An Integrated Annotation and Compilation Framework for Task and Data Parallel Programming in Java*

Henk J. Sips and Kees van Reeuwijk
Delft University of Technology
(sips,reeuwijk)@its.tudelft.nl

Abstract

We describe a set of language extensions to Java to support parallel programming with distribution annotations. The system provides an integrated system of placement annotations on both code and data. This allows the programmer to freely mix data-parallel programming similar to HPF, and task-parallel programming similar to OpenMP.

To evaluate the effectiveness of our parallel programming model, we did an experimental implementation of the language extensions in our compiler, Timber. We also present three programs from the NAS benchmark suite that use these extensions, and we compare them with their original Fortran77 implementation and an HPF implementation.

1 Introduction

For most applications Fortran has been replaced by languages such as ANSI C, C++. and Java. Yet Fortran has a number of features that still make it particularly useful for scientific programs, namely multi-dimensional arrays, complex numbers, and, in later versions, array expressions. Moreover, Fortran compilers tend to produce highly efficient code compared to compilers for other languages.

We are developing a set of language constructs, called *Spar*, that augment Java with a set of language constructs for scientific programs. The set consists of multi-dimensional arrays; complex numbers; a 'toolkit' to build specialized array representations such as block, symmetric, or sparse arrays; annotations; and parallelization. In this paper we concentrate on the constructs for parallel programming; the other extensions have been described in [17].

Fortran is also a popular programming language for implementing *parallel* scientific programs. In its simplest form, parallelization can be achieved by using an explicit parallel programming interface, such as provided by libraries like MPI. Although very efficient parallel programs can be written with the combination Fortran/MPI, all communication, synchronization, deadlock avoidance, and load balancing between processors must be implemented explicitly by the programmer. This is time-consuming and error-prone, since a detailed understanding of the complex interplay between program and machine is required. Moreover, the optimal approach to communication, synchronization and load balancing is system-dependent.

Therefore, there are two strong reasons to leave part of the parallelization process to the compiler: to shield the user from the hazards of parallel programming, and to make the parallel programs portable. To support this, a set of language constructs must be defined that provide a more abstract parallel programming model, and a compiler that generates efficient parallel code from the abstract language constructs. Obviously, it is important to keep the cost of this abstraction (reduced program efficiency) low.

*This research was supported by NWO (project 'Automap'), Esprit (LTR Project 'Joses' (#28198)), and Delft University of Technology (DIOC project 'Ubicom').

One approach is to start with a sequential programming language, and let the user add annotations to hint the compiler about the best way to parallelize the program. This approach is attractive because it allows an existing (sequential) programming language to be converted for parallel programming, although it is usually necessary to add a few new language constructs. A number of annotation-based parallel programming languages have been proposed, of which High Performance Fortran (HPF) [10] and OpenMP [13] are the most familiar. Both add annotation-based parallelism to Fortran.

In HPF the *data-parallel* programming model is used: the user only has to specify the distribution of arrays over the processors, and give the compiler some opportunities for parallelization by using the new `forall` loop construct, F90 array operations, or an annotation on an ordinary `DO` loop. All communication and synchronization are provided by the compiler. Due to symmetry, load balancing is largely inherent, although choosing a suitable distribution may have a significant impact.

Data-parallel programming is very effective for algorithms such as relaxation solvers, other stencil operations, and many linear algebra operations. However, it is strongly oriented towards array-based operations, and lacks the expressiveness to fully implement many other significant algorithms for scientific computation.

OpenMP is a set of language extensions for Fortran and C++ for shared memory *task-parallel* programming. The user specifies the parts of the program where parallel threads can be spawned. These threads are then distributed over the available processors; they execute their share of the workload, and terminate at the end of the parallel section. The main task of the compiler is to generate code for thread creation and termination. The burden of providing proper synchronization between threads is left to the programmer, although the structured parallelism of OpenMP makes this easier to do, and OpenMP provides a number of higher-level synchronization constructs. Similar to HPF, the inherent symmetry simplifies load balancing.

Task-parallel programming is effective for a much broader range of programs than HPF, but it isolates the programmer much less from explicit parallel programming.

Both HPF and OpenMP rely on annotations to hint the compiler about potential opportunities for parallelization. In HPF this is mainly done by specifying the distribution of data over the processors, in OpenMP computational threads are distributed. It is therefore tempting to consider a system where both approaches are combined, to reap the benefits of both approaches.

The integration of data and task parallelism has been an active area of research in recent years. Most approaches start with either a data or a task parallel programming model and add constructs to support the other domain. Approaches using the data-parallel programming model typically rely on the compiler to achieve efficiency, and therefore tend to add constructs that require compilation for their task-parallel extensions [10, 19]. In general, this leads to task-parallel constructs with limited expressiveness. For example, in HPF a the `ON HOME` clause provides a limited form of loop-level task parallelism.

In contrast, task-parallel programming languages are usually not annotation-based, but have constructs to explicitly create and destroy threads. This makes it very difficult for the compiler to intervene, and therefore these systems rely on efficient run-time systems. Parallelism in these languages is usually much more coarse-grained to mitigate run-time overhead. When such a system is extended with data-parallel constructs, they also rely on run-time support. Examples of such systems are Orca [1], CC++ [5], and even Java. Consequently, many proposals for data-parallel programming support in Java use run-time libraries [3, 2, 4, 21].

Even OpenMP, which uses more implicit thread creation and destruction through annotations, makes it difficult to optimize the parallelization constructs at compile-time. In [11] a proposal has been made to integrate HPF and OpenMP. This proposal clearly shows the difficulty and complexity of integrating data parallelism in OpenMP. The OpenMP threads are still too explicit to be easily handled by the compiler, as is necessary for efficient data parallelism.

In this paper, we present a unified model to describe data and task parallelism in an object oriented language. The approach allows compile-time and run-time techniques to be combined. For all constructs

there is a simple implementation using run-time support, but the compiler is given ample opportunity to apply optimizations at compile-time to generate more efficient code. There is no room in this paper to describe the used optimization techniques; they are described elsewhere [7, 16].

The language constructs that are presented in this paper are the `foreach` loop construct (Sec. 2), a general-purpose annotation construct (Sec. 3), and a number of distribution annotations expressed in this annotation language (Sec. 4).

We evaluate these constructs as follows. In Sec. 6, Spar's parallelization constructs are compared with those of HPF and OpenMP. We did an experimental implementation of the parallelization constructs in our compiler, Timber. This is described in Sec. 7. In Sec. 8 we present three programs from the NAS benchmark suite that use these extensions, and we compare them with their original Fortran77 implementation and an HPF implementation. Finally, in Sec. 9 related work is discussed, and in Sec. 10 we draw some conclusions.

2 The `foreach` construct

In a sequential program the execution order of all statements is specified exactly. This is often an over-specification: the programmer may not care about the execution order of statements, even if the observable results differ. For example, the code

```
for( int i=0; i<a.length; i++ ) a[i] = Math.sin( 2*Math.PI/i );
```

fills array `a` in the exact order that is specified in the `for` loop. This is an over-specification, since order in which the array is filled is irrelevant for the final result.

Since this form of overspecification often prevents parallelization of the program, Spar provides the `each` and `foreach` statements. Given an `each` statement such as:

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

the compiler may choose one of the execution orders `s1; s2;` or `s2; s1;`. Once a statement has been started, it must complete before another statement is started.

The `foreach` statement is a parameterized version of the `each` statement. For example, we can now write the loop above as:

```
foreach( i :- 0:a.length ) a[i] = Math.sin( 2*Math.PI/i );
```

Similar to the `each` statement, the iterations may be executed in any order, but an iteration must complete before the next one is started.

The `each` and `foreach` statements can only be executed in parallel when there is no observable interference between statements or iterations. To discover this, the compiler requires some data-dependency analysis (although less extensive than for sequential loops), or the user must inform the compiler with an annotation.

3 The annotation language

Statements, expressions, types, declarations, formal parameters, and the entire program can be annotated with pragmas. For example:

```
<$ reduction,iterations=@n $> foreach( i :- 0:n ){ sum += i; }
```

annotates a `foreach` statement with two pragmas: a `reduction` pragma, and an `iterations` pragma.

Spar provides a general annotation mechanism. An annotation consists of a list of *pragmas*. These pragmas allow the user to give the compiler information about the program, and give hints for efficient compilation. By convention, a pragma does not influence the behavior of a program[1]; it only improves the efficiency of the program in terms of execution time, memory use, or any other measure. For example, the annotation:

```
<$ independent,boundscheck=false,iterations=42 $>
```

consists of three pragmas. The first one, `independent`, does not have a value, and is called a *flag pragma*. The other two, `boundscheck` and `iterations`, have a value, and are called *value pragmas*.

Identifiers in the pragma name and pragma value are completely independent of those in Spar/Java; they have a different *name space*. It *is* possible, however, to refer to variables in the host language by prefixing a name with a '`@`'. For example, in the code fragment

```
foreach( i :- 0:n ) <$ iterations=@n $> { sum += i; }
```

the value of the `iterations` pragma contains a reference to the variable `n`.

Instead of a single value, a pragma may have a *list* of values. Such a list is written as a sequence of expressions surrounded by brackets. Since the lists can be nested to an arbitrary depth, this allows expressions of arbitrary complexity. For example:

```
<$cost=(lambda (i j) (sum (prod 5 i i j j) (prod 3 i j) 42)$>
```

As a convenience, Spar allows a number of binary operators to be used in pragma expressions. The traditional precedence rules on these operators are obeyed. These expressions are immediately translated to expression lists. For example, the pragma expression `3*n*n+5*n+1` is translated to '`(sum (prod 3 n n) (prod 5 n) 1)`'.

Furthermore, Spar allows subscript-like expressions. They are immediately translated to a list starting with the identifier '`at`'. For example, the pragma expression '`p[1,a]`' is translated to '`(at p 1 a)`'.

The following language constructs can be annotated: the entire program, statements, expressions, types, declarations, and formal parameters.

4 Pragmas for parallelization

Both data declarations and code fragments can be annotated with the `on` pragma, which specifies the placement of that data, or code. For example:

```
int[*,*] <$ on=(lambda (i j) dsp2D[(block j 5),_all]) $>
  b = new int[50,50];
```

annotates an array with a placement pragma in data-parallel fashion, and

```
foreach( i :- 0:100 ) <$ on=dsp1D[(block @i 25)] $> {
  a[i] = a[i] + 1; }
```

annotates the iterations of a foreach with a placement pragma in task-parallel fashion.

To help the compiler with the parallelization of a program, the user may annotate a program with pragmas to specify the distribution of data, or the place where a block of code is executed. For this purpose the parallelization engines support the following annotations:

The `ProcessorType` pragma is used to declare names of processor types and to associate them with processors characteristics (e.g. alignment of data structures and endianness of primitive types etc.) and capabilities (e.g. whether it contains a FPU etc.). `ProcessorType` pragmas must be global. For example:

---

[1]Since the `each` and `foreach` statements are non-deterministic, we consider a program to have the same behavior for all possible execution orders of these statements. However, an annotation may cause another execution order of such a statement.

```
<$ ProcessorType=((Gpp "Pentium2") (Dsp "Trimedia")) $>
```

The strings "Pentium2" and "Trimedia" refer to processor descriptions that are known to the compiler. A Processors pragma is used to name the processors in a system, and describe their arrangement to the compiler. The Processors pragma must be a global pragma. Either a single processor can be declared, or an array of processors. The processor array can have any number of dimensions. For example:

```
<$ Processors=((Gpp gpp1) (Dsp dsp1D[4]) (Dsp dsp2D[2,3])) $>
```

This system consists of a single processor gpp1 of type Gpp, a one-dimensional processor array dsp1D, and a two-dimensional array dsp2D, both of type Dsp. Each modeled processor corresponds to a single physical processor.

A Spar/Java program can be annotated with on pragmas to place data and work on specific processors. Pragmas can annotate expressions, statements, and member functions.

```
<$ on=Gpp $>
<$ on=Dsp[0] $>
```

Pragmas may be nested; an on pragma on an inner block or expression overrules the enclosing specification.

The special on value _all is used to denote all processors; the value '_' (a single underscore) is used to denote an unspecified distribution ('don't care'). For example:

```
<$ on=_all $>
<$ on=_ $>
<$ on=Dsp[_all] $>
<$ on=Dsp[_] $>
```

It is also allowed to use three special functions as on expressions: The (block a*i+b m) distribution function places index i onto processor $p = (a \cdot i + b)/m$. The value of p is bounded by the index range in the corresponding dimension of the processor type. If no m is specified, the value is derived from the context, if possible: if there are N elements in the corresponding array dimension, $m = N/P_{ext}$ is assumed. The (cyclic a*i+b) distribution function places index i onto processor $p = (a \cdot i + b) \bmod P_{ext}$. The (blockcyclic a*i+b m) distribution function places index i onto processor $p = ((a \cdot i + b)/m) \bmod P_{ext}$.

For data that is distributed with the block and cyclic functions, the compiler is able to generate highly efficient code to enumeration local elements and to translate global to local array indices; see [16] for details. With these functions, all the data mappings of High-Performance Fortran (HPF) 2.0 [10] can be specified. See Sec. 6 for more details.

Annotating declarations By annotating a member function, the user can specify the group of processors allowed to execute the member function. For example:

```
<$ on dsp1D[_all] $>
int myfunc( int i, String s ) { return i+1; }
```

Annotating statements A statement annotated with an on pragma will be executed only on the specified processor(s). In principle arbitrary statements may be annotated, but in practice the annotation is mainly interesting for code *blocks*. For example:

```
foreach( i :- 0:100 ) <$ on=dsp1D[(block @i 25)] $> {
    a[i] = a[i] + 1; }
```

The assignment of a[i] is executed on processor dsp1[i/25]. In other words, the complete iteration space is divided into blocks of size 25, and each block is executed on a different processor.

Annotating expressions In principle, any expression can be annotated with an on pragma. This specifies the place where the expression must be evaluated. The new expression is a special case, since this not only specifies the distribution of the constructor execution (if not overridden by an annotation on the constructor), but also the distribution of the newly constructed class or array instance. For example, the pragma:

```
String a = <$ on=gpp1 $> new String();
```

specifies that a new String must be constructed on gpp1.

In the case of array new expressions, a slightly extended version of the on pragma is allowed. For example, the pragma:

```
int[*,*]
  <$ on=(lambda (i j) dsp2D[(block j 5),_all]) $>
  b = new int[50,50];
```

specifies that every array element b[i,j] is constructed on processors dsp2D[(block j 5),_all]. The _all expression in the second dimension means that the elements are *replicated* in that dimension. Since the formal parameter i is not used in the distribution expression, the first dimension of the array does not influence the distribution of elements.

5 Other annotations

The Spar/Java programs that we discuss in Sec. 8 use two other annotations. We plan to implement analysis engines that generate these annotations, but we expect that there will remain cases where automatic analysis would be too complicated and time-consuming to be fruitful, and the annotation must be provided by the user.

The foreach statements allows reduction operations such as

```
int sum = 0;
<$ reduction $> foreach( i :- 0:a.length ) sum += a[i];
```

If the array that is being reduced is block or cyclic distributed, and if the loop is annotated as a reduction, the parallelization engines are able to generate efficient parallel code for such a reduction.

As mentioned in Sec. 2, the each and foreach statements can only be executed in parallel when there is no observable interference between statements or iterations. Such loops are annotated with the independent annotation.

6 Comparison with HPF

Since Spar integrates the parallelization models of both HPF and OpenMP, it is more general than either of them. Nevertheless, it is useful to compare the data-parallel and the task-parallel subset of Spar with HPF and OpenMP, respectively.

Spar supports the same class of distributions as HPF 2.0, but uses a significantly different notation. In HPF a distribution annotation is used that is more limited than the Spar version. Whereas in Spar the annotation (cyclic a*i+b) is allowed, in HPF only the annotation (cyclic i) is allowed. However, HPF provides *alignment* statements to specify a distribution relative to another distribution. Through this mechanism the equivalent of distribution (cyclic a*i+b) can be specified.

The absence of alignment makes distributions somewhat less convenient to express, but we think that this is a small inconvenience. On the other hand, distributions like `(cyclic a*i+b)` can be expressed more conveniently than in HPF, since no alignment construct is necessary. The `block`, `cyclic`, and `blockcyclic` distributions can also be used for the efficient distribution of loops.

Spar currently supports exactly one processor array, whereas HPF supports an arbitrary number of processor arrays. However, in HPF the correspondence between processors in the different processor arrays is not fully specified, which makes it difficult to write efficient programs that 'mix' processor arrays. Although not shown in this paper, the Spar annotation mechanism can easily be extended to allow 'views' on the processor array, so that, for example, a 4×4 processor array can also be seen as a one-dimensional array of 16 processors. This would provide similar functionality to the multiple processor arrays of HPF, but without the associated performance problems. We plan to incorporate this feature in a future version of the compiler.

At least conceptually, the `forall` loop construct of HPF requires that copies of the arrays used in the loop are made. However, in Java and Spar arrays are accessed through *references*, and several references can potentially point to the same array. It is therefore impossible to support a `forall` loop construct that still has easily understandable behavior, and that can be implemented efficiently. Instead, Spar defines the `foreach` statement, which *can* be implemented efficiently.

Also, the body of a HPF `forall` may only contain assignment statements, and these statements may only assign to array elements. In contrast, the `foreach` statement allows arbitrary loop bodies. In particular, the `foreach` allows reduction operations, see Sec. 5.

7 The compiler

To evaluate the practicality of our language extensions, we have implemented a compiler for Spar/Java. This compiler is called 'Timber'. It is available for downloading at [15]. It implements Java as described in the Java Language Specification 2nd edition [9], except for threads and dynamic class loading.

The compiler works on full programs. It consists of a frontend that generates code in the intermediate language Vnus, a number of parallelization engines that work on Vnus, and a backend that converts from Vnus to C++. The compiler contains optimizers that avoid compiling unused code, eliminate and optimize null pointer checks, eliminate static class administration, to simplify expressions, and to optimize and eliminate bound checks. Generating C++ has as advantage that the compiler is independent of the target processor, and hence is very portable. It has as disadvantage that the optimizers in the C++ compiler are not tuned to Spar/Java programs, and sometimes must make too conservative assumptions about code properties. For example, alias analysis for general C++ programs must be much more conservative than is necessary for our C++ code.

The parallelization engines generate SPMD (Single Program Multiple Data) code with explicit message passing. For this reason we can use almost any communication library. We currently support PVM, MPI, and Panda [18].

The current experimental implementation has full support for data-parallel programs. The support for task-parallel annotation is still somewhat restricted. Full support for task-parallel programming would require the integration of a Virtual Shared-Memory system as fall-back, and optimization of common cases.

8 Results

To measure the performance of the experimental compiler, we have implemented three benchmarks from the NASA Numerical Aerospace Simulation group (NAS) parallel benchmark suite Version 2.3 [12, 8].

Compiler	Language	Time	Time (no checks)
Timber	Java	34.8	24.0
Timber	Seq. Spar/Java	24.5	21.4
Timber	Par. Spar/Java	36.7	31.8
PGI HPF	HPF	–	48.1
PGI F90	F77	–	36.3
g++	C++	–	16.3
gcc	C	–	13.0

Figure 1: Execution times for sequential versions of the various NAS FT implementations using a $128 \times 128 \times 32$ array. All execution times are elapsed times in seconds. Where relevant, times with bounds checking and null pointer checks enabled, and with both disabled, are shown.

Guided by the HPF and Fortran 77 versions we have available, the algorithms were re-implemented in Spar/Java.

For the benchmarks, the original NAS implementations for Fortran 77/MPI were used as reference. For the FT and MG benchmarks the HPF version was also used as reference. For the EP benchmark no HPF version was available.

All measurements were done on the DAS [6] distributed supercomputer. Each node has a 200MHz Pentium Pro; and a Myrinet interface, and runs Linux. For sequential programs a single node on a single cluster was used; for parallel programs multiple nodes on a single cluster were used.

The Fortran/MPI implementation was compiled with the PGI Fortran 90 Version 3.1 compiler. The HPF version was compiled with the PGI HPF Version 3.1 compiler. All versions, including the Spar/Java version, used the communication library Panda [18]. The Fortran/MPI and HPF version used the MPI library implemented on top of Panda, the Spar/Java version used Panda directly.

Due to their method of implementation, the Fortran/MPI version of FT and MG can only be run for a number of processors that is a power of 2. The HPF and Spar/Java versions do not have this restriction.

8.1 Sequential performance of the compiler

To place the parallel performance of the compilers in context, we evaluate the sequential performance of the Timber compiler, the NAS FT benchmark was implemented in C, Java, and Spar/Java, and the execution time was measured, see Fig. 1.

All measurements for the Timber compiler were done twice: once with all required checking enabled, and once with bounds- and null pointer checking disabled. This shows the overhead of these checks. The unchecked versions arguably are better when comparing with Fortran and C++ versions, since in these implementations these checks are absent too.

In the Spar/Java version, type `complex` and 3-dimensional arrays were used. In the Java version nested arrays were used, and complex numbers were represented by pairs of `doubles`, stored in adjacent elements of the array. Apart from these aspects, the Spar/Java and Java versions are the same. Since the representation of complex numbers is not likely to impact the execution times significantly, the slower execution times for the Java version are mainly attributable to the use of nested arrays instead of true 3-dimensional arrays.

The parallel Spar/Java version contains an extra matrix transposition that is essential for parallel execution, but slows down sequential execution.

The difference between the Spar/Java and the g++ version is caused by the different representation of Spar/Java arrays: as a pointer to an array descriptor; with a pointer to a block of elements as one of the

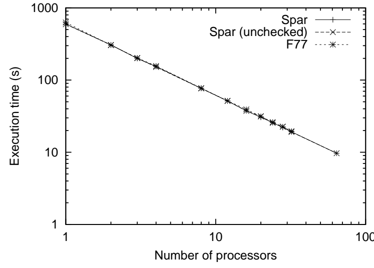

Figure 2: Results for the NAS EP 'A' dataset (2^{28} samples).

fields. In contrast, in the C and C++ versions arrays are represented by just a pointer to a block of elements. The difference between the g++ and gcc versions is caused by the use of the standard `complex` class respectively the type `__complex__` that is a language extension of GNU C. As Philipssen and Günthner have shown, a similar optimization is possible on the Java version [14].

8.2 NAS EP benchmark

The NAS EP benchmark approximates π by choosing random points in a square, and counting the percentage that falls in the inscribed circle of the square.

To parallelize the code, the calculations are divided in batches. The result for each batch (the percentage of 'hits') is stored in an element of an array; the global result can then be calculated with a reduction operation on this array.

The Spar/Java version contains the following annotations: a distribution annotation for the results array; a distribution annotation for the loop that iterates over the batches; and an annotation to label the result reduction loops as reductions. From these annotations only, the compiler is able to generate a parallel program, with the results shown in Fig. 2). Note that both data and task distribution is used. The Spar/Java program scales just as well as the Fortran/MPI version, and is slightly faster.

8.3 NAS FT benchmark

The NAS FT benchmark performs a 3D Fast Fourier Transform (FFT). The Spar/Java version distributes the FFT array in *cyclic* fashion in one dimension. Thus, if a $64 \times 64 \times 64$ array is distributed over 2 processors, each processor gets a $64 \times 64 \times 32$ elements slice of the array. For 4 processors the slice is $64 \times 64 \times 16$, and so on. In the original NAS benchmark program, The 3D FFT is implemented by doing a 1D FFT on all one-dimensional array sections parallel to the x axis, then parallel to y axis and then the z axis.

The Spar/Java version always performs 1D FFT transforms on array sections parallel to the x axis. For the last two phases, it copies and permutes the elements in the original array to a suitably distributed temporary array, performs the 1D transform, and copies the elements back again. Since we distribute the temporary array in a different dimension, all elements for the 1D FFT transform on one array section are always locally available, which allows it to be implemented very efficiently.

The Spar/Java version contains the following annotations: distribution annotations for the distributed arrays; a distribution annotation for the iteration over the 1D FFT transform; a `reduction` annotation on the loop that calculates the verification checksum, and a number of `independent` annotations. Using these annotations only, the compiler generates a parallel program, with the results shown in Fig. 3. Again, note that both task and data distributions were used.

In most cases the Spar/Java results compare favorably with those for Fortran/MPI and HPF. For larger

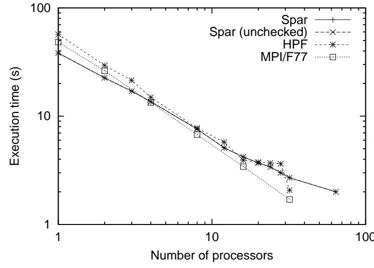

Figure 3: Results for the NAS FT 'W' dataset ($128 \times 128 \times 32$ elements).

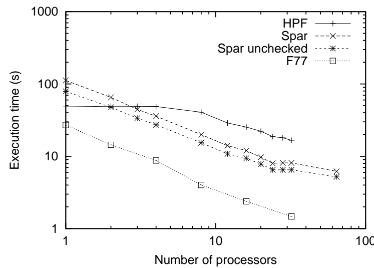

Figure 4: Results for the NAS MG 'W' dataset ($64 \times 64 \times 64$ elements).

number of processors secondary effects become significant, such as initialization and setup times, and load imbalance due to the distribution[2].

8.4 NAS MG benchmark

The NAS MG benchmark is a multigrid solver using grids of different coarseness. The Spar/Java version only uses distribution annotations for the different arrays; an annotation to label the loop that calculates the verification checksum as a reduction, and a number of annotations to label loops as 'independent'. Using these annotations only, the compiler generates a parallel program, with the results shown in Fig. 4. As these measurements indicate, the overhead of null pointer checking and bounds checking in the Spar/Java version is somewhat higher than for the other benchmarks. The performance difference with F77/MPI is caused by the larger number of copy operations on the data, and an inefficient use of the available cache hardware.

9 Related work

We have already compared Spar/Java to HPF in detail in Sec. 6. Spar is not the only proposal to extend Java for scientific computing. HPJava [4, 21] adds multi-dimensional arrays and data-parallel programming to Java. In contrast to Spar, the programs are explicitly data-parallel. That is, any data transfer between processors has to be explicitly done in the program by a call to the HPJava communication library.

Blount, Chatterjee, and Philippsen [2] describe a compiler that extends Java with a `forall` statement. To execute the `forall` statement, the compiler spawns a Java thread on each processor, and the

[2]For example, if an array of 64 elements is distributed over 24 processors, some processors get 2 elements, and some processors get 3 elements.

iterations are evenly distributed over these threads. Synchronization between iterations is done by the user using the standard Java synchronization mechanism. No explicit communication is performed; a shared-memory system is assumed. Due to the dynamic nature of the implementation, they can easily handle irregular data and nested parallelism. The paper also mentions a library of collection classes, but few details are given.

Titanium [20] provides vectors, multidimensional arrays, iteration ranges, and a `foreach` statement comparable to those in Spar. In most cases the Spar version of these constructs is more general. They explicitly state that their `foreach` is not intended for parallelization. Titanium supports iterations over arbitrary sets; moreover these iteration ranges are 'first-class citizens'; they can be handled and modified independent of any loop statements. Unsurprisingly, support for arbitrary iteration sets leads to inefficiencies. Therefore, Titanium provides a separate representation for rectangular iteration. In contrast, Spar supports 'classical' iteration sets that can be expressed as nested iteration ranges with strides. Spar's iteration ranges are more general than the rectangular iteration sets of Titanium, and can be implemented just as efficiently.

10 Conclusions

This paper presents language extensions to Java to allow annotation-based data and task parallel programming. To evaluate the effectiveness of our language extensions, we have implemented them in our experimental compiler. Using this compiler we have compared the performance of a number of scientific benchmark programs implemented in Spar/Java with two more traditional implementations in Fortran/MPI and in HPF.

As the results of our demonstration compiler show, we are able to compile the data-parallel side of our extensions, and part of the task-parallel side, into very effective parallel code. Effective compilation of the full set of extensions requires more work, though. In it's simplest form a Virtual Shared Memory (VSM) system could be used, but we expect very poor performance from such a configuration. We expect that significant improvement is possible by letting the compiler generate explicit code for shared-memory administration, and let the compiler optimize this as much as possible. This would give the compiler opportunity to aggregate or eliminate communication, serialization, and synchronization code.

References

[1] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: a language for parallel programming on distributred systems. *IEEE Trans. on Software Engineering*, 18(3):'90–205, Mar. 1992.

[2] B.Blount, S. Chatterjee, and M. Philippsen. Irregular parallel algorithms in Java. In *Irregular'99*, Apr. 1999.

[3] A. Bik and D. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, June 1997.

[4] B. Carpenter, Y. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with "HPJava". *Concurrency, Practice and Experience*, 9(6):633–648, June 1997.

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

[6] DAS website. www.asci.tudelft.nl/das/das.shtml.

[7] W. Denissen. *Design of an HPF Compiler: A compilation Framework for a Data-parallel Language*. PhD thesis, Delft University of Technology, 2000. ISBN 90-6464-197-8. Available at: pds.twi.tudelft.nl/pubs/ph_d/Denissen.pdf.

[8] M. Frumkin, H. Jin, and J. Yan. Implementation of NAS parallel benchmarks in high performance fortran. In *IPPS*, 1999.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Reading, Massachusetts, June 2000.

[10] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 2.0 edition, Feb. 1997.

[11] M. Leair, J. Merlin, S. Nakamoto, V. Schuster, and M. Wolfe. Distributed OMP – a programming model for SMP clusters. In *CPC*, pages 229–238, Aussois, France, Jan. 2000.

[12] NAS parallel benchmarks website. www.nas.nasa.gov/Software/NPB.

[13] *OpenMP Fortran Application Program Interface 2.0*, Nov. 2000. Available from www.openmp.org.

[14] M. Philippsen and E. Günthner. Complex numbers for Java. *Concurrency: Practice and Experience*, 12(6):477–491, May 2000.

[15] C. v. Reeuwijk. Timber download page. www.pds.twi.tudelft.nl/timber/downloading.html.

[16] C. v. Reeuwijk, W. Denissen, H. Sips, and E. 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, Sept. 1996.

[17] C. v. Reeuwijk, F. Kuijlman, and H. Sips. Spar: an extension of Java for scientific computation. In *ACM Java Grande – ISCOPE Conference*, pages 58–67, June 2001.

[18] T. Rühl, H. Bal, R. Bhoudjang, K. Langendoen, and G. Benson. Experience with a portability layer for implementing parallel programming systems. In *Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Aug. 1996.

[19] J. Subhlok and B. Yang. A new model for intergrated nested task and data parallel programming. In *PPoPP*, Las Vegas, June 1997.

[20] K. Yelick, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. C., and A. Aiken. Titanium: a high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, pages 1–13, Feb. 1998.

[21] G. Zhang, B. Carpenter, G. Fox, X. Li, and Y. Wen. Considerations in HPJava language design and implementation. In *Proc. of the 11th International Workshop on Languages and Compilers for Parallel Computing*, pages 18–33. Springer, Aug. 1998.