Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

# Compiling the parallel programming language NestStep to the CELL processor

by

**Magnus Holm**

LIU-IDA/LITH-EX-A--10/027--SE

2010-05-26

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköpings universitet
Institutionen för datavetenskap

Examensarbete

# Compiling the parallel programming language NestStep to the CELL processor

**av**

**Magnus Holm**

LIU-IDA/LITH-EX-A--10/027--SE

2010-05-26

Handledare: Christoph Kessler

Examinator: Christoph Kessler

# Abstract

The goal of this project is to create a source-to-source compiler which will translate NestStep code to C code. The compiler's job is to replace NestStep constructs with a series of function calls to the NestStep runtime system. NestStep is a parallel programming language extension based on the BSP model. It adds constructs for parallel programming on top of an imperative programming language. For this project, only constructs extending the C language are relevant. The output code will compile to form an executable program that runs on the multicore processor Cell Broadband Engine (Cell BE). The NestStep runtime system has been ported to the Cell BE and is available from start of this project.

# Acknowledgments

I would like to thank my examiner and supervisor Christoph Kessler for his time and advice he has given me during this project. Also I would like to thank Daniel Johansson for answering a question on the NestStep Runtime System he ported to the Cell BE as part of his final thesis work.

# Contents

**1 Introduction** **1**
1.1 Project Description . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Project Approach . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.3 Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Thesis Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 Cell Processor** **5**
2.1 PowerPC Processor Element . . . . . . . . . . . . . . . . . . . . . 5
2.2 Synergistic Processor Element . . . . . . . . . . . . . . . . . . . . 5
2.3 Memory access . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**3 NestStep Overview** **7**
3.1 Nested Supersteps . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.2 Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.3 Declarations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.3.1 Replicated shared data . . . . . . . . . . . . . . . . . . . . 10
3.3.2 Distributed shared data . . . . . . . . . . . . . . . . . . . 10
3.3.3 Private data . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.4 Step statement and Combine . . . . . . . . . . . . . . . . . . . . . 12
3.5 Mirror and Update . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.6 Forall statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
3.7 Seq statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**4 Compiler Building Base** **17**
4.1 Cetus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.1.1 ANTLR - A Parser Generator . . . . . . . . . . . . . . . . . 18
4.1.2 ANTLR Grammar Syntax . . . . . . . . . . . . . . . . . . . 20
4.1.3 Internal Representation . . . . . . . . . . . . . . . . . . . . 21
4.2 Cell-NestStep-C Runtime System . . . . . . . . . . . . . . . . . . . 24
4.2.1 The Executable . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.2.2 PPE tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.2.3 Main memory storage . . . 25
4.2.4 Memory manager . . . 25
4.2.5 Data structures . . . 25
4.2.6 Buffer transfers with DMA . . . 26

**5 Compiler Implementation** **27**
5.1 Cetus Extension . . . 27
5.1.1 Grammar and Internal Representation . . . 27
5.1.2 Group size symbol problem . . . 30
5.1.3 Transformation Passes . . . 31
5.1.4 Output from the Compiler . . . 33
5.2 Cell-NestStep-C Runtime System Extension . . . 33
5.2.1 Interface . . . 34
5.2.2 Inner workings . . . 35

**6 Evaluation** **37**
6.1 Test programs . . . 37
6.2 Overhead from the Runtime System . . . 41
6.3 Future extensions . . . 43

**Bibliography** **45**

**A Glossary** **47**
A.1 Words and Abbreviations . . . 47

**B Code for Test Programs** **49**
B.1 Pi . . . 49
B.2 Dot Product . . . 51
B.3 Prefix Sum . . . 54
B.4 Jacobi . . . 56

# Chapter 1

# Introduction

## 1.1 Project Description

The goal of this project is to create a source-to-source compiler which will translate NestStep code to C code, in which NestStep constructs have been replaced by a series of function calls to the NestStep runtime system. NestStep is a parallel programming language extension based on the BSP model. It adds constructs for parallel programming on top of an imperative programming language. For this project, only constructs extending the C language are relevant.

The compiler will output C code based on inputed NestStep code. The code will compile to form an executable program that runs on the Cell BE processor (Cell Broadband Engine). The NestStep runtime system has been ported to the Cell BE, as part of previous work [10] and is available from start of this project. The ported runtime system is referred to as the Cell-NestStep-C runtime system. This is the runtime system mentioned in this text.

The goal is not to create a source-to-source compiler from scratch, but to use an existing compiler framework to build upon.

## 1.2 Project Approach

An existing source-to-source compiler framework is chosen to be extended to form the new compiler. One could write a compiler from scratch but that would demand to much time in first creating a compiler that understands the C language followed by making it understand the NestStep extension. To save time, we needed to chose a source-to-source compiler framework that already had language support for the C language.

Cetus [8] was chosen to be the source-to-source compiler framework for this project. Cetus without any modifications will translate C to C. With this frame-

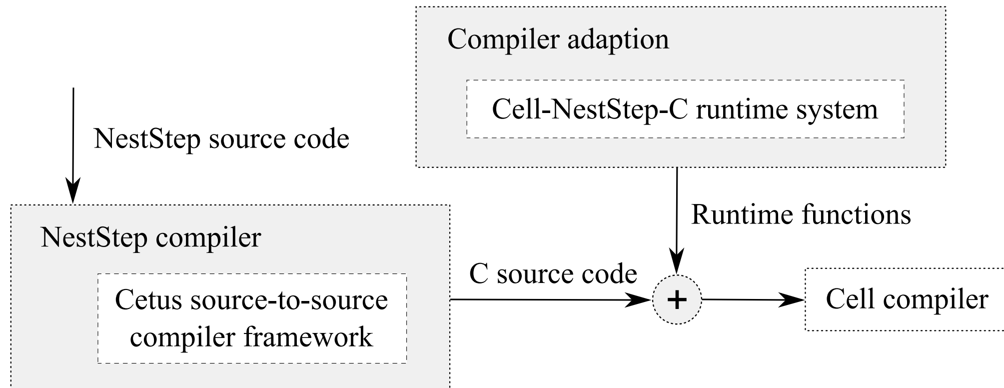



Figure 1.1: Compiler Overview. The grey marked areas correspond to the main implementation objectives.

work as base, support for NestStep constructs will be added on top of the C language support.

With Cetus comes a parser for C. It is generated by ANTLR, a Java-based generator. The source code for Cetus includes a file containing a grammar for the C language. This file can be modified to also include definitions of NestStep constructs.

There were some important issues, about Cetus that were necessary to learn before starting the implementation in order to understand how and where to add things. For instance:

- How is the intermediate representation (IR) structured and traversed?
- How is the code printed?
- What is the symbol table structure and how can it be used to store information about shared variables?

A bit into the project it was realized that the functionality of the Cell-NestStep-C runtime system needs buffers to store data and the buffers must be declared and allocated in the code using the runtime system, i.e. the code that the compiler generates. There would be a lot of code of the same kind generated and a decision was made to move the common code about buffering out from the compiler and place it in a kind of extension or adapter to the runtime system. The runtime system was extended without any modification to the original code, entirely on top of it, just adding new functionality calling existing functionality. Figure 1.1 illustrates a compiler overview with the main implementation objectives.

This project is not focused on how to program the Cell processor in detail. The details of the Cell processor is not highly relevant, since the details are addressed

by the Cell-NestStep-C runtime system. Testing of compiled Cell applications was done on the Sony PlayStation 3 and the IBM Cell Simulator running on Linux Fedora 7.

## 1.3 Objectives

The following are the two main objectives of this project. See figure 1.1 to get a better picture.

- Extending Cetus to form the NestStep source-to-source compiler.
- Create the compiler adaption of the Cell-NestStep-C runtime system.

## 1.4 Thesis Structure

This is a short presentation of the chapters of this report and their contents:

- **Chapter 2:** Cell Processor. This chapter presents the Cell processor and its main internal parts.
- **Chapter 3:** NestStep Overview. This chapter presents the NestStep language and how it is supported by the compiler.
- **Chapter 4:** Compiler Building Base. This chapter presents Cetus as the source-to-source compiler framework which the compiler implementation is building upon. It also presents the Cell-NestStep-C runtime system.
- **Chapter 5:** Compiler Implementation.
- **Chapter 6:** Evaluation.

# Chapter 2

# Cell Processor

The Cell BroadBand Engine (Cell BE) is a multiprocessor with nine processors on a single chip. There are two types of processors on the chip making it a heterogeneous multicore processor. Cell consist of one master PPE processor and eight slave SPE processors. They are all connected to each other and to other external devices by a bus, called the Element Interconnect Bus (EIB), with high bandwidth. The information of this chapter is based on [5].

Software development in C/C++ language is supported by language extensions. There are a Linux-based SDK (Software Development Kit), a full-system simulator and a rich set of application libraries, performance tools and debug tools [5].

The processor is a result of a collaboration between IBM, Sony and Toshiba. The processor is part of the hardware of Sony PlayStation 3 and IBM BladeCenter QS20, QS21 and QS22.

## 2.1 PowerPC Processor Element

The PowerPC Processor Element (PPE) is a 64-bit PowerPC core and can run both 32-bit and 64-bit operating systems and applications. It is the main processor. It controls processing, including the allocation and management of SPE threads.

## 2.2 Synergistic Processor Element

The Synergistic Processor Element (SPE) is a 128-bit RISC processor for SIMD applications. It consists of two main units; the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC). The SPU fetches and runs program instructions. It has 256 KB private local store (LS) memory that is software-controlled and used to store both program instructions and data. The MFC maintains and

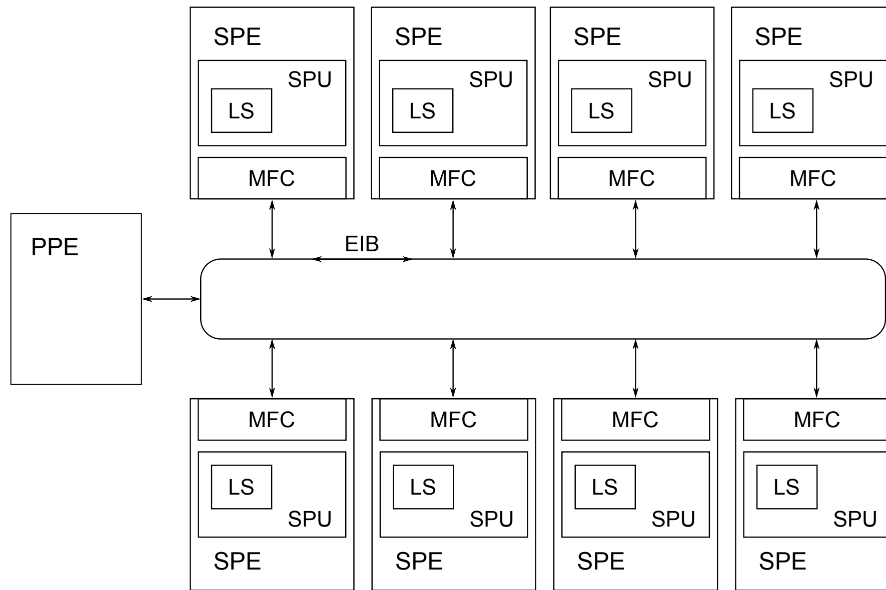



Figure 2.1: Cell Broadband Engine architecture

processes queues of DMA commands from the SPU. The eight SPEs are intended to run compute-intensive applications allocated to them by the PPE. Each SPE processor can run a different program at the same time as, and independently from, the other SPE processors.

## 2.3 Memory access

There is a difference between the PPE and the SPEs in how they access memory. The main memory is included in the effective-address space of the PPE while the SPE application accesses main memory through direct memory access (DMA) commands. The SPU will process data from its private local store memory. The MFC will on request copy memory back and forth between local store memory and main memory. The DMA transfers are asynchronous, which means that computation can continue during transfers. With double buffering it is possible to hide transfer time with computations operating on previously transfered data. It could be hidden completely if computation of a block takes longer than transferring it.

# Chapter 3

# NestStep Overview

NestStep [7] is a parallel programming language, i.e. a language for writing programs that are capable of using more than one processor and executing in parallel. It is constructed to be an extension for imperative programming languages. For this project we are concerned with the extension for the imperative C language. Several language additions are part of the parallel extension, such as supersteps (step construct), declaration of shared data, combining of data, sequential execution (seq construct), grouping of processors (neststep construct) and symbols. Details on these constructs are presented in sections below.

A NestStep program is of SPMD type (single program multiple data). It means that one instance of the same program will run on each processor. The instances will communicate results and data with each other. Originally NestStep programs were developed and implemented on top of MPI to run on processors connected in a cluster. Since the porting of the runtime system to the Cell BE, the communication is sent through the very fast on-chip EIB bus instead of a cluster network.

NestStep inherits properties from the BSP (bulk-synchronous parallel) programming model [7]. NestStep has a shared memory abstraction on distributed memory systems [15]. Programs are divided up in subsequent supersteps, each superstep separated from the other by a global barrier synchronization point. A superstep executes in the following order: first a computation stage followed by a communication stage ended by the global synchronization barrier. See figure 3.1.

During the computation stage, computations take place on every participating processor with no communication between the processors, processing data stored in local memory of the processor. Only data locally available can be accessed, like copies of replicated shared data (section 3.3.1), owned distributed shared data (section 3.3.2) and mirrored distributed shared data (section 3.5). This is according to the BSP programming model.

Distributed data wanted by one processor and owned by another, has to be requested and then transfered (mirrored) during the communication stage so that

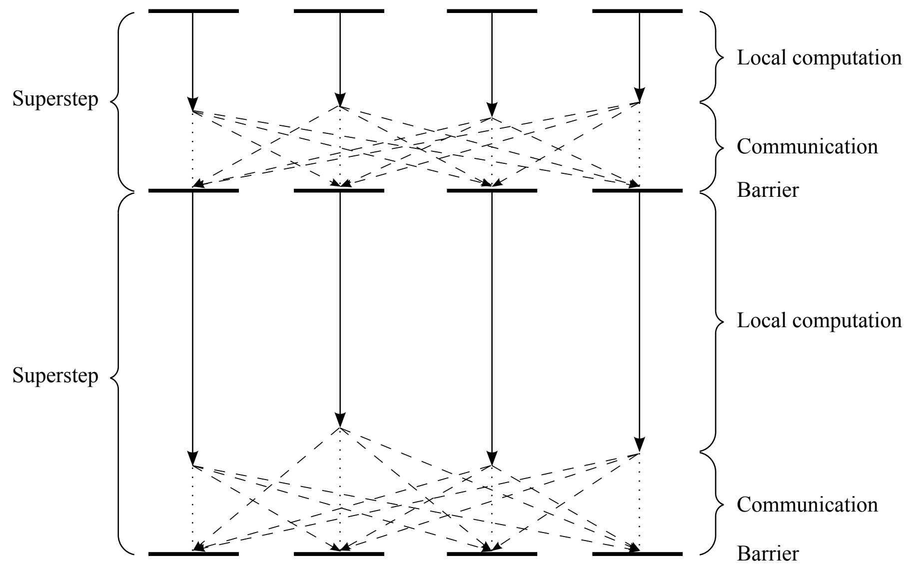



Figure 3.1: BSP execution flow. Idea of illustration from [14].

it is available for use in the following superstep. The combine phase is a part of the communication stage and has the purpose of restoring consistency of replicated shared data at the end of supersteps. In between supersteps all replicated data are consistent between all involved processors. Data is combined using some combine strategy (section 3.4).

A barrier synchronization is a point where a processor waits until all other processors have finished their communications. Barriers can be considered costly, if workload is unevenly distributed, since the execution of the program won't continue until all processors are done with communication.

This chapter will list features of NestStep supported by the compiler. The compiler does not support all features of NestStep: the group dividing feature is missing because that feature is missing from the Cell-NestStep-C runtime system.

## 3.1 Nested Supersteps

Participating processors of a NestStep program are organized into groups. From start all processors belong to a root group. A group can be split dynamically at runtime into subgroups. Each subgroup can have a different number of processors belonging to it. This feature is called nesting of supersteps and the construct is named neststep. The Cell-NestStep-C runtime system lacks support for nesting of supersteps[10]. Consequently, the compiler does not support it either. More information on nesting of supersteps can be found in [11].

## 3.2 Symbols

When programming parallel SPMD programs, to divide the work we need to know how many pieces the work should be divided in (limited by the number of processors). This number is called the group size and is represented in NestStep with the symbol `#`.

The programmer may need to identify the processors when dividing the work. Each processor knows which work to do because of a unique identifier assigned to it. It is called the processor rank and is represented in NestStep with the symbol `$`.

There is a symbol `@` for group identification. As mentioned in section 3.1 there is no support for dividing of groups. There is only a root group. The support for the symbol `@` is therefore omitted.

## 3.3 Declarations

NestStep supports two ways of sharing data, replication (section 3.3.1) and distribution (section 3.3.2). Replication is done with shared variables and replicated shared arrays. Distribution is done with two kinds of distributed arrays; block distributed array and cyclically distributed array.

The Cell-NestStep-C runtime system introduces a special way of storing private data; private variables and private arrays (section 3.3.3). The reason for this is mentioned in section 4.2.5.

Pointers can be declared as shared, but it is the data it points to that is shared. The pointer variable itself is always private. Shared pointers are limited. They may point to shared variables and whole arrays (not elements inside it). Pointer arithmetics is not allowed. A pointer to replicated shared data $x$ must have been declared with the same type and combine strategy as is used at the declaration of $x$ [11].

There is a way to dynamically allocate and deallocate replicated and distributed shared data. Shared data structures should be available from all processors. The dynamic allocation/deallocation should be in a place in the code where all processors for sure will run through. For instance, one should not place it inside a seq statement (section 3.7) because a seq statement is only executed by one processor.

The Cell-NestStep-C runtime system does only support three primitive datatypes together with its data structures. Those are `int`, `float` and `double`.

A limit has been set to the number of dimensions an array can have. The limit is three dimensions.

### 3.3.1 Replicated shared data

Each participating processor has one local copy of replicated shared data. Inside a superstep different copies may differ in value, but at the end of supersteps replicated data must be combined using some combine strategy, which means restoring consistency. Only replicated data that has been changed need to be combined.

The default combine strategy for a replicated variable can, as an option, be supplied in the variable declaration. This is also true for the prefix sum variable. See combine strategies in section 3.4.

```
// shared variable declaration
sh int a;
// replicated array declaration
sh int b[100];
// pointer to replicated data
sh int *c;
// combine strategy <+> added
sh<+> int d;
// combine strategy <+> with prefix sum variable added
pb int prefix;    // private variable, see section 3.3.3.
sh<+:prefix> int e;

// point to a shared variable
c = &a;
// dynamic allocation of replicated array (100 integers)
c = new_RepArray(100, Type_int);
free_RepArray(c);
```

### 3.3.2 Distributed shared data

Two types of distributed shared arrays are available; block distributed array and cyclically distributed array. Elements within the array are distributed among the participating processors in different ways depending on the type of array. See figure 3.2 for an illustration of how elements are blockwise and cyclically distributed. The elements are then owned by that processor and can be accessed by other processors through mirror requests (section 3.5). The BSP model suggests that modified values of elements of such an array become visible at the end of a superstep [12]. The processor can, because of this, work on its local copy until the end of the superstep.

```
// block distributed shared array declaration
sh int a[1000]</>;
// cyclic distributed array declaration (1000 integers, blocks of 50)
sh int b[20]<%>[50];
```

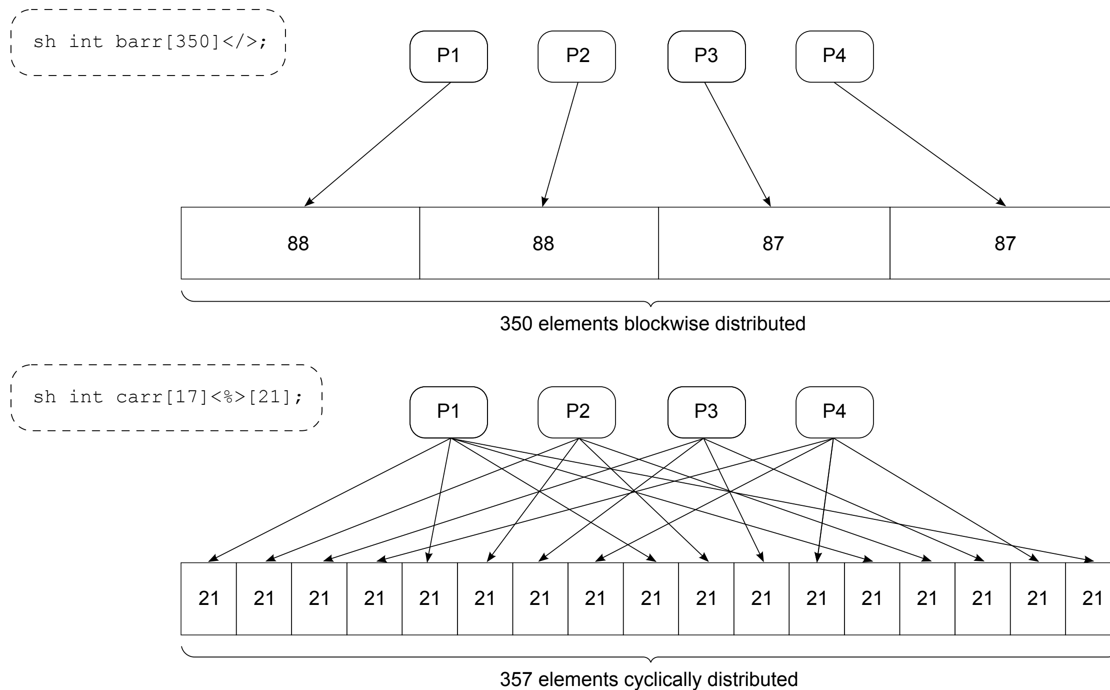



Figure 3.2: Block and cyclic element distribution with four processors

```
// pointer to a block distributed shared array
sh int *c</>;
// pointer to a cyclic distributed shared array
sh int *d<%>;

// point to a block distributed shared array
c = a;
// point to a cyclic distributed shared array
d = b;
// dynamic allocation of block distributed shared array
c = new_BlockArray(1000, Type_int);
free_BlockArray(c);
// dynamic allocation of cyclic distributed shared array (1000
    integers, blocks of 50)
d = new_CyclicArray(20, 50, Type_int);
free_CyclicArray(d);
```

### 3.3.3 Private data

The private type qualifier will store local data for each SPE in main memory. There is not enough space in local store memory for storing large amounts of private data.

One extra specifier, `pb`, has been introduced to mark which data is private. The Cell-NestStep-C runtime system requires the use of private data structures in situations when the mirror/update constructs (section 3.5) are used and when a combine operation includes calculation of the prefix sum (section 3.4). The specifier is special and used only with NestStep programs written for the Cell processor. When compiling for other targets, it could be ignored.

```
// private variable declaration
pb int a;
// private array declaration
pb int b[100];
// pointer to private data
pb int *c;

// point to a private variable
c = &a;
// dynamic allocation of private array (100 integers)
c = new_LocalArray(100, Type_int);
free_LocalArray(c);
```

## 3.4 Step statement and Combine

The step statement denotes a superstep. Optionally, a combine construct is placed at the end of the statement. If left out, the default combine strategy applies. Available strategies are listed in table 3.1. The following code is an example of a step statement with declaration of combine strategies.

```
sh int a, b;
pb int c;
sh float d[10];
sh<0> float e;
step {
  ...
  /* a is combined with addition, b is combined with multiplication
      with prefix sum stored to c, d is combined with MAX (elements in
       range only), e is combined with leader's value. */
}
combine(a<+>, b<*:c>, d[2:5]<MAX>);
```

Combining is performed on replicated data at the end of supersteps. The combine strategy can be declared either at the declaration of the replicated shared data or at the end of the step construct. The strategy can be omitted and in such case the default strategy "arbitrary" is used (see table 3.1).

A range can optionally be applied as part of the strategy for replicated arrays. Ranges for one dimension are currently supported by the runtime system.

| | |
|---|---|
| <0> | The leader's value (rank 0 of the group) is broadcasted. |
| <?> | An arbitrary updated copy is chosen and broadcasted (default strategy). |
| <=> | No combining is performed. Responsibility rests with the programmer to ensure that all local copies are equal. |
| <+> | Local copies are added together and the sum is broadcasted. |
| <*> | Local copies are multiplied together and the product is broadcasted. |
| <AND> | Similar to <+> but using bitwise AND instead. |
| <OR> | Similar to <+> but using bitwise OR instead. |
| <MAX> | The maximum value between the local copies are broadcasted. |
| <MIN> | Similar to <MAX> but using minimum value instead. |
| <**foo**> | User defined method (No runtime system support). |

Table 3.1: Combine strategies

Prefix sum calculation is an optional part of combining. It is denoted by `<+:var>`, where `var` is a private variable or array (section 3.3.3), depending on if the combined replicated shared data (section 3.3.1) is a variable or array, where the prefix sum result is stored after the combining. With c as the prefix variable, b as the shared variable and i as the rank of the processor, p as the number of participating processors, prefix sum calculation is defined as

$$c_i = \sum_{j=0}^{i-1} b_j, \forall i \in \{0, ..., p-1\}$$

Similarly, prefix calculations exist for other predefined operators such as `<*:var>`, `<MIN:var>` and `<MAX:var>`. They do not exist for combine strategies `?` and `0`. User defined method as combining strategy has no support from the runtime system. How combining is performed by the runtime system running on Cell is described in [10]. How combining can be performed when a NestStep program is running on a cluster computer is described in [16].

## 3.5 Mirror and Update

The BSP model specifies that processors can access data within the computation phase of a superstep as long as it is locally available. To access distributed data owned by another processor, the data has to be requested and transfered as part of the communication stage of the previous superstep to be available in the current one. The `mirror` and `update` constructs register for a transfer to be performed at the end of the superstep. The programmer should make several requests if the data interval belong to more than one processor.

```
sh int barr[800]</>; // distribute data in 8 pieces, assuming 8 SPEs
pb int buff1[100], buff2[50];
step {
  if ($ == 1) { // processor 1 condition
    // requesting elements 0 to 99 from processor 0
    mirror(barr, buff1, 0, 99);
  }
} // data is copied from barr to buff1 at the end of the superstep
step {
  if ($ == 1) {
    // --- use mirrored data ---
    // write array elements 50 to 99 to processor 0
    update(barr, buff2, 50, 99);
  }
} // data is copied from buff2 to barr at the end of the superstep
```

## 3.6 Forall statement

By the forall statement, all elements of a distributed shared array are iterated through. Each processor is only concerned with iterating through its owned elements.

```
sh int w1[21]</>;
sh int w2[21][21]</>;
sh int w3[21][21][21]</>;
int i, j, k;

step {
  forall(i, w1) {
    foo(w1[i]);
  }
  forall2(i, j, w2) {
    foo(w2[i][j]);
  }
  forall3(i, j, k, w3) {
    foo(w3[i][j][k]);
  }
}
```

## 3.7 Seq statement

The inside of the seq statement is executed only by the leader processor, i.e. the processor with rank 0 (see section 3.2).

```
seq {
  // processor 0 enters, other processors move forward
}
step seq {
  // processor 0 enters, other processors wait
}
```

# Chapter 4

# Compiler Building Base

Cetus is the source-to-source compiler framework that was chosen for this project to be the base on which the implementation of the compiler to build upon. Cetus without any modifications will translate C to C. With this framework as base, support for NestStep constructs will be added on top of the support for the C language. The end result will be an extended Cetus, a source-to-source compiler that translates from NestStep-C to C with function calls to the Cell-NestStep-C runtime system. This chapter describes the Cetus source-to-source compiler (section 4.1) and the Cell-NestStep-C runtime system (section 4.2).

## 4.1 Cetus

Cetus is a compiler infrastructure for the source-to-source transformation of programs. Cetus was created because there was a need for a compiler research environment that facilitates the development of interprocedural analysis and parallelization techniques for C, C++ and Java programs [13].

Cetus was originally created by graduate students as part of an advanced compiler project course at Purdue University [9]. Cetus can be downloaded at the Cetus project website [8]. Documentation like tutorials, manuals and a number of papers concerning Cetus are available at the project website. The Cetus API is available in Javadoc format via the website as well as bundled with the code.

The design is intended to be extensible for multiple languages [13]. Important design choices when Cetus was created were the implementation language, the parser, and the internal representation with its pass-writer interface. The implementation language for the Cetus infrastructure is Java and the ANTLR tool was selected and used as a parser generator.

### 4.1.1 ANTLR - A Parser Generator

ANTLR (ANother Tool for Language Recognition) provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions [4]. It is able to generate a parser in Java code, which was convenient for integration purposes since Java is the implementation language of Cetus. It is an LL(k) parser [13].

Cetus comes with a C language grammar written using version 2 of ANTLR [4]. There is a GUI tool, ANTLR Studio[1], that can be of assistance when creating and editing ANTLR grammar. ANTLR Studio is a plugin for the Eclipse development environment [6]. With the plugin activated you get for instance an outline of existing grammar rules, coloring of syntax and syntax diagrams. The syntax diagrams shift in color depending on where in the grammar code the editor marker is located.

The ANTLR tool will generate a number of files from a grammar file. The two most important files generated from a grammar are the Java classes for parser and lexer. They are illustrated with a UML class diagram in figure 5.2. One part of the grammar defines the parser and the other part defines the lexer.

The lexer is the scanner that reads the code from the input, character by character, to create a stream of tokens (the process is called lexical analysis). A token represents a string of characters, categorized according to the lexer rules. A token could be for instance an identifier, a comma, a number, a semicolon etc. depending on which lexer rule that matches the characters. The following code is an example extracted from the grammar file of Cetus, describing the lexer rule for an identifier. It is illustrated in figure 4.1.

```
protected ID
  options {
    testLiterals = true;
  }
  : (( 'a'..'z' | 'A'..'Z' | '_' | '$')
     ( 'a'..'z' | 'A'..'Z' | '_' | '$' | '0'..'9' )*)
  ;
```

The parser is determining the grammatical structure of the code (the process is called syntactic analysis). It is checking for correct syntax by matching the parser rules to the stream of tokens (given by the lexer). While parser rules are matched, a data structure (called a internal representation) is built to represent what is matched.

With ANTLR, the parser rules include action code written in Java. The action code is used to build the internal representation (section 4.1.3). The following code is an example extracted from the grammar file of Cetus, describing a smaller part of the parser rule for the nonterminal statement (showing the part for the if statement only).

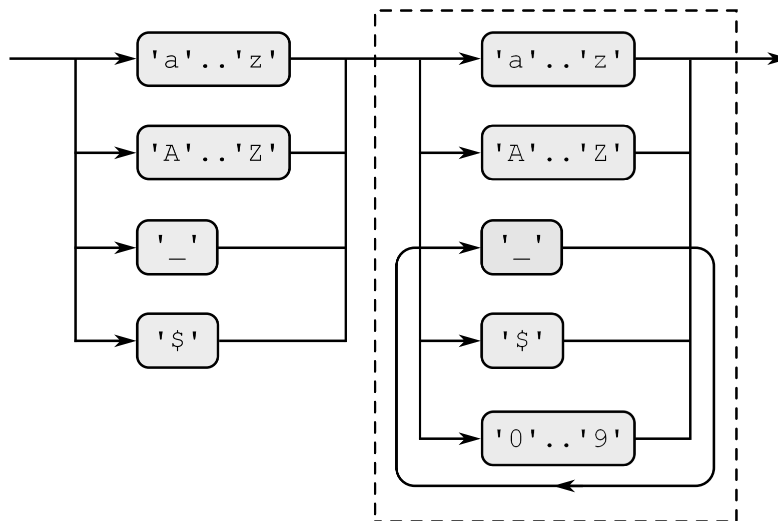



Figure 4.1: Illustration of the id lexer rule from the ANTLR C grammar. The illustration is based on the syntax diagram from ANTLRStudio[1]. See figure 4.3 for figure explanation.

```
statement returns [Statement statb]
{
  // init-action (Java-code)
  statb = null;
  Expression expr1=null, expr2=null, expr3=null;
  Statement stmt1=null, stmt2=null;
  // ...
}
: ...
| ...
| tif:"if"^ {
    // action (Java-code)
    // ...
  }
  LPAREN! expr1=expr RPAREN! stmt1=statement ("else" stmt2=
      statement )? {
    // action (Java-code)
    if(stmt2 != null)
      statb = new IfStatement(expr1,stmt1,stmt2);
    else
      statb = new IfStatement(expr1,stmt1);
    // ...
  }
| ...
;
```

The syntax is described in section 4.1.2. The rule is illustrated in figure 4.2.

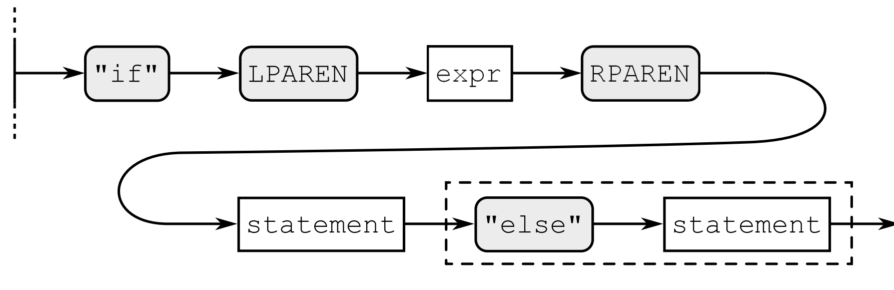



Figure 4.2: Illustration of the if statement part of the statement parser rule from the ANTLR C grammar. The illustration is based on the syntax diagram from ANTLRStudio[1]. See figure 4.3 for figure explanation.

### 4.1.2 ANTLR Grammar Syntax

The following text cover the parts of ANTLR grammar syntax that have been important for this project. Version 2 of ANTLR grammar is explained. There is no point of covering the whole ANTLR grammar definition. Grammar documentation can be found at [2] for further details. The majority of the grammar, the part that covers the C language, was available from start of the project. Because of this, there was no need to learn all the details about ANTLR grammar that would have been necessary to learn if the grammar was to be written from scratch.

The important part to cover for this project was defining of parser rules. An ANTLR rule definition corresponds to a method definition in the generated Java file, which means that the rule is able to have both parameters and return value. Both parameters and return value are passed to the generated code and must be defined using Java types and valid identifiers.

Each rule has one or more alternatives. The alternatives, in turn, reference other rules just as one function can call another function in a programming language. The basic form of an ANTLR rule is:

```
rulename
 :  alternative_1
 |  alternative_2
...
 |  alternative_n
 ;
```

Parameters are defined using the following form:

```
rulename[formal parameters] : ... ;
```

Return value is defined using using the `returns` keyword:

```
rulename returns [type returnvar] : ... ;
```

The programmer should not use a return instruction in action code. The return value of the rule is set by assigning a value to the return identifier (`returnvar` from the example above).

Actions are code blocks written in the target language (Java). It is inserted for example in conjunction with an alternative and will in such case execute when the alternative is matched. The action code will pass unchanged to the Java file when the grammar is generated. The syntax is arbitrary text surrounded by curly braces.

An init-action is an action specified before the colon and is executed before anything else in the rule. Init-actions will always be executed and will therefore serve well as a place for declaring and initializing local variables. Other actions, like the one mentioned above, will execute depending on the process of parsing the token stream, as a result of recognizing a sequence of tokens.

```
rulename
{
    // init-action code
}
 :  ...
 ;
```

ANTLR supports Extended BNF (EBNF) notation that allows optional and repeated elements. It also supports parenthesized groups of grammar elements called subrules [3]. See figure 4.3 for subrule syntax with syntax diagrams.

### 4.1.3 Internal Representation

Intermediate representation (IR) is a data structure that is constructed from the input to the compiler. The output is in turn constructed from the intermediate representation.

Cetus IR is implemented in the form of a Java class hierarchy. The data structure of the IR is a tree of traversable objects. The root is an instance of the `Program` class. A program instance contains one or more `TranslationUnit` instances (representing the files that make up the program). A translation unit contains declarations, that can for instance be an annotation, a procedure or a variable declaration. A procedure contains a compound statement which represents the procedure body. A compound statement contains both declarations and statements. There are a lot of different statements each built up by more instances, like expressions, from the IR class hierarchy tree.

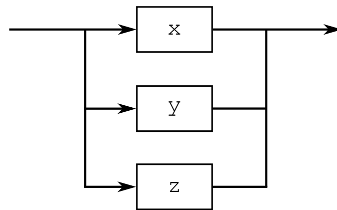
Examples of simple elements:

`LPAREN`
Match a token.

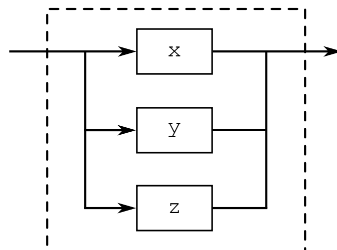
`"if"`
Match a string literal.

`statement`
Match a rule.

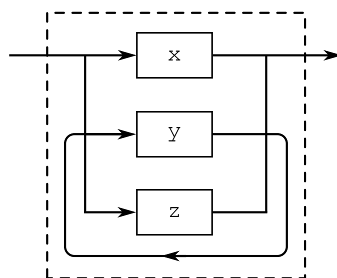
EBNF notations supported by ANTLR:



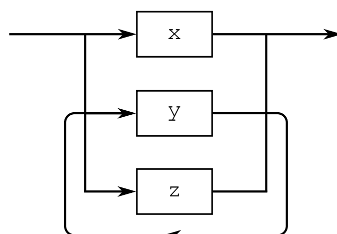

`(x|y|z)`
Match any alternative within the subrule exactly once.




`(x|y|z)?`
Match nothing or any alternative within subrule.




`(x|y|z)*`
Match an alternative within subrule zero or more times.




`(x|y|z)+`
Match an alternative within subrule one or more times.

Figure 4.3: Simple elements and EBNF grammar subrules. The syntax examples use three alternatives with `x`, `y` and `z` representing grammar fragments. The illustrations are based on syntax diagrams from ANTLRStudio[1].

The IR can be manipulated through access functions of the classes building up the IR. There are the usual set and get functions for access to internal objects. Expression and Statement classes offer a `swapWith` method that can for instance be used to swap a statement currently belonging to the IR with another statement created to replace the other. New statements can be inserted before or after existing ones in IR.

The IR tree is built when the parser code is executing. The code for building the IR tree is included when writing the ANTLR grammar. The parser is generated from the grammar and the final parser includes the IR building code as a result (see section 4.1.1). The following list contains important building blocks of the IR data structure:

- `Traversable` interface. It is implemented by every class that is a node, part of the IR tree. Through this interface the parent and children of a node can be accessed by the iterators.

- Iterators. Since Cetus is a source-to-source compiler it is natural that it comes with functionality for modifying the IR tree. Iterators are available for pass writers and constitutes an easy way of traversing the IR tree and finding nodes where changes need to be done. `pruneOn` is a member function available from the iterator classes that traverse the IR tree in depth (`BreadthFirstIterator`, `DepthFirstIterator` and `PostOrderIterator`) and it forces the iterator to skip everything beneath objects of a chosen IR node class. There is also a `FlatIterator` that only iterates over the immediate children of a IR tree node. Below is a Java code example using the breadth first iterator finding all `Procedure` nodes to make changes to them.

```
// prog is the Program object, the root of the IR.
BreadthFirstIterator procIter = new BreadthFirstIterator(prog);
procIter.pruneOn(Procedure.class);
for (;;) {
  Procedure proc = null;
  try {
    proc = (Procedure) procIter.next(Procedure.class);
    // --- make changes to the Procedure object ---
  } catch (NoSuchElementException e) {
    // iterator finished
    break;
  }
}
```

- `Printable` interface. `Traversable` interface extends `Printable` interface which means that every `Traversable` is also `Printable`. The interface is used for printing the code of IR tree nodes to an output stream.

- `SymbolTable` interface. It is implemented by a number of IR node types. All of them have member functions for adding declaration objects which in turn store information about identifiers and data types. Examples of these node types that implement the `SymbolTable` interface, particularly relevant for C language, are `CompoundStatement` (holds information about declarations inside a compound statement), `Procedure` (holds information about parameter declarations of a procedure) and `TranslationUnit` (holds information about global declarations). There is no symbol table storage separate from the IR. Symbol table information is stored at several levels of the IR Tree, corresponding to scope. Depending on starting point, the symbol searching algorithm must search a number of symbol tables on the way to the root of the tree. The root is the global scope, the level furthest away and last to be checked when searching for an identifier.

## 4.2 Cell-NestStep-C Runtime System

In this project, we are using a ported version of NestStep-C, called Cell-NestStep-C and designed to run on the Cell BE. NestStep-C was originally created to run on clusters. More information on the runtime system, in addition to the information presented in sections below, can be found in the thesis [10] about the project that was tasked with the porting of NestStep-C to the Cell BE.

### 4.2.1 The Executable

When compiling an executable for a Cell processor, the SPE programs are embedded within the PPE program. Execution will start with the PPE program. The PPE program will delegate tasks to the SPEs by dispatching an embedded SPE program to a SPE core.

A NestStep program is of SPMD type which means that the local store memory of each SPE is loaded, from start of execution, with an instance of the same program. The program will start to run at the same entry point (i.e. the main function).

The small amount of local store memory available with each SPE (256 kB) is limiting the size of the programs able to run. The binary of the user program and the SPE part of the runtime system must be able to fit together, as a whole, inside the local store memory, and still leave room for data that is processed by the program while executing.

### 4.2.2 PPE tasks

The runtime system is designed to perform some features on the PPE and the rest on the SPEs. Mirror/update requests (section 3.5), combine and prefix sum calculations (section 3.4) are performed on the PPE. The PPE has access to main memory without DMA and perform these operations with easier implementation[10]. The PPE starts the execution with runtime system initialization and loading of SPE programs. Then it enters a loop that will handle messages from each SPE continuously until the program is finished.

### 4.2.3 Main memory storage

Due to the low amount of local store memory, the runtime system is designed to allow the programmer to store variables in main memory instead of local store. For obvious reasons, a large array of, for instance, one million elements can not be stored in local store since the size is limited to only 256 kB. The runtime system comes with functionality for copying smaller pieces of the larger array into a buffer stored in the local store where it can be accessed and used by the SPE. Then, when the work is done, it can be copied back to main memory. Buffer transfers between main memory and local store are done with the use of DMA (see section 4.2.6).

### 4.2.4 Memory manager

The runtime system employs memory managers, present with the PPE and with each SPE. The memory managers keep track of information like for instance sizes of shared arrays and how the elements of a distributed array are distributed.

### 4.2.5 Data structures

With the runtime system comes two types of replicated shared data structures (shared variables and replicated shared arrays) and two types of distributed shared data structures (block distributed arrays and cyclic distributed arrays). Two types of private data structures (private variable and private array) are special for the runtime system for Cell. These private data structures are used when storing data in main memory which is not shared with the other SPEs. The private data structures are used by the PPE to store mirrored data and prefix sum results (mentioned in section 4.2.2). The runtime system supports three primitive datatypes together with its data structures. Those are `int`, `float` and `double`.

### 4.2.6 Buffer transfers with DMA

Data in buffers, stored in local store, can be transfered to and from main memory via asynchronous DMA (direct memory access) read and write requests. There are alignment rules and size rules for the data to be transfered. This has been important to notice when writing the runtime system extension (section 5.2). Transfers larger than 16 byte need to be 16 byte aligned and the size need to be a multiple of 16 byte. Transfers can be maximum 16 kB (16384 bytes). Transfers smaller than 16 byte must be 4 byte aligned, 8 byte aligned or 16 byte aligned depending on the size of the transfer [10].

# Chapter 5

# Compiler Implementation

The implementation task was divided up into two main objectives mentioned in section 1.3. The Cetus extension is described in section 5.1. The compiler adaption of the Cell-NestStep-C runtime system is described in section 5.2.

## 5.1 Cetus Extension

The compiler runs the input code through the parser and the IR is being built. When the parsing is done and the IR is complete, then transformation passes is run on the IR tree. The additions to the IR are described in section 5.1.1 and transformation passes are described in section 5.1.3.

### 5.1.1 Grammar and Internal Representation

The following method has been applied when adding support for new language constructs to the internal representation.

1. Create or modify existing parser rules in the ANTLR grammar to match the syntax of the new language construct.

2. Create new IR hierarchy classes to represent the new language construct.

3. Add action code to instantiate the new IR classes.

4. Test the new construct by writing a test program in NestStep code. The code is compiled with the source-to-source compiler. The output code is compiled and run in the IBM Cell Simulator.

There were in the end no manual adding of new lexer rules. Referring to a string literal within a parser rule automatically defines a token type for the string literal[3].

For instance, ”step” is a string literal connected with the step statement syntax. The following is a listing of parser rules that have been added or modified.

- Parser rules *declaration* and *declarator*: Declaration of shared data structures with specifier `sh` and private data structures with specifier `pb`. For shared data structures can also combine strategy and prefix sum variable be given. This applies to global declarations and to declarations inside bodies of functions.
- Parser rules *declaratorParameterList* and *parameterDeclaration*: Declaration of shared data structures with specifier `sh` and private data structures with specifier `pb`, in the parameter list of the function declaration.
- Parser rule *statement*: Step-statement, Seq-statement and Forall-statement were added. Rules for combining are subrules to the step-statement.
- A number of other subrules to parser rules mentioned above were also added.

There are actually two grammar files coming with Cetus (*NewCParser.g* and *Pre.g*), but only *NewCParser.g* has been relevant for implementation purposes. The other grammar file is a small one, performing some preprocessing on the input file before it passes through the external preprocessor (default external preprocessor is ”*cpp -C*”). Without the preprocessing grammar, the external preprocessor will remove all preprocessor directives (like `#include`) from the input file that should be part of the output. New IR hierarchy classes representing the new language constructs are illustrated in figure 5.1. Important classes which are generated from the grammar files are illustrated in figure 5.2.

Figure 5.1: UML class diagram overview of new classes added to the internal representation (without attributes and methods listed), excluding the vast majority of other IR classes. New statement classes are `ForallStatement`, `SequentialStatement` and `StepStatement`. New declaration classes are `CombineDeclaration` and `NestStepVariableDeclaration`. New declarator class is `NestStepVariableDeclarator`. `NestStepType` class holds information about the type and is a part of `NestStepVariableDeclarator`. Classes are implementing the `Traversable` interface, meaning that the classes are building blocks of the internal representation. The `Printable` interface make the classes print code to a output stream.

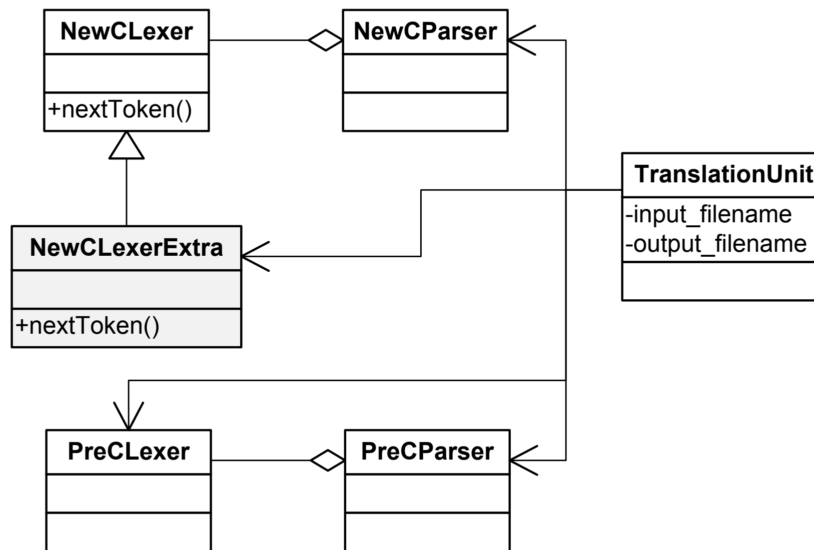



Figure 5.2: `TranslationUnit` represent a file to compile. `TranslationUnit` sets up relationships between lexers (scanners) and parsers and then initiates scanning and parsing. Preparsing is done first (`PreCParser`, `PreCLexer`), followed by external preprocessing, followed by main parsing (`NewCParser`, `NewCLexerExtra`). All irrelevant functions and attributes are left out from the the figure.

### 5.1.2 Group size symbol problem

Support for the group size symbol `#` is not easily inserted as part of the grammar. The symbol is occupied for use with preprocessor directives, i.e. the `#` symbol is part of a character sequence that is interpreted as a preprocessor directive. Although the code is passed through an external preprocessor before it goes through the parser, the parser still has to deal with some preprocessor directives. The support for the symbol had to be implemented outside the ANTLR grammar.

The symbols `$` (for rank) and `#` should be represented, in the IR tree, with identifiers, named the same as the symbols, before they are replaced with other identifiers with other names, used with the output code. The problem was that the `#` symbol is part of a preprocessor lexer rule and a conflict arises when trying to use it in the lexer rule for identifier.

The solution was to create a new class, `NewCLexerExtra`, which is inheriting from the `NewCLexer` class. The `NewCLexer` class implements the scanner and contains a `nextToken` method which purpose is to determine the next token from the character input stream. The scanner is mentioned in section 4.1.1.

The new class overrides the `nextToken` method (see figure 5.2), which will be called by the parser instead of the parent method. The overriding method

will analyze the look-ahead characters and the current state of passed tokens, to make a decision about which method, either the overriding method or the parent method, should handle the interpretation of the next token. If the `#` symbol is not present, the decision is always to let the parent `nextToken` method to interpret the next token.

The goal is to interpret the `#` symbol as a stand alone token, not a preprocessor directive token, when the symbol is encountered in program code that is not part of code that was inserted by the external preprocessor. The parser can read the token stream and interpret the stand alone token as an identifier, i.e. a group size identifier.

### 5.1.3 Transformation Passes

When the parsing is done and the IR is complete, then the source-to-source transformation can begin. A number of transformations are run on the IR tree. The transformations are adding and replacing a lot of statements and expressions in the IR tree. For most times, new function call statements are added. Those calls are directed to the runtime system (sections 4.2 and 5.2). The following is a listing with examples of what the transformations bring about.

- A variable declaration of type `Name` is added to every procedure that contain declarations of NestStep data structures. The `Name` is a struct with two fields; `procedure` and `relative`. The `procedure` field is unique to the procedure and the `relative` field is unique to the data structure it will name [16]. The name is passed by value to the allocation function of the data structures.

- The replicated shared variable and private variable are the two NestStep data structures which are not arrays and should be able to be passed by value to procedures. Since the data structures are always passed by reference, the value of the data structure is copied to a temporary data structure for use inside the procedure. This temporary data structure is set up for each replicated shared variable or private variable found in the list of procedure parameters.

- The header of the main method is transformed to the header used with a SPE program. Function calls for runtime system initialization and buffer size setting are added to the beginning of the main procedure body. Also initialization of the global variables `MY_RANK` and `MY_SIZE` to values of processor rank and group size respectively. A function call for runtime system finalization is added at the end of the main procedure body.

- Expressions are transformed when they include a symbol that is declared as a NestStep data structure. The tree is traversed recursively and when specific conditions are met from the expressions found, type checking is done for symbols to determine if transformation is necessary. The following listed conditions are checked in order. The type of `A` evaluates to a NestStep data structure.

  1. Assignment to an array element (example: `A[0] = 0`).
  2. Assignment to an identifier (example: `A = 0`).
  3. Assignment to an dereferencing unary expression (example: `*A = 0`).
  4. Read of array element (example: `A[0]`).
  5. Read of identifier (examples: `A`, `$` and `#`).
  6. Special constructs such as `mirror`, `update`, `owner` and `owned`. Also constructs for dynamic allocation/deallocation.
  7. Access through unary expression (examples: `*A` and `&A`).

- For every declaration of a NestStep data structure, local to a procedure as well as global, a number of statements are created. They are allocation, deallocation and initialization statements. For declarations local to a procedure these statements are placed at the beginning and end of the procedure. For global declarations, it is more complex, since the statements must be reached from the main method and global declarations can be present in more than one code file. These statements are placed in functions, which names begin with `init_globals_` and `free_globals_`, created and inserted in each file containing global declarations. Declarations marked with `extern`, pointing to these functions, are placed in the file in which the main procedure resides, so that the main procedure can call them.

- Single return transformation transforms all procedures so they only have one returning point. When returning from a procedure, deallocation of local NestStep data structures should be done. The single returning point is placed after the deallocation statements. Usually a return statement brings about an immediate return from the procedure without passing the deallocation statement. This is replaced with a registering of the return value followed by a goto statement, jumping to a label before the deallocation statements.

- The code behind forall statements is generated. Different codes are generated depending on if the forall loop is suppose to iterate over a block distributed or a cyclically distributed array. The generated codes have a for loop in common, but the sets of indexes they iterate over differs. For

a block distributed array, the for loop is set to iterate from a lower to a higher index depending on the owned block. For a cyclically distributed array, the set of indexes is calculated with help from the runtime system function `local_global_CArr_index`.

- For each step statement, function call statements are created for beginning and ending a step statement. Before the ending statement, the combine statements are placed. How many of these statements there are depends on how much combining there is a need of. Mirror and update requests are also placed in this area.

- Single access call transformation. It will transform a program such that every statement contains at most one access function call to a NestStep array data structure. The reason for this is that the access function call may trigger a switch of buffer contents. For example, if two access calls are allowed in a single statement, they might be to the same array data structure, i.e. both will return a pointer to an element inside the same buffer. There would be no control over if the function calls are separated with a buffer contents switch or not. One of the return pointers could point to an element that has been overwritten. Temporaries are introduced to hold the results of access calls.

### 5.1.4 Output from the Compiler

The code generated by the compiler is for the SPU part of the Cell program only. No code is generated for the PPU part, which is identical to every project. See sections 4.2.1 and 4.2.2 for information on the executable and what role the PPE part plays.

It falls on the programmer to retrieve the PPE part. It is easily done by making a copy of one of the folders containing example projects, that follows the source code of the runtime system. The Makefile files will also be included with the copy. It instructs the *make* utility how to automatically build the project. The SPE part of the copy is replaced with the generated file(s) from the compiler. Modifications to a Makefile is needed if filenames do not match.

## 5.2 Cell-NestStep-C Runtime System Extension

The reason for adding an extension to the runtime system was to reduce the amount of code generated by the compiler. The extension can also be called an adaption of the runtime system for the compiler. The following is a list of main issues that are handled by the extension:

- Allocation of buffers
- Automatic transferring of data between LS memory and main memory
- Handling mirror/update requests
- Handling data sizes and alignment for DMA transfers

Some of the functionality in the extension might not be used by every NestStep program. For example, a program might not use replicated shared arrays but the functionality is still included and consequently occupying space. This issue is mentioned in section 6.3 about future extensions. The file `libNestStep_spulib.a` is the part of the library that gets linked into all SPE-binaries. The current size of this file is approximately 93 KiB (which leaves 163 KiB of SPE local store for NestStep program and data). The size before the extension was 53 KiB [10].

The runtime system was extended without any modification to the original code. The new functionality is calling existing functionality. The extension is only concerned with the SPE part of the runtime system.

### 5.2.1 Interface

The compiler creates statements with function calls directed to not only the extended part of the runtime system. Function calls for marking the beginning and ending of a superstep are examples of such. The following is a description of the extended interface. The rest of the interface is presented in [16].

- A function for setting the maximum size of the buffers used by the array data structures.
- New allocation and deallocation functions for every NestStep data structure are replacing the underlying ones. The new functions will, besides setting up the data structure, allocate buffers for storing data.
- Functions for accessing data values within the data structures. For each type of data structure there are three such functions, one for each of the supported primitive datatypes; `int`, `float` and `double`.
- Functions for translating two dimensional and three dimensional indexes to an index that can be used with the access functions. Elements of arrays are stored flat with one dimension.
- Combine functionality for the replicated data structures and private data structures. Before the combine is performed, values of replicated copies

are sent with function calls. The combine strategies are set with function calls. After the combine, new values are transfered back with function calls, including possible results from prefix sum calculations.

- Initialization functions for the private array and replicated array data structures.
- Functions for accessing the mirror and update functionality of the distributed array data structures.
- Functions for locating owned elements (used with code for the forall feature) and determining the ownership of elements of the distributed arrays.
- Pointers to private data and replicated data are implemented with a wrapper data structure. The wrapper functionality is used to forward calls to the right data structure depending on the one stored within. For instance, a private pointer can point to both of the following data structures; private variable and private array.

### 5.2.2 Inner workings

The internal size of the distributed arrays are extended so that the individual distributed block of elements, meet the rules for size and alignment of DMA transfers (see section 4.2.6). The end result will be that data transfered to and from the block will automatically be aligned.

A more detailed explanation is as follows: The number of elements in a block is increased to be a multiple of four elements. The size of four elements is a multiple of 16 bytes. It follows that the size of a block is also a multiple of 16 bytes. The alignment rule is fulfilled because the extended block is pushing the next block in sequence into a position with correct alignment, i.e. the index of the first position of the block is a multiple of four elements. Figure 5.3 illustrates this with the distributed shared array data structures.

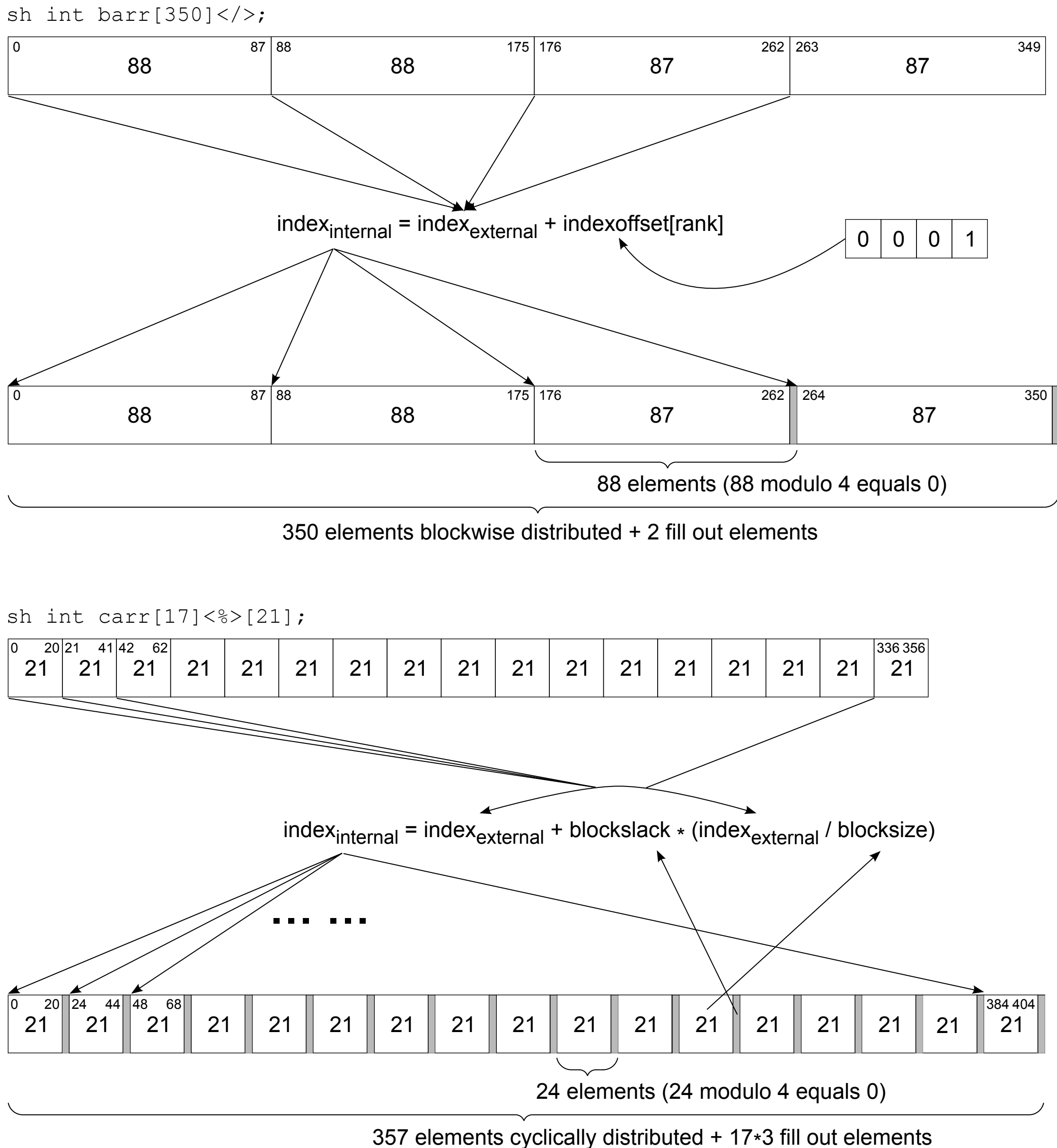



Figure 5.3: Illustrates size extension of distributed shared arrays. External index is the index that comes from the NestStep program. External index translates to an internal index, the position of the element in the extended array. The block distributed array (upper) is divided with four participating processors in mind. Integer divisions are used in the formulas.

# Chapter 6

# Evaluation

## 6.1 Test programs

A couple of test programs were run as part of testing the Cell-NestStep-C runtime system when it was first created. The result from those tests are presented in the report connected to that work [10]. The codes for those tests were handwritten and were passed as an appendix to the report as well as with the source code [7] of the runtime system.

Similar tests have been run again, but this time with compiler generated code that calls the runtime system extension. The input sizes below are rounded. For exact size, see code in appendix for respective test program. The tests involve parallel calculation of the following:

- Pi (appendix B.1). The pi test is a test with a lot of calculation time and with little communication time. The value of $\pi$ is calculated by doing a summation of ten million calculated terms. The summation is divided for parallel execution.
- Dot product (appendix B.2). The dot product test is doing a minimal amount of calculation per item, but with a lot of transferring of data. This test is suitable for testing how well a feature like double buffering works, since the code contains a forall loop (the next buffer contents in line for loading can be determined easily). However, the current state of the compiler does not generate code with double buffering in mind. That feature would have to be added on a later date. There is a separate library, called BlockLib skeleton library [14, 15], that includes double buffering automatically. It was implemented as part of another project. The input to the dot product test program consist of two arrays. They are equal in size and containing nearly 17 million floats each. Elements from the the same position are multiplied and the products are added to form the dot product.

- Prefix sum (appendix B.3). The input consist of an array with size of nearly 8.4 million floats. The output will be an array where the value in each position is the sum of input values before or equal to its position. The last element will contain the sum of all elements in the array. The parallel program will communicate partial sum results using the prefix sum feature of NestStep.

- Jacobi (appendix B.4). The Jacobi test program demonstrated the use of the mirror functionality. It does Jacobi-relaxation on a 1D signal. It takes a signal and flattens it. The input signal is an array of generated data with size is nearly 8.5 million floats. The value at the current position in the signal is approximated with a weighted version of the neighboring values [10]. It outputs to a number of files, equal to the number of participating processors. Merging of the outputted files, in order of ascending rank, will produce a larger result file. When comparing these larger files, from test runs with 1, 2, 4 and 6 participating processors, it is found that the files are the same, i.e. the result is independent of processor numbers. The comparisons were done with the *md5sum* program. Mirror requests are used during all test runs except the one with 1 participating processor and as the outputted results still match, it is reasonable to conclude that mirror works.

Tests have been run on Sony PlayStation 3. The Cell processor is a part of its hardware, but two SPEs are not available, leaving six SPEs left for the test.

The appendix contains code for the test applications, NestStep code as well as the compiler generated code. The compiler does not output code for time monitoring, which has been added manually. There are three measurements of time extracted from each test run; calculation time, DMA wait time and combine time. Calculation time is the accumulated time it takes to do calculation phases of the supersteps. DMA wait time is the accumulated time it takes to transfer data between local store memory and main memory. Combine time is the accumulated time it takes to do combine at the end of supersteps. The DMA wait time is measured from code inside the runtime system extension. The times is reset and retrieved with function calls.

Execution times and speedups are presented in table 6.1. They should not be compared to those presented in [10]. The execution times are included mainly to present relations between time for calculation, DMA wait and combine for each particular test. See figure 6.1 for an illustration of speedup results.

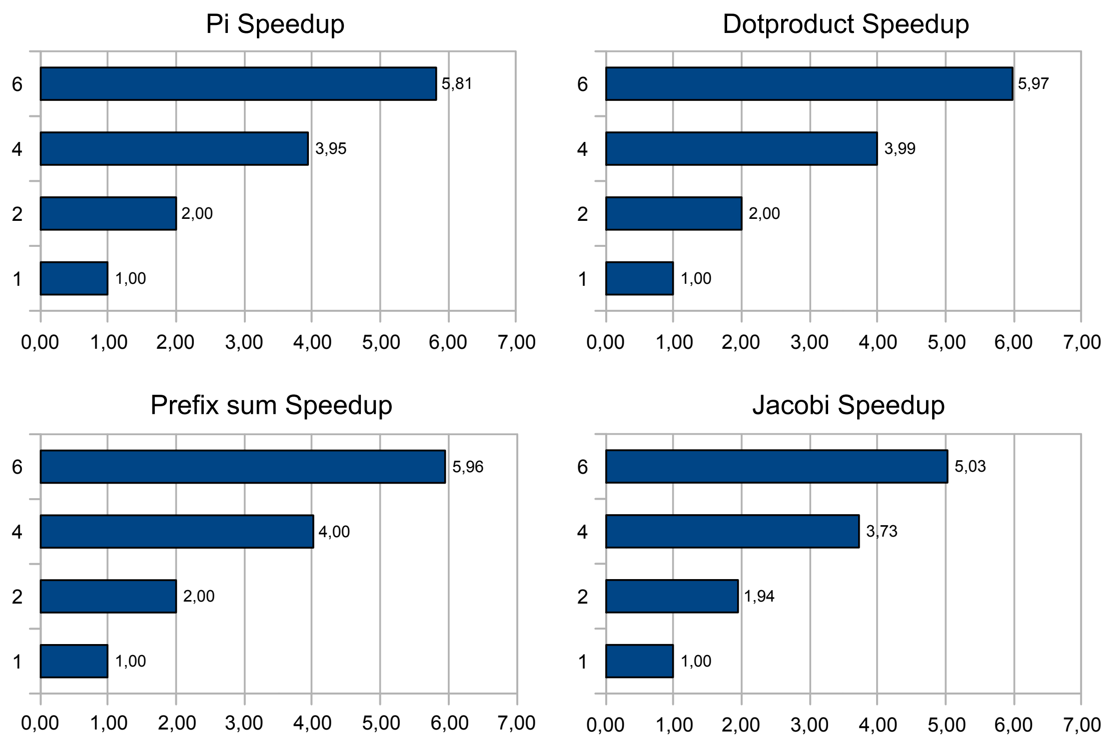



Figure 6.1: Speedup for test programs. The test runs are performed on Sony PlayStation 3 (PS3) using 1, 2, 4 and 6 SPEs. 8 SPEs is not available with PS3.

Table 6.1: Time measurements for the test programs.

**Test: Pi (time measured in seconds)**

| SPEs | 1 | 2 | 4 | 6 |
|---|---|---|---|---|
| Calculation | 2.281 | 1.140 | 0.570 | 0.380 |
| DMA wait | 0.000 | 0.000 | 0.000 | 0.000 |
| Combine | 0.001 | 0.003 | 0.008 | 0.013 |
| Total time (Speedup) | 2.282 (1.00) | 1.144 (2.00) | 0.578 (3.95) | 0.393 (5.81) |

**Test: Dot product (time measured in seconds)**

| SPEs | 1 | 2 | 4 | 6 |
|---|---|---|---|---|
| Calculation | 1.872 | 0.936 | 0.467 | 0.312 |
| DMA wait | 0.043 | 0.023 | 0.011 | 0.007 |
| Combine | 0.000 | 0.000 | 0.000 | 0.001 |
| Total time (Speedup) | 1.915 (1.00) | 0.959 (2.00) | 0.480 (3.99) | 0.321 (5.97) |

**Test: Prefix sum (time measured in seconds)**

| SPEs | 1 | 2 | 4 | 6 |
|---|---|---|---|---|
| Calculation | 1.683 | 0.842 | 0.421 | 0.280 |
| DMA wait | 0.035 | 0.017 | 0.007 | 0.004 |
| Combine | 0.001 | 0.001 | 0.002 | 0.003 |
| Total time (Speedup) | 1.718 (1.00) | 0.860 (2.00) | 0.429 (4.00) | 0.288 (5.96) |

**Test: Jacobi (time measured in seconds)**

| SPEs | 1 | 2 | 4 | 6 |
|---|---|---|---|---|
| Calculation | 2.933 | 1.466 | 0.732 | 0.488 |
| DMA wait | 1.148 | 0.634 | 0.351 | 0.318 |
| Combine | 0.001 | 0.004 | 0.012 | 0.005 |
| Total time (Speedup) | 4.082 (1.00) | 2.106 (1.94) | 1.095 (3.73) | 0.812 (5.03) |

## 6.2 Overhead from the Runtime System

The compiler generates code for a function call every time there is a need to access data values stored within a NestStep data structure. This section discusses the overhead execution time connected with this.

For each NestStep data structure there are one access function for each supported datatype. The name of an access function begins with `address`. The access functions returns pointers to values. The way of accessing values was introduced with the extension of the runtime system.

The main focus, when looking for overhead sources, is on whether the source is producing overhead in conjunction with code with lower or higher execution frequency. It is more important to try to limit overhead at sources in conjunction with higher execution frequency. Allocation functions, for instance, are examples of functions usually called with lower execution frequency. More important are the access function calls accessing data values inside the NestStep data structures. These functions are used often for obvious reasons. A typical place is from the inside of a loop, which would produce high execution frequency.

Time measurements have been done on the pi test code to observe the effects on performance when using the access function call. There are differences in execution times between different access functions. The access function for the shared variable simply return a pointer to where the value is stored, while the access functions for array data structures do tasks like boundary checking and buffer contents switching before a pointer can be returned.

The codes listed in figure 6.2 are two versions with the same calculation result but with different performance. The difference between the two is how the pi variables are accessed. In variation 1, an access function call is used every time access to the shared variable is required. In variation 2, a temporary variable is used inside the loop instead of the function call from variation 1, i.e. direct access instead of a function call returning a pointer. The variation using a temporary variable does execute faster, with a speedup of 1.38 in this particular case. A function call costs execution time in copying of parameters and return value etc. Similar variations were done with the prefix sum test and the dot product test, which produced speedups of 1.08 and 1.07 respectively.

Random access of elements within NestStep array data structures is implemented. A buffer is used to store the currently loaded index interval of data for an array. When an element outside this interval is requested, a buffer contents switch take place. There is some overhead execution time here, with boundary checking etc. No time measurements have been done here. It is enough to look at the code. And since the access functions for arrays are called often, for instance iteration with a loop, accumulated overhead could be substantial.

Variation 1:

```
...
sh double pi = 0.0;
step {
  for(i=$; i<N; i+=#) {
    pi += ...
  }
}
combine(pi<+>);
...
```

Code generated for Variation 1:

```
...
ShVarBuffered * pi;
...
NestStep_step();
{
  for (i=MY_RANK; i<N; i+=MY_SIZE) {
    ( * address_ShVarD(pi)) += ...
  }
}
...
NestStep_end_step();
...
```

Variation 2:

```
...
sh double pi;
double pi_temp = 0.0;
step {
  for(i=$; i<N; i+=#) {
    pi_temp += ...
  }
  pi = pi_temp;
}
combine(pi<+>);
...
```

Code generated for Variation 2:

```
...
ShVarBuffered * pi;
double pi_temp = 0.0;
...
NestStep_step();
{
  for (i=MY_RANK; i<N; i+=MY_SIZE) {
    pi_temp += ...
  }
  ( * address_ShVarD(pi))=pi_temp;
}
...
NestStep_end_step();
...
```

Figure 6.2: Two variations of the pi algorithm. The codes are partial extracts. The focus is not on the pi algorithm but on the difference between using a temporary variable and an access function call. They have different performance. Variation 2 has 1.38 times speedup compared to variation 1.

## 6.3 Future extensions

The following are ideas for future extensions to the compiler, partly based on the discussion of section 6.2. The compiler has no support for these in the current state.

- Adding analysis with the purpose of identifying where the number of unnecessary access function calls can be minimized.
- To generate code for a faster forall loop with direct buffer access, i.e. similar to using one access function call for the beginning of the buffer instead of one call for each element in the buffer. The compiler would have to keep track of possible index accesses that are not equal to the current iteration index. There can be no accesses outside the buffer memory.
- The order of accessing the elements inside a forall loop is predetermined. This is a situation when it is suitable to apply double buffering, i.e. the next element interval in-line for processing is queued to be loaded to a second buffer, while the current element interval is processed from the first buffer.
- Adding support for nested supersteps, i.e. the neststep construct (see section 3.1), to both the compiler and the runtime system.
- The size of the runtime system extension is fixed in current state. The whole is included in the SPE binary. To save space in local store, the compiler could analyze the NestStep code and make a list of used functionality. Then a runtime system extension could be generated, including only used components with dependencies. For instance, a program that does not use replicated shared arrays could have a runtime system extension without code for replicated shared arrays. Another example could be a program that use the mirror construct but not the update construct. Then code for the update construct could be omitted.

# Bibliography


[1] ANTLR Studio website. http://www.placidsystems.com/antlrstudio.aspx.

[2] ANTLR v2 Reference Manual. http://www.antlr2.org/doc/index.html.

[3] ANTLR v2 Specification - ANTLR Meta-Language. http://www.antlr2.org/doc/metalang.html.

[4] ANTLR v2 website. http://www.antlr2.org/.

[5] Cell Broadband Engine resource center - SDK 3.1 Programming Tutorial. http://www.ibm.com/developerworks/power/cell/documents.html.

[6] Eclipse Development Environment website. http://www.eclipse.org/.

[7] NestStep - A bulk-synchronous parallel (BSP) global address space language supporting nested parallelism. http://www.ida.liu.se/~chrke/neststep.html.

[8] The Cetus Project website. http://cetus.ecn.purdue.edu/.

[9] Hansang Bae, Leonardo Bachega, Chirag Dave, Sang-Ik Lee, Seyong Lee, Seung-Jai Min, Rudolf Eigenmann, and Samuel Midkiff. Cetus - A Source-to-Source Compiler Infrastructure for Multicores. Proceedings of the 14th Int'l Workshop on Compilers for Parallel Computing, CPC '09. http://www.ecn.purdue.edu/ParaMount/publications/CPC09.pdf.

[10] Daniel Johansson. Porting the NestStep Run-time System to the CELL Broadband Engine. Master's thesis, Department of Computer and Information Science, Linköping University, Sweden, 2007. LITH-IDA-EX–07/054–SE.

[11] Christoph W. Kessler. NestStep - Nested Parallelism and Virtual Shared Memory for the BSP model. *The Journal of Supercomputing*, 2000.

[12] Christoph W. Kessler. Managing Distributed Shared Arrays in a Bulk-Synchronous Parallel Environment. *Concurrency and Computation: Practice and Experience*, vol. 16:133–153, 2004.

[13] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. Proc. of the Workshop on Languages and Compilers for Parallel Computing(LCPC'03), Oct 2003. `http://www.ece.purdue.edu/~eigenman/reports/lcpc03.pdf`.

[14] Markus Ålind. A Skeleton Library for Cell Broadband Engine. Master's thesis, Department of Computer and Information Science, Linköping University, Sweden, 2008. LIU-IDA/LITH-EX-A–08/002–SE.

[15] Markus Ålind, Mattias V. Eriksson, and Christoph W. Kessler. BlockLib: A Skeleton Library for Cell Broadband Engine. Proc. Int. Workshop on Multicore Software Engineering (IWMSE-2008) at ICSE-2008, Leipzig, Germany, May 2008. ACM.

[16] Joar Sohl. A Scalable Run-Time System for NestStep on Cluster Supercomputers. Master's thesis, Department of Computer and Information Science, Linköping University, Sweden, 2006. LITH-IDA-EX–06/011–SE.

# Appendix A

# Glossary

## A.1 Words and Abbreviations

- ANTLR: ANother Tool for Language Recognition. A tool for generating a parser from a written grammar.
- BSP: Bulk-synchronous Parallel. Model for parallel programming.
- Cell-NestStep-C: The ported version of the NestStep runtime system to the Cell BE.
- Cell BE: Cell BroadBand Engine. Heterogeneous multicore processor from IBM, Sony and Toshiba. Part of the Sony PlayStation 3 hardware.
- DMA: Direct Memory Access. Way of accessing main memory directly from a SPE.
- EIB: Element Interconnect Bus. Fast internal communication bus of the Cell BE.
- IR: Intermediate representation. A data structure that is constructed from the input to the compiler. The output is in turn constructed from the intermediate representation.
- LS: Local Store. Private memory of each SPE.
- MFC: Memory Flow Controller. Part of each SPE, interfacing the EIB, handling DMA transfers.
- MPI: Message Passing Interface. API that allows many computers, organized in computer clusters and supercomputers, to communicate with each other.

- PPE: Power Processor Element. See section 2.1.
- SDK: Software Development Kit. It is a set of development tools that allow for creation of applications.
- SPE: Synergistic Processing Elements. See section 2.2.
- SPMD: Single Program Multiple Data. It is a common style of parallel programming. One instance of the same program will run on each processor.

# Appendix B

# Code for Test Programs

This appendix contains the code for the test programs (see section 6.1). C code for the Cell BE is generated from NestStep code by the compiler, but some code concerning time monitoring has been inserted manually into the generated code. The NestStep code corresponds to C code marked with gray background.

## B.1 Pi

### NestStep code

```
double parallel_pi() {
  int N = 10000000;
  int i;
  double h = 1.0 / N;
  sh double pi;
  double pi_temp = 0.0;
  step {
    for(i=$; i<N; i+=#) {
      pi_temp += 4.0 / (1 + h*(i-0.5)*h*(i-0.5));
    }
    pi = pi_temp;
  }
  combine(pi<+>);
  step {
    pi *= h;
  }
  combine(pi<=>);
  return pi;
}
```

# C code

```
#include "../../NestStep_spulib/neststep_buffered_spu.h"
int MY_RANK;
int MY_SIZE;
#include <stdio.h>
#define TIMEBASE 79800000
#define DEC_INIT_VAL 0XFFFFFFFF

uint32_t calc1 = 0, calc2 = 0, calctot = 0;
uint32_t comb1 = 0, comb2 = 0, combtot = 0;
uint32_t prog1 = 0, prog2 = 0, progtot = 0;
uint32_t waittot;
void print_time( )
{
    NestStep_step();
    {
        if ((MY_RANK==0))
        {
            printf("Tot programtime: %f\n", (((float)progtot)/TIMEBASE));
            printf("Time spent calculating: %f\n", (((float)(calctot-waittot))/TIMEBASE));
            printf("Time spent waiting: %f\n", (((float)waittot)/TIMEBASE));
            printf("Time spent in combine: %f\n", (((float)combtot)/TIMEBASE));
        }
    }
    NestStep_combine(NULL, NULL);
    NestStep_end_step();
    _done:
    return ;
}
double parallel_pi( )
{
    int N = 10000000;
    int i;
    double h = (1.0/N);
    ShVarBuffered * pi;
    double pi_temp = 0.0;
    double _ret_val;
    Name name;
    {
        name.relative=0;
        name.procedure=1;
    }
    {
        pi=new_ShVarBuffered(( & name), DVAR);
    }
    NestStep_step();
    {
        calc1=spu_read_decrementer();
        for (i=MY_RANK; i<N; i+=MY_SIZE)
        {
            pi_temp+=(4.0/(1+(((h*(i-0.5))*h)*(i-0.5))));
        }
        ( * address_ShVarD(pi))=pi_temp;
        calc2=spu_read_decrementer();
        calctot+=(calc1-calc2);
    }
    before_combine_ShVarD(pi);
    NestStep_ShVarBuffered_attach(pi, ADD, NO_PREFIX, NULL);
    comb1=spu_read_decrementer();
    NestStep_combine(NULL, NULL);
    comb2=spu_read_decrementer();
    combtot+=(comb1-comb2);
    NestStep_end_step();
    after_combine_ShVarD(pi);
    NestStep_step();
    {
        ( * address_ShVarD(pi))*=h;
    }
    NestStep_combine(NULL, NULL);
    NestStep_end_step();
    {
        _ret_val=( * address_ShVarD(pi));
        goto _done;
    }
    _done:
```

```
75      {
            free_ShVarBuffered(pi);
        }
        return _ret_val;
    }
80  int main(unsigned long long speid, Addr64 argp, Addr64 envp)
    {
        int m;
        double pi;
        int _ret_val;
85      Name name;
        {
            NestStep_SPU_init(argp);
            init_max_buffer_size(1024);
            MY_SIZE=NestStep_get_size();
90          MY_RANK=NestStep_get_rank();
            name.relative=0;
            name.procedure=2;
        }
        spu_write_decrementer(DEC_INIT_VAL);
95      prog1=spu_read_decrementer();
        reset_waiting_time();
        for (m=0; m<4; ++ m)
        {
            pi=parallel_pi();
100     }
        waittot=get_waiting_time();
        prog2=spu_read_decrementer();
        progtot+=(prog1-prog2);
        NestStep_step();
105     {
            if ((MY_RANK==0))
            {
                printf("Pi = %.10f\n", pi);
            }
110     }
        NestStep_combine(NULL, NULL);
        NestStep_end_step();
        print_time();
        {
115         _ret_val=0;
            goto _done;
        }
        _done:
        {
120         NestStep_SPU_finalize();
        }
        return _ret_val;
    }
```

# B.2 Dot Product

## NestStep code

```
float dotproduct(sh float *x</>, sh float *y</>) {
  int i;
  sh<=> float z;
  float z_temp = 0.0;
  step {
    // this requires an equal distribution of x and y.
    forall(i, x) {
      z_temp += x[i]*y[i];
    }
    z = z_temp;
  }
  combine(z<+>);
  return z;
}
```

# C code

```
#include "../../NestStep_spulib/neststep_buffered_spu.h"
int MY_RANK;
int MY_SIZE;
#include <stdio.h>
#define TIMEBASE 79800000
#define DEC_INIT_VAL 0XFFFFFFFF

uint32_t calc1 = 0, calc2 = 0, calctot = 0;
uint32_t comb1 = 0, comb2 = 0, combtot = 0;
uint32_t prog1 = 0, prog2 = 0, progtot = 0;
uint32_t waittot;
void print_time( )
{
    NestStep_step();
    {
        if ((MY_RANK==0))
        {
            printf("Tot programtime: %f\n", (((float)progtot)/TIMEBASE));
            printf("Time spent calculating: %f\n", (((float)(calctot-waittot))/TIMEBASE));
            printf("Time spent waiting: %f\n", (((float)waittot)/TIMEBASE));
            printf("Time spent in combine: %f\n", (((float)combtot)/TIMEBASE));
        }
    }
    NestStep_combine(NULL, NULL);
    NestStep_end_step();
    _done:
    return ;
}
float dotproduct(BlockDistArrayBuffered * x, BlockDistArrayBuffered * y)
{
    int i;
    ShVarBuffered * z;
    float z_temp = 0.0;
    float _ret_val;
    Name name;
    int _forall_low0, _forall_high0;
    {
        name.relative=0;
        name.procedure=1;
    }
    {
        z=new_ShVarBuffered((& name), FVAR);
    }
    NestStep_step();
    {
        calc1=spu_read_decrementer();
        // this requires an equal distribution of x and y.

        _forall_low0=(x->lower);
        _forall_high0=(x->higher);
        for (i=_forall_low0; i<=_forall_high0; ++i)
        {
            float _tmp0;
            float _tmp1;
            _tmp0=( * address_BArrF(y, i, 0));
            _tmp1=( * address_BArrF(x, i, 0));
            z_temp+=(_tmp1*_tmp0);
        }
        ( * address_ShVarF(z))=z_temp;
        calc2=spu_read_decrementer();
        calctot+=(calc1-calc2);
    }
    before_combine_ShVarF(z);
    NestStep_ShVarBuffered_attach(z, ADD, NO_PREFIX, NULL);
    comb1=spu_read_decrementer();
    NestStep_combine(NULL, NULL);
    comb2=spu_read_decrementer();
    combtot+=(comb1-comb2);
    NestStep_end_step();
    after_combine_ShVarF(z);
    {
        _ret_val=( * address_ShVarF(z));
        goto _done;
    }
```

```
75      _done:
        {
            free_ShVarBuffered(z);
        }
        return _ret_val;
80  }
    int main(unsigned long long speid, Addr64 argp, Addr64 envp)
    {
        int N = (1048576*16);
        int i;
85      BlockDistArrayBuffered * x;
        BlockDistArrayBuffered * y;
        float dotp;
        int _ret_val;
        Name name;
90      int _forall_low0, _forall_high0;
        int _forall_low1, _forall_high1;
        {
            NestStep_SPU_init(argp);
            init_max_buffer_size(1024);
95          MY_SIZE=NestStep_get_size();
            MY_RANK=NestStep_get_rank();
            name.relative=0;
            name.procedure=2;
        }
100     x=new_BlockDistArrayBuffered(( & name), FVAR, 1, 1, N);
        y=new_BlockDistArrayBuffered(( & name), FVAR, 1, 1, N);
        NestStep_step();
        {
            _forall_low0=(x->lower);
105         _forall_high0=(x->higher);
            for (i=_forall_low0; i<=_forall_high0; ++i)
            {
                ( * address_BArrF(x, i, 1))=3.0;
            }
110         _forall_low1=(y->lower);
            _forall_high1=(y->higher);
            for (i=_forall_low1; i<=_forall_high1; ++i)
            {
                ( * address_BArrF(y, i, 1))=2.0;
115         }
        }
        NestStep_combine(NULL, NULL);
        NestStep_end_step();
        spu_write_decrementer(DEC_INIT_VAL);
120     prog1=spu_read_decrementer();
        reset_waiting_time();
        dotp=dotproduct(x, y);
        waittot=get_waiting_time();
        prog2=spu_read_decrementer();
125     progtot+=(prog1-prog2);
        NestStep_step();
        {
            if ((MY_RANK==0))
            {
130             printf("Dot product: %f\n", dotp);
                printf("Expected (3.0 * 2.0 * N): %f  (fault size: %f)\n", ((3.0*2.0)*N),
                    (((3.0*2.0)*N)-dotp));
            }
        }
        NestStep_combine(NULL, NULL);
135     NestStep_end_step();
        free_BlockDistArrayBuffered(x);
        free_BlockDistArrayBuffered(y);
        print_time();
        {
140         _ret_val=0;
            goto _done;
        }
        _done:
        {
145         NestStep_SPU_finalize();
        }
        return _ret_val;
    }
```

# B.3 Prefix Sum

## NestStep code

```
void prefixsum(sh float *x</>) {
  int i;
  sh double z;
  double z_temp = 0.0;
  pb double prefix_sum;
  step {
    forall(i, x) {
      z_temp += (double) x[i];
      x[i] = (float) z_temp;
    }
    z = z_temp;
  }
  combine(z<+:prefix_sum>);
  step {
    forall(i, x) {
      x[i] += (float) prefix_sum;
    }
  }
}
```

## C code

```
#include "../../NestStep_spulib/neststep_buffered_spu.h"
int MY_RANK;
int MY_SIZE;
#include <stdio.h>
#define TIMEBASE 79800000
#define DEC_INIT_VAL 0XFFFFFFFF

uint32_t calc1 = 0, calc2 = 0, calctot = 0;
uint32_t comb1 = 0, comb2 = 0, combtot = 0;
uint32_t prog1 = 0, prog2 = 0, progtot = 0;
uint32_t waittot;
void print_time( )
{
    NestStep_step();
    {
        if ((MY_RANK==0))
        {
            printf("Tot programtime: %f\n", (((float)progtot)/TIMEBASE));
            printf("Time spent calculating: %f\n", (((float)(calctot-waittot))/TIMEBASE));
            printf("Time spent waiting: %f\n", (((float)waittot)/TIMEBASE));
            printf("Time spent in combine: %f\n", (((float)combtot)/TIMEBASE));
        }
    }
    NestStep_combine(NULL, NULL);
    NestStep_end_step();
    _done:
    return ;
}
void prefixsum(BlockDistArrayBuffered * x)
{
    int i;
    ShVarBuffered * z;
    double z_temp = 0.0;
    PVarBuffered * prefix_sum;
    Name name;
    int _forall_low0, _forall_high0;
    int _forall_low1, _forall_high1;
```

```
    {
        name.relative=0;
40      name.procedure=1;
    }
    {
        z=new_ShVarBuffered((&name), DVAR);
        prefix_sum=new_PVarBuffered((&name), DVAR);
45  }
    NestStep_step();
    {
        calc1=spu_read_decrementer();
        _forall_low0=(x->lower);
50      _forall_high0=(x->higher);
        for (i=_forall_low0; i<=_forall_high0; ++i)
        {
            z_temp+=((double)(*address_BArrF(x, i, 0)));
            (*address_BArrF(x, i, 1))=((float)z_temp);
55      }
        (*address_ShVarD(z))=z_temp;
        calc2=spu_read_decrementer();
        calctot+=(calc1-calc2);
    }
60  before_combine_ShVarD(z);
    NestStep_ShVarBuffered_attach(z, ADD, PREFIX, prefix_sum);
    comb1=spu_read_decrementer();
    NestStep_combine(NULL, NULL);
    comb2=spu_read_decrementer();
65  combtot+=(comb1-comb2);
    NestStep_end_step();
    after_combine_ShVarD(z);
    NestStep_step();
    {
70      calc1=spu_read_decrementer();
        _forall_low1=(x->lower);
        _forall_high1=(x->higher);
        for (i=_forall_low1; i<=_forall_high1; ++i)
        {
75          (*address_BArrF(x, i, 1))+=((float)(*address_PVarD(prefix_sum)));
        }
        calc2=spu_read_decrementer();
        calctot+=(calc1-calc2);
    }
80  comb1=spu_read_decrementer();
    NestStep_combine(NULL, NULL);
    comb2=spu_read_decrementer();
    combtot+=(comb1-comb2);
    NestStep_end_step();
85  _done:
    {
        free_ShVarBuffered(z);
        free_PVarBuffered(prefix_sum);
    }
90  return ;
}
int main(unsigned long long speid, Addr64 argp, Addr64 envp)
{
    int N = (1048576*8);
95  int i;
    BlockDistArrayBuffered * x;
    int _ret_val;
    Name name;
    int _forall_low0, _forall_high0;
100 {
        NestStep_SPU_init(argp);
        init_max_buffer_size(1024);
        MY_SIZE=NestStep_get_size();
        MY_RANK=NestStep_get_rank();
105     name.relative=0;
        name.procedure=2;
    }
    x=new_BlockDistArrayBuffered((&name), FVAR, 1, 1, N);
    NestStep_step();
110 {
        _forall_low0=(x->lower);
        _forall_high0=(x->higher);
        for (i=_forall_low0; i<=_forall_high0; ++i)
        {
```

```
115             ( * address_BArrF(x, i, 1))=0.02;
            }
        }
        NestStep_combine(NULL, NULL);
        NestStep_end_step();
120     spu_write_decrementer(DEC_INIT_VAL);
        prog1=spu_read_decrementer();
        reset_waiting_time();
        prefixsum(x);
        waittot=get_waiting_time();
125     prog2=spu_read_decrementer();
        progtot+=(prog1-prog2);
        NestStep_step();
        {
            if ((MY_RANK==(MY_SIZE-1)))
130         {
                printf("Expected last: %f\n", (0.02*N));
                printf("x[N-1]: %f\n", ( * address_BArrF(x, (N-1), 0)));
            }
        }
135     NestStep_combine(NULL, NULL);
        NestStep_end_step();
        free_BlockDistArrayBuffered(x);
        print_time();
        {
140         _ret_val=0;
            goto _done;
        }
        _done:
        {
145         NestStep_SPU_finalize();
        }
        return _ret_val;
    }
```

## B.4 Jacobi

### NestStep code

```
int N = 8454144;
void jacobi(sh float *A</>, sh float *B</>) {
  pb float left[2];
  pb float right[2];
  int bsize = N/#;
  int i;
  int bindex;
  step {
    if ($ != 0) {
      mirror(A, left, $*bsize - 2, $*bsize - 1);
    }
    else {
      left[0] = left[1] = 0.0;
    }
    if ($ != # - 1) {
      mirror(A, right, ($+1)*bsize, ($+1)*bsize+1);
    }
    else {
      right[0] = right[1] = 1.0;
    }
  }
  step {
    forall(i, B) {
```

```
        bindex = i % bsize;
        if (bindex == 0) {
          B[i] = (-left[0]+4*left[1]+4*A[i+1]-A[i+2]) / 6;
        }
        else if (bindex == 1) {
          B[i] = (-left[1]+4*A[i-1]+4*A[i+1]-A[i+2]) / 6;
        }
        else if (bindex == bsize - 2) {
          B[i] = (-A[i-2]+4*A[i-1]+4*A[i+1]-right[0]) / 6;
        }
        else if (bindex == bsize - 1) {
          B[i] = (-A[i-2]+4*A[i-1]+4*right[0]-right[1]) / 6;
        }
        else {
          B[i] = (-A[i-2]+4*A[i-1]+4*A[i+1]-A[i+2]) / 6;
        }
      }
    }
}
```

# C code

```
   #include "../../NestStep_spulib/neststep_buffered_spu.h"
   int MY_RANK;
   int MY_SIZE;
   #include <stdio.h>
 5 #define TIMEBASE 79800000
   #define DEC_INIT_VAL 0XFFFFFFFF

   uint32_t calc1 = 0, calc2 = 0, calctot = 0;
   uint32_t comb1 = 0, comb2 = 0, combtot = 0;
10 uint32_t prog1 = 0, prog2 = 0, progtot = 0;
   uint32_t waittot;
   void print_time(  )
   {
       NestStep_step();
15     {
           if ((MY_RANK==0))
           {
               printf("Tot programtime: %f\n", (((float)progtot)/TIMEBASE));
               printf("Time spent calculating: %f\n", (((float)(calctot-waittot))/TIMEBASE));
20             printf("Time spent waiting: %f\n", (((float)waittot)/TIMEBASE));
               printf("Time spent in combine: %f\n", (((float)combtot)/TIMEBASE));
           }
       }
       NestStep_combine(NULL, NULL);
25     NestStep_end_step();
       _done:
       return ;
   }
   // 43 * 4096 * 1 * 2 * 4 * 6 = 8454144
30 // (work can be divided between 1, 2, 4 or 6 SPEs)
   int N = 8454144;
   void read_file(FILE * file, BlockDistArrayBuffered * barr, int lbound, int ubound)
   {
       int i, u;
35     int size = ((ubound-lbound)+1);
       int blocksize = 4096;
       int noblocks = (size/blocksize);
       float * buffer = malloc((blocksize*sizeof (float)));
       int low;
40     Name name;
       {
           name.relative=0;
           name.procedure=1;
       }
45     fseek(file, (lbound*sizeof (float)), 0);
       for (i=0; i<noblocks; i ++ )
```

```
    {
        fread(buffer, sizeof (float), blocksize, file);
        low=(lbound+(i*blocksize));
50      for (u=low; u<(low+blocksize); ++ u)
        {
            ( * address_BArrF(barr, u, 1))=buffer[(u-low)];
        }
    }
55  free(buffer);
    _done:
    return ;
}
void write_file(FILE * file, BlockDistArrayBuffered * barr, int lbound, int ubound)
60 {
    int i, u;
    int size = ((ubound-lbound)+1);
    int blocksize = 4096;
    int noblocks = (size/blocksize);
65  float * buffer = malloc((blocksize*sizeof (float)));
    int low;
    Name name;
    {
        name.relative=0;
70      name.procedure=2;
    }
    for (i=0; i<noblocks; i ++ )
    {
        low=(lbound+(i*blocksize));
75      for (u=low; u<(low+blocksize); ++ u)
        {
            buffer[(u-low)]=( * address_BArrF(barr, u, 0));
        }
        fwrite(buffer, sizeof (float), blocksize, file);
80  }
    free(buffer);
    _done:
    return ;
}
85 void jacobi(BlockDistArrayBuffered * A, BlockDistArrayBuffered * B)
{
    PrivateArrayBuffered * left;
    PrivateArrayBuffered * right;
    int bsize = (N/MY_SIZE);
90  int i;
    int bindex;
    Name name;
    int _forall_low0, _forall_high0;
    {
95      name.relative=0;
        name.procedure=3;
    }
    {
        left=new_PrivateArrayBuffered(( & name), FVAR, 1, 1, 2);
100     right=new_PrivateArrayBuffered(( & name), FVAR, 1, 1, 2);
    }
    NestStep_step();
    {
        calc1=spu_read_decrementer();
105     if ((MY_RANK!=0))
        {
            mirror_register_BArr(A, left, ((MY_RANK*bsize)-2), ((MY_RANK*bsize)-1));
        }
        else
110     {
            float _tmp0;
            float _tmp1;
            _tmp0=( * address_PArrF(left, 1));
            _tmp1=( * address_PArrF(left, 0));
115         _tmp1=(_tmp0=0.0);
            ( * address_PArrF(left, 1))=_tmp0;
            ( * address_PArrF(left, 0))=_tmp1;
        }
        if ((MY_RANK!=(MY_SIZE-1)))
120     {
            mirror_register_BArr(A, right, ((MY_RANK+1)*bsize), (((MY_RANK+1)*bsize)+1));
        }
        else
```

```
            {
125             float _tmp2;
                float _tmp3;
                _tmp2=( * address_PArrF(right , 1));
                _tmp3=( * address_PArrF(right , 0));
                _tmp3=(_tmp2=1.0);
130             ( * address_PArrF(right , 1))=_tmp2;
                ( * address_PArrF(right , 0))=_tmp3;
            }
            calc2=spu_read_decrementer();
            calctot+=(calc1-calc2);
135     }
        mirror_prep_BArr(A);
        comb1=spu_read_decrementer();
        NestStep_combine(NULL, NULL);
        NestStep_end_step();
140     NestStep_step();
        {
        }
        mirror_transfer_BArr(A);
        NestStep_combine(NULL, NULL);
145     comb2=spu_read_decrementer();
        combtot+=(comb1-comb2);
        NestStep_end_step();
        NestStep_step();
        {
150         calc1=spu_read_decrementer();
            _forall_low0=(B->lower);
            _forall_high0=(B->higher);
            for (i=_forall_low0; i<=_forall_high0; ++ i)
            {
155             bindex=(i%bsize);
                if ((bindex==0))
                {
                    float _tmp4;
                    float _tmp5;
160                 float _tmp6;
                    float _tmp7;
                    float _tmp8;
                    _tmp4=( * address_PArrF(left , 1));
                    _tmp5=( * address_PArrF(left , 0));
165                 _tmp6=( * address_BArrF(A, (i+1), 0));
                    _tmp7=( * address_BArrF(A, (i+2), 0));
                    _tmp8=( * address_BArrF(B, i , 1));
                    _tmp8=((((( - _tmp5)+(4*_tmp4))+(4*_tmp6))-_tmp7)/6);
                    ( * address_BArrF(B, i , 1))=_tmp8;
170             }
                else
                {
                    if ((bindex==1))
                    {
175                     float _tmp9;
                        float _tmp10;
                        float _tmp11;
                        float _tmp12;
                        float _tmp13;
180                     _tmp9=( * address_BArrF(A, (i-1), 0));
                        _tmp10=( * address_PArrF(left , 1));
                        _tmp11=( * address_BArrF(A, (i+1), 0));
                        _tmp12=( * address_BArrF(A, (i+2), 0));
                        _tmp13=( * address_BArrF(B, i , 1));
185                     _tmp13=((((( - _tmp10)+(4*_tmp9))+(4*_tmp11))-_tmp12)/6);
                        ( * address_BArrF(B, i , 1))=_tmp13;
                    }
                    else
                    {
190                     if ((bindex==(bsize-2)))
                        {
                            float _tmp14;
                            float _tmp15;
                            float _tmp16;
195                         float _tmp17;
                            float _tmp18;
```

```
                            _tmp14=( * address_BArrF(A, (i-1), 0));
                            _tmp15=( * address_BArrF(A, (i-2), 0));
                            _tmp16=( * address_BArrF(A, (i+1), 0));
200                         _tmp17=( * address_PArrF(right , 0));
                            _tmp18=( * address_BArrF(B, i , 1));
                            _tmp18=((((( - _tmp15)+(4*_tmp14))+(4*_tmp16))-_tmp17)/6);
                            ( * address_BArrF(B, i , 1))=_tmp18;
                        }
205                     else
                        {
                            if ((bindex==(bsize-1)))
                            {
                                float _tmp19;
210                             float _tmp20;
                                float _tmp21;
                                float _tmp22;
                                float _tmp23;
                                _tmp19=( * address_BArrF(A, (i-1), 0));
215                             _tmp20=( * address_BArrF(A, (i-2), 0));
                                _tmp21=( * address_PArrF(right , 0));
                                _tmp22=( * address_PArrF(right , 1));
                                _tmp23=( * address_BArrF(B, i , 1));
                                _tmp23=((((( - _tmp20)+(4*_tmp19))+(4*_tmp21))-_tmp22)/6);
220                             ( * address_BArrF(B, i , 1))=_tmp23;
                            }
                            else
                            {
                                float _tmp24;
225                             float _tmp25;
                                float _tmp26;
                                float _tmp27;
                                float _tmp28;
                                _tmp24=( * address_BArrF(A, (i-1), 0));
230                             _tmp25=( * address_BArrF(A, (i-2), 0));
                                _tmp26=( * address_BArrF(A, (i+1), 0));
                                _tmp27=( * address_BArrF(A, (i+2), 0));
                                _tmp28=( * address_BArrF(B, i , 1));
                                _tmp28=((((( - _tmp25)+(4*_tmp24))+(4*_tmp26))-_tmp27)/6);
235                             ( * address_BArrF(B, i , 1))=_tmp28;
                            }
                        }
                    }
                }
240         }
            calc2=spu_read_decrementer();
            calctot+=(calc1-calc2);
        }
        comb1=spu_read_decrementer();
245     NestStep_combine(NULL, NULL);
        comb2=spu_read_decrementer();
        combtot+=(comb1-comb2);
        NestStep_end_step();
        _done:
250     {
            free_PrivateArrayBuffered(left);
            free_PrivateArrayBuffered(right);
        }
        return ;
255 }
    int main(unsigned long long speid , Addr64 argp , Addr64 envp)
    {
        BlockDistArrayBuffered * A;
        BlockDistArrayBuffered * B;
260     char infile[21];
        char outfile[21];
        int local_low;
        FILE * ifile , * ofile;
        int _ret_val;
265     Name name;
        {
            NestStep_SPU_init(argp);
            init_max_buffer_size(1024);
            MY_SIZE=NestStep_get_size();
270         MY_RANK=NestStep_get_rank();
            name.relative=0;
```

```
            name.procedure=4;
        }
        A=new_BlockDistArrayBuffered((&name), FVAR, 1, 1, N);
275     B=new_BlockDistArrayBuffered((&name), FVAR, 1, 1, N);
        sprintf(infile, "a.in");
        sprintf(outfile, "%i.out", MY_RANK);
        local_low=(MY_RANK*(N/MY_SIZE));
        // read file
280
        ifile=fopen(infile, "r");
        if ((!ifile))
        {
            printf("Could not open file\n");
285         {
                _ret_val=1;
                goto _done;
            }
        }
290     read_file(ifile, A, local_low, ((local_low+(N/MY_SIZE))-1));
        fclose(ifile);
        spu_write_decrementer(DEC_INIT_VAL);
        prog1=spu_read_decrementer();
        reset_waiting_time();
295     jacobi(A, B);
        waittot=get_waiting_time();
        prog2=spu_read_decrementer();
        progtot+=(prog1-prog2);
        // write file
300
        ofile=fopen(outfile, "w+");
        if ((!ofile))
        {
            printf("Could not open file\n");
305         {
                _ret_val=1;
                goto _done;
            }
        }
310     write_file(ofile, B, local_low, ((local_low+(N/MY_SIZE))-1));
        fclose(ofile);
        free_BlockDistArrayBuffered(A);
        free_BlockDistArrayBuffered(B);
        print_time();
315     {
            _ret_val=0;
            goto _done;
        }
        _done:
320     {
            NestStep_SPU_finalize();
        }
        return _ret_val;
    }
```

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Magnus Holm