Proceedings of the Working Conference on Massively Parallel Programming Models
Berlin, Germany, September 20–22, 1993

The Modula-2* Environment for Parallel Programming

Stefan U. Hänßgen, Ernst A. Heinz, Paul Lukowicz, Michael Philippsen, Walter F. Tichy
(haensgen|heinze|lukowicz|phlipp|tichy)@ira.uka.de
Universität Karlsruhe
Fakultät für Informatik
D-76128 Karlsruhe, F.R.G.

Abstract

This paper presents a portable parallel programming environment for Modula-2 – an explicitly parallel machine-independent extension of Modula-2. Modula-2* offers synchronous and asynchronous parallelism, a global single address space, and automatic data and process distribution. The Modula-2* system consists of a compiler, a debugger, a cross-architecture make, a runtime systems for different machines, and a set of scalable parallel libraries. Implementations exist for the MasPar MP series of massively parallel processors (SIMD), the KSR-1 parallel computer (MIMD), heterogeneous LANs of workstations (MIMD), and single workstations (SISD).*

The paper presents the important components of the Modula-2 environment and discusses selected implementation issues. We focus on how we achieve a high degree of portability for our system while at the same time ensuring efficiency.*

1 Introduction

The demand for increasing computer performance at reasonable cost leads to rising interest in parallel computer systems with tens of thousands of processors. To make such systems acceptable as serious platforms for scientific and commercial computing, they must be programmable in a problem-oriented and machine-independent manner. Hence, parallel programming languages must be freed of machine dependent communication and scheduling instructions. Programs need to be written independently of the number of available processors and the actual machine topology.

In this paper we present a parallel programming system that fulfills these requirements while achieving adequate performance of the compiled code [10]. It consists of a compiler for a problem-oriented language and a programming environment that provides a uniform problem view across different machine types (MIMD, SIMD and SISD).

Section 2 describes the main features of Modula-2*. In section 3 we then focus on the components of our system, in particular the architecture of the compiler and the source-level debugger. Finally, we give some benchmark results demonstrating the performance of the compiled codes.

2 Modula-2*

The programming language Modula-2* was developed to allow for high-level, problem-oriented, and machine-independent parallel programming. As described in [12] it embodies the following features:

- An arbitrary number of processes operate on data in the same *single address space* (or virtual shared memory). Note that shared memory is not required; a single address space merely permits all memory to be addressed uniformly but not necessarily at uniform speed.
- 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 or parallel) and at any nesting depth. Furthermore, additional parallel processes can be created inside procedures (recursive parallelism).
- All abstraction mechanisms of Modula-2 are available for parallel programming.

Modula-2* extends Modula-2 with just two language constructs:

1. The only way to introduce parallelism into Modula-2* programs is by means of the ***FORALL** statement* which has a synchronous and an asynchronous version.
2. The distribution of array data is optionally specified by so-called *allocators*. These are machine-independent data layout hints for the compiler without any semantic meaning.

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

```
FORALL <Ident> ":" <SimpleType> IN (PARALLEL | SYNC)
     <VarDecl>
     BEGIN
            <StatementSequence>
     END.
```

SimpleType is an enumeration or a possibly *non-static* subrange, i.e. the boundary expressions may contain variables. The `FORALL` creates as many (conceptual) processes as there are elements in *SimpleType*. The identifier introduced by the `FORALL` statement is local to it and serves as a runtime constant for every process created by the `FORALL`. The runtime constant of each process is initialized to a unique value of *SimpleType*.

Each process created by a `FORALL` obtains an instance of each variable declared in the declaration part and then executes the statements in *StatementSequence*. The `END` of a `FORALL` statement imposes a *synchronization barrier* on the participating processes; termination of the whole `FORALL` is delayed until each created process has finished its execution of *StatementSequence*.

In a synchronous `FORALL`, the created processes execute *StatementSequence* in lock-step while they run concurrently in the asynchronous case.

Hence, for non-overlapping vectors `X`, `Y`, and `Z` the following asynchronous `FORALL` statement suffices to implement the vector addition `X := Y + Z`.

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

In contrast to the above, parallel modifications of overlapping data structures may require synchronization provided by the synchronous version of the `FORALL` statement. Thus, even irregular data permutations are easy to formulate.

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

This synchronous `FORALL` permutes the vector `X` according to the 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 `FORALL`s is defined with a MSIMD (multiple SIMD) machine in mind. This means that Modula-2* does not require any synchronization between different branches of synchronous `CASE` or `IF` statements. The exact synchronous semantics of all Modula-2* statements, including nested `FORALL`s, and the effects of allocator declarations are described in [12].

3 The Modula-2* System

The Modula-2* system currently targets the MasPar MP series of massively parallel processors (SIMD), the KSR-1 parallel computer (MIMD), heterogeneous LANs of workstations (MIMD), and single workstations (SISD). The Modula-2* system consists of

1. an optimizing and restructuring compiler,
2. machine-dependent runtime systems,
3. a parallel debugger,
4. an automatic cross-architecture make, and
5. libraries of scalable parallel operations (enumeration, reduction, scan, etc.).

These components provide a uniform view of the system that is independent of the underlying hardware. By targeting workstations and LANs we ensure that parallel software can be developed and tested without actual access to expensive parallel hardware. Furthermore, the LAN compiler represents an easy way to exploit the computing power of workstation clusters.

3.1 Compiler

General Architecture. To keep major parts of the compiler machine-independent we use a two stage strategy to distinguish target machines. In the first stage we make a coarse distinction between different parallel programming models. To this end we use an abstract classification of parallel C dialects to control some structural aspects of the code generated by the compiler. This code is a general intermediate representation that we have chosen to be C augmented with machine-independent macros and keywords [8]. In the second stage we adapt our code to a specific target architecture by by combining the intermediate code with a machine-dependent macro package using a standard preprocessor. Thus,retargeting the compiler only requires the specification of the appropriate abstract class of the target language and the exchange of the macro package and some libraries. An example of code generated for different machines (16K PEs MasPar MP-1 and 4-processors KSR-1) can be found

Modula-2*

```
MODULE test
CONST N = 1024;
VAR X: ARRAY [0..N] SPREAD OF CARDINAL;
BEGIN
 FORALL i : [1..N-1] IN SYNC X[i] := X[i+1] END
END test.
```

MasPar

```
plural static struct testS__1 {CARDINAL A[1]}X;
plural static dispatcher() {
  _ParProcStart_();
   _ParItem_(Lwp_1_ID,Lwp_1);
  _ParProcEnd_();
}

void BEGIN_MODULE(){ /* begin main program */
  _InitPEs_();
  { /* begin FORALL IN SYNC */
    /* some declarations omitted here */
    plural LONGCARD H_1_i[1];

    /* FORALL initialization */
    FLow_i = _thispe_()+1;
    FHi_i = p_0_MIN(_thispe_()+1,1024);
    _InitBounds_(FHi_i,FLow_i,FNum_i);
    for(IR_i=0,i=FLow_i;i<=FHi_i;i++,IR_i++)
     {Num_i = 0;AB_i[IR_i] = 0;}_Sync_(0);

    /* virtualization loops */
    for(IR_i=0,i=FLow_i;i<=FHi_i;i++,IR_i++){
     { plural LONGCARD L_1;  /* get X[i+1]    */
       _GetGlobalPT_(L_1, _pl_((i + 1),16383,0),X.A[0]);
       H_1_i[IR_i] = L_1;}
     }_Sync_(0);              /* synchronize   */
    for(iIR_i=0,=FLow_i;i<=FHi_i;i++,IR_i++) {
     X.A[0] := H_1_i[IR_i];  /* store in X[i] */
    }_Sync_(0);              /* synchronize   */
  }/* end  FORALL IN SYNC */

  _EndPEs_();
} /* end main program */
```

KSR1

```
plural static struct testS__1 {CARDINAL A[N + 1];} X;
plural static dispatcher() {
  _ParProcStart_();
   _ParItem_(Lwp_1_ID,Lwp_1);
  _ParProcEnd_();
}

plural static Lwp_1()

 {/* begin FORALL IN SYNC */
  /* some declarations omitted here */
  plural LONGCARD H_1_i[256];

  /* FORALL initialization */
  FLow_i = 256*_thispe_()+1;
  FHi_i = p_0_MIN(256*(_thispe_()+1)-1+1,1024);
  _InitBounds_(FHi_i,FLow_i,FNum_i);IR_i=0;
  for(i=FLow_i;i<=FHi_i;i++,IR_i++)
   {Num_i = 0; AB_i[IR_i] = 0; } _Sync_(0);

  /* virtualization loops */
  for(IR_i=0,i=FLow_i;i<=FHi_i;i++,IR_i++){

    H_1_i[IR_i] = X.A[i + 1];    /* get X[i+1] */

  } _Sync_(0);                   /* synchronize */
  for(IR_i=0,i=FLow_i;i<=FHi_i;i++,IR_i++) {
    X.A[i] = H_1_i[IR_i];        /* store in X[i] */
  }_Sync_(0);
 } /* end  FORALL IN SYNC */

void BEGIN_MODULE() { /* begin main program */
 _InitPEs_();
  /* start the FORALL code on all PEs */
  {_CreateForallSProcs_(Lwp_1_ID);}
 _EndPEs_();
} /* end main program */
```

Figure 1: An example of Modula-2* code generation. A simple Modula-2* program (top) was compiled for a 16K MasPar MP-1 (bottom left) and a KSR-1 using 4 processors (bottom right). For clarity some less relevant declarations are omitted.

in Figure 1. It will be used to illustrate some principles of Modula-2* code generation throughout this section.

Abstract Classes of Parallelism. In the classification of a C dialect we first consider the mechanism for the creation of parallelism:

- **Data Parallelism.** Data parallelism means that all operations on distributed data are automatically executed in parallel. The parallelism is introduced by specifying a data distribution. An example of a data parallel model is the MasPar MPL language.
- **Thread Parallelism.** In this model the code to be executed in parallel has to be packed into separate program entities (e.g. procedure). To begin parallel execution the desired number of threads is explicitly launched. A good example of this model is the POSIX Thread Standard.
- **Region Parallelism.** In a region parallel model the parallel code sections are enclosed in an appropriate syntactic construct. Such parallel regions are automatically distributed to all PEs and launched. This concept can be found in many parallel Fortran dialects (e.g. KSR-1) and can easily be incorporated into C. Currently, however, we know of no parallel C dialect using region parallelism.

Each of the above classes is further subdivided according to the memory model into a *shared memory* and *distributed memory* subclass.

To provide some insight into the impact of the above classification on the code structure we consider the KSR-1 code in Figure 1. The KSR-1 falls into the thread parallel, shared memory category. Consequently, the contents of the `FORALL` statement is put into a procedure `Lwp1` and a thread creation macro `CreateForallSProc` is inserted into the main section. The array `X` is not distributed. It is declared the same way as it would have been in a sequential C code except for the `plural` keyword that has to be defined void. No communication statements are generated. In the MasPar code, on the other hand, the `FORALL` statement remains in the main section and no threads need to be created. Since the MasPar is a distributed memory machine the array `X` is distributed. It is defined with the length of 1 on each of the 16K PEs. As the overall size is `N=1024` only the elements on the first 1024 PEs are considered valid data. A communication macro `GetGlobalPT` is generated for remote access.

Intermediate Language. Our intermediate language is standard C with calls for data management, process management, processor control and other administrative tasks. Most of the time identical macros are generated for all machine classes described above. For a particular machine some macros are redundant and have to be defined void (e.g. the `plural` keyword on the KSR-1 or the synchronization macro `Sync` on the MasPar).

- **Data Management.** Data management macros consist of a `plural` keyword marking distributed data and statements for putting data to (`Put`) and getting data from (`Get`) arbitrary remote memory locations. There are different sorts of `Put` and `Get` statements for PE-PE, PE-frontend, nearest neighbor, and global communication. In Figure 1 a macro for PE-PE communication using the global router (`PutGlobalPT`) has been used. The parameters of the communication macros contain all information needed to compute the PE number and the local address of a data element from the Modula-2* array index.
- **Process Management.** This is a group of calls used to manage virtual processes. Depending on the bounds of a `FORALL` statement, an appropriate number of virtual processes is assigned to every PE. There is a `for` loop around every sequence of statements between two synchronization barriers. In Figure 1 there are two virtualization loops. The first one loads the values of `X[i+1]` into the temporary array `H_1_i`. The second stores this values into `X[i]`. Macros are generated to compute the values of the `FORALL` variable belonging to a given PE as well as the upper bounds, the lower bounds, and the increment of the loops from the PE number (`InitBounds`). Note that the parameters and the implementation of these macros determine the process distribution. Further macros are required to manage temporary storage, control the activation of individual processes in control constructs, and synchronize all processes after each virtualization loop (`Sync`).
- **Processor Control.** Except for the data parallel model macros are needed to schedule groups of virtual processes onto the individual processors. A close look at the KSR-1 program in Figure 1 reveals that in the thread parallel class two macro calls are generated to schedule each `FORALL` statement: a `CreateForallSProc` in the sequential code and a `ParItem` macro in a `dispatcher`

procedure. The scheduling is done in a master/slave manner. The dispatcher procedure loops on each processor waiting for the master to specify a `FORALL` to be executed. The master's call of `CreateForallSProc` sends a message containing an unique ID of the current `FORALL` statement(`Lwp_1_ID` in our case) to each processor. When this massage is received the dispatcher matches the ID with a local address of the `FORALL` procedure using the `ParItem` macro and starts parallel execution.

In the region parallel model the `CreateForallSProc` is placed at the beginning and a `EndForallSProc` at the end of each `FORALL`. In both the region and the data parallel models the dispatcher procedure has no meaning and all macros inside it are defined void.

Macros are also provided for initialization and cleanup (`InitPEs, EndPEs`), profiling, debugging and procedure parameter management.

Optimizations. On parallel machines optimizations tend to improve program runtime dramatically. Therefore, our Modula-2* compiler performs various optimizations and code restructuring as summarized below.

- **Elimination of Synchronization Barriers.** The semantics of synchronous `FORALL` statements in [12] require a large number of synchronization barriers. Most real synchronous `FORALL`s, however, only need a fraction thereof to ensure correctness. We have shown that the automatic elimination of such redundant synchronization barriers is possible [6]. To statically detect redundant synchronization barriers, the compiler applies a variant of standard data dependence analysis modified for our specific needs. Typically, this optimization improves performance by over 40% on a 16K MasPar MP-1 and by over a factor of two on sequential workstations.

- **Automatic Data and Process Distribution.** For distributed memory machines, the distribution of array data and processes over the available processors is a central problem.

  By analysis of usage patterns, the compiler statically derives arrangement information (a) to derive super-arrays that group together dimensions of different arrays and `FORALL`-ranges and (b) to transform subscript expressions accordingly. Implemented as a source-to-source transformation, this optimization results in dramatically enhanced locality and normally improves performance on the MasPar MP-1 by at least 40% [11].

  Furthermore, our compiler automatically maps arbitrary multi-dimensional arrays to the available processors. The scheme employed [9] enables nearest-neighbor communication and achieves efficient address calculations.

Implementation Restrictions. There are two major restrictions in the current compiler. The first one is the lack of pointers to distributed data and distributed open array parameters (except for shared memory machines). The second one concerns the generation of efficient code for nested and recursive parallelism and the automatic recognition of nearest neighbor communication which are both in the experimental stages.

3.2 Modula-2* Runtime System

The Modula-2* runtime system performs the initialization, maintenance, and cleanup of code sections executed in parallel. The runtime system functions, e.g. remote data access and synchronization, are used to implement the machine-independent macro interfaces described in the previous section.

For different architectures, different infrastructures are used. The MasPar runtime system makes use of the MasPar system library, while the LAN runtime system is built on top of p4, a message passing parallel programming system available for a variety of machines.

Figure 2 illustrates the setup of the LAN runtime system processes and their interaction. The system consists of two components per PE which execute all virtual PEs' instructions in virtualization loops. The *Worker* performs the distributed computations. For remote memory access the *Worker* performs synchronized read/write accesses itself, while the *Runner* deals with asynchronous get/put memory accesses. It uses the `ptrace` system call for directly accessing data of the *Worker*. A central *Syncer* provides global synchronization barriers and I/O[1].

The current version of the LAN system runs on a KSR-1 parallel computer and a network of SparcStations. Ports to other MIMD architectures are in progress.

[1]The *Control* and the *Master Debugger* processes are described more closely in the debugger description on page 8.

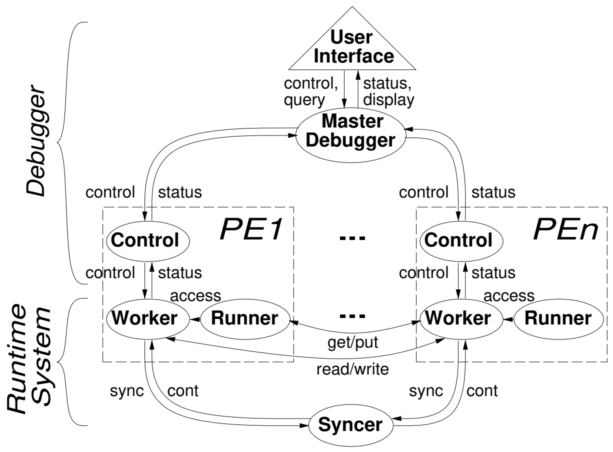

Figure 2: Process interaction in the LAN-based runtime system with debugging support

3.3 Debugger MSDB

Due to the high level of abstraction, many of the usual problems of parallel programming are of no concern to the Modula-2* user, e.g. data access deadlocks, virtualization, and communication operations. These are all taken care of by the compiler.

This observation shifts the focus of debugging from machine dependent issues to more abstract problems such as visualization of parallel activities, data, and dynamic activation trees. For performance tuning, we additionally support profiling [4].

Figure 3 shows a screendump presenting all of MSDB's display features.

Concepts. The debugger provides the usual features that are also found in sequential debuggers, such as setting and examining variables, setting breakpoints, and stepping through the program. Additionally, MSDB supports different views of distributed data and parallel execution.

All distributed information about the program is integrated on one screen with several windows. At each single step, or whenever a breakpoint is encountered, all displayed information is updated.

In the remainder of this section, we focus on the concepts that use high-level abstractions in terms of the language and help to understand the parallel execution.

- **Activation Trees and Grouping.** When many different parallel activities are running in parallel, it is difficult to keep an overview of the program's execution.

  The *activation tree* (or *structure tree*) shows, in terms of programming language constructs, where the activities inside the program are located at the moment. It is updated internally each time an activity enters a new statement that affects the control flow, e.g. a loop, an `IF`-statement or a `FORALL`.

  The leafs of the activation tree are the locations of the processing while their predecessors represent their dynamic history. This resembles a stack in a conventional program, only that MSDB shows all constructs, not just function calls, and displays parallel activities in form of a tree.

  To make this approach feasible even with a large number of concurrent activities, identical subtrees are *grouped* together. This means that all activities that are inside the same pass through the same control flow branch inside a `FORALL` are shown as one node. A simple example of grouping several nodes in an `IF` and a `WHILE` statement executed in parallel is presented in Figure 4.

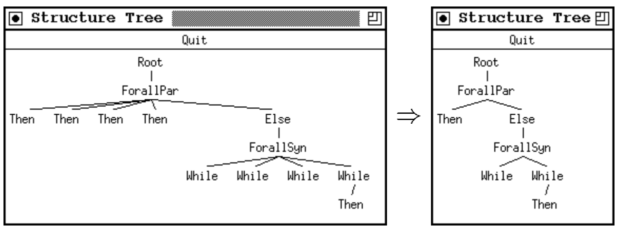

Figure 4: Dynamic activation tree before and after grouping of equivalent nodes

- **Multiple program counters.** We need to keep track of the program counter positions in the code. Since maintaining one source code window for each virtual PE quickly gets confusing with a large number of PEs, we use multiple program counter arrows in the source code window to represent the *grouped activities*. The size of the arrow indicates the number of processes that are at this position. More detailed information about the processes in the group can be obtained by clicking on the arrow.

- **Visualization of data.** In the debugger multidimensional arrays are visualized by 'cutting' through the data and viewing 2-dimensional slices thereof (1-dimensional data gets mapped onto a 2-dimensional array for a better screen presentation). The exact value of the array element under the mouse pointer is also displayed.

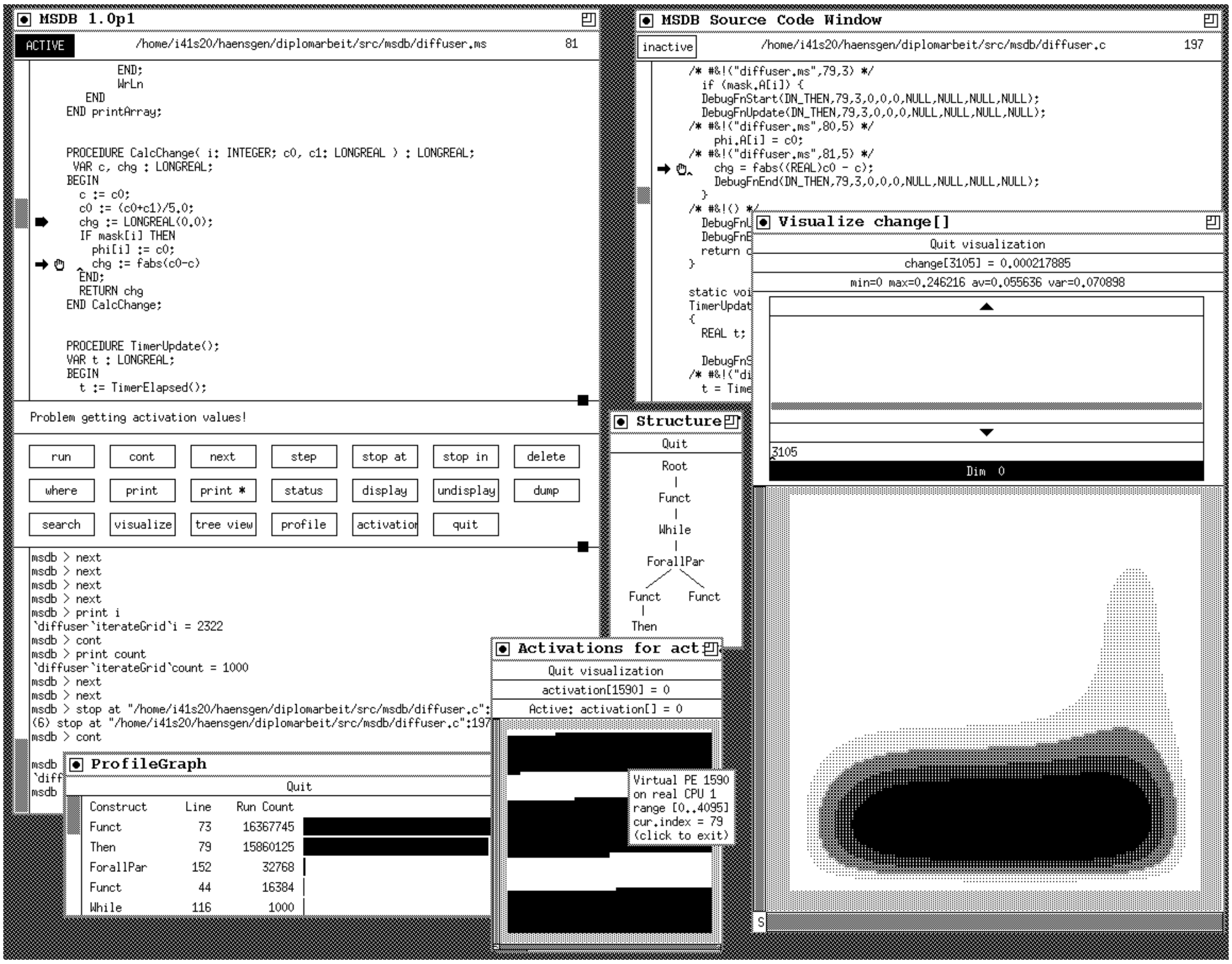

Figure 3: MSDB in action. The program being debugged is a heat diffusion simulation. Its results are shown in a value visualizer. A visualization of the virtual PEs' activities as well as the (grouped) activation tree and a profile graph can also be seen. The left window shows the Modula-2* source with two differently sized program counter markers and a breakpoint, the right window displays the corresponding C source.

Different kinds of visualizers are available: *Comparison* visualizers highlight the array elements which are equal to, greater than or less than a user defined value. *Range* visualizers represent a generalization of comparison which highlights all values inside a user-defined range. Finally, *Value* visualizers map array elements to a grayscale representation of the data, compensating for statistical peaks.

- **Profiling and activities.** For performance tuning the user needs to know which parts of the program are executed how often.

  MSDB keeps track of the processes entering, repeating and leaving constructs. Thus, it can provide the user with a display of the most-often used *structures*. The number of runs through a statement corresponds roughly to the time spent there.

  Abstracting from machine dependent displays of processor activities, the debugger has a specialized visualizer displaying the currently active virtual PEs as black regions in a PE matrix.

  Although the real PEs are not visible to the Modula-2* programmer, combining this view with the profiling view gives more information about the load distribution. One can easily distinguish one process running a loop 1000 times from 1000 processes running it one time.

- **Breakpoints.** The semantics of a breakpoint in parallel are more complicated than in the sequential case. A common problem is the halting of all processes in a consistent state.

  In MSDB breakpoints are global and asynchronous. A stop command is broadcast whenever one process encounters a breakpoint. The stop is *global* to provide consistent handling among MIMD, SIMD, and SISD machines, as on the latter two all stops are necessarily global. In addition, virtualization loops make it difficult to stop just one single process.

  Asynchronous breakpoints are consistent with Modula-2*'s semantics: the end of the `FORALL` statement is a synchronization barrier. Thus, the stop command reaches every process eventually. In a synchronous `FORALL`, all processes operate in lockstep, so they are stopped in the same state. The semantics of the asynchronous `FORALL` permit an indeterministic order of execution among the processes, so it is of no concern where exactly a process is stopped.

  The semantics of stepping through the program are similar. All processes execute the next statement unless they are at the end of a `FORALL`. There they wait for the remaining processes to synchronize with them.

Instrumentation for Debugging. To support the debugger, the Modula-2* compiler places additional information in the code:

- **Mapping from Modula-2* to C.** The compiler generates code for the parallel C dialect of the target machine. Thus, the debugger can rely on existing source level debuggers such as `dbx` and `gdb`. However, it has to transparently map Modula-2* to C and vice versa while debugging a program. This is not simple due to the extensive restructuring of the code done by the compiler [10].

  Thus, the compiler generates hints for the line and variable mapping and places them inside *special comments* which are parsed by the debugger when reading the source code.

- **Debug Functions.** To build the dynamic activation tree and also to generate the profiling information, *debug functions* are added to each Modula-2* structure (loops and branches). They are called at all crucial points:

  - Before entering the structure, a corresponding debug function adds a node to the activation tree and stores, among others, the current line number, the values of `FORALL` variables and loop boundaries.
  - A debug function records profiling information for every run through the structure.
  - When leaving the structure, a debug function updates the activation tree again.

  Thus, the program itself generates information on its run using the debug functions[2].

Debugger Architecture. Currently, prototype implementations of MSDB exist for a *network of workstations* and a *sequential program* running on a single workstation. The LAN implementation is based on the p4 system for communication and process creation. To Inspect and change both data and program state, the UNIX debuggers `dbx` and `gdb` are used. The X Windows based user interface is derived from `xdbx`.

The basic processes and their interaction have been described in the runtime system section 3.2. To support the debugger, we have two more kinds of processes which are also shown in Figure 2:

The *Control* process stops the *Worker*, modifies it, and retrieves information for visualization etc. It also monitors the *Worker*'s state and reports changes. It consists of the vendor's debugger (e.g. `dbx`) and an interface which parses this debugger's output and issues new instructions. Thus, the underlying debugger may be changed by just adapting the interface.

All processes are coordinated by a *Master Debugger* which also handles the user interface and merges information. For communication between the processes, a set of debugging primitives is used (e.g. for setting a breakpoint, performing a single step or obtaining the value of a single variable). Additionaly, a bulk data transfer mode makes array visualization more efficient.

This arrangement provides a uniform view of debugging on different architectures.

3.4 Cross-Architecture Make

The Modula-2* System provides a cross-architecture make that automatically generates standard Unix makefiles for Modula-2* programs. This enables separate compilation of libraries and selective recompilation of recently changed modules or libraries.

[2]At the moment, the runtime overhead is about 30% but the debug function code is not optimized yet. We expect to reduce the overhead to an acceptable 10%.

The generated makefiles reflect the module and library dependencies of a program according to the normal Modula-2 import rules. Internally, the Modula-2* make builds the complete transitive module dependence graph of a program in order to derive the correct makefile dependencies.

In addition, our top level make driver allows for automatic cross-architecture makes. Its command line syntax is as follows:

```
mm <program> [":"<architecture>] ["@"<machine>]
```

The program parameter names the Modula-2* program to be made. The optional architecture parameter specifies the desired target architecture and the optional machine parameter supplies the name of a machine on which C compilation and object code linking are to be performed. If no optional parameters are given the whole make process takes place on the machine and architecture under which the make is exectuted.

3.5 Parallel Libraries

The Modula-2* parallel libraries aim at scalability, portability, and efficiency of frequently used parallel operations. Scalability means that the library routines operate on open array parameters of arbitrary size. We ensure portability by providing the same machine-independent Modula-2* interfaces on all target machines. To achieve efficiency we exploit special low-level features of each target machine in the different library implementations.

Another interesting feature of these libraries is their functional diversity. Wherever possible, normal, masked, segmented, and universal (masked plus segmented) versions of the parallel operations are provided.

4 Benchmark Results

Our benchmark suite consists of 13 problems collected from literature [1, 2, 3, 5, 7]. For each problem we implemented the same algorithm in Modula-2*, in MPL[3], and in sequential C. Then we measured the runtimes of our implementations on a 16K MasPar MP-1 and a Sparc-1 for widely ranging problem sizes. The detailed results can be found in [10].

[3] MPL is an extension of C designed for the MasPar MP-1. In MPL the number of available processors, the SIMD architecture, the 2D mesh-connected processor network, and the distributed memory are visible. MPL provides special commands for general and neighborhood communication. Virtualization and address computations must be implemented by hand.

In the following discussion, t_{mpl} stands for the runtime on a MasPar MP-1 of a program written in MPL; t_c is the runtime of a sequential C program on a Sparc-1; t_{m2*} is the runtime of a Modula-2* program on either a MasPar MP-1 or a Sparc-1 (as appropriate).

- **MPL versus Modula-2* on a 16K MasPar.** The general relative performance t_{mpl}/t_{m2*} is quite stable over all problem sizes and averages to 80%. Modula-2* typically achieves 70%-90% with peaks at 100% of the MPL performance. Problems that can be implemented in MPL with a high amount of neighborhood communication on arrays with multiple dimensions currently perform quite bad in Modula-2* since the necessary optimization is not implemented yet. The Modula-2* program texts are, on average, half the size of the equivalent MPL programs.
- **C versus Modula-2* on a Sparc-1.** The general relative performance t_c/t_{m2*} is quite stable over all problem sizes and averages to 90%. Modula-2* typically achieves 70%-90% of the sequential C performance with peaks at 100%. The Modula-2* program texts are, on average, half the size of the equivalent C programs.

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

5 Conclusion

We have described a parallel programming environment that ensures full source code portability across the whole range of commercially available parallel hardware architectures. The system achieves competitive runtime performance and presents a uniform user interface that is independent of the target architecture.

References

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

[2] John T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan*

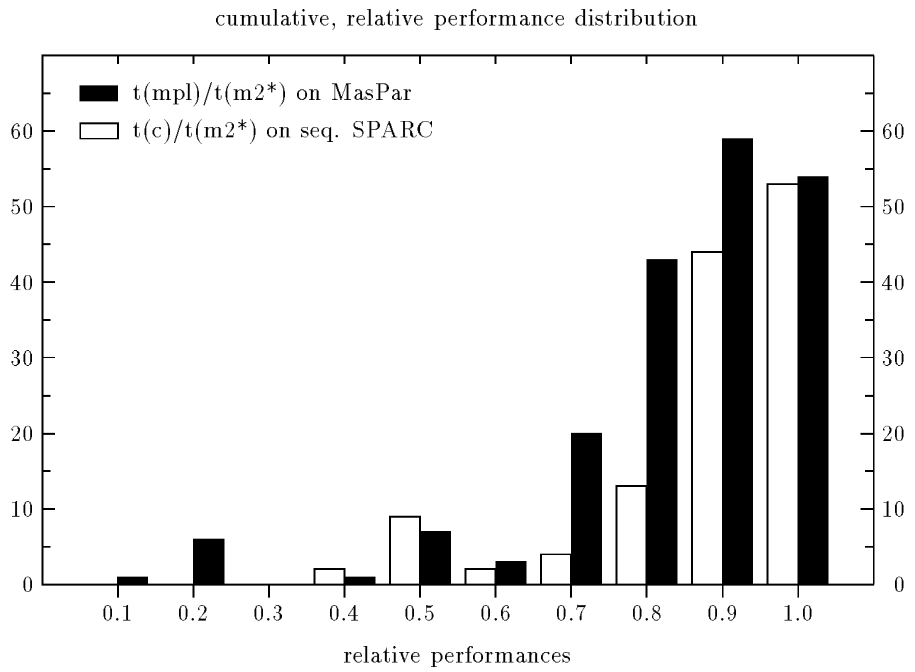

Figure 5: Distribution of relative performances

Problems. Elsivier Science Publishers, Holland, 1992.

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

[4] S.U. Hänßgen: *Ein symbolischer X Windows Debugger für Modula-2**. Master's thesis, Computer Science Dept., University of Karlsruhe (1992)

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

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

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

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

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

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

[11] Michael Philippsen and Markus U. Mock. Data and process alignment in Modula-2*. In *AP'93 International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction, Saarbrücken, Germany*, pages 141–149, March 1–3 1993.

[12] 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.