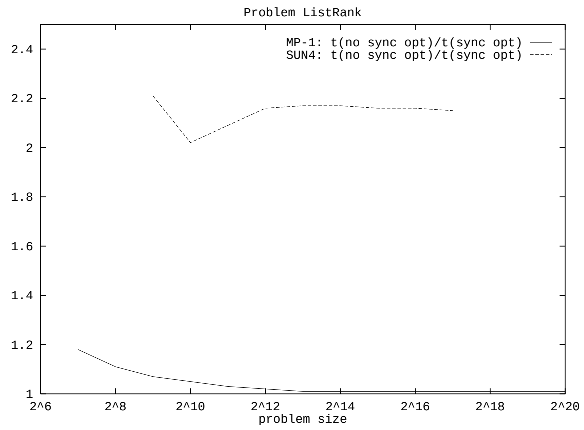


For problem sizes ranging from 2^3 to 2^{21} we derived the relative performances from our runtime measurements. The resulting general relative performance, averaged arithmetically over all test programs per problem size, is shown above. (Only results with at least three measurement points per problem size are shown in the average graph.) Originally, the programs had 278 synchronization barriers which have been reduced to 109 by applying the technique presented in this paper.

4.1 Problems

4.1.1 List Rank

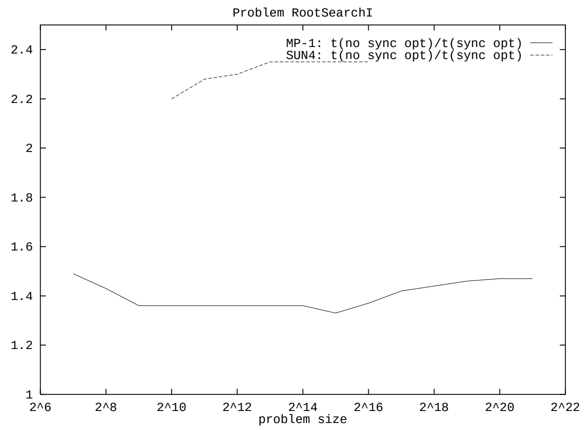
Problem: A linked list of n elements is given. All elements are stored in an array $A[1..n]$. Compute for each element its rank in the list.
Approach: This problem is solved by pointer jumping.
Note: Ranking the elements of a list is one of the elementary list processing tasks [16].



Reduction of synchronization barriers: 40 $\longrightarrow$ 16
Comment: This problem heavily relies on the general communication mechanism of the MasPar programming language (mpl). The cost of communication dominates the total work. Therefore, the elimination of synchronization barriers can only be effective when just a few packets are sent, which is the case for smaller problem sizes.

4.1.2 Root Search

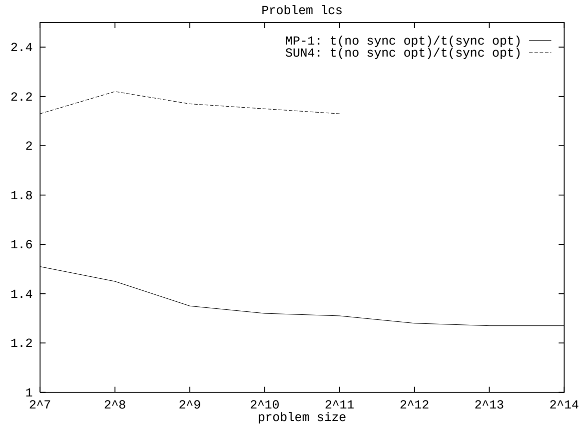
Problem: Determine the value of $x \in [a, b]$ such that $f(x) = 0$, given that f is monotone and continuously differentiable.
Approach: The problem is solved with multisection. The interval $[a, b]$ is equally divided over n processes. If f has a root in $[a, b]$ then there is exactly one process p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



Reduction of synchronization barriers: 20 $\longrightarrow$ 7
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.

4.1.3 Longest Common Subsequence

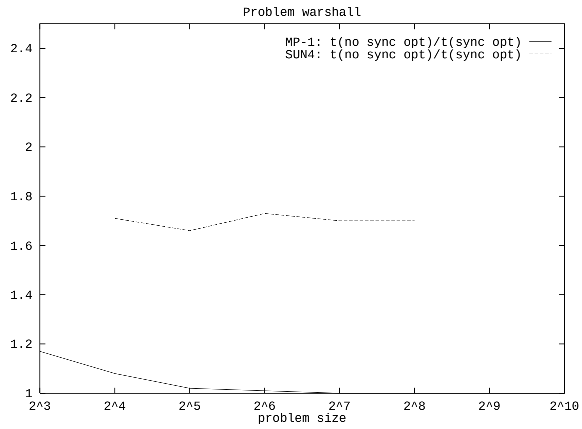
Problem: Two strings $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ are given. Find a string $C = c_1 c_2 \cdots c_p$ such that C is a longest common subsequence of A and B. (C is a subsequence of A if it can be constructed by removing elements from A without changing their order. A common subsequence must be constructible from both A and B.)
Approach: The solution uses a wave-front implementation of dynamic programming. It causes intensive access to neighboring data elements.
Note: The problem is presented in detail in [21]. The parallel solution is based on [4].



Reduction of synchronization barriers: 62 $\longrightarrow$ 26
Comment: see RootSearch.

4.1.4 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. The algorithm is due to Warshall [25].
Note: The problem was suggested by Hatcher [11].

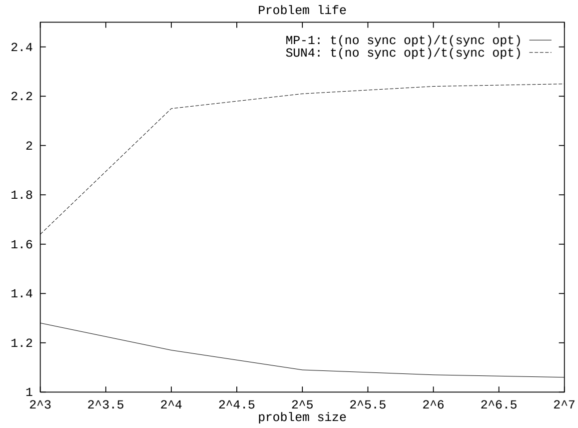


Reduction of synchronization barriers: 12 $\longrightarrow$ 8

Comment: see ListRank.

4.1.5 Game of Life

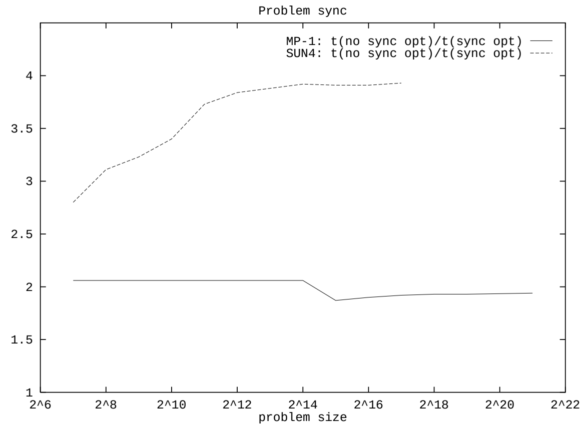
Problem: Apply John Conway's rules of life to a given matrix.
Approach: In a straightforward implementation the value of a grid point depends on the sum of the values of its neighboring grid points.
Note: The implementation can make easy use of the synchronous FORALL since no intermediate values must be stored manually.



Reduction of synchronization barriers: 66 ⟶ 30
Comment: see RootSearch.

4.1.6 Synchronous Example

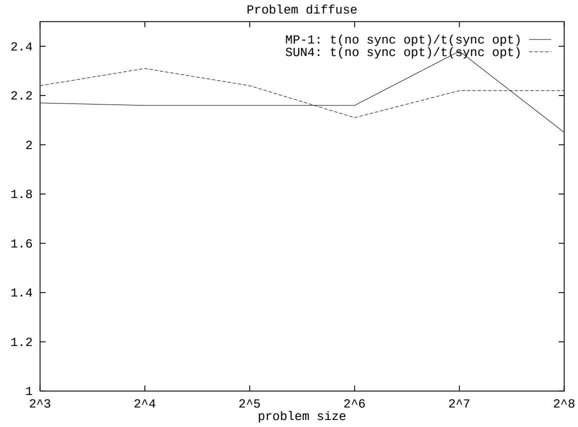
Problem: This is the example of a Modula-2* program containing one synchronous FORALL with a high potential of synchronization barriers that can be eliminated from its body.
Note: The graph has a differently scaled vertical axis.



Reduction of synchronization barriers: 44 ⟶ 6

4.1.7 Heat Diffusion Kernel

Problem: Calculate the temperature distribution on a square surface. The temperature on the edges of the surface are given as constants, while those on the inside are to be calculated with a diffusion equation.
Approach: In a straightforward implementation the value of a grid point is iteratively computed based on the values of neighboring grid points.
Note: The implementation can make easy use of the synchronous FORALL since no intermediate values must be stored manually.



Reduction of synchronization barriers: 34 ⟶ 16

5 Related Work

Most of the synchronous FORALLs introduced in parallel programming languages are restricted in their expressiveness to avoid the problem of high numbers of synchronization barriers. In parallel Fortran dialects for example, synchronous FORALLs contain only a single elementwise assignment or have copy-in-copy-out semantics. Thus, there are only few and obvious synchronization barriers that are not redundant.

The idea that the elimination of redundant synchronization barriers will result in better performance even on SIMD machines has been stated early in [6]. Christy argues against automatic virtualization as it has been used on the Connection Machine CM-2 in the PARIS [22] programming model. However, he concludes that the programmer should be able to detect the necessity of synchronization barriers and manually group statements together in virtualization loops. Although our approach is based on the same idea, we achieve the proposed improvement by an *automatic* and machine-independent optimization technique without manual virtualization.

Our approach has some similarities to the work of Hatcher and Quinn [12]. They deal with a data-parallel programming language in which each virtual processor has its own local address space. Data of different processes can only be accessed using explicit communication primitives. Synchronization is only necessary where communication operations occur. Therefore, Quinn and Hatcher restructure programs to group together communication operations and hence reduce the number of synchronization barriers. Because of the virtual shared memory in Modula-2* the problem is more complicated. In synchronous FORALLs it is not obvious where synchronization barriers really are necessary to achieve the intended semantics. A data dependence analysis is required. Furthermore, our solution is more general than the work of Quinn and Hatcher, since our restructuring works on a (sub-) expression basis and is extremely fine-grain.

6 Conclusion

In this paper we have presented a restructuring optimization technique for the compilation of synchronous FORALLs. Runtime performance has improved on Modula-2* benchmarks by over 40% on a MasPar MP-1 and by over a factor of 2 on sequential workstation. On MIMD machines we expect even better speedups.

Our work has solved some of the problems compiler writers typically have when dealing with synchronous FORALLs. Therefore, we encourage designers of parallel programming languages not to include restricted versions of synchronous FORALLs because of compiler simplicity, but to employ the general and most expressive form because of programming simplicity, instead.

References

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

[2] American National Standards Institute, Inc., Washington, D.C. *ANSI, Programming Language Fortran Extended (Fortran 90). ANSI X3.198-1992*, 1992.

[3] Marco Annaratone and Roland Rühl. A parallelizing compiler for distributed memory parallel processors. Technical Report 90/18, Swiss Federal Institute of Technology (ETH) Zürich, 1990.

[4] Alberto Apostoli, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. Technical Report CSD-TR-724, Purdue University, Department of Computer Sciences, May 1990.

[5] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Dordrecht, London, 1988.

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

[7] Constantine D. Polychronopoulos et al. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In Emily C. Plachy and Peter M. Kogge, editors, *Proc. of the 1989 International Conf. on Parallel Processing*, volume II (Software), pages 39–48. Pennsylvania State University Press, August 8-12 1989.

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

[9] 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.

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

[11] Philip J. Hatcher, Michael J Quinn, Anthony J. Lapadula, Ray J. Anderson, and Robert R. Jones. Dataparallel C: A SIMD programming language for multicomputers. In *Proc. of the 6th Distributed Memory Computer Conf.*, pages 91–98, Portland, Oregon, 28 April – 2 May 1991.

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

[13] Ernst A. Heinz. Transforming synchronous parallel FORALL contexts into equivalent asynchronous parallel FORALL contexts. Studienarbeit, University of Karlsruhe, Department of Informatics, July 1990.

[14] Ernst A. Heinz. Automatische Elimination von Synchronisationsbarrieren in synchronen FORALLs. Master's thesis, University of Karlsruhe, Department of Informatics, November 1991.

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

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

[17] Pawel Lukowicz. Code-Erzeugung für Modula-2* für verschiedene Maschinenarchitekturen. Master's thesis, University of Karlsruhe, Department of Informatics, January 1992.

[18] 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.

[19] Michael Philippsen, Thomas M. Warschko, Walter F. Tichy, and Christian G. Herter. Project Triton: Towards improved programmability of parallel machines. In *26th Hawaii International Conference on*

System Sciences, volume I, pages 192–201, Wailea, Maui, Hawaii, January 4–8, 1993.

[20] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, Dordrecht, London, 1989.

[21] 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.

[22] Thinking Machines Corporation, Cambridge, Massachusetts. *PARIS – Parallel Instruction Set, Version 6.0*, November 1990.

[23] 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.

[24] Walter F. Tichy, Michael Philippsen, Ernst A. Heinz, and Paul Lukowicz. From Modula-2* to efficient parallel code. In *3rd Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 6–9, 1992.

[25] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, January 1962.

[26] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[27] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, Reading, Mass., 1991.