Task Parallelism in *Spar*

Frits Kuijlman, Andrei Radulescu, Arjan J.C. van Gemund and Henk J. Sips

Department of Information Technology and Systems
Delft University of Technology, P.O.Box 356, 2600 AT Delft, The Netherlands
Email : F.Kuijlman@twi.tudelft.nl

Keywords: language design, distributed computing, scheduling

1 Abstract

In this paper we will give a small introduction to Spar, a programming language for semi-automatic parallel programming. Spar, a Java derivative is being developed as part of the Automap project, which aims to create completely automatic mappings for code and data at either compile time or runtime.

We will present how Spar supports task-parallel programming and how it interacts with the user or the scheduler to define code mappings.

2 Introduction

Issues that are of major importance in the design of parallel programming languages 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 [6] and *CC++* [2], as well as by message-passing interfaces such as *MPI* [14] offer full parallelizability, yet at a significant expense of programmability and portability. Although in high-performance computing for pragmatic reasons explicit languages are still widely used, programming requires a detailed understanding of the complex interplay between program and machine, is extremely laborious and error-prone.

In implicit parallel languages parallelism is deduced by the compiler. Examples are imperative languages such as *Fortran* where parallelism is deduced through compile-time data dependence analysis, thus providing portability [11]. Due to the inherent limitations of this analysis, however, parallelizability is typically restricted to regular cases [8]. Aimed to alleviate these detection problems, language extensions such as in *Jade* [13] and *Fx* [7] have been proposed that aid the compiler in the data dependence analysis.

Intended to offer a compromise, data parallel languages such as *Fortran-D* [9] and *HPF* [10] extend the implicit parallel paradigm by offering data mapping annotations that induce data parallelism. While this approach alleviates the data dependence analysis, problem parallelizability is limited to data parallelism, which has prompted a number of recent extensions such as *Fortran-M* [4] and *Opus* [3] to integrate task and data parallelism. Furthermore, because of the introduction of mapping directives, portability is significantly decreased.

In the design of *Spar* [16], a task parallel language that is developed within the *Automap* project [15], a different trade-off is made between parallelizability, programmability and portability. Illustrated by the project's name, the main emphasis is on an entirely automatic mapping of tasks and data by a scheduling engine, thus achieving maximum portability. The engine employs new compile-time and run-time scheduling algorithms aimed to deliver acceptable performance at very low cost [12]. A compromise between parallelizability and programmability is achieved by defining an implicit programming interface, yet providing extensions that allow a restricted form of non-determinism, thus offering better parallelizability.

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) data dependencies. To enhance scheduling performance, mapping and/or alignment information of tasks and data can be passed to the compiler through annotations.

In this paper we present the approach followed in designing *Spar*. We discuss the consequences our specific choice of language constructs and annotations have on parallelizability, programmability, and portability. In particular, we address the issues involved in a novel resource mapping concept that is introduced in *Spar* to control task synchronization and mapping.

The paper is organized as follows. Section 3 will describe the language constructs we intend to use to describe parallelism. Section 4 will describe briefly how we view task parallelism and the internal compiler representations that are needed. In Section 5 our compilation system will be briefly described, including annotations which are used to communicate between the different modules of the system. In Section 6 the previous sections will come together, and Section 7 will give an example of the code that will be generated.

3 Language requirements

In this paper we are only concerned with how we tackle task parallelism, and the way we achieve this through the use of the *Spar* language. Therefore we will only discuss these elements of the *Spar* language that have to do with expressing parallelism. For a more comprehensive discussion of the *Spar* language see [16].

The *Spar* language is based on *Java* with a few extensions. It should be stressed that our intention is not to create yet another *Java* variant. Instead we want to be able to explore different language constructs, and *Java* is a relatively simple language that provides a vehicle to explore these language constructs without having to create a complete language from scratch.

In *Java* parallelism is expressed using the Thread package. The use of such packages is of course consistent with doing as much as possible via standard objects. However, the use of such run-time libraries makes it much harder for a compiler to manipulate the parallelism in a program. For this reason we have chosen to express parallelism in *Spar* using language constructs.

We could of course have chosen for an explicitly parallel construct. However, this, in combination with shared global memory, mutual exclusion and condition synchronization [1] would have given the programmer complete expressiveness in describing parallelism. In our opinion this kind of parallelism, like pointers in the *C* language, makes it very hard for a compiler to analyse and manipulate programs. Therefore, we made the choice for implicit parallelism, which implies that some analysis is needed to find the parallelism in a program. *Spar* contains the `each` and `foreach` statements:

Example 3.1 *Spar : each and foreach statements*

```
each {
    s1;
    s2;
}
foreach (i=0:n) {
    a[i].init();
}
```

The `foreach` loop and the `each` statement allow specifying non-deterministic behaviour while preserving sequential semantics. They specify that the contained statements can be executed in any sequential order. If analysis concludes that the contained statements are independent then parallel execution is possible. The analysis of these non-deterministic statements is made easier compared to normal `for` statements because of the non-determinism.

We have sacrificed some parallel expressiveness here in return for a well defined semantics. The price we pay here is the need for some analysis. We will show a possibility to work around the need for analysis later in this paper.

4 Task parallelism

Another goal of the research is that we want to express programs in terms of medium to large grain tasks. The computation time of these tasks should be orders of magnitude larger than the time that is taken by the communication part of the program that coordinates the different tasks. A program can be partitioned and represented as a task graph, which we intend to automatically map onto the different processors of a parallel computer.

The programs we are concerned with are basic series-parallel (SP) programs. These programs can be viewed as a series of sequential and parallel compositions of smaller tasks.

The following example is such an SP program. Note that this program only make use of the low level explicitly parallel `par` statement instead of the high level `foreach` statement. Because this `par` statement is only available in the internal representation of our compiler and not in the *Spar* language, the examples using this statement are called pseudo-code instead of *Spar*. The *Spar* syntax is used though, because the real internal compiler representation is not very suited for purposes of presentation.

Example 4.1 *Pseudo-code : basic par statement*

```
par {
     s1;
     s2;
};
s3;
par (i=0:n) {
     f(A[i]);
}
```

In this example we view `s1`, `s2`, `s3` and each call to `f(A[i])` as a task. Here each task is only a simple statement. In the real programs we are concerned with, the tasks are substantially bigger.

Implicit interaction is already possible through shared global variables. Because such interaction can not be used to express all parallel operations (for example reductions), we also need something to express controlled interaction between tasks. We do this by providing a mechanism for mutual exclusion. This mechanism consists of a mutex and a mapping of several mutual exclusive tasks onto that mutex:

Example 4.2 *Pseudo-code : reduction with mutex*

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

In this example `n` parallel threads of control (not to be confused with *Java* threads) are defined by the `par` statement. Thus, all `compute()` calls can be executed in parallel. However, since all calls to `reduce()` are mapped onto the same mutex `m1` (see Figure 4.1), they will be executed one at a time, which will result in a correct computation of the reduction. In this example possible parallelism has been constrained by the use of a mutex mapping.

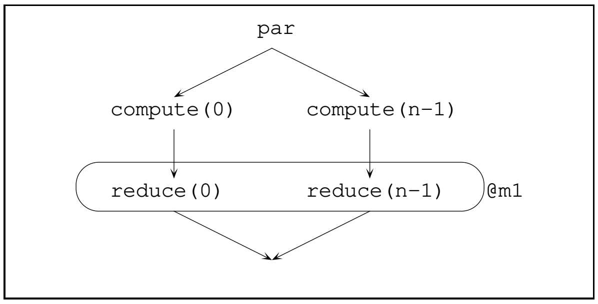

Figure 4.1: Reduction with mutex

5 Compiler system and annotations

In the previous sections we have given an overview of how parallelism is expressed in the *Spar* language, and how we view task parallelism. Before we go on to the subject of task and processor mappings we first want to describe how these two different ways of describing parallelism come together. For that we will take a look at our compiler system:

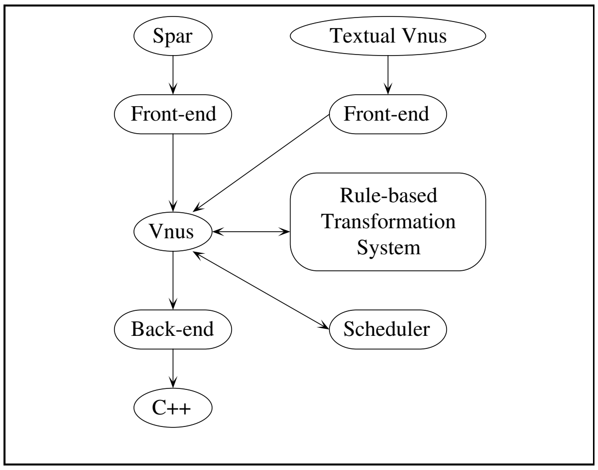

Figure 5.1: The compilation system

As Figure 5.1 illustrates our compilation system consists of several modules that communicate via the intermediate language *Vnus*. There are several front-ends, one of which is *Spar*. The central core is a rule based transformation system which allows us easy experimentation with all 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. Finally there is a back-end that generates C++ code that can be run on different computers 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 modules work on this representation. Sometimes a module might find it necessary to add extra information to the *Vnus* representation in order to relay the result of some analysis. We call this information 'annotations'. These can be compared to *HPF* directives[5].

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 also enables the programmer to pass information from the front-end to the other compiler modules. 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 5.1 *Spar : independent annotation*

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

Because the `foreach` statement is a high level statement, and because the analysis part of the compiler might not be able not adequately analyse all cases in which it is used, we introduced the `independent` annotation. This annotation ensures that the `foreach` statement is directly mapped onto an internal, explicitly parallel `par` statement.

To illustrate how we get from a *Spar* program to an explicitly parallel program we use the reduction example. First we give the *Spar* version:

Example 5.2 *Spar : reduction*

```
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 would point out that the only point of contention between the different iterations is the `reduce()` computation. If this point of contention is marked by a mutex then the rest of the iteration can be marked `independent`, which would result in:

Example 5.3 *Spar : reduction with annotations*

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

This program is equivalent to the reduction example4.2, and can be transformed into it by simply replacing the annotated `foreach` statement with an internal *Vnus* `par` statement.

6 Task parallelism and scheduling

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

Before we can get to the code generation stage we need to know which processor executes which code. This is done similarly to mutex mappings, namely by mapping tasks onto processors. These mappings can be generated via the scheduling module, or they can be written by hand using annotations in the *Spar* source code.

Processors and mutexes can both be viewed as resources onto which tasks are mapped. We will see later that they also result in very similar code generation. There is however a subtle difference between the two. 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 only specify where parts of the program are executed, but not how.

Example 6.1 *Pseudo-code : task annotation*

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

This is a simple example, but 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 `task` annotation is used to indicate to the scheduler at what level scheduling has to take place.

Of course instead of indicating what the tasks are that have to be mapped, the programmer can also directly fill in the desired mapping:

Example 6.2 *Pseudo-code : task mapping*

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

In this example each iteration instance is mapped onto a different processor via the `<@p[i]>` annotation.

As you can see the notation for processor mappings and mutex mappings are very similar, and during code generation they will be handled similarly. There is however a problem.

As shown in Figure 6.1, 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. However, we want to map tasks only, not parts of tasks, and even more importantly, we want to schedule these tasks non-preemptively. So, the question is, at what level are we scheduling? If we schedule the whole loop body, then it is not possible to map the call to `reduce()`. The conclusion is, especially if

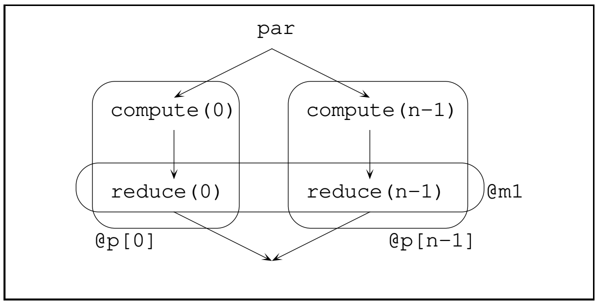

Figure 6.1: Reduction with mappings

we consider the original program and its dependency analysis, that the iteration instances, except for the `reduce()` call, can be interleaved, and that the mapping should look like:

Example 6.3 *Pseudo-code : final mapping*

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

As Figure 6.2 shows we now have separated each iteration instance into two tasks, one for the `compute()` part and one for the `reduce()` part. This is actually even a better mapping than Example 6.2 because it allows multiple iterations to be mapped onto the same processor without blocking each other completely because of the mutex. That is, when an iteration that has finished the `compute()` part is blocked because the mutex is busy, another iteration on the same processor can execute its `compute()` part. In Example 6.2 the second iteration would have to wait until the first iteration had finished its `reduce()` part.

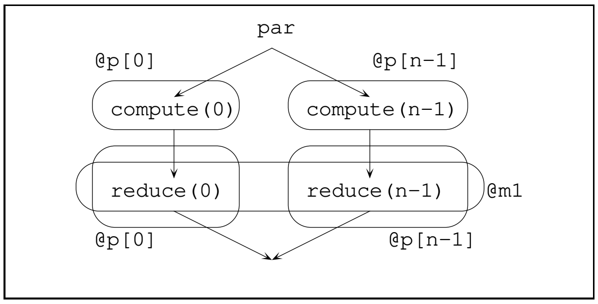

Figure 6.2: Reduction with alternative mappings

7 Code generation

Basic code generation is based on the generation of SPMD code using `if-owner()` schemes, that is, the same program will run on all processors, but code will only be executed if the current process is the owner of processors and/or mutexes as specified in the mappings.

The code that is presented in Example 7.1 is not optimized. We assume that the number of declared processors is not the same as the number of actual processors present. Therefore an extra outer `par` loop is used to span the actual physical processors, while the inner loop is used to to span the declared processors.

Example 7.1 *Generated code*

```
<mutex m1>
<processor p[n]>
par (j=0:m) {
    par (i=0:n) {
        if (get_processor(p[i])) {
            A[i] = compute(B[i]);
            free_processor(p[i]);
        }
        if (get_processor(p[i])) {
            get_mutex(m1);
                C = reduce(C,A[i]);
            free_mutex(m1);
            free_processor(p[i]);
        }
    }
}
```

The `get_processor()` function is used to acquire the virtual processor onto which the code was mapped in the original program. The function will return false if the current processor is not supposed to execute the code for the requested virtual processor. Otherwise it will block until the wanted virtual processor is available, as there can be only one process running at the same time within a virtual processor. The `get_mutex()` function will always block until the requested mutex is available. A mutex has always to be acquired, as the decision whether or not to execute code has already been taken at the `get_processor()` level.

In the above example after `compute()` a `free_processor()` call is used to free the used processor, immediately followed by a call to `get_processor()` to acquire it again. This is done to reflect the mapping in Example 6.3, which in effect specifies that there is a scheduling point between the calls to `compute()` and `reduce()`.

For a parallel program to execute correctly it is not enough to specify only code mappings. Global data also has to be mapped to give the program some meaning. The user can therefore specify a mapping for the data, or if there is no mapping the compiler will generate a random mapping.

Note that in general this way of specifying programs precludes the use of two-sided communication. Therefore, unless analysis points out otherwise, the compiler will assume one-sided communication, and will generate the necessary code for daemons/threads to handle memory access requests.

8 Conclusion

In this paper we have described the part of the *Spar* language that is concerned with parallelism. We also described how we used annotations to create hooks in the language so that the scheduler can create mappings that allocate different parts of a program to different processors. Such annotations can also be used to cover analysis deficiencies in current and future versions of the compiler.

Work is currently being done to complete the *Spar* front-end, and work on the scheduler has recently started. The rules by which the SPMD schemes have to be generated are well understood, and as we already have experience with the rule based transformation system, we hope to have a running implementation shortly.

In the future we will start work on optimising the generated code. We will also want to analyse more complex mappings, especially the possible relationships between processor and mutex mappings. The use of condition synchronization will also be added to the schemes presented here to allow more complex dependencies between tasks.

References

[1] G.R. Andrews and F.B. Schneider, "Concepts and notations for concurrent programming," *Computing Surveys*, vol. 266, no.24, 1983, pp. 132-145.

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

[3] 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*, Malaga, June 1995, pp. 454–466.

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

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

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

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

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

[9] S. Hiranandani, K. Kennedy and C-W. Tseng, "Compiling FORTRAN-D for MIMD distributed-memory machines," *Communications of the ACM*, vol. 35, Aug. 1992, pp. 66–80.

[10] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr. and M. Zosel, *The High-Performance Fortran Handbook*. MIT Press, 1994.

[11] 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.*, IEEE, Aug. 1989, pp. II:39–48.

[12] 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.*, Lommel, June 1998, p. submitted.

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

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

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

[16] 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*, vol. 9, Nov. 1997, pp. 1193–1205.