TARTU UNIVERSITY
Faculty of Mathematics and Computer Science
Institute of Computer Science
Computer Science

Ilja Kromonov

# BSPlib Java Interface for Parallel Scientific Computing Applications

B.Sc. Thesis (6 EAP)

Supervisor: Pelle Jakovits

Author: ..... " ....." may 2012

Supervisor: ..... " ....." may 2012

Approved for defence

Professor: ..... " ....." may 2012

Tartu 2012

# Contents

**Introduction** **4**

**1 BSP and Scientific Computing** **6**
1.1 Bulk Synchronous Parallel Model . . . . . 7
1.2 BSPlib Standard . . . . . 8
1.2.1 Advantages Over MPI . . . . . 9
1.3 State of the Art . . . . . 9

**2 Interface Implementation** **11**
2.1 Data Types . . . . . 11
2.2 Array Access . . . . . 12
2.2.1 Multidimensional Arrays . . . . . 13
2.3 Multiple Return Values . . . . . 13
2.4 API . . . . . 14
2.4.1 Structure . . . . . 16

**3 Use Case Scenario** **17**
3.1 Heat Equation . . . . . 17
3.2 Numeric Solution to the Heat Equation . . . . . 19
3.2.1 Finite Difference Approximation . . . . . 19
3.3 Solving Systems of Linear Algebraic Equations . . . . . 22
3.3.1 Conjugate Gradient Method . . . . . 24

**4 Experiment Results** **28**

**Conclusion** **31**

**Resümee** **33**

**References** **34**

## Abstract

This work presents a Java$^{\mathrm{tm}}$ interface to a native BSPlib library for implementing parallel algorithms in a structured way (as described by the BSP model), using the Java programming language. To compare the created library to existing parallel programming solutions, a typical physics simulation application is created. It employs the parallel conjugate gradient method for solving systems of linear algebraic equations, a common scientific computing algorithm, which is challenging from a parallelization standpoint. Using the results from running the test application, the Java BSPlib interface is compared to various MPI (Message Passing Interface) implementations.

# Introduction

This work presents a Java interface to a native BSPlib implementation, with the goal of showing the viability of the BSP (Bulk Synchronous Parallel) model for HPC (high performance computing). The BSPlib standard [Jon98] is a specification for a general-purpose parallel programming library based on the BSP model. The presented Java library can be used for implementing algorithms in a structured way (as described by the BSP model), using the Java programming language.

In addition to the aforementioned library, a large part of this work is finding and implementing a practical use case scenario, that can be utilized for giving an apt comparison of the presented BSPlib interface with existing parallel computing solutions. To perform adequate comparison, the use case must be sufficiently computationally intensive to require parallelization, which in turn has to be non-trivial, requiring synchronization at multiple points in the algorithm.

Cases that fulfill such requirements are plentiful in the area of scientific computing and the chosen problem for the given use case is simulation of heat conduction in solids. The computational bottleneck for this simulation is finding a solution to a large system of linear algebraic equations (SLAE). These systems are difficult to solve via direct methods (such as Gaussian elimination), so it is often more fitting to approximate[1] their solution through iterative approaches. One such approach is the conjugate gradient method [She94]. It's main advantage is a relatively quick convergence to the correct solution, however, parallelization is challenging due to it's complex structure and iterative nature, which fits the requirements.

The BSP model, proposed by Valiant as "neither a hardware nor a programming model but something in between" [Val90], is promoted in this work as a viable solution to parallel programming challenges faced by today's developers, as it is well suited for most types of algorithms and it's simplicity allows for easy adaptation of sequential computation to parallel infrastructures. This is important in light of migration of many computation work to the *cloud* environment. With

[1] Simulations do not generally require 100% accurate solutions, and approximation schemes usually allow for any desired accuracy.

it's promise of nearly infinite resources on demand, the *cloud* presents a lucrative prospect for any areas that benefits from parallelism.

Certain features of the BSP model allow for fault tolerance mechanisms to be seamlessly integrated into a parallel programming framework utilizing this model. Providing fault tolerance is not in the scope of this work, however, other advantages of the BSP model can be used by utilizing the BSPlib standard. The BSP model, with it's notions of supersteps and barriers, provides clear guidelines for designing parallel versions of algorithms in a structured way and allows for deadlock free communication patterns through asynchronous data transmission operations, making it a good choice for implementing fine-grained parallel algorithms and making a *cloud*-based 'BSP computer' all the more viable.

Chapter one of this work provides an introduction to the BSP model and the BSPlib standard and gives an overview of the current state of BSP in Java. Chapter two describes the provided interface and outlines the differences between it's API and native BSPlib one. Chapter three describes in detail the use case chosen to test the provided solution and presents a step-by-step derivation of the computational SLAE problem from the notion of simulating heat diffusion. In addition, this chapter also introduces the method used to solve the SLAE problem and describes it's algorithm under the BSP model. The fourth chapter presents experiment results of the example simulation utilizing BSPlib in Java and compares them to ones achieved using various MPI (Message Passing Interface) solutions.

# Chapter 1

# BSP and Scientific Computing

Scientific computing is an area of high performance computing (HPC) that deals with calculations based on models of certain phenomena. In general terms these are numeric simulations based on specific models, their analysis and optimization of the models' derived processes. These range from weather and galaxy formation simulations on the macro level, to processor cooling and subatomic particle simulations on the opposite side of the scale. These are usually computationally and data intensive tasks and require a large amount of computing power and memory capacity.

The physical limitations of current processor technology only allow so much computing power to be gained from a single CPU. The solution is parallelization, which in the area of HPC has been traditionally achieved by means of supercomputers - highly parallel systems with thousands of cores, consequently costing millions of dollars not only to build, but also for daily use and maintenance.

With the advent of the *cloud* a new alternative for scientific computing emerges. With it's illusion of infinite resources, *cloud* computing allows loaning of computation time on demand with a flexible pay-as-you-use billing model. However, applications are placed in an environment associated with a high risk of hardware failure. The cause - use of commodity equipment by most *cloud* service providers, to lessen the cost of data center components. This means fault tolerance is of utmost importance for any long-running process in this environment.

One framework, which has found widespread use in *cloud*-based parallel computing for exactly this reason is Apache$^{\mathrm{tm}}$ Hadoop$^{\mathrm{tm}}$ MapReduce [Apaa]. It provides fault tolerance and replication of both data and computation in attempt to guarantee that the started task will produce a result. Originally introduced by Google$^{\mathrm{tm}}$ in 2004 [DG04], MapReduce excels at solving data-heavy embarrassingly parallel problems, however has trouble with more sophisticated algorithms. This claim is evidenced by Hadoop's inability to cope with iterative algorithms

[SJV]. Furthermore, even MapReduce frameworks that are aimed at iterative computation, such as Twister MapReduce [ELZ+10], have trouble with most scientific computing problems, with one of the main reasons being that by design MapReduce processes are stateless. The stateless status of a process implies, that no state information is associated with the given process at any time, ensuring that any part of input data is eligible for any of the available processes without affecting the outcome. This concept ensures that failure of one of the nodes does not affect the sequential consistency of the program and is at the core of the MapReduce fault tolerance mechanism.

Embarrassingly parallel problems[1] are more rare than one might wish for, more so in the area of scientific computing. At the heart of many a scientific computing problem lies the challenge of solving large systems of linear algebraic equations (SLAE). The tried and trusted method of solving SLAE is the conjugate gradient algorithm - an iterative approach to approximating the solution with whatever accuracy desired. Unfortunately due to reasons stated above it's adaptation to the MapReduce model is unfeasible at best, both due to it's iterative nature, as well as large quantities of state information.

In previous work [JKS11] we attempted to remedy the issue of solving SLAE in the *cloud* by running an embarrassingly parallel algorithm, of the Monte Carlo kind, with Hadoop MapReduce. However that approach turned out to have more drawbacks than advantages and CG remains the better solution. For this work CG and the SLAE problem symbolize scientific computing, due to being as commonplace as they are in this field (the SLAE problem is at the base of almost every numeric simulation).

This is where the Bulk Sycnhronous Parallel model comes into play. A parallel computing framework based on the BSP model has all the properties that make it as viable for integration with the *cloud* environment as one based on MapReduce, while providing a more traditional experience for the programmer (similar to MPI). The majority of MapReduce frameworks employed in the *cloud* use the Java technology, meaning that there is a large target audience for a BSP-based framework on that platform. Several BSP solutions exist on the Java platform, but none of them are general-purpose programming libraries and follow the BSPlib standard at the same time.

## 1.1 Bulk Synchronous Parallel Model

In his 1990 paper [Val90] Leslie G. Valiant argued, that to properly utilize existing computing resources for parallel computation, a bridging model between

[1] Those that take little to no synchronization or communication between computing nodes.

software and hardware has to be introduced, that would streamline the move of sequential computation to the parallel infrastructure. BSP was his proposed candidate for this role, and while it may not have been widely adopted for it's initial purpose, it inspired new programming models and several parallel programming libraries, the most notable being BSPlib [Jon98].

A BSP based computation procedure consists of a series of supersteps, each divided into three stages:

- **Concurrent computation** - each process does it's part of the computation in parallel with others, using only data local to the process' memory, communication operations can be queued up but do not occur immediately.
- **Communication** - once a process' computation is finished, all communication operations, queued up during the previous stage, happen *en masse* (hence the *bulk* in the model's name).
- **Barrier synchronization** - ensures that every process has completed it's computation and all the communication operations before continuing to the next superstep

The advantage of this scheme is elimination of any circular data dependencies, hence no deadlocks may happen in a BSP program. Furthermore it is easy to adapt any algorithm to follow this mechanic and relatively easy to estimate the effect of this parallelization on performance of said algorithm.

At a glance, the parallelization cost can be judged by looking at the prevalence of barriers in a BSP program. The handling of communication operations of any single superstep as a whole allows one to estimate the cost of the given superstep separately from others. Since all processes have to complete their computation and communication stages before reaching the barrier, the cost of said stage can go up dramatically if any of the processes, for whatever reason, lags behind the rest. At present, the only way to tackle this problem is to make sure processes get roughly equivalent parts of the initial problem (which may not always be as trivial as it sounds) and nodes are of equal computational power. The quantification of the cost of parallel application synchronization is a large part of the BSP model and is used for analyzing the efficiency of parallel algorithms and how well the underlying architecture handles them [Val90], but is not discussed in this work.

## 1.2 BSPlib Standard

BSPlib is a programming library specification based on the BSP model. Two of it's major implementations are the Oxford BSP toolset [Hil] and the Paderborn PUB library [Ola03]. They provide 'Bulk Synchronous Message Passing'

(BSMP) and 'Direct Remote Memory Access' (DRMA) modules through a set of low-level primitives of various semantics, supporting different programming styles.

Unfortunately the most recent updates to the libraries in question have been done in 1998 and 2002 for the Oxford BSP toolset and PUB respectively. Consequently, these libraries are designed and optimized for specific (and for the most part outdated) architectures. Regardless, these are comprehensive low-level communication libraries, which still find use today, likely due to the lack of alternatives more than anything.

A more recent implementation of the BSPlib standard is BSPonMPI [Sui]. It is designed to more effectively make use of modern architectures, by using MPI to perform communication operations. It tries to be backwards compatible with the Oxford BSPlib toolset by using the same API (Application Programming Interface). MPI implementations are available on many platforms and are continuously being developed, making BSPonMPI a good choice for the experiments conducted for this work.

### 1.2.1 Advantages Over MPI

One of the main advantages of BSPlib over MPI is the simplicity of it's API - the core library consists of merely 20 primitives [Hil], compared to MPI's 273 [mpia]. This allows for a much easier experience for programmers not familiar with parallel computing. While MPI is more versatile and allows the creation of virtually any communication pattern using it's powerful API, it's code tends to be error prone and difficult to maintain.

Since BSPlib is based on the BSP model, the programmer does not have to worry about ensuring his code is deadlock free, which is often the most challenging part of writing MPI programs. Another advantage is the relative ease of providing fault tolerance to programs using it, as shown by Hill et al. [HDL97]. With the complexity and variety of MPI programs such a mechanism would be very difficult to implement, however MPI operations can be used to provide communication for parallel programming frameworks more suitable for such a goal, such as BSPonMPI[Sui], which encapsulates communication using MPI within a BSPlib implementation.

## 1.3 State of the Art

Currently there is a striking lack of general-purpose parallel programming libraries based on the BSP model on the Java platform. There is Google's Pregel [MAB+09], which is a framework inspired by the BSP model that is designed

strictly for graph processing problems. A similar approach is taken by the Apache Hama project [Apac], which aims to be a common computation engine, but as it is based on Apache Hadoop, it suffers from the same problems, one of which is a significant overhead from object serialization, caused by encapsulation of transmitted data in wrapper objects. In addition to that, the framework lacks many features of the alternative BSPlib standard, such as 'Direct Remote Memory Access' (DRMA) functionality, which makes it less suitable for high performance computation. On the other hand, it is built upon an established distributed system framework[2]. The other available Java library is MulticoreBSPlib [YB11], which follows the BSPlib specification, but is aimed strictly at shared-memory systems, so it's scope is limited.

In 2001 Yan Gu et al. [GsLC01] attempted to create a Java implementation of the BSPlib standard, but due to data serialization overhead in their proposed solution, that project (JBSP) was abandoned and the software was not made available to the general public.

[2]Hadoop, which includes it's own distributed file system HDFS [Apab], that is imperative for functionality of both Hadoop MapReduce and Hama.

# Chapter 2

# Interface Implementation

The approach taken when creating the presented BSPlib interface for Java is to use an existing native BSPlib implementation, such as the Oxford BSP toolset, and with the help of JNI (Java Native Interface) create a proxy library, that allows for low-level BSPlib primitives to be used in Java code. A similar approach has been used by ScientificPython [JOP$^+$ ], to provide a BSP interface for Python, and mpiJava [Car98] to do the same for native MPI (Message Passing Interface) and Java.

Writing a JNI wrapper involves the creation of a proxy library in a lower level language[1], that does all the necessary work for calling native functions, such as accessing pointers to native arrays behind Java array types. The interface has been created to use the Oxford BSP toolset API and will work with any BSPlib implementation, that conforms to it, such as BSPonMPI - an implementation of BSPlib built on top of native Message Passing Interface (MPI) libraries.

There is no *de facto* standard Java API specification for BSPlib as there is for MPI [Bry98], so the provided API (shown in table 2.4) mostly tries to be a one-to-one binding of the native libraries. The interface tries to stick to the naming convention used by the native libraries where possible, whenever new functions had to be created the Java standard method naming convention was used.

No specific name is given to the created solution, as such it is referred to simply as 'the BSPlib interface' or sometimes simply 'library'.

## 2.1 Data Types

The operations provided by the interface can only be used with arrays of Java primitive types, as the size of arbitrary objects cannot be effectively determined.

[1] There are JNI interfaces for *C++* and *C*.

Technically, it is possible to send user-made objects, by converting them into series of bytes and transmitting the resulting byte arrays. The process is known as serialization (or sometimes marshaling), and has a significant overhead, when dealing with HPC. This overhead was one of the reasons, due to which an implementation of BSPlib for Java from as early as 2001 was abandoned [GsLC01]. The authors used Java automatic serialization mechanisms when exchanging data between processes, which is one of the costliest means of serialization as determined by the 'Java serialization benchmark' project [ser]. For this reason the library does not provide any built-in way of transmitting arbitrary objects. They can be still be manually (or using Java's *Serializable* interface) converted to bytes and transmitted as such, but this practice should not be encouraged for the purposes of HPC in Java.

## 2.2 Array Access

To execute native communication operations data is passed as references to Java arrays through methods provided by the interface. However, these arrays are passed to JNI methods as objects of *jarray* type and data stored within cannot be accessed directly. Instead, several methods are provided by the JNI API for either getting a pointer to the native data or copying it into a new array. For performance reasons it's better to get a direct pointer to the original data, however, Java manages it's designated memory area by occasionally moving data around (defragmentation) or removing it (garbage collection), which means the location of arrays in native memory may change.

The function *Get*<type>*ArrayElements* attempts to retrieve a pointer to the native array elements (if the JVM supports a mechanism called 'pinning'[2]) but is likely to copy the array and return a pointer to this newly allocated memory. Even if only a small section of the array is needed, a copy of the whole array is created, which is obviously not efficient.

Since version 1.2 of Java JDK (Java 6 is the stable version at the time of writing) a new method for retrieving the pointer to array data is available. The function *GetPrimitiveArrayCritical* makes it much more likely that the data is not copied and a pointer to original data is returned. This is achieved by placing certain restrictions on native JNI code and temporarily disabling some features of the JVM, while the pointer is held by native code [jni].

This function is used by the wrapper library for getting the array pointers when creating DRMA registrations and the pointers are released when the registrations

[2]The JVM 'pins' the array on the heap, so it does not get moved during garbage collection and the pointer to it does not become invalid.

are removed, as such any registrations should be kept short lived as the use of this function may disable Java's garbage collecting mechanism among other things.

### 2.2.1 Multidimensional Arrays

A *C*/*C++* multidimensional array is stored as a contiguous one-dimensional array, which is not the case for multidimensional Java arrays. As a Java array is essentially a Java object, a multidimensional array is an array of objects, which are not necessarily allocated contiguously on the Java heap. This means that the traditional pointer and offset approach would not work for such a structure and the array element retrieval functions are almost certain to return a copy of the elements.

Use of one-dimensional arrays is encouraged, not only due to the aforementioned reason, but also to avoid the memory and access time overhead imposed by use of multidimensional Java arrays.

## 2.3 Multiple Return Values

In the *C* programming language multiple return values are provided through pointers, given as arguments to the function in question. As Java does not have pointer data types, it was decided to split such functions into separate ones for the Java side of the API. This is preferred to using a custom tuple-like object as the return type, for the sake of making the API more transparent. Functions that fit into this category are *bsp_get_tag* and *bsp_qsize*.

The function *bsp_get_tag* provides information on the BSMP message in the queue and is divided into three separate methods:

- *BSP.getMessageLength* - returns the element count of the next message
- *BSP.getMessageTag* - returns the 4 byte tag portion set by the user
- *BSP.getMessageType* - returns the type identifier of the next message

All of the aforementioned methods will throw a Java exception (*BSPException*) if the queue is empty. To make sure that is not the case the native function *bsp_qsize* provides the queue size in both packets and bytes, and is split into *BSP.getQueueSizePackets* and *BSP.getQueueSizeBytes* accordingly.

The overhead of calling native functions twice, in case both results are needed, can be avoided by storing the result and returning the appropriate part when needed, instead of doing a second native function invocation. Results are stored until *bsp_move* or *bsp_sync* operations occur, which change the state of the queue.

## 2.4 API

Most of the core BSPlib primitives (with a few exceptions) have been given an equivalent in the interface's API.

### Initialization

The initialization function *bsp_init*, which on the native side takes a pointer to user's SPMD (Single Program Multiple Data) code of the parallel program, is not applicable for implementing in a Java wrapper library. No function pointers exist in the Java language, and while it is possible to provide such functionality through a callback object, it's use would be unnecessarily restricting to the programmer, as such this approach is not supported.

The *BSP.init* method needs to be explicitly called by the user in order to load the native wrapper library. It's invocation also ensures that the native initialization function is called. While this behavior is not mandatory for Oxford BSP toolset, the alternative implementation (BSPonMPI) requires this in order to initialize the underlying MPI implementation. BSPonMPI requires the user to provide a pointer to his native SPMD code and terminates MPI processes that have completed it's execution. This means that using this library with the Java interface requires minor modifications to BSPonMPI's source code[3].

### BSMP (Bulk Synchronous Message Passing)

There is no *bsp_set_tagsize* method available, and while there is nothing obstructing the implementation of this function, the tag size has been fixed at 5 bytes for the purposes of this library. This change further simplifies the API, where the user has a 4 byte tag (that of a standard integer), to be used as one sees fit, and an additional byte is used to store data type information for BSMP messages. This allows for the API functions to have one argument less, as instead of explicitly giving the data type of each message it can be implicitly derived from the provided array, where *bsp_send* is implemented as a separate method for each primitive data type. On the receiving side the user can simply cast the object returned by *bsp_move* into the appropriate type. Should the program be written in a way, that makes the data type ambiguous on the receiving end, the type identifier for the next message can be retrieved with *BSP.getMessageType*.

[3]The *exit()* call, which terminates processes other than the root process needs to be removed and only the MPI initialization part should remain.

| CLASS | OPERATION | MEANING | JAVA API |
|---|---|---|---|
| Initialisation | bsp_begin | Start of SPMD code | BSP.begin |
| | bsp_end | End of SPMD code | BSP.end |
| | bsp_init | Simulate dynamic processes | BSP.init* |
| Halt | bsp_abort | One process stops all | BSP.abort |
| Enquiry | bsp_nprocs | Number of processes | BSP.nprocs |
| | bsp_pid | Find my process identifier | BSP.pid |
| | bsp_time | Local time | BSP.time |
| Superstep | bsp_sync | Barrier synchronisation | BSP.sync |
| DRMA | bsp_push_reg | Make area globally visible | BSP.push_reg |
| | bsp_pop_reg | Remove global visibility | BSP.pop_reg |
| | bsp_put | Copy to remote memory | BSP.put |
| | bsp_get | Copy from remote memory | BSP.get |
| BSMP | bsp_set_tagsize | Choose tag size | |
| | bsp_send | Send to remote queue | BSP.send |
| | bsp_qsize | Number of messages in queue | BSP.getQueueSizePackets<br>BSP.getQueueSizeBytes |
| | bsp_get_tag | Getting the tag of a message | BSP.getMessageLength<br>BSP.getMessageTag<br>BSP.getMessageType |
| | bsp_move | Move from queue | BSP.move |
| High Performance | bsp_hpput | Unbuffered communications | ** |
| | bsp_hpget | | |
| | bsp_hpmove | | |

* not the same semantics as the native library

** not methods of their own in the Java API

Table 2.1: BSPlib core primitives [Hil] and it's Java binding API.

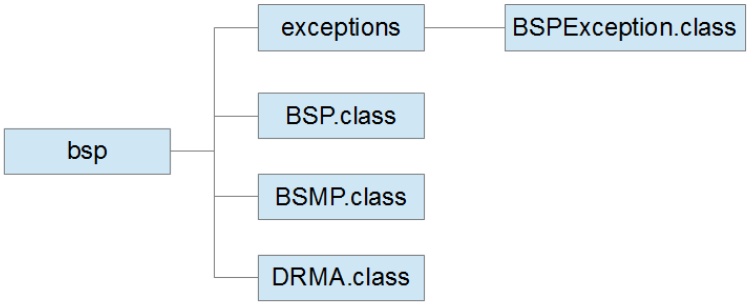


Figure 2.1: Structure of the BSPlib Java interface.

**DRMA (Direct Remote Memory Access)**

The last difference from the native BSPlib API is the way 'high performance' functions are invoked. Instead of using a method with a *hp* prefix for each of the Java primitive types, a flag is used to specify which version of the native function is to be used. This flag can be set using the *BSP.setUnbuffered* method. The flag can be set at any time, meaning the high performance and the general versions of these functions can be used interchangeably, should one have reason to do so. Additionally, it enables the use of the unbuffered BSMP function *bsp_hpmove*, which does not copy the message in the system queue, but receives a pointer to it directly.

It has to be noted, that *bsp_hpput* and *bsp_hpget* are in violation of the BSP model, as these communications may happen at any time during a superstep and not in bulk with the rest of the communication operations at the barrier.

### 2.4.1 Structure

Figure 2.1 shows the package structure and classes of the Java side of the BSPlib interface. All functions described in table 2.4 are accessible as static methods from the *bsp.BSP* class, the *bsp.BSMP* and *bsp.DRMA* classes contain the native methods declarations of their respective modules. Invocations of methods in these classes are declared as *final*, to encourage the compiler to inline them.

For each class file (except *BSPException*) a header file is generated for creation of the JNI proxy library and an implementation is written using the *C++* programming language.

# Chapter 3

# Use Case Scenario

Testing the applicability of the created BSPlib interface (and the BSP model in general) to scientific computing problems requires a practical use case scenario. It's solution has to be sufficiently computationally intensive to require parallelization with the need of synchronization at multiple points. This chapter describes such a use case, giving a step-by-step derivation of the computational SLAE problem from the notion of simulating heat diffusion. This provides incentive for using the CG algorithm on a SLAE of a very specific type, as opposed to solving randomly generated systems, which are not necessarily applicable to any real world scenario.

It has to be noted that the created experimental application does not in any shape or form resemble actual enterprise simulation software, that is applicable to a wide range of actual processes, e.g. CPU heatsink design. Rather it solves a general problem from the given domain with input data that is still very specific to the problems of it's kind. In other words, while the test case is not a real world scenario, it has all the characteristics of one, which should satisfy the goals we aim to accomplish.

The heat equation is used in modeling a number of different phenomena in physics, mathematics and even finance, as such the description of the solution method described in this chapter is specific to the problem at hand.

## 3.1 Heat Equation

Modeling of heat flux with the passage of time is achieved through solving a partial differential equation known as the heat equation, which is derived from Fourier's law and the law of conservation of energy [ATP84]. In it's most general

form the equation (also known as the diffusion-convection equation) is:

$$c_p\rho\left[\frac{\partial u}{\partial t} + \nabla\cdot(\mathbf{v}u)\right] = \nabla\cdot[k\nabla u] + q \tag{3.1}$$

where $k$ is the thermal conductivity, $c_p$ is the specific heat capacity, $\rho$ is the material density, $\mathbf{v}$ is the velocity field, and $q$ is the internal heat generation. The heat equation has two parts: the diffusion part characterized by the conductivity field function and the advection part characterized by the velocity field function. The symbol $u$ is the dependent variable, in three dimensions it is a function $u(x, y, z, t)$ of three spatial coordinates and time, and denotes the temperature at position $(x, y, z)$ at an instant in time $t$.

Finding the velocity field $\mathbf{v}$ is an integral part of simulating fluid dynamics and requires the solution of the more complex Navier-Stokes equation, which would increase the complexity of the test case (additional SLAEs that need solving). To keep it simple the advection part can be safely dropped from the equation 3.1, effectively removing the movement of material from the model, and focusing on heat conduction. The resulting simplified equation is:

$$c_p\rho\frac{\partial u}{\partial t} = \nabla\cdot[k\nabla u] + q \tag{3.2}$$

where on the right-hand side we have the divergence $(\nabla\cdot)$ of the gradient $k\nabla u$, which is $k$ times the Laplacian of $u$. Since the Laplacian is a linear operator and $k$ is a scalar value, we get:

$$c_p\rho\frac{\partial u}{\partial t} = k\nabla^2 u + q \tag{3.3}$$

Simplifying equation 3.3 further, we assume a model without internal heat generation and divide through the equation by $c_p\rho$ to get the form, which the simulation will based on:

$$\frac{\partial u}{\partial t} = \alpha\nabla^2 u \tag{3.4}$$

or the specific three dimensional case, expanding the Laplacian in Cartesian coordinates:

$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) \tag{3.5}$$

where $\alpha = \frac{k}{c_p\rho}$ and is the thermal diffusivity, a material specific quantity measured in $\frac{m^2}{s}$, which shows the speed of penetration into the body of an applied thermal load at an object's surface.

## 3.2 Numeric Solution to the Heat Equation

Solving partial differential equations (PDE) analytically can be difficult and sometimes even impossible. Even if a solution can be found, it is not guaranteed to be easy to compute for specific cases of the represented problem. One alternative way to find the solution to a PDE is to numerically approximate it. We will use the finite difference method, as it is easier to implement than some of the alternatives, such as the finite element method, but has the same computational complications, in the form of SLAE, that we aim to solve for this example simulation.

### 3.2.1 Finite Difference Approximation

In short, the finite difference method of solving differential equations is based around substituting continuous derivatives with difference quotients

$$\frac{\partial f}{\partial x} \approx \frac{f(x_{i+1}) - f(x_i)}{\Delta x} \tag{3.6}$$

to approximate the result of the equation on a mesh of points, obtained by discretizing the continuous solution domain $\Omega$ into a finite set of discrete points $\Omega^*$. Equation 3.6 is an example of a forward difference of quantity $x$ about the point $x_i$, where $i \in (0, 1, \ldots, N), N = |\Omega^*|$ and $\Delta x$ is the point difference between $x_i$ and $x_{i+1}$.

Other common difference types are the backward difference and central difference, from which finite difference based solution schemes for the heat equation can be derived, an overview for which can be found at [Rec04]. Suffice to say, both the forward and backward differences will provide biased approximations to the derivatives, so we will be using the 'Centered Time, Centered Space' (CTCS) scheme, otherwise known as the Crank-Nicolson method.

Before we can apply any approximation method to the problem, however, we must do the discretization step. Equation 3.5 models the whole infinity of space and time, to obtain a sensible solution to it we must pick a finite period and volume as the simulation domains. To obtain an approximation of the true solution in our chosen finite domains we will dissect them into a mesh of a discrete number of points, similarly to the example above.

Our spatial domains are $X, Y$ and $Z$ of length $L_x, L_y$ and $L_z$ accordingly, in

addition to the period of time $T$ with duration $t_{max}$:

$$\begin{aligned} i &\in (0, 1, \ldots, N_x), \quad N_x = |X^*|, \quad \Delta x = \frac{L_x}{N_x} \\ j &\in (0, 1, \ldots, N_y), \quad N_y = |Y^*|, \quad \Delta y = \frac{L_y}{N_y} \\ k &\in (0, 1, \ldots, N_z), \quad N_z = |Z^*|, \quad \Delta z = \frac{L_z}{N_z} \\ t &\in (0, 1, \ldots, D_t), \quad D_t = |T^*|, \quad \Delta t = \frac{t_{max}}{D_t} \end{aligned} \tag{3.7}$$

Following this convention, the continuous function $u(x, y, z, t)$, that tells us the temperature at specified coordinates $(x, y, z)$ in an instant of time $t$, which in our discretization scheme becomes $u(x_i, y_j, z_k, t_m)$, will from now on be written as $u^t_{i,j,k}$.

**Boundary Value Problem**

To really get a unique solution to the problem we must define the behavior of the model on the boundaries of our chosen finite domains. At $t = 0$ we have the initial condition, so for all values of $i, j$ and $k$, we must have a known value of $u^0_{i,j,k}$ in advance.

For the spatial boundaries it is a bit more difficult, as there are a number of conditions, that we may wish to simulate. One has to account for all the values of $u^t_{i,j,k}$, where either $i, j$ or $k$ are -1 or the number of nodes in the discrete mesh[1]. The first is the Dirichlet boundary condition (or sometimes called the first-type boundary condition), that takes the form

$$u^t = f(t) \tag{3.8}$$

where $u^t$ is the value of the solution located somewhere along the boundary of the spatial domain at time step $t$, and $f$ is a function defined by the user, that yields the value at this time step. Effectively the Dirichlet boundary condition maintains the temperature at a value chosen by the user, and may be used to simulate heat sources or sinks.

The second relevant condition is the Neumann (otherwise known as second-type) boundary condition. It specifies what values a derivative (as opposed to the result, for the first-type condition) takes on the boundaries of the domain. A

[1]We're not placing the boundary nodes in the coefficient matrix, hence their indices, but can handle them procedurally and by altering the structure of equations corresponding to internal points near the boundaries.

generalization of this boundary condition will be used to simulate an isolating boundary. The boundary condition

$$\frac{\partial u}{\partial x} = 0 \tag{3.9}$$

means, that the rate of change of $u$ on the boundary, in relation to the spatial variable $x$, is equal to zero - in other words, the isolated boundary will not have any effect on the change of temperature.

We will be using the so-called mixed boundary condition, which is just a combination of the first-type and second-type boundary conditions on different regions of the boundary.

### Crank-Nicolson Method

Starting from equation 3.5 we take a backward difference approximation for the time derivative

$$\frac{\partial u}{\partial t} \approx \frac{u^t_{i,j,k} - u^{t-1}_{i,j,k}}{\Delta t} \tag{3.10}$$

and an average of the second order central difference approximations for the spatial derivatives evaluated at the current and previous time steps.

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{1}{2}\left(\frac{u^t_{i-1,j,k} - 2u^t_{i,j,k} + u^t_{i+1,j,k}}{\Delta x^2} + \frac{u^{t-1}_{i-1,j,k} - 2u^{t-1}_{i,j,k} + u^{t-1}_{i+1,j,k}}{\Delta x^2}\right) \tag{3.11}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{1}{2}\left(\frac{u^t_{i,j-1,k} - 2u^t_{i,j,k} + u^j_{i,j+1,k}}{\Delta y^2} + \frac{u^{t-1}_{i,j-1,k} - 2u^{t-1}_{i,j,k} + u^{t-1}_{i,j+1,k}}{\Delta y^2}\right) \tag{3.12}$$

$$\frac{\partial^2 u}{\partial z^2} \approx \frac{1}{2}\left(\frac{u^t_{i,j,k-1} - 2u^t_{i,j,k} + u^t_{i,j,k+1}}{\Delta z^2} + \frac{u^{t-1}_{i,j,k-1} - 2u^{t-1}_{i,j,k} + u^{t-1}_{i,j,k+1}}{\Delta z^2}\right) \tag{3.13}$$

Replacing the partial derivatives in equation 3.5 with formulae from 3.10,3.11,3.12,3.13 and, knowing that the continuous spatial domain was discretized into a cubic grid, replacing $\Delta y$ and $\Delta z$ with $\Delta x$, we get a recurrence equation

$$\begin{aligned}\frac{u^t_{i,j,k} - u^{t-1}_{i,j,k}}{\Delta t} = \frac{\alpha}{2}\Bigg(&\frac{u^t_{i-1,j,k} - 2u^t_{i,j,k} + u^t_{i+1,j,k}}{\Delta x^2} + \frac{u^{t-1}_{i-1,j,k} - 2u^{t-1}_{i,j,k} + u^{t-1}_{i+1,j,k}}{\Delta x^2} + \\ &\frac{u^t_{i,j-1,k} - 2u^t_{i,j,k} + u^t_{i,j+1,k}}{\Delta x^2} + \frac{u^{t-1}_{i,j-1,k} - 2u^{t-1}_{i,j,k} + u^{t-1}_{i,j+1,k}}{\Delta x^2} + \\ &\frac{u^t_{i,j,k-1} - 2u^t_{i,j,k} + u^t_{i,j,k+1}}{\Delta x^2} + \frac{u^{t-1}_{i,j,k-1} - 2u^{t-1}_{i,j,k} + u^{t-1}_{i,j,k+1}}{\Delta x^2}\Bigg)\end{aligned} \tag{3.14}$$

Since we can safely assume that the value of $u$ at the previous time step is known, and moving all the unknowns in equation 3.14 to the left-hand side, we can approximate the values for the current step by solving a system of linear algebraic equations of the form:

$$\left(1 + \frac{6\mu}{2}\right) u^t_{i,j,k} - \frac{\mu}{2} U^t_{i,j,k} = \left(1 - \frac{6\mu}{2}\right) u^{t-1}_{i,j,k} + \frac{\mu}{2} U^{t-1}_{i,j,k} \tag{3.15}$$

where $U^m_{x,y,z} = u^m_{x-1,y,z} + u^m_{x,y-1,z} + u^m_{x,y,z-1} + u^m_{x+1,y,z} + u^m_{x,y+1,z} + u^m_{x,y,z+1}$ and $\mu = \frac{\alpha \Delta t}{\Delta x^2}$. It would be wise to do one more simplification step by dividing through the equation by $\frac{\mu}{2}$, so multipliers for the $U^m_{x,y,z}$ terms become -1 and 1 on the left- and right-hand sides respectively. This step will make it easier to store the linear system in computer memory and lower the memory footprint when dealing with simulations of heterogeneous media.

## 3.3 Solving Systems of Linear Algebraic Equations

To conduct the simulation through solving a system of linear algebraic equations with the use of matrix and vector operations, we express the SLAE based on equation 3.15 as matrix multiplication:

$$A v^t = v^{t-1} \tag{3.16}$$

where $A$ is a matrix, consisting of the coefficients of the system, $v^{t-1}$ is a known vector, consisting of terms from the right-hand side of equation 3.15, and $v^t$ is the solution vector, made up of the unknowns of the system.

According to our discretization scheme, the number of unknowns will be $N = n^3$, where $n$ is the simulation resolution (for simplicity's sake we will consider the case of a cube with $n = N_x = N_y = N_z$). The dimension of the matrix $A$ will then be $N \times N$, which can get very large for high resolution simulations. Increasing the resolution not only serves to make the simulation look more visually appealing, but improves the numerical accuracy of the approximation as well.

Luckily, for the chosen test case, the matrix $A$ is sparse and for a one-dimensional problem it is a tridiagonal matrix, but for two- and three-dimensional problems it becomes a banded matrix as shown in figure 3.1. $A$ is also symmetric and positive definite[2] - properties important for the method, that will be applied to solve the linear system. The matrix is similar to an adjacency matrix, in that location of off-diagonal elements represent nodes in the mesh. Specifically, nodes that are adjacent to the node the matrix row in question corresponds to. For a

[2]One definition of positive definiteness is that all of the matrix's eigenvalues are positive.

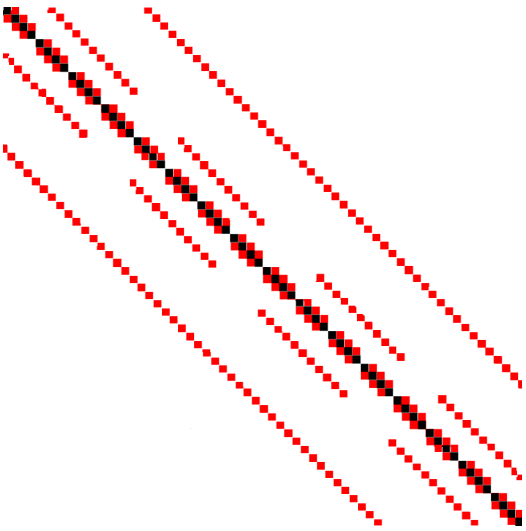

Figure 3.1: Shape of the sparse banded matrix $A$ for a cube with $n = 4$. Black and red cubes signify diagonal and off-diagonal non-zero elements respectively.

cube, each row has at least 4 and at most 7 elements. The exact off-diagonal element count for a cube can be found, as seen from figure 3.1, using the following equation:

$$2n^2\left(n-1\right)+2n\left(n^2-1\right)+2\left(n^3-1\right) \tag{3.17}$$

According to equation 3.15, the off-diagonal elements are $\frac{\mu}{2}$ and for a non-heterogeneous material are all equal. The diagonal values are $1+\frac{\lambda\mu}{2}$, note $\frac{\lambda}{2}$ as a variable multiplier for $\mu$ instead of $\frac{6}{2}$, as given in equation 3.15 - this is due to different behavior of the equation on the boundaries of the domain. Recall equation 3.9 for our second-type boundary condition, that states, that the value of the derivative on the boundaries is equal to 0. When constructing equation 3.14 for the boundary regions, some of the terms (depending on which point's spatial derivatives are 0) in it will not be present, producing a multiplier in it's simplified form, that is less than in the general case of all adjacent points of $u^t_{i,j,k}$ affecting the heat flux. For example: when constructing equation 3.14 for $u^1_{0,0,0}$ and having a second-type boundary condition at $u_{-1,0,0}$ and first-type boundary conditions at $u_{0,-1,0}$ and $u_{0,0,-1}$ results in

$$\begin{aligned}\frac{u^1_{0,0,0}-u^0_{0,0,0}}{\Delta t}=\frac{\alpha}{2}\Bigg(&\frac{u^1_{0,0,0}+u^1_{1,0,0}}{\Delta x^2}+\frac{u^0_{0,0,0}+u^0_{1,0,0}}{\Delta x^2}+\\&\frac{u^1_{0,-1,0}-2u^1_{0,0,0}+u^1_{0,1,0}}{\Delta x^2}+\frac{u^0_{0,-1,0}-2u^0_{0,0,0}+u^0_{0,1,0}}{\Delta x^2}+\\&\frac{u^1_{0,0,-1}-2u^1_{0,0,0}+u^1_{0,0,1}}{\Delta x^2}+\frac{u^0_{0,0,-1}-2u^0_{0,0,0}+u^0_{0,0,1}}{\Delta x^2}\Bigg)\end{aligned} \tag{3.18}$$

Note, that in equation 3.18 terms $u^1_{-1,0,0}$,$-u^1_{0,0,0}$,$u^0_{-1,0,0}$ and $-u^0_{0,0,0}$ are missing, since from equation 3.9 we know, that change in $u$ is independent of the corresponding spatial variable, meaning $u^1_{-1,0,0}$ and $u^1_{0,0,0}$ as well as $u^0_{-1,0,0}$ and $u^0_{0,0,0}$ are equal and cancel each other out. As for the second-order boundary conditions, variables $u^1_{0,0,-1}$,$u^0_{0,0,-1}$,$u^1_{0,-1,0}$ and $u^0_{0,-1,0}$ take on values given by $f(t)$ as per equation 3.8, since regardless of heat diffusion occurring in these directions, the Dirichlet condition maintains the temperature on the boundaries.

### 3.3.1 Conjugate Gradient Method

The conjugate gradient (CG) method [She94] is an iterative algorithm for solving systems of linear equations, whose coefficient matrix is real, symmetric and positive definite (as is the case with coefficient matrix $A$). The general idea of CG is to perform an initial inaccurate guess of the solution vector and then minimize the difference between the approximate and the actual solution at every subsequent iteration.

**Input**: vector $b$ and coefficient matrix $A$
**Output**: approximation for solution vector $x$

set error tolerance threshold and maximum number of iterations
initialize vectors $x$,$z$,$r$,$p$,$q$ as null vectors

$r = b - Ax$ and calculate current error from residual vector $r$

**while** *tolerance threshold not reached and iterations limit not exceeded* **do**
- $z = r$
- $\sigma_{old} = \sigma$
- $\sigma = z \cdot r$
- $p = z + \frac{\sigma}{\sigma_{old}} p$
- $q = Ap$
- $\gamma = \frac{u}{p \cdot q}$
- $x \mathrel{+}= \gamma p$
- $r \mathrel{-}= \gamma q$
- calculate current error from residual vector $r$

**end**

return $x$

**Algorithm 1:** Conjugate gradient method of approximating solution to $Ax = b$.

Algorithm 1 shows the serial iterative conjugate gradient algorithm in it's most commonly used form. While this algorithm is fairly optimized and includes only a single matrix vector multiplication operation, which is it's computational bottleneck, we still need to hold several vectors (of size $n$) in memory: the solution

| | Memory requirements (MB) | | |
|---|---|---|---|
| n | CG | Sparse matrix | Total |
| 8 | 0.02 | 0.02 | 0.04 |
| 16 | 0.19 | 0.12 | 0.31 |
| 32 | 1.5 | 0.99 | 2.49 |
| 64 | 12 | 7.97 | 19.97 |
| 128 | 96 | 63.87 | 159.87 |
| 256 | 768 | 511.5 | 1279.5 |
| 512 | 6144 | 4094 | 10238 |
| 1024 | 49152 | 32759.99 | 81911.99 |

Table 3.1: Memory requirements for simulation of a cube with resolution $n$.

vector $x$, an auxiliary vector $z$, residual vectors $r$ and $q$, and search vector $p$. Explaining how CG works is not in the scope of this work, for that refer to [She94].

**Parallel CG Algorithm**

Since we are limited by computational power and memory capacity of any single machine, performing faster and more accurate simulations requires parallelization. To distribute the memory footprint of the simulation, we split both the vectors used by the CG algorithm and the sparse coefficient matrix among the computing nodes. This also ensures, that the computation is done on smaller chunks of the problem concurrently, effectively speeding up the process. The approach is called 'Single Program, Multiple Data' (SPMD) and typically involves splitting a problem of size $N$ into $p$ processes, meaning all of the data structures have to be split evenly among computing nodes.

Table 3.1 shows the growth of memory requirements as the resolution of the simulation grows. The row labeled 'CG' displays the memory for six double precision floating-point number vectors, utilized by the algorithm, and the 'Sparse matrix' one for a double precision floating-point number vector, holding the diagonal elements, and the integers used to keep off-diagonal element indices (counted using equation 3.17). It has to be noted, that, while the numbers displayed in table 3.1 may seem large, CG is one of the more efficient algorithms memory-wise, as the coefficient matrix remains sparse throughout the computation, as opposed to, for example, matrix inversion based solution methods, which convert the problem to a dense one[3]. Considering the structure of the sparse matrix, the algorithm can be implemented without storing the matrix in memory at all, however such approach is more computationally intensive.

[3] $1073741824 \times 1073741824$ dense matrix of double precision floating-point numbers, roughly 8 exabytes for a $n = 1024$ cube.

Looking at algorithm 1 several issues can be identified, that rule it out from belonging to the embarrassingly parallel class of algorithms. The most evident is matrix vector multiplication, recall that for equation

$$Ab = c \tag{3.19}$$

elements $c_j$ of vector $c$ with length $N$ are found as

$$c_i = \sum_{j=0}^{N-1} A_{ij} b_j \tag{3.20}$$

This would imply, that computing matrix vector multiplication partially still requires vector $b$ in it's entirety. Consider the shape of the matrix from figure 3.1 - we know, that on row $i$ there can be elements at columns $i+1, i-1, i+n, i-n, i+n^2$ and $i-n^2$, meaning that one has to transmit at most $n^2$ (or $n$ for a two-dimensional case) elements from the previous and next vector chunks to perform the operation in parallel. This places a restriction on the number of processes, that we may use for this operation, to $n-1$. While smaller problems do not benefit much from parallelization regardless of this restriction, larger ones may still be sufficiently parallelized.

The second issue is less obvious - a local dot product can still be computed from partial vectors, however, for the algorithm to maintain sequential consistency, each process needs the full dot product, computed at two distinct steps during an iteration, placing the need to transmit each local dot product to every other process before computation can continue.

**Input**: partial vector $b$ and coefficient matrix $A$
**Output**: part of the approximation for solution vector $x$

set error tolerance threshold and maximum number of iterations
initialize partial vectors $x$,$z$,$r$,$p$,$q$ as null vectors

$r = b - Ax$ and calculate current error from residual vector $r$

**while** *tolerance threshold not reached and iterations limit not exceeded* **do**

$z = r$
$\sigma_{old} = \sigma$
$dot_{local} = z \cdot r$

broadcast $dot_{local}$ as well as locally calculated error
barrier()

$\sigma = \sum dot_{local}$
$p = z + \frac{\sigma}{\sigma_{old}} p$

send $n^2$ elements of $p$ to previous and next neighbors
barrier()

$q = Ap$ (using neighbor's $p$ when necessary)
$dot_{local} = p \cdot q$

broadcast $dot_{local}$
barrier()

$\gamma = \frac{u}{\sum dot_{local}}$
$x \mathrel{+}= \gamma p$
$r \mathrel{-}= \gamma q$

calculate current error from residual vector $r$

**end**

return $x$

**Algorithm 2:** BSP conjugate gradient method of approximating solution to $Ax = b$.

Algorithm 2 shows CG under the BSP model as having three barriers under each iteration. The first and last of the three barrier synchronization points are caused by computation of a dot product. While cheap to compute, this operation requires two full vectors, which are distributed and are worked on locally, so to reach a consistent state, each local result is broadcast to every other process. The second barrier is there to synchronize data between neighboring processes, which was established to be needed for matrix vector multiplication.

# Chapter 4

# Experiment Results

The heat diffusion simulation test case was run using three different communication libraries - MPJ Express 0.38 [SCB09], mpiJava 1.2 and the BSPlib interface with BSPonMPI as the native library. Both mpiJava and BSPonMPI are used with mpich2 1.4.1p1 [mpib] for MPI communications.

Table 4.1 presents results of running the application on a cluster of 4 Amazon instances of type m1.xlarge (Standard Extra Large Instance), each with 15 GB of memory and 8 EC2 Compute Units (4 virtual cores). Running times were measured for problems of increasing size to see how larger amounts of data transferred during synchronization would affect parallel slowdown, and at the same time, whether the increasing amount of computation in relation to communication would improve scalability.

| | | $n$=100 | | | | $n$=200 | |
|---|---|---|---|---|---|---|---|
| $p$ | MPJ Express | mpiJava | BSPonMPI | $p$ | MPJ Express | mpiJava | BSPonMPI |
| 1 | 3.34 | 3.34 | 3.34 | 1 | 31.01 | 31.01 | 31.01 |
| 2 | 1.89 | 1.87 | 1.89 | 2 | 16.80 | 16.27 | 16.02 |
| 4 | 1.13 | 1.17 | 1.24 | 4 | 9.57 | 8.93 | 8.91 |
| 8 | 0.88 | 0.91 | 0.78 | 8 | 6.06 | 5.50 | 5.88 |
| 16 | 1.22 | 1.22 | 0.96 | 16 | 5.05 | 4.13 | 4.68 |
| | | $n$=300 | | | | $n$=400 | |
| $p$ | MPJ Express | mpiJava | BSPonMPI | $p$ | MPJ Express | mpiJava | BSPonMPI |
| 1 | 139.57 | 139.57 | 139.57 | 1 | 411.47 | 411.47 | 411.47 |
| 2 | 72.15 | 83.86 | 70.39 | 2 | 216.12 | 285.21 | 212.55 |
| 4 | 41.13 | 55.31 | 40.15 | 4 | 127.80 | 194.29 | 121.30 |
| 8 | 26.66 | 34.44 | 26.12 | 8 | 80.13 | 122.92 | 76.34 |
| 16 | 20.91 | 19.72 | 19.78 | 16 | 54.36 | 75.85 | 51.28 |

Table 4.1: Running time (in seconds) of simulation on $p$ processes.

The test case for the simulation was heat diffusion in a cube of solid material with thermal diffusivity equal to that of air ($1.9 \times 10^{-5}$). The side of this cube was 1.75 meters and the simulated period lasted 30 seconds. The choice of parameters was governed by the need to keep the CFL (Courant-Friedrichs-Lewy) condition number [cfl] low, a requirement for numerical accuracy of the Crank-Nicolson method.

The amount of CG iterations needed for achieving an error margin[1] of $10^{-6}$ for the SLAE solution ranged from 2 iterations per one step of the simulation for the smallest tested problem size ($n = 100$) to 8 for the largest ($n = 400$).

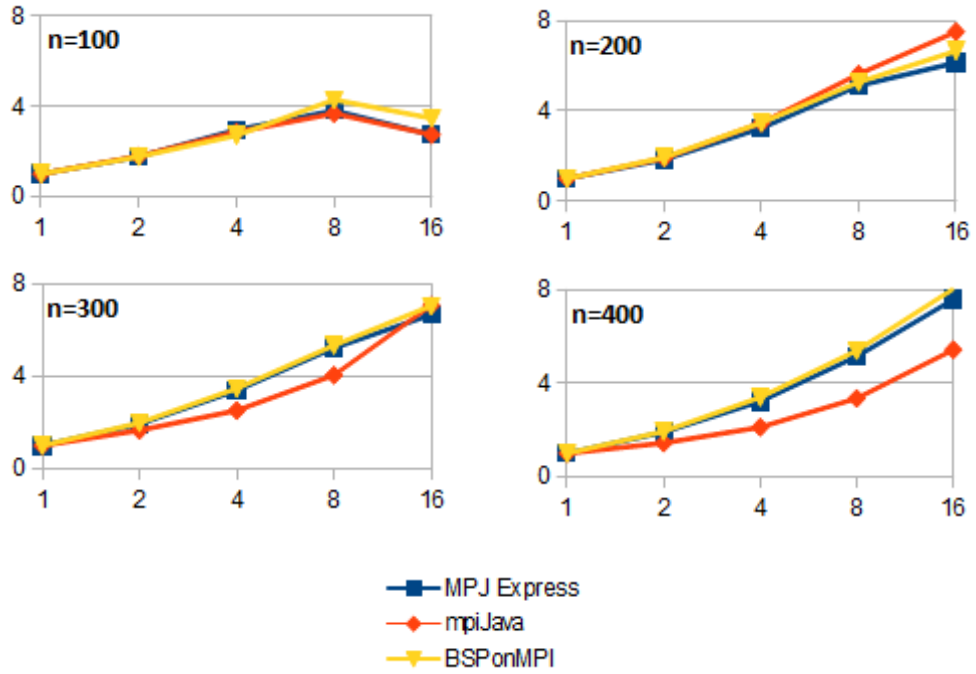


Figure 4.1: Parallel speedup of the simulations

The simulation was split into 10 steps, each step simulating 3 seconds of heat diffusion. The simulation was kept relatively short, in terms of steps, to be able to compare the BSPlib interface with MPJ Express. The way that library handles buffering of data turned out to be problematic for the test application. Fairly large messages are being exchanged between processes at relatively short intervals, but as MPJ Express uses *direct* buffers [Nei], which are intended to be few and long-lived, the continuous allocation of temporary buffers outside the Java heap, for

[1]Counted as the maximum norm ($\|a\|_{\max} = \max\{|a_i|\}$) of the residual vector $r$ used in the CG algorithm.

each communication operation, meant long simulations using MPJ Express would eventually run out of memory.

Figure 4.1 shows the speedup of the program when run with $p$ parallel processes. The best speedup for the given problem sizes, using 16 processes, was only 8[2], which is to be expected from an algorithm with as many synchronization points as parallel CG. Another expected result was the performance of MPJ Express, which is a communication library written purely in Java, so little ovarhead on the side is involved. The surprising part of the results was how well the BSPlib interface performed, both in relation with MPJ Express, performing on par and even pulling ahead most of the time, and compared to mpiJava, which at the lowest level uses the same MPI communications library. As the problem size grew, mpiJava experienced difficulties transmitting larger chunks of data as fast as the other libraries and, eventually, started to lag behind. Most likely this was the side effect of data buffering mechanisms it employs for data transmission operations.

[2]Only embarrassingly parallel algorithms would get the best possible speedup of 16.

# Conclusion

This work presented a BSPlib binding for the Java programming language, which was used to illustrate the suitability of the BSP model for scientific computing. This was demonstrated by means of solving a typical scientific computing problem, that of simulating heat conduction, using one of the most commonly encountered algorithms in scientific computing - the conjugate gradient method. The results of running the simulation with the created BSPlib interface were compared to ones from two alternative parallel programming libraries.

Alternative solutions chosen for comparison were two different MPI implementations - mpiJava and MPJ Express. For the native BSPlib implementation BSPonMPI was used. This choice was interesting due to the fact, that both mpiJava and BSPonMPI use a native MPI implementation for low-level communication operations, but while mpiJava is a straight binding for MPI, BSPonMPI uses MPI to perform barrier synchronization and miscellaneous communication necessary for an BSPlib implementation. The comparison between two would show, whether the additional intermediary layer in form of the BSPonMPI framework would cause an overhead to the process.

It turned out, that while mpiJava performed well for smaller resolution simulations, as the size of the problem grew the time needed for transmitting synchronization data increased, compared to the other tested libraries, suggesting that some form of data buffering mechanism for send and receive operations may be slowing it down. MPJ Express as a MPI library written purely in Java was expected to outperform the other solutions, but surprisingly the created BSPlib interface backed by BSPonMPI managed to be on par with it for all tests, even performing marginally better most of the time.

It has to be noted, that the BSPlib interface is in very raw state and it's error handling is very basic. The wrapper library native code attempts to detect several abnormal states, that may occur in case of programming errors, and throws Java exceptions if any of those states are detected. However, should the communication network fail or another severe anomaly occurs, it is not guaranteed to fail gracefully. Most likely a parallel application will simply stop execution without any

error messages and it will be up to the underlying native communication framework to clean up the remaining processes. Luckily the Hydra process manager (the default process manager of mpich2 since version 1.3) managed to handle this task successfully if any errors occurred during testing.

As it stands, the created BSPlib interface seems like a good alternative to writing MPI programs, due to BSPlib's much easier API and the ability to effortlessly write deadlock free parallel applications, while maintaining performance level on par with one achieved with the use of MPI. The BSP model has been shown to allow one to implement algorithms, that the currently ubiquitous parallel computing model in the *cloud* - MapReduce, is not capable of supporting. However, providing fault tolerance remains a topic for future work, as placing long-running tasks in the *cloud* environment with the current solution is a gamble.

## BSPlib Java liides paralleelsete teadusarvutuse rakendustele

## Bakalauruse töö (6 EAP)

## Ilja Kromonov

## Resümee

Töö eesmärgiks oli *native*[3] BSPlib implementatsioonile Java liidese loomine, mis võimaldaks BSP mudelil põhinevaid programme kirjutada Java programmeerimiskeeles, ning loodud prototüübi võrdlemine olemasolevate paralleelprogrammeerimiseks mõeldud lahendustega. Sellega üritati näidata BSP mudeli sobivust teadusarvutuste jaoks, kus perspektiiviks on Java platvormile ehitatud ja BSP mudelil põhineva raamistiku kasutamine teadusarvutuste läbi viimiseks pilvekeskkonnas.

BSP mudeli sobivuse demonstreerimiseks loodi Java programm, mis lahendas arvutuste ja paralleliseerimise mõttes keerulist ülesannet. Kasutusjuhtumiks oli valitud tüüpiline füüsika valdkonnast pärit probleem - soojuse võrrandi lahendamine soojusjuhtivuse simulatsiooni läbi viimiseks, mille arvutuslikuks kitsaskohaks on suurte lineaarvõrrandisüsteemide ligikaudsete lahendite välja arvutamine. Lineaarvõrrandisüsteemide lahendamise algoritmiks oli valitud *conjugate gradient* (CG) meetod. Seda algoritmi pole võimalik efektiivselt teostada MapReduce mudeli abil, mis on hetkel pilvekeskkonnas paralleelarvutuste raamistike ehitamisel kõige laiemalt kasutatav lahendus.

Antud kasutusjuhtumi abil võrreldi BSPlib Java liidese ja erinevate Message Passing Interface (MPI) teateedastusteekide kasutamisel saavutatavat kiiruse kasvu ning määrati kas BSPlib võib põhjustada paralleelse CG algoritmi aeglustumist võrreldes teiste lahendustega.

Katsetes kasutati võrdluseks kaht MPI implementatsiooni: mpiJava ja MPJ Express. Esimene nendest on *native* implementatsiooni kasutav lahendus, mille tööprintsiip on sarnane pakutud BSPlib prototüübiga. Teine on täielikult Java platvormil realiseeritud teateedastusteek. Loodud Java liides kasutas BSPlib implementatsioonina BSPonMPI'd, mis oma madalaimal tasemel kasutab sama teateedastusteeki kui mpiJava.

Tulemustest selgitati välja, et loodud BSPlib liides osutub heaks alternatiiviks MPI abil paralleelsete programmide kirjutamisele, kuna võimaldab kirjutada ummikute vaba koodi samas säilitades MPI'ga võrreldavat jõudlust. Seega näidati, et BSP mudelil saab efektiivselt realiseerida teadusarvutuste algoritme, mis ei sobi pilvekeskkonnas üldlevinud MapReduce mudelile.

[3] Kompileeritud protsessori poolt loetavaks baitkoodiks.

# Bibliography


[Apaa] Apache Software Foundation. Hadoop. URL: http://wiki.apache.org/hadoop/.

[Apab] Apache Software Foundation. Hadoop distributed file system. URL: http://hadoop.apache.org/hdfs/.

[Apac] Apache Software Foundation. Hama. URL: http://wiki.apache.org/hama/.

[ATP84] D.A. Anderson, J.C. Tannehill, and R.H. Pletcher. *Computational fluid mechanics and heat transfer*. Series in computational methods in mechanics and thermal sciences. Taylor & Francis, 1984. URL: http://books.google.ca/books?id=Bfk_AQAAIAAJ.

[Bry98] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum and Geoffrey Fox. MPI for Java - Position Document and Draft API Specification. Technical Report JGF-TR-03, Java Grande Forum, November 1998. URL: http://www.hpjava.org/reports/MPIposition/position.ps.

[Car98] Bryan Carpenter. mpiJava: A Java Interface to MPI. In *First UK Workshop on Java for High Performance Network Computing, Europar 98*, 1998.

[cfl] Courant-Friedrichs-Lewy condition. URL: http://www.encyclopediaofmath.org/index.php/Courant%E2%80%93Friedrichs%E2%80%93Lewy_condition.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, 2004.

[ELZ+10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, 2010.

[GsLC01] Yan Gu, Bu sung Lee, and Wentong Cai. JBSP: A BSP Programming Library In Java, 2001.

[HDL97] Jonathan M. D. Hill, Stephen R. Donaldson, and Tim Lanfear. Process migration and fault tolerance of bsplib programs running on networks of workstations. In *In EuroPar'98, LNCS*, pages 80–91. Springer-Verlag, 1997.

[Hil] Jonathan Hill. The Oxford BSP toolset. URL: http://www.bsp-worldwide.org/implmnts/oxtool/.

[JKS11] Pelle Jakovits, Ilja Kromonov, and Satish Narayana Srirama. Monte carlo linear system solver using mapreduce. In *UCC*, pages 293–299, 2011.

[jni] JNI Enhancements Introduced in version 1.2 of the JavaTM 2 SDK. URL: http://docs.oracle.com/javase/1.3/docs/guide/jni/jni-12.html.

[Jon98] Jonathan M.D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel , Thanasis Tsantilas and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947 – 1980, 1998.

[JOP+ ] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL: http://www.scipy.org/.

[MAB+09] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing - "abstract". In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 6–6, 2009.

[mpia] MPI Routines. URL: http://www.mcs.anl.gov/research/projects/mpi/www/www3/.

[mpib] MPICH2 - a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard. URL: http://www.mcs.anl.gov/research/projects/mpich2/.

[Nei] Neil Coffey. Java direct buffers. URL: http://javamex.com/tutorials/io/nio_buffer_direct.shtml.

[Ola03] Olaf Bonorden, Ben Juurlink, Ingo von Otte and Ingo Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187 – 207, 2003.

[Rec04] G. W. Recktenwald. Finite-Difference Approximations to the Heat Equation, 2004. URL: www.f.kth.se/~jjalap/numme/FDheat.pdf.

[SCB09] Aamir Shafi, Bryan Carpenter, and Mark Baker. Nested parallelism for multi-core hpc systems using java. *J. Parallel Distrib. Comput.*, 69(6):532–545, 2009.

[ser] Java Serialization Benchmark project. URL: https://github.com/eishay/jvm-serializers/wiki/.

[She94] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[SJV] Satish Narayana Srirama, Pelle Jakovits, and Eero Vainikko. Adapting scientific computing problems to clouds using mapreduce. *Future Generation Comp. Syst.*, (1):184–192.

[Sui] Wijnand J. Suijlen. BSPonMPI. URL: http://bsponmpi.sourceforge.net/.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[YB11] A. N. Yzelman and Rob H. Bisseling. An object-oriented BSP library for multicore programming. 2011.