Delft University of Technology
Parallel and Distributed Systems Report Series

Code generation techniques for the task-parallel programming language *Spar*

Frits Kuijlman, Kees van Reeuwijk, Arjan J.C. van Gemund and Henk J. Sips

report number PDS-1998-006

PDS

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Section
Faculty of Information Technology and Systems
Department of Technical Mathematics and Informatics
Delft University of Technology
Zuidplantsoen 4
2628 BZ Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
`reports@pds.twi.tudelft.nl`

Information about Parallel and Distributed Systems Section:
`http://pds.twi.tudelft.nl/`

Abstract

In this paper we describe a compilation scheme to translate implicitly parallel programs in the programming language Spar *(an extension to Java) to efficient code for distributed-memory parallel computer systems. The compilation scheme is formulated as a set of transformation rules.*

In Spar*, the language constructs for parallelization have been designed for comfortable use by the programmer, not for ease of compilation. Nevertheless, it is shown that the basic translation and optimization schemes can be implemented with a surprisingly small set of transformation rules. The paper discusses the systematic translation of* Spar *programs into parallel code by using these rules.*

1 Introduction

Issues that are of major importance in the design of parallel programming languages and associated compilation schemes for distributed-memory systems include:

- *parallelizability*, the extent to which parallelism can either be expressed by the user or deduced by the compiler such that problem parallelism is adequately captured.
- *programmability*, the extent to which the user is relieved from programming the complex synchronizations required between concurrently executing tasks in order to retain correctness.
- *portability*, the extent to which the program is machine-independent, most notably pertaining to the absence of code that defines the mapping of tasks and data onto machine resources.

Due to the inherently conflicting nature of these issues, existing parallel programming languages reflect various specific design trade-offs.

Explicit parallel programming interfaces such as offered by *Java* threads [8] and *CC++* [3], as well as by message-passing interfaces such as *MPI* [16] offer full parallelizability, yet at a significant expense of programmability and portability. Although for pragmatic reasons in high-performance computing explicit languages are still widely used, programming requires a detailed understanding of the complex interplay between program and machine, and is extremely laborious and error-prone.

In implicit (parallel) languages, parallelism is deduced by the compiler. For example, some compilers try to parallelize languages such as *Fortran* through compile-time dependence analysis, thus providing portability [13]. Due to the inherent limitations of this analysis, however, parallelizability is typically restricted to regular cases [10]. To alleviate these detection problems, language extensions such as in *Jade* [15] and *Fx* [9] have been proposed that aid the compiler in analyzing the program.

As a compromise, data parallel languages such as *HPF* [6] extend the implicit parallel paradigm by offering data mapping annotations that induce data parallelism. To express task based parallelism extensions such as *Fortran-M* [7] and *Opus* [4] have been proposed. Furthermore, because of the introduction of mapping directives, portability is significantly decreased.

In the design of *Spar* [19], a task parallel language that is developed within the *Automap* project [18], a different trade-off is made between parallelizability, programmability, and portability. As is illustrated by the project's name, the main emphasis is on entirely automatic mapping of (coarse grain) tasks and data by a scheduling engine, thus achieving maximum portability. The scheduling engine employs new low-cost compile-time, execute-time, and run-time scheduling algorithms aimed to deliver acceptable performance [14].

A compromise between parallelizability and programmability is achieved by defining an implicit programming interface. The above trade-offs maximize portability while minimizing programming effort with regard to coordinating parallelism. Yet, the fact that *Spar* is intended as a high-performance programming interface, implies that additional annotations should still be available to augment the code with information in order to optimize the compilation process when necessary. To enhance parallelizability, annotations are offered to supply additional information regarding (the absence of) dependencies and the grain size of the tasks. To enhance scheduling performance, mapping and/or alignment information of tasks and data can be specified by the programmer and passed to the compiler through annotations.

The *Spar* compiler generates SPMD target code. Although much work has previously been done on the efficient generation of SPMD data parallel programs [17], much less is the case for the efficient and systematic generation of SPMD programs with task parallel constructs. This paper shows how to systematically generate such programs.

The paper is organized as follows. Section 2 gives an introduction to *Spar* and describes the language constructs we intend to use to describe parallelism. Section 3 describes the virtual machine we use as a compilation target, and in Section 4 the compilation scheme is outlined. In Section 5 our *Timber* compilation system is briefly described, and Section 6 presents the annotations which are used to communicate between the different engines in the system. In Section 7 task mapping and scheduling are discussed, and Section 8 presents translation rules and examples of the code generation process.

2 An overview of *Spar*

Spar is a programming language that has been designed for scientific computations such as numerical computations, image processing, and other computationally intensive applications. The basic language constructs are taken from *Java*, and a number of new language constructs have been added to make the language more suitable for the intended application domain. Compared to *Java*, *Spar* provides additional language construct to:

- Support convenient parallel programming;
- Provide multidimensional arrays;
- Support array statements;
- Support sparse arrays and block arrays.

In this paper we discuss the parallelization of *Spar*, and only the language constructs for parallel programming will be discussed in detail. We will now briefly describe the two other groups of language constructs.

The support for multidimensional arrays in *Spar* is an obvious generalization of *Java* arrays, so that statements like this are allowed:

```
short b[*,*] = new short[6,8];

for( int i=0; i<6; i++ ){
    for( int j=0; j<8; j++ ){
        b[i,j] = i+j;
    }
}
```

To support sparse arrays and block arrays, the following language constructs have been added:

- **Parameterization of classes and interfaces**[1]; for example:

  ```
  class TypedStack( type t ) {
      void push( t e ){
          st[++top] = e;
      }
      t pop() {
          return st[top--];
      }
  }
  ```

- **Vector subscripts to arrays**; for example:

  ```
  int a[*,*] = new int[6,8];
  int^2 v = [0,0];
  a v = 4;
  ```

  This code defines a vector v of length 2 that is used to subscribe a two-dimensional array a.

The parameterization of classes and interfaces is similar as in the language Pizza [12].

Together these language constructs allow the definition of the following array interface:

```
interface Array(type t, int n)
{
    t getElt(int^n index)
        throws OutOfBoundsException;
    void storeElt(int^n index,t elm)
        throws OutOfBoundsException,
               ArrayStoreException;
    int^n getSize();
}
```

Any class that implements the Array interface can be accessed through the normal array access notation (e.g. a[1,2]). Sparse matrices and block matrices are supported by classes that implement this interface.

Spar also supports elastic arrays (arrays that can be grown and shrunk during their lifetime), macros, and array statements, but discussion of these features is beyond the scope of this paper. For a more comprehensive discussion of the *Spar* language see [19].

Unlike *Java*, the programming model used in *Spar* is not based on the Thread package provided in the standard *Java* class library. We feel that for the intended applications (computer-intensive, often array-based computations), this programming model is too inconvenient and too error-prone. For similar reasons parallel programming models based on message passing and/or SPMD support libraries [1, 2, 11] have not been chosen. Since the data-parallel programming model, as used in HPF [6], is restricted to semi-regular array computations, this was also not adopted.

Instead of all these programming models, we have chosen to provide a more implicit programming model through the language constructs each and foreach. This programming model was designed to be as convenient as possible for the programmer, at the cost of a more complicated job for the compiler. In this paper we show that it is nevertheless feasible to compile these constructs to sufficiently efficient parallel code.

In the each statement a list of statements is provided, and the compiler is allowed to make a non-deterministic choice for an execution order of these statements. For example, in the statement

```
each { s1; s2; }
```

the compiler is allowed to choose between the execution order

```
s1; s2;
```

and

```
s2; s1;
```

[1] An interface is used to define a protocol that can be implemented by a class. It consists of one or more function definitions. If a class has to conform to the protocol it must implement all the functions in the interface.

Since the choice is non-deterministic, the compiler is free to choose an execution order at compile-time, or to make the choice dynamically at run-time.

The `foreach` is a parameterized version of the `each` statement. Given an iteration variable and an upper bound, the body of the `foreach` will be executed once for every value of the iteration variable between 0 and the upper bound. Again, the compiler is allowed to make a non-deterministic choice for the order in which the iterations are executed. For example:

```
foreach [i:n] {
    a[i].init();
}
```

To simplify parallelization, the iteration range of the `foreach` loop is specified as an iteration variable and a range, and it is not allowed to modify the iteration variable. The standard *Java* notation for an iteration range (e.g. `for( i=0; i<n; i++ )`), is much more general, and makes it much harder to generate efficient parallel code.

3 The compilation target

3.1 Intermediate language

The generated code of our compiler is represented in *Vnus* [5], the intermediate language of our *Timber* compiler framework. Although in reality *Vnus* has a quite different syntax, for clarity reasons we show it in this paper as having a syntax that is very similar to *Spar*, except that it has no `each` and `foreach` statements. *Vnus* does have additional syntax for a number of communication and synchronization primitives. When this is not clear from the context, code fragments in *Vnus* and *Spar* will be labeled to distinguish between the two language levels.

3.2 Target model

The *Spar* compiler generates SPMD code for distributed-memory processors. That is, all processors execute the same static code; the workload is distributed by surrounding the computations with conditionals; each computation is only executed by the processor that 'owns' it.

To support this SPMD scheme, the run-time system of the *Spar* compiler provides mutual-exclusion semaphores, condition variables, a barrier primitive, multi-threading, and one-sided communication primitives, see Section 8.1.

3.3 Threads

At the intermediate level, explicit parallelism is specified by the Vnus statements `fork` and `forkall`. For example:

```
fork { s1; s2; }
```

creates two threads that execute `s1` and `s2` respectively.

The `forkall` statement is a parameterized version of the `fork` statement:

```
forkall [i:n] stmt;
```

creates `n` threads. Each thread has a distinct value of `i` between 0 and `n-1`.

Execution of these threads is pre-emptive. Therefore, execution order of instructions over different threads is non-deterministic. A `fork` or `forkall` statement is completed when all threads it has created have completed.

4 Translating `each` and `foreach` statements

To translate the `each` and `foreach` statements into parallel code, the compiler must do the following:

- Partition the program into schedulable program units called *tasks*.
- Determine a mapping for the tasks onto the processors, and possibly determine a schedule for the tasks.
- Insert synchronization and communication statements to enforce the semantics of the `each` and `foreach` statements.
- Provide a global data space.

The partitioning must result in tasks that are sufficiently large to be worth scheduling. The user specifies a partitioning with `each` and `foreach` statements at an appropriate level of granularity. A more sophisticated compiler will subdivide or aggregate the user-defined tasks to arrive at a better level of granularity.

The algorithms that are applied in the mapping and scheduling of tasks are not discussed in this paper. These algorithms are encapsulated in external functions that can are queried by the compiler or the runtime system to determine the placement and schedule of a task. These functions are constructed in a separate scheduler engine that is described elsewhere [14].

The `each` and `foreach` constructs specify non-determinism at the level of complete (possibly compound) statements. As the translation of these statements will be mostly deterministic, some synchronization will be necessary. We will show a simple compilation scheme that provides this, and a number of optimizations.

The *Spar* language provides a global data space. After scheduling, data is mapped onto distributed memory. Access to global data is done through a Virtual Shared Memory (VSM) layer.

5 Compiler system

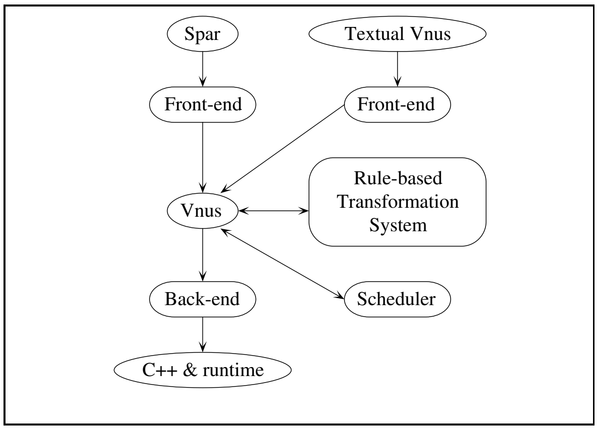

Figure 1: The *Timber* compilation system

Our *Timber* compilation system (see Figure. 1) consists of several engines that communicate via the intermediate language *Vnus* [5]. The system has several front-ends, one of which is *Spar*.

As *Vnus* has no support for object-oriented constructs, the *Spar* front-end translates all object-oriented *Spar* constructs to *Vnus* functions and structures.

The core of the compiler is a rule based transformation system (*Rotan*) which allows easy experimentation with different kinds of transformation schemes. Work is currently being done on a scheduler which will allow the compiler to generate mappings of tasks to processors automatically [14]. Finally there is a back-end that generates C++ code that can be run on different platforms using several possible run-time systems.

The *Vnus* language is used to represent the original program after it has been parsed and semantically analysed. All compiler engines work on this representation.

6 Annotations

Sometimes a compiler engine may add extra information to the *Vnus* representation in order to relay the result of some analysis. We call this information 'annotations'.

The use of annotations is twofold. Firstly, annotations give the flexibility to experiment with language constructs without having to change the compiler front-end. Secondly, it enables the programmer or a separate analysis engine to pass information from the front-end to the other compiler engines. This enables us to emulate all kinds of analysis without actually having that analysis already built into the compiler. This is illustrated with the following example:

Example 1 *Independent annotation*
Spar*:*

```
foreach <independent> [i=0:n] {
    f(A[i]);
}
```

The `independent` annotation ensures that the `foreach` statement is directly mapped onto an internal, explicitly parallel `forkall` statement.

To illustrate how we get from a *Spar* program to an explicitly parallel program we use the following example, which we use as a running example throughout the paper. First we give the *Spar* version:

Example 2 *Simple progam*
Spar*:*

```
C.init();
foreach [i=0:n] {
    A[i] = compute(B[i]);
    C = reduce(C,A[i]);
}
```

Because of the semantics of `foreach` this is a sequential program. Analysis however would point out that the only point of contention between the different iterations is the `reduce()` function[2]. If this point of contention is marked by a mutex (via the `<@m1>` annotation), then the rest of the iteration can be marked `independent`, which would result in:

Example 3 *Parallelism with annotations*
Spar*:*

```
<mutex m1>
C.init();
foreach <independent> [i=0:n] {
    A[i] = compute(B[i]);
    C = reduce(C,A[i]); <@m1>
}
```

7 Task parallelism and scheduling

We have seen how programs can be annotated, and how parallelism can be constrained through the use of mutex mappings.

Before code generation takes place (Section 8), it is necessary to determine which processor executes what code. This is done similarly to mutex mappings such as in Example 3, namely by mapping tasks onto processors. These mappings can be generated via the scheduling engine, or they can be written by hand using annotations in the *Spar* source code.

Note that there is a subtle difference between processor and mutex mappings. Mutex mappings are important to the semantics of the program. That is, without them the program might give completely different, or even non-deterministic results. Processor mappings are not important to the semantics of the program; they

[2]Note that in *Spar* the function `reduce()` is not an intrinsic function.

only specify where parts of the program are executed, but not how. We will see later that they also result in very different generated code.

The following example is an explicit parallel *Vnus* version of Example 3. The example declares a processor array and a `forkall` loop that has to be mapped onto the processors:

Example 4 *Task annotation*
Vnus*:*

```
<mutex m1>
<processor p[n]>
C.init();
forkall [i=0:n] { <task>
    A[i] = compute(B[i]);
    C = reduce(C,A[i]); <@m1>
}
```

In more complex programs it might be difficult for the compiler to determine the grain-size of the tasks that have to be scheduled. To help in this, the programmer can use the `task` annotation in *Spar* to tell the scheduler at what level scheduling is to take place. In Example 4, each instance of the loop body is declared as a task.

Of course instead of indicating what the tasks are that have to be mapped, the programmer can also directly fill in the desired mapping in *Spar*, which would for example result in:

Example 5 *Task mapping*
Vnus*:*

```
<mutex m1>
<processor p[n]>
C.init();
forkall [i=0:n] { <@p[i]>
    A[i] = compute(B[i]);
    C = reduce(C,A[i]); <@m1>
}
```

In this example each iteration instance is statically mapped onto a unique processor via the `<@p[i]>` annotation. Alternatively, the scheduler can determine the mapping. In that case the scheduler generates an annotation like `<@p[sched_forkall1(i)]>` which specifies a function `sched_forkall1(i)>` that is called either at compile-time or at run-time. Such functions require an exact identification of the task that is being scheduled. Here the identification is done through its name and its parameter (iteration number).

As shown in Figure 2, for each iteration instance the whole loop body is mapped onto a processor, whereas only a part of the loop body is mapped onto a mutex.

The processor and mutex mappings might seem to interfere with each-other. This is not the case as both mappings are completely orthogonal. In Section 8.4 we will see how that working parallel code can be generated for Example 5.

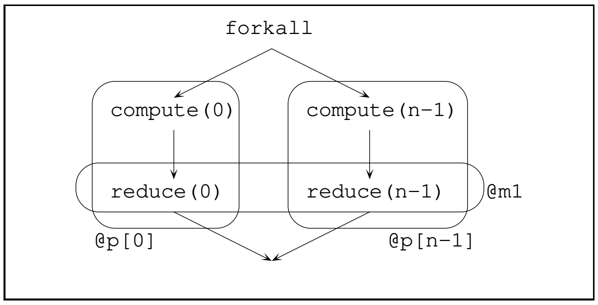

Figure 2: Reduction with mappings

8 SPMD Code Generation for task-parallelism

Assuming that in a previous phase proper dependency analysis has been done, that the program has been partitioned into schedulable tasks, and that these tasks have been completely mapped[3], code can be simply generated through the use of a number of translation rules[4]. Using a system with translation rules not only makes the translation process easier to understand, it also allows us to experiment with different optimization schemes and to ensure correctness. These rules will make use of mapping and other information put together by previous compiler phases.

The rules will be presented in the following form:

```
<input code>
  →
<output code>
```

This form resembles the basic form in which translations are specified in *Rotan*. The rules basically state that if a piece of source code matches `<input code>`, it will be translated into `<output code>`.

The code generation scheme we use is an SPMD scheme which relies on a processor level multithreaded execution model. In our initial and most simple translation, each task is assigned a thread on the processor to which it is mapped. All threads execute the same control structure, but in positions where a mapped task is encountered, only the thread to which the task is mapped will execute it. This scheme resembles the scheme for the owner-computes rule used in data-parallel programming. In the following we will outline the code generation process in more detail.

A program representation is the following example where the execution on different processors is represented by the outer `forkall` loop, while the inner loop is the original program:

[3]For the sake of simplicity the rules presented here will only show static processor mappings. Rules taking dynamic mappings into account will only be a little more complex than the ones presented here.

[4]The rules as presented here are meant to show only the basic ideas. In our compiler the rules contain much more details, such as recursion avoidance and dependency administration.

Example 6 *Multi-processor version*
Vnus*:*

```
<mutex m1>
<processor p[n]>
C.init();
forkall [j=0:n] {
  forkall [i=0:n] {
    if (on_processor(p[i])) {
      A[i] = compute(B[i]);
      C = reduce(C,A[i]); <@m1>
    }
  }
}
```

However, the outer `forkall` loop will not be explicitly generated, as this is done implicitly by executing the same program on multiple processors. As Section 8.4 will illustrate, the inner `forkall` loop will be replaced by a `for` loop, since there is no reason in this example to execute the iterations that are mapped onto the same processor, in multi-threaded mode.

This does not mean that there are no circumstances where multiple threads would not be necessary, just that examples that require multiple threads are outside the scope of this paper.

8.1 Primitives

The translation rules presented here are fairly straightforward, and will be shown for different types of statements. These rule examples show how a statement is implemented using several *Vnus* level primitive functions and constructs. However, before we show these rules, we explain the primitives that are used in the rule examples:

- synchronized() statement
- on_processor() function
- set_condition() and wait_condition() functions
- get_data() and put_data() functions
- pass_barrier() function

8.1.1 synchronized() statement

The `synchronized()` statement is taken from *Java* and specifies that a statement is mutual exclusive with respect to other statements that refer to the same mutex. This means that if a thread is executing such a statement, and another thread also wants to execute that statement, the latter will block until the first thread is finished with the statement.

8.1.2 on_processor()

This function is used to implement the SPMD scheme mentioned above. The `on_processor()` function will return `TRUE` if the current thread is supposed to execute code on the specified processor. Otherwise, it will return `FALSE`.

8.1.3 set_condition(), wait_condition()

The `set_condition()` function is used to set a condition variable. Condition variables are cleared until set and can not be reset. The `wait_condition()` function blocks until the specified condition variable is set. If, for example in a loop, a condition variable is needed multiple times, then for each use a new instance has to be created.

8.1.4 get_data(), put_data()

These functions are used to access global data, which is data that is accessible to multiple processors. The mapping of the data can be specified by the user, otherwise the scheduler will define a mapping. The functions access the virtual shared memory layer of the target run-time system.

8.1.5 pass_barrier()

A barrier is needed in some cases to make sure the program has reached a certain state. It can be expressed through condition variables via the following translation rule (see Section 8.2):

```
pass_barrier()
```

(SPMD Vnus) $\rightarrow$ *(SPMD Vnus)*

```
set_condition(c[p.me()]);
for [i=p.range()]
  wait_condition(c[i]);
```

In the Rules section below the mechanism behind such rules will be explained. Having a separate barrier will make the rules presented here more readable, and there are also several run-time systems that have efficient barriers as primitives which make having such a primitive useful.

8.2 Rules

Rules will be shown for the translation of:

- mapped statements
- if statements
- foreach statements
- for statements
- function calls

Note that most of the rules that will be presented here will transform <input code>, which is *Spar* code, into <output code>, which is SPMD *Vnus* code.

8.2.1 Mapped statements

This rule is intended for a statement or a compound statement that has a defined mapping. The input code in the following rule contains only one *Spar* language construct: stmt. This stmt stands for either a *simple statement* or a *compound statement*. The input code also contains one annotation: <@p>. This annotation defines that stmt should be mapped onto processor p.

```
stmt; <@p>
```

(Spar) → *(SPMD Vnus)*

```
if (on_processor(p)) {
  stmt;
}
```

The generated code[5] makes sure through the use of the on_processor() function that stmt is only executed on processor p.

8.2.2 If statements

This rule will only take care of if statements. The stmt that is contained in it will be handled by a rule that matches that stmt. Similar rules hold for other control statements like the while and do statements.

```
if (expr)
  stmt;
```

(Spar) → *(SPMD Vnus)*

```
pass_barrier(b1);
if (on_processor(p[0])) {
  tmp = expr;
  put_data(tmp);
  set_condition(c1);
} else {
  wait_condition(c1);
  get_data(tmp);
}
if (tmp)
  stmt;
```

[5]There is a problem with the application of this rule though. The stmt in the target code has no mapping anymore, so after application it will not match the rule input. However, if stmt is compound, the compiler tries to apply this rule to any statements inside stmt, or any other rule for that matter. To prevent this from happening, the rule specification we use in our compiler adds an attribute to the generated stmt marking it as final code. rules really work. However, because in this paper we merely present the principles of the translation rules, such details are left out.

It must be ensured that all processors execute the if statement the same way. This means that all data has to be in the correct state, which can be achieved by a barrier, which prevents expr to be evaluated before all modifications to data have been done. Also, as expr can contain calls to functions with side effects, we must guarantee that expr is evaluated by only one processor, which is why the evaluation is mapped onto processor p[0].

The result of the evaluation is then put into a global variable tmp through the use of put_data() and a condition variable is set. The other processors will wait for that condition variable before retrieving the result from tmp. Finally, all processors will execute the if statement with the same condition.

A bit of analysis can help to make the generated code simpler. For example, if expr does not contain any side effects, then every processor can execute it:

```
if (expr)
  stmt;
```

(Spar) → *(SPMD Vnus)*

```
pass_barrier(b1);
tmp = expr;
pass_barrier(b2);
if (tmp)
 stmt;
```

An unfortunate side effect of redundant local computations is that a closing barrier is needed as well to prevent any code coming after the evaluation of expr from interfering. The difference between this translation rule and the previous rule is that we have exchanged a condition variable and a global data access for a barrier, and it depends on the costs of the different communication primitives whether or not this alternative is more efficient.

Finally, if for the evaluation of expr no global data is needed, evaluated without interference from other processors, leaving only the entry barrier:

```
if (expr)
  stmt;
```

(Spar) → *(SPMD Vnus)*

```
pass_barrier(b1);
tmp = expr;
if (tmp)
  stmt;
```

8.2.3 Foreach statements

The definition of the foreach statement states that evaluation of the iterations can be done in a nondeterministic order, but each iteration must be done atomically. In this translation example, we fix the order at compile-time by choosing a standard sequential order by using a for statement:

```
foreach [i=range]
  stmt;
```

(Spar) → (SPMD Vnus)

```
pass_barrier(b1);
for [i=range] {
  stmt;
  pass_barrier(b2);
}
```

Barriers are used to ensure that all processors enter and leave each iteration in lockstep. Note, however, that we only transformed the `foreach` statement, and not `stmt` inside it. We still have to translate `stmt`. If `stmt` has a mapping `<@p[i]>` defined for it, we obtain the following code by applying the rule for mapped statements shown earlier in Section 8.2.1:

```
pass_barrier(b1);
for [i=range] {
  if (on_processor(p[i])) {
    stmt;
  }
  pass_barrier(b2);
}
```

The use of two separate rules produces working code, but leads to the use of an expensive barrier. If the iterations of a `foreach` loop are distributed over multiple processors, a mutex is sufficient to achieve the necessary synchronization. This is done by the following rule that combines the `if` statements with the mapped statement:

```
foreach [i=range]
  stmt; <@p[i]>
```

(Spar) → (SPMD Vnus)

```
for [i=range] {
  if (on_processor(p[i]) {
    synchronized(m1){
      stmt;
    }
  }
}
```

In the above example, the application of the combined rule avoids execution of the iterations in lockstep. This result is difficult to achieve by using separate rules for the mapped statement and the `foreach` statement. Each processor now only executes the iterations that are mapped onto it. The barriers have been replaced by a synchronization on mutex `m1` to ensure that each iteration is still executed atomically. Within each processor the order in which the iterations are executed is still sequential. Nevertheless, because the iterations of several processors are not done in lockstep anymore, the resulting total loop ordering may not be the same lexicographical ordering as the one of the `for` statement. This ordering still conforms to the `foreach` definition.

If the foreach is annotated with the `<independent>` directive, the following rule can be applied:

```
foreach <independent> [i=range]
  stmt;
```

(Spar) → (SPMD Vnus)

```
for [i=range]
  stmt; <independent>
```

The resulting annotated `stmt` would then have to be transformed in turn, keeping in mind that all synchronizations can be removed because of the `<independent>` annotation. If there is a mapping for `stmt` we can do the same trick as before:

```
foreach <independent> [i=range]
  stmt; <@p[i]>
```

(Spar) → (SPMD Vnus)

```
for [i=range] {
  if (on_processor(p[i]) {
    stmt;
  }
}
```

Again, each processor only executes its own iterations, but there are no mutexes now to enforce atomicity.

8.2.4 For statements

The `for` statement can be handled like the `foreach` statement, but the sequential ordering of the iterations always has to be enforced accross processors:

```
for [i=range]
  stmt;
```

(Spar) → (SPMD Vnus)

```
pass_barrier(b1);
for [i=range] {
  stmt;
  pass_barrier(b2);
}
```

Without taking mappings for `stmt` into account we get the same translation as for the `foreach`. If we do take mappings into account we get:

```
for [i=range]
  stmt; <@p[i]>
```

(Spar) → (SPMD Vnus)

```
set_condition(c[range.start()];
for [i=range] {
  if (on_processor(p[i]) {
    wait_condition(c[i]);
    stmt;
    set_condition(c[i+1]);
  }
}
```

Here each processor again executes only the iterations that are mapped onto it. However, to ensure that the iterations are executed in the original sequential ordering condition variables are used. Because of the condition variables we have no need for mutexes as was the case with the foreach statement.

8.2.5 Function calls

Function calls, like `if` statements, are dependent on their environment. Therefore we handle the parameters of a function call similar to the condition of an `if` statement:

```
func(parameters);
```

(Spar) → (SPMD Vnus)

```
pass_barrier(b1)
if (on_processor(p[0])) {
  tmp_parameters = parameters;
  put_data(tmp_parameters);
  set_condition(c1);
} else {
  wait_condition(c1);
  get_data(tmp_parameters);
}
func(tmp_parameters);
```

First a barrier is placed to ensure that all data is in the correct state. Then the parameters are evaluated on processor `p[0]` and put into global memory. The other processors can retrieve these parameters as soon as processor `p[0]` has set the condition variable `c1` indicating that they have been evaluated.

All the optimizations applied to the `if` statement can be applied here as well.

8.3 Optimizations

Rules can not only be used to describe the translation of *Spar* to *Vnus*, but they can also be used to optimize the generated code. Examples of such optimizations are:

- Barrier lifting
- Dead barrier elimination
- Condition variable lifting and aggregation

In the rule for `if` statements barriers are generated that are associated with the variables used in `expr`:

```
...
```

(Spar) → (SPMD Vnus)

```
pass_barrier(b1);
tmp = expr;
pass_barrier(b2); <Var(expr)>
...
```

where `<Var(expr)>` denotes an annotation that specifies the set of variables the barrier is associated with.

One optimization is barrier lifting:

```
pass_barrier(b); <Var(A)>
stmt; <NoDef(A)>
```

(SPMD Vnus) → (SPMD Vnus)

```
stmt; <NoDef(A)>
pass_barrier(b); <Var(A)>
```

where `<NoDef(A)>` denotes that `stmt` does not write to any variables in set `A`.

Such a rule is intended to work in combination with the following barrier removal rule:

```
pass_barrier(b1); <Var(B)>
pass_barrier(b2); <Var(C)>
```

(SPMD Vnus) → (SPMD Vnus)

```
pass_barrier(b); <Var(B+C)>
```

Removing barriers will reduce the synchronization time of the program, and such rules enable the specification of a barrier rule removal in a fairly straightforward manner. However, in these kind of rules we have to modify the annotations as well, as the new barrier has to preserve all the dependencies that were associated with both old barriers.

8.4 Using the rules

The following code is a translation of Example 6 and is not optimized. Because, in principle, we might not know what the mapping is of the iterations, all processors execute all iterations. However, through the use of the `on_processor()` function, each processor checks at runtime whether or not the code inside an iteration has to be done:

Example 7 *Generated code*
SPMD Vnus*:*

```
<mutex m1>
<processor p[n]>
for [i=0:n] {
    if (on_processor(p[i])) {
        A[i] = compute(B[i]);
        synchronized(m1){
            C = reduce(C,A[i]);
        }
    }
}
```

This example causes many iterations will be spawned that will not do anything. In general this is very inefficient. However, *Spar* and *Vnus* have *masked iterations*, which are similar to `WHERE` clauses in *F90* and *HPF*. This means that we can change the above `for` statement into:

```
for [i=0:n, on_processor(p[i])] {
```

An extra condition has been added to the `for` statement that causes only those iterations to be started for which the condition is `TRUE`. Here the `on_processor` function is used to check whether or not an iteration is supposed to execute on the current processor. By using this, we obtain:

Example 8 *Generated optimized code*
SPMD Vnus*:*

```
<mutex m1>
<processor p[n]>
for [i=0:n, on_processor(p[i])] {
    A[i] = compute(B[i]);
    synchronized(m1){
        C = reduce(C,A[i]);
    }
}
```

If the `on_processor(p[i])` expression is regular, we can use standard iteration squeezing techniques as described in [17].

9 Conclusion

In this paper we have described a compilation scheme to translate implicitly parallel programs in the programming language *Spar* to code for a distributed-memory parallel computer system. The compilation scheme has been formulated as a set of transformation rules.

Work is currently being done to complete the *Spar* front-end, and work on the scheduler is in progress. The rules by which the SPMD schemes have to be generated are well understood, and as we already have experience using rule based transformation systems, we expect to have a running implementation shortly.

Future work will include optimizing the generated code and analysis of more complex mappings. The use of condition synchronization will also be added to the schemes presented here to allow more complex dependencies between tasks. We will also look into the use of 2-sided communication, in order to achieve efficient more communication patterns wherever possible.

References

[1] B. Carpenter, Y.-J. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with HP Java. *Concurrency: Practice and Experience*, 9(6), June 1997.

[2] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. Introduction to java-ad. Technical Report SCCS-799, NPAC at Syracuse University, Syracuse, NY 13244, November 1997.

[3] K.M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In P. Wegner G. Agha and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT Press, 1993.

[4] B. Chapman, P. Mehrotra, J. Van Rosendale, and H. Zima. A software architecture for multidisciplinary applications: Integrating task and data parallelism. In *Proc. Fifth Workshop on Compilers for Parallel Computers*, pages 454–466, Malaga, June 1995.

[5] P.F.G. Dechering. *Semantics for Compiling Data Parallelism*. PhD thesis, Delft University of Technology, Delft, March 1998.

[6] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Rice University, 2.0 edition, Januari 1997.

[7] I. Foster. Task parallelism and high-performance languages. *IEEE Parallel and Distributed Technology*, Fall:27–36, 1994.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Cambridge, MA, 1996.

[9] T. Gross, D.R. O'Hallaron, and J. Subhlok. Task parallelism in a high-performance Fortran framework. *IEEE Parallel and Distributed Technology*, Fall:16–26, 1994.

[10] M.R. Haghighat and C.D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18:477–518, July 1996.

[11] S.F. Hummel, T. Ngo, and H. Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, 9(6):621–631, 1997.

[12] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *POPL97*, pages 146–159. ACM, January 1997.

[13] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proc. 1989 Int. Conf. Parallel Proc.*, volume II, pages 39–48. IEEE, August 1989.

[14] A. Radulescu, A.J.C. van Gemund, H-X. Lin, and H.J. Sips. Low-cost scheduling algorithms for distributed-memory architectures. In *Proc. 4th Int. ASCI Conf.*, pages 115–120, Lommel, June 1998.

[15] M.C. Rinard, D.J. Scales, and M.S. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, pages 28–38, June 1993.

[16] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA., 1996.

[17] C. van Reeuwijk, W. Denissen, H.J. Sips, and E.M. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, September 1996.

[18] C. van Reeuwijk, H.J. Sips, H.X. Lin, and A.J.C. van Gemund. Automap: A parallel coordination-based programming system. Technical Report 1-68340-44(1997)04, Delft University of Technology, Delft, The Netherlands, April 1997.

[19] C. van Reeuwijk, A.J.C. van Gemund, and H.J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9:1193–1205, November 1997.