From Modula-2* to Efficient Parallel Code

Walter F. Tichy, Michael Philippsen, Ernst A. Heinz, Paul Lukowicz

Universität Karlsruhe
Fakultät für Informatik
D-7500 Karlsruhe, F.R.G.
email: tichy@ira.uka.de

Abstract
Initial evidence is presented that machine-independent, explicitly parallel programs can be translated automatically into machine-dependent, parallel code that is competitive in performance with hand-written code.

The programming language used is Modula-2*, an extension of Modula-2, which incorporates both data and control parallelism in a portable fashion. An optimizing compiler targeting MIMD, SIMD, and SISD machines translates Modula-2* into machine-dependent C code. The performance of the resulting code is compared to the performance of equivalent, carefully hand-coded and optimized programs in MPL, a low-level parallel programming language for the MasPar MP-1 (a SIMD machine with 16K processors).

The Modula-2* programs typically achieve 40%-60% of the performance of the hand-coded, parallel versions. When targeting sequential processors, the Modula-2* programs reach 90% of the performance of hand-coded sequential C. With further optimizations, improvements are expected in those cases where the compiler still performs poorly.

These performance results indicate that it is feasible to write parallel programs in a machine-independent and portable way. Furthermore, when converting sequential programs into parallel ones, the conversion can concentrate on finding machine-independent parallel algorithms, while compilers perform the task of mapping the algorithms to a given parallel computer.

1 Introduction

Effective programmability of parallel machines is a pressing problem in parallel computing at this time. The most important aspect of this problem is portability: We simply cannot afford to rewrite parallel programs for every computer that comes along. Portability is as essential for parallel computing as it is for sequential computing. It can be achieved with machine-independent programming languages that allow clear expression of parallel algorithms and are free of hardware quirks that may differ from one computer to the next.

A second major concern is efficiency: Programs expressed in a high-level, portable language must be compilable into parallel machine code of satisfactory efficiency on a wide range of architectures. Efficiency is usually judged satisfactory when the compiled code approaches the performance of hand-tuned machine-code.

This paper is primarily concerned with efficiency. It provides a quantitative evaluation of the code produced by a compiler for a high-level, portable programming language with explicit parallelism. The language is Modula-2* (pronounced Modula-2-star), an extension of Modula-2. The extensions are extremely small and could be included in many other languages, including Fortran. The compiler presently targets the MasPar MP-1 computer (a SIMD machine with 16K processors) and sequential machines; work on code generation for MIMD machines is in progress. Measurements of a set of benchmarks provide initial evidence of the following

Hypothesis: Machine-independent, explicitly parallel programs can be compiled fully automatically into parallel machine code that is competitive in performance with hand-written code.
This result is important not only when writing new, explicitly parallel programs, but also for converting existing, sequential programs to parallel ones. With good compilers, the manual conversion of a sequential program can concentrate on finding parallel algorithms, while ignoring machine-dependent details. The following mapping to a given machine architecture is then performed fully automatically. The advantage of this separation of concerns is not only that it simplifies the conversion process, but it also assures that the result of the conversion is a machine-independent program that can be run on other computers simply by recompiling it.

We also present evidence that a compiler can produce highly efficient sequential code from parallel programs. This result is important for several reasons. First, it allows programmers to use parallel language constructs even when targeting sequential machines. Parallel constructs free programmers from the task of manually sequentializing an algorithm when parallel expression is more natural. Second, parallel programs can be developed and tested on sequential machines without incurring unjustifiable overhead. And finally, the fact that a compiler for parallel machines produces efficient sequential code simply by setting the number of processors to unity provides an indication about the generality and scalability of the code generation techniques employed.

Modula-2* is introduced in section 2, while section 3 presents the main features of our compiler. The benchmarks, experiments, and results are described in section 4. We conclude with a discussion of the important lessons learned.

2 Modula-2*

To allow for high-level, problem-oriented parallel programming, we have developed the machine-independent programming language Modula-2*. As described in reference [15], Modula-2* provides the following features:

- An arbitrary number of processes operate on data in the same single address space. (Note that shared memory is not required; a single address space merely permits all memory to be addressed, but not necessarily at uniform speed.)

- 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, synchronous, or asynchronous) at any nesting depth. Furthermore, there are no restrictions concerning the generation of new parallel processes inside procedures (recursive parallelism).

- All abstraction mechanisms of Modula-2 are also available to parallel programs.

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

1. The only way to introduce parallelism into Modula-2* programs is by means of the FORALL statement. There are two versions of this statement, a synchronous and an asynchronous one.

2. The distribution of array data may be optionally specified by so-called allocators in a machine-independent way. Array allocators do not have any semantic meaning; they are hints for the compiler.

Because of the compactness and simplicity of the extensions, they could easily be incorporated in other imperative programming languages, such as FORTRAN, C, or Ada. In Modula-2* the syntax of the FORALL statement is as follows:  

```
FORALLStatement = FORALL ident :: SimpleType IN (PARALLEL | SYNC)
      StatementSequence
END
```

\[1\] We use the EBNF syntax notation of the Modula-2 language definition with keywords in upper case, | denoting alternation, [...] optionality, and (...) grouping of the enclosed sentential forms.
**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**. Each process’ runtime constant is initialized to a unique value of **SimpleType**. All created processes then execute the statements in **StatementSequence**. The **END** of a **FORALL** statement imposes a synchronization barrier on the participating processes: The termination of the whole **FORALL** statement is delayed until all created processes have finished their execution of **StatementSequence**.

The version of the **FORALL** statement (synchronous or asynchronous) determines whether the created processes can execute **StatementSequence** in lock-step or concurrently.

Hence, for non-overlapping vectors X, Y, and Z a simple asynchronous **FORALL** statement suffices to implement the vector addition \( X := Y + Z \).

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

In contrast to the above, parallel modifications of overlapping data structures require synchronization provided by synchronous **FORALLs**. Thus, irregular data permutations can be implemented easily.

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

The effect of this **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 left-hand side 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.

Branches and loops inside synchronous **FORALLs** behave as if executed on a MSIMD (multiple SIMD) machine. This means that Modula-2* does not require any synchronization among different branches of synchronous **CASE** or **IF** statements. Consider the following **IF** statement as an example:

```plaintext
IF Cond-1 THEN
SS1
ELSIF Cond-2 THEN
SS2
... ELSIF Cond-k THEN
SSk
ELSE
SS0
END
```

If \( E \) is the set of processes executing the above **IF**, then its synchronous parallel execution proceeds as follows:

1. \( P := \{ p \in E \} \)
   \( j := 1 \)
2. All processes in \( P \) synchronously evaluate **Cond-j**. \( \forall j \in \{1, \ldots, k\} \) \( P_j := \{ p \in P \text{ for which } \text{Cond-}j \text{ evaluates to TRUE} \} \)
3. \( P := P \setminus P_j \)
   \( j := j + 1 \)
4. If \( j \leq k \) then continue with (2) otherwise continue with (5).
5. \( R_0 := P \)
6. For each \( j \in \{0, k\} \) all processes \( P_j \) then execute the statement sequence \( SS_j \) in synchrony. No synchronization is required between processes in different groups \( P_j \).
3 The Modula-2* Compiler

Modula-2* programs are translated first into an intermediate code to keep major parts of the compiler machine-independent. Based on a study of different parallel machines, we decided to use C augmented with machine-independent macros and keywords as an intermediate language. To generate the appropriate parallel C dialect, the intermediate code is combined with a machine-dependent macro package by a standard preprocessor. Thus, retargeting the compiler requires the exchange of the macro package and some libraries. The general structure of the Modula-2* compiler is shown below.

On parallel machines, optimizations tend to improve program runtime dramatically, often by an order of magnitude or more. Therefore, the Modula-2* compiler performs various optimizations and code restructurings, summarized below. Details can be found in [13].

Elimination of Synchronization Barriers

In the language definition of Modula-2* [15], the synchronous semantics require a large number of synchronization barriers. Most real synchronous FORALLs, however, only need a fraction thereof to ensure correctness.

As shown in reference [6], automatic elimination of such redundant synchronization barriers is possible. To detect redundant synchronization barriers, we apply data dependence analysis originally developed for parallelizing Fortran compilers [17, 3].

The elimination obviously pays off for machines without hardware-supported barrier synchronization. Most MIMD machines, for example, synchronize by message passing, which can be two or three orders of magnitude slower than instruction execution. However, synchronization point elimination is also beneficial on SIMD machines, because it reduces virtualization overhead, reduces the number of temporary variables needed, and may improve register usage.

To understand the techniques and advantages of automatic synchronization barrier elimination consider the synchronous FORALL statement below. The translation on the lower left shows an equivalent program, in which all synchronization points appear at the end of asynchronous FORALLs. The compiler detects that four of the six synchronizations are redundant and restructures the code accordingly. The result is shown on the lower right.

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

```
FORALL i: [1..N] IN PARALLEL
FORALL j: [1..N] IN PARALLEL
H[i] := Z[i+1];
```
END;
FORALL i: [1..N] IS PARALLEL
  z[i] := H[i];
END;

FORALL i: [1..N] IS PARALLEL
  Z[i] := I[i];
END;

FORALL i: [1..N] IS PARALLEL
  x[i] := H2[i];
END;

FORALL i: [1..N] IS PARALLEL
  Y[i] := Y[i];
END;

FORALL i: [1..N] IS PARALLEL
  Y[i] := H3[i];
END;

FORALL i: [1..N] IS PARALLEL
  Y[i] := H3[i];
END;

After unnecessary synchronization barriers have been eliminated, process virtualization takes place. This term refers to simulating n processes on p processors, where p < n. The most efficient technique is looping, where each processor loops over \( \lceil n/p \rceil \) processes. The result of this transformation is that the body of each asynchronous FORALL is wrapped into a loop. This transformation preserves semantics, because an asynchronous FORALL does not prescribe any temporal ordering of its processes. Observe also that the synchronization point elimination of the previous step has in effect resulted in loop fusion. Loop fusion distributes the virtualization overhead over several statements. Further sequential optimizations such as common subexpression elimination and strength reduction can now be applied. 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 variable or register, by taking advantage of the direction of the virtualization loop. However, such optimizations usually require additional synchronization points, trading space for time.

Data and Process Distribution

On distributed memory machines, the automatic distribution of array data over the available processors is a central problem. Two conflicting goals, (1) data locality and (2) maximum degree of parallelism, must be reconciled. Data locality means that data elements should be stored local to the processors that need them to minimize communication costs. Perfect data locality could be achieved by employing a single processor. Parallelism, which reduces runtime, may unfortunately reduce locality and increase communication costs. Additional goals for data distribution are: (3) Exploit special communication patterns supported by hardware and (4) simple address calculations to prevent addressing from becoming a dominant cost.

The problem of finding an appropriate data distribution can be divided into the subtasks of alignment and layout. Approaches to finding a data alignment include grouping together certain dimensions of different arrays, deriving super-arrays and index transfer functions, exploiting usage patterns, etc. See the work by Knobe [10] for examples. After alignment, a layout is determined: the resulting data structures are mapped onto the available processors.

In [12] we present a layout algorithm for mapping an arbitrary, multidimensional array onto an arbitrarily shaped multidimensional, nearest-neighbor network of a distributed memory machine. The individual dimensions of the array must be labeled with the machine-independent allocators of Modula-2*. These could even be derived by the compiler. The mapping algorithm exploits nearest neighbor communication and allows for efficient address calculations.

Known Weaknesses

Currently, our Modula-2* compiler does not perform any alignment. Our layout algorithm is used directly to distribute both arrays and processes. With this approach, we achieve locality of reference only if the distribution patterns of operands, the index expressions, and the processes match.

We intentionally omitted alignment in order to determine whether the general communication mechanism of the MasPar MP-1 is sufficiently fast. We further hoped that the Modula-2* programs would supply enough locality by themselves. However, a study of the resulting codes convinced us that alignment analysis is necessary.
At the moment, the compiler does not analyze memory references for locality. That is, the generated code checks at runtime whether a data element is local or must be accessed by general communication. Neighborhood communication is not used at all. The simple and frequently repeated case analyses between local, neighborhood and distant communication could actually be done much more efficiently in hardware. The software solution results in significant overhead when the compiler cannot detect locality statically. On the MasPar MP-1, for example, the necessary context stack activities cause a significant slowdown.

We are convinced that performance will improve when our Modula-2* compiler implements alignment and exploits locality of reference in the near future.

4 Test Problems and Results

At this time, our benchmark suite consists of nine problems collected from literature [1, 4, 8, 5]. For each problem, we implemented the same algorithm in Modula-2* in C, and in MPL. Then we measured the runtimes of our implementations on a 16k MasPar MP-1 and a Sparc-1 for widely ranging problem sizes.

In the Modula-2* programs, we use highly efficient library routines such as reductions and scans wherever possible. Because of known deficiencies in the current version of our Modula-2* compiler, two-dimensional arrays had to be “unrolled” manually into one-dimensional arrays. This will no longer be necessary in a forthcoming version of the compiler.

In MPL, we implemented the same algorithms, but carefully hand-tuned them for the architecture. We used local access, neighborhood communication, standard library routines, and other documented programming tricks whenever possible. To ensure the fairness of the comparison, the resulting MPL programs are as generally scalable as the Modula-2* programs. For example, scalability is not restricted to multiples of the number of available processors. Hence, boundary checks are required in every virtualization loop.

The C programs implement the parallel algorithms on a single processor. We use optimized sequential versions of the library routines wherever possible. The C code does neither contain C “hacks” nor does it exploit special sequential properties.

Performance Results

For each problem size of each algorithm, we measured its runtimes on both MasPar MP-1 and Sparc-1. On the MasPar MP-1, we used high-resolution DPU-timer, while we used the UNIX clock function to determine the sum of system and user time on the Sparc-1. \(t_{mpl}\) stands for the runtime on the MasPar MP-1 of a program written in MPL; \(t_c\) is the runtime of the sequential C program on the Sparc-1; \(t_{m2*}\) is the runtime of an Modula-2* program on either the MasPar MP-1 or Sparc-1, as appropriate. Since performance is defined as work (or problem size) per time we consider \(\frac{\text{size}}{t_{m2*}}\) as a measure of performance. Since \(\frac{t_c}{t_{mpl}} = \frac{t_{mpl}}{t_{m2*}}\) and \(\frac{\text{size}}{t_{m2*}}\), we can conclude that \(\frac{t_c}{t_{mpl}} = \frac{\text{size}}{t_{m2*}}\).

1. MPL versus Modula-2* on a 16k MasPar MP-1

   - The relative performance \(t_{mpl}/t_{m2*}\) is never worse than 10%. Modula-2* sometimes reaches 95% of the performance of MPL. Typically, Modula-2* achieves 40%-60% of the performance of the equivalent MPL program. \(t_{mpl}/t_{m2*}\) is shown as a dashed line in the graphs below.

   - The Modula-2* program texts are half the size of the equivalent MPL programs on average.

2. C versus Modula-2* on a Sparc-1

   - The relative performance \(t_c/t_{m2*}\) is typically around 90%. (This ratio is shown as a solid line in the graphs below.)

   - The Modula-2* program texts are half the size of the equivalent C programs.

---

\(^{2}\)MPL [11] is an extension of C designed for the MasPar MP-1 series of massively parallel processors. 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 sequential front-end program and a SIMD program with explicit interactions between the two. MPL provides special commands for general and neighborhood communication. Virtualization loops and address computations must be implemented by hand.
Resource Consumption

**Program Space.** Our compiler translates Modula-2* programs to MPL or C. The resulting programs consume negligibly more space than the hand-coded MPL or C programs.

**Data Space.** The only place where data structures need to be replicated into temporaries is during synchronous assignments. However, this replication is also required in hand-coded MPL. There is some overhead involved in controlling nested and recursive parallelism. A synchronous process presently requires approximately 40 bytes that can be avoided by optimizing for flat parallelism.

**Development time.** Due to compiler errors detected while implementing the benchmarks, no quantitative figures on implementation and debugging time can be provided. However, we estimate that the implementation effort in Modula-2* is a fifth of the effort required when writing MPL.

4.1 Estimation of Pi

**Problem:** Compute \( \pi \) using the equation \( \pi = \int_0^1 \frac{4}{1 + x^2} \).

**Approach:** Approximate the solution by computing

\[
\frac{1}{N} \sum_{i=0}^{N-1} \frac{4}{1 + x_i^2}
\]

(rectangular rule), where \( N \) is the problem size parameter and \( x_i = (i + \frac{1}{2})/N \) is the midpoint of the \( i \)th interval.

**Note:** In [9], Karp employs this problem to study parallel programming environments.

The runtimes of the Modula-2* implementation will improve when the compiler statically determines locality of data items. Another source of inefficiency is the implementation of virtualization loops: The MPL implementation does not allow for nested parallelism. The compiler for Modula-2* currently does not optimize for the special case of non-nested, flat parallelism. However, the overhead for nested parallelism becomes negligible for larger problem sizes.

4.2 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 [8].

The good result is caused by the fact that both MPL and Modula-2* must use general communication. Again, the general implementation of virtualization loops by the Modula-2* compiler results in some overhead for smaller problem sizes.
4.3 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 I:** 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 $b' - a' < \epsilon$.

**Approach II:** Again, the interval $[a, b]$ is divided evenly over all processes. Then each process performs Newton’s iteration. The algorithm terminates when a process finds the root.

**Note:** This problem occurs frequently in science and engineering applications [1].

4.4 Point in Polygon

**Problem:** A simple polygon $P$ and a point $q$ are given. Determine whether the point lies inside or outside the polygon. (A polygon is simple if pairs of line segments do not intersect except at their common vertex.)

**Approach:** Draw a line that is parallel to the vertical axis and starts at $q$. Count the number of intersections with $P$. The point $q$ lies inside $P$ if and only if this number is odd.

**Note:** This well-known algorithm from computational geometry appears in many text books, e.g. [1].

4.5 Longest Common Subsequence

**Problem:** Two strings $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$ 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 [14]. The parallel solution is based on [2].
The main reason for the superior performance of the hand-coded MPL programs is the way neighboring data elements are accessed. MPL exploits the hardware supported XNET communication, whereas the Modula-2* compiler currently uses the much slower general communication mechanism.

4.6 Prime Sieve

**Problem:** Compute all prime numbers in \([2, n] \).  
**Approach:** We implemented the classical prime sieve of Eratosthenes. However, rather than using a virtual process per candidate, the algorithm assigns a segment of candidates to each processor. This adaptive version works much faster since division can be replaced by indexing within each segment.  
**Note:** The problem was suggested by Hatcher [5].

The hand-coded version uses shift operations, while the MPL code generated by the Modula-2* compiler uses division. The current version of the MPL compiler does not detect this source of optimization. After replacing division by shifts in the code generated by the Modula-2* compiler, relative performance improved to above 75%. This optimization will be included in the forthcoming version 3.0 of the MPL compiler.

4.7 Pairs of relative primes

**Problem:** Count the number of pairs \((i, j)\) with \(2 \leq i < j \leq n\) that are relatively prime, i.e., the greatest common divisor of \(i\) and \(j\) is 1.  
**Approach:** The solution is based on a data-parallel implementation of the GCD algorithm followed by an add-scan.  
**Note:** The problem was suggested by Hatcher [5].
4.8 Red/Black Iteration

**Problem:** Implement a red/black iteration, i.e., the kernel of a solver for partial differential equations.

**Approach:** The implementation is straightforward. See for example [1]. It references neighboring data elements almost exclusively.

**Note:** This problem often serves as a case study for implementors of automatically parallelizing compilers, e.g. [7].

4.9 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 [16].

**Note:** The problem was suggested by Hatcher [5].
5 Lessons learned

While implementing the above problems in Modula-2* and MPL we learned the following lessons about compilers and parallel machines.

- Static detection and exploitation of local and, where relevant, neighborhood access is important, even on state-of-the-art parallel machines with high-speed communication networks. Fast hardware or microcoded selection of the best access mode at run time would be best. For future parallel machines we recommend:

  - Shared address space.3 All processors should be able to generate addresses for the entire memory of the system. On SIMD machines, the front-end’s memory should also be part of that address space. A shared address space is essential for pointers, but also helpful for indexing operations.

  - Simulating shared memory. A shared memory in which all of memory units can be accessed in uniform time would simplify programming and compiling greatly. Latency hiding and randomization techniques might help achieve a reasonable approximation of true shared memory. The total bandwidth of the network must be high enough to serve memory accesses for all processors at a rate that is comparable with accesses to local memory.

- Elimination of synchronization barriers and fusion of virtualization loops are essential for avoiding synchronization bottlenecks and for efficient virtualization.

- Hardware support for fast barrier synchronization is important for MIMD machines. The data communication networks introduce too much overhead for this purpose and are too slow to begin with.

- Special hardware instructions for reductions, scans, segmented scans, and specialized communication patterns are practically impossible to target from a higher-level language. The reason is that there are too many variations to express these operations, and the source language patterns to be analyzed are too large.

- Since reductions and scans are used frequently, highly optimized library routines for these are necessary, even for low-level languages such as MPL.

- Alignment of data and processes needs to be studied. (See the paragraph entitled “Known Weaknesses” on page 5.)

- Generality and orthogonality are expensive.

  In Modula-2* the programmer need not be concerned with the number of processors of the target

---

3A shared address space does not imply shared memory.
machine. To implement this generality, the generated code must contain boundary tests in every virtualization loop. The potentially nested and recursive parallelism in Modula-2* causes additional overhead.

6 Conclusion

We presented initial evidence that compilers for explicitly parallel, machine-independent programs can produce competitive code. The results were obtained by comparing compiled code with hand-written and hand-optimized code. Our compiler presently produces SIMD-code that reaches 40%-60% of the performance of the hand-coded versions. Further improvements are likely as additional optimization techniques are incorporated. A similar evaluation for MIMD machines is planned. Additional benchmarks are also being collected.

Good compilers for parallel machines not only provide portability for newly written programs. They also simplify the task of converting sequential programs to parallel ones, because the machine mapping can be done by the compiler, while the programmer can concentrate on finding machine-independent, parallel algorithms.

A study of the compiler-generated code should also provide insights for designers of future parallel machines, especially parallel RISCs.

A SPARC/SunOS 4.1.1 binary version of the Modula-2* compiler, the documentation, and the benchmarks are available via anonymous ftp from iran1.ira.uka.de under pub/programming/modula2star. In order to keep track of the Modula-2* community, we ask retrievers of our Modula-2* compiler to send us their full names and addresses. Send all correspondence to msc@ira.uka.de.

References


