Proceedings of the Working Conference on Programming Models for Massively Parallel Computers
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
Universität Karlsruhe, Fakultät für Informatik, D-76128 Karlsruhe, F.R.G.
email: (haensgen | heinze | lukowicz | phlipp | tichy)@ira.uka.de

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, graphical X Windows control panel, runtime systems for different machines, and sets of scalable parallel libraries. The existing implementation 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 paper describes 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 ever-increasing demand for more computing power at a reasonable cost has brought about a 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 especially 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.

Below, we present a parallel programming environment that fulfills these requirements while achieving adequate performance of the compiled codes [9]. It centers around a compiler for a problem-oriented language, and various tools, together with libraries that provide a uniform problem view across different machines and architectures.

The rest of the paper is organized as follows. Section 2 describes the main features of our language Modula-2*. Section 3 focusses on the different components of our system, explaining in particular the details of the compiler and source-level debugger. Section 4 provides some benchmark results that demonstrate the performance of our compiled codes. We conclude with a summary and discussion of future work.

2 Modula-2*

The programming language Modula-2* was developed to support high-level and machine-independent parallel programming. As described in [11], it embodies the following features:

- An arbitrary number of processes operate on data in the same *single address space*. Note that shared memory is not required. The 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. Additional parallel computations can be spawned inside procedures (recursive parallelism).
- All abstraction mechanisms of Modula-2 are also available for parallel programming.

Despite its many achievements, Modula-2* extends Modula-2 with just two new language constructs:

1. To support explicitly parallel computations, Modula-2* introduces the *`FORALL` statement* in a synchronous and an asynchronous version.
2. The distribution of array data may optionally be specified by so-called *allocators*. These are machine-independent layout hints for the compiler without any semantic meaning.

Because of their compactness and simplicity, these extensions could easily be incorporated into other imperative programming languages, e.g. 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 created process. The runtime constant of each process is initialized to a unique value of *SimpleType*.

All processes created by a `FORALL` obtain private instances of every variable declared in the optional *VarDecl* part. Then they execute the statements in *StatementSequence*. The `END` of a `FORALL` imposes a synchronization barrier on the participating processes; termination of the whole `FORALL` is delayed until all created processes have finished their execution of *StatementSequence*.

The processes of a synchronous `FORALL` 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. Here, synchronous `FORALL`s even allow for easy formulation of arbitrary irregular data permutations.

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

The above synchronous `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 behaviour of branches and loops inside synchronous `FORALL`s is defined with a MSIMD (multiple SIMD) machine in mind. Consequently, 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 [11].

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. an interactive parallel debugger,
4. libraries of scalable parallel operations (enumeration, reduction, scan, etc.),
5. an automatic cross-architecture make, and
6. a graphical X Windows control panel.

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 handle different target machines.

In the first stage, we make a coarse distinction between different parallel programming models. To that end, we use an abstract classification of parallel C dialects to control some structural aspects of the generated code. This code is a general intermediate representation that we have chosen to be C, augmented with machine-independent macros and keywords [7].

In the second stage, we adapt our code to a specific target architecture by combining the intermediate code with a machine-dependent macro package using a standard preprocessor. Thus, in principle, retargeting the compiler only requires the specification of the appropriate abstract class of the target language and the exchange of the macro package plus some libraries. Examples of code generated for different machines – the MasPar MP-1 and KSR-1 – are given

in Figure 1. These examples will be used throughout the section to illustrate some major principles of Modula-2* code generation.

Abstract Classes of Parallelism. For the classification of a C dialect we first consider the available mechanism of parallelism:

- **Data Parallelism.** This model implies that all operations on distributed data are automatically executed in parallel. Hence, parallelism is introduced by specifying a data distribution. Good examples of this are the MasPar MPL language and Dataparallel C [3].

- **Thread Parallelism.** In this model the code to be executed in parallel has to be packed into separate program entities, e.g. procedures. To start parallel execution the desired number of threads must be explicitly launched. A well-known example belonging to this category is the POSIX Thread Standard.

- **Region Parallelism.** In the region model all parallel code sections are enclosed in appropriate syntactic constructs. Such parallel regions are automatically launched and distributed to the available processors. Currently, this concept prevails in parallel Fortran dialects, e.g. KSR-1 Fortran, but it could easily be incorporated into C as well.

Each of the above classes is further subdivided according to the memory model into *shared memory* and *distributed memory* subclasses. To provide some insight into the impact of the above classification on the code structure, we now return to Figure 1.

The KSR-1 falls into the "thread parallel with shared memory" category. Consequently, the contents of the `FORALL` statement are put into a procedure `Lwp1` and a thread creation macro `CreateForallSProc` is inserted into the main section. The array `X` is not distributed; instead, it is declared in the same way as it would have been in sequential C, except for the `plural` keyword that has to be defined void. Because of the shared memory, no communication statements need be generated.

The MasPar MP-1 belongs to the "data parallel with distributed memory" category. Here, the `FORALL` statement remains in the main program, where data parallel loops perform the desired computations. Since the MasPar is a distributed memory machine, the array `X` is distributed. It is defined with a length of 1 on each of the 16K PEs. As its overall size is `N=1024`, only the elements on the first 1024 PEs are considered valid data. Communication macros like `GetGlobalPT` are generated for remote access.

Intermediate Language. Our intermediate language is C augmented with calls and macros for data management, process management, processor control, and other administrative tasks. Normally, identical macros are generated for all machine categories, although they may be redundant for a particular machine. If so, they have to be defined void, e.g. the `plural` keyword on the KSR-1, or the synchronization macro `Sync` on the MasPar.

- **Data Management.** These macros comprise a `plural` keyword marking distributed data, and statements for putting data to (`Put`) as well as getting data from (`Get`) arbitrary remote locations. There are different sorts of `Put` and `Get` templates for PE-to-PE, PE-to-frontend, nearest-neighbour, and global communication. The macro `GetGlobalPT`, as found in Figure 1, is an example of global PE-to-PE communication. 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 set of calls is used to manage virtual processes. Depending on the range of a `FORALL`, an appropriate number of virtual processes is assigned to every physical processor. The virtual processes are scheduled in loops between consecutive synchronization barriers. In Figure 1 there are two such *virtualization loops*. The first loads the values of `X[i+1]` into the temporary array `H_1_i`. Afterwards the second stores the temporary values into `X[i]`. Macros are generated to compute the subranges of the `FORALL`s executed by each processor together with their upper bounds, lower bounds, and loop increments (`InitBounds`). Note that the parameters and the implementation of these macros determine the process distribution. Additional macros are required to manage temporary storage, to control the activation of individual processes in control constructs, and to synchronize all processes after each virtualization loop (`Sync`).

- **Processor Control.** Except for data parallel machines, macros are needed to schedule groups of virtual processes onto the individual processors. A closer look at the KSR-1 program in Figure 1 reveals that, for thread parallelism, two macro calls are generated to schedule each

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++)
     {AB_i[IR_i] = 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 */
```

KSR-1

```
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++)
   {AB_i[IR_i] = 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. The simple Modula-2* program `test` (top) is compiled for a 16K MasPar MP-1 (bottom left) and a 4-processor KSR-1 (bottom right). Note that some less relevant declarations are omitted for the sake of clarity.

`FORALL`: namely, a `CreateForallSProc` in the sequential part, and a `ParItem` macro in the `dispatcher` routine. The scheduling is done in a master/slave manner. The dispatchers loop on every processor waiting for the master to specify the next `FORALL` to be executed. The master's call of `CreateForallSProc` sends a message to each dispatcher containing a unique ID of the desired `FORALL` statement (`Lwp_1_ID` in Figure 1). When a dispatcher receives this message, it matches the ID with a local address of the `FORALL` procedure, using the `ParItem` macro to start the local execution. In the region parallel model, a `CreateForallSProc` is placed at the beginning and an `EndForallSProc` at the end of each `FORALL`. In both the region and the data parallel models, the dispatcher routine is defined void.

More macros are provided for initialization, cleanup, profiling, debugging, and parameter management.

Optimizations. On parallel machines, optimizations tend to improve program runtime dramatically. Therefore, our restructuring Modula-2* compiler performs various sophisticated optimizations.

- **Elimination of Synchronization Barriers.** The semantics of synchronous `FORALL` statements [11] require a large number of synchronization barriers. Most real synchronous `FORALL`s, however, need only a fraction thereof to ensure correctness. We have shown that the automatic elimination of such redundant synchronization barriers is possible [5]. For their static detection, the compiler applies data dependence analysis with special Modula-2* modifications to the source programs. Typically, this optimization improves performance by over 40% on a 16K MasPar MP-1, and by over a factor of two on sequential workstations [9].

- **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% [10]. Furthermore, our compiler automatically maps arbitrary multi-dimensional arrays to the available processors. The scheme employed [8] enables nearest-neighbour communication and achieves efficient address calculations.

Implementation Restrictions. There are two major restrictions in the current compiler. The first is the lack of pointers to distributed data and distributed open array parameters on distributed memory targets. The second concerns the generation of efficient code for nested and recursive parallelism, as well as the automatic recognition of nearest-neighbour communication, both of which are 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. Different infrastructures are used for different architectures and machines. The MasPar runtime system makes use of the MPL system library, while the LAN runtime system is built on top of p4, a message passing interface available for a variety of machines.

Figure 2 illustrates the setup of the LAN runtime system, its basic processes, and their interaction. Two components per physical processor execute all statements of the virtual processes in simple loops. The *Worker* performs the actual computations and does synchronized remote read/write accesses itself. The *Runner* deals only with asynchronous remote memory accesses; it uses the Unix `ptrace` system call for direct access to data of its associated *Worker*. The central *Syncer* provides global barriers and I/O.[1] The current version of the LAN system runs on KSR-1 parallel computers, and on networks of SparcStations. Ports to other MIMD architectures are in progress.

3.3 Debugger

Due to its high level of abstraction, many of the usual problems of parallel computation are of no concern to the Modula-2* programmer – e.g. communication, process virtualization, and data access deadlocks. All of these are taken care of by the compiler. Hence, the focus of debugging shifts from machine-dependent

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

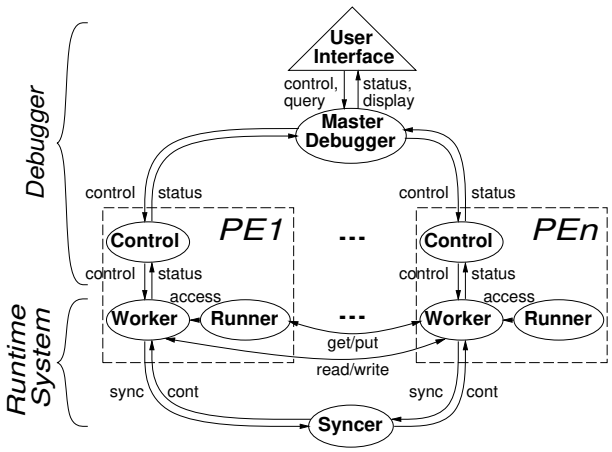

Figure 2: Process interaction in the LAN runtime system with debugging support.

issues to more abstract problems like the visualization of parallel activities, of potentially distributed data, and of dynamic activation trees [2]. For performance tuning we also support profiling. Figure 3 shows a screen dump presenting most display features of our Modula-2* debugger MSDB.

Concepts. MSDB provides standard sequential features such as breakpoints, stepping, and examining as well as changing variables. Furthermore, it supports different views of parallel execution and potentially distributed data. All information about the whole program is integrated on one screen with several windows. At each single step, or upon encountering a breakpoint, the display information is completely updated.

In the remainder of this section we focus on our new debugging concepts. We rely on high-level abstractions in terms of the source language in order to make the execution of parallel computations transparent for the programmer.

- **Activation Trees and Grouping.** When many different computations take place in parallel, it is often difficult to keep track of the overall program execution. Here, MSDB's structured *activation trees* greatly improve the user's insight by showing where the parallel activities are currently located inside the program in terms of Modula-2* statements. The trees are internally updated whenever any activity enters a new statement that affects control flow – e.g. branch, loop, and `FORALL`. The leafs of an activation tree show the actual locations of activities, while their predecessors represent their dynamic history. This resembles the activation stack of a sequential program, with the extension that MSDB traces all language constructs —not only function calls— and displays parallel activities in the form of a tree. To make this approach feasible for large numbers of parallel activities, identical subtrees may automatically be grouped together. Then, all activities in the same pass through the same control flow branch of a `FORALL` are shown as one node. Figure 4 illustrates grouping with a simple example.

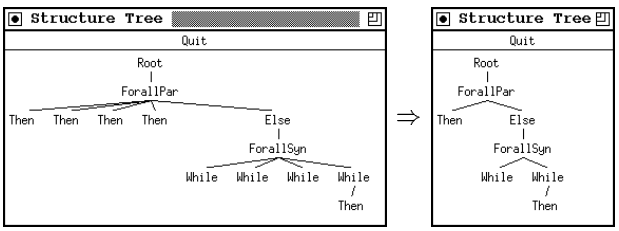

Figure 4: Dynamic activation tree before and after grouping of equivalent nodes.

- **Multiple Program Counters.** MSDB needs to keep track of the program counters of all virtual processes. Since maintaining one source code window for each of them quickly gets confusing, MSDB displays multiple counter arrows in a single source code window to represent all grouped activities. The size of the arrows indicates the number of processes residing at that position. More information about the process groups can be obtained by clicking on their arrows.

- **Breakpoints.** The semantics of breakpoints in parallel contexts are more versatile than those of their sequential counterparts. To stop all virtual processes in a consistent state is a central problem. MSDB provides *global* and *asynchronous* breakpoints. It broadcasts a stop command whenever any process encounters a breakpoint. The stop is global to ensure consistent behaviour on MIMD, SIMD, and SISD machines. Furthermore, virtualization loops make it nearly impossible to stop just single processes without causing deadlock or starvation. Asynchronous breakpoints are consistent with Modula-2* semantics because the end of each `FORALL` comprises a synchronization barrier. Hence, the stop eventually reaches all processes. In a synchronous `FORALL` all processes operate in lock-step such that they are always stopped in the same state. The semantics of asynchronous `FORALL`s permit an indeterministic order of execution among the

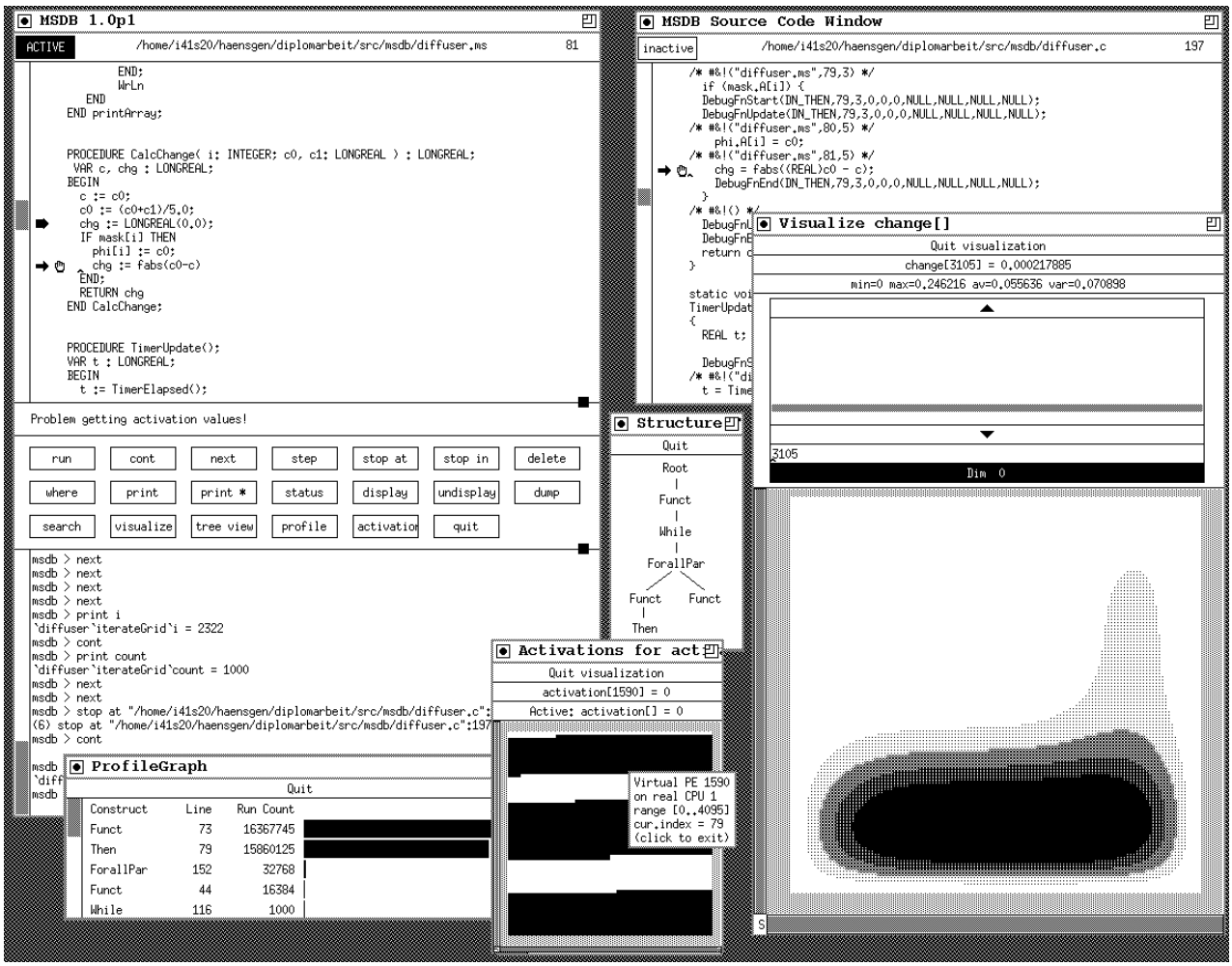

Figure 3: MSDB in action. The program being debugged is a heat diffusion simulation. Its results are shown in a value visualizer. A visualization of the parallel activities as well as the (grouped) activation tree and a profile graph have also been selected. The left window shows the Modula-2* source with two differently sized program counter arrows and one breakpoint. The right window displays the corresponding generated C source.

processes. Therefore, it is of no concern where exactly they are stopped.

- **Stepping.** The semantics of stepping through a program follow from breakpoint semantics. All processes execute the next statement until they reach the `END` of a `FORALL`. There they wait for the remaining processes to synchronize with them.

- **Data Visualization.** Our debugger is able to visualize multidimensional arrays, "cutting" through their data and viewing 2-dimensional slices thereof. Note that MSDB also displays 1-dimensional data in two dimensions because of better screen presentation. Three different kinds of visualizers are available. *Comparison visualizers* highlight all array elements that are equal to, greater than, or less than a user-defined value. *Range visualizers* provide a generalization of comparisons by highlighting all elements that fall into a user-defined range. *Value visualizers* map array elements to a gray scale representation of the data, thus compensating for statistical peaks.

- **Process Visualization.** Abstracting from machine-dependent displays of processor activities, MSDB incorporates a special visualizer to display all currently active virtual processes as black regions in an overall process matrix. Although the physical processors are not visible to

the Modula-2* programmer, a combination of activity view and profiling data sheds light on the actual load distribution.

- **Profiling.** For performance tuning, the user needs to know which parts of a program are executed how often. Because MSDB keeps track of all processes entering, repeating, and leaving Modula-2* statements, it can easily collect exact data about the frequency of every invocation.

Instrumentation for Debugging. When executed with its debug option set, the Modula-2* compiler places additional information into the generated code that MSDB requires for proper operation.

- **Mapping from Modula-2* to C.** The compiler generates code for the parallel C dialect of the target machine. Because we rely on existing source level debuggers, e.g. `dbx` or `gdb`, MSDB needs a transparent mapping from Modula-2* to C and vice versa for consistent presentation. This is far from trivial due to the extensive code restructuring done by the compiler [9]. MSDB builds the correct maps by parsing special comments that the compiler inserts into the generated C code. These comments contain raw information about line and identifier correspondence.

- **Debug Functions.** To build the dynamic activation trees, and to generate the profiling information, *debug functions* are added to the original Modula-2* source at all crucial points:

  - Before entering a compound statement, the corresponding debug function adds a new node to the activation tree, and saves the current line number as well as the values of `FORALL` variables and loop boundaries.
  - Another debug function records profiling information for every pass through the compound statement.
  - On exit of the compound statement, a debug function updates the activation tree again.

  Thus, the program generates much useful information about its execution all by itself.[2]

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

The basic LAN processes and their interaction have already been described in Section 3.2. Especially for debugging support, we have two more classes of processes, which are also shown in Figure 2. The *Control* process stops the *Worker*, modifies it, and retrieves information for visualization etc. Furthermore, the *Control* monitors the *Worker*'s state and reports changes. It consists of the machine's default debugger (e.g. `dbx`) and an interface that translates the output of this debugger into appropriate portable instructions. Thus, the underlying debugger may simply be changed by interface adaption. All processes are coordinated by a *Master Debugger* which handles the user interface and merges incoming data. 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 variable. Additionally, a bulk data transfer mode makes array visualization more efficient. This arrangement provides a uniform view of debugging on all machines and architectures.

3.4 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 operations are provided.

3.5 Cross-Architecture Make

The Modula-2* system provides a cross-architecture make that automatically generates standard Unix makefiles for Modula-2* programs. This supports both separate compilation of libraries, and selective recompilation of recently changed modules or libraries. The generated makefiles reflect the module and library dependencies of a program according to the normal Modula-2 import rules. Internally, the Modula-2*

[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%.

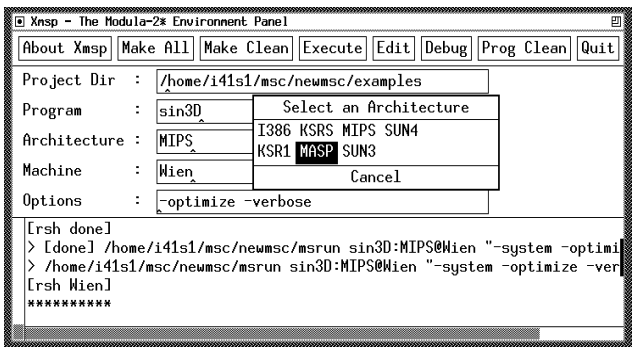

Figure 5: Xmsp in action. The program in work is our 3D sine wave projection example `sin3D`. The user has just opened an architecture selector window and is about to change the target architecture from MIPS (DECStation) to MASP (MasPar). The user-specified options and target machine transparently apply to all triggered actions.

make builds the complete transitive module dependency 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 specifies 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 on which the make is exectuted.

3.6 Control Panel

To make the whole Modula-2* system easy to use, we designed Xmsp, a special control panel with a graphical X Windows interface. Xmsp is completely customizable by each individual user and it serves as the central integrational unit of all the various Modula-2* tools. As depicted in Figure 5, the intuitive graphical interface of the Modula-2* control panel faciliates cross-architecture makes, program and project cleaning, logged local and remote program execution, as well as project, program, and architecture selection.

4 Benchmark Results

Our benchmark suite consists of 13 problems collected from literature [1, 3, 6]. For each problem, we implemented the same algorithm in Modula-2*, MPL and sequential C. We then measured the runtimes of our implementations on both a 16K MasPar MP-1 and a Sparc-1 for widely ranging problem sizes. The detailed results can be found in [9].

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 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 using multi-dimensional arrays with a high amount of neighbourhood communication currently perform far from optimal 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 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 6 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 – i.e. the whole sample of all our data points.

5 Conclusion

We have described a parallel programming environment that ensures full source code portability across a wide 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 machine.

Future research must focus on removing known restrictions and deficiencies that cause non-optimal code generation in our current compiler. Furthermore, we will migrate from Modula-2* to Modula-3* [4] because of its modern design and object-oriented features.

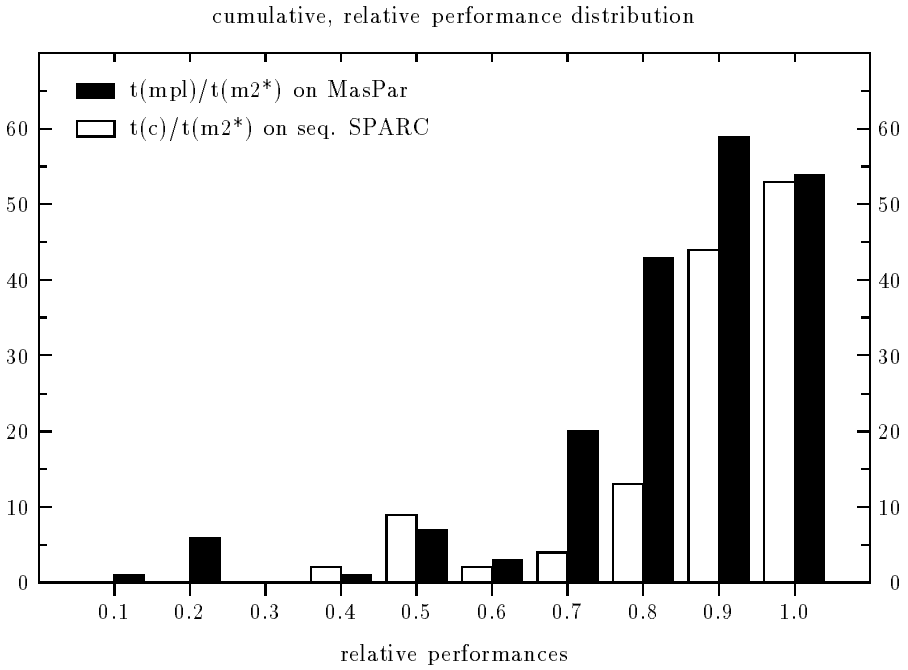

Figure 6: Distribution of accumulated relative performances

The Modula-2* system is freely available from the University of Karlsruhe. Please contact `msc@ira.uka.de` for further information.

References

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

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

[3] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press Cambridge, Mass., London, England, 1991.

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

[5] E.A. Heinz and M. Philippsen. Synchronization barrier elimination in synchronous forall statements. Technical Report No. 13/93, University of Karlsruhe, Dept. of Informatics, April 1993.

[6] J. JáJá. *An Introduction to Parallel Algorithhms*. Addison-Wesley, Reading, Massachusetts, 1992.

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

[8] M. 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.

[9] M. Philippsen, E.A. Heinz, and P. Lukowicz. Compiling machine-independent parallel programs. *ACM SIGPLAN Notices*, Vol. 28, No. 8, pages 99–108, August 1993.

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

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