Delft University of Technology
Parallel and Distributed Systems Report Series

Parallel Expressiveness of the Spar Programming Language
Parallel Benchmark kernels in Spar/Java

S.V. Niemeijer

report number PDS-2000-006

ISSN 1387-2109
Parallel Expressiveness of the Spar Programming Language

Implementation of the NAS Parallel Benchmark kernels in Spar/Java

S.V. Niemeijer

18th May 2000
Preface

From my second year at University I was already drawn to the field of Parallel Computing and Numerical Analysis. At that time there was also a professor within the Applied Mathematics Department, prof. L. Dekker, who was conducting research in this field. Unfortunately, at the time I was ready to start my final research, prof. Dekker had become a professor emeritus. A blessing in disguise, however, was that at the same time a new professor for Parallel and Distributed Systems (PDS), prof. H.J. Sips, had come to the Computer Science department. Together with prof. Sips and my mentor H.-X. Lin, a former member of the section of prof. Dekker, we decided on a research project.

One of the current research projects at PDS consist of the development of a new parallel language/compiler called Spar. This language was designed to describe parallelism on a high level, like in HPF, but with a Java-like frontend. My research consisted of reviewing the parallel expressiveness of Spar from a user perspective, in this case as a Numerical Analyst. Furthermore, it was also ‘appreciated’, if possible, if some benchmark timings with the current version of the compiler could be retrieved. The document before you, my final thesis, describes the research I have done and the findings that resulted from this research.

Next to my professor H.J. Sips and mentor H.-X. Lin, I would like to thank the following people: Michael Frumkin from NAS for supplying the HPF implementations of the NPB, Frits Kuijiman for the parallel benchmark results of the Spar-NPB implementations, Kees van Reeuwijk for his swift feedback on and implementation of my suggested ‘improvements’ for the Spar frontend, and Sidney Cadot for the joyfull times at our room on the second floor.
How to read this document

Although it is rather uncommon to have a section like this in a Thesis, I believe it to be an indis-
pendable part. In a Thesis, like the one before you, some conventions for style and structure are
used and some information was specifically included or left out. I hope the reader will gain from
the following remarks and thus will find his way more quickly in this document.

Spar code and annotations

Throughout the entire document and in the appendix samples of Spar code and actual Spar
programs are included. At the time of this writing, the annotations as described in Chapter 2
were not yet implemented and it is highly possible that these definitions will be changing in the
near future. For these reasons I have omitted the annotations from the Spar sourcecodes in this
document, except, of course, for the places where the usage of annotations was explicitly needed
to illustrate its working. What annotations need to be used for a piece of code are described in
a general form in the text that refers to the code. For instance, when a 1D array needs to have
a \(<$ \text{ on=}(\text{lambda (i) Gpp[(cyclic i 10)])} \text{ >}$ annotation the text will state that a cyclic
distribution with blocksize 10 needs to be used.

Conventions

- Usage of for all
  Normally in a pseudocode algorithm the loop for all is used for constructs like:
  \[
  S = \{1,2,3,4,5,6,7,8,9\} \\
  \text{for all } e \in S \text{ do} \\
  \quad \text{ProcessElement}(e) \\
  \text{end for}
  \]
  In this document, however, the for all is used to indicate a loop with the same semantics
  as the foreach from Spar.

- Difference between ‘Class’ and ‘class’
  Because a family of objects in Java and a set of a specific benchmark size in the NPB are
  both denoted by the word ‘class’, in this document the word is written with a capital ‘C’
  when it denotes a benchmark class from the NPB and with a small-caps ‘c’ otherwise.

Using this document as a reference

How to install the Spar-NPB

Readers who want to get the Spar implementation of the NAS Parallel Benchmarks running need
to refer to the README in the Spar-NPB package. The package can be retrieved from [11].
How to implement and adapt the Spar-NPB

Because of the ongoing research on the Spar compiler it may very well be the case that parts of the Spar-NPB need to be changed in order to get them working with new versions of the Spar compiler. Furthermore, because annotations are not yet included in the Spar-NPB code, it might be necessary for people who want to include those annotations to have a better understanding of the code. Because the descriptions of the kernel benchmarks in [3] are sometimes not that clear I have tried to provide a more detailed description. In Chapter 3 each benchmark has been given a separate section where both the mathematical background, the elaboration on the description and the implementation issues are described. These sections however do not contain the specific parameters for each benchmark Class. For this information, the reader is referred to the ClassData files in the Spar-NPB package [11]. Also, for an explanation of the file structure, configuration options and other issues that involve compilation of the Spar-NPB benchmarks, the reader should refer to the README in this package.
Contents

1 Introduction ........................................ 4
   1.1 Research Goal .................................. 4
   1.2 Parallel Programming Languages .............. 4
   1.3 Java for High Performance Computing .......... 5
   1.4 Parallel Benchmarks ............................ 5

2 Spar .................................................. 7
   2.1 Multidimensional Arrays ......................... 7
   2.2 Complex Numbers .................................. 8
   2.3 Parameterized Classes and Interfaces .......... 8
   2.4 Parallel Constructs ............................... 9
   2.5 Annotations ...................................... 11
   2.6 Current Status ................................... 11
   2.7 A Quick Comparison With HPF .................. 12

3 NAS Parallel Benchmarks ............................ 14
   3.1 Design Issues and Other Considerations for Implementation ... 15
   3.2 Implementing EP - Embarrassingly Parallel Benchmark .......... 18
   3.3 Implementing MG - Multi Grid Benchmark ................. 21
   3.4 Implementing CG - Conjugate Gradient Benchmark .......... 27
   3.5 Implementing FT - Fast Fourier Transform Benchmark ... 34
   3.6 Implementing IS - Integer Sort Benchmark ................ 39

4 Other Algorithms .................................... 45
   4.1 One Dimensional FFT ................................ 45
   4.2 Merge Sort ....................................... 48
   4.3 Poisson Solver .................................... 52
   4.4 Miscellaneous Algorithms and Programming Aspects .......... 54

5 Performance Results .................................. 58

6 Advantages, Disadvantages, and General Issues .......... 65
   6.1 Advantages ....................................... 65
   6.2 Disadvantages .................................... 65
   6.3 General Issues .................................... 66

7 Recommendations ..................................... 68
   7.1 Recommended Additions and Changes for Spar .......... 68
   7.2 Research Follow-up ................................ 69

8 Conclusions ........................................ 71
CONTENTS

A Spar-NPB source codes
A.1 EP.spar .............................................................. 74
A.2 MG.spar .............................................................. 76
A.3 CG.spar .............................................................. 83
A.4 FT.spar .............................................................. 90
A.5 IS.spar .............................................................. 95

B Source codes of other algorithms
B.1 DoFFT.spar .......................................................... 98
B.2 DoMergeSort.spar .................................................. 102
B.3 Poisson.spar ........................................................ 104

C Multi Dimension Divisor .............................................. 106
# List of Algorithms

1. \( r = m^k s \pmod{2^{46}} \) ........................................... 19
2. V-cycle multi grid solver (1) ........................................ 21
3. V-cycle multi grid solver (2) ........................................ 22
4. MG.spar - Initialization of \( u \) and \( v \) ......................... 23
5. MG.spar - EvaluateResidual() ..................................... 24
6. Conjugate Gradient .................................................. 27
7. Inverse Power method for calculating largest eigenvalue ...... 28
8. Prescribed CG method to calculate \( z \) from \( Ax = x \) ....... 28
9. Prescribed CG method for \( b = x \) and \( x = z \) (i.e. solve \( Ax = b \)) .................................................. 30
10. 3D FFT of \( u \) in terms of 1D FFTs ................................. 34
11. Stockham FFT, for \( N = 2^M \) ..................................... 35
12. FT - Main loop ....................................................... 36
13. Initialization of \( u_{i,j,k} \) at \( t = 0 \) .............................. 36
14. FT - ComputeInitialConditions .................................... 38
15. IS - Main code ...................................................... 39
16. IS - Rank ............................................................ 41
17. IS - Rank - Accumulate ........................................... 42
18. IS - Rank - Determine key-rank from key-count .............. 43
19. IS - FullVerify ..................................................... 44
20. 1D FFT .............................................................. 47
21. Binary Recursive Merge Sort .................................... 48
22. Iterative Merge Sort ............................................... 49
23. Annotated recursive calls ....................................... 52
24. Jacobi iterative solver with red-black relaxation .......... 53
25. Sequential search routine ........................................ 54
26. Parallel search routine (1) ....................................... 54
27. Parallel search routine (2) ....................................... 55
28. Output dependent on amount of processors .................... 56
29. Farmer-Worker distribution ...................................... 57
Chapter 1

Introduction

1.1 Research Goal

The goal was set to be an investigation into the parallel expressiveness of the Spar language from a user perspective; in this case the viewpoint of a mathematical numerical analyst. Several algorithms from the field of numerical computing and possibly other fields would therefore have to be implemented. If shortcomings had been found then, if possible, recommendations had to be made to overcome these issues. Furthermore, if the compiler would be in a state that timing-benchmarks could be done with these algorithms, then the option was to include these timings in the research project.

1.2 Parallel Programming Languages

There are several types of parallel programming languages and tools that try to make it easier for the programmer to build parallel programs. Most of them are based on the well known sequential languages of today, like C, Fortran, and Java. The first approach that is widely used because of it’s flexibility and portability is the usage of communication libraries like MPI (See [15]) and PVM (See [6]). These libraries give the programmer the availability to functions like send, receive, broadcast, and barrier to communicate data from one processor to the other and to place synchronization points in the code. The programmer himself has to make sure that all data is communicated when necessary and has to take care of issues like deadlock and resource contention. This approach is very portable, since only the MPI library has to be ported to another parallel hardware platform and the application source code can generally stay the same.

On the other hand, the message passing paradigm is not that useful for shared memory systems. For these systems other libraries exist. Because shared memory programming doesn’t require explicit communication statements it is regarded as a higher level of parallel programming (it is more common that one implements a shared memory library on a distributed memory architecture than the other way around).

Although distributed memory libraries like MPI are very flexible they often require a lot of extra code in the program (especially for the setup of the processor connectivity structure and communication datastructures) and programs are often very hard to debug. A solution to these shortcomings, at an even higher level than the shared memory approach, was the introduction of data-parallel programming languages like HPF (See [8]). In these languages one provides distributions for large datastructures with, for instance, annotation constructs. Whenever an operation for each of the data elements of a distributed datastructure has to be performed a construct like foreach or forall tells the compiler to try to perform these loop elements in parallel. All necessary communication that is needed to perform the operations is than handled by the compiler. Although this data-parallel approach is very efficient for certain types of algorithms, some algorithms can not be expressed with this scheme. One can see that over the years people have tried
to expand the amount of supported parallelism in data-parallel languages. For instance, HPF has embedded task-parallelism since version 2 of its specifications (See [9]).

In this document we look at some of the shortcomings and advantages of a new high-level parallel language/compiler called $\texttt{Spar}$. $\texttt{Spar}$ is a parallel language that is very similar to HPF, but with some distinctive differences. A very important difference is that it is not an extension to Fortran but to Java (See [7]). Java has lately become a very popular language and within all kinds of fields the applicability of this language is investigated. In the next section we will look at the current use of Java in the field of High Performance Computing (HPC) and in Chapter 2 we will get more into the details of $\texttt{Spar}$.

1.3 Java for High Performance Computing

Although Java was not intended as a HPC language, but more as a language for Internet applications, it has gotten a wide interest from scientific research groups. There is even a forum called Java Grande$^1$ that is dedicated to the application of Java for scientific computing. A common factor that can be found in the research performed by these groups is that Java currently lacks some essential features. For instance all groups try to implement support for multi-dimensional arrays and complex numbers. The first one is very useful for representation of matrices and multi-dimensional grids and it is very convenient to have complex numbers available as native types instead of dealing with them through objects and classes. Furthermore, several groups perform research on parallel computation with Java. For parallelization several options are available

- **Threads**
  This is the most obvious one, since threads are already a part of Java. At the NASA Ames Research Center they have used this approach to implement several algorithms from the NAS Parallel Benchmarks (See [14]).

- **MPI or other communication libraries**
  Because of the broad scale applicability many types of communication libraries have already been developed. In [10] some design issues for implementing an MPI library in Java are discussed and the document also contains references to currently developed communication libraries for Java.

- **Extending Java with foreach or forall**
  This is the approach that was taken by $\texttt{Spar}$. However $\texttt{Spar}$ is not the only proposal in this direction.

In [5] a method is proposed to incorporate nested data parallelism into Java by using a $\texttt{forall}$ statement. This statement and the aggregate functions (reductions) are implemented using inner classes and threads. However, they assume a shared memory system, so no communication is performed.

Another project is the Titanium compiler (See [17]). They propose a $\texttt{foreach}$ statement similar to $\texttt{Spar}$ and provide vectors, multidimensional arrays, and iteration ranges. However, communication and synchronization needs to be done explicitly with constructs like $\texttt{exchange}$, $\texttt{broadcast}$, and $\texttt{barrier}$.

1.4 Parallel Benchmarks

Measuring parallel performance can be done at different levels. The first level consists of parallel benchmarks that concentrate on measuring communication performance. These are the low-level parallel benchmarks and are usually used to test the performance of supercomputing architectures or implementations of communication libraries like MPI and PVM. Although these kinds of

---

benchmarks are useful to measure the performance of the communication back-end used by Spar
they are not suitable to test the total performance of the Spar compiler.
The second level of benchmarks consists of basic types of operations that are usually performed
in High Performance Computing applications. An example of such a benchmark is a parallel
implementation of BLAS (See [12]). This benchmark contains the standard linear operations like
multiplication of a vector by a scalar, vector addition, matrix-vector multiplication, and matrix-
matrix multiplication. These operations can be performed on both dense matrices and different
kinds of sparse matrices.
One level higher we find the application kernels. Kernels are the highly computational parts of
an application and thus should gain most from parallelization. These types of algorithms are
often found in parallel benchmark sets, because they are small enough to be implemented in a
short timeframe and provide a means of comparison for both parallel architectures as well as
programming frameworks (i.e. language, compiler, and communication libraries).
Finally at the highest level we have the full applications. Although it takes more work to imple-
ment these kind of benchmarks then the kernel benchmarks, it will provide you with the actual
performance results of using the application for everyday problems.

Although we could have chosen several independent algorithms to implement within Spar and thus
investigate the parallel expressiveness, it would be useful to have a set of algorithms that later
on could also be used for performance measurement. Therefore the set of possible algorithms
was confined to the collection of currently available benchmarks suites. Several of them were
considered.
First of all LAPACK (See [2]). LAPACK is a set of linear algebra routines, like for instance
eigenvalue decomposition, LU decomposition, etc. The set is however not really a benchmark
set, but more a math library. The set consists of sequential algorithms and, although individual
algorithms from LAPACK are often parallelized and used as a benchmark, the total LAPACK set
is generally not considered as a benchmark suite.
Secondly the NAS Parallel Benchmark suite (NPB) was considered. The main advantage of the
NPB compared to other benchmarks, is that the benchmarks are specified on a basic algorithmic
level (another term often used is ‘pen and paper specifications’). This means that only the rough
outlines of the algorithm are specified and one has broad freedom in how one wants to implement
this algorithm on a given combination of programming language and parallel hardware architec-
ture. It also means that a better performance comparison between two complementary hardware
architectures (for instance shared memory versus distributed memory) or two different parallel
language approaches (for instance the HPF data parallel model versus low level message passing
through MPI or PVM) can now be made.
The Genesis Distributed Memory Benchmarks (See [1]) could also have been a good suite to
(partly) implement, because it contains benchmarks from all the four classes like mentioned before.
Unfortunately there was some problem at the time to retrieve the source code of this suite to be
able to use it as a starting point, so this suite became less favorable compared to the NPB.
Finally the Java Grande benchmark suite was considered. Although it is a well-rounded benchmark
suite, it is not specifically intended as a parallel benchmark suite and therefore preference was given
to benchmarks like NPB and Genesis.
Because of the arguments mentioned above the final choice for the benchmark suite to implement
fell on the NPB. The description and implementation is described in Chapter 3.
Chapter 2

Spar

Spar is a set of language extensions to Java to support high-performance computation. Although in this document the term Spar is often used to denote the total package of Java and the Spar extensions, Spar officially only comprises the following elements: multi-dimensional arrays, complex numbers, parameterized classes and interfaces, parallel constructs, and annotations. Next to the Spar language, there is also the Spar compiler. The Spar compiler is a compiler that transforms Spar/Java code to native binaries for heterogeneous systems. The compiler does this in three steps. First it translates the Spar/Java code to an intermediate language called Vnus. At this level all necessary communication and synchronization is inserted. Then the Vnus code is translated into C(++) code with communication primitives from libraries like MPI, PVM, or Panda. And finally the C code is translated into native binaries for each of the appointed processor types (for instance, one for a Pentium and one for a set of DSPs).

Below, each of the Spar extensions is briefly described. For a full description of the language see [16].

2.1 Multidimensional Arrays

Spar provides multi-dimensional arrays in the following fashion

```java
1    double[*][*] A = new double[100,100];
2    A[0,0] = 1.234;
```

This example creates a $100 \times 100$ matrix with its first element ($A_{1,1}$) set to 1.234. For pointing to a specific index in a multidimensional array Spar also provides vectors. Vectors are native fixed-length array types. For instance, in the previous example we could have used a vector $v$ and write

```java
1    double[*][*] A = new double[100,100];
2    [int,2] v = [0,0];
3    A @v = 1.234;
```

As an extension to this, Spar also supports multi-typed vectors, which are called tuples. The following example illustrates the usage of a tuple $t$. This piece of code has the same result for $A$ as the previous examples.

```java
1    double[*][*] A;
2    [int,double,type double[*][*]] t;
3    t[0] = 1;
4    t[1] = 0.234;
5    t[2] = new double[100,100];
6    t[2][0,0] = t[0]+t[1];
7    A = t[2];
```
Of course tuples that contain elements not equal to an int can not be used to index a multi-dimensional array.

2.2 Complex Numbers

Complex numbers are represented as native types in Spar. For complex-specific operations on these numbers the Complex class can be used. For example in

```
1   complex z = 2 + 3i;
2   complex y = Complex.conj(z);
3   double x = (double) z * y;
```

x will attain the squared norm of z, which is 13.

2.3 Parameterized Classes and Interfaces

In Spar one can create classes and interfaces that depend on a not-yet determined type. It has thus become possible to create one class for a collection, like

```
1   class TypedCollection(| type t |) {
2       t[.] elements = new t[0];
3
4       void add( t obj ) {
5           ...
6       }
7
8       void remove( t obj ) {
9           ...
10      }
11
12      t find( t obj ) {
13          ...
14      }
15    ...
16 }
```

which than can be used for both doubles, integers, complex values, etc.

```
1   TypedCollection(| type double |) a =
2       new TypedCollection(| type double |)();
3   TypedCollection(| type int |) a = new TypedCollection(| type int |)();
```

2.4 Parallel Constructs

Spar provides two parallel constructs: each and foreach. The each is used to tell the compiler that all statements within the following block may be executed in any order. Thus in

```
1   each {
2       a = 1;
3       b = 2;
4   }
```

both the execution orders a=1; b=2; and b=2; a=1; are possible. This semantic does not mean that the two statements can be executed in parallel. For instance
is also valid, but raises a resource contention problem on the variable \( a \). The idea of an \texttt{each} is to give the compiler more information on the kind of independence between the different statements. In the future, the compiler should be able to recognize the absence of contention and perform the execution of the statements in parallel. However, currently, the compiler does not recognize this, furthermore there will always be cases in which the parallel independence can not be recognized by the compiler. For these reasons an \texttt{independent} annotation is available that is described in the next section.

The \texttt{foreach} statement is an \texttt{each} applied to a \texttt{for} loop. For instance, if we have a block distributed array \( x \) of length \( N \) then an initialization can be written as

\begin{align*}
1 & \quad \texttt{foreach}(i := 0:N) \, \{ \\
2 & \quad x[i] = 0; \\
3 & \} 
\end{align*}

The \texttt{foreach} has a different syntax and is more restricted compared to the \texttt{for} statement in Java. In the example, \( 0:N \) gives the range for the loop iteration variable \( i \), which means that for all values \( 0 \leq i < N \) (thus \( N \) not included) the loop-step will be performed and this may happen in any order.

The \texttt{foreach} also allows iteration over vectors, which means that for initialization of a \( N \times N \) matrix \( A \) we could write

\begin{align*}
1 & \quad \texttt{foreach}(v := [0,0],[N,N]) \, \{ \\
2 & \quad A@v = 0; \\
3 & \} 
\end{align*}

Because task parallelism can only occur at an \texttt{each} or \texttt{foreach} statement we are restricted to the use of Series Parallel programming. This means that implementation of a set of tasks with dependencies like in Figure 2.1 can not be done without introducing some extra dependencies (for instance between task 1 and 4).

### 2.5 Annotations

\texttt{Spar} provides annotations to give the compiler hints on how to parallelize data and code. This means that the compiler may also neglect annotations. Annotations are therefore only intended
to help the compiler optimize performance and should not change the semantics of the program.

2.5.1 Independent

The first annotation, which was already mentioned in the previous section, is the independent annotation. An independent attached to an each and foreach tells the compiler that the statements within { } can be performed independent of each other and thus in parallel. This annotation also immediately poses a problem with the rule of absence of semantic behavior. For instance if the compiler would acknowledge the independent annotation in the reduction

```plaintext
1   r = 0;
2   foreach(i := 0:N) <$ independent $> { 
3       r += x[i];
4   }
```

than the output for r could change. Take for instance N=2; x[0]=1; x[1]=2; If iteration steps i=0 and i=1 are performed in parallel then both steps start with r=0. Depending on which loop-step finishes first the value of r will either be 1 or 2, but in either case it will not be equal to the desired result of r=3. Therefore the semantics of the program has changed. Although it is not yet stated in the Spar manual, the use of independent has semantics, because in general the compiler will not be able to check whether the annotation is valid. The annotation should thus be used with care.

2.5.2 Reduction

To be able to do the reduction as described in the previous section, Spar provides a reduction annotation. The example will thus become

```plaintext
1   r = 0;
2   foreach(i := 0:N) <$ reduction $> { 
3       r += x[i];
4   }
```

In the future the compiler will also be able to recognize the reduction without the annotation, but only for straightforward reduction operators like +, -, *, /, and the somewhat less straightforward operations like Math.max() and Math.min().

2.5.3 ProcessorType and Processors

One thing that sets Spar apart from other parallel compilers is the fact that Spar targets heterogeneous systems. Therefore a global annotation is needed that tells the compiler what processor types should be used. Next we should tell the compiler how many processors we have for each type and in what configuration they are placed. For configuration only the dimensions of the processor-grid can be specified. Spar does not make any difference between the connectivity of one pair of processors or the other. This means that if we put a set of processors in a two dimensional grid in Spar and this set of processors is actually also physically arranged in a two dimensional grid, these grids do not necessarily have to agree. That is, if two processors are adjacent in the physical grid they do not have to be adjacent in the 2D grid in Spar. A small example that creates a set of one Pentium2 and 10 Trimedia's is

```plaintext
1   <$ ProcessorType=((Gpp “Pentium2”) (Dsp “Trimedia”)) $>
2   <$ Processors=((Gpp gpp1) (Dsp dsp1D[4]) (Dsp dsp2D[2,3])) $>
```

The strings "Pentium2" and "Trimedia" refer to configuration files that describe the details needed by the compiler for each processor type. The identifiers gpp1, dsp1D, and dsp2D can later on be used in on annotations to assign distributions.
2.5.4 On

The annotation that provides the distributions is the on annotation. This annotation instructs the compiler to distribute data and tasks among processors in a certain fashion. There are five different ways of distribution possible:

- On a single processor:
  
  ```$<$ on=gpp1 $>$ or $<$ on=dsp2D[0,0] $>$
  ```

- On all processors or all processors in a certain set:
  
  ```$<$ on=all $>$ or $<$ on=dsp1[,all] $>$
  ```

- A blocked distribution in one of the dimensions of a processor-set:

  ```
  1 foreach(i :- 0:100) $<$ on=dsp1D[(block @i 25)] $>$ { 
  2     a[i] = a[i] + i;
  3  }
  ```

  The assignment of a[i] is executed on dsp1D[i/25]. This means that we have a block distribution with a blocksize of 25.

- A (block-)cyclic distribution in one of the dimensions of a processor-set:

  ```
  1 foreach(i :- 0:100) $<$ on=dsp2D[(cyclic @i 25),all] $>$ { 
  2     a[i] = a[i] + i;
  3  }
  ```

  The first three processors of dsp2D will perform the steps for 0 ≤ i < 25, 50 ≤ i < 75 (they all do the same work) and the last three processors will do the steps for 25 ≤ i < 50 and 75 ≤ i < 100. Because we have a _all in the second dimension this means that the elements are collapsed in the second dimension of dsp2D (we map a lower dimensional grid to a higher dimensional processor grid).

- An unspecified distribution:
  
  ```$<$ on=_,_ $>$ or $<$ on=dsp1D[_,_] $>$
  ```

  When assigning this distribution the compiler is free to choose its own distribution. This annotation can also be used to have the compiler insert a run-time scheduler at this point.

Next to foreach statements these annotations can also be used on declarations, expressions, and the each statement. For the new expression a slightly adapted version of the on annotation can be used. With help of the lambda parameter we can express

```
  1     int[*,*] A = $<$ on=(lambda (i j) dsp2D[(block i 50),all]) $>$
  2     new int[100,100];
```

In the example A has a row-wise distribution. Because A and dsp2D are of equal dimension the _all in the second dimension of dsp2D in this case means that the elements are replicated in the second dimension of dsp2D.

2.6 Current Status

Of the items mentioned before, everything has been implemented except for the annotations. There are, however, also some features that will or might be supported by the compiler in the future¹.

¹While I was writing this document the array interfaces and macros became official features of Spar.
• array interfaces
It will be possible to create your own class that can make use of the subscript operators \( \[ \) and \( \] \) which are translated to \texttt{getElement} and \texttt{storeElement} methods. In this way it will be possible to hide the sparseness of a matrix in a self-defined array class or to create different views on the same matrix (for instance the transpose or the trace) with this class.

• macros
\texttt{Spar} will allow methods and classes to be defined as a macro, which means that these methods will be inlined. For instance, for the \texttt{getElement} and \texttt{storeElement} methods of a self-defined array class this can seriously speed up performance.

• partial compilation with multiple sourcecode files
Currently the compiler can only compile one sourcecode file. Although that file may include other source files, this still means all sourcecode will be recompiled. It is not yet possible to compile a \texttt{Spar} file to an objectfile and later on link all objectfiles into one executable.

• runtime scheduler
In the electro-technical engineering department a research group is currently developing a runtime scheduler that in the future may find its way into the \texttt{Spar} compiler. With some new annotations it will then be possible to express task parallelism that is not Series Parallel.

• garbage collection
Currently no garbage collection engine is available. To partly overcome this problem at the moment there is a \texttt{deleteO} function available with which one can manually remove datastructures from memory.

There are more planned extensions and modifications of \texttt{Spar}. These and some of the ones mentioned above are described in detail in [16].

2.7 A Quick Comparison With HPF

Although \texttt{Spar} and HPF do have a lot of similarities for the way they provide parallelization, they are not that equal. Below some differences and similarities are summarized.

• The distributions of \texttt{Spar} are similar to the kind of distributions that can be made with the \texttt{dist\,ute} annotation in HPF 2.0. Only \texttt{Spar} is more expressive, so the use of \texttt{templates} as in HPF is not needed. The absence of an \texttt{align} in \texttt{Spar} is however an omission if one wants to change the distribution. In HPF one has only to change the distribution of the template to change the distributions of the arrays that were aligned with this template. In \texttt{Spar} one has to change the distribution annotations for all the arrays.

• Another thing that \texttt{Spar} does not have is a \texttt{re\,dist\,ute} annotation. In HPF one can redistribute an array at any time in the code. In \texttt{Spar} one has to use a work around in which one introduces a new variable with the new distribution and copy the contents of the old array into the new array.

• The \texttt{for\,each} of \texttt{Spar} on the other hand is much more powerful than the \texttt{for\,all} in HPF. Although it takes much more analysis from the compiler to figure out if the \texttt{for\,each} can be performed in parallel, the parallel loop is not confined to the assignment of array elements like with the \texttt{for\,all}. Also, the \texttt{for\,each} can be used to express reductions, whereas in HPF the use of intrinsic functions is needed.

• Both languages/compilers require the use of an \texttt{in\,dependent} annotation to indicate that the calculations in one loop step do not influence the outcome of calculations in another loop step.
• Spar does not yet allow the creation of processor subsets to map tasks or data on. HPF already provides this feature through the use of a subset annotation.

• Spar also does not yet have an intrinsic NumberOfProcessors() like available in HPF.
Chapter 3

NAS Parallel Benchmarks

The NAS Parallel Benchmarks, described in [3], are a set of benchmarks that try to mimic the same computational and data movement characteristics as is found in large scale fluid dynamics applications.

Since version 2.0 of the NPB [4], NAS has provided an implementation of the benchmarks in Fortran 77 with MPI. The purpose of this implementation was to provide a portable set for comparison of distributed memory machines. According to their report [4] future releases may also contain portable versions for shared memory and/or vector architectures.

The NPB consists of five parallel kernels and three simulated applications, all intended to be implemented on a parallel architecture. The five kernel benchmarks are:

- **EP - Embarrassingly Parallel**
  This benchmark generates pairs of Gaussian random numbers and increases elements of a counter array according to a simple test on these numbers.

- **MG - Multi Grid**
  In this benchmark the 3D Poisson equation
  \[ \nabla^2 u = v \]  
  (3.1)
  is solved by using four iterations of the V-cycle multi grid algorithm.

- **CG - Conjugate Gradient**
  The CG benchmark uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non zeros.

- **FT - 3D Fast Fourier Transform**
  This benchmark uses forward and inverse FFTs to solve a given partial differential equation (PDE).

- **IS - Integer Sort**
  The goal of this benchmark is to sort N keys in parallel. This is an operation that is often used in ‘particle method’ codes.

The three applications all try to solve the 3D discretization of the Navier-Stokes equation \( K u = r \), with \( K \) being a 7 diagonal block matrix with 5 \times 5 blocks. Each application benchmark differs in the factoring of \( K \).

- **LU - LU solver**
  LU implements a version of SSOR by splitting \( K \) in an upper and lower triangular matrix
  \[ K = \omega(2 - \omega)(D + \omega Y)(I + \omega D^{-1}Z) \]
• **SP - Pentadiagonal solver**
  SP uses the Beam-Warming approximate factorization and Pulliam-Chaussee diagonalization with fourth order artificial dissipation

\[
K = T_x \cdot P_x \cdot T_x^{-1} \cdot T_y \cdot P_y \cdot T_y^{-1} \cdot T_z \cdot P_z \cdot T_z^{-1}
\]

\(T_x, T_y, T_z\) are block diagonal matrices and \(P_x, P_y, P_z\) are scalar pentadiagonal matrices.

• **BT - Block tridiagonal solver**
  Uses Alternating Direction Implicit (ADI) approximate factorization

\[
K = BT_x \cdot BT_y \cdot BT_z
\]

with \(BT_x, BT_y, BT_z\) being block tridiagonal matrices.

Of these benchmarks only the five kernel benchmarks have been ported to the Spar platform. The three applications were left out both because of the lack of time within this research project and because the kernel benchmarks already provide a decent overview of parallel numerical algorithms. The applications would not add much to the diversity of types of parallel algorithms and required parallel constructs.

Each benchmark can be run with a different test set. These are called ‘Classes’. Since version 2.0 there are three official benchmark Classes: A, B and C. These Classes can be defined by a set of parameters which define the scale of the used datasets and the corresponding verify values. As a quick test for running the NPB2.3 benchmarks as provided by NAS two extra Classes are provided. One is the sample class, called S, which should be able to run on even the simplest desktop and the other is the workstation class, called W. The W class was introduced, because class A (the least demanding of the official classes) was sometimes too demanding on resources when one wanted to run the benchmark on a single workstation. With the W class one can test the sequential performance differences between different kernel implementations without having the need for a high performance parallel computer.

### 3.1 Design Issues and Other Considerations for Implementation

Because the benchmarks are very loosely formulated we first have to formulate a general approach in which way the kernel benchmarks will be implemented. Furthermore, because Spar is a new parallel language, we will not be fully compliant with the rules that are set for the NPB. The deviation herein is also described in this section.

Next to the implementation of the kernels in Fortran 77 with MPI (with the exception of some which are in C or have Fortran 90 extensions) NAS also provided a sequential implementation of the NPB 2.3. Because there is no support for direct communication in Spar the sequential kernel benchmark implementations were chosen as a starting point. It would also have been possible to start from scratch and implement the Spar benchmarks from the original specifications, but there are several reasons why this approach was not chosen. First of all, the specifications of the kernel benchmarks as described in [3] were sometimes hard to follow, especially the initialization of matrices and starting vectors with random numbers\(^1\). Another reason is for comparison reasons. At PDS we are not only interested in the the parallel performance but also in the sequential performance. So therefore a maximal exploit of parallelism with a minimal change in the sequential code parts gives us the opportunity to reasonably compare both performances.

\(^1\)The setup for CG is very complex. So complex that NAS did not describe its behavior on paper but just supplied the Fortran code [i.e. a description in programming code].
In almost all kernel benchmarks NAS has inserted extra code to improve sequential speed. For instance for most iterative loops, one loop call is made before the timing begins just to initialize code page and data page tables. And sometimes an explicit loop-unroll has been applied. Because the behavior of the Spar compiler in this regard is not yet known, these sequential improvements were not transfered into the Spar implementation.

Another design issue is the usage of Object Oriented programming (OO). As opposed to Fortran and C, Spar/Java is an object oriented language. However there is still a lot of freedom in the amount in which one can make use of these OO techniques. There are several reasons why in this case (with some exceptions) the usage of OO has been kept to a minimum. First of all the OO paradigm is in general very useful for applications which can be modeled by objects and resemblance classes. The main area of application for this technique, user interfaces, fits this model quite nicely. The user interface elements can be divided into classes (like buttons, scrollbars, windows, etc.) and within each class the elements have large similarity but may differ to some extent in their look and behavior. For numerical mathematical applications on the other hand this model is less suitable. Classes that could be defined are for instance:

- Solvers: For instance, there are several solvers available to solve the classical $Ax = b$ problem in an iterative way. By defining each iterative solver as a subclass or through an interface, using a different iterative solver in your program could be as simple as changing one line of code in the program. But, because solvers are very coarse grained (they are often called only a few times within a program and usually only one or a few instances of a solver are created during the whole program) the advantages of OO in this case are minimal.

- Matrices: The place to use classes in mathematical oriented programs is of course matrices. Because matrices can have different type of sparseness, there are also different ways to efficiently store them in memory. This variety in storage methods can be conveniently concealed by use of OO. Spar, however, already provides such a mechanism, but in a slightly different way. As described in [16], one can use array interfaces to hide the sparseness structure of a matrix and deal with it as if it were a dense matrix. The implementation of array interfaces is done with macro classes which makes element retrieval a lot cheaper than an implementation with real objects encapsulating the matrix. So, for performance reasons, in the NPB implementation, matrices have always been implemented through the use of Spar arrays.

- BLAS routines: Next to providing an abstraction layer for matrices one could also use OO to build an abstract layer for standard operations on these matrices (BLAS level 2 and 3). Unfortunately there is a fundamental problem which prevents us from implementing this efficiently. In Chapter 6 I will come back on this fundamental issue.

Regarding the general rules for implementation it is inherently not possible to comply with all of them. The first part already states that the only languages that may be used are Fortran 77/90, HPF, or C. Further general rules concern which Fortran extensions may be used and some rules regarding code mixture and library usage. Since the essence of the port is not to provide an official implementation of the NPB, but to provide insight in the performance comparison between Spar and comparable languages/compilers these general rules were largely neglected. However, since Spar internally compiles to C++ code, which is parsed by a general C++ compiler, one could argue that we use C code. Anyhow, the implementation was in compliance with the restrictions that there is to be no mix of code from different languages (only Spar code was used), both the used libraries and compiler are freely available and all floating point arithmetic is done in 64-bit. Furthermore, compliance with the kernel specific rules was, because of the ‘pen and paper’ specification, not a problem.

There are some special considerations that were made regarding actual coding in Spar/Java and the current status of the compiler. First of all the compiler/runtime system as yet does not support garbage collection. This means that any memory block that is allocated with the new operator will
stay allocated until the program terminates. To overcome this problem a temporary solution is
provided in the form of the `delete()` function which deallocates memory used by the reference
that is passed to this function. In the benchmark implementations I only used `delete()` if a
memory block was used only locally within a method.
Because Class specific parameters that are included from a separate file are constants, the compiler
should be able to do operations done on these constants at compile time and thereby increase
runtime performance. To help the compiler recognize these parameters every variable that could
be declared final has been declared final. Also macro methods can gain dramatically in performance
by applying the final statement.
Another rather important issue is the scope of variables. This is because the location where the
variable is declared determines whether or not a reduction is recognized and performed by the
compiler and whether or not a foreach can be performed in parallel. For instance the foreach loop in

```java
1  int [] a = new int [1000];
2  foreach(i :- :100) {
3    int k = i*10;
4    for (j :- :10)
5      a[k++] = j;
6  }
```

can be performed in parallel without any problems, but when we use

```java
1  int [] a = new int [1000];
2  int k;
3  foreach(i :- :100) {
4    k = i*10;
5    for (j :- :10)
6      a[k++] = j;
7  }
```

each foreach step suddenly uses the same instance of k and the foreach loop can therefore not
be performed in parallel. If we would parallelize the loop then this will result in communication
to keep k in sync for all processors and running the program will result in unpredictable behavior
for the filling of a

Next, I will describe the implementations of each benchmark in detail. Issues that occur for more
than one benchmark are only worked out in detail within the section of the first benchmark where it
was encountered. This means that reading these sections in a chronological order is recommended.

\footnote{Later on it will be shown that this problem can sometimes also be solved by making use of a not yet introduced local annotation, but it is nevertheless good practice to declare a variable as local as possible.}
3.2 Implementing EP - Embarrassingly Parallel Benchmark

3.2.1 What it does

EP is the smallest benchmark in the set and is therefore the easiest to implement. It calculates \(N\) pairs \((x_j, y_j)\) of uniform pseudo-random floating point values between \(-1\) and \(1\), that represent coordinates within a square around the origin. By calculating \(t_j = x_j^2 + y_j^2\) the check is performed whether the coordinates are within a 1 unit radius \((t_j \leq 1)\). If this is the case an index \(k\) is increased and the pair \(X_k = x_j \sqrt{(-2 \log t_j)/t_j}, Y_k = y_j \sqrt{(-2 \log t_j)/t_j}\) is created. \(X_k\) and \(Y_k\) are Gaussian deviates with mean zero and variance one. With these values a data distribution \(Q_l\) is calculated by increasing \(Q_l\) for \(\{l; l \leq \max(\|X_k\|, \|Y_k\|) < l + 1, 0 \leq l \leq 9\}\). The total value list for \(Q_l\) should be printed in the end.

3.2.2 Class parameters

The parameters for this benchmark only consists of the value for \(M\), where the amount \(2^M\) represents the total number of random pairs generated, and two verification values.

3.2.3 Initialization

Because this benchmark does not use any predefined matrices or vectors there are no specific initialization issues.

3.2.4 Verification

Verification is done by calculating the sums \(s_x = \sum_k X_k\) and \(s_y = \sum_k Y_k\) and comparing them with the two Class specific verification values.

3.2.5 Implementation

A straightforward parallel implementation of this kernel would be to parallelize the \(j\)-loop and perform reductions on \(Q_l, s_x\) and \(s_y\). With implicit communication this would require only one gather at the end of the iteration loop. Unfortunately things are much more complicated because of the deterministic pseudo random number generation. In a general application one might not care for the determinism of the random generator. The starting seed may then be influenced by some ‘random’ factor like time or processor number and each different run of the program creates different output. With the NPB we have to comply with a verification scheme, which in this case are the summations \(s_x\) and \(s_y\). Furthermore, the rules of this particular benchmark require that each draw \((x_j, y_j)\) is done according to the given sequential random number generation scheme. So how can this benchmark then be ‘embarrassingly parallel’? Because of the size of \(N\) \((2^{28} \text{ for class A})\) it is not possible to sequentially calculate each pair in advance and then perform the calculations of \(t, Q,\) and \(s\) in parallel. Fortunately the random number generating algorithm provides a way to overcome this in a suitable fashion. I will describe this in the next section\(^3\).

The Random Number Generating Algorithm

Although the random number generation algorithm is described in detail in Section 2.3 of [3] it will be rephrased here to be able to better explain the solution for parallelization of EP.

The generation is done by providing the algorithm with a starting seed \(s\) and a multiplier \(m\). Internally it calculates pseudo random integers \(x\) which are then normalized to acquire the random

\(^3\) Although the solution is already provided in the Fortran implementation it is not described in any documentation and it is not very easy to reverse engineer. Furthermore, the use of a goto in this piece of Fortran code also requires a slightly different implementation in Spar.
numbers between 0 and 1. The first \( x \) is initialized to \( x_0 = s \). Each time the random number function is called it returns the next random number by calculating

\[
x_{k+1} = mx_k \pmod{2^{46}}
\]

(3.2)

and returning

\[
r_k = \frac{x_k}{2^{46}}
\]

(3.3)

So calculating the \( m \)-th random number is equal to

\[
r_k = \frac{m^k s \mod 2^{46}}{2^{46}}
\]

(3.4)

It is of course less costly to calculate \( r_k \) from \( x_{k-1} \) by using equations (3.2) and (3.3) then it is using equation (3.4). The drawback however is that for (3.2) all \( r_k \) need to be calculated in sequence, while each \( r_k \) in (3.4) can be calculated in parallel. To get the best of both for EP we divide the random numbers in batches which are distributed among the available processors. For each batch we use (3.4) to calculate the first random number of the sequence and the next numbers in the batch are calculated with (3.2) and (3.3). Algorithm (1) provides the pseudo code for calculating (3.4).4

<table>
<thead>
<tr>
<th>Algorithm 1 ( r = m^k s (\mod 2^{46}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q = m \mod 2^{46} )</td>
</tr>
<tr>
<td>( r = 1 )</td>
</tr>
<tr>
<td>( n = k )</td>
</tr>
<tr>
<td>( \textbf{while} \ n &gt; 1 \ \textbf{do} )</td>
</tr>
<tr>
<td>( \textbf{if} \ n \ \text{even then} )</td>
</tr>
<tr>
<td>( q = q^2 \mod 2^{46} )</td>
</tr>
<tr>
<td>( n = n/2 )</td>
</tr>
<tr>
<td>( \textbf{else} )</td>
</tr>
<tr>
<td>( r = rq \mod 2^{46} )</td>
</tr>
<tr>
<td>( n = n - 1 )</td>
</tr>
<tr>
<td>( \textbf{end if} )</td>
</tr>
<tr>
<td>( \textbf{end while} )</td>
</tr>
<tr>
<td>( r = rq \mod 2^{46} )</td>
</tr>
<tr>
<td>( r = rs \mod 2^{46} )</td>
</tr>
</tbody>
</table>

One important thing to mention is the storage of \( x_k \). Because we start with \( x_0 = s \) and do not need \( x_k \) anymore after we have calculated \( x_{k+1} \), in the implementation we will only use the variable \( s \) and just calculate \( s = ms \mod 2^{46} \), \( r = \frac{s}{2^{46}} \).

The implementation of this scheme in EP is now rather straightforward. The total of \( N \) random number couples are divided into \( B \) batches. Because \( N \) is a power of two we also choose a power of two for \( B \), so we will achieve equal batch sizes. The batches are processed in an outer \texttt{foreach} loop, which is distributed cyclic. A block distribution is not useful here, because first of all \texttt{Spar} does not provide a way to test for consecutive batches and eliminate the intermediate calls to the \texttt{skip} routine. Furthermore the calls to \texttt{skip} become more expensive as the batch index increases (\( k \) increases). So for load balancing purposes a processor should calculate both batches at the beginning of the batch list and batches which are more near the end. This can be achieved with the cyclic distribution.

---

4In the \texttt{Fortran} code this algorithm is part of the EP source code. In the other kernel benchmarks, however, this routine is also used, so I implemented this algorithm as a \texttt{skip} routine in the common source file \texttt{Rand on sps}.  

5As opposed to the number of batches that can be specified in the \texttt{Fortran} code, in \texttt{Spar} I have chosen to use the batch size as a parameter. It was set to 1024 but still needs to be tuned for optimal performance.
As mentioned in the general design issues, something that shouldn’t be overlooked is the scope of variables when using `for` each’s. Because the random number generation routine is provided in a separate file and internally uses only one variable $s$ to keep track of the value for $x_k$ we will have incorrect behavior if each processor starts working with this same instance of $s$. To overcome this problem the Random routines and accompanying state variables are implemented as a class. For each batch a new instance of the random generator is created on the processor that is processing the batch. In this way each batch/processor will have it’s own instance of the variable $s$ and correct output is guaranteed. Currently the drawback to this approach is that the performance penalty for creating new objects might be non-neglectable.

One final issue is reduction. EP contains three reductions for each pair of random numbers that are within the unit square. All of them can be written as ‘+=’ operators on a reduction variable and should therefore be recognized automatically by the compiler.
3.3 Implementing MG - Multi Grid Benchmark

3.3.1 What it does

Within the field of Numerical Partial Differential Equations we can distinguish between several classes of problems. First of all a differentiation can be made between time dependent and time invariant systems. Within the time invariant systems there is a further separation between elliptical, parabolic, and hyperbolic systems. An elliptical system can generally be described by

\[ -\partial_i a_{ij} \partial_j y + b_i \partial_i y + cy = F \]  

(3.5)

with the addition of a mixture of boundary conditions (Dirichlet, Neumann, mixed Robin, or periodic) on the edges \( \Gamma \) of our domain \( \Omega \). There are some restrictions on \( a_{ij}, b_i, \) and \( c \) for this set of equations in order to have a solution which is unique and stable for small changes in its parameters (these are the functions \( a, b, c, \) and \( F \) and the boundary conditions). A quite common and easy equation is Poisson's equation, which has \( a_{ij} = 1 \) \((i = j)\), \( a_{ij} = 0 \) \((i \neq j)\), \( b_i = 0 \), \( c = 0 \). In such a system, positive and negative values for \( F \) describe sources and sinks within \( \Omega \).

If we discretize the equations we retrieve for \( \Omega \)

\[ \nabla^2 u = v \]  

(3.6)

Where \( \nabla \) represents the discrete differential operator, \( u \) is the discrete vector representation of \( y \) and \( v \) the discrete version of \( F \).

This equation is precisely what is solved by the Multi Grid Benchmark. Furthermore the domain is considered to be 3D with periodic boundaries conditions. To solve this equation the V-cycle multi grid solver is used. In pseudo code this algorithm resembles Algorithm 2.

**Algorithm 2** V-cycle multi grid solver (1)

\[
\begin{align*}
    u &= 0 \\
    r_0 &= v - Au \\
    \text{for } k &= 0 \text{ to } K - 2 \text{ do} \\
    &\quad r_{k+1} = Pr_k \\
    \text{end for} \\
    z_{K-1} &= 0 \\
    z_{K-1} &= z_{K-1} + Sr_{K-1} \\
    \text{for } k &= K - 2 \text{ to } 1 \text{ step } -1 \text{ do} \\
    &\quad z_k = 0 \\
    &\quad z_k = Qz_{k+1} \\
    &\quad r_k = r_k - Az_k \\
    &\quad z_k = z_k + Sr_k \\
    \text{end for} \\
    z_0 &= Qz_1 \\
    r_0 &= v - Az_0 \\
    z_0 &= z_0 + Sr_0 \\
    u &= z_0 \\
    r_0 &= v - Au
\end{align*}
\]

Actually, because \( z_0 \) is equal to \( u \) every \( z_k \) \((k > 0)\) is named \( u_k \) within the Fortran code.

Strangely enough this pseudo code, which is reverse engineered from the Fortran code, is not equal to the specifications as described in [3]. Because in the specifications the line ‘\( u = z_0 \)’ is written as ‘\( u = u + z_0 \)’. The current NPB group at NAS unfortunately could not shed any light on this problem and because this difference does not really change the meaning of the type of calculations performed I chose to leave it at this and just use the algorithm as described in Algorithm 2. With \( z \) replaced by \( u \) this gives Algorithm 3.

\footnote{The index notation differs from the original specification; In here we use \( k = 0 \) for the most fine grained grid and use 0 \( \leq k < K \). In the specification \( 1 \leq k \leq K \) is used, where \( k = K \) represents the most fine grained grid.}
Algorithm 3 V-cycle multi grid solver (2)

\[ u_0 = 0 \]
\[ r_0 = v - Au \]
\[ \text{for } k = 0 \text{ to } K - 2 \text{ do} \]
\[ r_{k+1} = Pr_k \]
\[ \text{end for} \]
\[ u_{K-1} = 0 \]
\[ u_{K-1} = u_{K-1} + Sr_{K-1} \]
\[ \text{for } k = K - 2 \text{ to } 1 \text{ step } -1 \text{ do} \]
\[ u_k = 0 \]
\[ u_k = Qu_{k+1} \]
\[ r_k = r_k - Au_k \]
\[ u_k = u_k + Sr_k \]
\[ \text{end for} \]
\[ u_0 = Qu_1 \]
\[ r_0 = v - Au_0 \]
\[ u_0 = u_0 + Sr_0 \]
\[ r_0 = v - Au_0 \]

All matrices \((A, P, Q, \text{and } S)\) in the algorithm can be represented by parameters for fixed 3D grid molecules; i.e. each linear equation can be written as

\[
\sum_{i=-1}^{1} \sum_{j=-1}^{1} \sum_{k=-1}^{1} a_{i,j,k} u(l + ih_1, m + jh_2, n + kh_3) \quad (3.7)
\]

where \(a_{i,j,k}\) is independent of \((l, m, n)\). Therefore only the coefficients \(a_{i,j,k}\) \((2^7\) in total, but only 4 unique values, because values for \(a_{i,j,k}\) are the same for equal values of \(l = |i| + |j| + |k|\) and not the entire matrix has to be stored in memory and each matrix vector multiplication can be performed by evaluating (3.7). This also means that it is preferable to store the vectors \(u, v\), and \(r\) in the 3D index format corresponding with the grid (i.e. 3D arrays)\(^7\).

3.3.2 Class parameters

The parameters of each Class contains the size \(N\) of the grid dimension. The used grid is a square grid of size \(N \times N \times N\). Furthermore the number of times the V-cycle routine has to be performed is provided as a Max Iterations parameter. Also the smoothing matrix \(S\) differs per Class. For the two largest benchmarks (Class B and C) a different smoother is used then for the smaller ones (S, W, and A). As mentioned before, the matrices are stored as a series of coefficients, therefore \(S\) is given as an array of just four doubles. The last parameter is a verification value.

3.3.3 Initialization

The matrices \(A, P, \text{and } Q\) are constant throughout the entire benchmark and can be represented by only four doubles each (these values are provided in [3]). They are therefore also initialized as constant arrays.

Although the vector \(v\) has only 20 non-zero values, this vector is calculated each time at runtime by a random initialization routine. This routine starts by giving each element of \(v\) a (pseudo) random value in \([0,1)\). It then takes the indexes of the ten highest values and sets the values of \(v\) at these positions to 1. Corresponding it also takes the ten smallest values and initializes \(v\) to -1 at these positions. At all other positions \(v\) is set to 0.

\(^7\)Although in the code these vectors will be represented as 3D arrays, I will continue to refer to them as vectors.
3.3.4 Verification

For verification the $L_2$ norm of the residual $r_0$ is used. After the given number of V-cycle iterations
is performed this norm should be equal to the given verification value with an accuracy of $10^{-14}$.

3.3.5 Implementation

The main problem with parallelization of this algorithm is finding the right distribution for the
vectors $u_k$ and $r_k$. In the Fortran code each vector had all of its grid density variants consecutively
in one memory block ($u^K$, $u^{K-1}$, ..., $u^2$, $u^1$, with sizes $M^2$, $(\frac{M}{3})^3$, ..., 8, 1). Both because
Spar/Java cannot deal with such consecutive blocks (we can’t create a reference to a certain
sub block as is done within the Fortran implementation) and because mapping annotations would
become impossible, the blocks were stored separately and referenced to by an array (See Algorithm
4).

### Algorithm 4 MG.spar - Initialization of u and v

```
1   u = new double [MaxGridLevel][*,*,*];
2   r = new double [MaxGridLevel][*,*,*];
3   int m = N;
4   for (k := MaxGridLevel) {
5       u[k] = new double [m+2,m+2,m+2];
6       r[k] = new double [m+2,m+2,m+2];
7       m /= 2;
8   }
```

The +2 is used to create some ghost boundaries so we can also use (3.7) on the periodic boundaries.
Depending on the amount of processors available and the size of $N$ a [block,block,block],
[block,block,*], or [block,*,*] distribution can be used for each $u_k$ and $r_k$. These distributions
 correspond to blocks, bars, and slices. Which one is best depends on the amount of available
processors and the kind of architecture and should therefore be determined by measuring execution
times. Furthermore, because distribution of a variable is part of its type, $u[k]$ and $r[k]
need have the same distribution for every $k$. Otherwise we would have an array with different
element-types, which is not allowed.

Parallel execution of the operations performed on these vectors is in general rather straightforward.
Each matrix-vector multiplication or AXPY (matrix-vector multiplication plus addition of
a vector) is encapsulated in a function

- RestrictResidual : $r_{k+1} = Pr_k$
- Prolongate : $u_k = Qu_{k+1}$
- EvaluateResidual : $r_0 = v - Au_0$ and $r_k = r_k - Au_k$
- ApplySmother : $u_k = u_k + Sr_k$

Within these functions regular owner absorption can be used to determine where code should be
executed. The compiler should be able to recognize and use communication aggregation both for
the AXPY and the UpdateBorders routine (this routine is usually called at the end of an AXPY
to update the shadow borders of $u_k$ or $r_k$) to speed up performance. If we look at Algorithm 5
one very important aspect is the use of temporary variables $u_1$ and $u_2$. Such temporary variables
are also used in other routines and reside also in the Fortran code. These variables are introduced
to minimize the amount of operations (ops). As mentioned before, variables should be declared
as local as possible in order to eliminate any needless, or even erroneous, reduction. In this case,
however, the declaration of $u_1$ and $u_2$ within the $(i,j)$ loop is not desirable. Such a declaration
would cause the creation of $M \times M$ temporary arrays. Not only are new and delete calls
Algorithm 5 MG.spar - EvaluateResidual()

```java
static void EvaluateResidual(final double [*,*,*] r,
                              final double [*,*,*] v,
                              final double [*,*,*] u) {
    // r = v - A * u
    // (v and r are references to the same array in most cases!)
    final int M = u.getSize()[0];  // u, v and r are MxM*M, M=N+2
    final double [] u1 = new double [M];
    final double [] u2 = new double [M];

    foreach (i :- 1:M-1, j :- 1:M-1) {
        foreach (k :- :M) {
            u1[k] = u[i,j-1,k] + u[i,j+1,k] +
                    u[i-1,j,k] + u[i+1,j,k];
            u2[k] = u[i-1,j-1,k] + u[i-1,j+1,k] +
                    u[i+1,j-1,k] + u[i+1,j+1,k];
        }
    }
    foreach (k :- 1:M-1) {
        r[i,j,k] = v[i,j,k] - (A[0] * u[i,j,k] +
                              A[1] * (u[i,j,k-1] + u[i,j,k+1] + u1[k]) +
                              A[2] * (u2[k] + u1[k-1] + u1[k+1]) +
                              A[3] * (u2[k-1] + u2[k+1]));
        // A[1] is probably 0, but the compiler should
        // Otherwise just comment this line
    }

    // Drastic need for garbage collector
    delete(u1);
    delete(u2);
    UpdateBorders(r);
}```
expensive, but we would also be creating more temporary arrays then is necessary, because only one pair of \( u_1 \) and \( u_2 \) per processor is needed. The use of such temporary variables therefore asks for a kind of local annotation. Though not exactly, because the array itself may still be distributed. What we actually want is a kind of mapping annotation that lets us tell the compiler to replicate \( u_1 \) without update over the first two dimensions and to align \( u_1 \) with the third dimension of \( x, u, \) and \( v \) (who are all supposed to have the same distribution). For \( M = 10, 8 \) processors and a \([\text{block}, \text{block}, \text{block}]\) distribution for \( x, u, \) and \( v \) we need 4 instances of the temporary vectors \( u_1 \) and \( u_2 \). Figure 3.1 shows how each \( u_1 \) needs to be distributed relative to the distribution of \( u \). However, there is currently no way to annotate this kind of distribution. In Chapter 6 we will look at possible new annotations. Furthermore, if such an annotation would be available, we would still have the issue of task parallelism. The outer loop must be annotated in such a way that the loop gets distributed among four groups of two processors each. To do this we use the 3D processor-grid that is also needed to distribute the 3D arrays. Suppose we have a Gpp3D \([2, 2, 2]\) processor-grid of processor type Gpp, then we could annotate the outer foreach loop with a \(<$ on=Gpp3D[(\text{block } @i \ldots), (\text{block } @j \ldots), \text{all}] \>$\) and the inner foreach statements with a \(<$ on=Gpp3D[(\text{block } @i \ldots), (\text{block } @j \ldots), (\text{block } @k \ldots)] \>$\). Only, we do not want to have to calculate the blocksize by hand, therefore an altered version of the on annotation where one could use \((\text{block } @i)\) as parameter and have the compiler automatically calculate the blocksize by dividing the loop- or array-length by the total number of processors, would be very useful.

If we do not want to introduce any new annotations (besides the local annotation), solutions which would circumvent the problem are

- Replicate the third dimension of all 3D arrays. This means that each instance of \( u_1 \) only needs to be mapped on one processor and a simple local annotation can be used.
- Eliminate the use of temporary variables. Of course this solves the problem, but it will result in a decrease in sequential performance.

A routine that is much less easy to parallelize is the InitNonZeros function that sets the 1 and \(-1\) values of \( v \). Fortunately this routines falls out of the scope of the benchmark timings and therefore I have not tried to optimize this routine. When one wants to try to efficiently make this function run in parallel a rather similar approach to the EP benchmark can be taken.

One other calculation that can be done in parallel is the givenorm routine which calculates the \( L_2 \) norm and \( L_\infty \) norm of \( r_0 \). These are general \( \ast \) and \( \max \) type reductions and should be recognized automatically by the compiler.
Next to these parallel issues also some sequential issues require some further explanation. In `EvaluateResidual` the parameters are r, v, and u. But instead of calculating r = v − Au this function is also called to calculate r = r − Au, which means that r and v are references to the same 3D array. Although within the code of the algorithm this poses no real problem it is worth mentioning\(^8\).

Another small issue is the use of A[1] in `EvaluateResidual`. For all benchmark classes (S, W, A, B, and C) A[1] equals zero, which means that the multiplication with A[1] in the inner `foreach` loop can be eliminated. It was left in, both for generality and because in the Fortran code it wasn’t eliminated either. Furthermore, a good back end compiler should be able to eliminate multiplications with zero-valued `final` variables.

\(^8\)Especially for those readers who would like to make changes to the code.
3.4 Implementing CG - Conjugate Gradient Benchmark

3.4.1 What it does

There are several ways to solve the linear set of equations $Ax = b$. There are direct methods, like Gauss elimination, that result in a $A = LU$ factorization or, in the case of a symmetric matrix, in a $A = LDL^T$ (a Cholesky factorization). An other approach, however, is making use of iterative methods. Although they only approximate the solution, the calculation time is usually much shorter. The general approach of an iterative solver is to start with a first approximate of the solution $x_0$ and to use

$$
\begin{align*}
x_{m+1} &= x_m + Br_m, \quad m = 0, 1, \\
r_m &= b - Ax_m
\end{align*}
$$

or, written in another form

$$
\begin{align*}
x_{m+1} &= Ex_m + c, \quad m = 0, 1, \\
E &= (I - BA) \\
c &= Bb
\end{align*}
$$

(3.9)

(3.8)

to find a better solution $x_1$. And with each iteration a better solution $x_m$ should be attained. For symmetric positive definite matrices $A$ a special iterative solver, called the Conjugate Gradient method, is available. If the dimensions of the matrix are $N \times N$ then it can be proven with help of the matrix properties (symmetric and positive definite) that CG will converge within $N$ iterations (i.e. $r_N = 0$)\textsuperscript{9}. Instead of using (3.8), CG also looks at all previously calculated residuals ($r_0 \ldots r_{m-1}$). It will create a search direction $p_m$, a polynomial in $r_0 \ldots r_m$, with which we try to find $x_{m+1}$

$$
\begin{align*}
x_{m+1} &= x_m + \alpha_m p_m, \quad m = 0, 1, \\
p_m &= r_m + \beta_m p_{m-1}, \quad m \geq 1 \\
p_0 &= r_0
\end{align*}
$$

(3.10)

Algorithm 6 provides the general iteration scheme for CG.

**Algorithm 6 Conjugate Gradient**

$x_0 = \ldots$

$r_0 = b - Ax_0$

for $m = 0$ to $N - 1$ do

$\rho_m = r_m^T r_m$

if $m = 0$ then

$\beta_0 = 0$

else

$\beta_m = \frac{\lambda_m}{\rho_{m-1}}$

$p_m = r_m + \beta_m p_{m-1}$

end if

$\sigma_m = p_m^T Ap_m$

$\alpha_m = \frac{\rho_m}{\sigma_m}$

$x_{m+1} = x_m + \alpha_m p_m$

$r_{m+1} = r_m - \alpha_m Ap_m$

end for

Within the NPB CG benchmark the CG iterative solver is used to generate an estimate for the largest eigenvalue of the sparse symmetric positive definite matrix $A$. This algorithm is called the inverse power method and is given in pseudocode in Algorithm 7.

\textsuperscript{9} Although this doesn’t hold for non positive definite symmetric matrices, CG also proves to be a good solver for these types of matrices.
Algorithm 7 Inverse Power method for calculating largest eigenvalue

\[
x = [1, 1, \ldots, 1]^T;
\]

for \( i = 1 \) to \( M \) do

\[
z = A^{-1}x
\]

\[
\zeta = \lambda + \frac{1}{x^T z}
\]

\[
x = \frac{z}{\|z\|}
\]

end for

For calculation of \( z = A^{-1}x \) the exact CG implementation needs to be used as is stated in Algorithm 8. The difference between this algorithm and Algorithm 6 lies only in the naming of unknowns and a reordering of the calculations.

Algorithm 8 Prescribed CG method to calculate \( z \) from \( Az = x \)

\[
z = 0
\]

\[
r = x
\]

\[
\rho = r^T r
\]

\[
p = r
\]

for \( i = 1 \) to \( 25 \) do

\[
q = Ap
\]

\[
\alpha = \frac{p^T q}{\rho}
\]

\[
z = z + \alpha p
\]

\[
\rho_0 = \rho
\]

\[
r = r - \alpha q
\]

\[
\rho = r^T r
\]

\[
\beta = \frac{\rho}{\rho_0}
\]

\[
p = r + \beta p
\]

end for

\[
\|r\| = \|x - A z\|
\]

Although the residual \( \|r\| \) is not needed to calculate \( \zeta \) the benchmark specifications require the explicit calculation of this norm (\( \|r\| = \sqrt{r^T r} \)). Also, after each calculation of \( \zeta \) in the main iteration loop it is required to print the current values of \( i, \zeta \), and \( \|r\| \).

3.4.2 Class parameters

The parameters that are needed per Class are the dimensions of the matrix \( A \) (this is only one value, because \( A \) is square), the maximum amount of non-zeros of an initialization vector (the use is explained in the next section), the shift \( \lambda \), an \textit{round} parameter (also for initialization purposes), the amount of iterations for the main loop \( (it) \) and a verification value. The \textit{round} is an inverted condition number and appears to have the same value of 0.1 for all Classes. It is nevertheless considered a Class parameter.

3.4.3 Initialization

A phenomenon which now and then seems to arise is that part of the program code that is intended to support the primary code (in this case the CG routine) tends to grow larger and more complex than the primary code of a program. For the CG benchmark this is also the case. The major amount of code can be contributed to the initialization step, the creation of the sparse matrix \( A \). Because of the complex nature of this initialization scheme NAS has provided the functions to create this matrix in a Fortran include. Strangely enough, one of the requirements of the benchmark is that this code may not be changed. This is odd for two reasons:

First of all the initialization step is not included in the timings and any performance gain in this step can therefore not contribute to a better benchmark performance. Furthermore it is stated
that before the benchmark timing commences (and thus the main iteration loop \textit{it}) it is allowed to rearrange the data structures returned by \texttt{MakeA}. Nevertheless another requirement states that within the actual calculations only an explicit version of \( A \) may be used\(^\text{10}\). Therefore, as long as there is an explicit version of \( A \) that resembles exactly the data delivered by \texttt{MakeA} it does not really matter how this \( A \) was built up.

Another argument why it is odd that \texttt{MakeA} may not be changed relates to the size of \( A \). If one is not allowed to change this initialization code it has become impossible to give any distribution to the array structures returned by \texttt{MakeA}. For the smaller Classes this may not be a problem, but there are architectures that will not have the memory capacity to store the structures for Class C (and maybe even for Class B) within the memory space available for a single processor node. Thus giving implementors at least the possibility to insert distribution schemes in \texttt{MakeA} is rather a necessity.

I shall now try to give a brief description of the general sparse matrix scheme and the role of each of the contributing functions:

The main call is made to the function \texttt{MakeA}. It expects \texttt{rcond} and \texttt{shift} (\( \lambda \)) as input parameters and returns three arrays \( A \), \texttt{ARowOff}, and \texttt{AColIndex}\(^\text{11}\). These three arrays together form the sparse matrix representation of \( A \). The array \( A \) contains all non-zero elements of \( A \) in a row wise order. \texttt{ARowOff} contains a list of indexes for \( A \) that represents the position in \( A \) where the next row commences. So, for instance, \( A[\texttt{ARowOff}[0]] \) would give the first element of \( A \) in the first row and \( A[\texttt{ARowOff}[4]] \) would give the first element of \( A \) that is contained in the 5\textsuperscript{th} row. The array \texttt{AColIndex} subsequently gives the column number within \( A (0..N-1) \) of each of the corresponding elements in \( A \). So for each index \( k \) we have

\[
A_{ij} = A[k] \quad i = \arg\max_\ell \texttt{ARowOff}[\ell] \leq k \quad j = \texttt{AColIndex}[k] \tag{3.11}
\]

\texttt{MakeA} also uses a temporary representation of \( A \) through the arrays \texttt{AEIm}, \texttt{ARow}, and \texttt{ACol}. In this representational form \texttt{AEIm}[k] represents a contribution to \( A \) at position \( \texttt{ARow}[k], \texttt{ACol}[k] \). There can (and will) be more contributions to a single matrix element \( A_{ij} \), therefore in the end a summation has to occur (this is done in \texttt{SparseMat}). Also worth noticing is the change of meaning of \texttt{ARow} compared to \texttt{ARowOff}. Where \texttt{ARowOff} contains the first offset within \( A \) for each row number \( i \), \texttt{ARow} contains the row number \( i \) for each index \( k \). \texttt{ARowOff} can thus be considered as a kind of inverse for \texttt{ARow}.

\[
A_{ij} = \sum_k \{ \texttt{AEIm}[k] \mid \texttt{ARow}[k] = i \land \texttt{ACol}[k] = j \} \tag{3.12}
\]

Let \( A \) be a \( N \times N \) matrix\(^\text{12}\) then the filling of \texttt{AEIm}, \texttt{ARow}, and \texttt{ACol} is done with \( N \) sparse vectors of size \( N \). Each vector is calculated by a call to \texttt{SparseVec}\(^\text{13}\). In this function the vector is filled with an amount of elements in the range \([0, 1)\). The precise amount is determined by the ‘maximum amount of non-zeros’ Class parameter. In a loop, random numbers are drawn for the value and position of the vector elements until the right amount of non-zeros is reached\(^\text{14}\).

\(^{10}\)This is because it is possible to make use of the way \( A \) was build up. One can rewrite the matrix-vector product without having the actual matrix, but only the vectors it was build from.

\(^{11}\)Within the \texttt{Fortran} code these variables are called \texttt{a}, \texttt{colix}, and \texttt{rowstr}. But these names were in my opinion not very convenient and were therefore changed. The names as mentioned in the text are also the ones used in the \texttt{Spar} implementation.

\(^{12}\)\( N \) is denoted by \texttt{dimA} in the \texttt{Spar} code.

\(^{13}\)Also the names of the subfunctions were changed in the \texttt{Fortran} to \texttt{Spar} conversion. \texttt{spmvr} became \texttt{SparseVec}, \texttt{vecset} was turned into \texttt{SetVec} and \texttt{sparse} was renamed \texttt{SparseMat}.

\(^{14}\)Strange enough when the random generator returns a value of 0.0 this is also considered to be a non-zero fill for the vector.
Next, MakeA calls SetVec to set the $k^{th}$ element of vector $k$ to 0.5. If the vector did not already have a value at this position the amount of non-zeros is increased by 1 (thus one more than is prescribed by the Class parameter). This increase ensures the non-singularity of the matrix. Within MakeA the outer product of each vector is then added with a certain weight to the arrays ARow, ACol, and AEIm. And finally for each $A_{ij}$ an element is added to ensure a certain condition number. At the end of MakeA the local contribution arrays are summarized and converted into the arrays A, ARowOff, and AColIndex.

3.4.4 Verification

Verification is done by comparing the final value of $\zeta$ with the verification value. The absolute difference must be no more than $1.0 \cdot 10^{-10}$.

3.4.5 Implementation

As can be seen in the implementation, the majority of the code is spend on initializing the matrix $A$. Although these procedures do not take part in the benchmark timing they have been rewritten to a great extent to make them better readable and more compact. The calculations performed are nevertheless still identical.

The sections that do contribute to the timings are contained in the main iteration loop and the function ConGrad. The amount of parallelism in the main iteration loop is restricted to the calculation of 2 vector dot products, which are normal reductions, and a normalization of $x$ which can be done perfectly in parallel (as long as the arrays $x$ and $z$ are identically distributed). The most interesting part however is of course the ConGrad algorithm. In Spar the implementation was done with variables $x$ and $b$ so as not to get confused with the use of $x$’s in textbook examples (where one usually tries to solve $Ax = b$). In Algorithm 9 the pseudo code is restated with these new variables.

<table>
<thead>
<tr>
<th>Algorithm 9 Prescribed CG method for $b = x$ and $x = z$ (i.e. solve $Ax = b$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 0$</td>
</tr>
<tr>
<td>$r = b$</td>
</tr>
<tr>
<td>$p = r^T r$</td>
</tr>
<tr>
<td>$p = r$</td>
</tr>
<tr>
<td>for $i = 1$ to $25$ do</td>
</tr>
<tr>
<td>$q = Ap$</td>
</tr>
<tr>
<td>$\alpha = \frac{p}{r^T q}$</td>
</tr>
<tr>
<td>$x = x + \alpha p$</td>
</tr>
<tr>
<td>$r = r - \alpha q$</td>
</tr>
<tr>
<td>$\rho = r^T r$</td>
</tr>
<tr>
<td>$\beta = \frac{p}{p_0}$</td>
</tr>
<tr>
<td>$p = r + \beta p$</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>$</td>
</tr>
</tbody>
</table>

As can be seen the algorithm consists of 2 specific matrix vector multiplications ($q = Ap$ and $Ax$), 2 kind of dot products ($p^T q$ and $r^T r$) and some AXPYs and vector copies. A straightforward parallelization of this algorithm can be done by assigning a block distribution to all vectors $x$, $r$, $b$, $p$, and $q$. All AXPYs and vector copies can then be done perfectly in parallel and the dot products won’t pose a problem either.

Currently, the choice was not to give any special distribution to the matrix $A$ (i.e. it is replicated) because this gives the least amount of communication overhead. For the higher benchmark
Classes on some architectures it may nonetheless be necessary to come back on this and provide
a distribution for A. With the current implementation of the data structure for A (See the CG
implementation in the Appendix) it is unfortunately not possible within Spar to assign a decent
distribution. The preferred distribution for A, if all vectors are block distributed, would be to
align the columns of A with the vector it is multiplied by. However, since A is symmetric we could
also choose a row wise distribution and change the matrix vector multiplications into
\( q = A^T p \)
and \( b = A^T x \). Building forward on this transpose approach we could transform the data structure
A, ARowOff, and AColIndex into

- \( \text{double}[][\] A: An array of arrays. Each subarray can have a different length and contains
  the non-zero elements of a row, thus \( A[1][2] \) contains the third non-zero element of the
  second row of \( A \). The block distribution is done in the first dimension, so each processor
  gets appointed a group of subarrays.

- \( \text{double}[][\] AColIndex: Again an array of arrays and with the same distribution scheme
  as A.

Since both A and AColIndex now have 2 dimensional indexing the need for ARowOff has become
obsolete.

Some other considerations that are not of a parallel nature are the special use of tuples and
exceptions in this benchmark. In Java (and Spar) all parameters are passed by value and a
method can only return one return value. This means that if we want a method to return more
than one primitive type in Java we would have to encapsulate these variables in an object and
return its reference. With Spar we can now use tuples to do this as is can be seen in the definition
of MakeA

```
1       static public [type double [], type int [], type int []] MakeA(
2           final int dimA, final int MaxNonZerosA,
3           final int MaxNonZerosV, final double rcond,
4           final double shift) throws ToManyElementsException {
5           // Below we use MaxNonZeros V+1 because an extra diagonal
6           // element could be created in method SetVec
7           final double [] V = new double [MaxNonZerosV+1];
8           final int [] VIndex = new int [MaxNonZerosV+1];
9
10          // List of nonzero matrix elements (double entries allowed)
11           final double [] AElm = new double [MaxNonZerosA];
12           final int [] ARow = new int [MaxNonZerosA];
13           final int [] ACol = new int [MaxNonZerosA];
14
15           // Sparse representation of matrix A
16           final double [] A = new double [MaxNonZerosA];
17           final int [] ARowOff = new int [dimA+1];
18           final int [] AColIndex = new int [MaxNonZerosA];
19
20           int row, col;
21           double scale, size = 1.0, ratio;
22           int numNonZerosV, numNonZerosA = 0;
23
24           // Build symmetric sparse matrix out of outer products of
25           // sparse vectors
26
27           ratio = Math.pow(rcond,(1.0/dimA));
```

for (iteration := :dimA) {
    SparseVec(V, VIndex, dimA, MaxNonZerosV);
    Vec(V, VIndex, iteration, 0.5);
    numNonZerosV = V.getSize()[0];
    foreach (j := :numNonZerosV) {
        col = VIndex[j];
        scale = size * V[i];
        foreach (i := :numNonZerosV) {
            int k = numNonZerosA++;
            row = VIndex[i];
            if (k+1 >= MaxNonZerosA)
                throw new TooManyElementsException(iteration);
            ARow[k] = row;
            ACol[k] = col;
            AElm[k] = scale * V[i];
        }
        size *= ratio;
    }
    foreach (i := :dimA) {
        int k = numNonZerosA++;
        if (k+1 >= MaxNonZerosA)
            throw new TooManyElementsException(dimA+i);
        ARow[k] = i;
        ACol[k] = i;
        AElm[k] = rcond - shift;
    }
    AElm.setSize([numNonZerosA]);
    SparseMat(A, ARowOff, AColIndex, dimA, AElm, ACol, ARow);
    _delete(AElm);
    _delete(ARow);
    _delete(ACol);
    return [A, ARowOff, AColIndex];
}

this method returns three references to arrays. Although it is possible in Java to pass array
references to a method and allow the method to change the contents of the array, any change to
the reference (for instance assigning a different array to the reference) is not passed back to the
caller of the method. With this tuple construct it is now possible to let MakeA take care of the
complete initialization of the three arrays (including memory allocation).

Another added feature (in this case over Fortran) is the availability of Exceptions. Although in
other kernel benchmarks there is no need for Exceptions, this particular benchmark contains a
check in the initialization routines for A that can make perfect use of it. Because the amount
of non-zero elements in A depends on the amount of non-zeros in the randomly filled vectors (as
provided by SetVec), it might be possible that for some user defined benchmark Class this number
will exceed the amount of memory that was allocated for AElm\textsuperscript{15}. When this happens an exception

\textsuperscript{15} Although I have copied the check from the Fortran code it is very well possible that this check is superfluous
and that the amount of elements for AElm will never exceed MaxNonZerosA. However, the code makes a nice example
for the use of Exceptions, thus I have left it in.
is thrown and caught in the `main` method, where an error message is printed and the program is terminated.
3.5 Implementing FT - Fast Fourier Transform Benchmark

3.5.1 What it does

For some partial differential equations it is sometimes easier to find a solution in a transformed state. Common transformations in this regard are the Laplace and Fourier transform. For instance, let us consider the PDE

\[
\frac{\partial u(\vec{x}, t)}{\partial t} = \alpha \nabla^2 u(\vec{x}, t)
\]

(3.13)

where \( \vec{x} \) denotes a vector in 3D complex space (\( \mathcal{C}^3 \)). If we apply the Fourier transform to both sides we achieve

\[
\frac{\partial v(\vec{\xi}, t)}{\partial t} = -4\alpha \pi^2 |\vec{\xi}|^2 v(\vec{\xi}, t)
\]

(3.14)

with \( v(\vec{\xi}, t) \) being the Fourier transform of \( u(\vec{x}, t) \)

\[
v(\vec{\xi}, t) = \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} e^{-2\pi i (\vec{\xi} \cdot \vec{x})} u(\vec{x}, t) dx_1 dx_2 dx_3
\]

(3.15)

The general solution to (3.14) is

\[
v(\vec{\xi}, t) = e^{-4\alpha \pi^2 |\vec{\xi}|^2 t} v(\vec{\xi}, 0)
\]

(3.16)

So for any given starting value \( u(\vec{x}, 0) \) we can retrieve \( u(\vec{x}, t) \), for a certain value of \( t \), by taking the Fourier transform \( v(\vec{\xi}, 0) \), multiplying by \( e^{-4\alpha \pi^2 |\vec{\xi}|^2 t} \) and transforming the obtained value \( v(\vec{\xi}, t) \) back with an inverse Fourier transform.

For the FT benchmark this method is worked out with the use of the Fast Fourier Transform (FFT). The Fast Fourier Transform is an optimized algorithm for calculating the Discrete Fourier Transform (DFT). If we discretize \( u(\vec{x}, t) \) over \( \vec{x} \), resulting in a \( n_1 \times n_2 \times n_3 \) time dependent array, the forward DFT would be

\[
v_{j,k,l} = F_{q,r,s}(u(t)) = \sum_{i=0}^{n_3-1} \sum_{k=0}^{n_2-1} \sum_{j=0}^{n_1-1} u_{j,k,l}(t) e^{-2\pi ij q / n_1} e^{-2\pi ik r / n_2} e^{-2\pi i l s / n_3}
\]

(3.17)

And the inverse transform

\[
u_{j,k,l} = F_{q,r,s}^{-1}(v(t)) = \frac{1}{n_1 n_2 n_3} \sum_{i=0}^{n_3-1} \sum_{k=0}^{n_2-1} \sum_{j=0}^{n_1-1} v_{j,k,l}(t) e^{2\pi ij q / n_1} e^{2\pi ik r / n_2} e^{2\pi i l s / n_3}
\]

(3.18)

For an FFT implementation one usually expresses the 3D FFT in terms of collective 1D FFTs (See Algorithm 10).

**Algorithm 10** 3D FFT of \( u \) in terms of 1D FFTs

```
for i = 1 to n_1, j = 1 to n_2 do
    FFT1D(u_{i,j,*})
end for
for i = 1 to n_1, k = 1 to n_3 do
    FFT1D(u_{i,*k})
end for
for j = 1 to n_2, k = 1 to n_3 do
    FFT1D(u_{j,k,*})
end for
```
The benchmark description leaves the implementor free to choose any 1D FFT available. An odd thing however is that, since some FFT implementations leave the transformed vector in a permuted state, the benchmark rules do not require a reordering of this vector. It is even allowed to change the routines in the benchmark following the first 3D FFT (like the multiplication by exponentials and the verification routine) to work with these differently ordered vectors. This means that the butterfly ordering part of the FFT can be skipped completely and this could result in a substantial performance increase. In the Fortran implementation, however, an ordered version of the FFT, the Stockholm FFT (See Algorithm 11), is used\textsuperscript{16}. This same algorithm was also used for the Spar implementation.

\begin{algorithm}
\textbf{Algorithm 11} Stockholm FFT, for $N = 2^M$
\begin{algorithmic}
\Function{FFT}{direction, $N$, $x_0$}
\State $M = \log^2(N)$
\For{$t = 1$ to $M$}
\State $m = 2^t$
\If{direction = forward}
\State $\omega_m = e^{2\pi i/m}$
\Else
\State $\omega_m = e^{-2\pi i/m}$
\EndIf
\EndFor
\For{$k = 0$ to $\frac{N}{2} - 1$}
\For{$j = 0$ to $\frac{N}{2} - 1$ step $\frac{N}{2}$}
\State $x_l[k + 2j] = x_{l-1}[k + j] + \omega_m x_{l-1}[k + j + \frac{N}{2}]$
\State $x_l[k + 2j + \frac{N}{2}] = x_{l-1}[k + j] - \omega_m x_{l-1}[k + j + \frac{N}{2}]$
\EndFor
\State $\omega = \omega \cdot \omega_m$
\EndFor
\Return $x_M$
\EndFunction
\end{algorithmic}
\end{algorithm}

The full benchmark starts with a 3D array $u_{j,k,l}$ ($0 \leq j < n_1$, $0 \leq k < n_2$, $0 \leq l < n_3$) randomly filled with complex numbers. Then (3.17) is used to create the discrete Fourier transform $v_{j,k,l}$. Next, the discrete form of (3.16) is used to create the Fourier transform $w_{j,k,l}$ of the solution at time $t$.

$$w_{j,k,l} = e^{-4\pi^2 (j^2 + l^2 + F^2) t} v_{j,k,l}$$ \hspace{1cm} (3.19)

with\textsuperscript{17}

$$j = \begin{cases} j, & 0 \leq j < \frac{n_1}{2} \\ j - n_1, & \frac{n_1}{2} \leq j < n_1 \end{cases}$$ \hspace{1cm} (3.20)

Finally the inverse DFT (3.18) is applied to $w_{j,k,l}$ to achieve the solution $x_{j,k,l}$.

Because the FT benchmark would be too short with only one calculation of $x_{j,k,l}$, a part of the calculation is performed several times. Algorithm 12 states the pseudo code for the main loop. In this algorithm Evolve() performs the multiplication with the exponential factor. The time parameter $t$ used in the exponential is equal to the iteration step. After each iteration step a verification step is performed (this algorithm is explained further on).

\textsuperscript{16}Because the Fortran code uses a FFT variant that calculates the transform of multiple vectors in one routine (batch processing), it can at first be a little confusing to recognize the Stockholm FFT in this piece of code.

\textsuperscript{17}Similar definitions hold for $k$ and $l$. 
Algorithm 12 FT - Main loop
\[
\begin{align*}
\text{v} & = \text{FFT}(u) \\
\text{for } t = 1 \text{ to } N & \text{ do} \\
\text{w} & = \text{Evolve}(v,t) \\
x & = \text{FFT}^{-1}(w) \\
\text{Verify}(x) \\
\text{end for}
\end{align*}
\]

3.5.2 Class parameters

The Class parameters consists of the three separate dimension parameters \( n_1, n_2, \) and \( n_3, \) the number of iterations \( N \) and a list of \( N \) verify values.

3.5.3 Initialization

Initialization consists of three parts.
First an index map and list of exponentials is created (\texttt{ComputeIndexMap}). The list of exponentials is a 1D array
\[
ex[i] = (e^{-4\pi^2 i})
\] (3.21)
and the index map contains
\[
\text{indexmap}[j,k,l] = j^2 + k^2 + l^2
\] (3.22)

Thus if \texttt{Evolve} needs to multiply by \( e^{-4\pi^2 (j^2 + k^2 + l^2) t} \) this can now be done more computationally efficient by taking \( \text{ex}[t*\text{indexmap}[j,k,l]] \).

The next initialization step is the filling of \( u_{i,j,k} \). This is done by giving each element of \( u \) a random complex value \( c \) with \( 0 \leq \Re(c) < 1 \) and \( 0 \leq \Im(c) < 1 \) (See Algorithm 13).

Algorithm 13 Initialization of \( u_{i,j,k} \) at \( t = 0 \)
\[
\begin{align*}
\text{for } k = 1 \text{ to } n_3, j = 1 \text{ to } n_2, i = 1 \text{ to } n_1 & \text{ do} \\
\text{a} & = \text{Rand}() \\
\text{b} & = \text{Rand}() \\
\text{u}_{i,j,k} & = a + i b \\
\text{end for}
\end{align*}
\]
The Final initialization step creates a list of complex multipliers \( \omega_k \) for the 1D FFT routine

\[
\omega_k = e^{2\pi i k^2} = \cos\left(\frac{2\pi}{2k}\right) + i \sin\left(\frac{2\pi}{2k}\right)
\] (3.23)

3.5.4 Verification

After each iteration step a complex verification value is created from the calculated solution \( x_{j,k,l} \). At the end these \( N \) complex values are compared to the \( N \) values from the Class parameters and they must agree with an accuracy of \( 1.0 \cdot 10^{-12} \). Each verification value is calculated by taking the following sum
\[
\sum_{j=0}^{n_2} x_{q,r,s}, \quad \begin{cases} 
q = j \pmod{n_1} \\
 r = 3j \pmod{n_2} \\
s = 5j \pmod{n_3}
\end{cases}
\] (3.24)
CHAPTER 3. NAS PARALLEL BENCHMARKS

3.5.5 Implementation

The first step in parallelizing this benchmark is determining the granularity of parallelism. Two approaches can be taken. First, all we could use coarse grained parallelism, for which only the loops in Algorithm 10 are performed in parallel and each 1D FFT is calculated with a single processing unit. And on the other hand we have fine grained parallelism, where we also try to perform the calculations inside the 1D FFT in parallel. For the Spar implementation the coarse grained method was chosen. Not only is this much easier to annotate, but it is also justified because of the relative high loop count and almost identical calculation time for each of the 1D FFTs (i.e. perfect load balancing). Because parallelization of the 1D FFT also raises some interesting issues, in Chapter 4 we will also treat the parallelization of the 1D FFT.

Starting with the coarse grained approach, again two kinds of distributions are meaningful for the 3D matrix $u$. We could choose a [block,*,*] distribution, which means communication in Algorithm 10 only needs to take place for the last iteration loop ($j,k$ loop). On the other hand a [block,block,block] distribution is also possible. Although this kind of distribution requires communication for each of the three loops, per loop, the amount of processors needed to retrieve a full vector is minimal. For example with 8 processors and the first distribution it takes all 8 processors for the last loop to retrieve vector $u_{*,j,k}$ while with the same amount of processors and the triple block distribution it will only take 2 processors per vector $u_{*,j,k}$ (See Figure 3.2). Furthermore, with this second distribution, communication between one pair of processors will be completely independent with communication between another pair, that tries to exchange data for another vector. Just as with the distribution for the MG benchmark, performance analysis will be needed to determine the most efficient distribution for $u$.

Another routine that is unfortunately not that easy to parallelize is the initialization routine for $u$. Furthermore, this routine takes part in the benchmark timing and can therefore not be neglected. As mentioned with the previous benchmarks the random initialization routines can be quite hard to parallelize. Algorithm 14 shows the used Spar code.

Unlike with IS, where we chose our own batchsize and thus could determine ourselves how many instances of the Random class would be created, in this method Dim[2] automatically determines
Algorithm 14 FT - ComputeInitialConditions

```java
static void ComputeInitialConditions(final complex [*,*,*] u) {
    final double GenerationSeed = 314159265.0;
    final double GenerationMult = 1220703125.0;
    double Real, Imag;
    double M;

    foreach (k :- :Dim[2]) {
        Random random = new Random(GenerationSeed,GenerationMult);
        random.Skip(2*k*Dim[1]*Dim[0]);
        for (j :- :Dim[1], i :- :Dim[0]) {
            Real = random.RandLC();
            Imag = random.RandLC();
            u[i,j,k] = complex(Real,Imag);
        }
    }
    //delete(random);
}
```

the amount of instances of the Random class. A way to diminish the amount of instances for the Random class and the calls to Skip is by making use of batches. But, since Dim[2] is always the smallest of the 3 available dimensions and ranges from 32 (for Class S) to 512 (for Class C) it would only be interesting to make use of batches for the higher classes and then only when we have few processors available. So in this case the use of batches was omitted. Another possible solution would be to have some kind of annotation that tells the compiler to create only one instance of the Random class per processor. Although it might theoretically be possible to create such an annotation, all the calls to random.Skip would still be necessary, because the steps of a foreach may be executed in any order. Therefore such an annotation would not gain much performance in this case.

During this research project the Spar compiler was still under development and gradually became more able to handle certain parallel constructs. One of the first parallel benchmarks that was implemented was the NPB-FT. At that time some concessions had to be made to get the code running. One of the effects was the quite inelegant use of the two global temporary arrays temp1 and temp2. These two arrays were annotated with a local construct, so each processor had its own version. In the FFT 3D loop (See Algorithm 10) temp1 is filled with a copy of the 1D subarray of u. In the 1D FFT routine an FFT is then performed on this array (as can be seen in the FT implementation in the Appendix FFT3D does not pass an array parameter to FFT). In this same routine the second temporary array temp2 is used for the calculation of x0 (using the naming convention from Algorithm 11). For every step t, x_t is represented by temp2 and x_t-1 by temp1. At the end of each step the references to these arrays are swapped and temp1 then points to x_t and temp2 can be reused as x_t+1 in the next step. In the end temp1 thus contains (actually: is a reference to) the Fourier transformed vector.
3.6 Implementing IS - Integer Sort Benchmark

The Integer Sort benchmark is somewhat of an outcast within the NPB. First of all, although sorting is sometimes needed in large particle simulations, it is commonly not considered being a 'Computational Fluid Dynamics Algorithm’. Furthermore, the implementation as provided by NAS is done in C, while all other benchmarks are coded in Fortran. Therefore, in most other NPB implementations, IS is generally left out. On one hand this is a pity, since sorting algorithms are quite different from the benchmarks described in the previous sections and therefore make an interesting addition to a parallel benchmark set. On the other hand, the specific sorting algorithm that is used by IS, is a sort that can only be used under certain conditions; the elements must be integer values in a limited range. Because of these reasons I have also looked at a more general applicable sorting algorithm, which is described in Chapter 4.

3.6.1 What it does

The Integer Sort algorithm is used to sort N 'keys' in parallel by calculating the rank for each key. The keys are integer values and must be in the range \([0, B_{\text{max}}]\). Their initial values are randomly generated and have an approximate Normal distribution. If \(u_i\) are uniform random variables on \([0, 1)\) then each key is calculated by

\[
K_i = \lfloor B_{\text{max}} \frac{u_i u_{i+2} + u_{i+1} + u_{i+3}}{4} \rfloor
\]  

(3.25)

The distribution of \(K_i\) now resembles a clock curve like that of a Normal distribution, \(K_i\) has mean \(\frac{B_{\text{max}}}{2}\) and variance \(\frac{1}{3}(B_{\text{max}})^2\).

The keys \(K_i\) must initially, in case of a distributed memory system, be block distributed among all available processors. The way this distribution should be done is described in detail in [3].

One is free to choose the kind of sorting algorithm that is used to sort the keys. Furthermore the chosen sorting algorithm does not need to be stable, which means that the order of equal valued keys does not have to be preserved. The body of the program must however comply with Algorithm 15. This algorithm shows that instead of rearranging \(K_i\) it is required to build an array of ranks. Only at the end (after timing has stopped) a reordering of \(K_i\) takes place (but this is only necessary for the final verification).

<table>
<thead>
<tr>
<th>Algorithm 15 IS - Main code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate (K_i) and distribute among memory</td>
</tr>
<tr>
<td>(start timing)</td>
</tr>
<tr>
<td>for (i = 1) to (I_{\text{max}}) do</td>
</tr>
<tr>
<td>(K_i = i)</td>
</tr>
<tr>
<td>(K_{i+I_{\text{max}}} = B_{\text{max}} - i)</td>
</tr>
<tr>
<td>Find ranks (r_j) ((0 \leq j &lt; N)) such that (K_{r_j} \leq K_{r_{j+1}})</td>
</tr>
<tr>
<td>Perform partial verification</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>(end timing)</td>
</tr>
<tr>
<td>Reorder (K_i)</td>
</tr>
<tr>
<td>Perform final verification</td>
</tr>
</tbody>
</table>

An interesting aspect of this algorithm is the change of \(K_i\) and \(K_{i+I_{\text{max}}}\) at each iteration step. The values of these keys are altered to guarantee that the rank of a key will differ after each iteration step. The rank of a key will therefore be either increased or decreased by one or stay the same (except for \(K_i\) \((0 \leq i < 2I_{\text{max}})\) of course).
3.6.2 Class parameters

The parameters consists of the $\log^2$ of the total number of keys $N$, the $\log^2$ of the maximum key value $B_{\text{max}}$, the total amount of iterations $I_{\text{max}}$, and some verification variables consisting of 3 arrays and the size of these arrays.

3.6.3 Initialization

Initialization only contains the creation of $K_i$. The equation for this is given by (3.25). Each uniform random value is given by the pseudo random generator as used in the other benchmarks.

3.6.4 Verification

There are two verification steps. One partial verification test, which takes place after each iteration step, and one full verification test, that is performed at the end of the program.

The partial verification test takes $m$ keys and compares their ranks to the verification values. Because a key value might occur more than once and because the sorting algorithm does not have to be stable these verification keys are chosen such that their values are unique within the total set of values for $K_i$. Furthermore, because at each iteration step the list of keys is slightly changed, the verification keys are chosen in such a way that their rank will only increase by one or only decrease by one after each iteration step. The Class parameters for verification therefore consists of the total number of verification keys $m$, a list of key indexes $i_0,\ldots,i_{m-1}$, a list of $m$ rank values, and $m$ iteration aggregations (either $+1$ or $-1$).

The full verification test requires the filling of an array with the sorted keys (this array has the same distribution as $K_i$). This filling should be done according to the ranks $r_i$ that have been calculated. Verification is done by checking that the value of each key in the sorted array is greater than or equal to that of the previous key in the list.

3.6.5 Implementation

The sorting algorithm used in the C implementation by NAS is a bucket-counting-sort. The range $[0\ldots B_{\text{max}}]$ is divided among the available processors, where each subinterval for a processor has equal length. Then each processor gets all keys in his interval (bucket-sort) and performs the counting part. These partial summations are aggregated and the ranks of the keys are calculated. It is somewhat puzzling why NAS used a bucket-sort for this benchmark since the keys are not uniformly distributed on $[0\ldots B_{\text{max}})$ and this would therefore create a major load-imbalance. It would of course be possible to split the interval in pieces in such a way that the distribution of $K_i$ integrated over each piece would have equal values, but I consider it 'cheating' if one would make use of the distribution of $K_i$ beforehand. A counting sort, on the other hand, is a splendid sorting algorithm for this kind of problem, especially when $N > B_{\text{max}}$, which is true for all Classes. The bucket sort was therefore omitted, but the counting sort was kept and tried to be converted to a parallel version within Spar. In Algorithm 16 the ranking method is stated as implemented in Spar.

The keys $K_i$ are stored in the array Key[]. Unfortunately the block distribution that is prescribed for Key[] is not equal to the block distribution definition in Spar. NPB uses blocksize $N_p = \left\lceil \frac{N}{p} \right\rceil$, while Spar uses $N_p = \left\lceil \frac{N}{b^k} \right\rceil$. In this regard the implementation thus deviates from the specification. However for $p = 2^k (k \in \mathbb{N})$ the block distributions are equal and the rules are met.

Since $B_{\text{max}} < N$ the size of the array keyCount[] is also smaller then that of Key[]. For the benchmark Classes this is a factor of about 16 or 32. This is for most architectures of the same order as the amount of processors. Therefore it is justified to just replicate keyCount[]. Looking at the code this also has the benefit of reducing the complexity of the second foreach to a normal
Algorithm 16 IS - Rank

```java
static boolean Rank(final int iteration) {
    int passedVerificationCount = 0;

    // Set key counters to 0
    foreach (i := MaxKey+1) {
        keyCount[i] = 0;
    }

    // Count the number of occurrences of a value
    foreach (i := NumKeys) {
        keyCount[key[i]]++;
    }

    // Determine the amount less or equal to a value
    for (value := MaxKey) {
        keyCount[value+1] += keyCount[value];
    }

    // Determine the rank of each key
    foreach (i := NumKeys) {
        keyRank[i] = --keyCount[key[i]];
    }

    // Perform the partial verification test
    passedVerificationCount = TestArraySize;
    foreach (i := TestArraySize) {
        if (keyRank[testIndex[i]] != testRank[i] + testSign[i]*iteration) {
            System.out.println("Failed partial verification: ");
            System.out.println("iteration "+iteration + ", test key "+i);
            passedVerificationCount--;
        }
    }

    return (passedVerificationCount==TestArraySize);
}
```
reduction. If `keyCount[]` also would have been distributed it would have been very difficult to figure out which iteration step to run on which processor.

The final array in `Rank` that might have a distribution is `keyRank[]`. Because each key `K_i` has a corresponding rank `r_i` it is logical to choose identical distributions for `key[]` and `keyRank[]`. Therefore `keyRank[]` will also be block distributed.

If we look at parallelization of the several stages in `Rank()` the following can be said. The first loop could be executed in parallel were it not that `keyCount[]` is replicated, which means that letting each processor calculate the full loop might be much faster then having each processor only calculate a certain piece and communicating the rest. Experiments need to point out whether no distribution is to be preferred over a block task distribution.

The second loop, as already stated, now resembles a reduction, which is quite similar (although the array is larger) to the reduction of `Q` in EP.

The loop to calculate the accumulative totals is less well parallelizable. It can be performed by introducing batches as is shown in Algorithm 17 where `batchOffset` is cyclic distributed. This batch-approach introduces a perfectly parallelizable accumulation loop and a small accumulation reduction loop. The latter loop, however, is over few elements and should be pretty fast, especially when the compiler performs communication aggregation for this loop. The advantage of these extra loops is that the final `foreach` loop in Algorithm 17 can now be calculated fully in parallel.

**Algorithm 17 IS - Rank - Accumulate**

```java
int batchSize = 1 << 10;
int nBatches = MaxKey/batchSize;
int [] batchOffset = new int[batchSize];
batchOffset[0] = 0;
foreach (batch := 1:nBatches) {
batchOffset[batch] = 0;
for (value := (batch-1)*batchSize:batch*batchSize)
   batchOffset[batch] += keyCount[value];
}
for (batch := nBatches-1)
   batchOffset[batch+1] += batchOffset[batch];
foreach (batch := nBatches) {
   keyCount[batch*batchSize] += batchOffset[batch];
   for (value := batch*batchSize:(batch+1)*batchSize-1)
      keyCount[value+1] += keyCount[value];
}
```

The last loop of `Rank` before the partial verification poses the real problem. Here we bounce into the problem of having multiple keys with the same value. Suppose we have 4 keys with value 10 and their ranks should be 15...18. If this loop would be performed in a normal sequential order, the first occurrence of 10 in `key[]` would be assigned a rank of 18, the second key with value 10 would get 17, etc. To parallelize this loop we thus have to communicate the new value of `keyCount` to each other processor after each iteration step. Furthermore, two (or more) processors may not work at equal valued keys at the same time, otherwise keys might get equal ranks. The straightforward approach would be to have the compiler insert semaphores (this is something the compiler should be able to do automatically), but this would mean at least one communication step at each iteration for semaphore checking. There is however another solution, but this is much harder to implement in `Spar`, since we need to use the number of processors (i.e. a more explicit parallel programming approach). If we denote the key counters `keyCount[]` by `C_i` then
the pseudocode for this loop would be as shown in Algorithm 18. We let all processors cycle
over the key values. In the first cycle \( (c = 0) \) processor 0 processes the ranking of keys with
values \( 0, P, 2P, \ldots \), processor 1 processes \( 1, P + 1, 2P + 1, \ldots \), etc. In the second cycle processor
0 processes keys with values \( 1, P, 2P + 1, \ldots \), processor 1 also shifts one up and handles values
\( 2, P, 2P + 2, \ldots \), and processor \( P - 1 \) now takes the values for \( b = 0: K_i = 1, P, 2P, \ldots \). By doing
this, each processor now has to perform the inner loop \( P \) times instead of once, but the advantage
is that this inner loop can be performed perfectly in parallel. Communication for updating \( C \) can
be postponed until and aggregated at the end of a cycle \( (c \) loop), which means that we’ve gone
from \( N \) to \( P \) communication steps. However, a downside to this approach is that it uses, a not
yet supported, intrinsic \texttt{NumberOfProcessors()} of \texttt{Spar}, which is actually a more lower level way
to parallelize code and therefore does not really comply with the way \texttt{Spar} was intended.

\begin{algorithm}
\begin{algorithmic}
\STATE \textbf{P} = \texttt{NumberOfProcessors()}
\FOR {\textbf{c} = 0 \TO \textbf{P} - 1}
  \FOR {\textbf{p} = 0 \TO \textbf{P} - 1}
    \STATE \textbf{b} = (\textbf{p} + \textbf{c}) \mod \textbf{P}
    \FOR {\textbf{i} = 0 \TO \textbf{N} - 1}
      \IF {\textbf{b} = \textbf{K}_i \mod \textbf{P}}
        \STATE \textbf{r}_i = \textbf{C}_{\textbf{K}_i}
        \STATE \textbf{C}_{\textbf{K}_i} = \textbf{C}_{\textbf{K}_i} - 1
      \ENDIF
    \ENDFOR
  \ENDFOR
\ENDFOR
\end{algorithmic}
\end{algorithm}

The last parallelization issue takes place in the \texttt{FullVerify} procedure. Although this function is
executed after the benchmark timing has stopped and thus doesn’t have to be optimized, the loop
to do the final sorting is still quite interesting. In Algorithm 19 the final-sorting loop places all keys
in a sorted order in the \texttt{keySorted[]} array. This array has the same block distribution as \texttt{key[]}.
Because the rank value \texttt{keyRank[]} determines which processor needs to communicate with which
processor, the \texttt{foreach} loop will result in performing an unpredictable all-to-all communication.
It was not investigated how to provide a better way of parallelizing this loop, but these kinds of
loops might be interesting for further investigation.
Algorithm 19 IS - FullVerify

```java
static boolean FullVerify() {
    int incorrectKeys = 0;
    final int[] keySorted = new int[NumKeys];

    // Finally sort the keys
    foreach (i :- :NumKeys)
        keySorted[keyRank[i]] = key[i];

    // Confirm the correct sorting of keySorted
    // count incorrectly sorted keys, if any
    foreach (i :- :NumKeys-1)
        if (keySorted[i] > keySorted[i+1])
            incorrectKeys++;

    if (incorrectKeys != 0)
        System.out.println("Full_verify: number of keys out of sort: " +
            incorrectKeys);

    return (incorrectKeys == 0);
}
```
Chapter 4

Other Algorithms

Although the NPB kernels already capture a major section of possible types of algorithms that are used in the field of parallel programming, there are still some aspects that were not covered. In this chapter three more algorithms are described and the final section describes some general algorithmic issues that still need to be investigated.

4.1 One Dimensional FFT

As mentioned in Section 3.5 the 1D FFT routine was not parallelized because of the coarse-grained approach that was used. In this section we will describe in what way parallelization of this algorithm can be done. As a starting point we will use Algorithm 11. This algorithm contains three loops which individually can or cannot be turned into for each’s. Because each $x_t$ is calculated from $x_{t-1}$ the outer loop ($t$) will remain sequential. The two inner loops, however, have some parallelism in them. Because each element of $x_t$ gets only assigned once, all loop steps $(k,j)$ are independent for $x$. The problem, in this case, is the use of $\omega$. Because of the repetitive multiplication by $\omega$ the loop steps need to be performed in a fixed order, which means that the inner loops are not independent for $\omega$. One way to eliminate this dependency (only partial) is by recognizing that the $j$ and $k$ loop can be switched. If we do this the $j$ loop can now be performed in parallel.

begin for all $j = 0$ to $\frac{N}{2} - 1$ step $\frac{N}{2}$ do
  $\omega = 1$
  for $k = 0$ to $\frac{m}{2} - 1$ do
    $x_t[k + 2j] = x_{t-1}[k + j] + \omega x_{t-1}[k + j + \frac{N}{2}]$
    $x_t[k + 2j + m] = x_{t-1}[k + j] - \omega x_{t-1}[k + j + \frac{N}{2}]$
    $\omega = \omega \cdot \omega_m$
  end for
end for

Another approach is by storing every possible needed value of $\omega$. This would eliminate the $\omega$ products and would allow both the $j$ and $k$ loop to be performed in parallel. However, this approach was not worked out.

Having attained the task parallelism, a suitable distribution for $x_t$ has to be chosen. The best distribution, for an amount of processors that is a power of two, would be to have a block-cyclic distribution with blocksize $m$ for each $x_t^1$. With this distribution scheme each assignment in the $k$ loop as described above will only need data that is stored locally. Or, when $t$ reaches $M$, all data of one processor needs to be communicated to one other processor (See Figure 4.1 for the data dependency between $x_t$ and $x_{t-1}$). This means that $x_0$ will have a cyclic distribution

$^1$Note that $m$ depends on $t$ and that the distributions for $x_t$ therefore differ for different values of $t$. 

45
If we write a program for this like in Algorithm 20, a problem arises for the distribution of \(x_t\). For data efficiency only two arrays (\(x\) and \(y\)) are used, who swap their reference values after each loop step \(t\). Spar, however does not provide a \textit{redistribute} feature, therefore we are forced each time to create a new variable (\(y\)) with a new distribution. Unfortunately, this code will still not work properly. The problem is in the swap of \(x\) and \(y\). What we want to happen here for the swap, is that \(x\) would gain the data and distribution of \(y\) (and vice versa). But since Spar couples distributions to types, this would mean a retyping of \(x\) which cannot be done. To solve this problem one could create an array of \(M\) arrays with each its own distribution, but this would require a lot of memory space. Or we could choose 1 distribution for every \(x_t\) (for instance a block distribution), although this would provide a working program for Spar, it would result into a lot more communication. However, a solution which would neither increase memory usage nor communication overhead needs still to be found.
Algorithm 20 1D FFT

```c
void FFT_Sub(int is) {
    // is = forward(1)/inverse(-1)
    int pt, pM;
    int id_to, id_from;
    complex om, om1;

    for (t := 1:M+1) {
        pM = 1 ⇐ (M−t);
        pt = 1 ⇐ (t−1);

        // <$ on p cyclic(blocksize=2*pt) $>
        complex [] y = new complex [N];

        if (is==−1)
            om1 = Complex.conj(omega[t]);
        else
            om1 = omega[t];

        foreach (j := N/2:pt) { // <$ on p cyclic(blocksize=pt) $>
            om = 1.0;
            for (k := :pt) {
                id_from = j+k;
                id_to = id_from+j;
                complex x2 = om * x[id_from+N/2];
                y[id_to] = x[id_from] + x2;
                y[pt+id_to] = x[id_from] − x2;
                om = om * om1;

            }
        }
        [x,y] = [y,x];

        --delete(y);
    }

    if (is==−1)
        foreach (i := :N)
            x[i] /= N;
}
```
4.2 Merge Sort

For parallelization the Merge Sort is a very good general sorting algorithm. It doesn’t suffer from load balancing problems like with Quick Sort, it doesn’t require explicit parallelism (batch division) as with Bucket Sort and it can sort any type of elements, unlike the Count Sort used in the IS kernel benchmark. Furthermore it also provides insight in the application of recursive algorithms in Spar (a type of algorithm we have not yet encountered).

The binary recursive version of a Merge Sort is given in pseudo code in Algorithm 21.

Algorithm 21 Binary Recursive Merge Sort

function MergeSort($x, N$)
    for $i = 0$ to $N - 1$
        $temp[i] = x[i]$
        MergeSortSub($temp, x[0], N-1$)
    end for

function MergeSortSub($x_{from}, x_{to}, i_{first}, i_{last}$)
    $i_{middle} = \frac{i_{first} + i_{last} + 1}{2}$
    if $i_{first} \neq i_{middle} - 1$
        MergeSortSub($x_{from}, x_{to}, i_{first}, i_{middle} - 1$)
    end if
    if $i_{middle} \neq i_{last}$
        MergeSortSub($x_{from}, x_{to}, i_{middle}, i_{last}$)
    end if
    $i_1 = i_{first}$
    $i_2 = i_{middle}$
    $k = i_{first}$
    while $i_1 < i_{middle}$ and $i_2 \leq i_{last}$
        if $x_{from}[i_1] < x_{from}[i_2]$
            $x_{to}[k] = x_{from}[i_1]$
            $i_1 = i_1 + 1, k = k + 1$
        else
            $x_{to}[k] = x_{from}[i_2]$
            $i_2 = i_2 + 1, k = k + 1$
        end if
    end while
    while $i_1 < i_{middle}$
        $x_{to}[k] = x_{from}[i_1]$
        $i_1 = i_1 + 1, k = k + 1$
    end while
    while $i_2 \leq i_{last}$
        $x_{to}[k] = x_{from}[i_2]$
        $i_2 = i_2 + 1, k = k + 1$
    end while

What is interesting about this algorithm is that it has a lot of resemblance with the 1D FFT of the previous section. This becomes clear if we rewrite Algorithm 21 into a non-recursive form. This code can be seen, written in Spar notation, in Algorithm 22. Instead of the depth-first way of sorting, in the iterative algorithm, sorting is done in a breath-first fashion. The outer loop ($t$) in Algorithm 22 represents the recursive depth for Algorithm 21 and the second loop ($j$) sums up all possible instances of the recursive function at that depth. In the FFT code these two $t$ and $j$ loops are also apparent. This is actually not a big surprise, since the FFT stems from a recursive algorithm.
Algorithm 22 Iterative Merge Sort

```java
static public void Sort(int[] List) {
    int[] Buff;
    int N = List.getSize()[0];
    int M = ILog2(N-1)+1;

    Buff = new int[N];

    for (t := 1:M+1) {
        int pM = ((N-1) >> t) + 1;
        int pt = 1 << (t-1);

        foreach (j := pM) {
            int i1 = j*2*pt;
            int i2 = i1 + pt;
            int t1 = i2;
            int t2 = i2 + pt;
            if (N < i1 + 2*pt) {
                t2 = N;
                if (t1>N) t1 = N;
            }
        }

        int k = i1;
        while (i1 < t1 & & i2 < t2)
            if (List[i1]<List[i2])
                Buff[k++] = List[i1++];
            else
                Buff[k++] = List[i2++];

        while (i1 < t1)
            Buff[k++] = List[i1++];

        while (i2 < t2)
            Buff[k++] = List[i2++];

        [List,Buff] = [Buff,List];
    }

    if (M%2==1) { // M is odd
        [List,Buff] = [Buff,List];
        foreach(i := :N)
            List[i] = Buff[i];
    }
    _delete(Buff); // Until garbagecolletion is available
}
```
Just as the FFT code, the code of the iterative Merge Sort in Algorithm 22 also makes use of two arrays which are swapped at the end of the $t$ loop. We already discussed the problems that occur when the distributions of the arrays differ per value of $t$. However, the distribution scheme for the Merge Sort is not the same, because the Merge Sort does not have a butterfly type of data dependency, but a strict nested dependency as is shown in Figure 4.2. Both a block distribution for $Buff$ and $x$ might be a good choice here, since this kind of distribution not only fits the dataflow but also solves the swap problem. However, with a block distribution we will have to restrict ourselves to processor amounts of a power of 2.

We can conclude that it is possible to implement a parallel Merge Sort in Spar by writing the algorithm in an iterative way and using a more or less similar approach as with FFT. However, by writing the algorithm in an iterative form we have inserted some extra restrictions on independent calculation of sub-blocks. In the recursive form the task parallel construct would be

\begin{verbatim}
    each
      if i_first \neq i_middle - 1 then
        MergeSortSub(x_{1..n}, i_first, i_middle - 1)
      end if
      if i_middle \neq i_last then
        MergeSortSub(x_{1..n}, i_middle, i_last)
      end if
    end each
\end{verbatim}

which means that processors assigned to the first recursive call may perform all calculations in this recursive call and all subsequent recursive calls independent from the calculations by the processors that are assigned to the second function call. In the iterative form a barrier is placed after each recursive depth, which means that some parallel freedom is lost. Figure 4.3 shows the parallel partitioning (represented by blocks) for both the iterative and recursive form for the first four depths. Suppose every block in this figure has an execution time of $T_{d,i}$, with $d = 0,1,2,\ldots,D$ being the recursive depth and $i = 0,1,2,\ldots$ being the index of the block at that depth. The total execution time for the iterative method can then be described by

$$T_d = \sum_{d=0}^{D} \max_i T_{d,i}$$  \hspace{1cm} (4.1)
but for the recursive method this is

\[ T_{rec} = \max_i \sum_{d=0}^{D} T_{d,k_i} \quad \left( k_i = \left\lfloor \frac{i}{2^d - 1} \right\rfloor \right) \]  \hspace{1cm} (4.2)

\( T_{rec} \) can be regarded as a longest path in the recursive tree. Because \( T_{rec} \leq T_d \) we could loose performance when using the iterative algorithm. Unfortunately it is not possible within Spar to express the wanted parallelism in the recursive algorithm.

First of all the \texttt{on} annotation does not allow a mapping on multiple processors, which means that we can not split the set of processors in half and assign one half to the first recursive call and the other to the second. This problem is further described in Chapter 6.

Another problem is that Spar can not deal with \textit{Active Processor Sets}. Suppose we would have an \texttt{on} annotation that could map onto multiple processors, than each time we enter the recursive function we would have a different set of processors to divide among the two recursive calls. For example, with 8 processors we would first have to divide the set in \( \{0 \ldots 3\} \) and \( \{4 \ldots 7\} \). One depth further we would have either the first group which we have to split in 0, 1 and 2, 3 or the second group which needs to be split into 4, 5 and 6, 7. Since the \texttt{on} annotation requires mapping onto exact processor numbers we would have to have information on which processors are currently active. This problem can either be solved by providing features for this in the annotation domain (extra annotations or a change in the \texttt{on} annotation) or we would require the programmer to do the calculations himself with help of the \texttt{NumberOfProcessors} intrinsic. Suppose Spar provides range mappings, then the last option can be illustrated by the piece of code in Algorithm 23.

The index-positions \texttt{first}, \texttt{middle}, and \texttt{last} describe what part of the data (with size \( N \)) is worked on. This information is then used to make an equal partition for the total processors set. Note that the \texttt{NumberOfProcessors} intrinsic returns the \textit{total} number of processors and not the size of the Active Processor Set. Also note that once the Active Processor Set has diminished to only one processor, both recursive calls are mapped onto this processor, which means that the calls will not be performed in parallel.
Algorithm 23 Annotated recursive calls
1 int proc1first = (first * NumberOfProcessors())/N;
2 int proc1last;
3 int proc2first = (middle * NumberOfProcessors())/N;
4 int proc2last = (last * NumberOfProcessors())/N;
5 if (proc2first==proc1first)
6     proc1last = proc1first;
7 else
8     proc1last = proc1first-1;
9 each {
10     RecursiveProc(...) <$ on = Gpp[@proc1first..@proc1last] $>
11     RecursiveProc(...) <$ on = Gpp[@proc2first..@proc2last] $>
12 }

4.3 Poisson Solver

For benchmarking the sequential performance of the Spar compiler a stripped down version of MG, a simple 2D Poisson solver, has been used. The algorithm solves $A\phi = 0$, by using the Jacobi solver with red-black relaxation. On the borders ($x = 0, x = 1, y = 0, y = 1$) we have Dirichlet conditions: $\phi = 0$. Furthermore 3 sources are inserted at positions $(0.35,0.70), (0.625,0.75)$, and $(0.375,0.25)$. A source is represented as a fixed value for $\phi_{i,j}$ at the nearest discrete position. The boundary conditions are represented as zeros in a ghost boundary for $\phi$: $\phi_{0,*}, \phi_{N,*}, \phi_{*,0}, \text{and } \phi_{*,N}$ are all 0. Algorithm 24 states the pseudo-code for this solver.

Parallelization of this algorithm is usually done by defining a striped ([block,*]) or a blocked ([block,block]) distribution. In Spar this poses no problem and the communication needed within the parallel loops is handled by the compiler. Furthermore, the reduction on $\varepsilon$ should also be recognized by the compiler, because a Math.max in Spar is recognized as a reduction operator.
Algorithm 24 Jacobi iterative solver with red-black relaxation

for $i = 0$ to $N - 1$, $j = 0$ to $N - 1$ do
  $\phi_{i,j} = 0$
end for
SetSources($\phi$)
repeat
  $\varepsilon = 0$
  $k = k + 1$
for all $i = 1$ to $N - 2$ do
  for all $j = 1 + ((i + 1) \mod 2)$ to $N - 1$ step 2 do
    if NoSourceAt($i,j$) then
      $\psi = \phi_{i,j}$
      $\phi_{i,j} = \frac{\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1}}{4}$
      $\varepsilon = \max(\varepsilon, |\psi - \phi_{i,j}|)$
    end if
  end for
end for
for all $i = 1$ to $N - 2$ do
  for all $j = 1 + (i \mod 2)$ to $N - 1$ step 2 do
    if NoSourceAt($i,j$) then
      $\psi = \phi_{i,j}$
      $\phi_{i,j} = \frac{\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1}}{4}$
      $\varepsilon = \max(\varepsilon, |\psi - \phi_{i,j}|)$
    end if
  end for
end for
until $\varepsilon \leq \varepsilon_{\text{max}}$
4.4 Miscellaneous Algorithms and Programming Aspects

In this section some types of algorithms are described that introduce not yet mentioned parallel issues. Also implications for miscellaneous aspects like the usage of file I/O, etc. are discussed.

4.4.1 Search Routines

With Search Routines we mean the class of algorithms that try to find some data in a large dataset. If this dataset is very large and distributed among a set of processors we might want to perform a search for some data in parallel. The straightforward approach would be to have every processor search its own dataset for the required information and aggregate all findings after each processor has finished its search. For some applications it is however sufficient to find only the first occurrence that matches the query and than terminate the search-routine. For such applications it would be inefficient to have other processors continue searching their dataset, while one processor has already found a match. If we represent the dataset by a set of entries \( x_i \), then a general sequential search routine would be like the one in Algorithm 25.

\[\text{Algorithm 25 Sequential search routine}\]

\[
\begin{align*}
  a &= \text{somevalue} \\
  \text{index} &= -1 \\
  \text{for } i &= 1 \text{ to } N \text{ do} \\
  \quad \text{if } x_i &= a \text{ then} \\
  \quad \quad \text{index} &= i \\
  \quad \quad i &= N \\
  \quad \text{end if} \\
  \text{end for}
\end{align*}
\]

If we want to parallelize this loop (i.e. change the \textbf{for} into a \textbf{for all}) we will have a problem with the statement ‘\( i = N \)’, since only the last processor will still have an upper bound of \( N \). All other processors will have an upper bound half way in the loop. Java on the other hand supports the \texttt{break} statement for a \texttt{for} loop. In Spar we would thus like to say something like in Algorithm 26. However, Spar currently does not allow a \texttt{break} statement in a \texttt{foreach}. A possible solution would be an implementation as in Algorithm 27. Whenever the value for \texttt{index} changes this value is communicated to all other processors (\texttt{index} is replicated) and processors will terminate the inner loop. However, this solution requires explicit parallelism and is therefore not a preferable method. An introduction of a \texttt{break} construct for a \texttt{foreach} is therefore recommended.

\[\text{Algorithm 26 Parallel search routine (1)}\]

\[
\begin{align*}
  \text{index} &= -1; \\
  \text{foreach (i :- :N) <\$ block distributed \$>} \{ \\
  \quad \text{if (x[i]=a)} \{ \\
  \quad \quad \text{index} &= i; \\
  \quad \quad \text{break}; \\
  \quad \} \\
  \}
\end{align*}
\]

4.4.2 Pipelining

Because of the Series Parallel approach of Spar it is currently impossible to express pipelining algorithms. Especially in the field of image processing, which is the primary application for the Trimedia processor (one of the targets of Spar), pipelining algorithms can be found. A data dependency graph of a pipelining algorithm can be seen in Figure 4.4. Because this dependency
Algorithm 27 Parallel search routine (2)

1     index = -1;
2  foreach (p : = :P) <$ on p $> {
3       for (i : = p/N:(p+1)/N) {
4         if (x[i]== a)
5           index = i;
6         if (index!=-1)
7           break;
8       }
9  }

Figure 4.4: Data Dependency Graph of a Pipelining Algorithm

geraph is not in a Series Parallel (SP) form, it is not expressible with foreach and each statements within Spar. If we want to implement such an algorithm in Spar we would have to insert extra dependencies to transform the model into a NSP form. In Figure 4.4 this can be done by inserting synchronization points at each horizontal level (between \{A_1\} and \{A_2, B_1\}, between \{A_2, B_1\} and \{A_3, B_2, C_1\}, etc.). However, this diminishes the degree of parallelism and thus could result in a performance penalty.

4.4.3 Particle Simulations

The algorithms thusfar described always worked with arrays of fixed size with fixed distributions. For particle simulations this is not always the case. If the partitioning is done by subdividing the domain, then one processor can get more particles assigned to him than another processor. After each iteration step particles have moved from one subdomain to another, which means that the distribution among processors changes. It is questionable whether such dynamic mappings of elements can be expressed with the parallel constructs of Spar. Further investigation needs to be done in order to find what kind of particle simulation algorithms (or similar types of algorithms) can be efficiently implemented in Spar.
4.4.4 Domain Partitioning

For many large scale Computational Fluid Dynamics problems the partitioning of the data for the different processors is done in a separate program. The main reason for this is that efficient domain partitioning of real-life data (like landscapes or aircraft models) can be a time consuming task. Because domain partitioning can be completely separated from the numerical solver, a partitioning thus has to be made only once and can than be used many times for different runs of the solver. Because calculating a domain partitioning in advance is an explicit form of parallelism, in many cases this will also require explicit parallelism for the solver. The implementation of such a solver in Spar is possible with the help of a NumberOfProcesses intrinsic and the use of foreach to loop over all domains (usually one domain per processor) in parallel. A non-explicit form can be used when the domain is partitioned in sections which each require the same amount of data (with the exception of the last processors). In that case the domain data can be stored in a contiguous array with a block partition.

4.4.5 File I/O

An issue that should not be overlooked is communication of the processors with internal and external devices. For instance, a parallel architecture might be equipped with a single shared harddisk or may have a separate harddisk per processor. Both the options of having a program with one file for all processors, one file per processor on a single harddisk and one file per processor on a separate harddisk need to be possible within Spar (which is not yet the case). The most straightforward definition would be, that when a new file stream is declared, the first processor of the Active Processor Set handles this file stream and when the file is opened/created this processor will do this on his own filesystem. Every read or write is done only by this I/O processor and the data that has to be written or has been read is communicated to the other processors in the APS. However, such a definition poses a problem when the Active Processor Set is split and a processor that is not in the same APS anymore with the I/O processor tries to do some file access. When this happens inter processor-set communication will be needed to transfer the data to and from the file to this processor. But such communication is not in compliance with the Series Parallel design of Spar. Whether this is actually a real problem still needs to be answered.

4.4.6 Console I/O

A similar problem to File I/O is communication between the user and an interactive Spar program. In general, High Performance Computing applications are non-interactive or, if they are interactive, only perform I/O via one channel. For these applications input is usually handled by one processor and the data than communicated to all other processors (because this is on a global level we don't have the APS problem as mentioned for file I/O). Output is also not a problem, because running the program on a single processor should result in the same behavior as running the program on multiple processors. This means that if each processor prints its own piece of data in the multiple processor run, the single processor in the sequential run will print P lines of data. Running a program in parallel should not result in more or less output. The only exception is when an explicit parallel construction like in Algorithm 28 is used.

Algorithm 28 Output dependent on amount of processors

1. int P = NumberOfProcesses();
2. foreach (p := :P)
3. System.out.println("Hello World! I am No. "+p);

2. An extra feature might also be that a local annotated declaration of a file stream will result in each processor creating a separate file on its own filesystem.
Things become a lot more complicated however, when Spar should also support applications that have a separate user interface per processor. If such an application is needed, further research in this field will be required.

4.4.7 Farmer-Worker model

A paradigm that can often be found in parallel algorithms is the Farmer-Worker or Master-Slave paradigm. This paradigm is especially useful when load balancing becomes an issue. For instance we might have a set of batches with a lot of variance in the time to process each batch. If we would impose an initial distribution of these batches among the available processors, some processors may get finished while others may still have one or more batches in their queue (for a well load balanced problem this situation may also arise; for instance when we run the program on a heterogeneous system, where one processor is faster then another). The Farmer-Worker model tries to solve this load imbalance by assigning one processor as ‘Master’ or ‘Farmer’ and letting this processor take care of the scheduling of the batches on the rest of the processors, which are called the ‘Slaves’ or ‘Workers’. Each Worker signals the Farmer when he is ready for a new batch. The Farmer keeps on checking for these ‘jobless’ Workers and assigns a batch to them. This continues until there are no more batches and the program is terminated.

<table>
<thead>
<tr>
<th>Algorithm 29 Farmer-Worker distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  foreach (i :- nBatches) &lt;$&gt; on = P[i] $&gt; {</td>
</tr>
<tr>
<td>2      ProcessBatch(batch[i]);</td>
</tr>
<tr>
<td>3  }</td>
</tr>
</tbody>
</table>

In Spar there are currently constructs available that make a Farmer-Worker model possible. For instance the foreach in Algorithm 29 has a ‘_’ parameter for the on annotation. This parameter as described in Chapter 2 leaves the compiler free to choose its own distribution. This also means that the compiler may insert a run-time scheduler at this point. However, it depends on the run-time schedulers available what kind of scheduling approach can and will be implemented. In general, the scheduling system might thus not only be limited to a Farmer-Worker approach, where one processor is a dedicated scheduler and the other processors only execute the Spar program. An approach where each processor does some scheduling and all processors execute the Spar program (this is like each processor being both a Worker and a Farmer) is thus also possible.
Chapter 5

Performance Results

Although the Spar compiler was still in an early development stage it was already possible to run some benchmarks. The compiler was already capable of creating executables that could be run in single-processor mode (i.e. create executables like a normal sequential compiler). The parallel engines on the other hand were still in a very premature stage. Three benchmarks, FT, EP and the Poisson algorithm (described in Section 4.3), were extended with annotations to make it run with the current parallel engines. As mentioned in Section 3.5 describing the FT benchmark, some parts of the code had to be altered to have the parallel engines recognize and utilize the requested parallelism. Also, the annotation definitions as described in [16] were not yet implemented and an old set of annotation constructs had to be used.

The benchmarks, except for the Poisson algorithm which was not timed, were run on one cluster of the DAS\footnote{See \url{http://www.ascl.tudelft.nl/das/das.shtml}.} distributed supercomputer. Each node contains a 200 Mhz Pentium Pro with 64 MB RAM, 2.5 GByte local harddisk, and a Myrinet interface for intercommunication. The operating system on these nodes was Linux. Both the Fortran 77 with MPI and the HPF implementations from NAS as the Spar implementations were run on this cluster. On the following pages the graphs and timing results are shown for each of the NPB kernel benchmarks. For the Spar implementations the timings were only done for up to 24 processors. Furthermore, since some benchmarks implementations in Fortran 77 require a power of 2 for the amount of processors the timing results contains gaps. Some fields for Class A are also blank, because some of the benchmarks required to much resources (either or both time and memory) for a few amount of processors.

In Table 5.1 the used compilers and compiler options are shown for each of the set of implementations. The mpicc compiler was used to compile the MPI implementation of IS, because this benchmark was written in C. The pgf90 compiler was used to compile FT, because this benchmarks implementation used some Fortran 90 constructs and couldn’t be compiled with the Fortran 77 compiler.

All benchmarks were performed with the Panda 4.0 communication library (See [13]).

As can be seen from the results, EP in Spar scales similarly to the Fortran 77 implementations but with about 25% slower performance. The preliminary results for FT shows some curious degradation in performance when scaling from 2 to 3 processors. The reason for this is not yet clear and while the HPF code comes close to the Fortran performance, the Spar code thus keeps behind a factor 2 for larger numbers of processors.

Although the other NPB benchmarks do not yet have a parallel Spar implementation the timing results of Fortran and HPF have been included if available. An interesting aspect for instance is the bad scalability of the MG benchmark in HPF. This may hint to a parallelization problem that could also be apparent in Spar and thus may require further investigation.
Because sequential performance of Spar gives a good indication of the minimal performance difference with other compilers and because Spar, having several benefits over Java, could also be used to create sequential programs, the FT benchmark was also implemented in C and ‘pure’ Java. Table 5.2 gives the execution time of the total FT benchmark program (thus initialization and finalization are also included in the timings) for the different compilers and languages on a Pentium II at 400MHz with 192MB of main memory and Windows NT. The Spar implementation uses the complex type (implemented by the compiler through the complex class of C++) and multidimensional arrays. In Java nested arrays were used and complex numbers were represented by pairs of doubles. In C++ the standard complex class was used and in C a complex number was represented with the gcc-specific type __complex__. It shows that Spar performs very well in comparison to other Java compilers. Of course the JDK has very poor performance because pure interpretation is used. The other Java programs suffer from the usage of nested arrays in comparison to the true 3D arrays of Spar.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Language</th>
<th>Exec.time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java form JDK 1.2.2</td>
<td>Java</td>
<td>161.80</td>
</tr>
<tr>
<td>Spar with g++</td>
<td>Java</td>
<td>15.43</td>
</tr>
<tr>
<td>javac with Symantex JIT</td>
<td>Java</td>
<td>13.69</td>
</tr>
<tr>
<td>g++</td>
<td>C++</td>
<td>12.46</td>
</tr>
<tr>
<td>Spar with g++</td>
<td>Spar</td>
<td>11.21</td>
</tr>
<tr>
<td>gcc</td>
<td>C</td>
<td>6.98</td>
</tr>
</tbody>
</table>

Table 5.2: Sequential timing results for FT
<table>
<thead>
<tr>
<th># proc.</th>
<th>Class S F77</th>
<th>HPF</th>
<th>Class W F77</th>
<th>HPF</th>
<th>Class A F77</th>
<th>HPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.39</td>
<td>4.61</td>
<td>22.59</td>
<td>28.94</td>
<td>86.24</td>
<td>106.03</td>
</tr>
<tr>
<td>2</td>
<td>2.09</td>
<td>2.54</td>
<td>12.31</td>
<td>14.99</td>
<td>40.44</td>
<td>54.50</td>
</tr>
<tr>
<td>3</td>
<td>1.82</td>
<td></td>
<td>10.44</td>
<td></td>
<td>37.07</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.33</td>
<td>1.45</td>
<td>8.03</td>
<td>9.01</td>
<td>25.25</td>
<td>28.84</td>
</tr>
<tr>
<td>8</td>
<td>0.73</td>
<td>1.14</td>
<td>4.58</td>
<td>5.82</td>
<td>12.93</td>
<td>17.53</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>5.60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.58</td>
<td>1.92</td>
<td>3.06</td>
<td>6.70</td>
<td>7.82</td>
<td>14.59</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td>6.52</td>
<td></td>
<td>15.13</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>2.71</td>
<td></td>
<td>7.43</td>
<td></td>
<td>15.29</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td>8.39</td>
<td></td>
<td>15.60</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0.56</td>
<td>4.49</td>
<td>1.90</td>
<td>9.93</td>
<td>4.90</td>
<td>17.87</td>
</tr>
</tbody>
</table>

Table 5.3: Timing results for CG benchmark

Figure 5.1: Timing results for CG - F77

Figure 5.2: Timing results for CG - HPF
CHAPTER 5. PERFORMANCE RESULTS

<table>
<thead>
<tr>
<th># proc.</th>
<th>Class S</th>
<th>Class W</th>
<th>Class A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F77 Spar</td>
<td>F77 Spar</td>
<td>F77 Spar</td>
</tr>
<tr>
<td>1</td>
<td>39.42</td>
<td>78.50</td>
<td>101.5</td>
</tr>
<tr>
<td>2</td>
<td>19.00</td>
<td>38.00</td>
<td>50.2</td>
</tr>
<tr>
<td>3</td>
<td>12.77</td>
<td>25.38</td>
<td>202.75</td>
</tr>
<tr>
<td>4</td>
<td>9.83</td>
<td>19.63</td>
<td>25.4</td>
</tr>
<tr>
<td>8</td>
<td>4.84</td>
<td>9.67</td>
<td>12.5</td>
</tr>
<tr>
<td>12</td>
<td>3.33</td>
<td>6.50</td>
<td>9.2</td>
</tr>
<tr>
<td>16</td>
<td>2.45</td>
<td>4.90</td>
<td>6.4</td>
</tr>
<tr>
<td>20</td>
<td>2.00</td>
<td>4.09</td>
<td>5.2</td>
</tr>
<tr>
<td>24</td>
<td>1.70</td>
<td>3.37</td>
<td>5.0</td>
</tr>
<tr>
<td>28</td>
<td>1.54</td>
<td>2.91</td>
<td>2.24</td>
</tr>
<tr>
<td>32</td>
<td>1.23</td>
<td>2.46</td>
<td>1.90</td>
</tr>
</tbody>
</table>

Table 5.4: Timing results for EP benchmark

![Figure 5.3: Timing results for EP - F77](image)

![Figure 5.4: Timing results for EP - Spar](image)
Table 5.5: Timing results for FT benchmark

<table>
<thead>
<tr>
<th># proc.</th>
<th>Class S</th>
<th>Class W</th>
<th>Class A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F77 HPF Spar</td>
<td>F77 HPF Spar</td>
<td>F77 HPF Spar</td>
</tr>
<tr>
<td>1</td>
<td>23.50 24.24 20.3</td>
<td>48.30 56.90 42.2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12.73 13.01 12.8</td>
<td>26.22 29.43 26.8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9.46   16.6</td>
<td>21.40 35.7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3.32   3.43   5.7</td>
<td>6.75  7.76  16.3</td>
<td>111.18 149.86</td>
</tr>
<tr>
<td>12</td>
<td>2.77   4.2</td>
<td>5.75  9.8</td>
<td>103.97</td>
</tr>
<tr>
<td>16</td>
<td>1.63   1.77   3.5</td>
<td>3.42  3.91  8.3</td>
<td>57.35  74.68</td>
</tr>
<tr>
<td>20</td>
<td>1.81   3.6</td>
<td>3.76  8.2</td>
<td>66.15</td>
</tr>
<tr>
<td>24</td>
<td>1.41   3.2</td>
<td>3.69  7.6</td>
<td>56.00</td>
</tr>
<tr>
<td>28</td>
<td>1.37</td>
<td>3.64</td>
<td>46.99</td>
</tr>
<tr>
<td>32</td>
<td>0.80   0.93</td>
<td>1.70  2.07</td>
<td>29.41  38.27</td>
</tr>
</tbody>
</table>

Figure 5.5: Timing results for FT - F77

Figure 5.6: Timing results for FT - HPF

Figure 5.7: Timing results for FT - Spar
CHAPTER 5. PERFORMANCE RESULTS

<table>
<thead>
<tr>
<th># proc</th>
<th>Class S F77</th>
<th>Class W F77</th>
<th>Class A F77</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.20</td>
<td>3.50</td>
<td>29.41</td>
</tr>
<tr>
<td>2</td>
<td>0.14</td>
<td>2.28</td>
<td>19.22</td>
</tr>
<tr>
<td>4</td>
<td>0.06</td>
<td>1.49</td>
<td>12.09</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.05</td>
<td>0.53</td>
<td>4.64</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0.06</td>
<td>0.32</td>
<td>2.96</td>
</tr>
</tbody>
</table>

Table 5.6: Timing results for IS benchmark

Figure 5.8: Timing results for IS - F77
### Table 5.7: Timing results for MG benchmark

<table>
<thead>
<tr>
<th># proc</th>
<th>Class S</th>
<th>Class W</th>
<th>Class A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F77</td>
<td>HPF</td>
<td>F77</td>
</tr>
<tr>
<td>1</td>
<td>0.37</td>
<td>0.76</td>
<td>27.04</td>
</tr>
<tr>
<td>2</td>
<td>0.21</td>
<td>0.84</td>
<td>14.40</td>
</tr>
<tr>
<td>3</td>
<td>0.84</td>
<td></td>
<td>49.09</td>
</tr>
<tr>
<td>4</td>
<td>0.13</td>
<td>0.83</td>
<td>8.74</td>
</tr>
<tr>
<td>8</td>
<td>0.08</td>
<td>0.69</td>
<td>4.01</td>
</tr>
<tr>
<td>12</td>
<td>0.68</td>
<td></td>
<td>28.88</td>
</tr>
<tr>
<td>16</td>
<td>0.06</td>
<td>0.68</td>
<td>2.38</td>
</tr>
<tr>
<td>20</td>
<td>0.63</td>
<td></td>
<td>22.20</td>
</tr>
<tr>
<td>24</td>
<td>0.57</td>
<td></td>
<td>18.70</td>
</tr>
<tr>
<td>28</td>
<td>0.55</td>
<td></td>
<td>18.03</td>
</tr>
<tr>
<td>32</td>
<td>0.04</td>
<td>0.54</td>
<td>1.47</td>
</tr>
</tbody>
</table>

**Figure 5.9: Timing results for MG - F77**

**Figure 5.10: Timing results for MG - HPF**
Chapter 6

Advantages, Disadvantages, and General Issues

In this section we will summarize the advantages and disadvantages of Spar for parallel programming (also some non-parallel issues are mentioned) and discuss some of them in more detail. The last section describes some general parallel issues, that are already mentioned previously in this document and still need to be dealt with.

6.1 Advantages

- Expressive base language
  Compared to Fortran, Java usually needs fewer lines of code to express the same algorithm. Furthermore, the support of array interfaces will add even more to the readability of Spar code.

- Native support for the type complex

- Support for multidimensional arrays
  For data-parallelism this is not so much an advantage as a necessity.

- Support for most day-to-day regular data distributions
  The support of block, cyclic, and all together with support for multiple dimensions are often sufficient to express the mappings that are needed.

- Good task-parallel constructs
  An each or foreach (together with an independent annotation) gives sufficient support for most task-parallel algorithms. Although the Series Parallel paradigm does not capture all kinds of task-parallelism it does provide the programmer with well structured and readable code. However, a quite important set of algorithms requires the support of processor subsets, and this is not yet implemented.

- Support for Farmer Worker model
  The use of in the on annotation together with a (runtime) scheduler could help the programmer to a great extent with annotating irregular parallelism.

6.2 Disadvantages

- No align methodology
  Although it is more a convenience than a necessity to have support for alignment, the programmer could gain from such an annotation.
• No redistribute annotation
  It is not possible in Spar to redistribute a memory block. The only option is to introduce a
  new variable with the new distribution and copy the contents. However, when the variable
  is used in a loop and at each loop step the distribution needs to change, we would have to
  introduce a new variable for every loop step.
  Furthermore, Spar is even more restricted, since it also does not allow an array reference to
  switch form pointing to an array with one type of distribution to a similar sized array with
  another distribution. This problem is discussed in the last section of this chapter.

• Obligatory use of a blocksize parameter in a block and cyclic annotation
  It would be very convenient for the programmer if the blocksize parameter could be omitted
  if it is equal to $N_{\text{elements}}/N_{\text{processors}}$ for a block distribution or 1 for a cyclic distribution
  (these are the most common occurrences of these types of distributions).

• No garbage collection

• No ‘local’ annotation
  It would be very useful if variables could be created outside a foreach with an annotation
  describing that the variable is to be used locally on a processor and does not have to be kept
  in sync with the ones on other processors.

• Only one type of view on a set of physical processors is possible
  Currently, with the processors annotation in Spar one can specify that a group of processors
  is to be treated as a multidimensional processors set. However, each processor can only be
  appointed to one group, which means that a group of physical processors can not be treated
  as both a one dimensional and a three dimensional set.

• All communication is regarded equal
  Spar does not take into account that some processors may lie further apart in a parallel
  architecture then others and that communication between one couple of processors can
  thus be more expensive then communication between two other processors. Several parallel
  algorithms described in literature, like for instance AMS-CG, a multi-subgrid variant of the
  CG algorithm, make use of the fact that communication between adjacent processors is
  cheaper then communication between processors that are further away from each other in a
  processor grid. Because Spar does not guarantee that two adjacent processors in a multi-
  dimensional processor set are also adjacent in the hardware processor grid, such algorithms
  will lose their advantage when implemented in Spar.

• No support for non Series Parallel task-parallelism
  For instance, pipelining algorithms can not be expressed in Spar without degrading the
  amount of parallelism.

6.3 General Issues

In Section 3.1 we mentioned that the Object Oriented paradigm and parallelism pose some
fundamental problems for abstraction of BLAS routines. Suppose we have a BLAS 2 routine that
multiplies a matrix by a vector. In the non-parallel form we would have a routine for each type of
matrix: one for a lower triangular matrix, one for a dense matrix, one for a three diagonal matrix,
etc.

The problem is, that if we want to perform such a routine in parallel, the compiler has to know
about the distributions of the matrix and the vector it is multiplied by. If the ‘MatVecMul’
method is part of a matrix class, then we have have to introduce a new class for the same sparse
matrix structure for every distribution we want to have. Furthermore, because the vector that
is passed as a parameter to the multiplication method can also have a distribution we will also
have to annotate this distribution to the method parameter. This means that we have to create
a separate method for each vector distribution. Furthermore, the out-vector (for \( y = Ax \), this is the vector \( y \) ) can have a different distribution than the in-vector (\( x \)). This means that we also have to have a separate method for each type of out-vector distribution. So if \( N_{out} \) is the amount of different distributions for the matrix, \( N_{in} \) the amount for the in-vector, and \( N_{out} \) the amount for the out-vector, than for each sparse-matrix class we had in the beginning we will have to create \( N_{out} \times N_{in} \times N_{out} \) matrix-vector multiplication methods that are completely identical in operations but only differ in the kind of annotations that are given to the matrices and vectors.

In my opinion this is not a workable solution. A possible solution would be to expand the typed interfaces of Spar to incorporate distributions of types and to allow multiple types. In this way we would only have to supply the matrix-vector multiplication method once with the use of parameterized vector and matrix types. The compiler will then create only as many copies of this method as there are combinations of distributions used in the program.

Another issue relates to the swap of differently distributed arrays in the 1D FFT code as described in Section 4.1. Because Spar currently links the distribution of a variable to its type, it is not possible to change an array reference from pointing from one array structure with one distribution to another array structure with another distribution. For example, the following piece of code will not be accepted by Spar:

```plaintext
1       <$> on=(lambda (i) Gpp[(block i 10)]) $> int[10] x;
2       <$> on=(lambda (i) Gpp[(block i 20)]) $> int[10] y;
3       x = <$> on=(lambda (i) Gpp[(block i 10)]) $> new int[10];
4       y = x;
```

because in the last line the types of \( x \) and \( y \) do not agree. It is still an unsolved issue how to deal with situations where this conflict occurs (for instance how to rewrite the 1D FFT loop in an efficient way).
Chapter 7

Recommendations

In order to turn some of the disadvantages (or short-comings) of Spar into advantages, in the first section below a list of possible extensions to Spar is given. In the section thereafter some suggestions for further research are given.

7.1 Recommended Additions and Changes for Spar

7.1.1 Loadtime specification of Number Of Processors

For cases where the parallel expressiveness of the Spar annotations is not sufficient the programmer must be able to fall back to a more explicit way of parallelizing code. The availability of an intrinsic NumberOfProcessors is vital for this.

7.1.2 No blocksize parameter within block and cyclic annotations

The current definitions for the block and cyclic parameters in the on annotation currently require at all times both an index parameter and the blocksize. In most cases, however, for a block distribution, the blocksize is equal to the range size divided by the amount of processors. In these cases it would be convenient if the programmer could omit the blocksize parameter and have the compiler calculate this blocksize by performing the division. For the cyclic distribution a syntactic sugar option could be introduced, that allows the programmer to automatically assign a blocksize 1 by omitting the blocksize parameter in a cyclic distribution.

7.1.3 A local annotation

As mentioned in the implementation description of several NPB kernels (for instance MG) a local annotation, that tells the compiler to replicate a dataset without updating, is necessary to be able to use temporary arrays in a foreach loop without having multiple copies of this array on each processor. The straightforward approach would be to introduce a _local that is equal to the _all parameter of the on annotation, but with the extra property that replicated data will not get updated. With this construct we could use <$ on=_local $> to have a whole array mapped locally on all processors and for the distribution of the temporary vectors in Section 3.3.5 we could use the annotation <$ on=(lambda (i j k) Gpp3D[1 to n,1 to n,block k)) $>.

7.1.4 Processor Subsets

In Section 4.2 it was mentioned that recursive parallel algorithms could not be expressed with the current annotations, since there is no way to split the processor set into two or more subsets. Furthermore, in general it is useful to have a way of mapping a task onto a subset of processors and not just 1 or all processors (of course there is always the way of creating a multidimensional
processor grid and collapsing one or more dimensions, but this is neither elegant nor always sufficient). A good (set of) annotation(s) still needs to be found to be able to efficiently map a task on a processor subset.

### 7.1.5 Multiple processor views

Currently the processors annotation only allows one type of view for a set of processors. That is, if a set of physical processors are described by a two dimensional processor array, they can not be efficiently used anymore to map a three dimensional array on. Therefore it is recommended to initialize all processor groups as one dimensional arrays and create a new kind of annotation types that are multidimensional views on this one dimensional processor array. In this way a set of processors can both be referenced to as a 1D, 2D, 3D, or even more dimensional processors array and the same group of processors can be used to both map matrices as 3D fields on. A good set of annotations to express this needs to be found.

### 7.1.6 Garbage collection

Although garbage collection is already on the list of items that need to be implemented, it remains a crucial issue. For a serious (sequential) comparison of Spar with other Java compilers such an engine should be implemented rather sooner than later.

### 7.1.7 Feedback by the compiler

Of course a compiler needs to have decent feedback on errors and warnings it has found, but the following aspects are also very useful for the programmer and if possible should therefore also be accessible:

- Whether and how arrays are distributed and optionally the amount of memory used locally on each processor.
- The reductions the compiler found. Since the compiler should recognize reductions automatically this is an important feature.
- Parallel profiling information (either by the compiler or an external tool).

### 7.2 Research Follow-up

#### 7.2.1 General parallel issues

Some aspects that may require further investigation are, as already mentioned in Section 4.4, file I/O and I/O towards the end-user. Another issue not yet mentioned is parallel exception handling in a each and foreach. This problem is quite similar to the search-routine issue described in Section 4.4.1, since both the Exception and a break try to exit a running foreach loop. A strict semantical definition for the throw of an Exception in a foreach thus still needs to be given.

#### 7.2.2 Other algorithms

As mentioned in Section 4.4 not all types of parallel algorithms were covered in this thesis. Already some possibly interesting classes of algorithms were mentioned. Below a more extended list is given. These algorithms might be worth investigating further on parallelization within Spar.

- Pipelining algorithms
  In Section 4.4.2 the problems with efficiently parallelizing these types of algorithms were already discussed. A real-life example might be useful to determine whether, and if so, how, Spar should be extended to support these kinds of algorithms.
• N-body problems
  Particle simulation algorithms as mentioned in Section 4.4.3.

• One of the NPB application benchmarks
  Probably implementation of only one of the applications will suffice, since all of the three
  application benchmarks work with k-diagonal block matrices. Thus if parallelization of one
  of these benchmarks in Spar poses no problems, the other probably won’t either.

• The Genesis application benchmarks [1]
  These benchmarks contain several algorithms from fields like meteorology, quantum chromo-
  dynamics, molecular dynamics, general relativity, and electro-magnetism.

• Linear optimization
  For instance the Simplex method, the Internal Point method, or a Dynamic Programming
  algorithm.

• Chemistry problems
  Chemistry is a field that often uses High Performance Computing for calculation on molec-
  ular dynamics problems. Therefore implementation of such algorithms in Spar are worth
  investigating.

• Rendering and ray-tracing algorithms
  3D graphics is also a field that makes heavy use of computing power. The issue of imple-
  menting these kinds of applications in Spar becomes even more interesting when not only
  task parallelism is used, but when rendering data (object descriptions, etc.) needs to be
  distributed among the available processors.
Chapter 8

Conclusions

Overall the Spar language was capable of expressing the parallelism in the NAS Parallel Benchmark kernels. Some parts, however, could not be parallelized efficiently. For sequential loops, the loop often had to be split in batches. In a few occasions that was also not sufficient and an explicit parallel loop that depends on a not yet supported intrinsic NumberOfProcessors() was needed. Furthermore, Spar should be extended with a local annotation in order to use temporary arrays/objects in parallel loops (the full set of recommended extensions to Spar can be found in Chapter 7).

The SP task parallel paradigm appeared to be sufficient to express the task parallelism of the NPB kernels. Still, Spar lacks the possibility to map tasks on a subset of processors, which is especially needed in (binary) recursive algorithms.

Finally, indirect array addressing, like encountered in the IS benchmark, still remains an unsolved issue.

Looking at the first timing results, we can conclude that for sequential execution the Spar compiler compares very well to other sequential compilers. Although, we have to remark that Spar does not yet have support for garbage collection, so the comparison may not be that fair.

For parallel execution Spar already shows decent speed up curves and comes within a factor two of the Fortran MPI implementation. However, more of the benchmarks need to be run in order to have a good comparison.
Bibliography


Appendix A

Spar-NPB source codes

A.1  EP.spar

```java
import Random;
import Results;
import Timer;

public class EP {
    static final char Class = ClassData.Class;
    static final int M = ClassData.M;
    static final double [] TestValue = ClassData.TestValue;

    static final int BatchSize = 1024;

    static public void main(String [] args) {
        final double GenerationSeed = 271828183.0;
        final double GenerationMult = 1220703125.0;
        final double epsilon = 1.0E-8;
        double timecounter, mops;
        final Timer timer = new Timer();
        long N;
        int NBatches;
        double sx, sy;
        final int[] Q = new int[10];
        int globalQ;
        boolean passedVerification;

        N = 1;
        N >>= M;
        NBatches = (int) (N/BatchSize);

        System.out.println("N = 1
        N >>= M;
        NBatches = (int) (N/BatchSize);

        for (i := 10)
            Q[i] = 0;
        sx = 0.0;
```
sy = 0.0;

timer.Clear();
timer.Start();

foreach (k := :NBatches) {
    Random random = new Random(GenerationSeed,GenerationMult);
    random.Skip((double)2*k*BatchSize);
    for (i := :BatchSize) {
        double x = 2 * random.RandLC() - 1;
        double y = 2 * random.RandLC() - 1;
        double t = x*x + y*y;
        if (t <= 1.0) {
            t = Math.sqrt(-2.0*Math.log(t)/t);
            x *= t;
            y *= t;
            Q[(int)Math.max(Math.abs(x),Math.abs(y))]++;
            sx += x;
            sy += y;
        }
    }
    _delete(random);
}
timer.Stop();
timecounter = timer.Read();
globalQ = 0;
for (i := :10)
    globalQ += Q[i];
System.out.println();
System.out.println(" No. Gaussian Pairs: "+globalQ);
System.out.println(" Sums: "+sx+" "+sy);
System.out.println(" Counts: ");
for (i := :10)
    System.out.println(" "+i+" "+Q[i]);

passedVerification = ((sx-TestValue[0])/sx < epsilon) &&
    ((sy-TestValue[1])/sy < epsilon);
mops = Math.pow(2,M+1)/timecounter/1000000.0;
Results.Print(Class,M+1,0,0,0,timecounter,mops,
    "Random numbers generated",passedVerification);
A.2 MG.spar

```java
import ClassData;
import Random;
import Results;
import Timer;

public class MG {
    static final char Class = ClassData.Class;
    static final int N = ClassData.N;
    static final int M = N+2; // add ghost edges
    static final int MaxIterations = ClassData.MaxIterations;
    static final double VerifyValue = ClassData.VerifyValue;

    static int MaxGridLevel;

    static final double[] S = ClassData.S;
    static final double[] A = { -8.0/3.0, 0.0, 1.0/6.0, 1.0/12.0};

    static int ILog2(int n) {
        int l = 0;
        while (n>1) {
            n >>= 1;
            l++;
        }
        return l;
    }

    static void MakeZero(final double[] u) {
        foreach (i := u.getSize())
            u[i] = 0.0;
    }

    static [double,double] GiveNorm(final double[] r) {
        final int M = r.getSize()[0]; // r is M*M*M, M=M+2
        double norm2, norminf;
        norm2 = 0.0;
        norminf = 0.0;
        foreach (i := [1,1,1]:[M-1,M-1,M-1]) {
            norm2 += r[i] * r[i];
            norminf = Math.max(norminf, Math.abs(r[i]));
        }
        M -= 2;
        norm2 = Math.sqrt(norm2/((double)M*M*M));
        return [norm2,norminf];
    }

    static void InitNonZeros(final double[] v) {
        final double Seed = 314159265.0;
        final double Mult = 1220703125.0;
        final int N = 10;
    }
```
final int M = v.getSize()[0]; // v = M×M×M
final double[][] small = new double[N];
final double[][] high = new double[N];
final [int][ ] spos = new [int][ ] [N];
final [int][ ] hpos = new [int][ ] [N];
Random random = new Random(Seed,Mult);

foreach (i := :N) {
    small[i] = 1.0;
    high[i] = 0.0;
    spos[i] = [0,0,0];
    hpos[i] = [0,0,0];
}

// Parallelising this loop is not easy because of the RandLC()’s
for (k := 1:M-1, j := 1:M-1, i := 1:M-1) {
    double r = random.RandLC();
    if (r > high[0]) {
        high[0] = r;
        hpos[0] = [i,j,k];
        int s = 0;
        while (s<N-1 && high[s]>high[s+1]) {
            [high[s], high[s+1]] = [high[s+1], high[s]];
            [hpos[s], hpos[s+1]] = [hpos[s+1], hpos[s]];
            s++;
        }
    }
    if (r < small[0]) {
        small[0] = r;
        spos[0] = [i,j,k];
        int s = 0;
        while (s<N-1 && small[s]<small[s+1]) {
            [small[s], small[s+1]] = [small[s+1], small[s]];
            [spos[s], spos[s+1]] = [spos[s+1], spos[s]];
            s++;
        }
    }
    MakeZero(v);
}

foreach (i := :N) {
    v[hpos[i]] = +1.0;
    v[spos[i]] = -1.0;
}

static void UpdateBorders(final double [*,*,*] u) {
    // update repeating borders
    final int M = u.getSize()[0]; // u is M×M×M, M=N+2
    for (i := 1:M-1) {
        for (j := 1:M-1) {
            u[i,j,0] = u[i,j,M-2];
            u[i,j,M-1] = u[i,j,1];
        }
    }
}
void ApplySmooother(final double [*,*,*] u,
    final double [*,*,*] r) {
    int M = u.getSize()[0]; // u and r are MxMxM,
    double [] r1 = new double [M];
    double [] r2 = new double [M];
    // r1 and r2 should actually be declared in the loop!
    for (int i = 1; i < M-1; i++) {
        for (int k = 1; k < M-1; k++) {
            r1[i] = r[i-1,i] + r[i+1,i] +
                    r[i-1,i,k] + r[i+1,i,k];
            r2[i] = r[i-1,i-1,k] + r[i-1,i+1,k] +
                    r[i+1,i-1,k] + r[i+1,i+1,k];
        }
    }
    for (int k = 1; k < M-1; k++) {
        u[i,j,k] += S[0] * r[i,j,k] +
                    S[1] * (r[i,j,k-1] + r[i,j,k+1] + r1[k]) +
                    S[2] * (r2[k] + r1[k-1] + r1[k+1]) +
                    S[3] * (r2[k-1] + r2[k+1]);
        // S[3] is probably 0, but the compiler should
        // recognise this and eliminate the S[3] part
    }
    // Drastic need for garbage collector
```java
  _delete(r1);
  _delete(r2);
  UpdateBorders(u);
}

  static void EvaluateResidual(f final double [*,*,*] r,
                          f final double [*,*,*] v, f final double [*,*,*] u) {
    // r = v - A * u
    // (v and r are references to the same array in most cases!)
    final int M = u.getSize()[0]; // u, v and r are MsxM, M=N+2
    final double [] u1 = new double [M];
    final double [] u2 = new double [M];
    foreach (i :- 1:M-1, j :- 1:M-1) {
      foreach (k :- :M) {
        u1[k] = u[i,j-1,k] + u[i,j+1,k] +
        u[i-1,j,k] + u[i+1,j,k];
        u2[k] = u[i-1,j-1,k] + u[i-1,j+1,k] +
        u[i+1,j-1,k] + u[i+1,j+1,k];
      }
      foreach (k :- 1:M-1) {
        r[i,j,k] = v[i,j,k] - (A[0] * u[i,j,k] +
        A[1] * (u[i,j,k-1] + u[i,j,k+1] + u1[k]) +
        A[2] * (u2[k] + u[i,k-1] + u[i,k+1]) +
        A[3] * (u2[k-1] + u2[k+1]));
        // A[1] is probably 0, but the compiler should
        // Otherwise just comment this line
      }
    }

    // Drastic need for garbage collector
    _delete(u1);
    _delete(u2);
    UpdateBorders(r);
}

  static void RestrictResidual(f final double [*,*,*] r1,
                             f final double [*,*,*] r2) {
    // r_{l-1} = P * r_{l-1}, r2 = P * r1
    // (l and k run the other way round!)
    final int M1 = r1.getSize()[0]; // r1 is MsxM1xM1, M1=N+2
    final int M2 = r2.getSize()[0]; // M2 = N+2 + 2
    final double [] x1 = new double [M1];
    final double [] x2 = new double [M1];
    foreach (i :- 1:M2-1, j :- 1:M2-1) {
      final int i1 = i*2;
      final int j1 = j*2;
      foreach (k1 :- :M1) {
```
\[
x_1[k_1] = r_1[i_1,j_1-1,k_1] + r_1[i_1,j_1+1,k_1] +
\]
\[
r_1[i_1-1,j_1,k_1] + r_1[i_1+1,j_1,k_1];
\]
\[
x_2[k_1] = r_1[i_1-1,j_1-1,k_1] + r_1[i_1-1,j_1+1,k_1] +
\]
\[
r_1[i_1+1,j_1,k_1] + r_1[i_1+1,j_1+1,k_1];
\]

```
foreach (k := 1:M2-1) {
    final int k1 = k*2;
    r2[i,j,k] = 0.5 * r1[i1,j1,k1] +
        0.25 * (r1[i1,j1,k1-1] + r1[i1,j1,k1+1] + x1[k1]) +
        0.125 * (x2[k1] + x1[k1-1] + x1[k1+1]) +
        0.0625 * (x2[k1-1] + x2[k1+1]);
}
```

// Drastic need for garbage collector

```
~delete(x1);
~delete(x2);
UpdateBorders(r2);
```

```
static void Prolongate(final double [*,*,*] u1,
    final double [*,*,*] u2) {
    // u_1[0] = Q * u_1[0], u2 = Q * u1
    // (I and k run the other way round)
    final int M1 = u1.getSize()[0]; // u1 is M1*M1*M1, M1=N1+2
    final int M2 = u2.getSize()[0]; // M2 = N1*2 + 2
    final double [][] x1 = new double [M1];
    final double [][] x2 = new double [M1];
    final double [][] x3 = new double [M1];

    foreach (i := M1-1, j := M1-1) {
        final int i2 = i*2;
        final int j2 = j*2;
        foreach (k := M1) {
            x1[k] = u1[i,j,k] + u1[i+1,j,k];
            x2[k] = u1[i,j,k] + u1[i,j+1,k];
            x3[k] = x1[k] + u1[i,j+1,k] + u1[i+1,j+1,k];
        }
    }

    foreach (k := M1-1) {
        final int k2 = k*2;
        u2[i2,i2,k2] += u1[i,j,k];
        u2[i2+1,i2,k2] += 0.5 * x1[k];
        u2[i2,i2+1,k2] += 0.5 * x2[k];
        u2[i2,i2,k2+1] += 0.5 * (u1[i,j,k] + u1[i,j,k+1]);
        u2[i2+1,i2,k2+1] += 0.25 * x3[k];
        u2[i2+1,i2+1,k2+1] += 0.25 * (x1[k] + x1[k+1]);
        u2[i2,i2+1,k2+1] += 0.25 * (x2[k] + x2[k+1]);
        u2[i2+1,i2+1,k2+1] += 0.125 * (x3[k] + x3[k+1]);
    }
```

// Drastic need for garbage collector

```
~delete(x1);
```
```java
    _delete(x2);
    _delete(x3);
}

class RecursiveSolve(final double [*,*,*] u,
    final double [*,*,*] r, final double [*,*,*] v, final int K) {
    final int M = u.getSize() + 1;  // u = M*M*M, M=N+2
    for (k := K-1)
        RestrictResidual(r[k], r[k+1]);
    MakeZero(u[K-1]);
    ApplySmooth(u[K-1], r[K-1]);
    for (int k=K-2; k>=0; k--)
        MakeZero(u[k]);
        Prolongate(u[k+1], u[k]);
        CalculateResidual(r[k], r[k], u[k]);
        ApplySmooth(u[k], r[k]);
    Prolongate(u[1], u[0]);
    CalculateResidual(r[0], v, u[0]);
    ApplySmooth(u[0], r[0]);
}

public static void main(String [] args) {
    final double GenerationSeed = 271828183.0;
    final double GenerationMult = 1220703125.0;
    final double epsilon = 1.0E-8;
    double time = 0;
    double timer = new Timer();
    boolean passedVerification;
    double norm2, norminf;
    double [*,*,*] u;
    double [*,*,*] r;
    double [*,*,*] v;
    System.out.println("NAS Parallel Benchmarks SPAR version " +
        "- MG Benchmark\n");
    System.out.println(" Size: " + N + " + \n + N + \n + N + " + Class + + " + MaxIterations);
    System.out.println(" Iterations: " + MaxIterations);
    MaxGridLevel = ILog2(N);
    u = new double [MaxGridLevel][*,*,*];
    r = new double [MaxGridLevel][*,*,*];
    int m = N;
    for (k := MaxGridLevel) {
        u[k] = new double [m+2, m+2, m+2];
        r[k] = new double [m+2, m+2, m+2];
        m /= 2;
    }
    v = new double [N+2, N+2, N+2];
    MakeZero(u[0]);
```

InitNonZeros(v);

timer.Clear();
timer.Start();

for (i := :MaxIterations) {
    EvaluateResidual(r[0],v,u[0]);
    RecursiveSolve(u,r,v,MaxGridLevel);
}

EvaluateResidual(r[0],v,u[0]);
[norm2,norminf] = GiveNorm(r[0]);
timer.Stop();
timecounter = timer:Read();

passedVerification = (Math.abs(norm2 - VerifyValue) <= epsilon);
System.out.println();
System.out.println("L2 Norm = " + norm2);
System.out.println("Error = " + (norm2 - VerifyValue));
if (timecounter > 0)
    mops = 58.0*MaxIterations*N*N*N/timecounter/1000000.0;

Results.Print(Class,N,N,N,MaxIterations,timecounter,mops,
    "floating point",passedVerification);
}
A.3 CG.spar

```java
import ClassData;
import Random;
import Results;
import Timer;

public class CG {
    static final char Class = ClassData.Class;
    static final int dimA = ClassData.dimA;
    static final int MaxNonZerosV = ClassData.MaxNonZerosV;
    static final int MaxNonZerosA = dimA*(MaxNonZerosV+1)*(MaxNonZerosV+1)
        + dimA*(MaxNonZerosV+2);
    static final double shift = ClassData.shift;
    static final double rcond = ClassData.rcond;
    static final double zetaVerVal = ClassData.zetaVerVal;
    static final int MaxIterations = ClassData.MaxIterations;

    static double zeta;

    static Random random = new Random(0,0,0);

    static public [type double [], type int [], type int []] MakeA(
        final int dimA, final int MaxNonZerosA,
        final int MaxNonZerosV, final double rcond,
        final double shift) throwsToManyElementsException {
        // Below we use MaxNonZeros V+1 because an extra diagonal
        // element could be created in method SetVec
        final double [] V = new double [MaxNonZerosV+1];
        final int [] VIndex = new int [MaxNonZerosV+1];

        // List of nonzero matrix elements (double entries allowed)
        final double [] AElm = new double[MaxNonZerosA];
        final int [] ARow = new int[MaxNonZerosA];
        final int [] AColl = new int[MaxNonZerosA];

        // Sparse representation of matrix A
        final double [] A = new double[MaxNonZerosA];
        final int [] ARowOff = new int[dimA+1];
        final int [] ACollOff = new int[MaxNonZerosA];

        int row, col;
        double scale, size = 1.0, ratio;
        int numNonZerosV, numNonZerosA = 0;

        // Build symmetric sparse matrix out of outer products of
        // sparse vectors

        ratio = Math.pow(rcond,(1.0/dimA));

        for (iteration := 0; iteration < dimA) {
            SparseVec(V,VIndex,dimA,MaxNonZerosV);
            SetVec(V,VIndex,iteration,0.5);
            numNonZerosV = V.getSize()[0];
        }
    }
}
```
foreach (j := numNonZerosV) {
    col = VIndex[j];
    scale = size * V[j];
    foreach (i := numNonZerosV) {
        int k = numNonZerosA++;
        row = VIndex[i];
        if (k+1 >= MaxNonZerosA)
            throw new TooManyElementsException(iteration);
        ARow[k] = row;
        ACol[k] = col;
        AElm[k] = scale * V[i];
    }
    size *= ratio;
}

foreach (i := xdimA) {
    int k = numNonZerosA++;
    if (k+1 >= MaxNonZerosA)
        throw newToManyElementsException(dmA+i);
    ARow[k] = i;
    ACol[k] = i;
    AElm[k] = rcond - shift;
}
AElm.setSize(numNonZerosA);
SparseMat(A, ARow, ACol, xdimA, AElm, ACol, ARow);

_delete(AElm);
_delete(ARow);
_delete(ACol);
return [A, ARow, ACol, Index];
}

static public void SparseVec(final double[] V, final int[] VIndex,
final int dimV, final int MaxNonZeros) {
    int NPow2 = 1;
    int k = 0;
    double value;
    int index;
    boolean doubleOccurance;
    V.setSize([MaxNonZeros]);
    VIndex.setSize([MaxNonZeros]);
    // NPow2 is the smallest power of two not less than dimA
    while (NPow2 < dimA)
        NPow2 <<= 1;
    // Create non-zero vector values
    while (k < MaxNonZeros) {
        do {

value = random.RandLC();
index = (int) (NPow2 * random.RandLC());
} while (index >= dimV);
// Check for double position of vector elements
doubleOccurrence = false;
for (i := :k)
  if (index==VIndex[i])
    doubleOccurrence = true;
if (doubleOccurrence) {
  V[k] = value;
  VIndex[k] = index;
  k++;
}
}

static public void SetVec(final double [][] V, final int [][] VIndex,
final int diag, final double value) {
  boolean occurs = false;
  int numNonZeros;
  int k = 0;

  numNonZeros = V.getSize()[0];
  for (i := :numNonZeros)
    if (VIndex[i]==diag) {
      occurs = true;
      k = i;
    }
  if (occurs) {
    k = numNonZeros++;
    V.setSize([numNonZeros]);
    VIndex.setSize([numNonZeros]);
    VIndex[k] = diag;
  }
  V[k] = value;
}

static public void SparseMat(final double [][] A, final int [][] ARowOff,
final int [] AColIndex, final int dimA,
final double [] AElm, final int [] ACol,
final int [] ARow) {
  final boolean [] mark = new boolean [dimA];
  final double [] x = new double [dimA];
  final int numNonZerosA = AElm.getSize()[0];

  int numNonZerosRow;
  final int [] nonZeroLoc = new int [dimA];

    foreach (i := :dimA) {
      ARowOff[i] = 0;
      mark[i] = false;
      x[i] = 0.0;
    }
ARowOff[\text{dimA}] = 0;

\textbf{for} (i := :numNonZerosA)
\begin{align*}
&\text{ARowOff}[\text{ARow}[i]+1]++;
&\textbf{for} (i := :\text{dimA})
\end{align*}
\begin{align*}
&\text{ARowOff}[i+1] += \text{ARowOff}[i];
\end{align*}

// buckets AElm and put result in A
\textbf{foreach} (i := :numNonZerosA) {
\begin{align*}
&\text{int} \ \row = \text{ARow}[i];
&\text{int} \ \off = \text{ARowOff}[\row]++;
&\text{A}[\off] = \text{AElm}[i];
&\text{AColIndex}[\off] = \text{ACol}[i];
\end{align*}
}

// We used ARowOff as offsetcounter and now have to reshift it
\textbf{for} (\text{int} \ i=\text{dimA}; i\geq 0; i--)
\begin{align*}
&\text{ARowOff}[i] = \text{ARowOff}[i-1];
&\text{ARowOff}[0] = 0;
&\text{numNonZerosA} = 0;
\end{align*}

\begin{align*}
&\text{int} \ \off = 0;
&\textbf{for} (i := :\text{dimA}) {
&\quad \text{numNonZerosRow} = 0;
&\quad \textbf{for} (k := :\text{numNonZerosRow}) {
&\quad \quad \text{int} \ \j = \text{AColIndex}[k];
&\quad \quad \text{x}[\j] += \text{A}[k];
&\quad \quad \textbf{if} (!\text{mark}[\j] \&\& \text{x}[\j]! = 0.0) {
&\quad \quad \quad \text{mark}[\j] = \text{true};
&\quad \quad \quad \text{nonZeroLoc}[\text{numNonZerosRow}++] = \j;
&\quad \quad \}
&\quad \}
&\quad \textbf{for} (k := :\text{numNonZerosRow}) {
&\quad \quad \text{int} \ \j = \text{nonZeroLoc}[k];
&\quad \quad \text{mark}[\j] = \text{false};
&\quad \quad \textbf{if} (\text{x}[\j]! = 0.0) {
&\quad \quad \quad \text{A}[\text{numNonZerosA}] = \text{x}[\j];
&\quad \quad \quad \text{AColIndex}[\text{numNonZerosA}++] = \j;
&\quad \quad \}
&\quad \}
&\quad \text{x}[\j] = 0.0;
&\}
&\text{off} = \text{ARowOff}[i+1];
&\text{ARowOff}[i+1] = \text{ARowOff}[i] + \text{numNonZerosRow};
\}
&\text{A.setSize(\text{numNonZerosA});}
&\text{AColIndex.setSize(\text{numNonZerosA});}
&\}
\text{System.out.println();}
\text{System.out.println("final nonzero count in sparse");}
\text{System.out.println("number of nonzeros = "+\text{numNonZerosA});}
\}
\}
\]
\text{static public double ConGrad(final double[]} A, \text{final int []} \text{ARowOff},
final int [] AColIndex, final double [] b,
final double [] x) {
final int cgItMax = 25;
double rho0, rho, alpha, beta, nrm;
final double [] p = new double [dimA];
final double [] q = new double [dimA];
final double [] r = new double [dimA];

    foreach (i :- xdimA) {
        q[i] = 0.0;
        x[i] = 0.0;
        r[i] = b[i];
        p[i] = r[i];
    }

// rho = r.r
rho = 0.0;
foreach (i :- xdimA)
        rho += r[i] * r[i];

for (cgIt := cgItMax) {
    // q = A.p
    foreach (i :- :dimA) {
        q[i] = 0.0;
        foreach (k := ARowOff[i]:ARowOff[i+1])
            q[i] += A[k] * p[AColIndex[k]];
    }

    // alpha = rho / p.q
    alpha = 0.0;
    foreach (i :- :dimA)
        alpha += p[i] * q[i];
    alpha = rho / alpha;

    rho0 = rho; // save temporary of rho

    // x += alpha * p, r -= alpha * q
    foreach (i :- :dimA) {
        x[i] += alpha * p[i];
        r[i] -= alpha * q[i];
    }

    // rho = r.r
    rho = 0.0;
    foreach (i :- :dimA)
        rho += r[i] * r[i];

    // beta = rho / rho0
    beta = rho / rho0;

    // p = r + beta * p
    foreach (i :- :dimA)
        p[i] = r[i] + beta * p[i];
}
// Computer residual explicitly: ||r|| = ||b - A.x||
// r = b - A.x --> ||r||
    foreach (i := xdimA) 
        r[i] = b[i];
    foreach (k := ARowOff[i]:ARowOff[i+1])
        r[i] -= A[k] * x[AColIndex[k]];
}
    rmom = 0.0;
    foreach (i := xdimA)
        rmom += r[i] * r[i];
    rmom = Math.sqrt(Math.abs(rmom));
    return rmom;
}

static public void main(String [] args) {
    final double GenerationSeed = 314159265.0;
    final double GenerationMult = 1220703125.0;
    double timecounter, mops;
    final Timer timer = new Timer();
    double [][] A;
    int [] ARowOff;
    int [] AColIndex;
    final double [][] z = new double [dimA];
    final double [][] x = new double [dimA];
    double rmom, rmom1, rmom2;
    boolean passedVerification;

    System.out.println("NAS Parallel Benchmarks SPAR version " +
            "- CG Benchmark\n");
    System.out.println(" Size: " +dimA+" (class "+Class+")");
    System.out.println(" Iterations: " +MaxIterations);
    random.SetSeed(GenerationSeed,GenerationMult);
    // because of some performance based initialisations
    // in the Fortran example one call to RandLC is made
    // Therefor the next call is obligatory
    zeta = random.RandLC();

    try {
        [A,ARowOff,AColIndex] = MakeA(dimA,MaxNonZerosA,
            MaxNonZerosV,rcord,shift);
    } catch (TooManyElementsException e) {
        System.out.println(" Space for matrix elements exceeded in MakeA");
        System.out.println(" iteration "+(e.iteration+1)+" out of "+dimA);
        System.exit(1);
    }

    // Set starting vector to (1,1,...,1)
    foreach (i := xdimA)
        x[i] = 1.0;
    zeta = 0.0;
timer.Clear();
timer.Start();

System.out.println();
System.out.println("iteration, \|r\|, zeta");
for (iteration := MaxIterations) {
    norm = ConGrad(A,ARowOff,AColIndex,x,z);
    norm1 = 0.0;
    norm2 = 0.0;
    foreach (i := :dimA) {
        norm1 += x[i] * z[i];
        norm2 += z[i] * z[i];
    }
    norm2 = 1.0 / Math.sqrt(norm2);
    zeta = shift + 1.0 / norm1;
    System.out.println("C"+iteration+" ,"+norm+" ,"+zeta+");
    foreach (i := :dimA) {
        x[i] = norm2 * z[i];
    }
    timer.Stop();
    timecounter = timer.Read();
    passedVerification = Math.abs(zeta-zetaVal)<1E-10;
    if (timecounter>0) {
        mops = 2*MaxIterations*dimA * (3 + MaxNonZerosV*(MaxNonZerosV+1) + 3 + 25 * (5 + MaxNonZerosV*(MaxNonZerosV+1)) ) / timecounter / 1000000.0;
    } else
        mops = 0.0;

    Results.Print(Class,dimA,0,0,MaxIterations,timecounter,mops,
"floating point",passedVerification);
}

class TooManyElementsException extends Exception {
    int iteration = 0;
}

public TooManyElementsException() {
}

public TooManyElementsException(int it) {
    iteration = it;
}

A.4  FT.spar

```java
import ClassData;
import Random;
import Results;
import Timer;

public class FT {
    static final char Class = ClassData.Class;
    static final [int^3] Dim = ClassData.Dim;
    static final int TotalDim = Dim[0]*Dim[1]*Dim[2];
    static final complex [] TestValue = ClassData.TestValue;
    static final int MaxIterations = ClassData.MaxIterations;
    static final complex [] sums = new complex [MaxIterations];

    static int MaxDim;
    static complex [] omega;
    static double [] ex;
    static int [*,*,*] indexmap;

    static complex [] temp1;
    static complex [] temp2;

    static final double Pi = 3.141592653589793238;
    static final double alpha = 1.0E-6;

    static int ILog2(int n) {
        int l = 0;
        while (n>1) {
            n >>= 1;
            l++;
        }
        return l;
    }

    static void FFT_Init(final int N) {
        int m, k;
        double t;

        // omega[t] = e^((2*pi*i/2)*t)
        m = ILog2(N)+1;
        omega = new complex [m];
        k = 1;
        for (i :- :m) {
            t = 2.0*Pi/k;
            omega[i] = complex(Math.cos(t),Math.sin(t));
            k *= 2;
        }
    }

    static void FFT3D(final int is, final complex[*,*,*] u,
```
```java
final complex[*,*,*] v) {
  // is = forward(1)/backward(-1)
  final [int,3] N = u.getSize();
  temp1.setSize([MaxDim]);

  // dim 1
  foreach (j : -N[1], k : -N[2]) {
    for (i : -N[0])
      temp1[i] = u[i,j,k];
    FFT(is,N[0]);
    for (i : -N[0])
      v[i,j,k] = temp1[i];
  }

  // dim 2
  foreach (i : -N[0], k : -N[2]) {
    for (j : -N[1])
      temp1[j] = v[i,j,k];
    FFT(is,N[1]);
    for (j : -N[1])
      v[i,j,k] = temp1[j];
  }

  // dim 3
  foreach (i : -N[0], j : -N[1]) {
    for (k : -N[2])
      temp1[k] = v[i,j,k];
    FFT(is,N[2]);
    for (k : -N[2])
      v[i,j,k] = temp1[k];
  }

  static void FFT(final int is, final int N) {
    // is = normal(1)/inverse(-1)
    // x.length and y.length must be 2^M
    // x is input, y is temporary vector
    // both x and y will be altered
    int pt, pM;
    int id_to, id_from;
    complex om, om1;
    final int M = ILog2(N);
    complex [] x = temp1;
    complex [] y = temp2;

    for (t := 1:M+1) {
      pM = 1 << (M-t);
      pt = 1 << (t-1);
      if (is==1)
        om1 = Complex.conj(omega[t]);
      else
        om1 = omega[t];
      om = 1.0;
```
for (k := 0; pt) {
    for (j := N/2; pt) {
        id_from = j+k;
        id_to = id_from + j;
    
        complex x2 = om * x[id_from+N/2];
        y[id_to] = x[id_from] + x2;
        y[pt+id_to] = x[id_from] - x2;
    }
    om *= om1;
}

[x,y] = [y,x];

#include a = x;

static void ComputeInitialConditions(final complex [*,*,*] u) {
    final double GenerationSeed = 314159265.0;
    final double GenerationMult = 1220703125.0;
    double Real, Imag;
    double M;

    foreach (k := Dim[2]) {
        Random random = new Random(GenerationSeed, GenerationMult);
        random.Skip(2*k*Dim[1]*Dim[0]);
        for (j := Dim[1], i := Dim[0]) {
            Real = random.randLC();
            Imag = random.randLC();
            u[i,j,k] = complex(Real, Imag);
        }
    }

    __delete(random);

    static void ComputeIndexMap() {
        int expMax;
        int r, s, t;
        int d0, d1, d2;

        indexmap = new int [Dim[0], Dim[1], Dim[2]];
        [d0,d1,d2] = Dim;
        foreach (i := d0, j := d1, k := d2) {
            r = (i+d0/2)%d0 - d0/2;
            s = (j+d1/2)%d1 - d1/2;
            t = (k+d2/2)%d2 - d2/2;
            indexmap[i,j,k] = t*r + s*r + r*r;
        }

        expMax = MaxIterations*(Dim[0]*Dim[0] + Dim[1]*Dim[1] + Dim[2]*Dim[2]);
        ex = new double [expMax+1];
        ex[0] = 1.0;
        ex[1] = Math.exp(-4.0*alpha*Pi*Pi);
        for (i := 2*expMax+1)
ex[i] = ex[i-1] * ex[1];
}

static void Evolve(int iter, final complex [*,*,*] v,
    final complex [*,*,*] w) {
    final [int*3] d = v.getSize();
    iter++;
    for (i := 0; i < d; i++)
        w[i] = (v[i] * ex[iter*(indexmap[i])];
}

static void CheckSum(final int iter, final complex [*,*,*] u) {
    final [int*3] dim = u.getSize();
    int q, r, s;
    complex chk = complex(0,0,0);
    for (k := 1:1025) {
        q = k%dim[0];
        r = (3*k)%dim[1];
        s = (5*k)%dim[2];
        chk += u[q,r,s];
    }
    chk /= TotalDim;
    sums[iter] = chk;
}

static boolean Verify() {
    double err;
    final double epsilon = 1.0E-12;
    boolean verified = true;
    for (i := MaxIterations) {
        err = Math.abs((Complex.real(sums[i]) -
            Complex.real(TestValue[i])) /
            Complex.real(TestValue[i]));
        if (err >= epsilon)
            verified = false;
        err = Math.abs((Complex.imag(sums[i]) -
            Complex.imag(TestValue[i])) /
            Complex.imag(TestValue[i]));
        if (err >= epsilon)
            verified = false;
    }
    return verified;
}

static public void main(String [] args) {
    double timecounter = 0.0, mops = 0.0;
    final Timer timer = new Timer();
    boolean passedVerification;
    final complex [*,*,*] u = new complex [Dim[0],Dim[1],Dim[2]];
    final complex [*,*,*] v = new complex [Dim[0],Dim[1],Dim[2]];
    final complex [*,*,*] w = u;
final complex [*,*,*] x = new complex [Dim[0],Dim[1],Dim[2]];

System.out.println("NAS Parallel Benchmarks SPAR version "+"- FT Benchmark\n");
System.out.println(" Size: "+Dim[0]+"x"+Dim[1]+"x"+Dim[2]+" (class "+Class+")");
System.out.println(" Iterations: "+MaxIterations);

// Calculate MaxDim
MaxDim = Dim[0];
if (Dim[1]>MaxDim)  MaxDim = Dim[1];
if (Dim[2]>MaxDim)  MaxDim = Dim[2];

// initialize temporary FFT vectors
temp1 = new complex [MaxDim];
temp2 = new complex [MaxDim];
timer.Clear();
timer.Start();
ComputeIndexMap();
ComputeInitialConditions(u);
FFT_Init(MaxDim);
FFT3D(1,u,v);

for (iter :- :MaxIterations) {
    System.out.println(" iteration ” +(iter+1));
    Evolve(iter,y,w);
    FFT3D(-1,w,x);
    CheckSum(iter,x);
}
timer.Stop();
timecounter = timer.Read();
passedVerification = Verify();

if (timecounter>0.0)
    mops = 1.0E-6*TotalDim * (14.8157+7.19641*Math.log((double)TotalDim) + 5.23518+7.21113*Math.log((double)TotalDim)
          ) * MaxIterations/timecounter;

Results.Print(Class,Dim[0],Dim[1],Dim[2],MaxIterations,
              timecounter,mops,"floating point",passedVerification);
A.5 IS.spar

```java
import ClassData;
import Random;
import Results;
import Timer;

public class IS {
    static final char Class = ClassData.Class;
    static final int NumKeysLog2 = ClassData.TotalKeysLog2;
    static final int MaxKeyLog2 = ClassData.MaxKeyLog2;
    static final int NumKeys = (1 << NumKeysLog2);
    static final int MaxKey = (1 << MaxKeyLog2);
    static final int SizeOfBuffers = NumKeys;
    static final int MaxIterations = ClassData.MaxIterations;
    static final int TestArraySize = ClassData.TestArraySize;

    static final int [] key = new int[NumKeys];
    static final int [] keyRank = new int[NumKeys];
    static final int [] keyCount = new int[MaxKey+1];

    static final int [] testIndex = ClassData.TestIndex;
    static final int [] testRank = ClassData.TestRank;
    static final int [] testSign = ClassData.TestSign;

    static boolean FullVerify() {
        int incorrectKeys = 0;
        final int [] keySorted = new int[NumKeys];

        // Finally sort the keys
        foreach (i :- :NumKeys)
            keySorted[keyRank[i]] = key[i];

        // Confirm the correct sorting of keySorted
        // count incorrectly sorted keys, if any
        foreach (i :- :NumKeys-1)
            if (keySorted[i]>keySorted[i+1])
                incorrectKeys++;

        if (incorrectKeys!=0)
            System.out.println("Full_verify: number of keys out of sort: "+
                incorrectKeys);

        return (incorrectKeys==0);
    }

    static void CreateSeq(final double seed, final double a) {
        final double G = MaxKey/4;
        double x;
        final int BatchSize = 64;
        final int NBatches = NumKeys/BatchSize;

        foreach (k :- :NBatches) {
            Random random = new Random(seed,a);
```
random.Skip((double)4*k+BatchSize);
for (i :- BatchSize) {
    x = random.RandLC();
    x += random.RandLC();
    x += random.RandLC();
    x += random.RandLC();
    key[k+BatchSize+i] = (int) (G*x);
}
}

static boolean Rank(final int iteration) {
    int passedVerificationCount = 0;
    // Set key counters to 0
    foreach (i :- MaxKey+1)
        keyCount[i] = 0;
    // Count the number of occurrences of a value
    foreach (i :- NumKeys)
        keyCount[key[i]]++;
    // Determine the amount less or equal to a value
    for (value :- MaxKey)
        keyCount[value+1] += keyCount[value];
    // Determine the rank of each key
    foreach (i :- NumKeys)
        keyRank[i] = -keyCount[key[i]];
    // Perform the partial verification test
    passedVerificationCount = TestArraySize;
    foreach (i :- TestArraySize) {
        if (keyRank[testIndex[i]] != testRank[i] + testSign[i]*iteration) {
            System.out.println("Failed partial verification:" +
            " iteration " + iteration + ", test key "+i);
            passedVerificationCount--;
        }
    }
    return (passedVerificationCount==TestArraySize);
}

static public void main(String [] args) {
    final double GenerationSeed = 314159265.0;
    final double GenerationMult = 1220703125.0;
    int passedVerificationCount = 0;
    double timecounter, mops;
    final Timer timer = new Timer();
    System.out.println("NAS Parallel Benchmarks SPAR version "+
    "- IS Benchmark\n");
    System.out.println(" Size: " +NumKeys+" (class "+Class+")");
System.out.println(" Iterations: "+MaxIterations);

timer.Clear();

// Generate random keys
CreateSeq(GenerationSeed,GenerationMult);
timer.Start();

for (iteration :- 1:MaxIterations+1) {
    key[iteration] = iteration;
    key[iteration+MaxIterations] = MaxKey - iteration;
    if (Rank(iteration))
        passedVerificationCount++;
}

timer.Stop();

// This tests that keys are in sequence: sorting of last
// ranked key seq occurs here, but is an untimed operation
if (FullVerify())
    passedVerificationCount++;

timecounter = timer.Read();
mops = ((double) (MaxIterations*NumKeys))/timecounter/1000000.0;
Results.Print(Class,NumKeys,0.0,MaxIterations,timecounter,mops,
    "keys ranked",passedVerificationCount==MaxIterations+1);
Appendix B

Source codes of other algorithms

B.1 DoFFT.spar

```java
public class DoFFT {
    static int NumberOfIterations = 4;
    static boolean GivePrintout = true;

    public static void main(String [] args)
    {
        int N;
        complex [] x;
        FFT.fft = new FFT(); // <$ on p replicated $>

        N = 1024*128; // Must be a power of 2
        N=16;

        x = new complex[N]; // <$ on p cyclic(blocksize=1) $>

        foreach (i :- :N)
            x[i] = complex(i,(double)i/N);

        if (GivePrintout) {
            System.out.println("X: ");
            for (i :- :N)
                System.out.println(x[i]);
        }

        fft.SetVector(x);
        fft.Forward();

        if (GivePrintout) {
            x = fft.GiveResult();
            System.out.println("FFT of X: ");
            for (i :- :N)
                System.out.println(x[i]);
        }

        fft.Inverse();
        for (i :- NumberOfIterations) {
```
APPENDIX B. SOURCE CODES OF OTHER ALGORITHMS

```java
37     fft.Forward();
38     fft.Inverse();
39
40    
41    if (GivePrintout) {
42        x = fft.GiveResult();
43        System.out.println("Inv FFT of FFT of X: ");
44        for (i := :N)
45            System.out.println(x[i]);
46    }
47    }
48    }
49
50    class FFT {
51        complex [] omega;
52        int N, M;
53        complex [] x;
54
55        public FFT() {
56            N = 0;
57            M = 0;
58        }
59
60        public FFT(complex [] newx) {
61            N = 0;
62            SetVector(newx);
63        }
64
65        public void SetVector(complex [] newx) {
66            int prevN;
67
68            prevN = N;
69            N = newx.length;
70
71        if (N!=prevN) {
72            x = new complex[N]; // <\$ on p cyclic(blocksize=1) >
73            CalcM();
74            InitOmega();
75        }
76
77        foreach (i := :N)
78            x[i] = newx[i];
79    }
80
81        private void CalcM() {
82            int N2 = N;
83
84            M = 0;
85        while (N2>1) {
86            N2 = N2 >> 1;
87            M++;
88        }
89    }
90
91
```

private void InitOmega() {
    int k, m = M+1;
    double t;

    omega = new complex[m]; // <\$ on each p \$>
    k = 1;
    for (i := :m) {
        t = 2.0*Math.PI/k;
        omega[i] = complex(Math.cos(t),Math.sin(t));
        k *= 2;
    }
}

public complex [] GiveResult() {
    return x;
}

public void Forward() {
    FFT_Sub(1);
}

public void Inverse() {
    FFT_Sub(-1);
}

void FFT_Sub(int is) {
    // is = forward(1)/inverse(-1)
    int pt, pM;
    int id_to, id_from;
    complex om, om1;

    for (t := 1:M+1) {
        pM = 1 << (M-t);
        pt = 1 << (t-1);

        // <\$ on p cyclic(blocksize=2*pt) \$>
        complex [] y = new complex [N];

        if (is==-1)
            om1 = Complex.conj(omega[t]);
        else
            om1 = omega[t];

        foreach (j := :N/2:pt) { // <\$ on p cyclic(blocksize=pt) \$>
            om = 1.0;
            for (k := :pt) {
                id_from = j+k;
                id_to = id_from+j;
                complex x2 = om * x[id_from+N/2];
                y[id_to] = x[id_from] + x2;
                y[pt+id_to] = x[id_from] - x2;
                om = om * om1;
            }
        }
    }
}
145     [x, y] = [y, x];
146
147     _delete(y);
148 }
149     if (is == -1)
150         foreach (i := :N)
151             x[i] /= N;
152 }
153 }
154 }
B.2 DoMergeSort.spar

    globalpragmas <$ boundscheck = false $>;

    public class DoMergeSort {
        public static void main(String[] args) {
            int[] Vertex;
            int[] Offset;
            int N;

            N = 11;  // Final size will be N*N
            Vertex = new int[N*N];  // <$ on p(block) $>

            // Make an inverse cyclic sort of the list (pseudo random)
            foreach (i := :N, j := :N)
                Vertex[i*N+j] = N*(N-i)-j
            N *= N;

            for (i := :N)
                System.out.print(Verteck[i] + " ");
            System.out.println();

            MergeSort.Sort(Vertex);

            for (i := :N)
                System.out.print(Verteck[i] + " ");
            System.out.println();
        }
    }

    class MergeSort {
        static private int ILog2(int N) {
            int M, A = N;
            M = 0;
            while (A>1) {
                A >>= 1;
                M++;
            }
            return M;
        }

        static public void Sort(int[] List) {
            int[] Buff;
            int N = List.getSize()[0];
            int M = ILog2(N-1)+1;

            Buff = new int[N];

            for (t := 1:M+1) {
                int pM = ((N-1) >> t) + 1;
                int pt = 1 << (t-1);
                foreach (j := :pM) {
                    Buff[j] = List[t-1] + pt;
                }
            }
        }
    }
```c
int i1 = j*2*pt;
int i2 = i1 + pt;
int t1 = i2;
int t2 = i2 + pt;
if (N < i1 + 2*pt) {
    t2 = N;
    if (t1>N)
        t1 = N;
}
int k = i1;
while (i1<t1 && i2<t2)
    if (List[i1]<List[i2])
        Buff[k++] = List[i1++];
    else
        Buff[k++] = List[i2++];
    while (i1<t1)
        Buff[k++] = List[i1++];
    while (i2<t2)
        Buff[k++] = List[i2++];
    [List,Buff] = [Buff,List];
}
if ((M&1)==1) { // M is odd
    [List,Buff] = [Buff,List];
    foreach(i := N)
        List[i] = Buff[i];
}
~delete(Buff); // Until garbagecollection is available
```
B.3 Poisson_spar

```java
1  globalpragmas <$ boundscheck = false $>;
2
3 public class Poisson {
4    static double [*,*] phi1;
5    static double [*,*] phi2;
6    static tuple int^2 dim = [70,70];
7    static double precision_goal = 0.0001;
8    static int max_iter = 5000;
9    static int no_sources;
10    static tuple int^2 [] source_pos;
11    static double [] source_val;
12
13    public static void main(String [] args) {
14        Setup_Grid();
15        Solve();
16    }
17
18    static void Setup_Grid() {
19        tuple int^2 v;
20
21        phi1 = new double [dim[0],dim[1]];
22        phi2 = new double [dim[0],dim[1]];
23        foreach (w :- :dim) {
24            phi1@w = 0.0;
25            phi2@w = 0.0;
26        }
27        source_pos = new tuple int^2 [3];
28        source_val = new double [3];
29        no_sources = 0;
30        v = [(int) ((double)0.35*dim[0]),(int) ((double)0.70*dim[1])];
31        source_pos[no_sources++] = v;
32        v = [(int) ((double)0.625*dim[0]),(int) ((double)0.75*dim[1])];
33        source_pos[no_sources++] = v;
34        v = [(int) ((double)0.375*dim[0]),(int) ((double)0.25*dim[1])];
35        source_pos[no_sources++] = v;
36        v = [(int) ((double)0.375*dim[0]),(int) ((double)0.25*dim[1])];
37        source_pos[no_sources++] = v;
38        source_pos[no_sources++] = v;
39
40        for (i :- :no_sources) {
41            phi1@source_pos[i] = source_val[i];
42            phi2@source_pos[i] = source_val[i];
43        }
44    }
45
46    static double Do_Step(int parity, double[*,*] phi1, double[*,*] phi2) {
47        double max_err = 0.0;
48
49        foreach (x :- 1:dim[0]-1)
50            foreach (y :- 1+(x+parity+1)%2:dim[0]-1:2) {
51                phi1[x,y] = phi2[x,y];
52                phi2[x,y] = (phi1[x+1,y] + phi1[x-1,y]
+ phi1[x,y+1] + phi1[x,y-1])
* 0.25;
}

for (i := no_sources)
    phi2(source_pos[i]) = source.val[i];

foreach (x := 1:dim[0]-1)
    foreach (y := 1+(x+parity+1)%2:dim[0]-1:2)
        max_err = Math.max(max_err, phi2[x,y]-phi1[x,y]);

return max_err;
}

static void Solve() {
    int count = 0;
    double delta;
    double delta1, delta2;
    delta = 2*precision_goal;
    while (delta > precision_goal && count < max_iter) {
        delta1 = Do_Solve(0,phi1,phi2); // red
        delta2 = Do_Solve(1,phi2,phi1); // black
        delta = Math.max(delta1,delta2);
        count++;
    }
    System.out.println("Number of iterations : "+ count);
}
Appendix C

Multi Dimension Divider

For problems like the Poisson solver for a 2D grid one usually distributes the multi-dimensional array in both dimensions. By doing this, one minimizes the amount of data that needs to be communicated to another processor. When the amount of processors that is used is a square, the creation of a 2D processor grid to map the grid on is rather straightforward. For instance with 16 processors and a $4 \times 4$ processor grid where each processor gets appointed a grid block of size $25 \times 25$. For an amount of processors that is not a square, the creation of a processor grid becomes less trivial. For instance with 8 processors we would choose a $4 \times 2$ processors grid and with 20 processors we would choose $5 \times 4$. But with 5 processors we are forced to use a $5 \times 1$ grid, because 5 is a prime number. Especially when Spar starts supporting a `NumberOfProcessors()` intrinsic so that the amount of processors can be specified at load time, there will be a need for an algorithm to divide the number of processors into $n$ factors for creating a $n$D processor grid, where the size in each dimension deviates as few as possible from the other dimensions in order to have a minimal amount of communication. The following Spar code provides a `PDiv` method that does this division of $N$ processors in $D$ factors. The dimension sizes are returned in an integer array of size $D$. The `main` method in this code prints the results for processors amounts $1 \ldots 100$ and dimensions $1 \ldots 6$ as an example of the usage of `PDiv`. The `ISqrt` method is used by `PDiv` and calculates an integer version of $a^{1/p}$.

```java
1 // Program to show how to divide any number
2 // of processors among N dimensions
3
4 public class PDivide {
5     static int ISqrt(int a, int p) {
6         int s, t, e;
7         s = 0;
8         e = 1;
9         for (int n = a >> p; n > 0; n >>= p)
10             e >>= 1;
11         while (e > 0) {
12             s += e;
13             t = s;
14             for (int k=1;k<p;k++)
15                 t *= s;
16             if (t>a)
17                 s= e;
18             e >>= 1;
19         }
20         return s;
21     }
22 }
```

106
static int[] PDiv(int N, int D) {
    int s;
    int[] a = new int[D];
    for (i := D−1) {
        s = ISqrt(N, D−i);
        while (N/s != 0) {
            s−=;
            a[i] = s;
            N /= s;
        }
    }
    a[D−1] = N;
    // sort array a
    for (i := D−1) {
        int k = i;
        for (j := i+1:D)
            if (a[j] < a[k])
                k = j;
        if (k!=i) {
            int swap = a[i];
            a[i] = a[k];
            a[k] = swap;
        }
    }
    return a;
}

public static void main(String [] args) {
    int[] a;
    for (i := 1:6)
        for (j := 1:100)
            a = PDiv(i, j);
        System.out.print(i+" -> ");
    for (k := j−1)
        System.out.print(a[k] +", " );
    System.out.println(a[j-1] + "\" ");
}