This paper appeared in: H. Küsters and W. Werner, editors, Proceedings of International Conference on Mathematical Methods and Supercomputing in Nuclear Applications, volume 2, pages 3–9, Karlsruhe, Germany, April 19–23, 1993

Programming Parallel Supercomputers

Walter F. Tichy and Michael Philippsen
Universität Karlsruhe
Fakultät für Informatik
D-7500 Karlsruhe, F.R.G.
email: `(tichy|phlipp)@ira.uka.de`

Abstract

This paper discusses future directions in tools and techniques for programming parallel supercomputers. We base the discussion on two important observations:

- Automatic parallelization of sequential programs will not achieve supercomputer performance in real applications. Instead, applications will have to be written with explicit parallelism.
- Machine-independence of parallel programs is a precondition for wide acceptance of parallel computers.

We comment on High Performance Fortran (HPF) and conclude that HPF will achieve machine-independence to an initially satisfactory degree, but that another language revision can be expected.

Machine-independence does not imply poor performance. We present evidence that explicitly parallel, machine-independent, and problem-oriented programs can be translated automatically into parallel machine code that is competitive in performance with handwritten code.

Furthermore, we show that interactive, source-level and problem-oriented debugging of explicitly parallel program has recently become a reality.

1 Introduction

The leading users of parallel supercomputers have long abandoned the hope that automatically parallelized, sequential programs will achieve supercomputer speed. It appears that the class of algorithms that can be transformed automatically from sequential to a highly efficient parallel form is quite small. For older applications, we can therefore expect a long, drawn-out process of manual or semi-automatic parallelization, similar to the process of vectorization. Various transcription tools will ease this transformation, but programmers will be responsible for the difficult part, namely the development of parallel algorithms. As a significant start, most vendors of parallel supercomputers are deeply involved in (manually) parallelizing large libraries of mathematical subroutines.

Users demand that parallel programs, both newly written and reengineered, be machine-independent. Otherwise, the software investment necessary to use parallel supercomputers will be affordable by relatively few users. Machine-independence is achieved with a programming language that does not reveal the idiosyncrasies of a given parallel machine, such as message-passing primitives, synchronization constructs, or details about the memory layout and interconnection network. These details typically change from one computer model to the next. Any program that depends on such details will either have to be rewritten for every new generation of supercomputers or be thrown away after a short time of use.

2 Whither Fortran?

Fortran90 [2] provides instructions geared for vector computers, but lacks facilities for SIMD or MIMD parallel machines. Consequently, vendors were forced to extend either Fortran77 or Fortran90 with new language constructs for parallelism. However, these constructs are machine-specific and therefore lock users into a particular vendor.

High Performance Fortran (HPF) [10], however, provides a reasonable chance of achieving machine-independence. The standardization process is fortuitously helped by a consolidation trend among parallel computer architectures to the extent that programmers will be able to rely on a few rules of thumb to hold across a spectrum of parallel computers. These rules of thumb make it possible to expect good performance on a reasonably large subset of parallel machines without reprogramming.

Steele writes about the purpose of HPF in [18]:

> The goal of High Performance Fortran (HPF) is to extend Fortran90 to provide additional support for data parallel programming (defined as a style of programming with a single conceptual thread of control, a global name space, and loosely synchronous parallel computation) to facilitate top performance on MIMD and SIMD computers with non-uniform memory access costs, while also promoting, or at least not impeding, performance on other machines. The idea is to promote portability of Fortran programs over a large class of computers, multihead vector computers, shared-memory multicomputers, mainframes, and workstations.
>
> Note that support for explicit MIMD computation is *not* one of the goals of HPF.

The main extensions in HPF are array alignment and distribution directives and an element-wise FORALL statement. Alignment and distribution are critical for distributed memory machines, where the cost of accessing nonlocal memory is high. Since compiler technology has not yet solved the problem of deriving alignment and distribution automatically, HPF leaves the decision to the programmer.

A simple example, taken from reference [18], follows. Suppose A is an array of dimensions 1000 x 1000, while B has dimensions 998 x 998. Assume elements A(I,J) interact with elements B(I+1,J+1), so A and B should be aligned accordingly. Furthermore, elements within a column of A interact frequently, but elements in different columns interact much less often, so columns should not be split across multiple processors. The following HPF directives encode this information:

```
      REAL A(1000,1000),B(998,998)
CHPF$ ALIGN B(I,J) WITH A(I+1,J+1)
CHPF$ DISTRIBUTE A(*,BLOCK)
```

The ALIGN directive is self-explanatory. The DISTRIBUTE directive specifies that each column of A is to remain within a single processor, while the rows should be divided into equal segments. The effect is that groups of columns are spread over the available processors. It is also possible to direct the compiler to redistribute arrays at runtime.

The language features for expressing parallelism are the Fortran90 array intrinsics and a limited form of a FORALL statement. This statement executes an enclosed assignment statement in parallel on all elements in the given index ranges.

```
FORALL (I=2:N-1,J=2:N-1) B(I,J) = (A(I-1,J)+A(I,J+1))/2
```

Conceptually, all parallel executions of the enclosed assignment execute synchronously, but a compiler may detect that synchronization steps may safely be omitted to improve performance. Note that explicit communication instructions are unnecessary because of the shared name space. A compiler or runtime system may have to insert special instructions to read or write non-local array elements.

As compiler technology improves, it may be possible to omit the alignment and distribution directives, because the compiler may be able to generate them and even take the target machine architecture into account. For instance, Wholey [21] shows that the best mapping varies with the interconnection structure of the machine, the size of the machine, and the problem size. Therefore, a good compiler might actually override a directive with a better choice for a given architecture and problem size.

Redistribution of arrays is a difficult aspect for programmers, because they must weigh the cost of the actual redistribution against the access cost without redistribution, possibly for a number of target architectures. Again, the compiler may be better equipped to solve this optimization problem, especially because the compiler already has to solve the general alignment and distribution problem for program-generated temporary variables. First promising results [12, 17, 6, 3] indicate that the alignment and distribution problem may be compiler-solvable.

The FORALL statement in HPF is severely limited. For instance, it is not possible to place subroutine calls, IF statements, or other FORALL statements into its body. This restriction eliminates nested and recursive parallelism and often forces unnecessary distortions of otherwise clear, data-parallel programs. Furthermore, asynchronous parallelism is completely lacking, except through the escape of calling subroutines written in different language, possible in another Fortran.

We think that HPF is a significant step in the right direction, especially since it achieves a high degree of machine-independence and does not perpetuate the myth of automatic parallelization. However, if standardized in its present form, it seems that yet another extension will be needed before long.

3 Results from the Compiler Front: Generating Efficient Parallel Code

In this section, we present some recent results showing that high-level, machine-independent programs can be compiled into machine code that is comparable in efficiency with hand-written machine code.

The language we use is Modula-2* [19], an extension of Modula-2. We chose Modula-2, because it is both more modern and smaller than Fortran. The extensions are a superset of those in HPF, however. Just as HPF, Modula-2* provides a single, global name space with potentially non-uniform access cost. Array distribution is similar to array distribution in HPF, except that alignment directives are unnecessary. Instead, the compiler derives the proper alignments automatically. Parallelism is expressed with an elementwise `FORALL`. The synchronous version of this `FORALL` operates much like the HPF `FORALL`, except that it is fully orthogonal to the rest of the language: Any statement, including conditionals, loops, other `FORALL`s, and subroutine calls may be placed in its body. Thus, the language explicitly supports nested and recursive parallelism. Finally, there is an asynchronous version of a `FORALL`, allowing full control parallelism. The unrestrained orthogonality of Modula-2* (such as mixing data and control parallelism in one program) makes it possible to write parallel programs that are both easy to understand and machine-independent. However, the orthogonality makes great demands on compiler technology.

Our compiler research [16, 9, 13, 15] indicates that machine-independent, explicitly parallel programs can indeed be compiled into parallel machine code that is competitive in performance with hand-written code. This result is not only valid for translation of Modula-2* programs, since the translation of the general `FORALL` statement of Modula-2* is more demanding than translation of less expressive forms of data parallelism, e.g. vector parallelism or the `FORALL` of HPF.

3.1 The Compiler Benchmark

At the moment, our benchmark suite consists of 12 problems collected from literature [1, 5, 11, 8, 4]. The problems are given briefly in the appendix. For each problem, we implemented the same algorithms in Modula-2*, in sequential C, and in MPL[1]. Runtimes were measured on a 16K MasPar MP-1 (SIMD) and a SparcStation-1 (SISD) for widely ranging problem sizes. Measurements for LANs are not yet available because the tedious and error-prone task of implementing hand-coded versions is still in progress.

Modula-2* Programs. In Modula-2* we employ optimized parallel libraries for reductions and scans. A technical deficiency in our current Modula-2* compiler forced us to manually "unroll" two-dimensional arrays into one-dimensional equivalents. This will no longer be necessary in the near future.

MPL Programs. In MPL we implemented the same algorithms as in Modula-2* and carefully hand-tuned them for the MasPar MP architecture. The MPL programs make extensive use of local access, neighborhood communication, standard library routines, and other documented programming tricks. To ensure the fairness of the comparison, the resulting MPL programs are as generally scalable as their Modula-2* counterparts. Hence, scalability is not restricted to multiples of the number of available processors. Therefore, boundary checks are required in all virtualization loops.

Sequential C Programs. The sequential C programs implement the parallel algorithms on a single processor. We use optimized sequential libraries wherever possible. The C code does not contain dirty "hacks".

In the following, we first present performance results and then compare the resource consumption of these three program classes. We only summarize the results. For detailed information see [20].

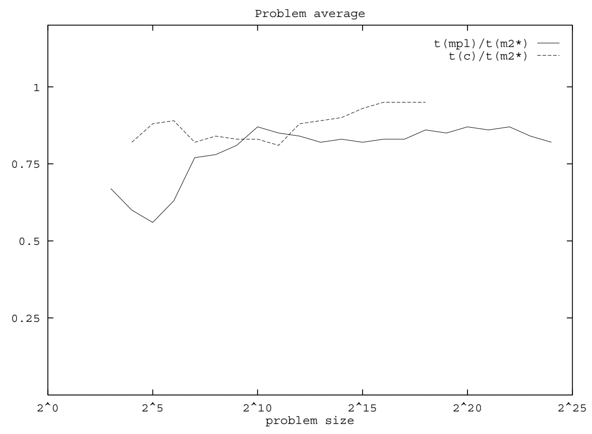
3.2 Performance Results

For different problem sizes we measured the runtime of each test program on a 16K MasPar MP-1 and a SparcStation-1. Time was measured with the high-resolution DPU timer on the MasPar and the UNIX `clock` function on the SparcStation (sum of user and system time). Below, t_{m2*} represents the Modula-2* runtime on either a 16K MasPar MP-1 or a SparcStation-1 (as appropriate); t_{mpl} gives the MPL runtime on a 16K MasPar MP-1; t_c stands for the sequential C runtime on a SparcStation-1.

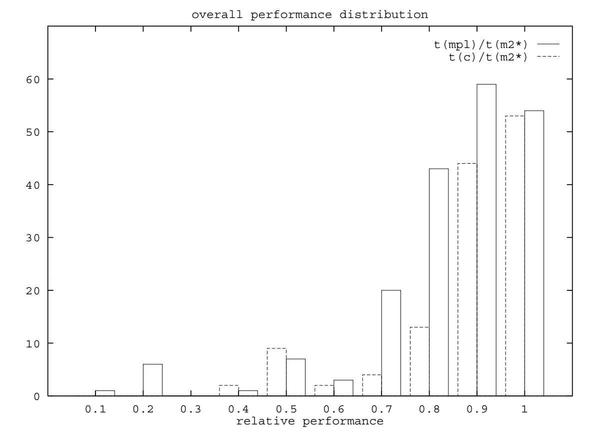
We define performance as work or problem size per time and focus on the following relative performances: $\frac{size}{t_{m2*}} / \frac{size}{t_{mpl}} = t_{mpl}/t_{m2*}$ and $\frac{size}{t_{m2*}} / \frac{size}{t_c} = t_c/t_{m2*}$.

[1]MPL [14] is a data-parallel extension of C designed for the MasPar MP series. 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 SIMD program and a sequential frontend program with explicit interactions between the two. MPL provides special commands for neighborhood and general communication. Virtualization loops and distributed address computations must be implemented by hand.

Note the ratio scale as the vertical axis; good performance is indicated by curves approaching unity.



For problem sizes ranging from 2^3 to 2^{25} we derived the relative performances from our runtime measurements. The resulting general relative performance, averaged arithmetically over all test programs, is shown above.



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

MPL versus Modula-2*:

- The general relative performance of Modula-2* is quite stable over all problem sizes and averages to 75%.
- Modula-2* typically achieves 60%–90% of the MPL performance.
- Modula-2* often reaches over 90% of the MPL performance, with peaks at 100%. The average relative performance for a single problem is never worse than 30%.

Sequential C versus Modula-2*:

- The general relative performance of Modula-2* is again quite stable over all problem sizes and averages to 90%.
- Modula-2* typically achieves 80%–100% of the sequential C performance.
- Modula-2* often reaches over 95% of the sequential C performance. The average relative performance is never worse than 70% for a single problem.

3.3 Resource Consumption

The comparison is based on the criteria program space, data space, development time and runtime performance.

Program Space. Our compiler translates Modula-2* programs to MPL or C. The resulting programs consume slightly more space than the hand-coded MPL or C programs. Regarding source code length, Modula-2* programs are typically half the size of their corresponding MPL or C programs.

Data Space. The memory requirements of the Modula-2* programs are typically similar to those of the MPL and C programs. Memory overhead, i.e. variable replication into temporaries, occurs during synchronous assignments. This replication, however, is also required in hand-coded MPL. Furthermore, there is some additional overhead involved in controlling synchronous, nested, and recursive parallelism (16 bytes per `FORALL`).

Development time. Due to compiler errors detected while implementing the benchmarks, we cannot give exact quantitative figures on implementation and debugging time. However, we estimate that the implementation effort in Modula-2* is a fifth of that for MPL.

Comparison with sequential code is important for two reasons. First, it shows that one may develop parallel applications on sequential machines, without facing an undue overhead. Once a program runs correctly on the development platform, it can be recompiled and run with larger data sets on a parallel supercomputer.

Second, good performance on sequential machines is indicative of the scalability and quality of the compiler: The sequential code was produced by essentially setting the number of target processors in the compiler to unity. The performance figures show that the virtualization introduced by the compiler (virtualization is needed whan mapping a given degree of parallelism to a machine with fewer processors) has been optimized adequately and introduces little overhead.

4 MSDB – Parallel Debugger, Profiler, and Visualizer

Due to the high level of abstraction in Modula-2*, 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.

A high-level, parallel language shifts the focus of debugging from machine-dependent to problem dependent issues, such as visualization of data, activity tracking, and performance profiling [7].

We have built an interactive, source-oriented debugger and data visualizer for parallel programs written in Modula-2* called MSDB. Figure 1 shows a screendump presenting all of MSDB's display features. MSDB provides the usual debugger features such as setting and examining variables, setting breakpoints and stepping through the program. Breakpoints are global, i.e. all processes are stopped if one of them encounters a breakpoint. In the following, we describe the concepts that directly support high-level Modula-2* abstractions.

- **Activation Trees and Grouping.** The *activation tree* shows where the parallel processes of the program are currently located. It is updated each time a process enters or leaves a statement that affects the control flow. Whenever new parallel processes are created, the corresponding branch may split into several edges. Thus the display is similar to showing a stack of invocations on a sequential machine, except that MSDB displays a tree (or "cactus stack") to visualize the parallel activities.

  To keep the tree manageable, equivalent processes (going through the same control flow) are collected into a single edge in the tree. For example, when control flow enters a synchronous `FORALL`, only the `FORALL` will be displayed. When execution reaches an `IF` statement within the `FORALL`, the branch will split into two edges, representing the two sets of activities in the two branches of the `IF`. Thus, grouping makes process tracking feasible even for massive parallelism. In its center window, Figure 1 shows an extremely simple activation tree, where all parallel processes of a `FORALL` have been collapsed into one edge.

  Multiple program counters provide a link between the activation tree and the source code. Each counter represents a process group and points to the group's current source code location in the open source code windows.

- **Data Visualization.** Multidimensional arrays are visualized in MSDB by viewing 2-dimensional slices through the data.

  Different kinds of visualizers are available, depending on the kind of information the user wants to obtain: *Comparison* and *Range* visualizers highlight all values that fulfill a given condition. *Value* visualizers provide a grayscale representation of array data. See the bottom right window of Figure 1 for an illustration.

- **Profiling.** For performance tuning, the user needs to know which parts of the program are executed how often and on which process. MSDB provides two views of this information: (a) a display of the most often used *structures* and (b) a specialized visualizer for the current *activities* of the virtual processes. The latter visualizer represents each process as a black or white pixel in an array, depending on whether it is active or not. The bottom center window of Figure 1 contains an example.

The *sequential version* of a program running on a single workstation and the *distributed runtime system* for a network of workstations are now in use. Work on other platforms, such as MasPar MP-1 and KSR1, is in progress.

5 Conclusion

Machine-independent programming of parallel supercomputer applications is within reach. With the necessary training in parallel algorithms and the availability of optimizing compilers and source debuggers, developing parallel programs should eventually become only moderately more complex than programming sequential computers.

Appendix: Benchmark Problems

- **List Rank:** 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.
- **Root Search:** Determine the value of $x \in [a, b]$ such that $f(x) = 0$, given that f is monotone and continuously differentiable.
- **Point in Polygon:** 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.)
- **Longest Common Subsequence:** Two strings $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ are given. Find a string $C = c_1 c_2 \cdots c_p$ such that C is a longest common subsequence of A and B. (C is a subsequence of A if it can be constructed by removing elements from A without changing their order. A common subsequence must be constructible from both A and B.)
- **Estimation of Pi:** Use the equation $\pi = \int_0^1 \frac{4}{1+x^2}$.
- **Prime Sieve:** Compute all prime numbers in $[2..n]$.
- **Pairs of Relative Primes:** 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.
- **Red/Black Iteration:** Implement a red/black iteration, i.e. the kernel of a solver for partial differential equations.
- **Transitive Closure:** The adjacency matrix of a directed graph with n nodes is given. Find its transitive closure.
- **Mandelbrot Set:** Compute the Mandelbrot set.
- **Hamming's Problem:** A set of primes $\{a, b, c, \ldots\}$ of unknown size and an integer n are given. Find all integers of the form $a^i \cdot b^j \cdot c^k \cdot \ldots \leq n$ in increasing order and without duplicates.
- **Doctor's Office:** Simulate the following queuing problem from [4]: a set of patients, a set of doctors, and a receptionist are given. Patients become sick at random, are assigned to one of the doctors by the receptionist, and treated in a random amount of time.

References

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

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

[3] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *Proc. of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–28, Charleston, SC, January 10–13, 1993.

[4] John T. Feo, ed., *A Comparative Study of Parallel Programming Languages: The Salishan Problems.* Elsevier Science Publishers, Holland, 1992.

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

[6] Manish Gupta and Prithviraj Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proc. of the 6th Distributed Memory Computing Conference*, pages 43–50, Portland, Oregon, April 28 – May 1, 1991.

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

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

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

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

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

[12] Kathleen Knobe and Joan D. Lukas. Data optimization and its effect on communication costs in MIMD Fortran code. In *Fifth SIAM Conference on Parallel Processing in Scientific Computing*, Houston, TX, March 1991.

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

[14] MasPar Computer Corporation. *MasPar Parallel Application Language (MPL) Reference Manual*, September 1990.

[15] 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.

[16] Michael Philippsen and Walter F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991*, pages 169–183. Springer Verlag, Lecture Notes in Computer Science 591, 1992.

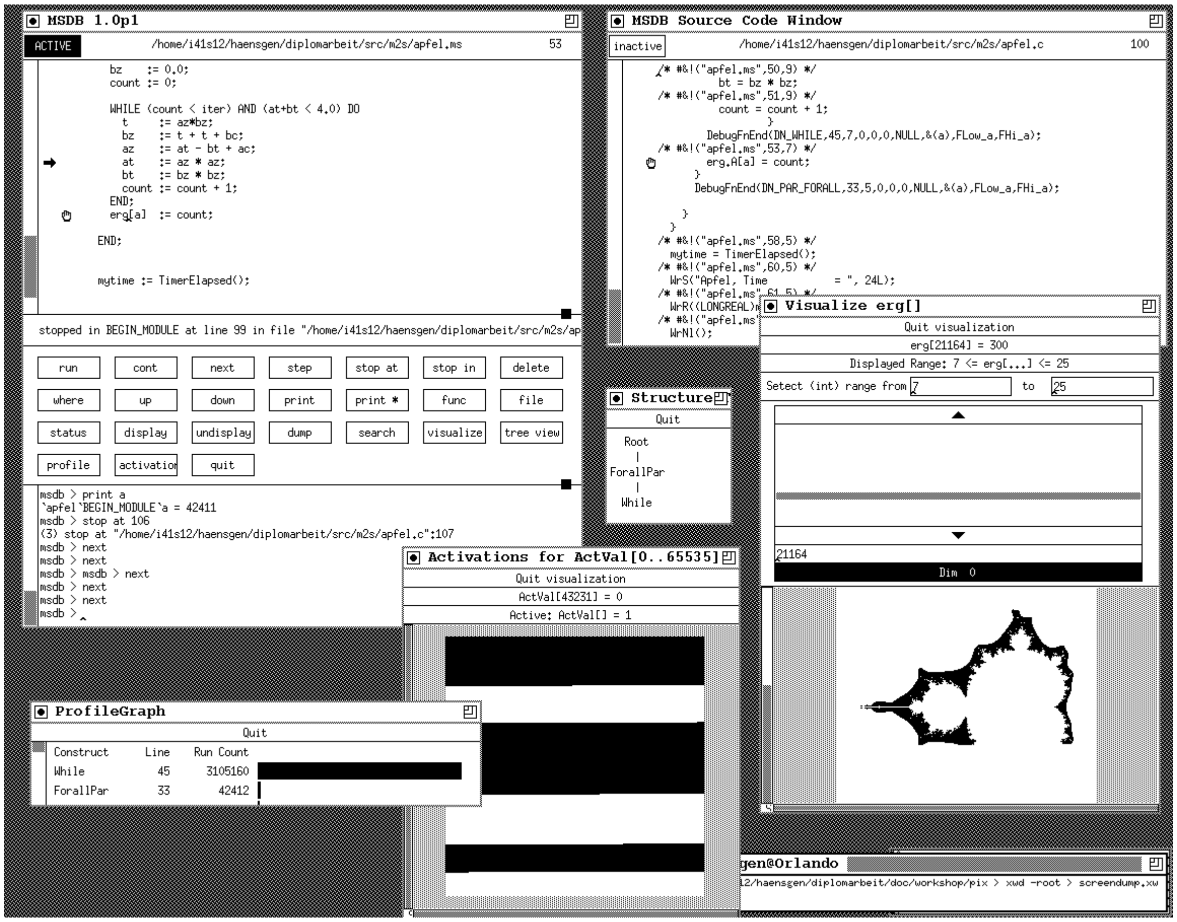
[17] J. Ramanujam and P. Sadayappan. Access based data decomposition for distributed memory machines. In *Proc. of the 6th Distributed Memory Computing Conference*, pages 196–199, Portland, Oregon, April 28 – May 1, 1991.

[18] Guy L. Steele. High Performance Fortran: Status report. In *Proc. of the 1993 Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, pages 1–4, Boulder, CO, September 30 – October 2, 1992, January 1993. ACM SIGPLAN Notices 28(1).

[19] 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, Dept. of Informatics, January 1990.

[20] Walter F. Tichy, Michael Philippsen, Ernst A. Heinz, and Paul Lukowicz. From Modula-2* to efficient parallel code. Technical Report No. 21/92, University of Karlsruhe, Dept. of Informatics, August 1992.

[21] Skef Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers.* PhD thesis, Carnegie-Mellon University, Dept. of Computer Science, Pittsburg, PA, May 1991.



```
/home/i41s12/haensgen/diplomarbeit/src/m2s/apfel.ms                53
          bz    := 0.0;
          count := 0;

          WHILE (count < iter) AND (at+bt < 4.0) DO
            t     := az*bz;
            bz    := t + t + bc;
            az    := at - bt + ac;
            at    := az * az;
            bt    := bz * bz;
            count := count + 1;
          END;
          erg[a]  := count;

        END;


        mytime := TimerElapsed();
```

```
stopped in BEGIN_MODULE at line 99 in file "/home/i41s12/haensgen/diplomarbeit/src/m2s/ap
```

```
msdb > print a
`apfel`BEGIN_MODULE`a = 42411
msdb > stop at 106
(3) stop at "/home/i41s12/haensgen/diplomarbeit/src/m2s/apfel.c":107
msdb > next
msdb > next
msdb > msdb > next
msdb > next
msdb > next
msdb >
```

```
/home/i41s12/haensgen/diplomarbeit/src/m2s/apfel.c                100
    /* #&!("apfel.ms",50,9) */
             bt = bz * bz;
    /* #&!("apfel.ms",51,9) */
             count = count + 1;
                     }
           DebugFnEnd(DN_WHILE,45,7,0,0,0,NULL,&(a),FLow_a,FHi_a);
    /* #&!("apfel.ms",53,7) */
           erg.A[a] = count;
         }
         DebugFnEnd(DN_PAR_FORALL,33,5,0,0,0,NULL,&(a),FLow_a,FHi_a);

       }
     }
    /* #&!("apfel.ms",58,5) */
      mytime = TimerElapsed();
    /* #&!("apfel.ms",60,5) */
      WrS("Apfel, Time            = ", 24L);
    /* #&!("apfel.ms",61,5) */
      WrR((LONGREAL)m
    /* #&!("apfel.ms"
      WrNl();
```

```
ActVal[43231] = 0
Active: ActVal[] = 1
```

```
erg[21164] = 300
Displayed Range: 7 <= erg[...] <= 25
Setect (int) range from 7   to 25
21164
```

Construct	Line	Run Count
While	45	3105160
ForallPar	33	42412

```
12/haensgen/diplomarbeit/doc/workshop/pix > xwd -root > screendump.xw
```

Figure 1: The MSDB environment in action. The program being debugged is a simple fractal computation, the (partial) results of which are shown in a range visualizer. A visualization of the processes as well as the (grouped) dynamic activation tree and a profile graph can also be seen. The left window shows the Modula-2* source with a program counter and a breakpoint, the right window displays the corresponding C source.