This paper appeared 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

Modula-2* and its Compilation

Michael Philippsen and Walter F. Tichy

Universität Karlsruhe
email: philippsen@ira.uka.de

Abstract

Modula-2*, an extension of Modula-2, is a programming language for writing highly parallel programs in a machine-independent, problem-oriented way. The novel attributes of Modula-2* are that programs are independent of the number of processors, independent of whether memory is shared or distributed, and independent of the control modes (SIMD or MIMD) of a parallel machine.

This article briefly describes Modula-2* and discusses its major advantages over the data-parallel programming model. We also present the principles of translating Modula-2* programs to MIMD and SIMD machines and discuss the lessons learned from our first compiler, targeting the Connection Machine. We conclude with important architectural principles required of parallel computers to allow for efficient, compiled programs.

1 Introduction

Highly parallel machines with thousands and tens of thousands of processors are now being manufactured and used commercially. These machines are of rapidly growing importance for high-speed computation. They have also initiated a major shift within Computer Science from the sequential to the parallel computer. One of the major problems we face in the use of these new machines is programmability: How to write, with no more than ordinary effort, programs that bring the raw power of a parallel computer to bear on a problem.

Two major approaches to the programming problem can be distinguished: The first is to automatically parallelize sequential software. Although there is overwhelming economic justification for it, this approach will meet with only limited success in the short to medium term (see, for instance, [18]). The goal of automatically producing parallel programs can only, if ever, be achieved by program transformations that start with the problem specification and not with a sequential implementation. In a sequential program, too many opportunities for parallelism have been hidden or eliminated.

The second approach is to write programs that are explicitly parallel. We claim that only minor extensions of existing programming languages are required to express highly parallel programs. Thus, programmers will need only moderate additional training, mainly in the area of parallel algorithms and their analysis. This area, fortunately, is well developed; see for instance textbooks [1] and [5]. In compiler technology, however, new techniques must be found to map machine-independent programs to existing architectures, while at the same time parallel machine architecture must evolve to efficiently support the features that are required for problem-oriented programming styles.

We take the approach of expressing parallelism explicitly, but in a machine-independent way. In section 2 we analyze the problems that plague most parallel programming languages today. Section 3 then presents Modula-2*, an extension of Modula-2 [20], for the explicit formulation of highly parallel programs. The extension is small and easy to learn, but provides a programming model that is far more general and machine independent than other proposals. Next, we discuss compilation techniques for targeting MIMD and SIMD machines and report on experience with our first Modula-2* compiler [8] for the Connection Machine. We conclude with properties of parallel machine architectures that would improve the efficiency of high-level parallel programs.

2 Related Work

Most current programming languages for parallel and highly parallel machines, including *LISP, C*, MPL, VAL, Sisal, Occam, Ada, FORTRAN90, Blaze, Dino, and Kali [10, 9, 14, 19, 2, 12, 11, 15, 7] suffer from some or all of the following problems:

- Whereas the number of processors of a parallel machine is fixed, the problem size is not. Because most of the known parallel languages do not support the virtual processor concept, the programmer has to write explicit mappings for adapting the process structure of each program to the available processors. This is not only a tedious and repetitive task, but also one that makes programs non-portable.

- Co-locating data with the processors that operate upon the data is critical for the performance of distributed memory machines. Poor co-location results in high communication costs and poor performance. Good co-location is highly dependent on the topology of the communication network and must, at present, be programmed by hand. It is a primary source of machine dependence.

- All parallel machines provide facilities for interprocess communication; most of them by means of a message passing system. Nearly all parallel languages support only low level *send* and *get* communication commands. Programming communication with these primitives, especially if only nearest neighbor communication is available, is a time consuming and error prone task.

- There are several control modes for parallel machines, including MIMD, SIMD, data-flow, and systolic modes. Any extant, parallel language targets exactly one of those control modes. Whatever the choice, it severely limits portability as well as the space of solutions.

Modula-2* provides solutions to the basic problems mentioned above. The language abstracts from the memory organization and from the number of physical processors. Mapping of data to processors is performed by the compiler, optionally supported by high-level directives provided by the programmer. Communication is not directly visible. Instead, reading and writing in a (virtually) shared address space subsumes communication. A shared memory, however, is not required. Parallelism is explicit, and the programmer can choose among synchronous and asynchronous execution mode at any level of granularity. Thus, programs can use SIMD-mode where proper synchronization is difficult, or use MIMD-mode where synchronization is simple or infrequent. The two modes can even be intermixed freely.

The data-parallel approach, discussed in [6] and exemplified in languages such as *LISP, C*, and MPL is currently quite successful, because it has reduced machine dependence of parallel programs. Data-parallelism extends a synchronous, SIMD model with a global name space, which obviates the need for explicit message passing between processing elements. It also makes the number of (virtual) processing elements a function of the problem size, rather than a function of the target machine.

The data-parallel approach has three major advantages: (1) It is a natural extension of sequential programming. The only parallel instruction, a synchronous **forall** statement, is a simple extension of the well known **for** statement and is easy to understand. (2) Debugging data-parallel programs is not much more difficult than debugging sequential programs. The reason is that there is only a single locus of control, which dramatically simplifies the state space of a program compared to that of an MIMD program with thousands of independent loci of control. (3) There is a wide range of data-parallel algorithms. Most parallel algorithms in textbooks are data-parallel (compare for instance [1, 5]). According to Fox [4], more than 80% of the 84 existing, parallel applications he examined fall in the class of synchronous, data-parallel programs. Furthermore, systolic algorithms as well as vector-algorithms are special cases of data-parallel algorithms.

But data-parallelism, at least as defined by current languages, has some drawbacks: (1) It is a synchronous model. Even if the problem is not amenable to a synchronous solution, there is no escape. In particular, parallel programs that interact with stochastic events are awkward to write and run inefficiently. (2) There is no nested parallelism. This means that once a parallel activity has started, the involved processes cannot start up additional, parallel activity. A parallel operation simply cannot expand itself and involve more processes. This property seriously limits parallel searches in irregular search spaces, for example. The effect is that data-parallel programs are strictly bimodal: They alternate between a sequential and a parallel mode, where the maximal degree of parallelism is fixed once the parallel mode is entered. To change the degree of parallelism, the program first has to stop all parallel activity and return to the sequential mode. (3) The use of procedures to structure a parallel program in a top-down fashion is severely limited. The problem here is that it is not possible to call a procedure in parallel mode, when the procedure itself invokes parallel operations (this is a consequence of (2)). Procedures cannot allocate local data and spawn data parallel operations on it, unless they are called from a sequential program. Thus, procedures can only be used in about

half of the cases where they would be desirable. They also force the use of global data structures on the programmer.

When designing Modula-2*, we wanted to preserve the main advantages of data-parallel languages while avoiding the above drawbacks. The following list contains the main advances of Modula-2* over data-parallel languages.

- The programming model of Modula-2* is a strict superset of data-parallelism. It allows both synchronous and asynchronous parallel programs.
- Modula-2* is problem-oriented in the sense that the programmer can choose the degree of parallelism and mix the control mode (SIMD-like or MIMD-like) as needed by the intended algorithm.
- Parallelism may be nested at any level.
- Procedures may be called from sequential or parallel contexts and can generate parallel activity without any restrictions.
- Modula-2* is translatable effectively for both SIMD and MIMD architectures.

3 The Language Modula-2*

Modula-2 has been chosen as a base for a parallel language because of its simplicity. There are no reasons why similar extensions could not be added to other imperative languages such as FORTRAN or ADA. The necessary extensions were surprisingly small. They consist of synchronous and asynchronous versions of a **forall** statement, plus simple, optional declarations for mapping array data onto processors in a machine independent fashion. An interconnection network is not directly visible in the language. We assume a shared address space among all processors, though not necessarily shared memory. There are no explicit message passing instructions; instead, reading and writing locations in shared address space subsume message passing. This approach simplifies programming dramatically and assures network independence of programs. The burden of distinguishing between local and non-local references, and substituting explicit message passing code for the latter, is placed on an (optimizing) compiler. The programmer can influence the distribution of data with a few, simple declarations, but these are only hints to the compiler with no effect on the semantics of the program whatsoever.

3.1 Overview of the forall statement

The **forall** statement creates a set of processes that execute in parallel. In the asynchronous form, the individual processes operate concurrently, and are joined at the end of the **forall** statement. The asynchronous **forall** simply terminates when the last of the created processes terminates. In the synchronous form, the processes created by the **forall** operate in unison until they reach a branch point, such as an **if** or **case** statement. At branch points, the set of processes partitions into two or more subsets. Processes within a single subset continue to operate in unison, but the subsets are not synchronized with respect to each other. Thus, the union of the subsets operate in MSIMD[1] mode. A statement causing a partition into subsets terminates when all its subsets terminate, at which point the subsets rejoin to continue with the following statement.

Variants of both the synchronous and asynchronous form of the **forall** statement have been introduced by previously proposed languages, such as Blaze, C*, Occam, Sisal, VAL, *LISP [11, 17, 14, 9, 10, 16] and others [3]. Note also that vector instructions are simple instances of the synchronous **forall**.

None of the languages mentioned above include *both* forms of the **forall** statement, even though both are necessary for writing readable and portable parallel programs. The synchronous form is often easier to handle than the asynchronous form, because it avoids synchronization hazards. However, the synchronous form may be overly constraining and may lead to poor machine utilization. The combination of synchronous and asynchronous forms in Modula-2* actually permits the full range of parallel programming styles between SIMD and MIMD.

The syntax of the **forall** is as follows.[2]

```
FORALL ident ":" SimpleType IN (PARALLEL | SYNC)
        StatementSequence
END.
```

The identifier introduced by the forall statement is local to the statement and serves as a run-time constant for every process created by the **forall**. *SimpleType* is an enumeration or a (sub-)range. The **forall** creates as many processes as there are elements in *SimpleType* and initializes the run-time constant

[1] MSIMD: Multiple SIMD. Few but more than one instruction streams operate on many data streams. A compromise between SIMD and MIMD.

[2] We use the EBNF syntax notation of the Modula-2 language definition, with keywords in upper case, | denoting alternation, [...] optionality, and (...) grouping of the enclosed sentential forms.

of each process to a unique value in *SimpleType*. The created processes all execute the statements in *StatementSequence*.

3.2 The asynchronous forall

The created processes execute *StatementSequence* concurrently, without any implicit, intermediate synchronization. The execution of the **forall** terminates when all created processes have finished. Thus, the asynchronous **forall** contains only one synchronization point at the end. Any additional synchronization must be programmed explicitly with semaphores and the operations *WAIT* and *SIGNAL*.

In the following example, an asynchronous **forall** statement implements a vector addition.

```
FORALL i:[0..N-1] IN PARALLEL
  z[i] := x[i] + y[i]
END
```

Since no two processes created by the **forall** access the same vector element, no temporal ordering of the processes is necessary. The N processes may execute at whatever speed. The **forall** terminates when all processes created by it have terminated.

A more complicated example, illustrating recursive process creation, is the following. Procedure *ParSearch* searches a directed, possibly cyclic graph in parallel fashion. It can best be understood by comparing it with depth-first-search, except that *ParSearch* runs in parallel. It starts with a root of the graph and visits nodes in the graph in a parallel (and largely unpredictable) fashion.

```
PROCEDURE ParSearch( v: NodePtr );
BEGIN
  IF Marked( v ) THEN RETURN END;
  FORALL s:[0..v^.successors-1] IN PARALLEL
    ParSearch( succ(v, s) )
  END;
  visit( v );
END ParSearch;
```

The procedure *ParSearch* simply creates as many processes as a given node has successors, and starts each process with an instance of *ParSearch*. Before visiting a node, *ParSearch* has to test whether the node has already been visited and marked. Since multiple processes may reach the same node simultaneously, testing and setting the mark is done in a critical section (implemented with a semaphore associated with each node) by the procedure *Marked*. If the graph is a tree, no marking is necessary.

3.3 The synchronous forall

The processes created by a synchronous **forall** execute every single statement of *StatementSequence* in unison. To illustrate this mode, its semantics for selected statements is described in some detail below:

- A statement sequence is executed in unison by executing all its statements in order and in unison.
- In the case of branching statements such as `IF C THEN SS1 ELSE SS2 END`, the set of participating processes divides into *disjoint* and *independently operating* subsets, each of which executes one of the branches (`SS1` and `SS2` in the example) in unison. Note that in contrast to other data-parallel languages, *no* assumption about the relative speeds or relative order of the branches may be made. The execution of the entire statement terminates when all processes of all subsets have finished.
- In the case of loop statements such as `WHILE C DO SS END`, the set of processes for any iteration divides into two disjoint subsets, namely the *active* and the *inactive* ones (with respect to the loop statement). Initially, all processes entering the loop are active. Every iteration starts with the synchronous evaluation of the loop condition `C` by all active processes. The processes for which `C` evaluates to `FALSE` become inactive. The rest forms the active subset which executes statement sequence `SS` in unison. The execution of the whole loop statement terminates when the subset of active processes becomes empty.

Hence, synchronous parallel operation closely resembles the lock-step operation of SIMD machines with an important generalization for parallel branches.

As an example, consider the computation of all postfix sums of a vector V of length N. The program should place into $V[i]$ the sum of all elements $V[i] \ldots V[N-1]$. A recursive doubling technique as in reference [6] computes all postfix sums in $O(\log N)$ time, where N is the length of the vector.

Figure 1 illustrates the process. The program operates by computing partial sums of length $s = 2^j$, where j counts the iterations. The inner **forall** creates N processes. Note that there is a one-to-one mapping between process numbers and elements of the vector. In each iteration, the length of the partial sums is doubled by parallel summation of neighboring sums. The **if** statement inside the **forall** disables all processes that must not participate in the computation during a given iteration.

```
VAR V : ARRAY[0 .. N-1] OF REAL;
VAR s : CARDINAL;
BEGIN
   s := 1;
   WHILE s < N DO
      FORALL i:[0..N-1] IN SYNC
         IF (i+s)<N THEN
            V[i]:= V[i]+V[i+s]
         END
      END;
      s := s * 2
   END
END
```

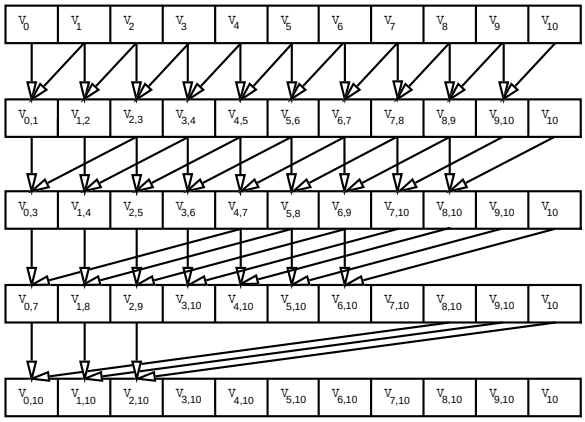

Figure 1: Computing postfix sums of a vector

3.4 Allocation of array data

Co-location of data with the processors that access the data is important for parallel machines without uniform access time to memory locations. Poor alignment of data and processors may cause excessive communication overhead. We therefore provide a simple, machine-independent construct for controlling the allocation of array data. This construct is optional and does not change the meaning of a program; it affects only performance. A compiler for a machine with uniform memory access time may ignore the construct.

The allocation of array data to processors is controlled with one allocator per dimension. The modified declaration syntax for arrays is as follows:

ArrayType = `ARRAY` SimpleType [allocator]
{"," SimpleType [allocator]} `OF` type.
allocator = `LOCAL` | `SPREAD` | `CYCLE` | `RANDOM` |
`SBLOCK` | `CBLOCK`.

Array elements whose indices differ only in dimensions that are marked `LOCAL` are associated with the same processor. This facility is used to avoid distribution of data in a given dimension.

Dimensions with allocator `SPREAD` are divided into segments, one for each of the available processors. A vector with n elements is assigned to P processors by allocating a segment of length $\lceil n/P \rceil$ to each processor. While utilizing all available processors, it minimizes the cost of nearest-neighbor communication.

Dimensions with allocator `CYCLE` are distributed in a round-robin fashion over the available processors. Given P processors, the elements of a vector whose indices are identical modulo P are associated with the same processor. In contrast to `SPREAD`, `CYCLE` maximizes the cost of nearest-neighbor communication: neighboring array elements are always in different processors, leading to better processor utilization if a parallel algorithm operates on subsegments of a vector at a time.

Dimensions with allocator `RANDOM` are distributed randomly over the available processors. In contrast to `CYCLE`, `RANDOM` leads to a better processor utilization if a parallel algorithms accesses the dimension in a random pattern.

If either `SPREAD`, `CYCLE`, or `RANDOM` apply to several successive dimensions, then these dimensions are "unrolled" into one pseudo-vector with a length that is the product of the lengths of the individual dimensions. This scheme idles fewer processors than applying `SPREAD`, `CYCLE`, or `RANDOM` to individual dimensions.

Allocators `SBLOCK` and `CBLOCK` apply `SPREAD` and `CYCLE` resp. to each dimension individually. For two successive dimensions, `SBLOCK` has the effect of creating rectangular subarrays and assigning those to the available processors. With this arrangement, nearest-neighbor communication in all dimensions is best supported when the interconnection network can be configured into the same number of dimensions as the arrays.

`CBLOCK` for two dimensions also creates two-dimensional subarrays, but the rows and columns of these subarrays are then distributed independently in a round-robin fashion over the processor grid. Again, `SBLOCK` minimizes nearest-neighbor communication, while `CBLOCK` allows high processor utilization if smaller subarrays are processed in parallel.

4 Implementing Modula-2*

When discussing the principles of compiling Modula-2*, we first present the more difficult case of compiling for MIMD, and then introduce the simplifications for SIMD. The latter have actually been implemented in our CM compiler. Although this first

compiler does not contain sophisticated optimization, it helped us in understanding the main sources of optimizations when compiling for massively parallel machines.

4.1 **Asynchronous** forall

4.1.1 **Asynchronous** forall**, MIMD implementation**

A straightforward approach to implementing the **forall** is to create the required number of lightweight processes, or *threads*. Threads of a **forall** share the same program (the enclosed statement sequence), but have their own stacks. The resulting run-time structure is a tree of stacks.

On a distributed memory machine, it pays to replicate the entire program to all processors. Code replication can be accomplished quickly during startup, either using a broadcast facility or recursive doubling.

The main problems when compiling **forall** statements are thread creation, thread termination, and load balancing. All of these problems must be solved in a parallel fashion. Sequential implementations would cause a serious bottleneck, and, for algorithms with fine granularity, result in essentially sequential programs. Note also that **forall**s may be nested, so there may be several new sets of threads being created simultaneously.

The process reaching a **forall**, called the *spawner*, must create the set of threads prescribed by the **forall**. The spawner actually creates only the initial thread of the **forall**, called the *leader*. (A simple optimization is to let the spawner take on the role of the leader.) The leader then replicates itself. All threads created keep replicating themselves again and again, until the required number is obtained. This method is another variant of recursive doubling. A small number of parameters controls the replication. When a thread has replicated itself a sufficient number of times, it simply jumps to the beginning of the **forall**'s code sequence and begins execution.

Synchronization and thread termination at the **forall**'s end follow the same pattern. Each thread has a semaphore for receiving termination signals from other threads. A thread that reaches the end of its **forall** first waits for termination signals from all the threads it spawned during the replication process, then signals its creating thread, and destroys itself. If all n threads of a **forall** terminate at about the same time, then the leader learns about the combined termination in time proportional to $O(\log n)$, signals the spawner, and kills itself (or simply resumes the role of the spawner).

The problem of load balancing is to distribute threads over the available processors so that (1) the load on the processors is equalized, (2) the threads are co-located with their data, and (3) co-scheduling of threads within the same **forall**-instance becomes possible. Again, a centralized solution must be avoided. One possibility is for processors to keep a running total of ready processes and the overall average. The overall average can be updated periodically (say at the end of a time-slice) by another recursive doubling technique in which all processors participate. Newly created threads are moved between neighboring processors depending on the current load in comparison to the average. Under certain circumstances, migrating a long-running thread (including its data) to another processor may be advantageous. In addition, static compiler analysis can indicate preferred processors for co-locating data and threads.

Co-scheduling of threads in the same **forall** is necessary to avoid delays inherent in context swaps when the threads communicate. Without co-scheduling, communicating threads may enter a situation where they execute alternatingly or in co-routine fashion instead of in parallel [13]. Co-scheduling can be accomplished by increasing the thread priority with the nesting depth of **forall**s, or by providing special mechanisms for "task forces", i.e., for scheduling groups of threads simultaneously.

Obviously, thread creation, termination, and load balancing must be as fast as possible. Various optimizations for bulk thread generation are feasible, but will not be discussed here for lack of space.

The above techniques have not been implemented in our first compiler, since the CM is a SIMD machine. However, work has started on a Modula-2* compiler targeting a Transputer cluster, where the techniques will be used. We are also exploring special hardware facilities to speed up these tasks.

4.1.2 **Asynchronous** forall**, SIMD implementation**

The synchronous nature of a SIMD machine, coupled with the broadcast bus from the front-end, makes all three of thread creation, termination, and load balancing operate in constant time or nearly constant time. For generality, assume nested **forall**s: m threads each execute a **forall** statement, each creating n new threads. Thus, the number of threads to be created is $t = nm$. If n is not uniform for all the m spawners, then a $O(\log m)$-time summation instead of (constant-time) multiplication must be performed to

compute t.

Once t is known, t stacks are created by assigning to each of p processors a segment of $\lceil t \div p \rceil$ stacks. This operations takes constant time and balances the load perfectly. Process termination also takes constant time, since there is no synchronization overhead. However, it may be necessary to provide each thread with some initial data (such as its number) during creation. Spreading this information takes again logarithmic time, but as demonstrated by the Connection Machine, special instructions for spreading data are so fast that, in practice, they can be regarded as constant.

What remains to be discussed is the scheduling of instructions. Since the asynchronous **forall** prescribes no scheduling of the threads at all, the compiler writer can choose one that works well on a given SIMD or MSIMD machine. We describe briefly the implementation we chose for the Connection Machine (CM). We assume initially, that the number of available processors equals the number of threads.

Activity Bits. The central idea of control flow on SIMD computers is deactivation and reactivation of processors, controlled by an activity bit associated with each processor. When the activity bit is off, the processor does not execute the instructions issued by the front-end. This facility is sufficient for simulating the usual control flow constructs in a parallel context. All that is needed is a stack of activity bits for each thread. The top of each activity stack is stored into the activity bit of a processor. Suitable manipulation of the activity bits turns threads on and off, as required by the instruction stream issuing from the front-end.

There are two small extensions of the usual control flow mechanism for SIMD machines. They are needed for recursion and for **exit** and **return** statements. First, consider parallel loops (i.e., loops within a **forall**). On a SIMD-machine, the front-end repeatedly issues the instructions for the loop body, until the termination conditions of all threads executing the loop are met. The usual technique is to evaluate a thread's termination condition directly into its activity bit. Before each iteration, the front-end tests whether there are any positive activity bits left. If not, the loop terminates. An **exit** statement may also terminate a loop, by turning off the activity bit of the corresponding thread. However, since an **exit** statement may be nested several levels deep within a loop, it must not only set the topmost activity bit to false, but all those that have been stacked since the last loop was entered. Similar considerations apply to the **return** statement.

Consider the following example.

```
FORALL i:[0..N-1] IN PARALLEL
    LOOP
        IF ODD(i) THEN EXIT END;
        SS
    END
END
```

When control flow reaches the **exit**, then two activity bits have been stacked for each thread: one for the **loop**, and one for the **if** statement. To prevent a thread that has already executed the **exit** from being reactivated after the **if**, its top *two* activity bits must be set to `FALSE`.

Recursion termination is similar to loop termination. If a recursive call occurs inside a parallel **if** or **case**, then the front-end must sense whether there is any active thread left in a branch. If not, then the branch terminates. Without this provision, unbounded recursion would ensue.

Parallel Procedure Call. Because procedures can be called from both sequential and parallel contexts, each procedure must be compiled twice: Once for executing entirely on the front-end in sequential mode and a second time for executing within a **forall** statement. The difference is that in the parallel version, the procedure call and return instructions are executed only on the front-end. Thus, we need two types of stacks: On the front-end, we stack return addresses. On the stacks associated with the parallel threads, we store parameters and local data. This division is a direct consequence of SIMD and would even occur if front-end and parallel processors had the same instruction set. On the CM, the instruction sets differ, and so the sequential and parallel versions are completely different.

Our compiler relies on a minor language restriction: Procedures may not be nested within each other. The reason is that up-level addressing is quite expensive. Since it is in general unpredictable in what context a procedure is called, each memory access would have to distinguish at run-time whether it references data on the front-end or the parallel processors.

Processor Virtualization. Simulating more threads than there are processors available is called *processor virtualization*. In SIMD mode, it is not possible to simply create new processes on demand and let the operating system schedule them. Instead, the

front-end has to issue the instructions implementing the body of a **forall** in a loop. The number of iterations of this loop is given by the ratio of threads to available processors.

The PARIS instruction set of the CM provides automatic processor virtualization. This means that processor virtualization is transparent to the programmer. The firmware simulates as many threads as required. The maximum number of threads is only limited by the available memory, because the local memory of each processor must be shared out among the assigned threads.

Our Modula-2* compiler uses the automatic processor virtualization. However, this virtualization is quite expensive. The main reason is that the virtualization actually implements synchronous virtualization, which requires many temporary variables. In essence, this virtualization wraps every single instruction into a virtualizing loop, even though a loop around the entire body of a **forall** would suffice (since the asynchronous **forall** prescribes no scheduling of threads). The latter simulation would be obviously much more efficient.

4.2 **Synchronous** forall

4.2.1 **Synchronous** forall**, MIMD implementation**

The synchronous **forall** requires many more synchronization points than the asynchronous form. There must be a synchronization point between every two statements inside a **forall**, and in the case of the assignment, even within a single statement. A parallel assignment of the from `L := R` means that the value of `R` is evaluated synchronously and stored in a temporary. Similarly, the address represented by `L` is evaluated synchronously and stored in a temporary. Only after both of these parallel evaluations have completed can the assignment be made. Otherwise, interference is possible, as in the assignment `A[i] := A[i+1]`.

A synchronization point is implemented with a scheme similar to the one used to terminate an asynchronous **forall**, except that now the threads do not terminate, but wait for a signal to proceed. First, a logarithmic reduction informs the leader that all threads in the process have reached the synchronization point. Then a logarithmic doubling process sends signals back out to the threads to continue.

Clearly, synchronization points are expensive. We are currently investigating methods to eliminate them where possible. For instance, the synchronization point inside an assignment is not necessary if the left and right hand sides do not interfere. Furthermore, by scheduling processes in a certain fashion, the overlaps may be reduced greatly. Even synchronization points between statements can be eliminated if there are no dependencies. Much of the dependency analysis developed for parallelizing compilers applies here.

4.2.2 **Synchronous** forall**, SIMD implementation**

The SIMD implementation of the synchronous **forall** was simple on the CM: the built-in virtualization does the job. However, this virtualization cannot take advantage of the optimizations described above. Instead, it must make conservative assumptions. The resulting virtualization is far from efficient. An optimizing compiler could produce a much faster virtualization in the majority of cases. Consider the following example.

```
FORALL i: [0..N-1] IN SYNC
    A[i] := (A[i] + 1) / 2
END
```

Below are two possible virtualizations on p processors, expressed in Modula-2*.

```
s := CEILING(N, p)
FORALL j : [0 .. p-1] IN PARALLEL
    FOR i:= j*s TO MIN((j+1)*s,N)-1
    DO
        TMP[i] := A[i] + 1;
        TMP[i] := TMP[i] / 2
    END
END
FORALL j : [0 .. p-1] IN PARALLEL
    FOR i:= j*s TO MIN((j+1)*s,N)-1
    DO
        A[i]   := TMP[i]
    END
END
```

```
s := CEILING(N, p)
FORALL j : [0 .. p-1] IN PARALLEL
    FOR i:= j*s TO MIN((j+1)*s,N)-1
    DO
        reg := A[i];
        reg := reg + 1;
        reg := reg / 2;
        A[i]:= reg
    END
END
```

The program on the left shows the conservative virtualization, as performed by PARIS. The optimized version on the right hand side exploits the fact that only one temporary location is required. By using a single register for it on every processor, the number of writes to memory are reduced to one third of the unoptimized version. Furthermore, no synchronization is necessary. On a SIMD machine, this means that the two loops can be merged; on a MIMD machine,

we save the synchronization point. Furthermore, if the individual processors have a vector capability, the computation in each processor can even be interleaved.

While implementing the synchronous **forall** for the CM we have identified the main sources of optimization in compiling for massively parallel machines. We have started to include these optimizations in the next compilers for MasPar, CM, and Transputer, including the necessary data-dependence analysis.

5 Recommendations for Parallel Machine Architectures

The following list itemizes some broad requirements that parallel machine architectures should fulfill to allow for efficient, compiled programs. These requirements are likely to be encountered when designing the translation schemes for parallel, imperative languages.

- Hardware support for fast process creation and synchronization.
- Shared address space. All processors should be able to generate addresses for the entire memory on the system. In particular, the front-end's memory should be part of that address space. A source of great difficulty in our compiler were the many different types of addresses. The compiler has to distinguish between local addresses, global addresses, addresses in the front-end, general communication addresses, and communication addresses on a grid. Optimizing for all these cases is often impossible, even with detailed interprocedural analysis. Furthermore, parallel pointers are quite expensive to implement without a shared address space – one basically has to simulate the shared address space in software.[3]
- Uniform communication mechanism. Most parallel machines today provide a set of instructions for accessing local memory, a second one for accessing memory in direct neighbors, and a third set for accessing distant memory units. The differences in speed are significant and therefore require that the compiler detect the faster cases. However, it is often impossible to know statically for which case to optimize. For instance, we found that in most cases it was impossible to determine in the compiler whether a procedure would access local or non-local memory. The generated code thus has to check all three cases at run-time. Such a simple and frequently repeated case analysis could be done much more efficiently in hardware.
- Autonomous addressing capability. An autonomous addressing capability means that each processor can generate its own address for accessing memory. The Connection Machine does not have such a facility – on the CM, each processor must use the same address. The lack of autonomous addressing not only makes many applications awkward to write, but also precludes certain optimizations in processor virtualization.
- Single instruction set. SIMD machines today typically have different instruction sets for front-end and parallel processors. This property implies that the code generator of the compiler has to be written twice. Also, each procedure has to be translated twice, doubling code size. A speed differential between front-end and parallel processors, however, does not appear to be a major problem.
- Small instruction set. The CM offers about 400 PARIS instructions, only a few of which a compiler can actually generate. A study determining the most frequently used instructions in parallel programs is sorely needed.

[3] A shared address space does not imply shared memory.

6 Conclusion

Ease of programming as well as portability of programs will be of overwhelming importance for the acceptance of highly parallel machines. Modula-2* supports both: few extensions of a sequential programming language suffice for writing highly parallel, problem-oriented programs, and compilers that can generate efficient code for a wide range of parallel machines appear feasible. Improvements in hardware architecture, operating systems, programming languages and compiler technology should eventually render the current practice of machine dependent, parallel programming as obsolete as machine dependent, sequential programming.

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] Henry E. Bal, Jennifer S. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[4] Geoffrey C. Fox. What have we learnt from using real parallel machines to solve real problems? In *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications*, volume 2, pages 897–955, Pasadena, CA, 1988. ACM Press, New York.

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

[6] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[7] Charles Koelbel and Piyush Mehrotra. Supporting shared data structures on distributed memory architectures. In *Proc. of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 177–186, March 1990.

[8] Ralf Kretzschmar. Ein Modula-2*-Compiler für die Connection Machine CM-2. Master's thesis, University of Karlsruhe, Department of Informatics, May 1991.

[9] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL Language Reference Manual*. Lawrence Livermore National Laboratory, March 1985.

[10] James R. McGraw. The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.

[11] Piyush Mehrotra and John Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, November 1987.

[12] Michael Metcalf and John Reid. *Fortran 90 Explained*. Oxford Science Publications, 1990.

[13] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu. Medusa: An experiment in distributed operating system structure. *Communications of the ACM*, 23(2):92–205, February 1980.

[14] Prentice Hall, Englewood Cliffs, New Jersey. *INMOS Limited: Occam Programming Manual*, 1984.

[15] M. Rosing, R. Schnabel, and R. Weaver. DINO: Summary and example. In *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 472–481, Pasadena, CA, 1988. ACM Press, New York.

[16] Thinking Machines Corporation, Cambridge, Massachusetts. **Lisp Reference Manual, Version 5.0*, 1988.

[17] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Programming Guide, Version 6.0*, November 1990.

[18] Walter F. Tichy. Parallel matrix multiplication on the Connection Machine. *International Journal of High Speed Computing*, 1(2):247–262, 1989.

[19] U.S. Government, Ada Joint Program Office. *ANSI/MIL-Std 1815 A, Reference Manual for the Ada Programming Language*, January 1983.

[20] Niklaus Wirth. *Programming in Modula-2 (Third corrected Edition)*. Springer-Verlag Berlin, Heidelberg, New York, 1985.