Metaprogrammed algorithmic skeletons : implementations, performances and semantics

Noman Javed

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE présentée par :

Noman JAVED

soutenue le : **21 Octobre 2011**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

Squelettes algorithmiques méta-programmés : implantations, performances et sémantique

THÈSE dirigée par :

Frédéric LOULERGUE	Professeur, Université d'Orléans

RAPPORTEURS :

Fabrice MOURLIN	Maître de conférences HDR, Université Paris Est Créteil
Susanna PELAGATTI	Associate Professor, Università di Pisa

JURY :

Sébastien LIMET	Professeur, Université d'Orléans
Herbert KUCHEN	Professeur, Westfische Wilhelms- Universität Münster
Joël FALCOU	Maître de conférences, Université Paris-Sud 11

To my parents.

Acknowledgements

My praises to God for giving me the diligence and perseverance to endure the long PhD journey. Only by His grace was I able to complete the degree requirements.

I would like to thank Fabrice MOURLIN and Susanna PELAGATTI for reviewing this thesis and providing their valuable suggestions. I would like to thank Herbert KUCHEN, Joël FALCOU and Sébastien LIMET for being members of the jury.

The thesis could never have been written without the helpful support of a number of people to whom I would like to pay tribute. First and foremost I would like to thank my supervisor Frédéric LOULERGUE who honoured me by becoming my thesis director. I would like to express my gratitude for all his constant support, friendly encouragement, great patience, invaluable advice, and excellent guidance. A part from the technical work, I always found him willing to support me in every aspect. Without him, it is simply impossible for me to complete my work.

I feel fortunate to work at LIFO. A great place full of wonderful people. I feel proud to be a part of it. I am grateful to all the lab mates. Julien TESSON and Matthieu LOPEZ deserve my special thanks as they made my life easy and help me overcoming my language deficiency. I would like to thank my office mates Mustafa BAMHA and Jean Jacque LACRAMPES for their great patience and support.

I had the privilege to work with Joel FALCOU several times and he inspired me every time I met him. I pay my gratitude to him for boosting me towards the world of C++ meta-programming.

Special thanks to Higher Education Commission, Pakistan who provide me the financial aid to complete my work. A very special gratitude to Atta ur Rehman, founder of HEC Pakistan for enabling fellows like me to pursue higher studies.

Last but not least, I would like to thank my very strong family support system. These include my parents, my wife, my son and my siblings for their great love, patience and sacrifice. I am truly blessed.

Noman JAVED

Contents

Contents vii

List of Figures ix

1 Introduction 1
- 1.1 Parallel Architectures 2
- 1.2 Parallel Programming Models 2
  - 1.2.1 Architecture Oriented Models 3
  - 1.2.2 Object Oriented Models 4
- 1.3 Parallel Algorithmic Models 4
  - 1.3.1 Network Models 4
  - 1.3.2 Parallel Random Access Machine 5
  - 1.3.3 Bridging Models 5
- 1.4 Formal Models 7
- 1.5 Criteria to Evaluate Programming Model 7
  - 1.5.1 Criteria 8
  - 1.5.2 Assessment of Programming Models 8
- 1.6 Structured Parallelism 9
- 1.7 Contribution and Structure of the Dissertation 10

2 State of the Art 13
- 2.1 Classification of Skeletons 13
  - 2.1.1 Data Parallel Skeletons 14
  - 2.1.2 Task Parallel Skeletons 14
  - 2.1.3 Control Skeletons 14
- 2.2 Skeleton Libraries and Languages 15
  - 2.2.1 Contributions of Pisa Group 15
  - 2.2.2 Contributions of Herbert Kuchen 19
  - 2.2.3 BSP based Skeleton Libraries 20
  - 2.2.4 C++ based Skeleton Libraries 21
  - 2.2.5 Skandium 22
- 2.3 Other High Level Frameworks 22
  - 2.3.1 Eden 22
  - 2.3.2 STAPL 23

2.3.3 HPC++ . . . 23
2.3.4 TBB . . . 23
2.4 Discussion . . . 24

3 OSL Design and Implementation 27
3.1 Data Parallel Skeletons . . . 28
3.1.1 Distributed Arrays . . . 28
3.1.2 Map Skeletons . . . 28
3.1.3 Communication Skeletons . . . 30
3.1.4 Reduce . . . 33
3.1.5 View Changing Skeletons . . . 33
3.2 A Prototype Implementation in BSML . . . 35
3.2.1 Bulk Synchronous Parallel ML . . . 35
3.2.2 A Library of Algorithmic Skeletons . . . 38
3.2.3 Heat Diffusion Simulation Example and Experiments . . . 43
3.3 A C++ Implementation of OSL . . . 45
3.3.1 Distributed Arrays . . . 45
3.3.2 Skeletons . . . 47
3.4 Summary . . . 54

4 Programming with OSL 55
4.1 A First Example . . . 55
4.2 Heat Equation: A case study . . . 57
4.2.1 One Dimensional Heat Equation . . . 57
4.2.2 Two Dimensional Heat Equation . . . 60
4.3 Fast Fourier Transform . . . 61
4.4 Reduce and Map Over Pairs . . . 63
4.5 Sorting . . . 66
4.6 Experiments . . . 68
4.7 Summary . . . 69

5 Performance Prediction and Portability 73
5.1 Benchmarking the BSP Parameters . . . 74
5.2 Performance Prediction: A Case Study . . . 75
5.3 Performance Portability . . . 77
5.3.1 Reduce by Using Gather and Broadcast . . . 79
5.3.2 Reduce by Using All Gather . . . 79
5.3.3 Reduce by Tree Gather and Broadcast . . . 81
5.3.4 Reduce by Tree All Gather . . . 81
5.3.5 Best Algorithm Selection . . . 82
5.3.6 Variance: A Case Study . . . 83
5.4 Summary . . . 84

6 Formal Semantics of OSL 87

6.1 A Formal Programming Model . . . 87
6.1.1 Syntax . . . 88
6.1.2 Type System . . . 89
6.1.3 Operational Semantics . . . 89
6.2 Implementation in the Coq Proof Assistant . . . 91
6.2.1 Distributed Arrays . . . 91
6.2.2 Syntax and Typing . . . 93
6.2.3 Big-Step Semantics . . . 95
6.3 Verification of a Heat Diffusion Simulation . . . 98
6.4 Summary . . . 100

7 Conclusions and Perspectives **101**

A Advanced C++ Programming Techniques **107**
A.1 C++ Expression Templates . . . 107
A.2 Template Metaprogramming . . . 110
A.3 Move Semantics and Rvalue References . . . 111

B A Short Introduction to The Coq Proof Assistant **115**

C Machines used for testing OSL **119**
C.1 Mirev . . . 119
C.2 Speed . . . 119

Bibliography **121**

List of Figures

3.1 Distributed Array . . . 28
3.2 Map . . . 29
3.3 Zip . . . 29
3.4 Shift (right) . . . 30
3.5 Permute . . . 31
3.6 Gather . . . 32
3.7 Broadcast . . . 32
3.8 Balance . . . 33
3.9 Reduce . . . 33
3.10 GetPartition . . . 34
3.11 Flatten . . . 34

3.12 Summary of BSML Primitives . . . 35
3.13 Skeletons of the Prototype Library . . . 38
3.14 Distributed Arrays for Heat Diffusion Simulation . . . 43
3.15 Heat Diffusion Simulation . . . 44

4.1 Communications for Heat Diffusion . . . 61
4.2 One Step of Heat Diffusion Simulation in OSL . . . 62
4.3 N-Body Simulation: Code Excerpt . . . 65
4.4 Regular Sampling Sort . . . 66
4.5 Heat Equation Timings (dt=1) . . . 69
4.6 Heat Equation Timings (dt=0.0001) . . . 70
4.7 FFT Timings . . . 70

5.1 BSP Parameters of the clusters . . . 76
5.2 One Step of Heat Equation . . . 76
5.3 Performance Prediction of the 1D Heat Equation . . . 78
5.4 Gather and Broadcast . . . 80
5.5 All Gather . . . 80
5.6 Tree Gather and Broadcast . . . 81
5.7 Tree Gather in Pairs . . . 82
5.8 Variance Program . . . 84
5.9 Performance Portability of Variance . . . 85

6.1 OSL Typing Rules . . . 90
6.2 OSL Formal Programming Model . . . 92
6.3 OSL Syntax in Coq . . . 95
6.4 OSL Heat Diffusion Simulation in Coq . . . 99

Introduction

1

Contents

2.1 Classification of Skeletons 13
2.1.1 Data Parallel Skeletons 14
2.1.2 Task Parallel Skeletons 14
2.1.3 Control Skeletons 14
2.2 Skeleton Libraries and Languages 15
2.2.1 Contributions of Pisa Group 15
2.2.2 Contributions of Herbert Kuchen 19
2.2.3 BSP based Skeleton Libraries 20
2.2.4 C++ based Skeleton Libraries 21
2.2.5 Skandium 22
2.3 Other High Level Frameworks 22
2.3.1 Eden 22
2.3.2 STAPL 23
2.3.3 HPC++ 23
2.3.4 TBB 23
2.4 Discussion 24

The computer systems has matured by passing through the ages of vacuum tubes, transistors, integrated circuits, very large scale integration(VLSI) and the current generation systems. The trend towards Parallel computing dates back to 1950s with the introduction of IBM 704. Gene Amdahl was one of the principal architect in that project. Since then the parallel systems have passed through the various eras. The era of shared memory multiprocessors based supercomputing (60s - 70s), the introduction of new form of parallelism (Massively parallel processors MPP) in 80s by the Caltech Concurrent Computation Project with 64 Intel 8086/8087 off the shelf microprocessors. The MPP's were replaced gradually by the clusters in late 80s. And now the parallel systems are developed enough to become pervasive and become a part of our daily lives. At one end there are the parallel and distributed computers for high performance computing and at the other end there are multi-core desktops, notebooks,

laptops and smart-phones. This evolutionary advancement is the result of the efforts to solve the computationally intensive problems [95].

The chapter begins by giving a brief overview of today's parallel architectures and the most popular parallel programming models. As it is important to reason about the parallel programs, either algorithmically or semantically, the chapter also discusses parallel algorithmic models and formal models of parallel programs. The case for the need of more structured approaches towards parallelism is established by the parallel algorithmic models, and by assessing the widely used parallel programming models against some criteria. The chapter concludes by presenting the contributions and structure of the dissertation and the work already published in this regard.

1.1 Parallel Architectures

Now the parallelism can be found in everywhere in the modern day machines and at different levels. A number of flavours of machines falling in the different classes of Flynn [66] are in use today. On one hand the vector machines like Cray Y-MP, Convex C3880 based on the pipeline architecture are in use, while on the other hand there are systems based on the RISC architectures like PowerPCs of which the most known examples are the IBMs cell processor and the Blue Gene/L and the Blue Gene/P. Multi-core clusters are now the most commonly used architectures for the high performance computing. Machines based on the graphical processing units are now leading the top500 list Tianhe-I crossing the peta-flops barriers. Other than these very powerful machines there are reconfigurable architectures e.g FPGA (Field Programmable Gate Arrays). The fully FPGA based computer is relatively new but hybrid computers based on the CPU and FPGA are already present. One such example is Convey Computer Corporation's HC-1, which has both an Intel x86 processor and a Xilinx FPGA coprocessor. The introduction of LG Optimus 2X based on Nvidia's Tegra Processors marks a new era of multi-core mobile computing. Android has announced its dual core range in 2011 with Motorola Droid Bionic 4G and Motorola Atrix.

1.2 Parallel Programming Models

The software world has been very active through out the history of parallel computing. It has played a vital role in the evolution of the parallel systems. The parallel programs are harder to write as they have to take account of the communications and synchronisations required between different tasks/processes. With the course of time and the evolution of parallel hardware different parallel programming models have emerged.

The section focuses on models that are very popular for parallel programming which does not mean that there are really mainstream libraries as they are still complex. It also examine other popular ways used with mainstream object-oriented languages for parallel and distributed computing.

1.2.1 Architecture Oriented Models

The commonly used programming models are models inspired by architectural constraints: shared memory programming takes it roots in the shared memory machines while message passing takes its roots in the distributed memory machines. This is probably the most widely used category of the programming models.

Shared Memory Programming

As the name implies, all the processor can simultaneously access a single memory space where each memory location is given a unique address. The most commonly used are POSIX threads or Pthreads [106] and OpenMP [108, 45].

Pthreads Pthreads are the Portable Operating System Interface for Unix (POSIX) standard for threads. Historically, every hardware vendor implemented their own proprietary versions of the threads. Later a standard programming interface has been specified and become a IEEE POSIX 1003.n standard where n is a number. The Pthreads API provides the routines for thread Management (create, detach, join), mutexes (create, destroy, lock and unlock mutexes), condition variables and the synchronisation (read/write locks, barriers). These wide range of routines make the Pthreads library fairly comprehensive and offers enough control to the programmer over the threads. This makes it a low level API for thread programming where the programmer has to focus much on the thread specific code rather than the high level application logic.

OpenMP The Open multi-processing is a C, C++ and Fortran API for shared memory programming. It supports parallelism via the compiler directives and currently, it is supported by all the major compilers. The OpenMP Architecture Review Board (ARB) in 1997 release the first specification of OpenMP and now the latest (July 2011) specification of the OpenMP is in the version 3.1. In contrast to the Pthreads it offers a high level approach towards parallelism and abstracts a lot of low level work. The section of the sequential code that is meant to run in parallel must be preceded by a pre-processor directive that creates threads working independently. The work between the threads can be shared with each thread working on its part of code. To support the irregular parallelism, the latest standards of the OpenMP now offers the notion of the tasks [125, 17]. This extension offers more expressiveness but at the cost of a greater complexity.

Message Passing

A set of processes use their local memory for performing the computations. More than one process can reside on the same physical machine. The processes communicate with each other by the exchange of messages. The framework exists in the form of library calls for sending and receiving messages. A number of approaches for such style of programming converges to Message Passing Interface(MPI) [122] in

mid 90s. MPI has matured enough since then to become the de facto standard for the distributed memory machines. A number of implementations of MPI are available covering majority of the features of MPI1 and MPI2.

With the advent of multi-core architecture and the graphical processing units, the trend is now towards the hybrid style of programming. Hybrid programing is supported in multiple ways: Translating OpenMP programs [21] to MPI for the distributed memory machines, using MPI for shared memory architectures [72, 117], but the most promising approach in terms of computational and memory efficiency (but not in terms of simplicity) is to mix MPI and OpenMP for the multi-core clusters [114]. The intra-node operations are performed by the OpenMP and the inter-node communications are carried by the MPI.

1.2.2 Object Oriented Models

Threads and message passing programming [102] are possible with virtual machine based object oriented languages such as Java. However for distributed memory machines preferred approaches for this king of language are more distributed oriented, with a set of heterogeneous machines, rather that strictly parallel with more homogeneous nodes.

Other programming models are also maybe more popular with object oriented languages such as Java: agents and actors [83]. Agents in particular may be well fitted to program scientific computing applications [61, 62]. Actors are gaining popularity with the growing popularity of the Scala language [73] which use actor as the main model for shared memory parallel programming. Another abstraction to be noticed for distribution and concurrency for Java is asynchronous distributed objects [42, 20].

1.3 Parallel Algorithmic Models

It is important to be able to reason about algorithms without reference to a particular implementation using a given programming language. As opposed to sequential programming there is yet no agreement about an algorithmic model for parallel programming.

1.3.1 Network Models

In the network models, the focus is on the topology of the interconnection network of the parallel machine. There is a wide literature about algorithm design for specific topologies such as meshes, hypercubes, butterflies. The direct communication is possible only with the directly connected processors. Other indirect communications are routed along the path. The main strength of these models is the natural mapping of some of the algorithms over the topologies, e.g. parallel prefix on a tree, PDE on a mesh, and sorting or FFT on a butterfly. These models expose too much details of parallelism and lack portability (algorithms can not be mapped over other topologies).

1.3.2 Parallel Random Access Machine

PRAM is a natural extension of RAM introduced by Fortune and Willey [68]. It is one of the earliest and best-known model of parallel computation. It consists of a shared memory and a number of processors with their local memories. The processors operate synchronously and are controlled by a common clock. Any location can be accessed in a single instruction time. The PRAM is further classified in the following subclasses:

EREW Exclusive Read Exclusive Write: Different locations in the shared memory can be read by or written to exclusively by only one processor in the same clock cycle.

CREW Concurrent Read Exclusive Write: Same locations in the memory can be read by several processors but only written to exclusively by one processor in the same clock cycle.

CRCW Concurrent Read Concurrent Write: Same locations in the memory can be read by or written to by several processor in the same clock cycle.

PRAM is an over simplified model that ignores practical considerations like synchronisation and communications. This results in unreliable predictions of the execution costs. The complexity of a PRAM algorithm is given in terms of the number of time steps and maximum number of processors required in any one of those time steps.

1.3.3 Bridging Models

Bulk Synchronous Parallelism

The Bulk Synchronous Parallel (BSP) model [134, 105, 121, 30] describes: an abstract parallel computer, a model of execution and a cost model.

The BSP architecture. A BSP computer has three components: (a) a set of homogeneous processor-memory pairs, (b) a network allowing point-to-point inter processor communications, (c) a global synchronisation unit that performs synchronisation barriers.

Any general purpose parallel architecture can be seen as a BSP computer. For example a shared memory machine could be used in a way such as each processor only accesses a sub-part of the shared memory (which is then "private") and communications could be performed using a dedicated part of the shared memory. Furthermore in most cases the synchronisation unit is not a hardware unit but is rather emulated by software.

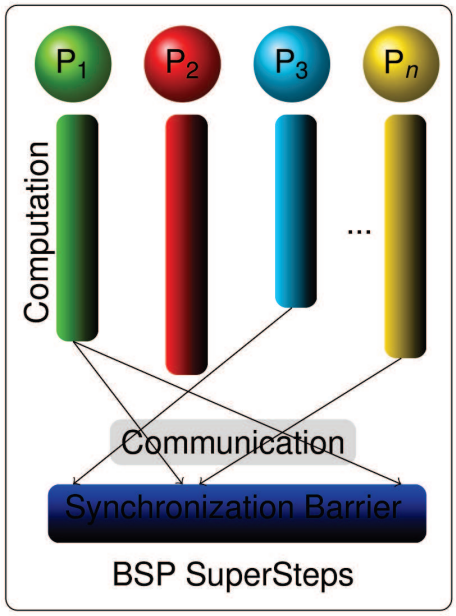
The performance of the BSP computer is characterised by four parameters (including the local processor speed) or three parameters (expressed as multiples of the local processing speed): **p** the number of processor-memory pairs ; **L** the time required for a global synchronisation ; **g** the time required for collectively delivering a 1-relation

(communication phase where every processor receives/sends at most one word), the network can deliver an h-relation (communication phase where every processor receives/sends at most h words) in time $g \times h$. These parameters can easily be obtained using benchmarks [78].

The execution model.

A BSP program is a sequence of *super-steps*. The execution of a super-step is divided into (at most) three successive and logically disjointed phases:

1. Each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes ;
2. The network delivers the requested data transfers ;
3. A global synchronisation barrier occurs, making the transferred data available for the next super-step.



The cost model. The execution time of a super-step s is thus the sum of the maximal local processing time, the data delivery time, and the global synchronisation time. It is expressed by the following formula:

$$\text{Time}(s) = \max_{0 \le i < p} w_i^{(s)} + \max_{0 \le i < p} h_i^{(s)} \times g + L$$

$w_i^{(s)}$ = local processing time on processor i during super-step s
$h_i^{(s)}$ = $\max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during super-step s.

The execution time $\sum_s \text{Time}(s)$ of a BSP program composed of S super-steps is, therefore, a sum of 3 terms:

$$W + H \times g + S \times L \text{ where } W = \sum_{s=1}^{S} \max_{0 \le i < p} w_i^{(s)} \text{ and } H = \sum_{s=1}^{S} \max_{0 \le i < p} h_i^{(s)}.$$

In general, W, H and S depends on the number of processor-memory pairs, on the size of data n, or on more complex parameters like data skew. The design of BSP algorithms is therefore a trade-off in order to minimise execution time by jointly minimise the number S of super-steps, the total volume H and imbalance of communication and the total volume W and imbalance of local computation.

The LogP Model

LogP [53] is a more elaborate model in which processors communicates by point to point messages. It predicts the time of network communication using a small set of machine parameters. These are the latency(L), overhead(o) in sending or receiving a message, the gap (g) between two consecutive messages, and the number of processors/memory modules P. According to LogP, the time of point to point communication can be estimated as: $L + 2o$. The variants of the LogP model, like LogGP [13] accounts the message size by introducing gap per byte G parameter, while the PLogP [85] slightly changes the definitions of the parameters and the overhead of senders and receivers and the gap are parameterized by the message size. [28] presents a comparison between the BSP and LogP models and concludes by preferring BSP over LogP.

1.4 Formal Models

To reason about the correctness of programs, possibly using interactive or fully automated tools, it is necessary to have formal models of the programming models.

It is know from a long time that reasoning about programs with shared memory is complex [109, 16]. Actually with the current multi-core processors, the hardware optimisations are such that it is even very difficult to model the informal semantics of the reference manual of processors such that the semantics corresponds to what is informally described and also corresponds to what is actually observed when running code on these processors [14, 118]. Therefore it seems very difficult to take into account memory models when programming with shared memory programming models such as Pthreads and OpenMP and be confident that the informally described memory model is the actual one. Moreover there are many different extensions of Hoare Logic [79] to reason about programs with pointers [110] and also several proposals to reason in the presence of concurrency primitives. It does not seems reasonable to require a user to use such formal notions for reasoning about her program. Still these formal models are necessary to proof that the implementation of more high-level programming models on top of low level ones are correct.

For message passing there exist work on the widely used MPI libraries [70, 71]: it belongs to the model checking domain, which means some properties may be verified automatically, but it is more difficult to verify a full correctness with respect to a specification with some computational parts.

In both popular shared memory programming and message passing models, the formal models are very complex.

1.5 Criteria to Evaluate Programming Model

Now when the parallelism is in the mainstream, the key challenge at this point in time is the transition of the software industry towards parallel programming. To address the transition challenge, it seems reasonable to assess the parallel programming models according to a criteria.

1.5.1 Criteria

Programmability The model should be available in a form well known to the vast community of sequential programmers. The idea is to free the sequential programmer from the burden of learning new tools/languages. The ideal is to deliver the model in the form of library of widely used programming language (C++, Java).

Portability With parallel machines coming in all flavours, the model should be portable to different architectures.

Performance Although pure efficiency is an important measure and should be preserved, but it is not the main criterium in mainstream programming. It can be traded off (not totally compromised) with the expressiveness and usability. In fact like the sequential programming (Big O asymptotic notation) the first main criterium related to the performance is the performance prediction. The second one is the portability of performance. Only the portability of code is not sufficient to guarantee the performance on changing architectures and data.

Proof of correctness As the parallelism is wide spread now and many of the critical systems are based over the parallel programming models, programming model should provide the proof of correctness to ensure the safety of the systems. Programming libraries should have a clear semantics and the correctness of programs could be verified.

1.5.2 Assessment of Programming Models

The most commonly used parallel programming models 1.2 are evaluated in the light of the criteria presented in the section 1.5. MPI and OpenMP are the two approaches most widely used in today's world. Although MPI and OpenMP have solved the issue of code portability, and partially addressed the programmability they are unable to predict the performance, can not guarantee the portability of performance and lacks the verification mechanism.

In fact the MPI and OpenMP lies at two different extremes in terms of the expression of parallelism. In MPI programmer need to mention all the details of parallelism while the simple constructions of OpenMP abstract almost everything from the programmer (it is no longer true for a more advanced use). Both the ways of expression has advantages and drawbacks. The explicit expression of parallelism (MPI) gives on one hand more control over the parallelism and much optimised program can be written but the approach makes the life difficult for the programmer. The programmer can not only concentrate on the business logic of the program, instead he needs to focus on the ways the processes communicates there data and when and how they can be synchronised. OpenMP on the other hand is much more expressive and with negligible effort (in comparison to MPI) the programmer can express the parallelism

in simple cases. But the programmer can not intervene to optimise his program, without facing concurrent memory access problems that make the use of OpenMP closer in complexity to MPI.

Another important issue with these models is that the programmer can not observe the underlying pattern. They lack in providing a global view of the program which is required to apply the optimisations other than the primitives (domain specific optimisations).

1.6 Structured Parallelism

Limitations of the existing approaches to parallelism 1.5.2 create a need to express the parallelism in a much structured manner and satisfy the criteria 1.5. Murray Cole in [49] proposed such a structured model and named it "Parallel Algorithmic Skeletons". He states:

> *"The structured approach to parallelism proposes that commonly used patterns of computation and interaction should be abstracted as parameterisable library functions, control constructs or similar, so that application programmers can explicitly declare that the application follows one or more such patterns"*

This is important as the programmer can introduce the parallelism in a much expressive way without compromising the efficiency and even with a chance to apply the domain specific optimisations.

Skeletons belong to a finite set of higher-order functions or patterns that can be run in parallel. Usually the programming semantics of a skeleton is similar to the functional semantics of a corresponding sequential pattern (for example the application of a function to all the elements of a collection) and the execution semantics remains implicit or informal. Thus the skeletons abstract the communication and synchronisation details of parallel activities. To write a parallel program, users have to combine and compose the existing skeletons.

Skeletons are not in general any parallel operations, but try to capture the essence of well-known techniques of parallel programming such as parallel pipeline, master-slave algorithms, the application of a function to distributed collections, parallel reduction, *etc*.

Since the introduction of skeletons a number of frameworks are developed but most of them share the limitations of other programming models in satisfying the criteria 1.5. Many of the skeleton frameworks are developed in the form of functional languages because of the natural match between the skeletons and the higher order functions of the functional programming. And this is the reason of their non acceptance in the sequential programmers community. The other frameworks that address this issue fails to offer a mechanism for performance prediction, performance portability and proof of correctness.

It is to be noted that the skeletons are the programming model but not a cost model to estimate the cost of the parallel programs. Several parallel cost models exists in literature and among them one of the most accurate is the Bulk Synchronous Parallelism (BSP) 1.3.3 as it not only counts the computations but also take account of the communications and synchronisations. The advantage of BSP is that it can not only be used to predict the performance but it also offers a way to port the performance.

So, together the parallel algorithmic skeletons and BSP constitutes a structured approach to satisfy the limitations of the other models. The Orléans Skeletons Library (OSL) offers a structured approach towards the systematic development of parallel programs by combining the advantages of parallel algorithmic skeletons with the cost model of the BSP.

1.7 Contribution and Structure of the Dissertation

The contribution of the thesis is the Orléans Skeleton Library or OSL, a library that offers structured parallelism.

- OSL improves programmability by working with the latest standards (C++0x) and libraries(Boost Libraries). It offers expressiveness both to the sequential programmers and to the skeleton experts for the development of new skeletons. The programmability of OSL does not come at the cost of efficiency.
- OSL accommodates non-evenly distributed arrays by adding few skeletons to its arsenal.
- OSL follows the pure BSP model based performance prediction.
- OSL offers portability of performance by comparing the cost of the underlying algorithms using the BSP cost formula.
- OSL comes with a formal programming model, allowing formal verification of programs developed with OSL.

The work focusing on different aspects of the OSL has already been published in the proceedings of the several conferences:

- Noman Javed and Frédéric Loulergue. OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays. In Y. Don, R. Gruber, and J. Joller, editors, *8th international Conference on Advanced Parallel Processing Technologies (APPT'09)*, LNCS 5737, pages 436-451. Springer, 2009
- Noman Javed and Frédéric Loulergue. Parallel Programming and Performance Predictability with Orléans Skeleton Library. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 257-263. IEEE, 2011

- Noman Javed and Frédéric Loulergue. A Formal Programming Model of Orléans Skeleton Library. In Victor Malyshkin, editor, *11th International Conference on Parallel Computing Technologies (PaCT)*, LNCS 6873. Springer, 2011
- Noman Javed, Frédéric Loulergue, Julien Tesson, and Wadoud Bousdira. Prototyping a Library of Algorithmic Skeletons with Bulk Synchronous Parallel ML. In *The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*, pages 520-526. CSREA Press, 2011
- Noman Javed and Frédéric Loulergue. Verification of a Heat Diffusion Simulation written with Orléans Skeleton Library. In *9th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, LNCS. Springer, 2011, to appear.

The dissertation is structured in a way to present how OSL addresses the criteria specified in section 1.5. Next chapter presents the review of the state of the art in skeletal and other related high-level parallel programming frameworks and presents their comparison along different axes. Chapter 3 explains in detail the parallel algorithmic skeletons packaged in OSL, their prototype implementation using BSML to get the insights in the semantics not obvious otherwise, and in the end presents in detail the implementation in C++. Chapter 4 elaborates, with different example applications, the process of application development with OSL. It also presents the comparison case studies with other skeleton libraries. Follows a chapter 5 which explores the ability of OSL to predict the performance of the applications. It presents the benchmark program for the computation of machine parameters used in performance prediction and also the performance portability. A case study application is presented to highlight the portability of performance. Chapter 6 presents the formal programming model of OSL and details the verification of an example developed with OSL. Conclusions and perspectives are developed in chapter 7.

For the convenience of the reader, the appendix contains short chapters that respectively introduce C++ advanced programming techniques (Chapter A) and the Coq proof assistant (Chapter B).

State of the Art

2

Contents

3.1 Data Parallel Skeletons . . . 28
3.1.1 Distributed Arrays . . . 28
3.1.2 Map Skeletons . . . 28
3.1.3 Communication Skeletons . . . 30
3.1.4 Reduce . . . 33
3.1.5 View Changing Skeletons . . . 33
3.2 A Prototype Implementation in BSML . . . 35
3.2.1 Bulk Synchronous Parallel ML . . . 35
3.2.2 A Library of Algorithmic Skeletons . . . 38
3.2.3 Heat Diffusion Simulation Example and Experiments . . . 43
3.3 A C++ Implementation of OSL . . . 45
3.3.1 Distributed Arrays . . . 45
3.3.2 Skeletons . . . 47
3.4 Summary . . . 54

The chapter after introducing the basic skeletons presents the state of the art in the domain. The literature covered in this chapter is not exhaustive, but the work is presented in terms of the evolution of the frameworks. Most of the skeleton frameworks form the families as they are developed by the same group of people to overcome the shortcomings of the previous ones, or to address some other challenges. Some of the recent and active work is considered as well. The idea is to give insights in the state of the art that influence the decision making for the design of OSL.

2.1 Classification of Skeletons

The Parallel Algorithmic Skeletons is a topic of continuous research since its introduction by Murray Cole [49]. The major categories of skeletons evolved through out these years are the data parallel, the task parallel and the control skeletons. The skeletons introduced in the following classes do not form a complete set of skeletons but are the very basic ones.

2.1.1 Data Parallel Skeletons

The skeletons belong to this class work over some data structure. They abstract both the computational and communication patterns. The main skeletons belonging to this class are:

Map Map distributes the data among the participating processors. Each processor applies the supplied function over its share of data. The function supplied to the Map is the simple sequential function. The function is applied independently to every element, hence no communication is needed during the execution of the pattern. The result of the Map is a collection.

Zip Zip is the generalisation of the Map operating over two lists in place of one. The supplied function takes two elements as its arguments one from each list.

Reduce Reduce applies the reduction operator over the input list. Unlike Map and Zip, the result of the Reduce operation is a single element. While reducing the list, the communications are needed after the computation of the local results. The local results are then further reduced and broadcast if required.

Permute Permutes takes a function working over the index of the data. The function, for each index, returns the destination index. The data is then communicated accordingly. Depending upon the permutation function the communications might be needed.

Shift Shift is a specialisation of the permute, permuting the data one place to the right or left. The boundary elements need to be communicated with the neighbouring processors.

2.1.2 Task Parallel Skeletons

The task parallel skeletons distributes the tasks among the participating processors. Each processor then executes the task assigned to it over the data. The data may be distributed or it may be a stream of data.

Pipeline Pipeline, as the name implies, creates different stages each one corresponding to a different task. The data is then streamed through these stages such that the output of one stage becomes the input to the next one.

Farm Farm creates a pool of workers and divide the work among them. The results from the workers are then merged back. The term is sometime used interchangeably with the divide and conquer skeleton.

2.1.3 Control Skeletons

The skeletons belonging to this category controls the execution in some ways. The skeletons are applied to control the execution of the other skeletons.

For The iterative skeletons working in a similar fashion to its sequential counterpart the traditional for loop.

If Controls the execution of some other skeletons based on the result of the condition.

2.2 Skeleton Libraries and Languages

2.2.1 Contributions of Pisa Group

The Pisa group is one of the oldest and the most active group in the skeletons research. The contributions of the group are immense and almost all the later skeleton frameworks are influenced by their work. The principle investigators belonging to the group are Marco Danelutto, Susanna Pelagatti, Marco Aldinucci and Marco Vanneschi. The foremost contribution of the group is P3L: Pisa Parallel Programming Language. Though P3L influence the development of many of the frameworks its direct family includes SkIE, Skelib, Eskimo, Lithium, ASSIST and Muskel.

P3L: 1992

P3L [111, 112, 18] is the parallel coordination language of the basic parallel paradigms (skeletons/modules). Three types of modules are provided in the P3L language, the sequential module, the main module and the parallel module. The former module provides a structured way of the integration of the sequential code in P3L's application. The main module is the application starter. The parallel module is supported in the form a set of basic parallel paradigms from task parallel to data parallel skeletons. The parallel application is structured by the hierarchical composition of these basic paradigms.

A P3L compiler [46] translates the high level program specification into a set of implementation templates according to the target architecture. The compiler is supported with the template library, optimisation library, reduction library and a process-template libraries for the generation of a portable and optimised code. The compiler first translates the parallel specifications in a process network using the template library. The process network is then globally optimised using the optimisation rules presented in the optimisation library. The optimised process network is then reduced for further optimisations using the reduction library. The process network in than scanned to extract the process template from the process-template library. This final process network in then tuned and forwarded to generate the architecture specific code.

The basic skeletons provided in P3L are the pipe, farm, reduce, map, loop and seq. A comp skeleton is provided for the composition of the different data parallel skeletons [56]. The target language of the P3L compiler is C.

SkIE: 1998

SkIE (Skeleton-based Integrated Environment) [19] can be considered as the industrial outcome of the advancement in P3L. The basic objective of the SkIE is to provide a industrial standard heterogeneous environment for high performance computing application development. Fast application development and good performance are the key objectives of the industry. To achieve this the SkIE system is comprised of the Graphical user interface (VisualSkIE), SkIE coordination language (SkIECL), debugging and the performance analysis tools.

For rapid application development the integration of the sequential user code in the languages like (C, Fortran and Java) is supported. The integration of the parallel modules developed using standards like MPI and HPF are supported as well.

The application development in SkIE starts by specifying the parallel structure of the application in VisualSkIE. This phase is followed by the global optimisation and code generation phases in a similar fashion to P3L. During the debugging phase the parallel part of the application is skipped as it has already been verified by the SkIE implementors. The debugging of the sequential part of the user code is very much similar to the sequential debugging and is carried out by the debuggers like xxgdb, DDD. The performance analysis allows transparent inspection of the program. User can inspect the problematic areas and bottlenecks in the code through the visualiser of the performance data.

The skeleton set supported by SkIECL is same as of in P3L.

SKElib: 1999

The goal of the SKElib[57] is to provide the unix programmer with a familiar framework to structure parallel applications in the form of skeletons. To achieve this goals the skeletons in SKElib are implemented in simple C using TCP/IP Unix sockets to implement inter-process communications. The other purpose of using TCP/IP unix sockets is to accommodate the ad-hoc parallelism that can not be captured by the skeletons.

The coordination language is no more needed and the skeletons are packaged in the form of a library. The skeletons are declared and composed in the form of simple C functions. The same idea of P3L and SKIE for template implementation of the skeletons is used. The skeletons nesting is allowed. The call to skeletons are evaluated by a call to a library function SKE_CALL. User functions to be executed by the skeletons are the simple void C function with two parameters: the pointer to input data and the pointer to output data. Processes are created by the SKE_CALL function following the SPMD execution model. The optimisations can be performed by explicitly providing the OPTIMISE flag to the SKE_CALL function.

SKElib is one of the first framework which contributes the skeleton parallelism in the form of a library in C.

Eskimo: 2002

Eskimo: Easy SKeleton Interface (Memory Oriented) [6], is designed to address the irregular problems and the dynamic data structures. It is the parallel extension of C based on Shared address programming model targeted for beowolf clusters. The philosophy behind the application design in Eskimo is to provide programmer with the ability to co-design the data structure along with the algorithm. The other details like process scheduling and the load balancing will be taken into account by the Eskimo run-time.

The parallelism can be introduced in the form of asynchronous fork/join calls called e-call/e-join. These primitive split the basic program control flow which will later converge back to the main flow. e-call/e-join primitives enable the programmer to set up a dynamic and variable number of e-flows, an important feature when dealing with dynamic data structures. The two e-flows coming out from an e-call may be executed in parallel, interleaved or serialised in any order depending on the algorithm, the input data and the system status. These e-flows are mapped to the processes/threads by the Eskimo run-time. e-foreach/e-joinall are provided as a generalisations of the e-call/e-join. They can create and destroy an arbitrary number of e-flows.

The Eskimo language is specifically designed for distributed memory architectures. These architectures naturally supply a very efficient memory access to local memory and a more expensive access to remote memory. Eskimo exposes to the programmer two memory spaces: private and shared. All the variables belonging to a particular e-flow are private and the shared variables, the containers like arrays and trees can be shared among the e-flows. The global variables in the C language are not allowed in Eskimo and should be implemented as shared variables.

e-foreach constructs introduce the data-parallelism (map and divide and conquer) in Eskimo. Three variants of the e-foreach are provided to run through each element of the shared container. Both form of the data parallelism can be freely interleaved. If the application design doesn't fit these two paradigms, programmer can accommodate the ad-hoc parallelism by the free use of the e-call/e-join.

Lithium: 2002

Lithium offers the structured parallel programming in Java [58, 12]. It was the first library of skeletons in Java. It was the first one based on the macro data flow implementation technique.

The skeletons provided by Lithium are pipeline, task farms, iterative and data parallel skeletons. All the skeletons work on the data streams. It allows the nesting of skeletons. The operational semantics of the skeletons are provided in [10]. The Java RMI is used to distribute the computations among different processing elements. The object orientation in Lithium make the debugging task of the functional application by emulating it on a single machine. The architectural design of Lithium also allows relatively easy extension of the skeleton set.

The stream parallel optimisations are implemented in the Lithium [9]. Every stream parallel computation can be transformed to a normal form with the better or the same performance. The equivalence of the normal form and the non-normal form is derived from the functional semantics of the skeletons. These functional semantics are derived from the operational semantics of the skeletons. The performance relationship between the raw form and the normal form is derived from a LogP like model taking into account both the computation and the communication time. The nesting of the data parallel computations with the stream parallel workers are optimised as well. The optimisation regarding the minimisation of the resources is provided as well.

The skeleton program is processed to obtain the macro data flow (MDF). The data flow (arcs of the MDF) can be derived by looking at the structure of the skeletons nesting. The resulting graphs have a single MDF instruction getting the data from the input stream and a single MDFi delivering the data items to the output stream.

ASSIST: 2003

The objective of the ASSIST [135, 7] framework is to provide an Invisible Grid like platform [8]. The ASSIST provides a layered architecture to achieve this objective. At the application layer level, the programmer is provided with a structured coordination language for the abstraction of parallel programs. ASSIST coordination language programs are build of two specific parts: a module graph modelling the interactions among the modules both sequential and parallel, and a set of modules. The modules can be programmed as sequential or parallel, the former are the procedure-like wrappings of the C code and the later are instances of the ASSIST parmod (parallel module). The compiler layer and the run-time layer compiles and generates the object programs for the underlying grid layer. A Grid abstract machine layer (GAM) decouples the compiler and run time layer from the actual Grid middleware used, providing a suitable interface/API to the mechanisms used in the ASSIST framework for code and data staging, remote commanding, communications and synchronisations, etc. Globus grid and POSIX grid frameworks are supported. ASSIST can accommodate heterogeneity in both the previous mentioned Grid frameworks.

Muskel: 2005

Muskel [11] is the Java based skeleton library derived from Lithium. The skeleton set in Lithium was fixed and it is not easy to accommodate unstructured parallelism or the creation of the new skeletons [55]. The Muskel is designed with the objective of extensibility (addressing unstructured parallelism) in mind. Muskel is based on the data flow model like Lithium. The source program is parsed into a data flow graph representing the skeleton tree. For each of the input tasks, a copy of the data flow graph is instantiated, with the task appearing as an input token to the graph. The new graph is delivered to the distributed data flow interpreter "instruction pool". The distributed data flow interpreter fetches fire-able instructions from the instruction

pool and the instructions are executed on the nodes in the target architecture. User-defined, possibly unstructured parallelism exploitation patterns can be programmed by explicitly defining data flow graphs. These data flow graphs can be used in the skeleton system in any place where predefined skeletons can be used, thus providing the possibility of seamlessly integrating both kinds of parallelism exploitation within the same program. User can add new skeletons by defining the data flow graphs for these skeletons. The Muskel supports the stream parallel skeleton set of the Lithium. The annotations and the aspect oriented techniques are used to provide the control of the non functional features like code security, source to source optimisations etc to the user.

The family clearly demonstrates the evolution of the skeletons with the passage of time. SkIE enhances the ancestor (P3L) to provide a industrial outcome, Skelib omits the need of a coordination language and becomes the first one to provide the skeletal framework in the form of library. It further addresses the problem of learning new language by supplying the library in the C language. Eskimo handles the irregular parallelism and also becomes the first skeletal framework on shared memory architectures. Lithium is the first one to provide skeletons in Java, exploit macro data flow, and admits the formal specifications. ASSIST using its skeletons and coordination language provides a platform for the development of the distributed high performance applications over grids. The deployment of the applications is handled through the known grid middle-ware. Muskel(a java library) enhances the Lithium to address the unstructured parallelism.

The evolution of this family can be considered as the evaluation of these frameworks in the prism of the Cole's manifesto [50].

2.2.2 Contributions of Herbert Kuchen

The research begins with the development of Skil and matured in the form of Muesli.

Skil

The aim of the Skil (Skeletons Imperative Language) [34, 33, 36] is to provide efficient implementation of the structured parallelism. Skil is an imperative language and it is basically an extension of the C language with features from functional programming to support skeletons. The skeletons are implemented in the form of the higher order functions. Partial application of the functions is supported by implementing currying. The arrays are used as the distributed data structures and the skeletons are implemented for this data structure. Skeletons are implemented as polymorphic functions. The polymorphism is implemented in the form of polymorphic type system using type variables.

The skeleton set of the Skil consists of three categories of the skeletons: The skeletons related to the creation, destruction and the copying of the arrays, The data parallel computational skeletons like map, fold and the communication skeletons like

permute, array-broadcast-part. The communication skeletons are mostly affected dynamic data structures like the arrays or variable size array [35]. The issue is addressed by providing a language level support in the form of packing / unpacking of the data. The performance of the Skil was demonstrated by developing different applications emphasising on some aspects of the language.

Muesli

The goal behind this project is to integrate the existing work on the skeletons, as there were many groups working in this area with their own approaches. The other issue is that many of the frameworks are provided as an independent language or an extension to the existing language. This is the problem with Skil as well. A compiler is needed to translate the Skil programs to the normal C programs. The issue is addressed by implementing skeletons in the form of a library "Munster Skeleton Library" using C++ [87]. To implement the skeletons in C++ the features from the functional programming are required to be implemented in C++. The C++ template mechanism is exploited to implement elegantly skeletons in the form of polymorphic higher order functions. The partial application of the function in the form of currying is supported as well [91, 86, 92]. The serialisation is implemented to cope the problem of irregular structures. The integration of the task parallel and data parallel skeletons is supported [89].

The underlying parallelism is implemented using MPI. The Distributed Arrays and the Distributed Matrix are the basic data structures. The skeletons are defined for both these data structures. The work on scalability of the skeletons and the domain specific optimisations [88] is the part of the enhancement in the library. A multicore version of the library is developed as well [47].

This family brings the skeletal parallelism to the C/C++ community. Features from the functional languages like currying, higher order functions are implemented in C and C++ to provide a natural transition towards skeletons in imperative/object oriented languages. The features missing in these languages are the, lack of a cost model which makes performance prediction difficult, performance portability and the formal specifications of the model.

2.2.3 BSP based Skeleton Libraries

The section presents the work based on the performance prediction for the skeletal frameworks using BSP model.

Skel-BSP

Skel-BSP [141] is the work of Andrea Zavanella focusing on the skeletons based on the Bulk Synchronous parallelism model. The main objective of the thesis is to highlight the performance portability of the skeletons using the BSP cost model. Both the data

parallel and stream parallel skeletons are provided in Skel-BSP. The skeletons are optimised both at the local level [120, 139, 142] and at the global level [140]. Global level means optimising the combination of the skeletons. A. Zavanella used EdD-BSP an extension of the pure BSP model. The Skel-BSP compiler choose the best implementation template of the skeleton among various templates, based on the performance equations and the system parameters captured by the EdD-BSP model.

Skel-BSP is the very first to demonstrate the use of BSP to port the performance. Skel-BSP optimises both the implementation of the skeletons and their combinations. The other work regarding the performance prediction using BSP was done by Murray Cole [51] and a VEC-BSP language was developed. Their approach is to statically predict the performance using BSP and shape based methodologies. BSFC++ [54] is a library for functional bulk synchronous parallel programming in C++. Although it is not a skeleton library but it offers some of the collective operations. And as it is purely based on the BSP model it can predict the performance of the application.

2.2.4 C++ based Skeleton Libraries

A number of skeletal frameworks exists, but in this section we present the three having the similarities with OSL and developed with C/C++.

eSkel

It is a library of algorithmic skeletons in C built over MPI. eSkel [50] is designed to ease integration of algorithmic skeletons programs within MPI code and to avoid to put to many constraints on the user. Being closer to MPI, the signatures of eSkel's skeletons are more complicated than the signatures of other skeletons libraries. [23] presents the second version of the library. In [22] explains the nesting and interaction modes and compares the existing libraries with respect to these modes.

SkeTo

SkeTo [104, 63] is the C++ library of data parallel skeletons. The skeletons in SkeTo are developed for matrices and trees in addition to the distributed lists. Moreover, the skeletons for the variable length lists are provided as well [124]. The work for the optimisation of the skeletons have been done along different axes. One such optimisation is the proposition of the Diff skeleton [4] based on the diffusion theorem. The Skeletons following a specific recursive form are translated to the Diff skeleton. The efficient implementation of the Diff skeleton was provided. The latest research in the development of the library was focused around the optimised implementation of the library using the expression templates [103]. [84] presents the version of the library for the multi-core systems.

Quaff

Quaff [65] is one of the latest and best optimised libraries in the domain. It relies heavily on the C++ meta-programming techniques for optimising its skeletons. Quaff supports the stream parallel skeletons. It also provides the possibility of expressing the data parallel skeletons through the farm skeleton. The composition and the nesting of the skeletons is supported. The process network is generated and the tasks are mapped to the process network using the production rules. The communication strategy for each node is then generated. The Quaff can optimise the skeletons appearing in some specific patterns. The optimisations are performed at compile time using meta programming, thus decreasing the run-time cost of the algorithm. A formal model for the Quaff skeleton system is presented in [64]. The article further describes the implementation of the formal model in terms of C++ meta-programming.

2.2.5 Skandium

Skandium [98] is a skeleton library for multi-core systems developed in Java. The shared memory algorithmic model of the Skandium is based on the skeletons like Seq, Farm, Pipe, If, For, While, Map, Fork and D&C, and the muscles. The later represents the sequential functions applied inside the body of the skeletons and are used to implement the business logic. The Skandium is the successor of Calcium, a library based on the research of Mario Leyton's thesis work [96].

2.3 Other High Level Frameworks

The section presents some of the high level frameworks that are not the skeletons but have similarities with the skeletons.

2.3.1 Eden

Eden [39, 40] is a parallel functional language which extends Haskell for the creation and instantiation of the parallel processes. These extensions allow the easy definition of skeletons as higher order functions. It supports parallel programming at two levels. At process level by creating the processes through the recursive definitions and at the higher level using the skeletons. Unlike most of the skeleton frameworks Eden supports irregular parallelism and its process model provides direct control over process granularity, data distribution and communication topology. A run-time environment is developed for communications and synchronisation. [100] presents the sum up of the Eden project. The operational semantics of the Eden programming language are detailed in [76]. The development with Eden is presented using a number of applications like FFT [24] and google map reduce [25] framework in Eden.

2.3.2 STAPL

The goal of the STAPL [115, 15, 41] (the Standard Template Adaptive Parallel Library) is to provide parallel version of the STL (Standard Template Library). The library is designed to work on both the shared and distributed memory systems. The pContainers(distributed data structure), views, pAlgorithms(parallel algorithms) are the parallel counter parts of the STL containers, iterators and algorithms respectively. STAPL provides the appropriate selection of the algorithm depending upon the system and the data parameters [131]. This adaptivity in STAPL is based on the empirical techniques based on the previously collected data regarding the system. The communications in STAPL are abstracted by ARMI (Adaptive Remote Method Invocation) communication library [130]. It supports both the blocking RMI and non-blocking RMI. ARMI can be used independently of STAPL by expressing fine grain parallelism and mapping it to the underlying shared memory or message passing system. Low level details like scheduling the incoming communications and aggregating outgoing communications are handled by ARMI and can be tuned depending upon the parameters of the system. A number of applications like sorting and parallel protein folding [132] presents the expressiveness of STAPL.

2.3.3 HPC++

High Performance C++ [82] is based on the same approach as STAPL. It consists of classes and templates to support synchronisation, collective parallel operations and remote memory references. It also includes the parallel implementation of the standard template library. The HPC++ supports multi-threaded shared memory and SPMD style of programming.

2.3.4 TBB

Intel threading building blocks [116] offers an expressive way of taking advantage of multicore performance without being a threading expert. It is more than a threading replacement in fact Intel TBB could be seen as a skeleton library: it offers a kind of map, and also reduce and scan parallel algorithms. It provides many useful classes, from different kinds of mutexes and atomic operations, through thread-safe, concurrent scalable containers and memory allocators, till sophisticated task scheduler. Programmers using TBB can turn the execution of loop iterations parallel by treating chunks of iterations as tasks and allowing the TBB task scheduler to determine the task sizes, number of threads to use assignment of tasks to those threads, and how those threads are scheduled for execution. The task scheduler will give precedence to tasks that have been most recently in a core with the idea of making best use of the cache that likely contains the task's data.

2.4 Discussion

The skeletal frameworks and other well known higher level frameworks are presented in terms of evolution or in terms of relevance with our work. Many of them have similarities with OSL and influence the design decisions of the OSL. eSkel [50] is designed to ease integration of algorithmic skeletons programs within MPI code and to avoid to put to many constraints on the user. Being closer to MPI, the signatures of eSkel's skeletons are more complicated than the signatures of other skeletons libraries. SkeTo and Muesli share with OSL a number of classical data-parallel skeletons on distributed arrays. SkeTo is much richer in terms of the data structures and the skeletons over them, SkeTo lacks performance prediction, performance portability and formal model. Quaff is highly optimised with meta-programming techniques. However to attain its very good performances, some constraints are put on the algorithmic structure of the skeletons (that are mainly task parallel skeletons). Quaff is based on a formal model. The performance prediction and performance portability are the lacking features in the library.

One of the main strength of the BSP programming model is prediction of performance [77]. Most of the research regarding the performance prediction of parallel programs is based on the BSP programming model [78, 32]. VEC-BSP and Skel-BSP are also based on (extensions of) the BSP model. Our approach differs from Skel-BSP in several aspects. At global level, we consider the performance portability as an optimisation problem. And by using distributed array and the expression templates technique we do not need a transformation system as did by A. Zavanella. The other difference is that SKEL-BSP library by A. Zavanella needs a dedicated compiler and he implements a compiler to perform the transformation. We do not need any compiler. A. Zavanella follows an extension of BSP model while we use the pure BSP model. In [99] the authors do not use the BSP cost model. Instead they calculate the performance penalty or overhead of parallelisation for estimating the performance.

Another related area to the performance portability is automatic tuning. Most of the work in the area is empirical and is focused on determining different parameters to tune the collective communications. In [133] the authors focus on determining the optimum parameters of the system. In [127] use performance tuning to improve the performance. Their goal is to minimise latency for small message sizes and minimising the bandwidth for large messages. Other studies that focus on automatic tuning for MPI collectives are presented in [5, 107]. The main difference between all these studies and ours is that they focus on the experimental determination of optimal parameters to fine tune the collectives. This requires and lot of experimentation and computational time. Further it lacks any mathematical foundations to predict the performance. A much similar study to ours in terms of automatic tuning is conducted in [113]. It uses the Hockney, LogP and PLogP models for the tuning of the application. The literature on the parallel computational models reveals that the BSP is more realistic than all of these models. That is why we base our study on BSP model.

Some other work focus on algorithmic skeleton libraries, to our knowledge none is formalised and the properties of the semantics verified using a proof assistant. [60]

presents a data-parallel calculus of multi-dimensional arrays, but it is a formal semantics without any related implementation. The Lithium [12] algorithmic skeleton library for Java differs from OSL as it is stream-based. [10] proposes in a single formalism a programming model and a (high-level) execution model for Lithium. The skeletons of [64] are also stream-based but the semantics is used rather as a guideline for the design of the meta-programmed optimisation of the skeletons in C++.

The semantics of the Calcium library is described in [43] and further extended in a shared memory context to handle exceptions [97]. In [43], the focus in on a programming model semantics (operational semantics) as well as a static semantics (typing) and the proof of the subject reduction property (the typing is preserved during evaluation). In this work the semantics of the skeletons are detailed, but not the semantics of what the authors call the "muscles" i.e. the sequential arguments of the skeletons (the semantics of the host language of the library, in the particular case Java). The set of skeletons of Calcium includes a set of task parallel skeletons, which contains, among others, skeletons that give a sequential control but at the global level of all the parallel program. These skeletons are parallel because their branches or bodies are parallel (conditionals and while/for loops). In OSL we mix the skeletons with the usual constructs of the host C++ language to write the sequential control flow at the global level of the parallel program. The remaining skeletons in Calcium are data-parallel skeletons including map, and divide-and-conquer skeletons. The map skeleton, for example, is however different from our map. The OSL map is more similar to map functions in functional programming as it takes two arguments: a function f to be applied to each element of the collection l which is the second argument. In functional programming this collection is a list, in OSL it is a distributed array. In Calcium the map skeleton takes two additional functions: one that describes how the input collection is cut into pieces and another function that describes how the pieces (obtained by applying f to the previous pieces) are combined together to form the output collection.

To our knowledge, none of these semantics have be used as the basis of programs proofs of correctness, but are rather formal reference manuals for the libraries. Moreover, none have been formalised using a proof assistant.

Program calculation [29, 80] is the usual way to prove the correctness of algorithm skeletons programs (by construction), the proof being done "by hand". However a new algorithmic skeleton called BH as been implemented in Bulk Synchronous Parallel ML (see section 3.2.1) and its implementation proved correct using the Coq proof assistant. This BH skeleton is used in a framework for deriving programs written in BH from specifications [69], and to *extract* BSML programs that could be compiled and run on parallel machines. A heat diffusion simulation directly written in BSML (therefore not at all in the algorithmic skeleton style) has also been proved correct and the BSML program extracted from the proofs [126].

OSL Design and Implementation

3

Contents

4.1 A First Example . . . 55
4.2 Heat Equation: A case study . . . 57
4.2.1 One Dimensional Heat Equation . . . 57
4.2.2 Two Dimensional Heat Equation . . . 60
4.3 Fast Fourier Transform . . . 61
4.4 Reduce and Map Over Pairs . . . 63
4.5 Sorting . . . 66
4.6 Experiments . . . 68
4.7 Summary . . . 69

Orléans Skeletons Library(OSL) is a library of parallel algorithmic skeletons. The library is implemented in C++ and the message passing interface (MPI) is used as the communication mechanism. The skeletons in the OSL are based on the Bulk Synchronous Parallelism (BSP) model. The OSL provides data parallelism through its skeletons. The goals of the OSL are:

- To ease the programming of the parallel machines by abstracting the low level parallelism details.
- To provide a library of pure C++ functions which can be executed by the standard compilers.
- To provide an extensible approach so that new skeletons can be added with ease.
- To stick the skeletons with a cost model making the prediction and portability of performance possible.
- To provide a formal semantics making possible the verification of programs.

OSL is designed in a way to address the above mentioned goals. For the better understanding of the design concepts and the implementation issues, the chapter starts by presenting the implementation independent generic definitions of the skeletons used in OSL, followed by the details of the prototype version implemented in BSML. Finally the implementation of the latest version of OSL in C++ is presented.

3.1 Data Parallel Skeletons

The section presents the generic definitions of the skeletons packaged in OSL. The data structure at which these skeletons are operating is considered to be distributed among the processors as is the case in OSL. For the better comprehension, the data structure which happens to be a distributed array is explained here first.

3.1.1 Distributed Arrays

The data structure to be operated by the data parallel skeletons is distributed array. The array is evenly distributed among the processors. The uneven distribution of the distributed array is possible depending upon the constructor or the acting skeletons that may create in-balance. To ease the understanding of the readers, an almost evenly distributed array is considered and is presented in the figure 3.1. The numbers from 0 to 16 in the figure represent the global index of the element, while the local index (not mentioned here) always starts from zero.

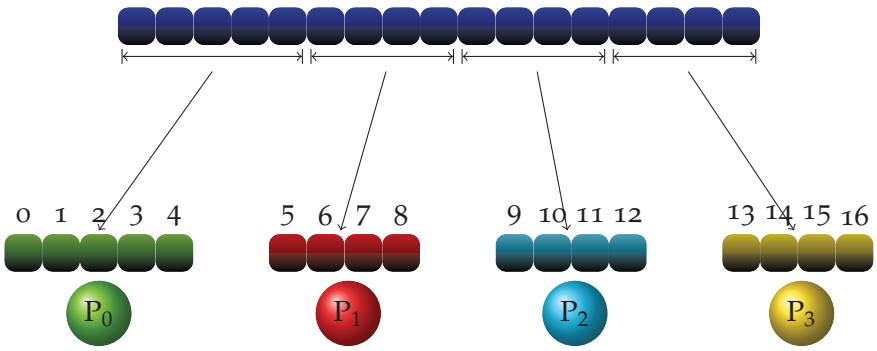


Figure 3.1 – *Distributed Array*

3.1.2 Map Skeletons

All of these skeletons are the variants of the Map skeleton.

Map Map takes two arguments, a unary function f and a distributed array. It applies the function to every element of the distributed array and returns the resultant distributed array. As Map applies the function to each element independently, there is no need of communicating anything. The map skeleton is presented in the figure 3.2.

Zip Zip is an extension to the Map skeleton. It accepts a binary function and the two distributed arrays. The application of the function to the input arrays is similar to that of Map. Figure 3.3 represents the general schema of the Zip skeleton.

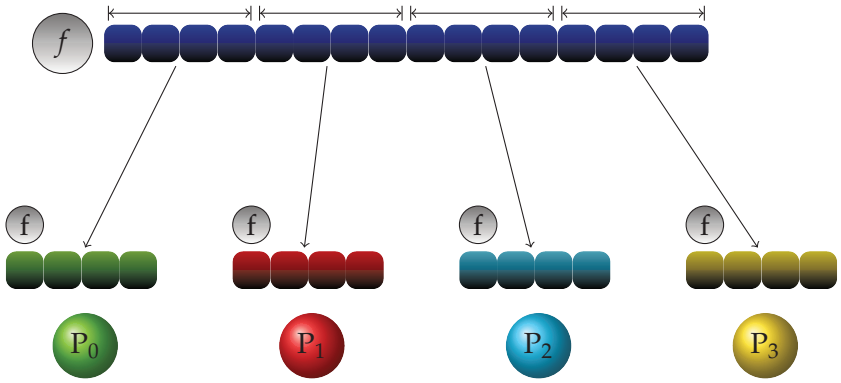


Figure 3.2 – *Map*

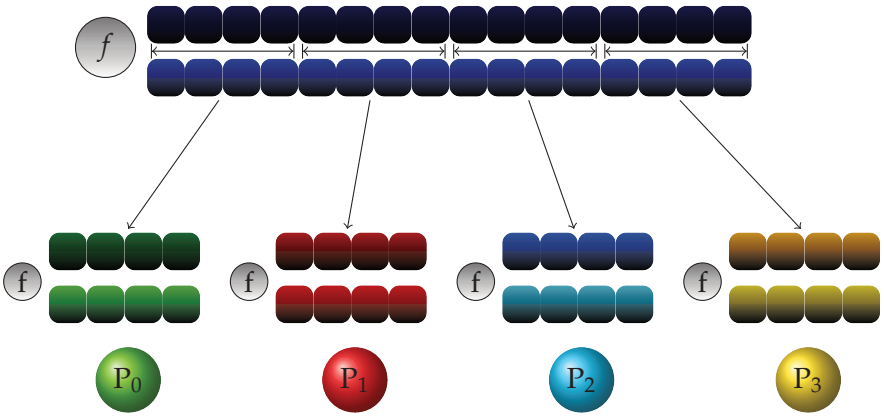


Figure 3.3 – *Zip*

MapIndex It passes to the function the global index of the element along with the element itself.

ZipIndex Same as the MapIndex but it operates on two distributed arrays in a similar fashion to Zip.

As all the skeletons in the Map family iterates the distributed array once, assuming the function f having constant complexity c_f, their BSP cost is in the order of $C_f \times \frac{n}{p}$ when the distributed array is evenly distributed. But in case where the processors contain unequal number of elements the BSP cost is in the order of $c_f \times \max_i n_i$, where i is the index of the processor and n_i means number of elements present at the processor i. For the sake of simplicity, from now onwards we assume an equal distribution of the elements.

3.1.3 Communication Skeletons

The skeletons presented here require at least a full super-step, with communications, to be evaluated. Two of the communication skeletons change the indexes of the elements in the distributed array, without changing the number of elements per processor, i.e. without changing the shape of the distribution. The balance and gather skeletons do not change the indexes but change the distribution whereas the **`Bcast`** skeleton changes both the content and the distribution.

Shift The shift skeleton (Figure 3.4) takes three arguments: the number d of elements to be shifted (positive the elements are shifted to the right, negative they are shifted to the left), a function f and a distributed array. As the shifting is not circular, the function f is used to fill the holes at the beginning (resp. the end) of the distributed array.

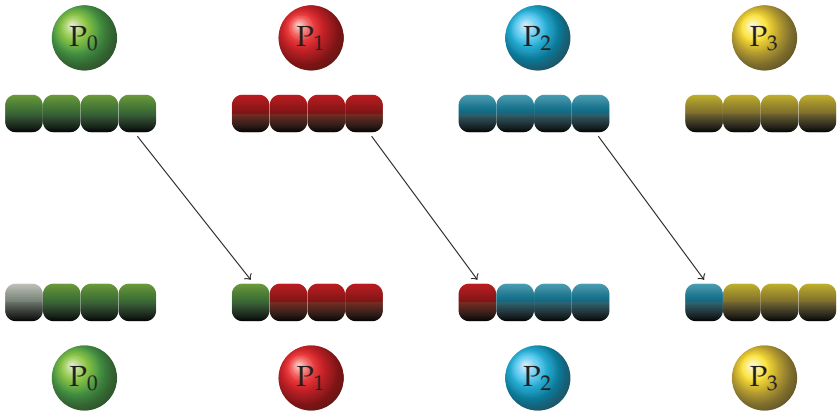


Figure 3.4 – *Shift (right)*

This skeleton requires communication as the first (resp. last) elements at each processor need to be communicated to the left (resp. right) hand neighbour.

Assuming function f has a constant complexity c_f, the elements have the same size s, and each processor has at least $|d|$ elements, the BSP cost of the evaluation of an application of the shift skeleton is in the order of:

$$|d| \times c_f + (\frac{n}{p} - |d|) + s \times |d| \times g + L$$

where $\frac{n}{p} - |d|$ represents the local shifting of the array.

Permute The Permute skeleton (Figure 3.5) takes two arguments: a permutation function f and a distributed array, where $f : \mathbb{N} \to \mathbb{N}$ and for every x in $\mathbb{N}$ there exists a unique y in $\mathbb{N}$ and $x, y \geq 0$ and $<$ to the total length of the distributed array. The function f is applied to the global index of each element for computing

the elements destination global index. The element is then communicated to the new address. The skeleton involves communications depending upon the permutation function. Figure 3.5 represents a special case where each processor communicates its element with the other one with respect to the index of the elements.

Assuming function f has a constant complexity c_f, the elements have the same size s, processors having $\frac{n}{p}$ elements and the function f choose h_i elements for that need to be communicated for processor i, the BSP cost of the application of such an instance will be in the order of:

$$c_f \times \frac{n}{p} + s \times \max_i h_i \times g + L$$

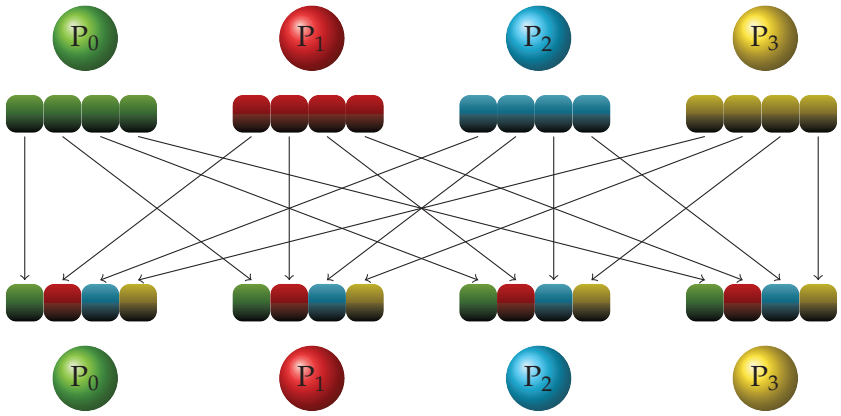


Figure 3.5 – *Permute*

Gather Gather takes a distributed array as its argument, and gathers the data from all the processors to the root processor (default to processor 0). The skeleton is usually used to gather the results of some computational operations from the involved processors to the root processor. After the application of the Gather the distributed array is no more evenly distributed, instead the root processor contains the whole array while the sizes of the other become zero. Assuming each processor has $\frac{n}{p}$ elements, and the size of all the elements is same i.e. s, the BSP cost for the instance of the skeleton will become:

$$(p-1) \times s \times \frac{n}{p} \times g + L$$

where n is the total number of elements in the distributed array and p represents the number of processors.

Broadcast Broadcast takes a distributed array as its arguments and communicates it to all the processors. The broadcast operation is assumed to be carried out by the root processor (processor 0). After the application of the broadcast the distributed array becomes evenly distributed and its length changes as well. The new length of the

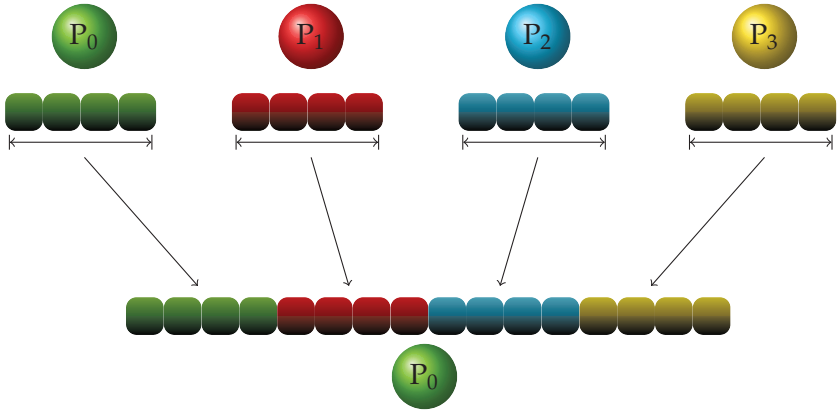


Figure 3.6 – *Gather*

distributed array is p times the size of the partition of the root processor. Assuming a distribution where root processor is holding n_0 elements all of same size s, the BSP cost can be captured by:

$$(p-1) \times s \times n_0 \times g + L$$

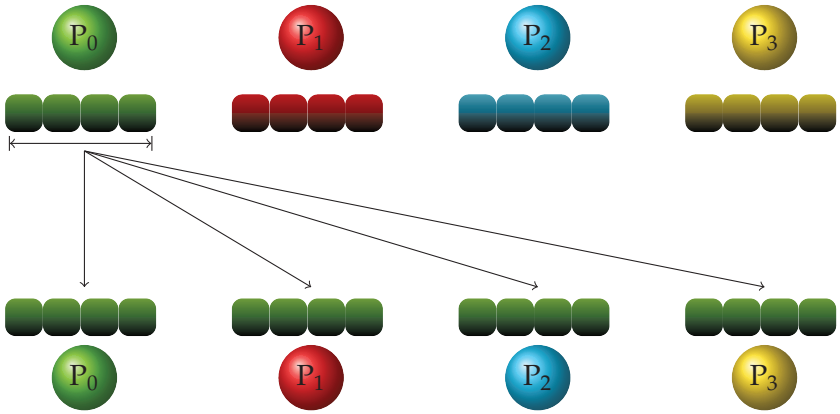


Figure 3.7 – *Broadcast*

Balance Balance accepts a distributed array as its argument. It re-balances the load of the processors. In data parallel settings, the load can be defined as the number of data elements a processor holds. The unequal distribution of the data may be introduced by the application of GetPartition followed by one of Map skeletons using some filter function and then Flatten it back. After the application of this skeleton every processor holds equal number of elements as captured in figure 3.8.

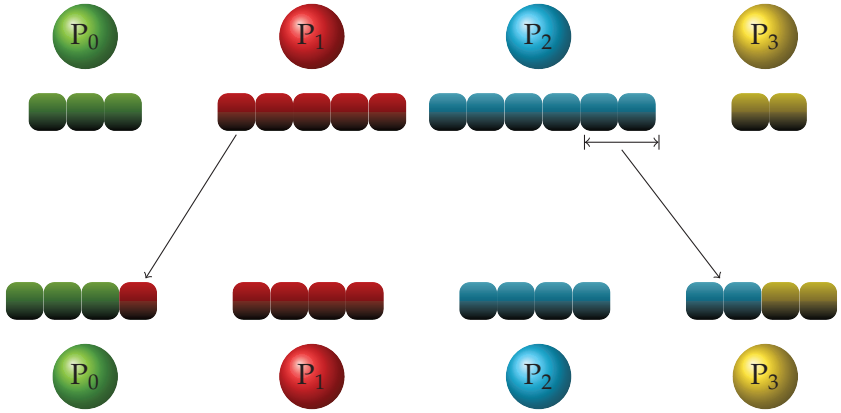


Figure 3.8 – *Balance*

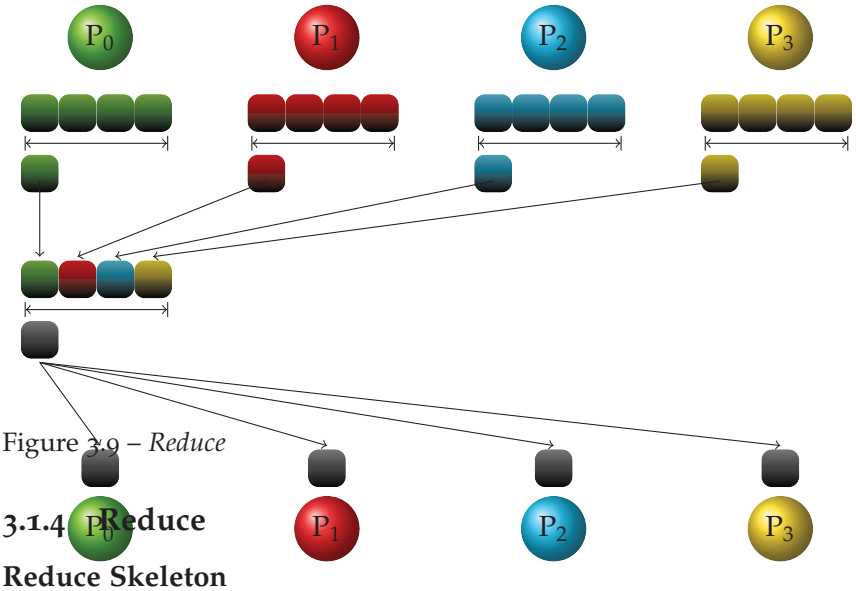


Figure 3.9 – *Reduce*

3.1.4 Reduce

Reduce Skeleton

It accepts a binary reduction operator f and a distributed array as input. It applies the reduction operator f to the input distributed array. The result of the application of the reduction operation is a single element. Every processor after reducing its local data communicates the result to others. The results are then further reduced to get the final result. Figure 3.9 represents one of the possible reduction algorithm. The final results are then broadcasted to all the processors.

3.1.5 View Changing Skeletons

It is sometimes convenient to be able to expose the distribution of elements of a distributed array, and also to remove this exposure. This is done by two skeletons: Get-

Partition and Flatten. These two skeletons do not change neither the content, not the distribution of the distributed array: They only change the view we have on the data.

GetPartition

GetPartition takes as argument a distributed array of type `T`. It exposes the distribution of the distributed array's elements by transforming it to a distributed array of type `vector<T>`. The new distributed array is of size p where p is the number of processors. Each processor now holds one element of type `vector<T>`. Figure 3.10 demonstrates this change of view(type) by enclosing the elements in a rectangle.

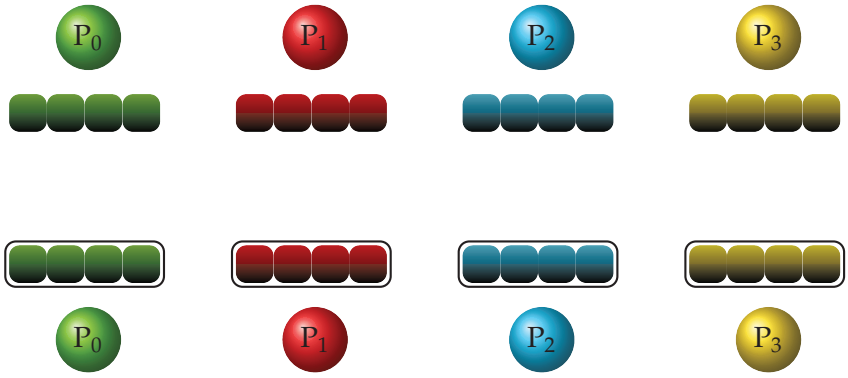


Figure 3.10 – *GetPartition*

Flatten

It represents the inverse operation of the GetPartition and gives back the distributed array its original type back i.e. from type `vector<T>` to type `T` as demonstrated in the figure 3.11.

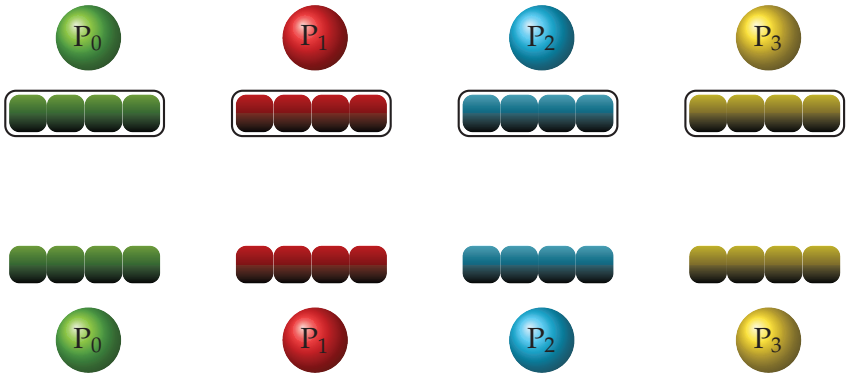


Figure 3.11 – *Flatten*

Primitive	Type	Description
$\ll$ e $\gg$	t par if e : t	$\langle e, \ldots, e \rangle$
$this$ (within a local section)	int	i on processor i
v (within a local section)	t (if v : t par)	v_i on processor i (if $v = \langle v_0, \ldots, v_{p-1} \rangle$)
proj	'a par $\rightarrow$ int $\rightarrow$ 'a	$\langle v_0, \ldots, v_{p-1} \rangle \mapsto$ (**fun** $i \rightarrow v_i$)
put	(int $\rightarrow$ 'a) par $\rightarrow$ (int $\rightarrow$ 'a) par	$\langle f_0, \ldots, f_{p-1} \rangle \mapsto \langle$ **fun** $i \rightarrow f_i 0, \ldots,$ **fun** $i \rightarrow f_i(p-1) \rangle$

Figure 3.12 – *Summary of BSML Primitives*

3.2 A Prototype Implementation in BSML

The latest version of OSL aims at improving the safety of the library while preserving its expressiveness. In this section we present the prototyping of the new OSL library with a parallel functional programming language that follows the BSP model: Bulk Synchronous Parallel ML [101, 38, 128]. Using such a language allows to focus on the design of the underlying algorithms of the skeletons. There is no need to take into account the C++ mechanisms that can be error-prone, while preserving the ability to run the skeletons in parallel. Bulk Synchronous Parallel ML being currently implemented as a library of the OCaml language [94, 44], the skeletons in BSML could be mostly reused in the formal development using the Coq proof assistant [27, 129] (and chapter B) and form the basis of a formal execution model of OSL.

I first give an overview of the programming with BSML (section 3.2.1), before explaining the design and prototyping of the library of algorithmic skeletons in BSML (section 3.2.2). An example application is implemented using this library and some experiments are performed (section 3.2.3).

3.2.1 Bulk Synchronous Parallel ML

In BSML the parameters of the BSP machine can be accessed through 4 constants. The number p of memory-processor pairs of the BSP machine is fixed during execution. p is accessible to the programmer, it is named **bsp_p**. The other BSP parameters are respectively **bsp_g** (network bandwidth), **bsp_l** (synchronisation time), and **bsp_r** which is a measure of the processors computing power. All parameters, but **bsp_p**, can be obtained by a benchmark program.

BSML is based on a distributed data-type called parallel vector. A parallel vector has type 'a par and embeds p values of any type 'a at each of the p different processors in the parallel machine. The nesting of parallel vectors is not allowed.

The p processors are labelled with natural numbers from 0 to $p-1$. We use the following notation for a parallel vector:

$$\langle x_0, x_1, \ldots, x_{p-1} \rangle : \text{'a par}$$

or $\langle x_i \rangle_i$ for short. This vector holds the value x_i at processor i, with all x_i of type 'a. We distinguish this structure from a usual "sequential" vector of size p because the different values, that will be called *local*, are blind to each other. It is only possible to access the local value x_i in two cases:

1. locally, on processor i (by the use of a specific primitive), or
2. after some communications.

These restrictions are inherent to the distributed memory parallelism. This makes the parallelism explicit and the programs more readable. Since the BSML program deals with a whole parallel machine and the individual processors at the same time, a distinction between the levels of execution that take place is needed:

- *Replicated* execution is the default. Code that doesn't involve BSML primitives (nor, as a consequence, parallel vectors) runs on the parallel machine as it would on a single processor. Every processor executes the replicated code at the same time that leads to the same result everywhere.
- *Local* execution is what happens inside the parallel vectors, on each of their components: the processor uses its local data to do computation that may be different from the others.
- *Global* execution concerns the set of all processors together, but as a whole and not as a single processor. Typical example is the use of communication primitives.

BSML programs can be compiled in byte-code, native code or can be evaluated in an interactive fashion using the BSML interactive loop. In this case, when one gives an expression to the top-level, possibly a name, a type and the value of the expression are returned. For example:

```
# bsp_p;;
- : int = 4
```

is the prompt, **bsp_p** is the expression to evaluate, the answer is the second line, giving the name of the value (here no name is given to the value), the type (int) and the value (4). In the remaining of this section, our BSP machine will have 4 processors.

To build a parallel vector containing the same value at all the processors, one can write $\ll$ e $\gg$ where e is a usual "sequential" OCaml expression. If the value of e is v then the value of $\ll$ e $\gg$ is the parallel vector $\langle$ v , . . . , v , . . . , v $\rangle$. For example:

```
≪ "THESIS" ≫ ;;
- : string par = <"THESIS", "THESIS", "THESIS", "THESIS">
```

There also exists a predefined parallel vector, the value named this, that contains the value i at processor i:

```
# this;;
- : int par = <0, 1, 2, 3>
```

The so-called *local section* notation $\ll \gg$ can be used to access the local values of a vector. Let us consider an expression e of type t par. Being an expression with a parallel type, its value is a parallel vector $\langle$ v_0 , . . . , v_{p-1} $\rangle$. *Inside* a local section, for all processor i, the notation e represents at processor i the local value v_i. In combination with this we have a way to build parallel vectors with different values on the different processors:

```
# let hello = ≪ (string_of_int $this$)^":hello" ≫ ;;
hello : string par = <"0:hello", "1:hello", "2:hello", "3:hello">
```

OCaml is a higher-order functional programming language, so it is possible to build parallel vectors of functions:

```
# ≪ ( + ) $this$ ≫ ;;
− : (int → int) par = < <fun>, <fun>, <fun>, <fun> >
```

Here (+) is the integer addition function in prefix notation, partially applied. At processor i we have the function **fun** x → i + x.

The only way to obtain a sequential value from a parallel expression is the use of the proj primitive. The type of this function is 'a par→ int→ 'a. Given a parallel vector, it returns a function such that, applied to the processor identifier of a processor, it returns the value of the vector at this processor. proj is often used at the end of a parallel computation to gather the computed results. For example, if we want to convert a parallel vector into a list, we write:

```
# let f = proj hello in List.map f [0; 1; 2; 3];;
− : string list = ["0:hello"; "1:hello"; "2:hello"; "3:hello"]
```

A (almost) total exchange occurs when the proj function is applied to its first argument. In BSML this ends the super-step. Note that the communication and synchronisation phases occur only when proj is applied to its *first* argument. In the example further applications of f do not imply additional communications and synchronisations.

Some values are considered to be an empty message (or to have size 0) and are thus not communicated through the network, even if the yielded results may suggest they were. It is the case for example for empty lists, the value None of the 'a option type, *etc.* Thus the communication schema of proj may be not a full total exchange if some of the values in the argument vector are values of size 0.

put is the comprehensive communication primitive: It allows any local value to be transferred to any other processor. It is synchronous, and ends the current super-step. Canonical use of put is

```
let com = put ≪ fun sendto → e($this$, sendto, $x$)≫
```

where expression e computes (or usually, selects) the data of vector x that should be sent depending on the destination processor sendto. The return value of put is another vector of functions. At a processor j the function, when applied to i, yields the value *received from* processor i by processor j.

For example, the following **shift** function shifts a parallel vector circularly to the right:

```
# let shift v =
   let vdst = ≪ ($this$+1) mod bsp_p ≫
   and vsrc = ≪ (bsp_p+($this$−1)) mod bsp_p ≫ in
   let shifted =
    put ≪ fun dst→ if dst = $vdst$ then [$v$] else []≫ in
   ≪ List.hd ($shifted$ $vsrc$) ≫ ;;
```

```
make: int→ 'a→ 'a distArray
init: int→ (int→ 'a)→ 'a distArray
atRoot: (unit→ 'a array)→ 'a distArray
getPartition: 'a distArray→ 'a array distArray
flatten: 'a array distArray→ 'a distArray
map: ('a→ 'b)→ 'a distArray→ 'b distArray
mapIndex: (int→ 'a→ 'b)→ 'a distArray→ 'b distArray
zip: ('a→ 'b→ 'c)→ 'a distArray→ 'b distArray→ 'c distArray
zipIndex: (int→ 'a→ 'b→ 'c)→ 'a distArray→ 'b distArray→ 'c distArray
shift: int→ (int→ 'a)→ 'a distArray→ 'a distArray
permute: (int→ int)→ 'a distArray→ 'a distArray
balance: 'a distArray→ 'a distArray
reduce: ('a→ 'a→ 'a)→ 'a→ 'a distArray→ 'a
gather: 'a distArray→ 'a distArray
bcast: 'a distArray→ 'a distArray
```

Figure 3.13 – *Skeletons of the Prototype Library*

```
val shift : 'a par → 'a par = <fun>
# shift hello;;
− : string par = <"3:hello", "0:hello", "1:hello", "2:hello">
```

A summary of BSML primitives is given in figure 3.12.

3.2.2 A Library of Algorithmic Skeletons

We implemented a prototype library of a new version of Orléans Skeleton Library in BSML. We first describe the underlying data structure before detailing the main (but not all) skeletons.

Distributed arrays

The data structure manipulated by the skeletons of our prototype library are distributed arrays. In BSML such a data structure could be implemented as a parallel vector of arrays. However it is convenient, and more efficient, to store some additional information rather than to compute on demand:

- the start index of each processor with respect to the global array: each processor contains one array, but this array is a sub-array of the distributed array considered as a whole array; computing it from the parallel vector of arrays would require communications,
- the global length of the distributed array: if it is not stored then a parallel reduction is needed to compute it from the parallel vector of arrays,
- the distribution: each processor knows the local length of the other local arrays without having to communicate; we represent the distribution as an array of integers of size **bsp_p**.

The type for distributed arrays is thus defined as:

```
type 'a distArray = {
  data : 'a array par;
  startIndex: int par;
  globalSize : int;
  distribution : int array;
}
```

globalSize and distribution are not parallel vector but have usual sequential types. This makes clear that these fields cannot have a different value on two different processors. On the contrary startIndex is a parallel vector: at each processor it contains the start index of the local array in the global array. It is also possible to choose startIndex to be the array of the start indices of all the local arrays. However, for all skeleton but one, it is not necessary for a processor to know the start indices of other processors: therefore it is better to save memory by having only one integer value per processor for startIndex rather than having a *replicated* array of size p (than in practice is p integer values on each processor).

An example of value, if we have 4 processors, is:

```
# let da = init 11 string_of_int;;
val da : string distArray =
{data = <[|"0"; "1"; "2"|], [|"3"; "4"; "5"|],
         [|"6"; "7"; "8"|], [|"9"; "10" |]>;
 startIndex = <0, 3, 6, 9>;
 globalSize = 11;
 distribution = [|3; 3; 3; 2|]}
```

The set of skeletons provided to the user of the library is given in figure 3.13. The three first skeletons are, if we take an object oriented programming terminology, constructors of distributed arrays. make creates a distributed array of a given size with the given value everywhere, init creates a distributed array of a given size with its elements given by applying a function from indices to values, and atRoot builds a distributed array from a sequential array at root processor. The two first functions give an evenly distributed array whereas the third one returns a distributed array with values only at processor 0. For the two first constructors, the startIndex, globalSize and distribution fields do not need any communication to be computed as the distribution is known from the global size when the array is evenly distributed (if the size is not divided by the number of processors, the processors with a low process identifier may have one additional element). For the third constructor, communications are required.

The getPartition and flatten skeletons

The **getPartition** and **flatten** skeletons are such that:

flatten(**getPartition** da) = da

Basically **getPartition** makes the distribution of the distributed array apparent, and is mostly a change of point of view, that is inexpensive to compute: It just, at each processor, puts the local array into an array of one element. However we wish the **flatten** skeleton to be also inexpensive to compute. Therefore we would like to keep the information related to the global size, start indices and distribution before the **getPartition** to be able to restore them when there is a call to **flatten**. In order to do that, we actually have a fifth field in the distArray type: partitioned, a boolean. When this field is **true** this means that the globalSize, startIndex and distribution fields refers to a distribution before a call to **getPartition**. The actual distribution could be computed without any communication if we assume that it is a distribution obtained after a call to **getPartition**: the global size is **bsp_p**, the startIndex is equal to this and the distribution is such that there is one element per processor.

For example:

```
# let pda = getPartition da;;
val pda : string array distArray =
{ data = < [| [|"0"; "1"; "2"|] |], [| [|"3"; "4"; "5"|] |],
           [| [|"6"; "7"; "8"|] |], [| [|"9"; "10"|] |]>;
  startIndex = <0, 3, 6, 9>; globalSize = 11;
  distribution = [|3; 3; 3; 2|]; partitioned = true }
```

However, if other skeletons are applied to a partitioned distributed array, it is not always guaranteed that the distributions can be updated without additional communications. In this case the startIndex, globalSize and distribution fields are replaced by their actual values and the field partitioned is set to **false**. A call to **flatten** on a distributed array that is not partitioned incurs communications: each processor contains an array of arrays that is flattened to an array, and the sizes of these arrays are totally exchanged. From these sizes the various fields can be computed (without new communications).

The map, zip and balance skeletons

The **map** skeleton does not change the distribution of a distributed array. However if the distributed array is partitioned (i.e. its field partitioned is **true**), the startIndex, globalSize and distribution fields still contain the values they had before the call to **getPartition**. We call this set of fields and their values "the distribution before partitioning". With a call to **map**, the distribution before partitioning is not guaranteed to be preserved.

For example:

```
let da' =
  let f a = let l = Array.length a in Array.sub a 0 (l/2) in
  map f (getPartition da);;
```

does not preserves the distribution of da. If the local sizes of da are even, the global size of **flatten** da' will be half of the one of da. Moreover the distribution depends on the *local* application of the function f, so it is not possible to update the distribution before

partitioning without communication. Thus for an application of **map** we remove the distribution information before partitioning (it requires no communication):

```
 val da' : string array distArray =
{ data = <[|[|"0"|]|], [|[|"3"|]|], [|[|"6"|]|], [|[|"9"|]|]>;
  startIndex = <0, 1, 2, 3>; globalSize = 4;
  distribution = [|1; 1; 1; 1|]; partitioned = false }
```

The **mapIndex** variant of **map** has the same properties. It benefits from having the startIndex field.

The **zip** skeleton is a generalisation of the **map** skeleton to two distributed arrays. As the **map** skeleton, the distribution is preserved, but the distribution before partitioning is not. There is also a problem that may occur with the **zip** skeleton. In functional programming, this **zip** function exists and can be applied to two lists that have different sizes. The results will have the length of the smallest input list. However in a distributed settings, this is not so easy as even same global sizes may correspond to unaligned distributed arrays. Therefore to have a safe library, we check that the two distributed arrays have the *same distribution*: if not, an exception, corresponding to a programming error, is raised. This is not a limitation as we provide a **balance** skeleton. This check does not imply any communication as the distributions are stored in the field distribution.

The **balance** skeleton changes the distribution of a distributed array to an even distribution: communications are required if the distributed array is not already evenly distributed. It is to be noticed that partitioned arrays are actually evenly distributed (there is one element per processor). Therefore the **balance** skeleton preserves the distribution before partitioning.

The shift skeleton

The **shift** skeleton preserves distribution, but it does not preserve distribution before partitioning: the sizes of the values generated by the replacement function are known only locally to the processor where they are produced.

No communications are needed if d is 0 or $|d|$ is greater than the global size of the distributed array. In the first case **shift** is identity, in the second case the application **shift** $d\ f$ is equivalent to **mapIndex** f' where $f'\ i\ x = f\ i$.

The **shift** skeleton is easy to write when the distribution is even. It is a bit more complicated for an arbitrary distribution, especially with "holes" in the distribution. We will illustrate the different steps on the following distributed array for a shift with 3 as offset:

```
val da : string distArray =
{ data = < [|"A"; "B"|], [|"C"; "D"|], [||], [|"E"; "F"; "G"|] >;
   startIndex = <0, 2, −1, 4>; globalSize = 7;
   distribution = [|2; 2; 0; 3|]; partitioned = false }
```

It proceeds as follows.

At each processor, we compute the sub-array of elements to be communicated (shifted) to other processors, information about distribution is necessary to do so. As at each processor we have the starting index of the local array in the parallel vector startIndex, we can compute locally the global destination index of each element to be shifted to other processors: it is the current global index plus the offset. Then for each element of this sub-array, we should compute the destination processor that corresponds to this global destination index: (a) from the distribution array, we compute the prefix sum of the distribution (the start indices) where the processors with no elements have been filtered out, each value being paired with its corresponding processor. From example, if the distribution is the one of da, we obtain: [|(0, 0); (1, 2); (3, 4)|] ; (b) then using a binary search on this array it is possible to obtain the destination processor (first component of the pair) for each global destination index in time $\mathcal{O}(\log_2 p)$; (c) the values to be sent to the same processor are packed together.

With the distributed array da above, we obtain the following parallel vector:

```
− : (int ∗ string array) list BSML.par =
< [(1, [|"A"|]); (3, [|"B"|])], [(3, [|"C"; "D"|])], [], [] >
```

We use a variant of the BSML put primitive, that takes as input a parallel vector of lists of pairs (destination, message) instead of a parallel vector of functions, for the communications. At the end of this step we obtain the following parallel vector:

```
string array BSML.par = < [||], [|"A"|], [||], [|"B"; "C"; "D"|] >
```

Finally we perform a local shift whose replacement function either calls the global shift replacement function or returns elements communicated in the previous step. In the example, the global replacement function is called on processor 0 and 1 (for the first element), and the elements are taken from the previous parallel vector of arrays for processor 1 (second element), and processor 3 (for the 3 last elements):

```
# shift 3 (fun i→ "Nothing") da;;
− : string distArray =
{ data = < [|"Nothing"; "Nothing"|], [|"Nothing"; "A"|],
           [||], [|"B"; "C"; "D"|]>;
  startIndex = <0, 2, −1, 4>; globalSize = 7;
  distribution = [|2; 2; 0; 3|]; partitioned = false }
```

The permute skeleton

The **permute** skeleton takes as input a bijective function from 0 to the global size of its distributed array argument. The **permute** skeleton is based on the same auxiliary functions than the **shift** skeleton. For each element of the distributed array, we compute: first its global destination index, obtained by applying the bijection f to the global index, then we compute the destination processor according to this global destination index.

One concern with the **permute** skeleton is to check whether the function f is bijective or not. One possibility is to perform the check independently on each processor,

applying the function to all possible indices: this would requires $\mathcal{O}(n)$ operations at each processor, compared to the $\mathcal{O}(\frac{n}{p})$ applications (if the distributed array is evenly distributed) of f needed to compute the destination processor. This may be quite costly if $p \ll n$. Moreover, we have written the code in such a way that if f is not bijective, no run-time error will occur but the global size of the obtained distributed array will not be the same than the original size. Therefore we can perform a total exchange of the local sizes, then compute the global size and compare it to the initial global size to check if the function was actually a bijection (and raise an exception to indicate the programming error if there is one). The additional cost of this check is $\mathcal{O}((p-1) \times g + L)$. Thus we could dynamically determine, depending on the BSP parameters of the machine, if the first version of the check is more expensive or not than the second version of the check, and choose accordingly the best version. In the current prototype only the second version is performed.

Unlike **shift**, it is possible with the **permute** skeleton to update both the actual distribution and the distribution before partitioning without additional communications.

3.2.3 Heat Diffusion Simulation Example and Experiments

The goal of this application is to give an approximate solution of the following partial differential equation describing the diffusion of heat in a one dimensional bar of metal of length 1:

$$\frac{\delta h}{\delta t} - \kappa \frac{\delta^2 h}{\delta^2 x} = 0 \qquad \forall t,\ h(0,t) = l \qquad \forall t,\ h(1,t) = r \tag{3.1}$$

where κ is the heat diffusivity of the metal, and l and r some constants (the temperature outside the metal). A discretised version of this equation is:

$$h(x, t+dt) = \frac{\kappa dt}{dx^2} \times \big(\ h(x+dx,t) + h(x-dx,t) - 2 \times h(x,t)\big) + h(x,t) \tag{3.2}$$

for a space step dx and a time step dt and with the same boundary conditions.

In sequential the temperature at the discretisation points in the metal bar is stored in an array, and a loop is used to update this array when a new step time occurs. For the OSL version, we use a distributed array for storing the values of h. If we consider the equation 3.1 for the whole array h, and not element-wise, then the update of h is is a linear combination of the distributed arrays in figure 3.14.

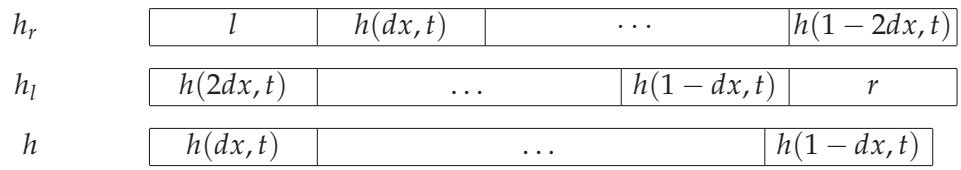


Figure 3.14 – *Distributed Arrays for Heat Diffusion Simulation*

Array h_r (resp. h_l) can be computed from array h using the **shift** skeleton with offset 1 (resp. -1) and with replacement function the constant function returning always l

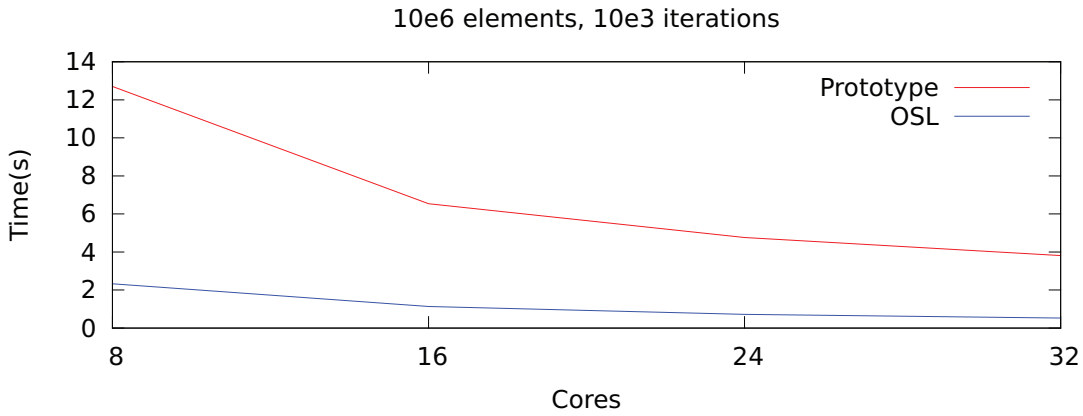


Figure 3.15 – *Heat Diffusion Simulation*

(resp. r). The linear combination of these arrays can be computed using the **map** and **zip** skeletons. Thus if the temperature in the bar is represented by a distributed array of floating point numbers, the update of the temperature is performed by the following function:

```
(* step: float→ float→ float→ float→ float→
          float distArray→  float distArray *)
let step kappa dt dx l r bar =
  let barR = shift 1 (fun _→ l) bar
  and barL = shift (−1) (fun _→ r) bar in
  zip
    (fun vl ul → (kappa *. dt)/.(dx*.dx)*.(vl−.2.*.ul) +. ul)
    (zip ( +. ) barR barL)
    bar
```

Simulating the diffusion of heat in the bar of metal is then an iterative application of the function step.

We ran two different versions of a one dimensional heat diffusion simulation:

- the above version written using the proposed prototype library,
- a version written in C++ using the OSL library (section 4.2), whose code uses mostly the same skeletons than the first version.

The experiments were conducted on the SPEED parallel machine(for details see appendix C) of the university of Orléans: it is a shared-memory machine with 4 AMD Opteron 6174 processors, each being a 12 cores processor. We set 10^6 discretisation points for the bar of metal, and run 10^3 time steps. The timings are presented in figure 3.15.

The version implemented using the prototype BSML implementation of OSL is less efficient: this is due to the fact that quite many intermediate arrays are created. However the difference is less than one order of magnitude: it is still reasonable to

experiment parallel runs with this prototype. In a real implementation of the new OSL library these intermediate copies are removed using the C++ expression templates optimisation technique.

3.3 A C++ Implementation of OSL

The C++ version of OSL is developed by following the specifications obtained from the prototype presented in the previous section.

3.3.1 Distributed Arrays

As in the prototype, the BSP parameters can be accessed using the constants including **bsp_p** the number of processors of the BSP machine.

The data structure "Distributed Array" is implemented using a template class `DArray<T>`. The 'T' represents the type of the data. To keep the information about the size, indexing and the distribution of the data among processors, `DArray` class is supported with the following members:

Members

globalSize The total number of elements present in the distributed array.

localSize The number of local elements of the processor.

startIndex The global starting index of the data.

partitioned A boolean flag indicating whether there is a change in view of the data in the distributed array or not. In case the data is partitioned the global size of the data becomes the **bsp_p** and the local size becomes 1. The start index is the `bsp_id` and the distribution is 1 for each element.

distribution The vector containing the number of elements present at each processor.

data The vector containing the local elements.

The `DArray` class is further supported with the getter and setter members for all the above mentioned members of the distributed array.

Constructors

The constructors presented below are inspired from the `make`, `init` and `atRoot` functions of the prototype version.

DArray(int globalSize) Instantiates the object by evenly partitioning the data among the processors. The global size of the data is the one presented as the argument. The even distribution is computed by calling a `evenDistribution(gs)` function. The global start index is calculated by calling the method `evenDistributionStartIndex(gs, bsp_id)` where `bsp_id` is the processor identifier. Partitioned is set to false. The memory for the data is allocated without initialisation.

DArray(int gs, T value) Initialises all the elements of the object by the same value.

template<typename F> DArray(F f, int gs) Every element is initialised by applying the function object `f` to the corresponding global index.

template<typename F> DArray(F f) The `f` acts as a generator of the data. The size of the data is known to the root processor (`bsp_id = 0`) on which the data is generated using the function `f`. The global size is therefore broadcast to the other processors. The start index of the root processor is zero while of the others is -1. The local size of the root processor is equal to the global size of the data while that of others is zero. The constructor is usually used when the input source of the data is a file.

DArray (DArray<T>) The object is initialised with the elements of the distributed array in arguments. The constructor is provided in two flavours: first creates a copy of the given distributed array while the second is a move constructor that moves the contents of the given distributed array, hence destroying the source array. The move constructor is invoked when the source is a temporary. It is the optimisation that is supported in the new standard of the C++ named C++0x and is presented in detail in chapter A of the appendix.

operator=

The assignment operator is overloaded for addressing the two problems. The first one is the assignment of a distributed array to another distributed array and the second one is the initialisation of the resultant distributed array. The later is the result of the evaluation of a skeleton expression. And this method acts as the starting point of the evaluation of the OSL expressions. The OSL expressions are actually the calls to the skeletons. The size and the distribution of the resultant array depends on the operating skeletons. The method is also implemented in both the copy and move flavours. The right flavour is invoked by the compiler depending upon the right hand side expression.

operator[]

The index operator is overloaded to access the elements of the data. The index here is the local index of the data not the global index. The method is overloaded to return the reference of the indexed element or the **const** reference to the indexed element. For the safety purpose, user is recommended to access the elements of distributed array through some skeleton.

3.3.2 Skeletons

The skeletons in OSL are data parallel skeletons. The skeletons can further be grouped in three categories:

- the skeletons that independently operates on the every element of the distributed array without any need of communication,
- the skeletons representing certain communication patterns,
- the skeletons that changes the view of the distributed array.

Map, **MapIndex**, **Zip** and **ZipIndex** belong to the first class of the skeletons. The former two needs a function object and a distributed array as their arguments while the later two needs two distributed arrays along with a function object. Both applies the function object over every element of the distributed array(s).

Shift, **Permute**, **Gather** and `Broadcast` belongs to the second category. The skeletons perform no computation over the data. They are the pure communicating skeletons representing some pattern of the communication.

`Reduce` and `Scan` form another category of the skeletons that mix the computations and the communications.

Last class of the skeletons are the **GetPartition**, and the **Flatten** skeletons. They change the view of the data. The former represents the local data of the processor in the form of a single element vector containing the whole data. While the later do the inverse to give the data its original view back.

Most of the specifications of the skeletons are finalised in the prototype. In this section the focus will be on the C++ implementation of these skeletons. All the OSL skeletons are represented by the template classes. To simplify the instantiation of the skeleton classes and their composition, the generator functions are supplied. The following listing presents the signature of a generic skeleton class and its generator function.

```
template<typename F, typename Exp>
class SkeletonName
{
  ...
};
```

```
template<typename F, typename Exp>
SkeletonName<F, Exp> skeletonName(F f, const Exp& exp)
{
  return SkeletonName<F, Exp>(f, exp);
}
```

Evaluation of skeletons expressions

Most of the skeletons in OSL follows the pattern presented in the previous section. There are some exceptions, in which case the difference is in the return type of the generator function. Instead of returning the instance of the skeleton they return the instance of the distributed array. This difference is due to the different types of evaluations of the skeletons. In OSL the skeletons can be classified in three categories in terms of their evaluation

- The skeletons that can be evaluated in terms of the expression templates. As a result no temporary is created and the expressions containing such skeletons can be evaluated in a single go (in one loop) following the principle of expressions templates [136]. **Map**, **MapIndex**, **Zip** and the **ZipIndex** belong to this class of skeletons. They can be termed as lazily evaluated skeletons.
- The skeletons that creates the temporaries. All the communicating skeletons create temporary objects during evaluation. As these skeletons needs to communicate they can not be evaluated lazily. Hence they can not be embedded in the expression template mechanism. In order to provide a uniform mechanism, these skeletons are evaluated normally/eagerly and hence the generator function of these skeletons returns the distributed array. `Scan`, **Shift**, **Permute**, **Balance**, **Gather**, **GetPartition**, **Flatten** belong to this class of skeletons.
- The `Reduce` skeleton is the only one in this category as the result type of the `Reduce` skeleton is a single element not a distributed array.

Map, MapIndex, Zip, ZipIndex

These skeletons belong to the first class of the skeletons and hence are evaluated lazily following the mechanism of the expression templates. The principle of expression template is: types represent terms of a kind of domain specific language, operators are overloaded so to build the terms of this language, other operators such as = and [] are used to launch the evaluation of these terms. The mechanism is explained in detail in the section A.1 of the appendix A. To evaluate these skeletons in terms of the expression templates the classes of these skeletons are provided with an overloaded index operator (**operator**`[](`**int** `index)`). The function object is applied to the indexed element and the resultant element is returned. This method is called from the starting

point of the evaluation (whether in the distributed array class or in some other skeleton). Hence the evaluation loop to call the skeletons index operator is present outside these skeletons body. The code below presents the index operator of the **Map** skeleton.

```
typename F::result_type inline operator[] (int index) const
{
  return f(exp[index]);
}
```

The other three skeletons follow the same approach. Some other methods are required in order to implement the expression templates. These methods are added to the skeleton classes and are presented below:

```
// returns the size of the partition of the current processor
inline int getLocalSize() const {
  return exp.getLocalSize();
}
// returns the partition starting index of the current processor
inline int getStartIndex() const {
  return exp.getStartIndex();
}
// return the size of expression/distributed array
inline int length() const {
  return exp.length();
}
// return false indicating if getPartition
// is applied it is followed by some Map skeleton
inline bool getPartitioned() const {
  return false;
}

// return the current distribution of the data
inline std::vector<int> getDistribution() const{
  return exp.getDistribution();
}
```

It is noted that these methods are actually the getters of the distributed array. They need to be present in all the expressions taking part in the expression template mechanism.

GetPartition

GetPartition changes the view of the data. It does so by squeezing the whole data partition into a single element vector, where the element of the vector represents the whole data. The skeleton can be implemented by just assigning the pointer of the

original data to the new single element vector. But the skeleton is not implemented in this manner for the safety reasons and for providing the uniform interface for the uniform composition of the skeletons. The safety concern is that if instead of creating a copy the original pointer is assigned and the source vector is used somewhere else, there is a risk of getting the memory segmentation faults or data consistency issues. The other reason is to support composition of this skeleton with some other skeleton instead of the distributed array. In that case the skeleton acts as the starting point of evaluation for the input skeleton expression. To achieve this two generator functions are provided, one for the classic use of the skeleton over a distributed array and the second for the skeleton expressions.

```
template<typename T>
DArray<vector<T> > getPartition(const DArray<T>& temp)
{
  return GetPartition<DArray<T> >(temp)();
}

template<typename Exp>
DArray<vector<typename Exp::result_type> >
getPartition(const Exp& exp)
{
  typedef typename Exp::result_type result_type;
  return GetPartition<DArray<result_type> >(Evaluate<Exp>()(exp))();
}
```

The first generator function simply calls the overloaded parenthesis operator of the GetPartition class. While the second one first evaluates the given expression and then calls the same operator over the result of the expression. The reason for calling the overloaded parenthesis operator is that unlike the first class of skeletons complete evaluation lies inside the body of the skeleton. The implementation of the skeleton is such that the `getData()` function of the input distributed array is called to access the data and the copy of that data is assigned to the resultant distributed array containing the whole partition as single element vector. The optimisations applied in the implementation of the skeleton is the move semantics and the rvalue references of the C++0x skeleton. Instead of copying the temporary created by the skeleton the temporary object is moved to the final result, in this way saving a copy operation and utilising the same memory resources. The move semantics are presented in detail in section A.3.

Flatten

Flatten follows the same philosophy of the GetPartition skeleton. Two generator functions are provided in the same way, and the copy operations are avoided by following the move semantics. The result of the flatten operation is the normal distributed array

with the updated distribution. If the `partitioned` member of the distributed array is false, this means that the some other skeleton that can update the distribution is applied after the application of the GetPartition skeleton. Hence the distribution can not be updated without communicating the new local sizes. The local sizes of the data is exchanges using MPI all to all routine. Rest of the information can then be computed at each processor on the basis of the new distribution.

Balance

The situation may arise when the data is not evenly distributed among the processors. This may create a load unbalance. To re-balance the load among the processor Balance skeleton is applied over the unbalanced distributed array. The processors containing the major share of the elements need to communicate their elements to the less loaded processors. A lot of algorithms exist in literature to balance the load. The precomputed sliding algorithm [67] is used to balance the distributed array. The algorithm computes the minimum number of exchanges required to balance the data. During the first step, the possible load received by each processor (may be negative) from the right hand neighbour is computed. The second step realises the load transfer towards the left. And the third and the last step takes into account any transfers towards the right side. After the application of the algorithm the load is evenly balanced among the processors. In case the even balance is not possible the low rank processors contains one more element than the higher rank processors. And this is how OSL distributes data normally. A number of functions are required to make the skeleton work. These are the trimming of the vector containing data, insertion of the new elements. The trim from both the sides and the data insertion from both the ends are implemented and optimised using the move semantics to ensure the efficiency.

```
template<typename T>
vector<T> trimLeft(vector<T>& vec, int no)
{
  vector<T> res = std::move(vector<T>(
                std::make_move_iterator(vec.begin()),
                std::make_move_iterator(vec.begin() + no),
                vec.get_allocator()
                ));
  vec = std::move(vector<T>(
        std::make_move_iterator(vec.begin() + no),
        std::make_move_iterator(vec.end()),
        vec.get_allocator()
        ));
  return std::move(res);
}

template<typename T>
vector<T> trimRight(vector<T>& vec, int no)
{
```

```
  vector<T> res = std::move(vector<T>(
                std::make_move_iterator(vec.end()-no),
                std::make_move_iterator(vec.end()),
                vec.get_allocator()
                ));
  vec = std::move(vector<T>(
        std::make_move_iterator(vec.begin()),
        std::make_move_iterator(vec.end()- no),
        vec.get_allocator()
        ));
  return std::move(res);
}
```

The data is passed as a reference to avoid the copying operation. The trimmed elements are returned as a vector. As the new vector containing trimmed elements is created inside the trim function, it is returned using the move to reuse the memory resources and avoid the copy operation.

Shift

As the Shift skeletons also belongs to the second category of the skeletons, its implementation follows the same pattern of the above presented two skeletons. The skeleton handles various use cases to remain safe, shifting the partitioned distributed array, both left and right shifting. It takes three arguments, the offset, the replacement function and the distributed array to be shifted. If the offset is greater than the global size of the distributed array, the shift operation becomes the application of the map index. To shift the array normally towards left or right, the first thing that needs to be computed is the destination processors and the elements to be communicated to the destination processors. The procedure in C++ version is not similar to the procedure in BSML, because for send operation in MPI the destination processors should be known, in a similar way for the receive operation the source processors should be known as well. So, using the binary search over the scanned distribution (start indexes) of the processors both the destination and the source processors need to be determined. The send buffer (a vector of vector buffer containing the elements to be send to the respective processors) is created using the destination processor ranks and is then communicated. Similarly the receive operation proceeds and collect the elements in a receive buffer. The elements are then assigned to the locally shifted data. To optimise the local shift, assignment of the received elements and boundary elements, several auxiliary functions are developed. These are the append and prepend methods for joining two vectors. These methods are optimised using the move semantics.

```
template<typename T>
vector<T> append(vector<T>& original, vector<T>&& elements)
{
  auto elements_begin = std::make_move_iterator(elements.begin());
```

```
  auto elements_end = std::make_move_iterator(elements.end());
  original.insert( original.end(), elements_begin, elements_end);
  return original;
}

template<typename T>
vector<T> prepend(vector<T>& original, vector<T>&& elements)
{
  auto elements_begin = std::make_move_iterator(elements.begin());
  auto elements_end = std::make_move_iterator(elements.end());
  original.insert( original.begin(), elements_begin, elements_end);
  return original;
 }
```

To avoid the copy operations and the creation of temporary vectors, the original data is received as the simple reference while the elements that need to be added are received as rvalue references. Then instead of copying, the elements are inserted using the move iterators. The local shift operation follows the same approach

```
data = std::move(vector<result_type>(
                   std::make_move_iterator(data.begin() + offset),
                   std::make_move_iterator(data.end()),
                   data.get_allocator()
                  ));
```

Permute

Permute is based on the same set of the auxiliary functions that are required for the Shift skeleton. But contrary to the shift where the communication is uni directional i.e either left or right, in Permute the messages can be bidirectional. So, in the implementation of the skeleton every processor sends a vector to the every other processor. The vector may contain zero to 'n' elements depending on the permutation function. Another issue is the arrangement of the data after reception. To fix the issue, the global destination index of the data is communicated as well along with the data.

Gather

It is used to collect the local partitions of all the processors at the root processor. Gather is implemented in the same manner as that of Permute and Shift. It changes the distribution of the distributed array. The root processor which gathers all the data now contains the total number of the elements i.e. global size of the distributed array, while the other processors hold nothing. It further sets the partitioned flag to false.

Broadcast

The skeleton is used to broadcast the local partition of root processor to the others. Its implementation is quite similar to the Gather. The global size of the data changes to the total no of processors times local size of the root processors partition. Hence the distribution of the distributed array is modified. It also sets the partitioned flag to false. If there exists some data on the receiving processors, that data will be lost after the broadcast operation.

Reduce

Different algorithms for the reduction skeleton are implemented and are presented in detail in the chapter 5.

3.4 Summary

The chapter focuses on presenting the design of the OSL, a library of data parallel skeletons developed in C++ and based over MPI for communication. The goals presented at the start of the chapter are addressed through several design choices and the two implementations one in BSML and the other using C++. The base data structure is a distributed array. The idea behind distributing the array a construction is to avoid the scattering and gathering of the data with skeleton calls and keep the skeletons clean. The information about the distributed array is captured by different members of the distributed array to allow the safe programming (to avoid distribution inconsistencies and the resultant memory errors). The safe programming became a must after the introduction of the new skeletons, the view changing skeletons. The OSL is prototyped using the BSML to allow the reasoning about the skeletons in the functional programming style which is not possible with C++. It helps in the evolution of the formal programming and execution semantics. The C++ implementation focus on the advanced C++ programming techniques like expression templates and the move semantics to allow the development of an optimised version. The proceeding chapters address the goals established in the beginning of this chapter.

Programming with OSL

4

Contents

5.1 Benchmarking the BSP Parameters 74
5.2 Performance Prediction: A Case Study 75
5.3 Performance Portability 77
5.3.1 Reduce by Using Gather and Broadcast 79
5.3.2 Reduce by Using All Gather 79
5.3.3 Reduce by Tree Gather and Broadcast 81
5.3.4 Reduce by Tree All Gather 81
5.3.5 Best Algorithm Selection 82
5.3.6 Variance: A Case Study 83
5.4 Summary 84

To reduce the complexity of parallel program development OSL offers a structured approach. The data is represented by instantiating the `DArray` class. The operations on the data are executed through the calls to the OSL skeletons. The applications can be developed using the different combinations of these skeletons. The chapter presents several case studies each one elaborating the expressiveness from a different point of view.

4.1 A First Example

The example demonstrates how a simple parallel program can be developed using OSL. The objective is to increment every element of a distributed array. As every element can be incremented independently, the program is embarrassingly parallel and there are no communications involved. The program is an ideal candidate for the application of **Map** skeleton. The first argument to the **Map** is a simple sequential function object for incrementing a single element (simple sequential function object). Once we have the required sequential function object and the data (as an instance of distributed array), a call to the **Map** with these arguments will solve the problem.

```
struct Initialize : Boost::function<int(int)>
{
  int operator()(int i) const
  {
    return i;
  }
};

struct Increment : Boost::function<int(int)>
{
  int operator()(int i) const
  {
    return ++i;
  }
};

int main()
{
  DArray<int> darray(Initialize(), 100);
  darray = map(Increment(), darray);
  return 0;
}
```

The data in the example presented above is generated through an `Initialize` function object. It takes as input an integer and returns the same integer. So the `darray` contains elements from 0 to 99. `Increment` represents the sequential function object for incrementing the element. And the distributed array containing data is `darray`.

One important thing to note here is that the classes of the function objects are inherited from the `Boost::function`. It is necessary for the calculation of the return type of the result.

The code presented above is verbose. One can question the creation of such function objects when there are a lot already present in the standard libraries like STL. The function object `Increment` can be replaced by an already existing STL function object `std::plus<int>`. The call to map will become

```
  darray = map(bind(std::plus<int>(), 1, _1), darray);
```

The expression `bind(std::plus<int>(), 1, _1)` binds the `std::plus<int>()` function object with 1. It means it turns `std::plus<int>()` into a function object

that adds one to its argument. The $_1$ represents the arguments that are passed during the execution of the map skeleton.

The terms function objects/functors can be used interchangeably. These are the first class functions that are represented in the form of the objects of the classes.

4.2 Heat Equation: A case study

The section presents two case studies. The aim of these case studies is twofold: to demonstrate the extraction of the skeletal algorithm for a problem, and to demonstrate how the application can be optimised.

4.2.1 One Dimensional Heat Equation

We remind (see section 3.2.3) that the simulation of one dimensional heat diffusion could be performed by solving the following discretised equation:

$$u(x,t+1) = \text{diffuse} \times \frac{\Delta_t}{\Delta_x^2} \times \big(u(x+1,t) + u(x-1,t) - 2 \times u(x,t)\big) + u(x,t)$$

In the code that follows, the initial temperature of the metallic line is fixed to 30 degree celcius. The temperature of the heat sources are 60 degree celcius. Now to systematically extract the skeleton based algorithm from the equation presented above, the first step is to think of the representation of the data. Data can be represented by a simple distributed array of doubles initialised with a value of 30.0. So, the constructor for creating the object is presented below

```
DArray<double> bar(Initialize(), length);
```

where length denotes the `length` of the bar i.e the number of elements of the distributed array. `Initialize()` is the function object that initialises the elements of the distributed array with the initial temperature 30 degree celcius. The code for the `Initialize()` is presented below

```
struct Initialize : Boost::function<double(int)>
{
  double operator()(int i) const
  {
    return 30.0;
  }
};
```

Once the creation and initialisation of the data is done, the next step is to focus on the operations. From the equation of heat diffusion presented at the start of the section, it can be observed that there are some simple addition and multiplication operations. The operands of the addition operations are the arrays and the operands of the multiplication are a constant and the array. The mapping of the parts of the equation to the operation and then to skeletons are presented below:

1. $u(x+1,t)$: Shifting of the array towards left to access the right hand neighbour of the element. The boundary value at the right ending point is the temperature of the heat source. The operation can be represented by a shift operation.

```
right = shift(-1, Boundary(), bar);
```

   where -1 represents the direction of the shift i.e left and the `Boundary()` inserts the boundary value at the right end of the bar.

2. $u(x-1,t)$: Same as the previous step but to access the left hand neighbour i.e a right shift operation.

```
left = shift(1, Boundary(), bar);
```

3. $u(x+1,t)+u(x-1,t)$: Addition of the right hand and left hand neighbouring elements. The operands of the addition are the distributed array created in the last two steps. That kind of operation can be captured by a **zip** skeleton

```
neighboursum = zip(std::plus<double>(), left, right);
```

4. $-2\times u(x,t)$:Multiplication of the original data with -2. As the operation involves only one array, it can be represented by a **map** skeleton

```
bar2 = map(bind(std::multiplies<double>(),-2,_1), bar);
```

5. $u(x+1,t)+u(x-1,t)-2\times u(x,t)$: Represents the result of the addition of the neighbouring elements with the negative double of the current data. Both the sum of neighbouring elements and the negative double of current data are already calculated in the previous steps. Their addition is captured by **zip** skeleton

```
bar3 = zip(std::plus<double>(), neighboursum, bar2);
```

6. diffuse $\times \frac{\Delta_t}{\Delta_x^2} \times \big(u(x+1,t) + u(x-1,t) - 2 \times u(x,t)$: Represents the multiplication of the resultant array of the last step with the gamma.

```
diffusedbar = map(bind(std::multiplies<double>(),
                       delta_x*delta_x)/(diffuse*delta_t),
                       _1
                      ),
              bar3
             );
```

7. diffuse $\times \frac{\Delta_t}{\Delta_x^2} \times \big(u(x+1,t) + u(x-1,t) - 2 \times u(x,t)\big) + u(x,t)$: Addition of the original data with the diffused bar of the last step

```
bar = zip(std::plus<double>(), diffusedbar, bar);
```

The final result of a single time step is captured in the bar itself as during the next iteration the operations are performed over the data of the previous iteration.

The complete code of the main heat equation algorithm during one time step is presented below

```
right = shift(-1, Boundary(), bar);
left = shift(1, Boundary(), bar);
neighboursum = zip(std::plus<double>(), left, right);
bar-2 = map(bind(std::multiplies<double>(),-2,_1),bar);
bar3 = zip(std::plus<double>(), neighboursum, bar-2);
diffusedbar = map(bind(std::multiplies<double>(),
                       (delta_x*delta_x)/(diffuse*delta_t),
                       _1),
              bar3);
bar = zip(std::plus<double>(), diffusedbar, bar);
```

The problem with the code presented above is that it is not efficient. The reason is the creation of the temporaries during each operation. The elimination of temporaries is supported in OSL, but the code has to be modified. Basically, the skeletons are composed to eliminate the binaries. So, the composition of the above mentioned code will eliminate the temporaries and the resultant code is much efficient. Another benefit of the composition is the fusion of the loops. Every skeleton operation above proceeds in O(n) steps. So, instead of looping 7 times, they can be fused in a single iteration

```
bar = zip(std::plus<double>(),
        map(bind(std::multiplies<double>(),
                    (delta_x * delta_x) / (diffuse * delta_t),
                    _1),
            zip(std::plus<double>(),
                zip(std::plus<double>(),
                    shift(1, Boundary(), bar),
                    shift(-1, Boundary(), bar)
                   ),
                map(bind(std::multiplies<double>(), -2, _1),
                    bar
                   )
               )
           ),
        bar
       );
```

4.2.2 Two Dimensional Heat Equation

The section above presents the program for the heat diffusion in one dimension. In a similar manner the two dimensional heat diffusivity problem can be presented by the equation below:

$$u(x,y,t+\Delta t) = \frac{\gamma \Delta t}{\Delta s^2}\left(\begin{array}{l} u(x+\Delta s,y,t) + u(x-\Delta s,y,t) + \\ u(x,y+\Delta s,t) + u(x,y-\Delta s,t) - 4u(x,y,t) \end{array} \right) + u(x,y,t)$$

The two dimensional region is parametrised by the `width` and the `height` parameters. It is represented by a one dimensional distributed array of length (`width`×`height`). The systematic development skeleton based algorithm of the two dimensional heat equation is presented below. The skeletons are extracted from the equation presented above:

- Access to the neighbouring elements, as shown in figure 4.1, are done using the **shift** skeleton: Left and right neighbouring elements are accessed using the **shift** skeleton when the offset is 1 or -1, and the replacement value is a single value rather than a function or functor. However in this two dimensional case the shifting is incorrect: we have to use **mapIndex** to replace the values at left and right boundaries by the one computed by the left and right boundary functions. To access the top and bottom neighbouring elements, the array should be shifted `width` times. This operation is done using again the **shift** skeleton.
- The multiplication of the element by the diffusivity and -4 is captured by the **map** skeleton

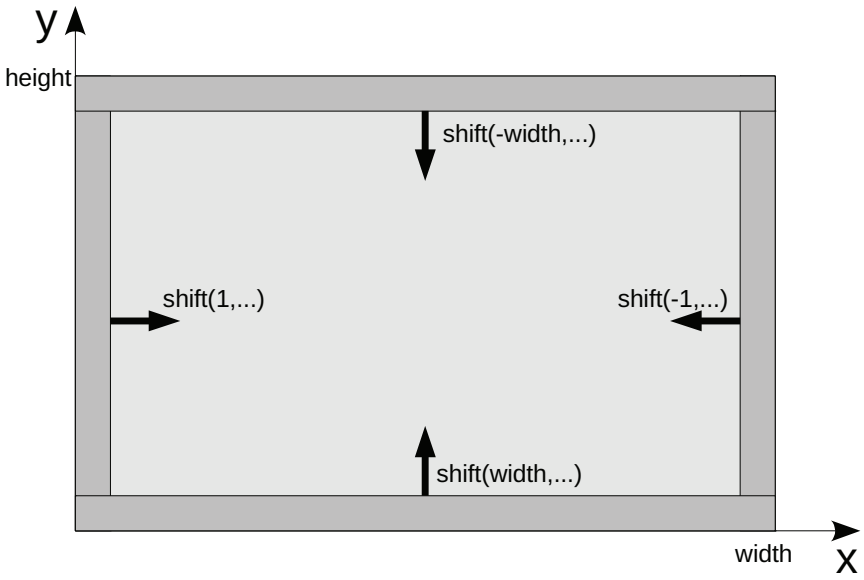


Figure 4.1 – *Communications for Heat Diffusion*

- The addition of the neighbouring elements and the final addition operation is performed by the **zip** skeleton.

The code snippet for one step of simulation of heat diffusion is presented in figure 4.2. As visible in the listing all the skeleton operations are composed in a single expression. The composition of the skeletons in this manner triggers the expression templates implementation of the skeletons which results in optimised performance.

During each step of the heat diffusion simulation, there are four synchronisation barriers, corresponding to the four shift operations. The two calls to **shift** with 1 and -1 as offset both communicate one element to the neighbouring processor while the two other calls to **shift** communicates *width* elements. Thus the BSP cost of the algorithm can be captured by the following formula:

$$\mathcal{O}(\frac{n}{p}) + (2 \times s \times g) + (2 \times width \times s \times g) + 4 \times L$$

where s is the size of a single element.

4.3 Fast Fourier Transform

The current section and the next section presents the examples that use the skeletons developed using the existing ones. This section presents `permutePartition` a skeleton developed using the code listed below:

```
flatten(permute(permute_fun, getPartition(data)));
```

```
plate = zip(std::plus<double>(),
         plate,
         map(bind(multiplies<double>(), (diffuse*dt)/(ds*ds),_1),
            zip(std::plus<double>(),
               map(bind(std::multiplies<double>(), -4, _1),
                  plate
                 ),
               zip(std::plus<double>(),
                  mapIndex(rightBound,
                           shift(1,rightBound,plate)
                          ),
                  zip(std::plus<double>(),
                     mapIndex(leftBound,
                              shift(-1, leftBound, plate)
                             ),
                     zip(std::plus<double>(),
                        shift(-width, topBound, plate),
                        shift( width, bottomBound, plate)
                       )
                    )
                 )
              )
           )
        );
```

Figure 4.2 – *One Step of Heat Diffusion Simulation in OSL*

where the `permute_fun` is the permutation function having the domain from 0 to $p-1$ where p represents the number of processors.

We borrow the implementation of FFT algorithm in terms of skeletons from Muesli. First step in FFT is to decompose the original n size data in n discrete signals in an interlaced fashion. In case of sequential algorithm it takes $\log_2 n$ stages. But in parallel as the original n is already distributed among p processors, it takes $\log_2 p$ stages. Next step is to find the frequency spectra of 1 dimensional time domain signals. Frequency spectra of 1 point signal is equal to itself. Last step is to combine n frequency spectra in exact reverse order that time decomposition takes place. This process requires $\log_2 n - \log_2 p$ iterations in parallel.

First step of this algorithm can be implemented by creating a copy of the original array and then calling **mapIndex**, `permutePartition`, **mapIndex**. Second step can be implemented by **mapIndex** and third step can be implemented by **zipIndex**. In our implementation we use the optimised OSL composition of second and third step:

```
int main (int argc, char *argv[])
{
  osl::init(&argc,&argv);
  DArray< double > bar( problemsize, 1.0 );
  init_complex initc(bar);
  DArray<complex > bar_comp(initc,problemsize);
```

```
 DArray<complex > bar_t(problemsize);
 log2p = (int)log2(mysize);
 log2size = (int)log2(problemsize);
 for(int j = 0; j < log2p; j++){
    bar_t = bar_comp;
    bitcomplement bitcomp(log2p - 1 - j);
    permutePartition(bitcomp,bar_t);
    combine comb(j);
    bar_comp = zipIndex(comb,bar_comp,bar_t);
  }
 for(int j = log2p; j < log2size; j++){
    fetch fch(bar_comp,j);
    combine cmbin(j);
    bar_comp = zipIndex(cmbin,bar_comp,mapIndex(fch,bar_t));
  }
 // Outputting the result
 osl::finalize();
}
```

`cmbin` and `fch` are function objects. In OSL we have not implemented currying (which is implemented in Muesli). Thus in order to pass a curried function in OSL, the programmer should create a function object encapsulating certain parameters via the constructor. Then function object could act as a curried function.

4.4 Reduce and Map Over Pairs

This section presents an application to demonstrate the development of other skeletons using the existing ones. The N-Body simulation is an application that simulates the motion of n points masses interacting under the presence of gravitational force. Assuming a set of n points with mass m_i, position $\vec{p}_i(t)$, and velocity $v_i(t)$ as continuous functions with respect to time t, with $0 \leq i < n$, the evolution of the system is described by a set of differential equations which could be transformed into the following discrete form with time step dt:

$$\begin{cases} \vec{p}_i(t+dt) &= \vec{p}_i(t) + \vec{v}_i(t)dt \\ \vec{v}_i(t+dt) &= \vec{v}_i(t) + \dfrac{\vec{F}_i(t)}{m_i}dt \\ \vec{F}_i(t) &= \gamma \sum_{j \neq i} \dfrac{m_i m_j (\vec{p}_j(t) - \vec{p}_i(t))}{|\vec{p}_j(t) - \vec{p}_i(t)|^3} \end{cases}$$

where γ is the gravitational constant.

Thus, the problem of calculating the sum of the forces for all the particles is of the order n^2.

In our OSL implementation, the particles are partitioned among the processors, with $\frac{n}{p}$ particles per processor. To calculate the sum of the forces, each processor communicates its particles with the others. This can be achieved in two ways:

- By using an all to all communication, so that every processor gets the current positions and velocities of all the other particles, to be able to update the positions and velocities of the particles it owns;
- By using a systolic loop, i.e in $p-1$ steps (in a BSP setting, super-steps), where at each step, each processor: (1) Computes the forces applied to the particles it owns all the time by $\frac{n}{p}$ other particles it owns at the current step (at the first step the two sets of particles are equal); (2) Receives $\frac{n}{p}$ new particles from its left neighbour, and sends the $\frac{n}{p}$ particles it was owning on the current step to its right neighbour.

Both versions have the following BSP computational cost for one step of the simulation: $\mathcal{O}(\frac{n^2}{p})$. In the first version each processor receives $\frac{n}{p} \times (p-1)$ particles, thus the communication and synchronisation BSP cost is: $\mathcal{O}(\frac{n}{p} \times (p-1) \times g) + L$. In the systolic version, at each step, each processor sends and receives $\frac{n}{p}$ particles, and there are $p-1$ steps. Therefore the communication and synchronisation BSP cost is: $\mathcal{O}((p-1) \times (\frac{n}{p} \times g) + L)$.

The total exchange version would thus have a better performance. However the advantage of the systolic version is that it allows a much smaller memory consumption than the total exchange version. In the next sections we proceed with the systolic version, its generalisation and the N-body problem specific optimisation.

We present now how to extract a skeletal implementation out of the generic description of the algorithm. Following are the main operations along with their skeleton implementation for N-body simulation:

1. make a copy B of the original particles A,
2. for each A_i compute interactions with all B_j ; these interactions are captured by rotating the copied partition implemented in terms of circular shift right (to avoid the self computation of the force a boolean is used).
3. calculate force between two particles A_i and B_j,
4. sum all the forces applied to a particle B_i,
5. update the positions and the velocities of the particles.

The steps 2–4 can be optimised by composing them in the following way:

```
for(int i=0; i < A.getLocalSize(); ++i)
  zip(sumF, force, zip(calcF, A, shift_rcl(true,B)));
```

The parallel version can be developed in the same way just by replacing the circular shift operation by a `permutePartition`. The `permutePartition` can be implemented by a **permute** skeletons preceded by a **getPartition** and followed by a **flatten**.

First we consider a systolic version: Two systolic loops are used to compute the sum of the forces on each particle. The outer loop models the partition level force computation while the inner loop represents the force computation between the processor's local particles and the received particles. Thus, the outer loop executes $p-1$ times while the inner loop executes local size -1 times. In each iteration of the inner loop three operations are performed: (1) shifting the local particles circularly towards right: a circular shift right function is written for this purpose; (2) computing the force between the two particles by using **zip** skeleton; (3) add the newly computed force to the previous one by a **zip** skeleton. A partial listing for calculating the force is presented in figure 4.3. Once the force for each particle is calculated a **zip** skeleton using a `movement` functor can be used to update the positions and velocities of all the particles.

```
// true: avoids self computation of force
totalForce = computeLocalForce(particles,tmp,true);
for(int i = 1; i < bsp_p-1; ++i) {
  // sends partition to the right neighbour
  tmp = permutePartition(bind(permuteRight(),i,_1),tmp);
  // computes local force
  localForce = computeLocalForce(particles,tmp,false);
  totalForce = zip(sumF,localForce,totalForce);
}
// update positions & velocities
particles = zip(bind(movement(),dt,_1,_2),particles,totalForce);
```

Figure 4.3 – *N-Body Simulation: Code Excerpt*

A second implementation relies on the implementation of a new skeleton: Darlington [59] used RaMP (Reduce and Map over Pairs) for the computation of the sum of the forces for each particle. This is actually a generalisation of the systolic version. As this pattern may occur often in the parallel scientific applications, we added the new skeleton RaMP to the OSL, not as a primitive skeleton but rather as a user-defined skeleton. The RaMP skeleton requires a reduction operator, a binary function, and the two expressions. The binary function is used to calculate the interaction between the two expressions. The result is then reduced by the reduction operator. As in the case of N-body the binary function is `calcF` to calculate the force between the particles and the reduction operator is `sumF` for summing up the forces. The body of the `RaMP` **operator** `()` is same as the one presented in systolic loop.

A third implementation improves the systolic version: As the force "applied" by the body i to the body j is the opposite of the force applied by the body j to the body i, it is of course possible to avoid computing these two forces. We did so by adding circular shift left function at the inner level and `permutePartition` at the outer level. The newly computed force by body i is sent back to the body j as follows:

```
(* sort: ('a→ 'a→ int) → 'a distArray → 'a distArray *)
let sort cmp da =
  let partitions = map (sortArray cmp) (getPartition da) in
  let fstSamples = map getSamples partitions in
  let sndSamples = bcast (map (compose getSamples (mergeArrays cmp))
                              (getPartition (gather fstSamples))) in
  let pieces = flatten (zip (cut cmp) partitions sndSamples) in
  flatten (map (mergeArrays cmp)
               (getPartition (permute (fun i→ (i/bsp_p)+bsp_p*(i mod bsp_p))
                                      pieces)))
```

Figure 4.4 – *Regular Sampling Sort*

	b0	b1	b2	b3	b4	b5
b0		1	2	3	2	1
b1	1		1	2	3	1
b2	2	1		1	2	3
b3	3	2	1		1	2
b4	2	3	2	1		1
b5	1	2	3	2	1	

	Originally calculated
	Never computed
	Communicated result
1,2,3	No of iteration

This reduces the inner loop iterations to $\frac{\text{local\_size}}{2}$ and the outer loop iterations to the $\frac{p}{2}$.

4.5 Sorting

All the previous applications are purely regular in nature. To show the expressiveness of our library in terms of capturing the unequal distribution of data, we implement a parallel regular sampling sort [119]. We assume that there are at least **bsp_p**−1 elements on each processor, that the array is evenly distributed among the processors, and that the elements are distinct. It is not a limitation since elements could be made distinct by transforming each element into a pair composed of the index of the element in the distributed array and the initial value. Moreover the distributed array can be evenly distributed using the **balance** skeleton. We begin by presenting the implementation in BSML prototype of OSL.

The comparison functions we use in this program are such that applied to two values v_1 and v_2, the result is negative if v_1 is smaller than v_2, is zero if the values are equal, and positive otherwise.

If we assume to have the following sequential functions, the parallel regular sampling sort can be implemented as shown in figure 4.4:

- `sortArray: ( α → α → int) → α array → α array` takes a comparison function and returns a sorted version of the array argument,
- `getSamples: α array → α array` returns **bsp_p**`-1` samples, regularly taken from its array argument,
- `mergeArrays: ( α → α → int) → α array array → α array` takes a comparison function, and an array of sorted arrays, and it returns the sorted array obtained by merging the input arrays,
- `cut: ( α → α → int) → α array → α array → α array array` cuts the first array argument into pieces according to the samples of the second array argument: there is one more piece than the number of samples
- `compose` is usual function composition.

The C++ version of the parallel sample sort is quite similar to the prototype version presented above. The C++ code for the algorithm is presented below:

```
DArray<std::vector<int> > partitions;
partitions = osl::map(SortArray(), getPartition(da));
auto result =
flatten
 (map(MergeSort(),
     getPartition
       (permute(PFun(),
          flatten
            (zip(Cut(),
                partitions,
                bcast(
                   map(bind(GetSamples(), da.length(), _1),
                       map(MergeSort(),
                           getPartition
                             (gather(
                                map(bind(GetSamples(),
                                         da.length(),
                                         _1),
                                    partitions
 ))))))))))));
```

4.6 Experiments

All the experiments were conducted on Mirev C. The MPI library used was Open MPI 1.3. The compiler was GCC 4.2.3. All the examples were compiled using the second level of optimisation.

In the experiments we used each core as a BSP processor. However the number of processes by multi-core processor is balanced (separately on each part of the cluster: for less than 64 BSP processors only the first half is used with a balanced number of BSP processors on each physical processor. For more than 64 BSP processors the second part of the cluster is also used. The BSP parameters are thus worsened when p is increased since only one network card is used by several processors.

We also ran the examples written with two other libraries: SkeTo [104, 63] and Muesli [90, 48]. For data-parallel operations on distributed arrays, OSL, SkeTo and Muesli are very similar.

SkeTo offers data-parallel operations on other data structures: matrices and trees. We used the public release[1]: 0.21. The SkeTo release contains the heat equation example. It is not possible to program efficiently the FFT example since there is no communication skeleton in SkeTo similar to `permute_partition` (the only available communication skeletons are **shift** and **gather** skeletons).

Muesli offers data-parallel and task-parallel skeletons [89]. The set of skeletons operate on the distributed arrays, distributed matrices and distributed sparse matrices. In Muesli the size of distributed arrays should be a multiple of the number of processors. This constraint does not exist for OSL and SkeTo. There is no **shift** skeleton in Muesli. The **shift** skeleton could be obtained by a composition of **map** and **permute**. However the **permute** and `fold` skeletons and all their variants could not be used if the number of processors is not a power of two. Thus heat equation example could be executed with Muesli only for particular cases. The experiments were only performed for the FFT example, included in the Muesli release[2]: 1.79.

We have compared the performances of our heat equation programs with SkeTo for heat diffusion in copper. The program takes as input the length of the metal, Δ_x, the duration of the simulation, and Δ_t the time step. We experimented on a 100mm bar of copper and fix the time of simulation to 1 second. We have experimented with both the oblivious and non oblivious versions of our program, where oblivious synchronisation is the one in which the processors known in advance the number of messages exchange during the super-step. The timings (average of 5 runs) are presented in figures 4.5 and 4.6 for some input values. The oblivious version of OSL always attain better performances than SkeTo. The non-oblivious version is closer to SkeTo in term of performances. For only one iteration, the non-oblivious OSL version is about 10 times faster than SkeTo for a large number of processors: it is due to our optimised composition of skeletons. For 1000 iterations, OSL is still more than 40% faster than SkeTo.

[1] http://www.ipl.t.u-tokyo.ac.jp/sketo
[2] http://www.wi.uni-muenster.de/pi/forschung/Skeletons/index.html

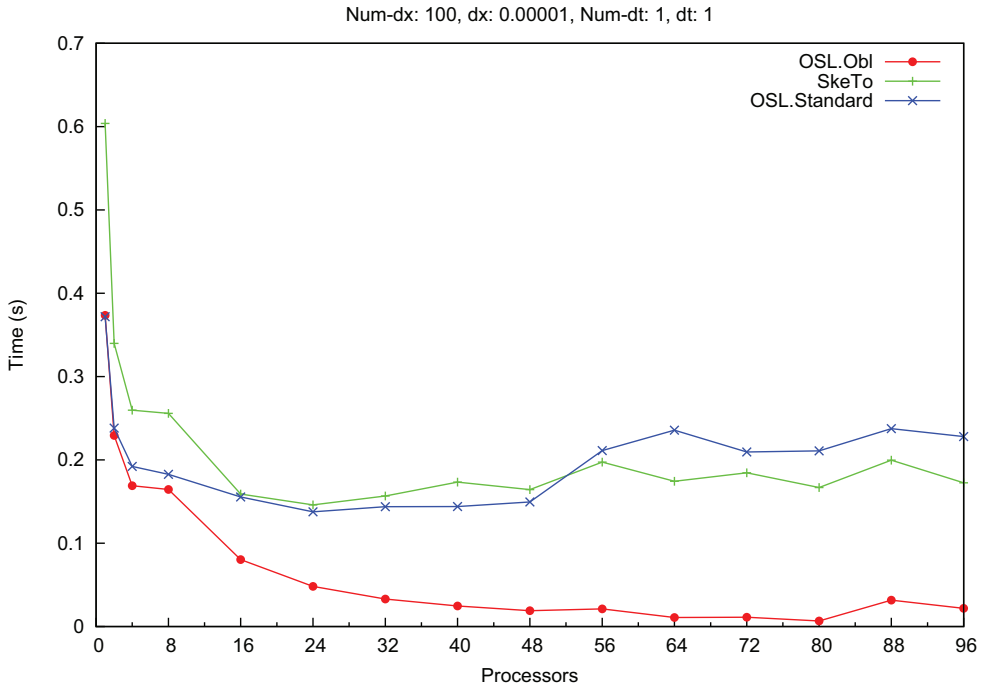


Figure 4.5 – *Heat Equation Timings (dt=1)*

If for a given number of processors we examine the timings by varying the sizes of distributed arrays, we could see that the performances follow the BSP cost given in the previous section.

The FFT program takes as argument the size of the array. It should be a multiple of the number of processors. The number of processors should be a power of 2. We measured the performances of OSL FFT with both type of synchronisations and also of the Muesli version of FFT. For small sizes, depending on the number of processors, Muesli and oblivious OSL have similar performances but one may be slightly better than the other. For large sizes, oblivious OSL have better performances than Muesli. In figure 4.7, for 64 processors, OSL is more than 20% faster than Muesli.

4.7 Summary

The chapter presents the expressiveness of OSL using different case studies. Each case study demonstrates the expressiveness of OSL in a specific way. The simulation of heat equations presents the extraction of the skeleton programs from the mathematical formulation of the problem. The FFT and Reduce and Map over Pairs present the development of the new skeletons using the existing ones. The Reduce and Map over Pairs also provides the insight towards the algorithmic optimisations of the problem and how they can be expressed in terms of skeletons. The sorting problem deals with the case of unequal distribution of the data and the communication of unequal number

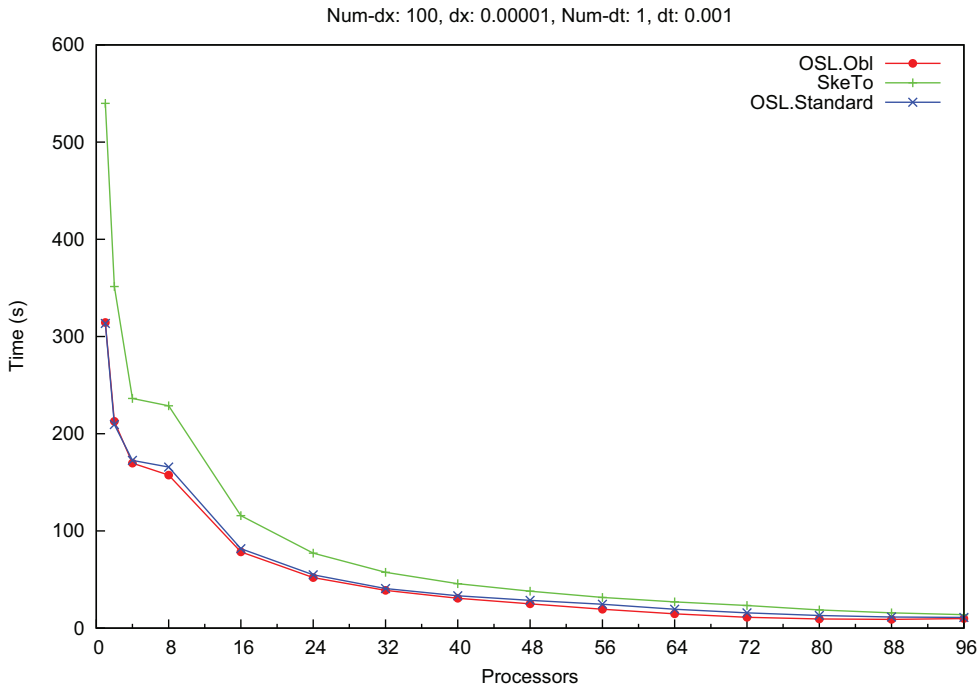


Figure 4.6 – *Heat Equation Timings (dt=0.0001)*

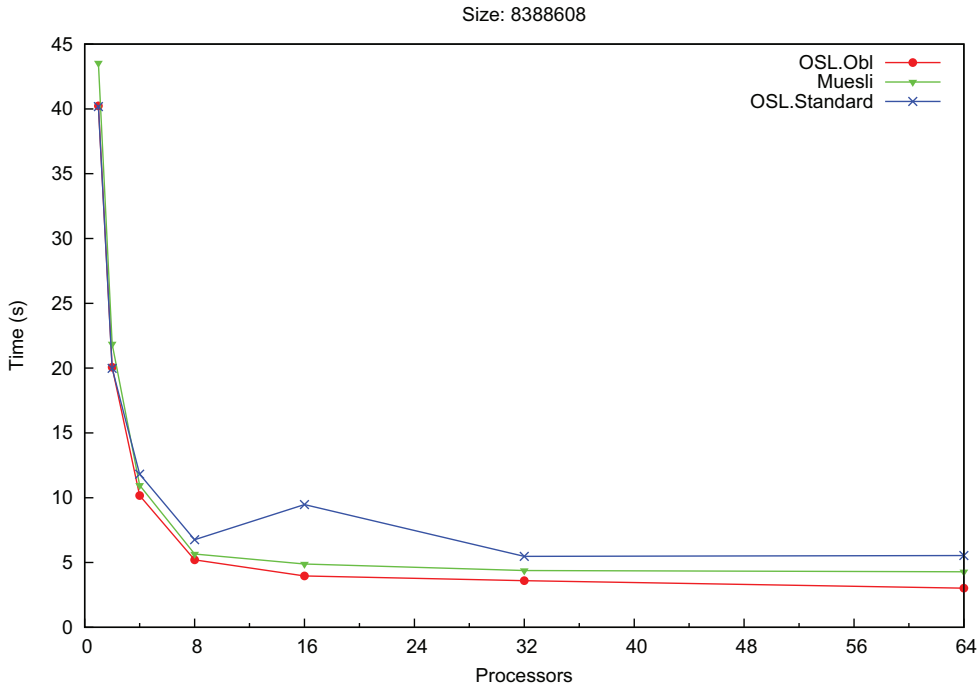


Figure 4.7 – *FFT Timings*

of elements. The chapter concludes by presenting the experimental results obtained while comparing OSL with some other skeleton libraries.

Performance Prediction and Portability

5

Contents

6.1 A Formal Programming Model 87
6.1.1 Syntax 88
6.1.2 Type System 89
6.1.3 Operational Semantics 89
6.2 Implementation in the Coq Proof Assistant 91
6.2.1 Distributed Arrays 91
6.2.2 Syntax and Typing 93
6.2.3 Big-Step Semantics 95
6.3 Verification of a Heat Diffusion Simulation 98
6.4 Summary 100

With the increase in architecture and software complexity, it becomes difficult to develop the large scale parallel applications. The accurate modelling and prediction of their performance is much complex than their development. The key contributing factors are taking account of the communications and the synchronisation details. There are different theoretical models that claim to predict the performance of the parallel applications. A well established, conceptually simple and pragmatically accurate model is the Bulk Synchronous Parallelism (BSP) as presented in chapter 1.

The performance of parallel application can not be modelled as a single parameter, as is the case with the sequential applications. The high performance in parallel programming is achieved by exploiting the features specific to the underlying architecture. Thus it is difficult to achieve a comparable performance on a different class of architectures. The idea to achieve the high performance independent of the architecture is termed as performance portability or performance tuning. The change of the architecture in sequential computing affects the performance to a constant factor and often negligible. But in case of the parallel computing it is extremely important as the performance can be very poor with the change of architecture.

Orléans Skeletons Library (OSL) follows a systematic approach to equip itself with the performance prediction and performance portability features. The steps constituting the systematic approach of OSL are presented below in order:

- A BSP cost formula is assigned to every skeleton in OSL
- BSP parameters of the system are captured
- BSP cost of the application is estimated using the cost of the skeletons
- On the basis of the estimates of the different variants of a skeleton, the best one is selected for performance portability

The rest of the chapter will focus on the above mentioned four steps. The BSP costs of OSL skeletons are presented in the chapter 3. The benchmarking of the BSP parameters is presented in the next section, followed by a case study of performance prediction. The last section explains the mechanism of performance portability in OSL using a case study application.

5.1 Benchmarking the BSP Parameters

In order to predict the cost of OSL programs we need to obtain the parameters of the BSP system. A BSP system is characterised by four parameters: number of processors p, the processor speed r, network permeability g and the synchronisation overhead l. The number of processors are usually supplied by the user of the application. There are various approaches of obtaining the BSP parameters on a parallel machine. The principle work in this regard has been done by the authors of the Oxford BSP Toolset [78], PUB library [31] and by Bisseling [30].

The Processor speed (r) The computation speed of an individual processor in a BSP computer is usually determined by measuring the number of floating point operations it can perform per second and is denoted by r. The other BSP parameters g and L are represented as the multiples of r. So, this is the first parameter that need to be measured. Many tools exist for determining the flop rate of a processor like LINPACK [81] and SPEC [75].

The Oxford BSP Toolset measures the average of the flop rates for a dot product and dense matrix multiplication using an IJK loop. A program bspprobe is dedicated for this purpose. The input data size is usually kept larger than the CPU cache size to obtain a lower bound. The PUB library measures an approximation of r based on measuring the time for memory copy operations.

The benchmark program of OSL (OSLprobe)is inspired by the work of the Bisseling. The value of r is computed by the so called DAXPY operation (double precision A times X plus Y).

Network Permeability (g) The parameter g, network permeability, is the time needed to transfer one word of data under the condition of continuous traffic. The Oxford BSP tool-set benchmark program measures g for two kinds of communications: a cyclic shift in which every processor sends and receives messages from exactly one other processor, and an all-to-all exchange in which every processor sends and receives messages from all the other processors.

Another approach for obtaining the parameters g and L is to measure communication times of full h-relations. Several supersteps are timed with an increasing h relation. To become insensitive to the measurement error between different sizes of h, it is better to perform a least square approximation. If communication time shows approximately linear increase with the number of messages, the slope of the fitted line gives g.

We estimate the value of g by circularly permuting the whole partition from a processor to the other. In this way we achieve the full h-relation condition as every processor sends and receives equal number of messages.

The synchronisation overhead (L) The value of L is determined along with the value of g. The intersection of the slope line of g with the time-axis gives L.

OSL Probe A benchmarking program (oslprobe) is developed to capture the BSP parameters of the system. The oslprobe uses the approaches presented above to determine the values of r, g and L.

The BSP parameters of the two clusters Mirev and Speed (see chapter C for details of these machines) are presented in the figure 5.1

5.2 Performance Prediction: A Case Study

Once the BSP parameters of the system are known, they can be used to predict the performance of the applications. A Heat Equation application is presented as a case study to demonstrate the performance prediction capability of OSL. Heat Equation is already explained in the chapter 4. The code of the heat equation is represented here to avoid the back consulting.

Assuming the distributed array is evenly distributed among the processors: There are two map, three zip, two shift skeletons and a copy operation for copying the results that constitutes the computational cost of the algorithm. And the operations applied by these skeletons have the same constant complexity: one floating point operation (and we count the copy as one floating point operation). So the computational cost of every skeleton is $\frac{n}{p}$, and the total computational cost of the algorithm becomes:

$$8 \times \frac{n}{p}$$

Machine	p	r	g	L
Mirev	1	889.04	1.17	145.46
	2	888.92	223.39	30168.08
	4	888.67	365.16	26132.6
	8	887.21	451.8	29456.06
	16	879.63	335.94	76851.26
	24	863.96	156.36	115297.7
	32	888.43	210.68	115005.6
	40	888.11	247.11	110241.8
	48	886.71	234.34	114350
	56	885.40	230.37	116948
	64	887.58	229.91	121424.2
Speed	1	1070.75	1.48	179.7
	2	1079.12	44.78	7427.05
	4	1069.2	44.15	7680.75
	8	1070.51	45.14	8052.74
	16	1058.11	44.64	8277.2
	24	1056.41	47.62	7943.35
	32	1058.34	54.74	7311.47
	40	1058.64	47.46	8309.94
	48	1059.62	48.12	8468.98

Figure 5.1 – *BSP Parameters of the clusters*

```
bar = zip(std::plus<double>(),
        bar,
        map(boost::bind(std::multiplies<double>(),
                    (diffuse*delta_t)/(delta_x*delta_x ), _1),
            zip(std::plus<double>(),
                zip(std::plus<double>(),
                    shift(1, bfun, bar ),
                    shift(-1, bfun,bar )),
                map(boost::bind(std::multiplies<double>(), -2, _1),
                    bar) ) ) );
```

Figure 5.2 – *One Step of Heat Equation*

There are two shift operations in the algorithm, contributing two barrier synchronisations in addition to the communication. Only one element is communicated to and from each processor during the shift operation. So the communication and the synchronisation cost of the algorithm becomes:

$$2 \times s \times g + 2 \times L$$

where s is the size of a single element which happens to be a double in this case. The BSP cost of the whole algorithm is the sum of the computation, communication and synchronisation costs. The BSP cost of heat equation is:

$$8 \times \frac{n}{p} + 2 \times s \times g + 2 \times L$$

On the basis of this BSP cost formula the original cost of the heat equation can be estimated by replacing the BSP parameters of the system.

The performance prediction results of the heat equation are presented in the figure 5.3

5.3 Performance Portability

A hard coded, single algorithm based implementation of the skeleton, is not sufficient to guarantee the good performance on all systems. The factors creating the hurdles are the differences in the underlying system architectures, networks parameters and the program parameters. To overcome the problem, the best suited algorithm/implementation must be selected. This means for the same skeleton different versions of the algorithm are implemented and the best one is selected following some criteria. One such criteria is the BSP cost calculated in terms of the BSP parameters.

In the skeleton libraries, the performance portability can be applied at two levels. At the global level the target for such optimisation is the transformation of a combination of skeletons to the other to minimise the BSP cost. In case of OSL, this type of optimisation is not our primary concern, as with the OSL skeleton set same can be achieved using the expression templates. At the local level, the skeletons can be optimised according to the architecture. Every skeleton that can have multiple underlying algorithms is the target of such an optimisation. In OSL reduce, scan and balance are the candidates of such optimisation.

The current work deals with the optimisation of the reduce skeleton. The reduce skeletons applies an associative binary reduction operator over the distributed array. The distributed array is already distributed among the p processors.

The signature of the reduce skeleton in OSL is presented below:

```
<T> reduce(T ⊕(T,T), T id, DArray<T> t)
```

Every processor first reduces its local partition. This first step is purely computational and doesn't involve any communication. The computed results are then shared with other processors for the computation of the final result. Different patterns of communication are possible at this step. We present the four different reduction algorithms based on four different patterns of communication.

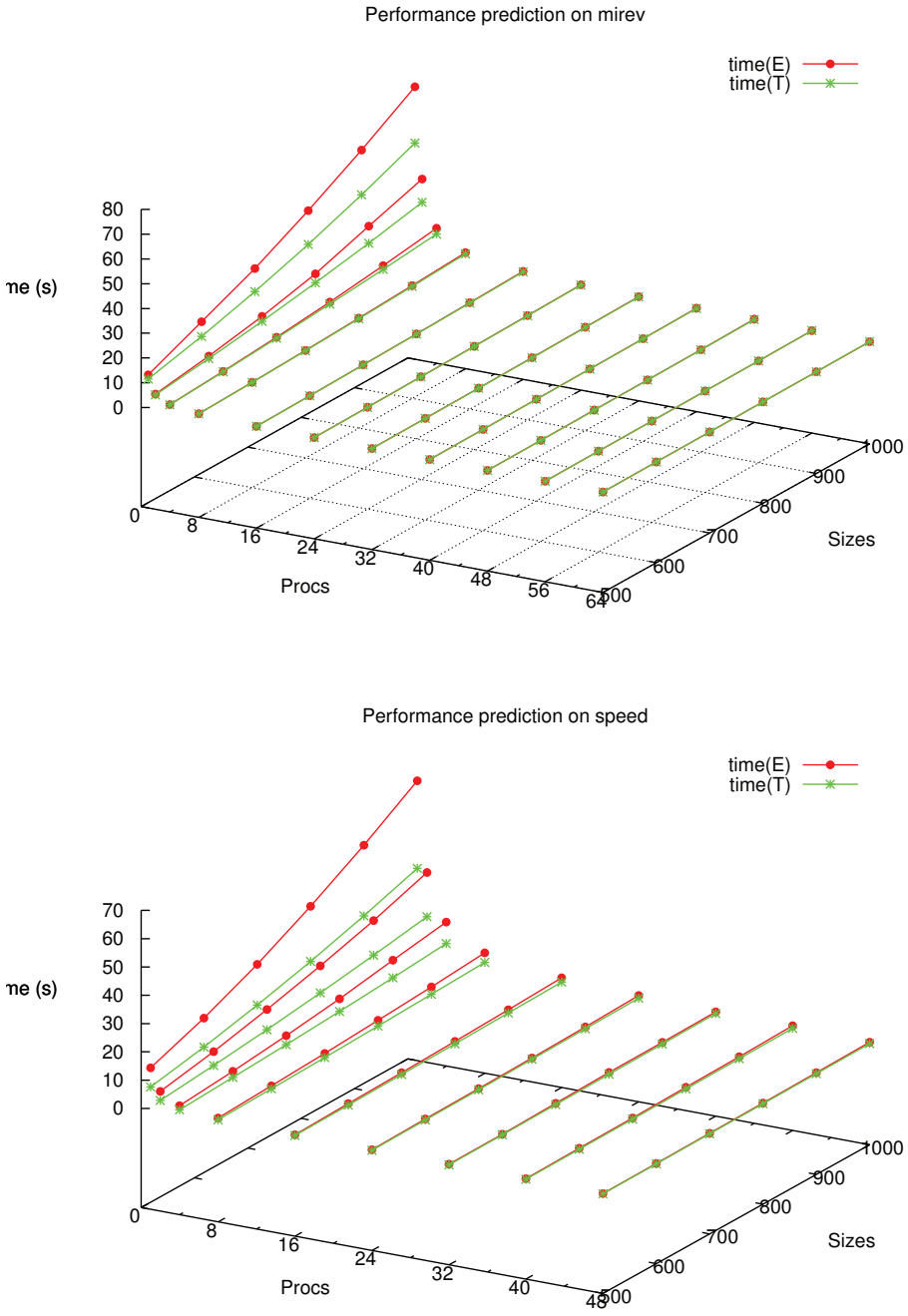


Figure 5.3 – *Performance Prediction of the 1D Heat Equation*

5.3.1 Reduce by Using Gather and Broadcast

After the computation of the local results, the results are gathered at the root processor. Root processors then reduces the results to obtain the final result which is then broadcast to the remaining processors as presented in figure 5.4. The algorithm proceeds in two super-steps. The computational phase of the first step is the local result computation. This phase is followed by the communication phase, in which all processors submit their local results to the root processor. In this way the root processor receives $p-1$ messages. This gather operation is followed by a synchronisation barrier. So the BSP cost of the first super-step is by assuming the cost of the reduction operator $c_{\oplus}$:

$$c_{\oplus} \times \frac{n}{p} + ((p-1) \times sizeof(result) \times g) + L$$

During the computational phase of the second super-step, only the root processor computes the final result. All the other processors remain free. The root processor then broadcasts the final result to all the other processors, which constitutes the communication phase. Every processor receives one message and sends nothing except the root processor which sends $p-1$ messages. So the BSP cost of the second super-step becomes:

$$c_{\oplus} \times p + (p - 1 \times sizeof(result) \times g) + L$$

And the cost of the whole algorithm is the sum of the costs of the two super-steps

$$(c_{\oplus} \times \frac{n}{p} + c_{\oplus} \times p) + 2 \times ((p-1) \times sizeof(result) \times g) + 2 \times L$$

5.3.2 Reduce by Using All Gather

In the second version the results of the reduction of the local data are gathered by all processors and every one computes the final result as shown in figure 5.5. Thus the algorithm needs only one super-step to proceed. As every processor gathers the results of the other processors, thus during the communication phase every processor sends and receives $p-1$ messages. Every processor, after synchronisation, computes the final result. So, the BSP cost of this version becomes:

$$c_{\oplus} \times \frac{n}{p} + c_{\oplus} \times p + ((p-1) \times sizeof(result) \times g) + L$$

In terms of the BSP cost, this version of the algorithm always outperforms the first version. In terms of the processor cycles, this version consumes more. But as the computations on every processors proceeds in parallel, this penalty has no effect on the final BSP cost. As OSL is based over MPI, the all gather operations of the MPI is used to collect the results. The real time performance of the all gather is not at par with that of the gather of the first version as shown in the experimentation section.

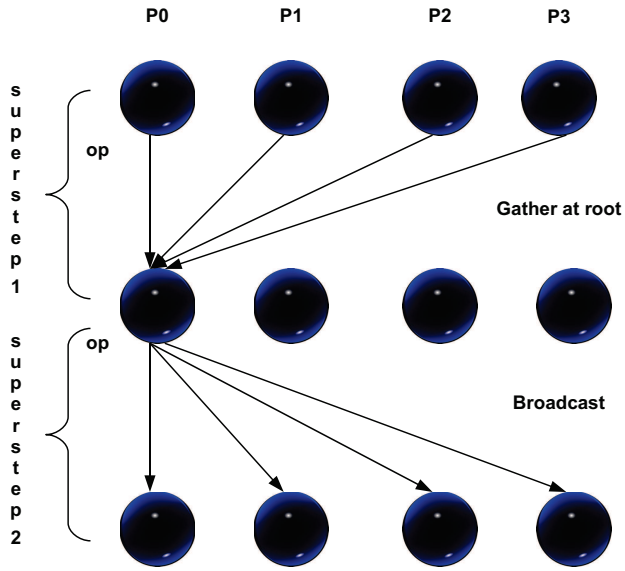


Figure 5.4 – *Gather and Broadcast*

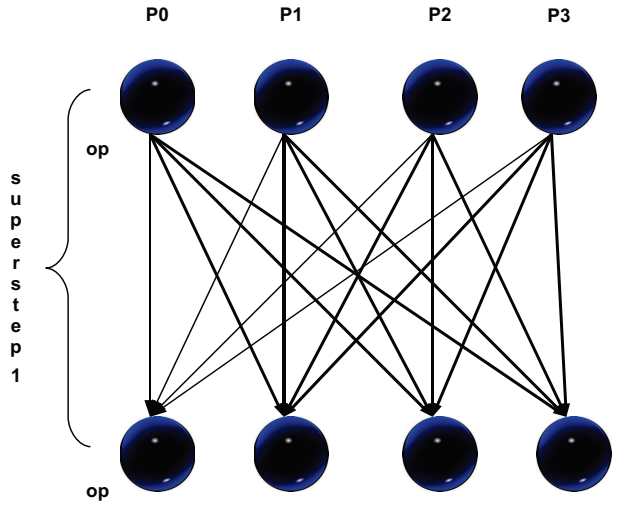


Figure 5.5 – *All Gather*

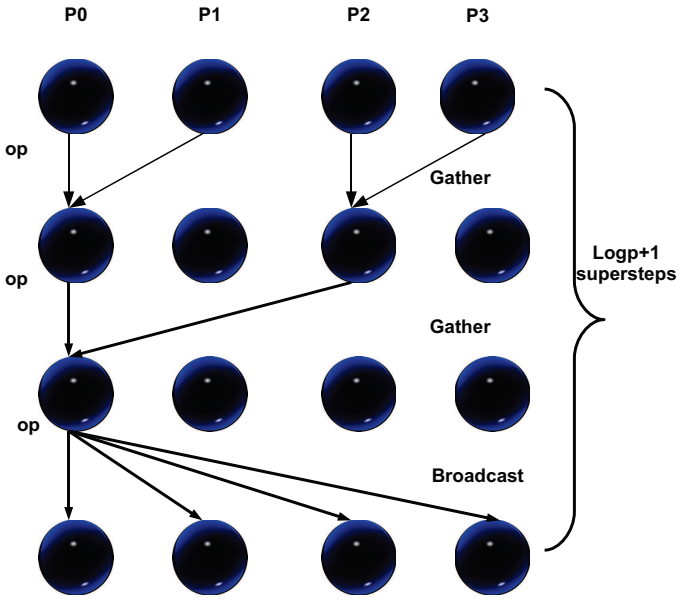


Figure 5.6 – *Tree Gather and Broadcast*

5.3.3 Reduce by Tree Gather and Broadcast

In the third version the communication pattern is the tree like reduction which proceeds in $\log_2 P$ steps. At each step, a neighbour processor is selected and the results are communicated to it, which then reduces the results. At the end of the last step the final result is calculated by the root processor, which then broadcasts the result to the other processors, represented by figure 5.6.

The tree reduction version proceeds in $\log_2 P + 1$ super-steps. During each super-step after computing the local results one processor sends its result to its pair processor. Thus the maximum number of messages received or send by a processor is one. The BSP cost of a single super-step is thus

$$c_{\oplus} + sizeof(result) \times g + L$$

.And there are $\log_2 P$ such super-steps. The last super-step is the broadcast of the final result to the other processors. So the BSP cost of the complete algorithm becomes

$$c_{\oplus} \times \frac{n}{p} + \log_2 P \times c_{\oplus} + \log_2 P \times sizeof(result) \times g + \log_2 P \times L + g + L$$

This version outperforms the first two versions when the effect of g and L is small.

5.3.4 Reduce by Tree All Gather

The last version is the variant of the third one in which during each super-step both pair processors gather the results 5.7. Every processor takes part in the computation

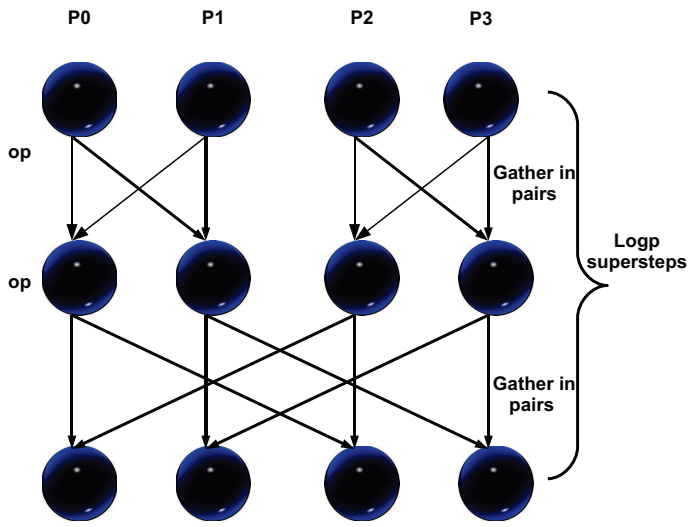


Figure 5.7 – *Tree Gather in Pairs*

in all super-steps, so in the end there is no need for a broadcast operations. That is why this version needs $\log_2 P$ super-steps.

During each super-step of the last version, each processor sends and receives one message from its pair processor, so the BSP cost of a single super-step becomes

$$c_{\oplus} + sizeof(result) \times g + L$$

In contrast to the third version, as all the processors takes part in computing and gathering results, the broadcast operation at the end of $\log_2 P$ super-steps in not needed. So the version proceeds in $\log_2 P$ super-steps and its total BSP cost is

$$c_{\oplus} \times \frac{n}{p} + \log_2 P \times c_{\oplus} + \log_2 P \times sizeof(result) \times g + \log_2 P \times L$$

The limitation of this version is that it works only for the power of two number of processors.

5.3.5 Best Algorithm Selection

The first step is to calculate the performance of all the reduction algorithms. The performances of the algorithms depend on the input parameters of the program and the underlying architecture. The input parameters are number of flops consumed by the reduction operator and the size of the data. The parameters of the architecture are actually the classic BSP parameters.

Size of the data and the number of data elements are usually known at the runtime. The number of flops consumed by the reduction operator are provided by the application developer. The developer have to add a **int** `numFlops()` function to the reduction function object. A meta-function detects at compile time whether the `numFlops` function is provided or not. If it is present, the performance of all the versions of the reduce algorithm are compared and the best one is selected. Otherwise the default reduction algorithm is applied. The advantage of implementing this strategy in the form of metaprogramming is that it has no runtime cost. The code for the respective case is generated at compile time.

The BSP parameters (p, g, L, r) of the underlying architecture are captured using the benchmarking program called oslprobe as explained in the section 5.1.

Once all the parameters are known, the execution time of each algorithm is estimated by calculating the BSP cost of each algorithm. The best algorithm is then selected for execution. The BSP costs of the algorithms can be calculated by estimating the computational time, the communication time and the synchronisation time. The computational time is estimated by counting the number of floating point operations performed by the reduction operator, multiplied by the number of data elements, divided by the BSP parameter r. The communication time is the product of maximum number of messages sent or received by the processors and the network permeability g. The synchronisation time is the time taken by the barrier to synchronise all the processors i.e the parameter L. The total estimated time is the sum of the above mentioned three.

Whenever in any application there is a call of such a skeleton, the best version of the underlying algorithm is selected using the above mentioned strategy.

5.3.6 Variance: A Case Study

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

The equation above presents the variance for n value. The variance is used as a measure of how far a set of numbers are spread out from each other. As the purpose of this toy application is to demonstrate the performance portability of OSL, the data for the variance is considered as a matrix. By varying the size of the matrix(rows and columns) the number of operation needed to calculate the variance can be varied. Each column of the matrix is representing a unique attribute and the rows represent the values about that attribute. So for each column of the matrix, the variance is calculated. A wrapper class `VarianceData` is used to wrap the matrix data. It defines a number of operations like sum, mean, scalarDifference and scalarProduct which are used to calculate the variance.

The distributed array is instantiated by representing the Matrix data through the wrapper class i.e. `DArray<VarianceData<Matrix<`**double**`> > >`. We use the default value constructor of the distributed array to instantiate. In this way every element of the distributed array is an object of `VarianceData` holding a $h \times w$ matrix.

Now to calculate the variance of the data, the first step is to sum up all the data and then compute the mean of the data. The sum operation of the whole data can be captured by the `reduce` skeleton. And this is the target skeleton of the performance portability and is detailed in the next section. The reduction operator of the sum operation is implemented in the form of a function object in the `VarianceData` class. This function object is supplied with a `numFlops()` operation which is required for performance portability calculation in the reduce skeleton. As the summation of two matrices involve $h \times w$ operations, so this method simply returns $h \times w$.

After summing up all the elements of the distributed array, the resultant matrix is summed up and then mean of the data is calculated. This step involves no parallelism and is a simple sequential operation. The next step is to compute the squared difference between each data element and the arithmetic mean. These operations are composed in a sequence of two **map** skeletons. The inner one calculating the difference while the outer one squaring it up.

Another `reduce` skeleton is required to sum up the result of the preceding **map** skeletons. The reduced result divided by the total number of data elements gives us the variance of the data.

```
DArray<VarianceData<Matrix<double> > > dvm(length, m);
VarianceData<Matrix<double> > sigma=reduce(identity.sum,identity,dvm);
std::vector<double> meanVec = sigma.mean(length * rows);
dvm = map(boost::bind(&VarianceData<Matrix<double> >::square, _1, _1),
        map(bind(&VarianceData<Matrix<double> >::scalarDifference,
                    _1, meanVec, _1),
            dvm));
VarianceData<Matrix<double> > sigma2=reduce(identity.sum,identity,dvm);
std::vector<double> varianceVec = sigma2.mean(length * rows);
```

Figure 5.8 – *Variance Program*

The program is parametrised by the number of data elements, the height and the width of the matrix. All the individual elements of the distributed arrays are matrices. Thus by varying the height and width parameter the size of the program the size of the matrix can be varied, in turn varying the number of floating point operations for each element. The type of the matrix is double to get the same effect of floating point operation.

The experiments are conducted in two ways: by varying the size of the elements of the distributed array and by increasing the number of processors. The former increases the number of floating point operations by the reduction operator, while the later is used for the scalability 5.9.

5.4 Summary

The chapter presents the reasoning about the performance of the applications from two axes: the performance prediction and the portability of performance. BSP cost

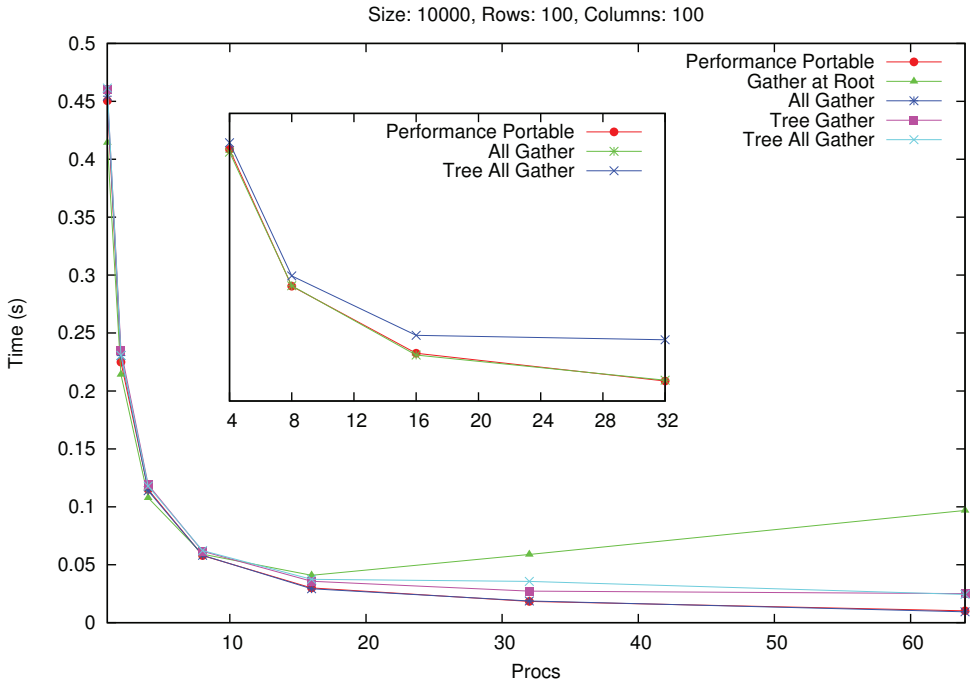


Figure 5.9 – *Performance Portability of Variance*

model is used to predict the cost of the application. Every skeleton in OSL is assigned a BSP cost. Thus the cost of the application can be predicted using the cost of the individual skeletons. The strategy is implemented using the C++ metaprogramming technique to capture the cost of the applied argument function of the skeletons. The BSP cost is calculated using the BSP parameters of the systems benchmarked through a probing program. The reduce skeleton is used to demonstrate the portability of performance in OSL. The other skeletons like broadcast, gather and balance can also benefit from the mechanism. The reason behind selecting reduce as the first to equip with the ability is its wide range of applicability and the presence of both the computations and the communications. In future versions of the OSL other skeletons will also be equipped with this feature.

Formal Semantics of OSL

6

It is important to have a formal semantics of the programming model of a language or a library. A formal semantics offers an unambiguous reference manual for the language and it may serve as a basis for the verification of programs.

In this chapter, formal semantics of OSL distributed arrays and skeletons are presented. In the first section, the semantics is presented in a classical way: the syntax, a type system and an operational semantics of OSL are defined. In section 6.2, the implementation of this formal semantics in the Coq proof assistant is presented. The use of depend types gives for free the proof of type soundness with respect to the operational semantics. This implementation in Coq also allows to evaluate OSL formal programs. This is a very interesting form of reference manual that can allow the co-testing of the formalisation and the implementations of OSL. In section 6.3, this formal semantics forms the basis for the verification of an OSL application: the one dimension heat diffusion simulation.

For convenience of the reader, a short introduction to the Coq proof assistant is available in chapter B.

6.1 A Formal Programming Model

As in [43], we would like to model the semantics of our library without being obliged to model the whole syntax of the host language. In this formalisation, the syntax (and semantics) are parametrised by the underlying sequential language on top of which OSL is built. This means no formal semantics of the C++ language is given. Instead the syntax and semantics of the sequential language is assumed.

More formally, we assume:

- a set $\mathcal{E}_s$ of expressions (elements of this set will be written *sle*), containing a subset $\mathcal{V}_s$ of *values* (elements of this set will be written v and it includes values of the form $[v_1, \ldots, v_n]$);
- a set $\mathcal{T}_s$ of types (elements of this set will be written τ_s) that is supposed:
  - to be closed by arrow, i.e if $\tau_1 \in \mathcal{T}_s$ and $\tau_2 \in \mathcal{T}_s$ then $\tau_1 \to \tau_2 \in \mathcal{T}_s$,
  - to be closed by array construction, i.e if $\tau_s \in \mathcal{T}_s$ then $\mathsf{vector}\langle\tau_s\rangle \in \mathcal{T}_s$,
  - to contain the type of integer values: $\mathsf{int} \in \mathcal{T}_s$,

- a typing relation $:_s$ of $\mathcal{E}_s \times \mathcal{T}_s$ that relates a well-formed and typed expression to a type,
- an evaluation function $\downarrow_s$ of $\mathcal{E}_s \times \mathcal{V}$ that relates an expression to its ultimate simplification or value, such that:
  - values evaluate to themselves, i.e $\downarrow_s$ is reflexive,
  - if $sle :_s \tau_s$ then there exists v such that $\downarrow_s (sle) = v$,
  - if $sle :_s \tau_s$ and $\downarrow_s (sle) = v$ then $v :_s \tau_s$.

The next three sub-sections focus respectively on: the syntax of OSL programs, the type system for OSL programs, and an operational semantics for OSL programs.

6.1.1 Syntax

The terms representing OSL programs are given by the following grammar:

$$
\begin{array}{rcl}
e & ::= & se \mid pe \\
se & ::= & sle \mid se\ se \mid \mathsf{reduce}(se, se, pe) \\
pe & ::= & pv \mid \mathsf{make}(se, se) \mid \mathsf{init}(se, se) \mid \mathsf{atRoot}(se) \\
 & \mid & \mathsf{map}(se, pe) \mid \mathsf{mapIndex}(se, pe) \mid \mathsf{zip}(se, pe, pe) \mid \mathsf{zipIndex}(se, pe, pe) \\
 & \mid & \mathsf{shift}(se, se, pe) \mid \mathsf{permute}(se, pe) \mid \mathsf{balance}(pe) \\
 & \mid & \mathsf{bcast}(pe) \mid \mathsf{gather}(pe) \mid \mathsf{getPartition}(pe) \mid \mathsf{flatten}(pe)
\end{array}
$$

An expression e could be either an expression se whose type is a sequential one, or a parallel expression pe. A sequential expression could be either an expression of the sequential language sle (possibly a value, a function), an application of a sequential expression to another sequential expression (for example to be able to apply a sequential function to the result of a parallel skeleton expression ended by a call to reduce), or an application of the reduce skeleton that takes as input a binary associative operator, an identity value for this operator, and a distributed array to be reduced.

A parallel expression could be either a parallel value (the formalisation of a distributed array), either the building of distributed array using a constructor (I put here only the constructors present in the BSML prototype but they model most of the C++ constructors as well), or a call to one of the skeletons.

A parallel value pv is of course a distributed array. A distributed array of size n containing sequential value v_i at index i will be written:

$$[\ v_0\ , \ldots ,\ v_{n-1}\ ]_D$$

D is the distribution of the array, i.e. the number of elements on each processor. $D(i)$ denotes the number of elements at processor i. Elements are distributed in a contiguous way: For example at processor 1, the sub-array is $[v_{D(0)}; \ldots ; v_{D(0)+D(1)-1}]$. The domain of D is $\{0, \ldots, p-1\}$ where p is the number of processors of the BSP machine. In the following we may consider sometimes for the convenience of notation that $D(-1) = 0$.

6.1.2 Type System

The grammar for the types follows:

$$\tau \quad ::= \quad \tau_s \quad | \quad \mathsf{DA}\langle\tau_s\rangle$$

We assume that a function of the sequential language taking two arguments of type τ_1 and τ_2 respectively and returning a value of type τ has type $\tau_1 \to (\tau_2 \to \tau)$ also written $\tau_1 \to \tau_2 \to \tau$ (functions are considered to be in curried form). $\mathsf{DA}\langle\tau_s\rangle$ is the type of a distributed array containing elements of type τ_s.

The type system for OSL is given in figure 6.1. The typing relation is written $e : \tau$ for OSL expressions. These rules are classical and correspond roughly to the types of the functions in the BSML prototype of OSL. For more readability, some expressions of the sequential language that are supposed to be functions, operators, identity elements for an operator, integers and arrays are respectively written f, $\oplus$, $i_\oplus$, n (or d) and a instead of *sle* and variants.

6.1.3 Operational Semantics

Formal operational semantics for OSL are presented in this subsection. This semantics models how the programmer should understand the skeletons, not how they are implemented. That is why it is called a formal programming model.

This semantics, as the assumed operational semantics of the sequential language, is a big-steps semantics: it relates OSL expressions to OSL values. In OSL the values are either the values of the sequential language or distributed arrays of values as presented in section 6.1.1.

This function is written $\downarrow$ and is defined in figure 6.2. The two constructors init and make both returns evenly distributed arrays. Thus the specific distribution function E_n (for evenly distributed global size n) is defined as:

$$E_n(i) = \begin{cases} (n/\mathsf{bsp\_p}) + 1 & \text{if } i < n \bmod \mathsf{bsp\_p} \\ (n/\mathsf{bsp\_p}) & \text{otherwise} \end{cases}$$

The E_n distribution is used in several other rules:

- In rule (6.28), the balance skeleton only changes the distribution. Initially it is an arbitrary distribution D, and it becomes and even distribution E_n.
- In rule (6.29), the bcast skeleton creates a perfectly balanced distribution. If the initial distribution is D, after the broadcast, each processor will hold $D(0)$ elements. Thus we obtain an even distribution of size $p \times D(0)$ where p is the number of processors of the BSP machine.
- In rule (6.31), the getPartition skeleton returns one element per processor (each element being an array). The distribution is thus E_p.

$$\frac{sle :_s \tau_s}{sle : \tau_s} \tag{6.1}$$

$$\frac{f : \tau_1 \to \tau_2 \qquad se : \tau_1}{f\ se : \tau_2} \tag{6.2}$$

$$\frac{\oplus :_s \tau_1 \to \tau_1 \to \tau_1 \qquad i_\oplus :_s \tau_1 \qquad pe : \mathsf{DA}\langle\tau_1\rangle}{\mathsf{reduce}(\oplus, i_\oplus, pe) : \tau_1} \tag{6.3}$$

$$\frac{n : \mathsf{int} \qquad sle : \tau_s}{\mathsf{make}(n, sle) : \mathsf{DA}\langle\tau_s\rangle} \tag{6.4}$$

$$\frac{n : \mathsf{int} \qquad f : \mathsf{int} \to \tau_s}{\mathsf{init}(n, f) : \mathsf{DA}\langle\tau_s\rangle} \tag{6.5}$$

$$\frac{sle : \mathsf{vector}\langle\tau_s\rangle}{\mathsf{atRoot}(sle) : \mathsf{DA}\langle\tau_s\rangle} \tag{6.6}$$

$$\frac{f : \tau_1 \to \tau_2 \qquad pe : \mathsf{DA}\langle\tau_1\rangle}{\mathsf{map}(f, pe) : \mathsf{DA}\langle\tau_2\rangle} \tag{6.7}$$

$$\frac{f : \tau_1 \to \tau_2 \to \tau_3 \qquad pe_1 : \mathsf{DA}\langle\tau_1\rangle \qquad pe_2 : \mathsf{DA}\langle\tau_2\rangle}{\mathsf{zip}(f, pe_1, pe_2) : \mathsf{DA}\langle\tau_3\rangle} \tag{6.8}$$

$$\frac{f : \mathsf{int} \to \tau_1 \to \tau_2 \qquad pe : \mathsf{DA}\langle\tau_1\rangle}{\mathsf{mapIndex}(f, pe) : \mathsf{DA}\langle\tau_2\rangle} \tag{6.9}$$

$$\frac{f : \mathsf{int} \to \tau_1 \to \tau_2 \to \tau_3 \qquad pe_1 : \mathsf{DA}\langle\tau_1\rangle \qquad pe_2 : \mathsf{DA}\langle\tau_2\rangle}{\mathsf{zipIndex}(f, pe_1, pe_2) : \mathsf{DA}\langle\tau_3\rangle} \tag{6.10}$$

$$\frac{n : \mathsf{int} \qquad f : \mathsf{int} \to \tau_s \qquad pe : \mathsf{DA}\langle\tau_s\rangle}{\mathsf{shift}(n, f, pe) : \mathsf{DA}\langle\tau_s\rangle} \tag{6.11}$$

$$\frac{f : \mathsf{int} \to \mathsf{int} \qquad pe : \mathsf{DA}\langle\tau_s\rangle}{\mathsf{permute}(f, pe) : \mathsf{DA}\langle\tau_s\rangle} \tag{6.12}$$

$$\frac{pe : \mathsf{DA}\langle\tau_s\rangle}{\mathsf{balance}(pe) : \mathsf{DA}\langle\tau_s\rangle} \tag{6.13}$$

$$\frac{pe : \mathsf{DA}\langle\tau_s\rangle}{\mathsf{gather}(pe) : \mathsf{DA}\langle\tau_s\rangle} \tag{6.14}$$

$$\frac{pe : \mathsf{DA}\langle\tau_s\rangle}{\mathsf{getPartition}(pe) : \mathsf{DA}\langle\mathsf{vector}\langle\tau_s\rangle\rangle} \tag{6.15}$$

$$\frac{pe : \mathsf{DA}\langle\mathsf{vector}\langle\tau_s\rangle\rangle}{\mathsf{flatten}(pe) : \mathsf{DA}\langle\tau_s\rangle} \tag{6.16}$$

Figure 6.1 – *OSL Typing Rules*

On the contrary, the atRoot constructor returns a distributed array that contains elements only at root processor. Thus the specific distribution function R_n (for all at

root of global size n) is defined as:

$$R_n(i) = \begin{cases} n & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases}$$

The R_n distribution is also used in the (6.30) rule for the gather skeleton. All skeletons either return an evenly distributed array or keep the distribution of the input arrays but the atRoot constructor and the gather skeleton.

The distribution of the output of the flatten skeleton may or may not be evenly distributed. As opposed to all other skeletons the final distribution depends on the *values* contained in the input distributed array. These values are *sequential arrays* thus the final distribution of the concatenation of these arrays depend on initial distribution and length of the sequential arrays. In rule (6.32), $|a|$ denote the length of array a.

6.2 Implementation in the Coq Proof Assistant

The section deals with the modelling the programming model of OSL using the Coq proof assistant. It starts by explaining how the modelling of the data structure of distributed arrays and of the syntax is done. It then presents the big-step semantics and its properties.

6.2.1 Distributed Arrays

First of all we need to model the parallel data structure of our OSL library: the distributed arrays. The content of a distributed array can be seen as a usual sequential array plus the information about its distribution. In Coq we model the content of the arrays by lists. The distribution is modelled by a data structure similar to lists but with the size of the collection inside the type: vectors. A vector of type vector A n has size n and contains values of type A. A distribution is a vector of *natural numbers*: each natural number is the number of elements per processor. The size of a vector of distribution is bsp_p, the number of processor of the BSP machine. bsp_p is strictly positive. To cleanly formalise the fact that the syntax and semantics are parametrised by the number of processors of the parallel machine, the semantics is a functor, i.e. a module that takes as argument another module. This argument module has the following type:

```
Module Type BSP_PARAMETERS.
  Parameter lastProcessor : nat.
End BSP_PARAMETERS.
```

lastProcessor is supposed to be the processor identifier of the last processor. We then define:

```
Definition bsp_p := S Bsp.lastProcessor.
```

$$\downarrow(\ [\ v_0\ ,\ldots,\ v_{n-1}\ ]_D\ ) = [\ v_0\ ,\ldots,\ v_{n-1}\ ]_D \tag{6.17}$$

$$\downarrow(\ \mathsf{make}(n, sle)\ ) = [\ \downarrow_s(sle)\ ,\ldots,\ \downarrow_s(sle)\ ]_{E_n} \tag{6.18}$$

$$\downarrow(\ \mathsf{init}(n, f)\ ) = [\ \downarrow_s(f(0))\ ,\ldots,\ \downarrow_s(f(n-1))\ ]_{E_n} \tag{6.19}$$

$$\downarrow(\ \mathsf{atRoot}(sle)\ ) = [\ v_0\ ,\ldots,\ v_{n-1}\ ]_{R_n} \text{ if } \downarrow_s(sle) = [v_1,\ldots,v_n] \tag{6.20}$$

$$\downarrow(\ \mathsf{map}(f, [\ v_0\ ,\ldots,\ v_{n-1}\ ]_D)\ ) = [\ \downarrow_s(f(v_0))\ ,\ldots,\ \downarrow_s(f(v_{n-1}))\ ]_D \tag{6.21}$$

$$\downarrow(\ \mathsf{zip}(f, [\ v_0\ ,\ldots,\ v_{n-1}\ ]_D, [\ w_0\ ,\ldots,\ w_{n-1}\ ]_D)\ ) = [\ \downarrow_s(f(v_0, w_0))\ ,\ldots,\ \downarrow_s(f(v_{n-1}, w_{n-1}))\ ]_D \tag{6.22}$$

$$\downarrow(\ \mathsf{mapIndex}(f, [\ v_0\ ,\ldots,\ v_{n-1}\ ]_D)\ ) = [\ \downarrow_s(f(0, v_0))\ ,\ldots,\ \downarrow_s(f(n-1, v_{n-1}))\ ]_D \tag{6.23}$$

$$\downarrow(\ \mathsf{zipIndex}(f, [\ v_0\ ,\ldots,\ v_{n-1}\ ]_D, [\ w_0\ ,\ldots,\ w_{n-1}\ ]_D)\ ) = [\ \downarrow_s(f(0, v_0, w_0))\ ,\ldots,\ \downarrow_s(f(n-1, v_{n-1}, w_{n-1}))\ ]_D \tag{6.24}$$

$$\downarrow(\ \mathsf{shift}(d, f, [\ v_0\ ,\ldots,\ v_{n-1}\ ]_D)\ ) = [\ \downarrow_s(f\ 0),\ldots,\downarrow_s(f\ (d-1)), v_0,\ldots,v_{n-1-d}\ ]_D \text{ if } d \geq 0 \tag{6.25}$$

$$\downarrow(\ \mathsf{shift}(d, f, [\ v_0\ ,\ldots,\ v_{n-1}\ ]_D)\ ) = [v_d;\ldots;v_{n-1};\downarrow_s(f(n-d-1));\ldots;\downarrow_s(f(n-1))] \tag{6.26}$$

$$\text{if} \quad d \leq 0$$

$$\downarrow(\ \mathsf{permute}(f, [\ v_0\ ,\ldots,\ v_{n-1}\ ]_D)\ ) = [\ v_{f^{-1}\ 0}\ ,\ldots,\ v_{f^{-1}\ (p-1)}\ ]_D \tag{6.27}$$

$$\downarrow(\ \mathsf{balance}([\ v_0\ ,\ldots,\ v_{n-1}\ ]_D)\ ) = [\ v_0\ ,\ldots,\ v_{n-1}\ ]_{E_n} \tag{6.28}$$

$$\downarrow(\ \mathsf{bcast}([\ v_0\ ,\ldots,\ v_{n-1}\ ]_D)\ ) = [\ v_0,\ldots,v_{D(0)-1};\ldots;v_0,\ldots,v_{D(0)-1}\ ]_{E_{p\times D(0)}} \tag{6.29}$$

$$\downarrow(\ \mathsf{gather}([\ v_0\ ,\ldots,\ v_{n-1}\ ]_D)\ ) = [\ v_0\ ,\ldots,\ v_{n-1}\ ]_{R_n} \tag{6.30}$$

$$\downarrow(\ \mathsf{getPartition}([\ v_0\ ,\ldots,\ v_{n-1}\ ]_D)\ ) = \left[\ [v_0,\ldots,v_{D(0)-1}],\ldots,[v_{j_i},\ldots,v_{j_i+D(i)-1}],\ldots,[v_{j_{p-1}},\ldots,v_{n-1}]\ \right]_{E_p}$$

$$\text{where} \quad j_i = \sum_{k=0}^{k=i-1}(D(k)) \tag{6.31}$$

$$\downarrow(\ \mathsf{flatten}([\ a_0\ ,\ldots,\ a_{n-1}\ ]_D)\ ) = \left[\ a_0[0],\ldots,a_0[n_0-1],a_1[0],\ldots,a_{n-1}[n_{n-1}-1]\ \right]_{D'}$$

$$\text{where} \quad D'(i) = \sum_{D(i-1)\leq k<D(i)} |a_k| \tag{6.32}$$

Figure 6.2 – *OSL Formal Programming Model*

This allows to instantiate the functor with a module containing a specific value for lastProcessor in order to write examples and execute our semantics within Coq.

The type of distributed array is a record type:

```
Record distributedArray (A:Type) := mkDistributedArray {
  distributedArray_data : list A;
  distributedArray_distribution: vector nat bsp_p;
  distributedArray_invariant:
    List.length distributedArray_data = sum distributedArray_distribution
}.
```

This type contains the two fields already described: the content of the parallel vector (distributedArray_data), and the distribution of this content on the processors (distributedArray_distribution).

However there is a third field: a proof that the two fields form indeed a coherent representation of a distributed array. The sum of the elements of the distribution (computed using the function sum, omitted here) should be the length of the content list.

Values of this type are a kind of inner representation of distributed arrays that the user of the Orleans Skeleton Library could not used directly. She will be given a syntax for writing OSL programs.

6.2.2 Syntax and Typing

The language of the Coq proof assistant can be seen as a pure functional programming language plus the ways to express the logical properties. Therefore the sequential values and the functions of the host language (here C++) can be written as Coq values and functions. In the case of functions we thus model only their input/output behaviour. The typing and evaluation of Coq values and functions respect the assumption of section 6.1, but for type vector$\langle\tau\rangle$ replaced by list τ. Of course the user of this formalisation should also choose the Coq type and values that corresponds the best to its C++ counterparts when she models a program. For example one could choose to represent values of type int in C++ by values of type Z or nat in Coq.

The result of a computation, a value, could be either a usual sequential value, for example the result of the application of the reduce skeleton, or a distributed array, for example the result of the application of the map skeleton.

There are several ways of formalising the syntax of OSL programs. We shall illustrate this by two short examples dealing only with the construct for distributed arrays and the map skeleton. The first solution follows:

```
Inductive expression :=
| DistributedArray: ∀A:Type, list A →expression
| Map : ∀A B, (A→B) →expression →expression
```

To simplify the example, a distributed array is just modelled as a list of values. All values being typed in Coq, the constructor for this case of the inductive type expression should also take as argument the type of the elements of the list. For the Map constructor, the first argument is the "muscle" argument, the function f to be applied to each element of the distributed array, the second expression. Here again the input and output types of the function should be given.

This grammar however models possibly ill-typed expressions of our language of skeletons. It is possible to define the following Coq term:

```
Definition e : expression :=
  Map string string (append "!") (DistributedArray nat [1;2;3]).
```

In Coq it is possible to indicate that some arguments may be implicit: it is the case here for the types arguments of the two constructors Map and DistributedArray and we could write:

```
Definition e : expression := Map (append "!") (DistributedArray [1;2;3]).
```

The expression e is well typed for Coq but it *represents* an ill-typed expression of our skeleton language as the muscle function append operates on strings instead of natural numbers. We could in Coq closely follow the syntax, type system and operational semantics of section 6.1 and prove that the operational semantics we will define follows the subject reduction property (i.e. it preserves the typing).

However there is another solution: we could model the grammar in such a way that only well-typed (in the skeleton language point of view) expressions could be modelled in Coq:

```
Inductive typedExpression (A:Type) :=
| TDistributedArray : list A →typedExpression A
| TMap: ∀B, (B→A) →typedExpression B →typedExpression A.
```

Here the grammar is typed. An expression of type typedExpression A represents an expression whose value is a distributed array whose elements have type A. The expression e could not be defined in Coq as a typedExpression: the input type of the muscle function in the Map constructor should be the type of the elements of the second argument of Map.

Therefore by defining the operational semantics by a function or a relation that relates only expressions that represent skeleton expressions of the same type, then we have the subject reduction for free.

The syntax of OSL is actually a bit more complicated as we distinguish between the expressions whose values have a sequential type and the expressions whose values have parallel types, these two kinds of expressions being mutually recursive. The whole syntax is in figure 6.3. In order to be able to apply a "sequential" program to the result of the evaluation of a skeleton expression, we provide a SeqApply constructs. The SeqValue constructors is simply used to provide "muscles" to the skeletons.

```
Inductive seqExpr : Type →Type :=
| SeqValue: ∀A, A →seqExpr A
| Reduce: ∀A, seqExpr (A→A→A) →seqExpr A →parExpr A →seqExpr A
| SeqApply: ∀A B, seqExpr (A→B) →seqExpr A →seqExpr B
with parExpr : Type →Type :=
| ParValue: ∀A, distributedArray A →parExpr A
| Replicate: ∀A, seqExpr A →seqExpr nat →parExpr A
| Init: ∀A, seqExpr (nat→A) →seqExpr nat →parExpr A
| CreateAtRoot: ∀A, seqExpr (list A) →parExpr A
| Map: ∀A B, seqExpr (A→B) →parExpr A →parExpr B
| Zip: ∀A B C, seqExpr (A→B→C) →parExpr A →parExpr B →parExpr C
| MapIndex: ∀A B, seqExpr (nat→A→B) →parExpr A →parExpr B
| ZipIndex: ∀A B C, seqExpr (nat→A→B→C) →parExpr A →parExpr B →parExpr C
| Shift: ∀A, seqExpr Z →seqExpr(nat→A) →parExpr A →parExpr A
| GetPartition: ∀A, parExpr A →parExpr(list A)
| Flatten: ∀A, parExpr(list A) →parExpr A
| Permute: ∀A, seqExpr (nat→nat) →parExpr A →parExpr A
| Balance: ∀A, parExpr A →parExpr A
| Gather: ∀A, parExpr A →parExpr A
| Bcast: ∀A, parExpr A →parExpr A.

Inductive expr : Type →Type :=
| Seq: ∀A, seqExpr A →expr A
| Par: ∀A, parExpr A →expr (distributedArray A).
```

Figure 6.3 – *OSL Syntax in Coq*

The three first Coq constructors of the parExpr type are the usual OSL C++ class constructors: we can build a distributed array by specifying its size and a value that will be replicated everywhere (Replicate), or the content of the distributed array could be specified by a function from array indices to values (Init). In these two cases, the data is distributed evenly on the processors. The third constructor is used to build a distributed array containing values only at the root processor (CreateAtRoot). The other Coq constructors model the skeletons and their typing presented in sections 6.1.1 and 6.1.2.

6.2.3 Big-Step Semantics

For the formalisation of the big-step semantics of OSL, we define three functions, the two first being mutually recursive:

- seqEvaluation: ∀A : **Type**, seqExpr A →result A
- parEvaluation: ∀A : **Type**, parExpr A →result (distributedArray A)

- evaluation: ∀A : **Type**, expr A →result A

In addition to the semantics of section 6.1, we add the possibility for a program to raise an error. In the previous section if there are problems the evaluation is just not defined. Here it will be defined and will return a special kind of value: an error message. To do so we use the technique of monadic programming [138].

The result type is used in a monadic style in order to model possible errors during evaluation, without being too cumbersome to use compared to a solution with optional values and pattern-matching. As in [93] for example, we use a convenient Coq feature that allows to define notations:

```
Inductive result (A: Type) : Type :=
| Ok: A →result A
| Error: string →result A.

Definition bind (A B: Type) (r: result A) (g: A →result B) : result B :=
  match r with
  | Ok x ⇒g x
  | Error msg ⇒Error msg
  end.

Notation "'do' X <− A ; B" := (bind A (fun X ⇒B)).
```

The bind function is used to first evaluate a result (it r argument): if it is not an error then this value is passed as argument to the second argument that could used it to produce a new result. If r is an error then this error is return without evaluating the body of g.

With the do notation, the big-step semantics functions are quite readable. For example the case for the evaluation of the reduce skeleton in the seqEvaluation function is written as follows:

```
| Reduce A op neutral pe ⇒
    do op <− seqEvaluation op;
      do neutral <− seqEvaluation neutral;
        do da <− parEvaluation pe ;
          Ok(List.fold_right op neutral (distributedArray_data da))
```

We first evaluate the "muscles" of the skeletons. If one of these calls raises an error, then the function immediately returns this error, otherwise it binds the obtained value with the variable before the <− arrow and continues to evaluate the expression after the ;.

The parEvaluation function produces values of type distributedArray. In order to keep this function short, we defined auxiliary functions that transforms distributed arrays. The parEvaluation function thus first recursively calls itself and seqEvaluation on the arguments of the expression it evaluates, and obtains *values*, in particular in

the parallel case, values of type distributedArray. Then it calls the appropriate auxiliary function. For example:

```
| Replicate _ se se' ⇒
    do v <− seqEvaluation se;
      do size <− seqEvaluation se';
        Ok (replicate v size)
```

The replicate function, and all the auxiliary functions, are defined using the **Program** feature of Coq:

```
Program Definition replicate(A:Type)(value:A)(size:nat) : distributedArray A :=
  mkDistributedArray
  (List.map (fun index⇒value) (List.seq 0 size))
  (evenDistribution size)
  _.
Next Obligation.
  autorewrite with length; rewrite sumEvenDistribution; trivial.
Defined.
```

For building a value of type distributedArray, we need three components:

- the content of the distributed array, in this case it is defined on the third line (we apply a constant function to all the elements of a list of natural numbers, of the specified size),
- the distribution, in this case it is defined on the fourth line, by a call to the function evenDistribution,
- a *proof* that the content and the distribution are coherent.

The two first components are written very similarly to functional programs. For the proof however, it is easier to use the interactive proof mode. Thus we do not give this third component: we use the wild-card _ instead. Coq then generates proof obligations that should be proved in order for the value replicate to be defined. The proof is here quite simple because most of the work is done in the lemma sumEvenDistribution that it itself proved using several other lemmas.

This replicate function could not directly raise an error. Few skeletons can: the zip skeleton if the two parallel arguments do not have the same distribution, the permute skeleton if the function in argument is not bijective, and the flatten skeleton if the distribution of its argument is not one element (of type list) per processor.

By construction the type of the expressions are preserved during evaluation: we have subject reduction for free.

evaluation, seqEvaluation and parEvaluation are functions. They can be applied to OSL program examples in Coq. The results of such evaluations can be output. This allows to design and implement automatic tests to check if the formal semantics and the implementation are coherent. This can serve to debug both: the formal semantics

may be erroneous because we were wrong in the modelling, or the implementation may contain bugs.

6.3 Verification of a Heat Diffusion Simulation

We remind (see section 3.2.3) that the simulation of the one dimensional heat diffusion could be performed by solving the following discretised equation:

$$u(x,t+1) = \text{diffuse} \times \frac{\Delta_t}{\Delta_x^2} \times \big(u(x+1,t) + u(x-1,t) - 2 \times u(x,t) \big) + u(x,t)$$

The OSL program for one step of update is thus (see section 4.2.1 for a discussion about the following code):

```
bar = zip(std::plus<double>(),
          bar,
          map(boost::bind(std::multiplies<double>(), (kappa∗dt)/(dx∗dx), _1),
              zip(std::plus<double)(),
                  zip(std::plus<double>(),
                      shift(1, leftBound, bar),
                      shift(−1, rightBound, bar)),
                  map(boost::bind(std::multiplies<double>(), −2, _1), bar)))
```

where:

- std::plus<double>() is the addition on double precision floating point numbers of the C++ standard library,
- boost::bind(std::multiplies<double>(), (kappa∗dt)/(dx∗dx), _1) is a function obtained as partial application of the multiplication to the expression $\frac{\kappa dt}{dx^2}$,
- leftBound and rightBound are the constant replacement functions that return respectively l and r,
- and boost::bind(std::multiplies<double>(), −2, _1) is the function that multiplies by 2.

For proving the correctness of the OSL heat diffusion simulation, we use the Coq proof assistant and the formalisation of OSL programming model of the previous section. We shall not present Coq source code, but we shall sketch the formalisation and the proof in an informal manner.

As we do not rely on the properties of the arithmetic operators, we only assume that we have a type number of numbers with usually operations +, −, ∗, /. When we write (op) for an operator, we consider it has a binary function that can be partially applied.

```
zip( ( + ),
     bar,
     map ( ( ∗ ) ( / ) ( ( ∗ ) kappa dt ) ( ( ∗ ) dx dx ),
         zip ( ( − ),
             zip ( ( + ), shift(1, leftBound, bar), shift(−1, rightBound, bar) ),
             zip ( ( + ), bar, bar ) ) ) )
```

Figure 6.4 – *OSL Heat Diffusion Simulation in Coq*

The formalisation of the OSL heat diffusion program, is therefore the expression shown in figure 6.4, provided bar is a distributed array of numbers, kappa, dt and dx are numbers, and leftBound and rightBound are constant functions from array index to numbers.

The first step for the proof of correctness in the Coq proof assistant is to write the specification of the function that computes a step of the heat diffusion simulation, as indicated by equation (3.2) but with formally taking into account the boundary conditions. We model the array of temperatures by a list of numbers. For this list structure we have a function nth that takes as input an index i, a list, and a default value d: it returns the value in the list at index i, if the index is valid, and it returns the default value otherwise. It is a kind of array access but with the default value in case the index is not valid. The specification could then be written, for non-empty lists *bar* and valid indexes i (these conditions are omitted here but not in the formal development):

$$
\begin{aligned}
&\forall \kappa\ dx\ dt\ l\ r\ bar\ i\ d,\\
&\quad \mathsf{nth}\ i\ (\textbf{step}\ \kappa\ dx\ dt\ l\ r\ bar)\ d =\\
&\quad \kappa \times dt/(dx \times dx) * \big(\\
&\qquad (\mathsf{nth}\ (1+i)\ bar\ r) +\\
&\qquad (\textbf{if}\ i = 0\ \textbf{then}\ l\ \textbf{else}\ \mathsf{nth}\ (i-1)\ bar\ d) -\\
&\qquad ((\mathsf{nth}\ i\ bar\ d) + (\mathsf{nth}\ i\ bar\ d))\big) +\\
&\quad (\mathsf{nth}\ i\ bar\ d).
\end{aligned}
$$

The function step takes as argument the numbers κ, dt, dx, l and r as well as a list of numbers *bar* and returns an updated list of numbers. For the parallel version of the heat diffusion simulation, we can use the expression of figure 6.4 and the evaluation function that models the OSL programming model as a big-step semantics. However what we can obtain directly from these two components is a function that takes as arguments the same numbers but a *distributed array* instead of a list and a returns either a distributed array or *an error message*.

The first problem can be easily solved: we compose the obtained function with a distributedArrayOfList function that takes a list and returns an evenly distributed array.

The second problem needs a *proof* that the evaluation of the expression of figure 6.4 will not raise an error. As a matter of fact, there is only one skeleton in this expression that can lead to an error message: the zip skeleton in case its two distributed array

arguments do not have the same distribution. We have a short lemma that states that an evaluation of zip does not raise an error message when the two distributed arrays have the same distribution, and this results is used together with the fact that the other skeleton preserve distribution to prove that the evaluation of the OSL heat diffusion program does not raise an error. From this proof we can build a function that evaluates the expression of figure 6.4 and returns a distributed array.

The main theorem thus states that this last function follows the specification defined above. The proof of this theorem proceeds as follows:

- the skeletons preserve the distribution, so the guards of the application of zip can be removed,
- we then obtain for the content part of the result an expression similar to figure 6.4 but on the lists instead of the distributed arrays: using results on the commutation of nth with other functions on lists such as map, this expression can be simplified in such a way that the call to nth on the left hand side of the equation is not on top, but rather copied inside the expression,
- finally with reason by cases on i: i is 0, i is the length of *bar* minus 1, or i is in the middle. For each case the assumptions are enough to simplify both side of the equation to the same expression.

All the Coq source code of this formalisation is available at:

http://traclifo.univ-orleans.fr/OSL.

6.4 Summary

Formal semantics of the programming model is an unambiguous reference for the users of libraries or programming languages. Very often the formal semantics are given only as text. In the case of OSL, both a hand-written formal semantics and a mechanised one, using the Coq proof assistant, are provided.

Mechanising such a semantics offers the following benefits:

- The semantics is executable, thus it is possible to run a program using the implementation of OSL and its formalisation, to test their conformance. This helps to improve confidence in both the formalisation and the implementation.
- During the proofs, and even during the formalisation, the Coq proof assistant helps to avoid forgetting subtle details.
- As the semantics of the host sequential language of OSL is considered as a black box, it is very convenient to use the Coq functional language to model sequential parts of OSL programs.

Conclusions and Perspectives

7

Contents

A.1 C++ Expression Templates 107
A.2 Template Metaprogramming 110
A.3 Move Semantics and Rvalue References 111

Summary

The initial motivation of the work is to provide a platform to the sequential programmers for writing the parallel programs. The goal was to provide a infrastructure satisfying the core limitations of the other parallel programming models, like programmability, portability, performance prediction and performance portability, and the formal proof of correctness. The outcome of the research is OSL, a library of data parallel skeletons developed in C++ based over the BSP model and using MPI as the communication library. OSL provides a way to sequential programmers for developing their parallel programs just by choosing a limited set of skeletons. The skeletons in OSL can be composed to further optimise the code. OSL can express parallelism on non evenly distributed arrays through its unique skeletons getPartition and flatten. Unbalanced distributed arrays could be made evenly distributed through a balance skeleton. As the OSL is based over the BSP model, it is capable of accurately predict the performance. It also provides a mechanism based on the BSP cost formula to port the performance. The formal semantics of the skeletons in OSL are captured in Coq and used for the proving the correctness of the programs developed with OSL.

Contribution of OSL in the Skeleton World

OSL being a skeleton library has contributed in several ways to the skeleton world.

- Improved programmability and improved interfacing with the existing standards. As the library is implemented using the latest C++0x standard, it offers unique programmable features in comparison to the other skeleton frameworks.

- OSL is the first library based over the pure BSP model. It does not use any extension of the BSP.
- Performance prediction and performance portability feature is not the main concern in most of the skeleton frameworks. OSL shares its interest of these aspects with Skel-BSP.
- As the skeletons are the constraining ways of expressing parallelism, it is important to offer a good trade-off between constraints and expressivity. OSL adds two new skeletons to the arsenal of skeletons for changing the orientation of the data. The change in orientation offers other views of the data and helps in expressing irregular problems.
- OSL comes with a library for a proof assistant that provides a mechanism to formally prove the correctness of the programs developed with OSL.

Limitations

OSL has certain limitations as well.

- The nesting of the skeletons is not supported. The reason for not supporting the nesting is that, as the skeletons in OSL are based on BSP model, some of the nesting yields combinations that do not follow the pure BSP model.
- There are skeletons that can benefit from the performance portability but the feature is not still provided.
- The library can be made more efficient by adding the transformational optimisations.

Future Work

We believe that the skeleton parallelism has the potential to attract the masses of the sequential programmers towards parallel programming. The research in OSL will continue in several directions to make it a easy to use and performing framework.

Several real world applications will be developed with OSL. The parallelization of a shallow water flow application of the Mapmo Lab of Université d'Orléans is the first example in this regard.

Currently OSL is targeting only the cluster environment, the development of hybrid versions of OSL will start soon. The hybrid versions will be the OpenMP/MPI version and the one with gpu programming.

The data structure at which OSL skeletons are operating is limited to the distributed arrays. The other data structures specially the addition of the irregular data structures will be an important aspect of the upcoming work on OSL.

We believe that by adding full support of performance portability in OSL, we can make it a framework capable of performing well on various architectures. There are skeletons in OSL that can be equipped with this ability and some new skeletons can be added as well. So, the performance portability will remain an important area of research and development regarding OSL.

The PaPDAS[1] project that is already started in 2011 is by some aspects and extension of the OSL project. In the PaPDAS project we are interested in providing a framework to ease the development of parallel programs in a systematic way using constructive algorithms, and to either execute very efficiently the obtained programs or to compile these programs with a verified optimising parallel compiler. The methodology of the PaPDAS project is to rely on the structured model of skeletal parallelism.

A theory of program calculation could be designed in order to provide a sound basis for a methodology of systematic development of correct parallel programs, as well as supporting tools. Restricting the parallelism is also a mean to reduce the semantic complexity of parallel programs. This makes possible the development of a verified parallel compiler of an extension of C: Algorithmic Skeleton C. The ASC language will be inspired by the OSL and SkeTo libraries.

The PaPDAS project also plans new versions of SkeTo and OSL, in particular we shall implement hybrid OpenMP/MPI versions of our skeletons. I also aim at improving the OSL framework where every skeleton and the combination of the skeletons is equipped with the performance portability. The balance, scan and bcast skeletons can be equipped with the property. I shall also work on applications developed with OSL, including a numerical simulation of water movement on land with colleagues for the university of Orléans and INRA.

[1] http://traclifo.univ-orleans.fr/PaPDAS

Appendix

Advanced C++ Programming Techniques

A

The section details some of the advanced C++ programming techniques applied in the development of OSL.

- C++ Expression Templates:
- C++ Template Metaprogramming
- Move Semantics and Rvalue References

A.1 C++ Expression Templates

Traditional operator overloading for solving the mathematical expression suffers from the following problems:

- Production of the temporaries.
- Multiple passes of the loop.
- Extra trips to the memory due to repetitive stores and loads.
- Cache thrashing

The answer to these problems is the Expression Templates technique introduced by Todd Veldhuizen [136]. The technique is based on the templates and the operator overloading. The expressions are encoded in the form of the templates. Operator overloading is used to build the template object of the expression at compile time. This template representation of the expression corresponds to the prefix order of the expression. The evaluation of the expression is delayed until it is assigned to a container (vector/array/matrix). The assignment operator in the container class evaluates the expression by iterating through the index space. During each step the result of the right hand side expression is assigned to the left hand side. In this manner the whole expression evaluation is accomplished in a single loop. The resulting code is nearly

efficient as its C's counterpart. The temporaries are avoided, expression is executed in a single pass and the operands are fetched from the memory only once.

The technique is explained by using an expression template array class. The but is to construct the simple array arithmetic expressions. As described earlier the container class should be provided with an overloaded assignment operator, for the lazy evaluation of the expression. To represent the right hand expressions with a single interface the need of a wrapper class is emerged. Specifically, `Expr<T>` is the wrapper class and `T` is the original wrapped expression.

```
template <class A>
 struct Expr {
   operator const A&() const {
      std::cout≪ ''I am here !''≪ std::endl;
     return *static_cast<const A*>(this);
    }
 };
```

The cast operator in the `Expr` class delegates control to the derived class. The container class `Array<T>` now can be inherited from the `Expr` class, providing its class type as the template parameter of the `Expr` class i.e `Expr<Array<T> >`. This design pattern is called "Curiously recursive template pattern" [52].

```
template<class T>
class Array : public Expr<Array<T> > {
public:
  template <class A>
  void operator = (const Expr<A>& a_) {
    const A& a(a_);
    for (int i = 0; i < n; ++i)
      data[i] = a[i];
  }

  T operator[] (int i) const {
     return data[i];
  }

// other public or private members

};
```

Now, it is clear from the above code, why the wrapper class is needed. It differentiates the expression templates expressions from all the other occurring at the right hand side of the assignment operator.

What else should be wrapped inside the `Expr` wrapper? All the operators representing the non leaf nodes of the expression trees are the valid expressions. The

operators can be be encoded in types using the templates, and wrapped inside the `Expr` as follows

```
template <class A, class B>
 class Add : public Expr<Add<A,B> > {
    const A& a_;
    const B& b_;
  public:
    Add(const A& a, const B& b) : a_(a),b_(b){}
    double operator[](int i) const {
      return a_[i] + b_[i];
     }
 };
```

Some important things to be noted here

- The operator class stores the references of the operands since the life time of the expression objects lasts until the evaluation is completed.
- The operator class overloads the access **`operator`**`[]` for the evaluation of the *ith* index by evaluating the corresponding expression objects.
- The inheritance of the operator class from the `Expr` using CRTP allows efficient inheritance achieved at compile time and results in full inlining optimisation.

Operator overloading is used to represent expression objects.

```
template <class A, class B>
 inline Add<A,B>
 operator+(const Expr<A>& a, const Expr<B>& b){
    return Add<A,B>(a,b);
  }
```

The drawbacks or limitations associated with this approach are

- Increased compilation time due to nested template types and inlining optimisations.
- Long and non-understandable error messages.
- Not every programmer is familiar with the templates (limited audience).

Because of the above mentioned drawbacks it is suggested [137] not to build the expression templates libraries from scratch. Some expression templates building frameworks like PETE (Parallel Expression Templates Engine) and Proto are present and help in writing easy, time saving and efficient code. A number of mathematical and scientific libraries are developed using this. Some critiques on the traditional expression template techniques are presented [74] and improvements are suggested.

A.2 Template Metaprogramming

Metaprogramming refers to the programs that represent or manipulate other programs. Classic examples are the interpreters, program generators and the compilers. In the context of C++, the term refers to the mechanism based on the C++ template system to perform computations at the compile time. In C++ it is not a planned and designed feature, infact it is discovered as an accident and is first demonstrated by the Erwin Unruh by his prime computation program.

```
#include<iostream>

template <int p, int i>
class is_prime {
public:
 enum { prim = (p==2) || (p\%i) && is_prime<(i>2?p:0),i-1>::prim };
};

template<>
class is_prime<0,0> {
public:
 enum {prim=1};
};

template<>
class is_prime<0,1> {
public:
      enum {prim=1};
};

template <int i>
class Prime_print {
// primary template for loop to print prime numbers
public:
 Prime_print<i-1> a;
 enum { prim = is_prime<i,i-1>::prim };
 void f() {
  if (prim)
    std::cout << "prime number:" << i << std::endl;
  a.f();
 }
};

template<>
class Prime_print<1> { // full specialisation to end the loop
public:
 enum {prim=0};
 void f() {};
};
```

```
#ifndef LAST
#define LAST 18
#endif
int main() {
  Prime_print<LAST> a;
  a.f();
}
```

Although the program presented above is a metaprogram, but it is a value computational metaprogram. But mostly the usage and main power of the metaprogramming is dedicated to the type based programming.

```
template<typename X, typename Y>
struct SameType
{
  enum { result = 0 };
};

template<typename T>
struct SameType<T, T>
{
  enum { result = 1 };
};
```

The program above outputs 1 if the two types supplied to the `SameType` as parameters are same. A lot of the basics like conditional constructs, looping constructs, asserts, object generators, compile time containers and the relevant algorithms etc,that can be used to build more sophisticated metaprograms are packaged in the form of Boost metaprogramming library [3]. There are certain limitation of template metaprogramming as well like no error reporting mechanism, increase compilation times, almost unreadable source code and hence difficult to debug these programs.

A.3 Move Semantics and Rvalue References

Copying is one of the most expensive operation. It becomes a performance bottleneck in the efficiency of the software if there are unwanted copy operations. There are situations in which the unnecessary copy operations can be avoided. Consider the code presented below:

```
std::vector<T> vec;
vec = create();
```

`create()` is a factory function creating an instance of the `std::vector<T>`. The created vector object is copied to the variable `vec` as return value before destroying the local copy. This copy operation is useless as the locally allocated memory for the object is never used. The operation can become much efficient if the `vec` variable starts pointing the local memory of the object created in `create()` method. This idea is called **Moving** and is supported in C++ox with the introduction of the Rvalue references and the move semantics.

Contrary to the **lvalues** like named objects **rvalues** are the ones for which the address can not be taken of. The examples of the rvalues are the unnamed temporary objects. `vec` is the lvalue reference and the return temporary object of the `create()` is the rvalue reference. Moving from lvalues is not safe because the original data in the lvalue is lost. But moving from the rvalues is safe as they are the temporaries and no data will be lost. Thus rvalues are used for the identification of the target of the move operation.

In C++ox the move semantics can be implemented by implementing the move constructor and move assignment operator.

```
class MemoryBlock
{
public:
 // Copy constructor.
 MemoryBlock(const MemoryBlock& other)
   : _length(other._length)
   , _data(new int[other._length])
 {
   std::cout << ``In MemoryBlock(const MemoryBlock&). length = ''
           << other._length << ``. Copying resource.'' << std::endl;
   std::copy(other._data, other._data + _length, _data);
 }

 // Copy assignment operator.
 MemoryBlock& operator=(const MemoryBlock& other)
 {
   std::cout << ``In operator=(const MemoryBlock&). length = ''
           << other._length << ``. Copying resource.'' << std::endl;
   if (this != &other)
   {
     // Free the existing resource.
     delete[] _data;
     _length = other._length;
     _data = new int[_length];
     std::copy(other._data, other._data + _length, _data);
   }
   return *this;
 }

private:
```

```
 size_t _length; // The length of the resource.
 int* _data; // The resource.
};
```

The code presented above is taken from the msdn Microsoft's web page [1]. The code contains a traditional copy constructor and a copy assignment operator.

```
// Move constructor.
 MemoryBlock(MemoryBlock&& other)
   : _data(NULL)
   , _length(0)
 {
   std::cout << ''In MemoryBlock(MemoryBlock&&). length = ''
           << other._length << ''. Moving resource.'' << std::endl;
   // Copy the data pointer and its length from the
   // source object.
   _data = other._data;
   _length = other._length;
   // Release the data pointer from the source object so that
   // the destructor does not free the memory multiple times.
   other._data = NULL;
   other._length = 0;
}
// Move assignment operator.
MemoryBlock& operator=(MemoryBlock&& other)
{
   std::cout << ''In operator=(MemoryBlock&&). length = ''
           << other._length << ''.'' << std::endl;
   if (this != &other)
   {
     // Free the existing resource.
     delete[] _data;
     // Copy the data pointer and its length from the
     // source object.
     _data = other._data;
     _length = other._length;
     // Release the data pointer from the source object so that
     // the destructor does not free the memory multiple times.
     other._data = NULL;
     other._length = 0;
   }
   return *this;
}
```

The above presented move constructor and the move assignment operator equip the `MemoryBlock` class with the move semantics. The `&&` operator represents the rvalue reference. `std` is provided with a `move` function to properly handle the move operation. The modified constructor is presented below

```
MemoryBlock(MemoryBlock&& other)
  : _data(NULL)
  , _length(0)
{
  *this = std::move(other);
}
```

The other details about the move semantics and the rvalue references can be found in the articles [123, 2].

A Short Introduction to The Coq Proof Assistant

B

Contents

C.1 Mirev 119
C.2 Speed 119

The Coq proof assistant [129, 27, 26] is based on the calculus of inductive constructions. This calculus is a higher-order typed λ-calculus. The Curry-Howard correspondence is at the core of Coq: Theorems are types and their proofs are terms of the calculus. The Coq system provides a language of tactics to help the user to build proof terms.

We illustrate quickly all these notions on a short example. We first define a new inductive type, the type of natural numbers in the Peano style:

```
Inductive nat : Set :=
  | O : nat
  | S : nat →nat.
```

`nat` has type `Set`: it is similar to a usual data-type in a functional language.

We also define the `plus` recursive function on natural numbers:

```
Fixpoint plus (n1 n2:nat) {struct n1} : nat :=
match n1 with
  | O ⇒n2
  | S n ⇒S(plus n n2)
end.
```

In this recursive definition we should specify which argument is structurally decreasing (`n1` in the example). This is because all functions must be terminating in Coq. In both definitions, we gave the type of the new name we wanted to define as well as a term of this type. We then define a lemma:

```
Lemma plus_n_O : ∀n, plus n O = n.
Proof.
  induction n.
```

```
(* case n=0 *) simpl. reflexivity.
(* case n>0 *) simpl. rewrite IHn. reflexivity.
Qed.
```

If we check (using the `Check` command of Coq) the type of expression, we would obtain `Prop`. This definition is a proposition. It belongs to the logical realm. To define `plus_n_O` we should not only provide a type, but also a term of this type: a proof of the lemma. We could write directly such a term, but it is usually complicated. Coq provides a language of tactics to help the user to build proof terms. In the top-level of Coq, entering line beginning with `Lemma` activates the interactive proof mode. The Coq proof assistant indicates that we should prove the following goal:

```
============================
 forall n : nat, plus n O = n
```

We prove this goal by induction on `n` using the tactic `induction n`. The system indicates now two goals to prove:

```
============================
 plus O O = O

subgoal 2 is:
 plus (S n) O = S n
```

The first one is proved using the definition of `plus` using the tactic `simpl` which yields the goal `0 = 0` and this case is ended by the application of the tactic `reflexivity`. The second one is the inductive case:

```
n : nat
IHn : plus n O = n
============================
 plus (S n) O = S n
```

After simplification, we obtain the goal `S(plus n O) = S n`. We solve it first by rewriting `plus n O` in `n` using the `IHn` hypothesis and then we conclude by reflexivity. Actually, Coq has some automation. The `plus_n_O` lemma could be proved using one tactic: **`auto`**.

Mixing logical and computational parts is possible in Coq. For example a function of type `A→ B` with a precondition `P` and a post-condition `Q` corresponds to a constructive proof of type: `forall x:A, (P x) → exists y:B → (Q x y)`. This could be expressed in Coq using the inductive type `sig`:

```
Inductive sig (A:Set) (P:A→Prop) : Set := | exist: ∀(x:A), (P x) →(sig A P).
```

It could also be written, using syntactic sugar, as `{x:A|(P x)}`.

This feature is used in definition of the function `pred`:

```
Require Import Program.

Program Definition pred (n:nat | n<>O) : {q:nat|(S q)=n} :=
```

```
match n with
  | O ⇒_
  | S n ⇒n
end.
```

The specification of this function is: `forall n : {m : nat | m<>O}, {q : nat | S q = 'n}` where `'n` represents the natural number part of `n` (the other part being a proof that this natural number is not zero). We define `pred` using the `Program` feature of Coq. This feature allows the user to write a function with post-conditions as if there were no post-condition. `Program` generates proof obligations to be proved to ensure that the function result indeed meets the post-condition. Moreover in this example the proof obligations are proved automatically by the system.

Machines used for testing OSL

C

C.1 Mirev

Number of nodes	8
Processors per node	2
Cores per processor	4
RAM	16Gb
Total number of cores	64
Processor type	AMD Opteron QuadCore 2376 2.3 GHz
Network	Giga Ethernet (Cooper)
MPI Version	Open MPI 1.4.2

C.2 Speed

Number of nodes	1
Processor per node	4
Cores per processor	12
Mémoire vive	64Gb
Total number of cores	48
Pprocessor type	AMD Opteron Processor 6174 2.2 GHz
Network	-
MPI Version	MPICH 2

Bibliography

[1] How to: Write a move constructor. http://msdn.microsoft.com/en-us/library/dd293665.aspx, 2010. Cited page 113.

[2] D. Abraham. Series of articles on move semantics. http://cpp-next.com/archive/2009/08/want-speed-pass-by-value/, 2009. Cited page 114.

[3] D. Abrahams and A. G. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. Cited page 111.

[4] S. Adachi, H. Iwasaki, and Z. Hu. Diff: A Powerfull Parallel Skeleton. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, volume 4, pages 425–527. CSREA Press, 2000. Cited page 21.

[5] F. Ahmad and Y. Xin. Automatic generation and tuning of mpi collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 393–402, New York, NY, USA, 2005. ACM. Cited page 24.

[6] M. Aldinucci. Eskimo: Experimenting with Skeletons in the Shared Address Model . *Parallel Processing Letters*, 13(3):449–460, 2003. Cited page 17.

[7] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In *9th Intl Euro-Par 2003: Parallel and Distributed Computing*, volume 2790, pages 712–721, 2003. Cited page 18.

[8] M. Aldinucci, M. Coppola, M. Danelutto, N. Tonellotto, M. Vanneschi, and C. Zoccolo. High level grid programming with ASSIST. *Computational Methods in Science and Technology*, 12(1):21–32, 2006. Cited page 18.

[9] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proc. of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS99)*, pages 955–962, 1999. Cited page 18.

[10] M. Aldinucci and M. Danelutto. Skeleton-based parallel programming: Functional and parallel semantics in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, 2007. Cited pages 17 et 25.

[11] M. Aldinucci, M. Danelutto, and P. Dazzi. Muskel: An expandable skeleton environment. *Scientific International Journal for Parallel and Distributed Computing*, 8:325–341, 2007. Cited page 18.

[12] M. Aldinucci, M. Danelutto, and P. Teti. An Advanced Environment Supporting Structured Parallel Programming in Java. *Future Generation Computer Systems*, 19:611–626, 2002. Cited pages 17 et 25.

[13] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 95–105. ACM, 1995. Cited page 7.

[14] J. Alglave, A. C. J. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of power and arm multiprocessor machine code. In L. Petersen and M. M. T. Chakravarty, editors, *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*, pages 13–24. ACM, 2009. Cited page 7.

[15] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. Stapl: an adaptive, generic parallel c++ library. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 193–208, Berlin, Heidelberg, 2003. Springer-Verlag. Cited page 23.

[16] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer, 2nd edition edition, 1997. Cited page 7.

[17] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20:404–418, 2009. Cited page 3.

[18] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: a structured high-level parallel language, and its structure support. *Concurrency: Practice and Experiences*, 7(3):225–255, May 1995. Cited page 15.

[19] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. Skie: a heterogeneous environment for hpc applications. *Parallel Computing*, 25:1827–1852, 1999. Cited page 16.

[20] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, Deploying, Composing, for the Grid. In J. Cunha and O. F. Rana, editors, *Grid Computing: Software Environments and Tools*. Springer, 2006. Cited page 4.

[21] A. Basumallik, S.-J. Min, and R. Eigenmann. Programming distributed memory sytems using openmp. In *IPDPS*, pages 1–8, 2007. Cited page 4.

[22] A. Benoit and M. Cole. Two fundamental concepts in skeletal parallel programming. In *The International Conference on Computational Science (ICCS 2005) , Part II, LNCS 3515*, pages 764–771. Springer Verlag, 2005. Cited page 21.

[23] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In J. C. Cunha and P. D. Medeiros, editors, *11th International Euro-Par Conference*, LNCS 3648, pages 761–770. Springer, 2005. Cited page 21.

[24] J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Parallel fft with eden skeletons. In *Proceedings of the 10th International Conference on Parallel Computing Technologies*, PaCT '09, pages 73–83, Berlin, Heidelberg, 2009. Springer-Verlag. Cited page 22.

[25] J. Berthold, M. Dieterle, and R. Loogen. Implementing parallel google map-reduce in eden. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 990–1002, Berlin, Heidelberg, 2009. Springer-Verlag. Cited page 22.

[26] Y. Bertot. Coq in a hurry, 2006. http://hal.inria.fr/inria-00001173. Cited page 115.

[27] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004. Cited pages 35 et 115.

[28] G. Bilardi, A. Pietracaprina, G. Pucci, K. T. Herley, and P. Spirakis. Bsp versus logp. *Algorithmica*, 24:405–422, 1999. Cited page 7.

[29] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, number 55 in NATO ASI, Marktoberdorf, BRD, 1989. Springer. Cited page 25.

[30] R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004. Cited pages 5 et 74.

[31] O. Bonorden, B. Judoiink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003. Cited page 74.

[32] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library - Design, Implementation and Performance. In *Proc. of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, San-Juan, Puerto-Rico, April 1999. Cited page 24.

[33] G. H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In Bougé et al. [37]. Cited page 19.

[34] G.-H. Botorog and H. Kuchen. Skil: an imperative language with algorithmic skeletons for efficient distributed programming. In *5th Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996. Cited page 19.

[35] G.-H. Botorog and H. Kuchen. Using algorithmic skeletons with dynamic data structures. In *Irregular'96*, pages 263–276. Springer Verlag, LNCS 1117, 1996. Cited page 20.

[36] G.-H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196:71–107, 1998. Cited page 19.

[37] L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors. *Euro-Par'96 Parallel Processing*, number 1123–1124 in LNCS, Lyon, August 1996. LIP-ENSL, Springer. Cited pages 123, 124 et 127.

[38] W. Bousdira, F. Gava, L. Gesbert, F. Loulergue, and G. Petiot. Functional Parallel Programming with Revised Bulk Synchronous Parallel ML. In K. Nakano, editor, *First International Conference on Networking and Computing (ICNC 2010), 2nd International Workshop on Parallel and Distributed Algorithms and Applications (PDAA)*, pages 191–196. IEEE Computer Society, 2010. Cited page 35.

[39] S. Breitinger. *Design and Implementation of the Parallel Functional Language Eden*. PhD thesis, 1998. Cited page 22.

[40] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. P. na Marí. Eden– the paradise of functional concurrent programming. In Bougé et al. [37]. Cited page 22.

[41] A. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: standard template adaptive parallel library. In G. Haber, D. D. Silva, and E. L. Miller, editors, *The 3rd Annual Haifa Experimental Systems Conference (SYSTOR 2010)*. ACM, 2010. Cited page 23.

[42] D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2004. Cited page 4.

[43] D. Caromel, L. Henrio, and M. Leyton. Type safe algorithmic skeletons. In *16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 45–53. IEEE Computer Society, 2008. Cited pages 25 et 87.

[44] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly France, 2000. freely available in english at http://caml.inria.fr/oreilly-book. Cited page 35.

[45] B. Chapman, G. Jost, and R. van Der Pas. *Using OpenMP*. MIT Press, 2008. about OpenMP 2.5. Cited page 3.

[46] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based P3L compiler. In *Proc. of the Parallel Computing Workshop (PCW'97)*, 1997. Cited page 15.

[47] P. Ciechanowicz and H. Kuchen. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 108–113, 2010. Cited page 20.

[48] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library Muesli – A Comprenhensive Overview. Technical Report Working Paper No. 7, European Research Center for Information Systems, University of Münster, Germany, 2009. Cited page 68.

[49] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989. Available at `http://homepages.inf.ed.ac.uk/mic/Pubs`. Cited pages 9 et 13.

[50] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004. Cited pages 19, 21 et 24.

[51] M. Cole and Y. Hayashi. Static Performance Prediction of Skeletal Programs. *Parallel Algorithms and Applications*, 17(1):59–84, 2002. Cited page 21.

[52] J. O. Coplien. Curiously recurring template patterns. *C++ Rep.*, 7:24–27, February 1995. Cited page 108.

[53] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993. ACM SIGPLAN. Cited page 7.

[54] F. Dabrowski and F. Loulergue. Functional Bulk Synchronous Programming in C++. In 21*st IASTED International Multi-conference, Applied Informatics (AI 2003), Symposium on Parallel and Distributed Computing and Networks*, pages 462–467. ACTA Press, february 2003. Cited page 21.

[55] M. Danelutto and P. Dazzi. Joint Structured/Unstructured Parallelism Exploitation in Muskel. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2006)*, LNCS. Springer, 2006. Cited page 18.

[56] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in p3l. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing*, Euro-Par '97, pages 619–628. Springer-Verlag, 1997. Cited page 15.

[57] M. Danelutto and M. Stigliani. Skelib: Parallel programming with skeletons in c. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par 2000, pages 1175–1184, 2000. Cited page 16.

[58] M. Danelutto and P. Teti. Lithium: A Structured Parallel Programming Environment in Java. LLNCS 2330, pages 844–853. Springer, 2002. Cited page 17.

[59] J. Darlington, A. J. Field, P. G. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel Programming Using Skeleton Functions. In *PARLE'93*. Springer Verlag, 1993. Cited page 65.

[60] R. Di Cosmo, S. Pelagatti, and Z. Li. A calculus for parallel computations over multidimensional dense arrays. *Computer Language Structures and Systems*, 33(3-4):82–110, 2007. Cited page 24.

[61] C. Dumont and F. Mourlin. A mobile computing architecture for numerical simulation. *CoRR*, 2007. Cited page 4.

[62] C. Dumont and F. Mourlin. Space based architecture for numerical solving. In *CIMCA/IAWTIC/ISE*, pages 309–314, 2008. Cited page 4.

[63] K. Emoto, K. Matsuzaki, Z. Hu, and M. Takeichi. Domain-Specific Optimization Strategy for Skeleton Programs. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings*, LNCS 4661, pages 705–714. Springer, 2007. Cited pages 21 et 68.

[64] J. Falcou and J. Sérot. Formal Semantics Applied to the Implementation of a Skeleton-Based Parallel Programming Library. In C. H. Bischof, H. M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007*, volume 15 of *Advances in Parallel Computing*, pages 243–252. IOS Press, 2007. Cited pages 22 et 25.

[65] J. Falcou, J. Sérot, T. Chateau, and J.-T. Lapresté. Quaff: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32:604–615, 2006. Cited page 22.

[66] M. J. Flynn. Very high speed computing systems. *Proc. IEEE*, 54(12):1901–1909, 1966. Cited page 2.

[67] C. Fonlupt, P. Marquet, and J.-L. Dekeyser. Data-parallel load balancing strategies. *Parallel Computing*, 24(11):1665 – 1684, 1998. Cited page 51.

[68] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual Symposium on Theory of Computing (STOC)*. ACM, May 1978. Cited page 5.

[69] L. Gesbert, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson. Systematic Development of Correct Bulk Synchronous Parallel Programs. In *The 11th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 334–340. IEEE Computer Society, 2010. Cited page 25.

[70] G. Gopalakrishnan and R. M. Kirby. Formal Methods for MPI Programs. *Electronic Notes in Theoretical Computer Science*, 193:19–27, 2007. Cited page 7.

[71] G. Gopalakrishnan and R. M. Kirby. Runtime verification methods for MPI. In *IPDPS*, pages 1–5. IEEE, 2008. Cited page 7.

[72] T. Y. H. Tang, K. Shen. Program transformation and runtime support for threaded mpi execution on shared-memory machines. *ACM Transactions on Programming Languages and Systems*, 22:673–700, 2000. Cited page 4.

[73] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202 – 220, 2009. Cited page 4.

[74] J. Hardtlein, A. Linke, and C. Pflaum. Fast expression templates. In *International Conference on Computational Science (2)*, pages 1055–1063, 2005. Cited page 109.

[75] J. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34:1–17, September 2006. Cited page 74.

[76] M. Hidalgo-Herrero and Y. Ortega-Mallén. An Operational Semantics for the Parallel Language Eden. *Parallel Processing Letters*, 12(2):211–228, 2002. Cited page 22.

[77] J. M. D. Hill, P. I. Crumpton, and D. A. Burgess. Theory, practice, and a tool for BSP performance prediction. In Bougé et al. [37]. Cited page 24.

[78] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998. Cited pages 6, 24 et 74.

[79] C. A. R. Hoare. An axiomatic basis for computer programmation. *Communication of the ACM*, 12(10):576–580, 1969. Cited page 7.

[80] Z. Hu, M. Takeichi, and W.-N. Chin. Parallelization in calculational forms. In *POPL'98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–328. ACM Press, 1998. Cited page 25.

[81] A. P. J. Dongarra, P. Luszczek. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. Cited page 74.

[82] E. Johnson and D. Gannon. Hpc++: experiments with the parallel standard template library. In *Proceedings of the 11th international conference on Supercomputing*, ICS 97, pages 124–131. ACM, 1997. Cited page 23.

[83] D. Kafura and J.-P. Briot. Introduction to Actors and Agents. *IEEE Concurrency*, 6(2):24–29, 1998. Cited page 4.

[84] Y. Karasawa and H. Iwasaki. A Parallel Skeleton Library for Multi-core Clusters. In *International Conference on Parallel Processing (ICPP)*, pages 84–91. IEEE Computer Society, 2009. Cited page 21.

[85] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. H. Hofman. Network performance-aware collective communication for clustered wide-aera systems. *Parallel Computing*, 27:1431–1456, 2001. Cited page 7.

[86] H. Kuchen. Implementing an object oriented design in curry. In *WFLP*, pages 499–509, 2000. Cited page 20.

[87] H. Kuchen. A Skeleton Library. In *8th International Euro-Par Conference*, LNCS 2400, pages 620–629. Springer, 2002. Cited page 20.

[88] H. Kuchen. Optimizing sequences of skeleton calls. In *Domain-Specific Program Generation*, pages 254–273, 2003. Cited page 20.

[89] H. Kuchen and M. Cole. The Integration of Task and Data Parallel Skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002. Cited pages 20 et 68.

[90] H. Kuchen and M. Poldner. On Implementing the Farm Skeleton. *Parallel Processing Letters*, 18(1):204–219, 2008. Cited page 68.

[91] H. Kuchen and J. Striegnitz. Higher-order functions and partial applications for a c++ skeleton library. In *Java Grande*, pages 122–130, 2002. Cited page 20.

[92] H. Kuchen and J. Striegnitz. Features from functional programming for a c++ skeleton library. *Concurrency - Practice and Experience*, 17(7-8):739–756, 2005. Cited page 20.

[93] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. Cited page 96.

[94] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.12. `http://caml.inria.fr`, 2010. Cited page 35.

[95] E. Levin. Grand challenges to computation science. *Commun. ACM*, 32:1456–1457, 1989. Cited page 2.

[96] M. Leyton. *Advanced features for algorithmic skeleton programming*. PhD thesis, Université de Nice Sophia Anti Polis, OCT 2008. Cited page 22.

[97] M. Leyton, L. Henrio, and J. M. Piquer. Exceptions for algorithmic skeletons. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *16th International Euro-Par Conference*, LNCS 6272, pages 14–25. Springer, 2010. Cited page 25.

[98] M. Leyton and J. M. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. In *PDP*, pages 289–296, 2010. Cited page 22.

[99] O. Lobachev and R. Loogen. Estimating Parallel Performance, A Skeleton-Based Approach . In *4th workshop on High-Level Parallel Programming and Applications (HLPP)*. ACM, 2010. Cited page 24.

[100] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005. Cited page 22.

[101] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS)*, LNCS 3515, pages 1046–1054. Springer, 2005. Cited page 35.

[102] A. S. M. Baker, B. Carpenter. Mpj express: Towards thread safe java hpc. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona, Spain*. IEEE, 2006. Cited page 4.

[103] K. Matsuzaki and K. Emoto. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *21st International Workshop on Implementation and Application of Functional Languages (IFL)*, LNCS. Springer, 2009. Cited page 21.

[104] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *InfoScale'06: Proceedings of the 1st international conference on Scalable information systems*. ACM Press, 2006. Cited pages 21 et 68.

[105] W. F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996. Cited page 5.

[106] B. Nichols, D. Buttlar, and J. Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly, 1996. Cited page 3.

[107] R. Nishtala, N. Patel, K. Sanghavi, and K.Chakrabarti. Automatic tuning of collective communication operations in mpi. 2003. Cited page 24.

[108] OpenMP Architecture Review Board. OpenMP Application Program Interface version 3.0, may 2008. Cited page 3.

[109] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976. Cited page 7.

[110] M. Parkinson. The next 700 separation logics. In G. Leavens, P. O'Hearn, and S. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010. Cited page 7.

[111] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs*. PhD thesis, Univ. Pisa, 1993. Cited page 15.

[112] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998. Cited page 15.

[113] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10:127–143, June 2007. Cited page 24.

[114] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *PDP*, pages 427–436, 2009. Cited page 4.

[115] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library (stapl). In *LCR*, pages 402–409, 1998. Cited page 23.

[116] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007. Cited page 23.

[117] G. S. R.L. Graham. Mpi support for multi-core architectures: Optimized shared memory collectives. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140. Springer-Verlag, 2008. Cited page 4.

[118] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. Cited page 7.

[119] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992. Cited page 66.

[120] D. B. Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. Optimising data-parallel programs using the BSP cost model. In *Europar'98*, volume 1470 of *LNCS*, pages 698–715. Springer Verlag, 1998. Cited page 21.

[121] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997. Cited page 5.

[122] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998. Cited page 3.

[123] B. Stroustrup. To move or not to move. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3174.pdf`, 2010. Cited page 114.

[124] H. Tanno and H. Iwasaki. Parallel skeletons for variable-length lists in sketo skeleton library. In H. J. Sips, D. H. J. Epema, and H.-X. Lin, editors, *Euro-Par*, volume 5704 of *Lecture Notes in Computer Science*, pages 666–677. Springer, 2009. Cited page 21.

[125] X. Teruel, C. Barton, A. Duran, X. Martorell, E. Ayguadé, P. Unnikrishnan, G. Zhang, and R. Silvera. OpenMP tasking analysis for programmers. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'09)*, pages 32–42. ACM, 2009. Cited page 3.

[126] J. Tesson and F. Loulergue. A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In *11th International Conference on Computational Science (ICCS 2011)*, Procedia Computer Science, pages 36–45. Elsevier, 2011. Cited page 25.

[127] R. Thakur and W. Gropp. Improving the performance of collective operations in mpich. In *PVM/MPI'03*, pages 257–267, 2003. Cited page 24.

[128] The BSML Development Team. The BSML Library version 0.5. `http://traclifo.univ-orleans.fr/BSML`, august 2010. Cited page 35.

[129] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`. Cited pages 35 et 115.

[130] N. Thomas, S. Saunders, T. G. Smith, G. Tanase, and L. Rauchwerger. Armi: a high level communication library for stapl. *Parallel Processing Letters*, 16(2):261–280, 2006. Cited page 23.

[131] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in stapl. In *PPOPP*, pages 277–288, 2005. Cited page 23.

[132] S. L. Thomas, G. Tanase, L. K. Dale, J. M. Moreira, L. Rauchwerger, and N. M. Amato. Parallel protein folding with stapl. *Concurrency and Computation: Practice and Experience*, 17(14):1643–1656, 2005. Cited page 23.

[133] S. Vadhiyar, G. Fagg, and J. Dongarra. Towards an accurate model for collective communications. volume 18, pages 41–50, 2001. Cited page 24.

[134] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103, 1990. Cited page 5.

[135] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28:1709–1732, 2002. Cited page 18.

[136] T. Veldhuizen. Expression templates. pages 475–487, 1996. Cited pages 48 et 107.

[137] T. Veldhuizen. Techniques for scientific C++, 1999. Cited page 109.

[138] P. Wadler. Monads for Functional Programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 24–52. Springer, 1995. Cited page 96.

[139] A. Zavanella. Optimising skeletal-stream parallelism on a BSP computer. In *Euro-Par*, volume 1685 of *LNCS*, pages 853–857. Springer, 1999. Cited page 21.

[140] A. Zavanella. The skel-BSP global optimizer: Enhancing performance portability in parallel programming. In *Euro-Par*, volume 1900 of *LNCS*, pages 658–667. Springer, 2000. Cited page 21.

[141] A. Zavanella. Skel-BSP: Performance portability for skeletal programming. In *HPCN*, volume 1823 of *LNCS*, pages 290–299. Springer, 2000. Cited page 20.

[142] A. Zavanella and S. Pelagatti. Using BSP to optimize data distribution in skeleton programs. In *HPCN*, volume 1593 of *LNCS*, pages 613–622. Springer, 1999. Cited page 21.

Ce document a ecrit a l'aide de texte GNU Emacs et du logiciel de composition typographique LaTeX 2ε.

Noman JAVED

Squelettes algorithmiques méta-programmés : implantations, performances et sémantique

Les approches de parallélisme structuré sont un compromis entre la parallélisation automatique et la programmation concurrentes et réparties telle qu'offerte par MPI ou les Pthreads. Le parallélisme à squelettes est l'une de ces approches. Un squelette algorithmique peut être vu comme une fonction d'ordre supérieur qui capture un algorithme parallèle classique tel qu'un pipeline ou une réduction parallèle. Souvent la sémantique des squelettes est simple et correspondant à celle de fonctions d'ordre supérieur similaire dans les langages de programmation fonctionnels. L'utilisation combine les squelettes disponibles pour construire son application parallèle.

Lorsqu'un programme parallèle est conçu, les performances sont bien sûr importantes. Il est ainsi très intéressant pour le programmeur de disposer d'un modèle de performance, simple mais réaliste. Le parallélisme quasi-synchrone (BSP) offre un tel modèle. Le parallélisme étant présent maintenant dans toutes les machines, du téléphone au super-calculateur, il est important que les modèles de programmation s'appuient sur des sémantiques formelles pour permettre la vérification de programmes.

Les travaux menés on conduit à la conception et au développment de la bibliothèque Orléans Skeleton Library ou OSL. OSL fournit un ensemble de squelettes algorithmiques data-parallèles quasi-synchrones. OSL est une bibliothèque pour le langage C++ et utilise des techniques de programmation avancées pour atteindre une bonne efficacité. Les communications se basent sur la bibliothèque MPI. OSL étant basée sur le modèle BSP, il est possible non seulement de prévoir les performances des programmes OSL mais également de fournir une portabilité des performances. Le modèle de programmation d'OSL a été formalisé dans l'assistant de preuve Coq. L'utilisation de cette sémantique pour la preuve de programmes est illustrée par un exemple.

Mots clés : Squelettes algorithmiques, parallélisme quasi-synchrone, prévision et portabilité des performances, sémantique formelle, applications

Metaprogrammed algorithmic skeletons : implementations, performaces and semantics

Structured parallelism approaches are a trade-off between automatic parallelisation and concurrent and distributed programming such as Pthreads and MPI. Skeletal parallelism is one of the structured approaches. An algorithmic skeleton can be seen as higher-order function that captures a pattern of a parallel algorithm such as a pipeline, a parallel reduction, etc. Often the sequential semantics of the skeleton is quite simple and corresponds to the usual semantics of similar higher-order functions in functional programming languages. The user constructs a parallel program by combined calls to the available skeletons.

When one is designing a parallel program, the parallel performance is of course important. It is thus very interesting for the programmer to rely on a simple yet realistic parallel performance model. Bulk Synchronous Parallelism (BSP) offers such a model. As the parallelism can now be found everywhere from smart-phones to the super computers, it becomes critical for the parallel programming models to support the proof of correctness of the programs developed with them.

The outcome of this work is the Orléans Skeleton Library or OSL. OSL provides a set of data parallel skeletons which follow the BSP model of parallel computation. OSL is a library for C++ currently implemented on top of MPI and using advanced C++ techniques to offer good efficiency. With OSL being based over the BSP performance model, it is possible not only to predict the performances of the application but also provides the portability of performance. The programming model of OSL is formalised using the big-step semantics in the Coq proof assistant. Based on this formal model the correctness of an OSL example is proved.

Keywords : Algorithmic Skeletons, bulk synchronous parallelism, performance predictability and portability, formal semantics, applications

PUCVL Pôle Universités Centre Val de Loire

Laboratoire d'Informatique Fondamentale d'Orléans

UNIVERSITE D'ORLEANS