Final thesis

Modelica PARallel benchmark suite (MPAR) - a test suite for evaluating the performance of parallel simulations of Modelica models

by

Afshin Hemmati Moghadam

LIU-IDA/LITH-EX-A—11/042—SE

2011-11-11
Final Thesis

Modelica PARallel benchmark suite (MPAR) - a test suite for evaluating the performance of parallel simulations of Modelica models

by

Afshin Hemmati Moghadam

LIU-ID/EX-A—11/042—SE

2011-11-11

Supervisor: Kristian Stavåker
Examiner: Peter Fritzson
Dedicated to my brother’s soul and memory.

May you rest in peace dear Amin.
Abstract

Using the object-oriented, equation-based modeling language Modelica, it is possible to model and simulate computationally intensive models. To reduce the simulation time, a desirable approach is to perform the simulations on parallel multi-core platforms. For this purpose, several works have been carried out so far, the most recent one includes language enhancements with explicit parallel programming language constructs in the algorithmic parts of the Modelica language. This extension automatically generates parallel simulation code for execution on OpenCL-enabled platforms, and it has been implemented in the open-source OpenModelica environment. However, to ensure that this extension as well as future developments regarding parallel simulations of Modelica models are feasible, performing a systematic benchmarking with respect to a set of appropriate Modelica models is essential, which is the main focus of study in this thesis.

In this thesis a benchmark test suite containing computationally intensive Modelica models which are relevant for parallel simulations is presented. The suite is used in this thesis as a means for evaluating the feasibility and performance measurements of the generated OpenCL code when using the new Modelica language extension. In addition, several considerations and suggestions on how the modeler can efficiently parallelize sequential models to achieve better performance on OpenCL-enabled GPUs and multi-core CPUs are also given.

The measurements have been done for both sequential and parallel implementations of the benchmark suite using the generated code from the OpenModelica compiler on different hardware configurations including single and multi-core CPUs as well as GPUs. The gained results in this thesis show that simulating Modelica models using OpenCL as a target language is very feasible. In addition, it is concluded that for models with large data sizes and great level of parallelism, it is possible to achieve considerable speedup on GPUs compared to single and multi-core CPUs.
I am very thankful to God for giving me a life to live and for giving me strength to cope with all difficulties.

I would like to thank my parents and my brother Iman who have been supporting me in every situation especially during my education here in Sweden. I really appreciate their generosity, patience, and support toward my successes.

I would like to express my deepest gratitude to my examiner Professor Peter Fritzson for guiding me through all the phases of this thesis work. I would also like to thank my supervisor Kristian Staväker and Mohsen Torabzadeh Tari for sharing their time, knowledge, and experience with me regarding technical issues and non-technical issues respectively. I also appreciate the help that Mahder Gebremedhin has given during this thesis work.

Finally, I would like to thank all my teachers, administrative staff, and all my friends here at Linköping University for their help and support.

Afshin Hemmati Moghadam
November 11, 2011
Linköping, Sweden.
# Contents

1 Introduction .................................................. 11
  1.1 Motivation .................................................. 12
  1.2 Methodology ................................................ 12
  1.3 Limitations ................................................ 12
  1.4 Goal ....................................................... 13
  1.5 Thesis Outline ............................................. 13

2 Background .................................................... 14
  2.1 Modeling and Simulation .................................... 14
  2.2 The Modelica Modeling Language ............................. 15
    2.2.1 The OpenModelica Development Environment .......... 16
  2.3 General Purpose Parallel Computing Architectures ............ 17
    2.3.1 OpenCL ............................................... 17
      2.3.1.1 Platform Model .................................. 19
      2.3.1.2 Execution Model ................................ 19
      2.3.1.3 Programming Model ............................... 20
      2.3.1.4 Memory Model ................................... 22

3 The MPAR Benchmark Suite ................................... 23
  3.1 Sequential Implementations ................................. 23
    3.1.1 Matrix-Matrix Multiplication .......................... 23
    3.1.2 Reduction .............................................. 24
    3.1.3 Computation of Eigenvalues ........................... 25
    3.1.4 Heat Conduction ..................................... 31
      3.1.4.1 Stationary Heat Conduction ....................... 31
  3.2 Parallel Implementation ................................... 33
    3.2.1 Matrix-Matrix Multiplication .......................... 33
    3.2.2 Reduction .............................................. 35
    3.2.3 Computation of Eigenvalues ........................... 37
    3.2.4 Stationary Heat Conduction ............................ 41
List of Figures

2.1 A planar pendulum [6]. .................................... 15
2.2 Simulation plot of the planar pendulum model. ........ 17
2.3 The OpenModelica environment structure [5]. .......... 18
2.4 OpenCL platform model [18]. .............................. 19
2.5 OpenCL execution model [18]. ............................. 20
2.6 OpenCL memory model [18]. ............................... 22

3.1 Equidistant grid with boundary conditions. ............. 32
3.2 Parallel reduction of a block in a work-group. .......... 37

4.1 Simulation time plot as function of the parameter M (ma-
trix size M×M), for matrix multiplication model using kernel
function. ....................................................... 48
4.2 Speedup graph as function of the parameter M (matrix size
M×M), for matrix multiplication model using kernel function. 49
4.3 Simulation time plot as function of the parameter M (matrix
size M×M), for matrix multiplication model using parfor. ... 50
4.4 Speedup graph as function of the parameter M (matrix size
M×M), for matrix multiplication model using parfor. ........ 50
4.5 Simulation time plot as function of the parameter N (array
size) for reduction model. .................................. 51
4.6 Speedup graph as function of the parameter N (array size)
for reduction model. ........................................ 51
4.7 Simulation time plot as function of the parameter N (array
size) for eigenvalue model. ................................. 52
4.8 Speedup graph as function of the parameter N (array size)
for eigenvalue model. ........................................ 53
4.9 Simulation time plot as function of the parameter N (matrix
size N×N) for stationary heat conduction model using parfor-
loop. ............................................................ 54
4.10 Speedup graph as function of the parameter N (matrix size
N×N) for stationary heat conduction model using parfor-loop. 54
4.11 Simulation time plot function of the parameter $N$ (matrix size $N \times N$) for stationary heat conduction model using kernel function. ................................................................. 55

4.12 Speedup graph function of the parameter $N$ (matrix size $N \times N$) for stationary heat conduction model using kernel function. 55
List of Tables

3.1 The MPAR benchmark test suite. ............................... 23
3.2 Reduction operations [25]. ................................. 25

A.1 Matrix-matrix multiplication simulation times for the Intel Xeon E5520 CPU (serial). ............................... 60
A.2 Matrix-matrix multiplication simulation times for the Intel Xeon E5520 CPU (parallel using parfor). ..................... 60
A.3 Matrix-matrix multiplication simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel using parfor). ....... 60
A.4 Matrix-matrix multiplication simulation times for the Intel Xeon E5520 CPU (parallel using kernel function). .......... 61
A.5 Matrix-matrix multiplication simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel using kernel function). 61
A.6 Reduction simulation times for the Intel Xeon E5520 CPU (serial). ............................................................ 61
A.7 Reduction simulation times for the Intel Xeon E5520 CPU (parallel). ............................................................. 61
A.8 Reduction simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel). ................................................ 62
A.9 Computation of eigenvalues simulation times for the Intel Xeon E5520 CPU (serial). ........................................... 62
A.10 Computation of eigenvalues simulation times for the Intel Xeon E5520 CPU (parallel). ...................................... 62
A.11 Computation of eigenvalues simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel). ......................... 63
A.12 Stationary heat conduction simulation times for the Intel Xeon E5520 CPU (serial). ............................................. 63
A.13 Stationary heat conduction simulation times for the Intel Xeon E5520 CPU (parallel using parfor). ......................... 63
A.14 Stationary heat conduction simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel using parfor). ........... 63
A.15 Stationary heat conduction simulation times for the Intel Xeon E5520 CPU (parallel using kernel function). .............. 64
A.16 Stationary heat conduction simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel using kernel function). . . 64
# List of Listings

2.1 Modelica code for the planar pendulum model [6]. .................. 16
2.2 Modelica sequential code for vector multiplication. ............. 21
2.3 Data-parallelism in OpenCL for vector multiplication. .......... 21
3.1 Modelica function for matrix multiplication. ...................... 24
3.2 Modelica function for summation of an array. .................... 24
3.3 Modelica function for product of a matrix. ....................... 25
3.4 Modelica function for computing the Gerschgorin interval. ..... 27
3.5 Modelica function for calculating the number of eigenvalues
within an interval. ................................................. 28
3.6 Modelica function for calculating the eigenvalues of a tridiag-
onal symmetric matrix. ............................................ 29
3.7 Stationary heat conduction. ...................................... 32
3.8 Parallel implementation of matrix-matrix multiplication us-
ing parfor-loop. .................................................. 34
3.9 Parallel implementation of matrix-matrix multiplication us-
ing kernel function. .............................................. 35
3.10 Parallel implementation of reduction using kernel function. ... 36
3.11 Kernel function that calculates the number of eigenvalues
within an interval. .................................................. 38
3.12 Kernel function that recalculates and divides the eigenvalue
intervals. .......................................................... 38
3.13 Parallel function that calculates the number of eigenvalues
less than x. ........................................................ 40
3.14 Parallel implementation of stationary heat conduction using
parfor-loop. ......................................................... 41
3.15 Parallel implementation of stationary heat conduction using
kernel function. ..................................................... 44
4.1 The main simulation function of the OpenModelica runtime
system. ............................................................. 46
B.1 Matrix-matrix multiplication. .................................... 65
B.2 Reduction. ....................................................... 67
B.3 Computation of eigenvalues. .................................... 68
B.4 Stationary heat conduction. ...................................... 73
B.5 Matrix-matrix multiplication. .................................... 74
B.6 Reduction .................................................. 76
B.7 Computation of eigenvalue. ............................... 78
B.8 Stationary heat conduction. ............................. 85
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CU</td>
<td>Compute Unit</td>
</tr>
<tr>
<td>DAE</td>
<td>Differential Algebraic Equation</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General-Purpose computing on Graphics Processing Units</td>
</tr>
<tr>
<td>OSMC</td>
<td>Open Source Modelica Consortium</td>
</tr>
<tr>
<td>OMC</td>
<td>OpenModelica Compiler</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>PELAB</td>
<td>Programing Environment Laboratory</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SPMD</td>
<td>Single Program Multiple Data</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Recently, computer supported modeling and simulation have become a practical tool in almost all areas of engineering and scientific computing including safety engineering [1], flight engineering [2], and automotive engineering [3]. Consequently, more complex models and designs are being constructed and simulations can be very computationally heavy. Due to this increase in complexity, it is essential to develop more powerful modeling and simulation tools as well as improving the performance of existing tools. But, how to ensure that the new implemented or enhanced tool is feasible enough to be used for simulating computationally intensive models? To address this problem, it is required to evaluate the performance characteristic of the new tool by simulating some models which represent high-performance and heavy computations.

In this thesis the focus of study is on performance evaluation of the generated code and the runtime system of an open-source Modelica [6] modeling and simulation environment called OpenModelica [4]. This is done by performing a systematic benchmarking of some computationally heavy Modelica models on both single-core, and multi-core platforms including the new 2-teraflop Nvidia 2050 Graphics Processing Unit (GPU) [7].

Motivation for why benchmarking of OpenModelica simulations is an interesting problem as well as a closer look at benchmarking will be given in this chapter. Some motivations behind the choice of models which are used as benchmarks will be presented as well. In addition, limitations and constraints will be discussed. Finally the main goal of this thesis will be presented.
1.1 Motivation

As a result of increases in the applicability of simulations, more powerful modeling and simulation tools are required. The OpenModelica development environment, as will be discussed in Chapter 2, is known as a practical and useful tool when it comes to modeling and simulation of Modelica models. Thus, it is essential to keep its applicability by constantly improving it and making it more powerful. Generating parallel simulation code which aims to reduce the computation time by performing simulations more efficient on parallel and multi-core platforms is one of the approaches regarding OpenModelica improvements. For this purpose several works have been carried out so far, for instance, automatic parallelization of Modelica models [8, 9, 10, 11, 12, 13], extending the language with explicit parallel programming constructs [14], and coarse-grained explicit parallelization using components [15]. However, evaluating the performance and feasibility of these extensions is still a challenge that needs to be addressed, which motivates the work done in this thesis.

1.2 Methodology

One of the most appropriate and standard ways of doing performance evaluation is known as benchmarking. In computing, “benchmarking is the process of running certain computer program or operations for the purpose of assessing relative performance. This is done mostly by running some standard trials and tests” [16].

In order to perform the benchmarking a benchmark test suite named Modelica PARallel benchmark suite (MPAR) has been developed and it is presented in detail in Chapter 3. The MPAR benchmark suite contains a set of suitable algorithmic Modelica models that not only are computationally intensive, but also are relevant to be simulated and measured on parallel platforms. Time measurements are performed for both sequential and parallel Modelica versions of these models on different hardware configuration including single and multi-core CPUs as well as GPUs. This benchmark suite can be used to evaluate the performance of current and future Modelica tool implementations when simulating computationally intensive models.

1.3 Limitations

Since this thesis mainly focuses on evaluation of the new Modelica language extensions presented in [17], therefore the MPAR benchmark suite as well as other work done in this study are restricted in the following way.
The models in the benchmark suite only cover algorithmic parts of the Modelica language.

Parallelism is explicitly stated in the models by using new explicit parallel language constructs such as parallel variables (parglobal, parlocal), parallel function, kernel function (parkernel), and parallel for-loop (parfor).

Only simulation time is measured, compile time is not considered.

The number of models added to the benchmark suite were limited by the 20 weeks time constraint.

1.4 Goal

The goal of this thesis is to construct a suitable benchmark test suite and perform a systematic benchmarking of OpenModelica generated code with both single-core and multi-core OpenCL-enabled platforms. To achieve this goal the following major tasks need to be accomplished. The first task is to collect, construct and implement a set of appropriate Modelica models. These models should address high performance characteristics, and also be relevant for running on parallel platforms. The second task is to measure the simulation time of both sequential and parallel Modelica implementation of these models with OpenModelica generated code on target platforms. And finally, based on the measurements results try to improve the parallel code generator prototype for the explicit parallel language constructs, as well as other part of the OpenModelica Compiler, so that reasonable speed up can be achieved.

1.5 Thesis Outline

The reminder of this thesis is organized as follows. Chapter 2 gives an overview of some theoretical background knowledge regarding the process and purpose of modeling and simulation, the Modelica language as well as the OpenModelica environment, and the OpenCL software framework. In Chapter 3 the implemented Modelica benchmark test suite is presented, and then the performance measurements of this test suite are given in Chapter 4. Finally, in Chapter 5 concluding remarks are provided and future work is discussed.
Chapter 2

Background

This chapter outlines the theoretical background knowledge related to the work in this thesis. The first section provides an overview of the basic concepts regarding the process and purpose of modeling and simulation. The second section will give a brief introduction to the Modelica modeling and simulation language [6] as well as the OpenModelica simulation environment [4]. The Open Computing Language (OpenCL) [18, 19] software framework is then covered in section three, since a good understanding of this framework will be necessary for modelers to exploit parallelism in their models appropriately when using the new language extensions.

2.1 Modeling and Simulation

According to [6, 20], simulation is the process of performing an experiment on a mathematical or a symbolic model of a specific system in order to describe and study its behavior. To clarify this definition it is required to define what is meant by the terms system, experiment, and model. A system is defined as an organized structure composed from a set of interconnected and correlated objects [6]. To be able to study the properties of a system, it is necessary to exercise the system via different inputs and observe the output results. This process which enables us to extract required information from the system by examining it with various input conditions is called an experiment [6]. However, it is not always possible to perform an experiment on a real system for some reasons. First, it may be too expensive and dangerous to do an experiment. For instance, investigating accuracy, correctness, and safety of a rocket by launching it can be a very expensive and dangerous method. Second, input and output variables may not be accessible or even observable in the real system. Third, system may not be completely ready or simply may not yet exist at the time of experiments. Thus, to overcome these shortcomings, it is necessary to construct an appropriate model of the real system, then perform various experiments on the model instead [6]. As
2.2 The Modelica Modeling Language

The Modelica modeling language is known as one of the most powerful and practical languages in the areas of modeling and simulation. It is an object-oriented equation-based language, which is being developed by the Modelica Association [21] toward modeling and simulation of physical systems with high complexity [6]. The approach in Modelica is different from other object-oriented programming languages such as Java and C++. The approach is declarative rather than imperative, since the emphasize in Modelica is more on structured mathematical modeling. Consequently, Modelica models and dynamic system properties are explicitly stated in a declarative way by using mathematical equations and descriptions [6].

As an example, a planar pendulum (Figure 2.1) is represented by a mathematical Modelica model in Listing 2.1. Also, a plot of a four seconds simulation of this model is depicted in Figure 2.2, where the values of $x$ are given on the y-axis and the corresponding time values on the x-axis. The behavior of this model is represented by the following differential and algebraic equations 2.1. For more details about this model, the reader is referred to [6].
\[ m \ddot{x} = -\frac{x}{L} F \]
\[ m \ddot{y} = -\frac{y}{L} F - mg \]
\[ \dot{x} = v_x \]
\[ \dot{y} = v_y \]
\[ x^2 + y^2 = L^2 \]

Listing 2.1: Modelica code for the planar pendulum model [6].

```
model Pendulum "Planar Pendulum"
  constant Real PI = 3.141592653589793;
  parameter Real m = 1, g = 9.81, L = 0.5;
  Real F;
  Real x(start=0.5), y,
  Real vx, vy;
  equation
    m*der(vx) = -(x/L)*F;
    m*der(vy) = -(y/L)*F - m*g;
    der(x) = vx,
    der(y) = vy;
    x^2 + y^2 = L^2;
end Pendulum;
```

Modelica models can for instance be defined by using keywords `model`. Variables \( m, g, L, \) and \( PI \) are constant variables which have been declared as constant by keyword `parameter` and `constant`. A parameter is a constant with the additional characteristic that its value can be changed between simulation runs. Variable \( x, y, v_x, \) and \( v_y \) are dynamic variables and their values can be changed during the simulation. In Modelica, all variables have a start default value equal to zero at the time when simulation starts. But, it is also possible to set a different start value, as in this model \( x \) is set to 0.5 by using the modifier (start=0.5). Moreover, time derivative variables such as \( \dot{x} \) and \( \dot{y} \) in this model are represented in Modelica as `der(x)` and `der(y)` respectively. More information about the Modelica language can be found in [6].

2.2.1 The OpenModelica Development Environment

OpenModelica is an open-source Modelica development environment which was initially developed at the Programming Environment Laboratory (PELAB) at Linköping University. It is now supported by the Open Source Modelica Consortium (OSMC) [22], and the main goal of this project is to develop an environment for creating and simulating Modelica models for research,
teaching, and industrial purposes [4]. Figure 2.3 illustrates different inter-connected subsystems of OpenModelica, and among these subsystems the OpenModelica Compiler (OMC) is the interesting one for our work.

2.3 General Purpose Parallel Computing Architectures

Recently, GPUs have been growing in popularity and they have become more versatile due to the need for accelerating the performance of many different kinds of computationally intensive scientific and engineering applications. The highly parallel structure of these processors have been providing an efficient way for high-performance and parallel computing. However, to be able to effectively utilize the power of these massively parallel architectures, it is required to have appropriate frameworks and programming languages to exploit parallelism in our applications. For this purpose, some software frameworks such as Open Computing Language (OpenCL) [18, 19] and Compute Unified Device Architecture (CUDA) [23] have been developed. They provide a defined API for doing general purpose parallel programming across heterogeneous platforms. As mentioned earlier, since the target platform for parallel simulation in this thesis is OpenCL, therefore in the following section only OpenCL is described. However, for more detailed descriptions of both of these two frameworks, it is recommended to read [18] and [23].

2.3.1 OpenCL

OpenCL is a standard framework suitable for leveraging multi-core CPUs, GPUs, and other modern processors such as DSPs or IBM Cell/B.E [19]. The key goal of OpenCL is to provide a way to accelerate parallel compu-
Figure 2.3: The OpenModelica environment structure [5].
tations and improve the performance of computationally intensive applications. By using the OpenCL framework, it is possible to develop applications that not only have high performance, but also are portable across different platforms and devices. According to [18], the architecture of OpenCL can be described as a hierarchy of models including a platform model, an execution model, a memory model, and a programming model.

2.3.1.1 Platform Model

As illustrated in Figure 2.4, the platform model of OpenCL consists of a Host, OpenCL Compute Devices, Compute Units (CUs), and Processing Elements (PEs). A host is any CPU or processor on a computer that runs the standard operating system and the applications [18]. The host is connected to one or more OpenCL devices, where in this case, an OpenCL device is something such as a GPU, a DSP, or even a multi-core processor which provides processing power for OpenCL. Each device is further divided into one or more CUs. A CU also includes one or more PEs which can execute codes as Single Instruction Multiple Data (SIMD) units, or as Single Program Multiple Data (SPMD) units [18].

2.3.1.2 Execution Model

The OpenCL execution model mainly deals with the way kernel functions are submitted and executed by host program on OpenCL devices [18]. A host program is the application source code where kernel functions, auxiliary functions and other properties are declared, and it is executed on host. A kernel function is basically similar to a C function which mainly performs a specific task. However, it is executed on OpenCL devices and is declared using the _kernel qualifier. Kernel function is the key element in OpenCL
Figure 2.5: OpenCL execution model [18].

As shown in Figure 2.5, for execution of an OpenCL application three main steps including code compilation, data creation, and commands execution are involved. When the source code is compiled a context consisting of available devices is created, then program objects containing the implementation of kernel functions are built. Accordingly, kernel objects, memory objects, and command queue objects will be created. Command queue is a data structure used to enqueue and schedule commands for execution on OpenCL devices. These includes kernel execution commands, synchronization commands, and memory operation commands, which can be executed in order or out of order depending on the structure of the command queue.

2.3.1.3 Programming Model

Data parallel and task parallel programming model are the two main programming models supported by the OpenCL execution model [18]. Moreover,
OpenCL allows to have task parallelism and data parallelism working together to fully utilize the parallelism in our applications [18]. In data parallel programming model, the data domain, i.e., a vector will be distributed across available parallel computing nodes and mapped to work-items, so that the same task (defined by the kernel function) can be performed on each piece of distributed data in parallel. However, in the task parallel programming model, independent tasks will be distributed across available parallel computing nodes, so that they can be executed as a single work-item simultaneously on the same or different data domain. Data parallel programming in OpenCL can be specified implicitly or explicitly. In the implicit way the total number of work-items which should be executed in parallel is specified by the programmer, and the division into work-groups will be done by the OpenCL automatically [18]. However, in explicit way both the total number of work-items and their division into work-groups are specified by the programmer [18].

Assume we want to express data-parallelism using $n$ work-items in a one-dimensional computation domain such as an array of size $n$. To do so, we can implement a kernel function (Listing 2.3) for the sequential code in Listing 2.2. In this case, each element of the array will be mapped to a work-item, so that every instance of this kernel will be executed in parallel over $n$ work-items to multiply each element of the array $A$ with the corresponding element in the array $B$. Each work-item is accessed by its own global id using the OpenCL built-in function `get_global_id()`.

**Listing 2.2:** Modelica sequential code for vector multiplication.

```modelica
function scalar_mul
    input Integer n;
    input Real a[n];
    input Real b[n];
    output Real result[n];
algorithim
    for i in 1:n loop
        result[i] := a[i] * b[i];
    end for;
end scalar_mul;
```

**Listing 2.3:** Data-parallelism in OpenCL for vector multiplication.

```c
_kernel void scalar_mul(global const float *a,
global const float *b,
global float *result)
{
    int globalThreadId = get_global_id(0);
    result[globalThreadId] = a[globalThreadId] * b[globalThreadId];
}
```
2.3.1.4 Memory Model

As illustrated in Figure 2.6, the OpenCL memory model consists of five distinct memory regions including host memory, global memory, constant memory, local memory, and private memory. A compute device has a global memory as well as a constant memory. The global memory is accessible for read/write by all work-items within all work-groups. The constant memory is a region of the global memory which remains constant and is accessed by the host for allocating and initializing memory objects required during the execution of the kernels. Each work-item has a private memory which can be accessed by that work-item, so other work-items cannot access each other’s private memory. Each work-group also has a local memory and this is shared within work-items in that work-group. It is important to note that the memory management and synchronization in OpenCL is explicit, which means that the programmer must explicitly define when and how memory regions should be accessed. Also the programmer must explicitly manage copying data to memory objects from host to devices, and vice versa.

Figure 2.6: OpenCL memory model [18].
Chapter 3

The MPAR Benchmark Suite

In this chapter the implemented benchmark test suite called MPAR will be presented. The MPAR benchmark suite contains both sequential and parallel Modelica implementations of four different algorithms. The implemented models in this suite mostly deal with large matrix computations and all are computationally intensive which impose heavy workloads at simulation time. In addition, they are parallel in nature and contains many for-loops as well as function calls, which provide a great level of parallelism for both domain and task decomposition. These models are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Linear Algebra</th>
<th>Matrix-Matrix Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reduction</td>
</tr>
<tr>
<td></td>
<td>Computation of Eigenvalues</td>
</tr>
<tr>
<td>Heat Conduction</td>
<td>Stationary Heat Conduction</td>
</tr>
</tbody>
</table>

Table 3.1: The MPAR benchmark test suite.

3.1 Sequential Implementations

3.1.1 Matrix-Matrix Multiplication

Matrix multiplication is a well known matrix operation that produces an \( m \times n \) matrix \( C \) from the product of an \( m \times p \) matrix \( A \) by an \( p \times n \) matrix \( B \) [24], see Equation 3.1. This can be implemented as seen in Listing 3.1.

\[
c_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{p} a_{ik} \cdot b_{kj} \quad \text{(3.1)}
\]
Listing 3.1: Modelica function for matrix multiplication.

```modelica
function matrixmultiply
input Integer m;
input Integer p;
input Integer n;
input Real A[m,p];
input Real B[p,n];
output Real C[m,n];

Real localtmp;

algorithm
for i in 1:m loop
    for j in 1:n loop
        localtmp := 0;
        for k in 1:p loop
            localtmp := localtmp + (A[i,k] * B[k,j]);
        end for;
        C[i,j] := localtmp;
    end for;
end for;
end matrixmultiply;
```

3.1.2 Reduction

Reduction is one of the most frequently used operations in many arithmetic algorithms for calculating a result by accumulating the values obtained across iterations of a for-loop. This means that the result of each iteration is dependent on the previous iteration. For instance, sum or product of an array or a matrix is a reduction operation which can be performed by using some for-loops as seen in Listing 3.2 and Listing 3.3.

Listing 3.2: Modelica function for summation of an array.

```modelica
function reduce_sum
input Integer n;
input Real A[n];
output Real result;

algorithm
result := 0;
for i in 1:n loop
    result := result + A[i];
end for;
end reduce_sum;
```
Listing 3.3: Modelica function for product of a matrix.

```modelica
function reduce_product
  input Integer row;
  input Integer col;
  input Real A[row, col];
  output Real result;

algorithm
  result := 1;
  for i in 1:row loop
    for j in 1:col loop
      result := result * A[i,j];
    end for;
  end for;
end reduce_product;
```

<table>
<thead>
<tr>
<th>X := expr + X</th>
<th>X := X + expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>X := expr - X</td>
<td>X := X - expr</td>
</tr>
<tr>
<td>X := expr * X</td>
<td>X := X * expr</td>
</tr>
<tr>
<td>X := expr / X</td>
<td>X := X / expr</td>
</tr>
<tr>
<td>X := max(expr, X)</td>
<td>X := max(X, expr)</td>
</tr>
<tr>
<td>X := min(expr, X)</td>
<td>X := min(X, expr)</td>
</tr>
</tbody>
</table>

Table 3.2: Reduction operations [25].

As it seems, the reduction algorithm is not a very complex problem. However, for large data sizes, sequential execution of this simple portion of the code may take a great amount of time. In addition, reduction operations may be used in several places in a complex and big algorithm, so that parallelizing them will obviously increase the performance of that algorithm. Moreover, it is possible to use the same parallel pattern for any associative operation shown in Table 3.2.

3.1.3 Computation of Eigenvalues

Finding the eigenvalues and eigenvectors of a square matrix is a very applicable and an important problem in many fields such as linear algebra, quantum physics, and mechanics for linear transformation [26], molecular orbitals [27], and vibration analysis [28] respectively. The model that is implemented in this benchmark suite mainly deals with the computation of all eigenvalues of a tridiagonal symmetric matrix. Details of this model, as well as the concepts of eigenvalue and eigenvector are described further in this section.

*Tridiagonal symmetric matrix:* As shown in Equation 3.2, a tridiagonal
matrix consists of zero elements everywhere except in the main diagonal, in the first upper diagonal, and first lower diagonal [31]. And, this matrix is symmetric if for all i and j \( a_{ij} = a_{ji} \), so that \( A = A^T \) [31].

\[
A = \begin{pmatrix}
\times & \times & 0 & 0 & 0 \\
\times & \times & \times & 0 & 0 \\
0 & \times & \times & \times & 0 \\
0 & 0 & \times & \times & \times \\
0 & 0 & 0 & \times & \times \\
\end{pmatrix}
\]

\[
(A - \lambda I) = \begin{vmatrix}
a_{11} - \lambda & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} - \lambda & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn} - \lambda \\
\end{vmatrix} = 0 \quad (3.3)
\]

The Equation 3.3 is called the characteristic polynomial of \( A \), and all possible solutions to this equation are eigenvalues of matrix \( A \).

**Eigenvalue:** For a given square matrix \( A \), an eigenvalue of the matrix is any scalar \( \lambda \) such that \( A - \lambda I \) has a zero determinant [30, 31],

\[
\begin{pmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n \\
\end{pmatrix} = \lambda \begin{pmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n \\
\end{pmatrix} \quad (3.4)
\]

\[
\begin{align*}
  a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= \lambda x_1 \\
  a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= \lambda x_2 \\
  \vdots & & \vdots \\
  a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= \lambda x_n
\end{align*} \quad (3.5)
\]

**Eigenvector:** An eigenvector corresponding to any eigenvalue \( \lambda \) is a nonzero vector \( \vec{x} \) such that \( A\vec{x} = \lambda \vec{x} \) [30, 31]. As shown in 3.4 and 3.5, this equation can be represented as the system of linear equations 3.5, and by solving this system for each eigenvalue \( \lambda \), we can obtain the corresponding eigenvectors.
Although the above method (characteristic polynomial) provides an appropriate way for solving eigenvalue problems, for large matrices it is not very applicable. It is likely because of the limitations in finding solutions for higher order polynomials equations which cannot be solved by a finite sequence of arithmetic operations [34]. For this reason, this method is rarely used in eigenvalue problems, and instead more efficient method such as Gerschgorin Theorem and Bisection algorithm will be applied for finding the eigenvalues of a matrix.

The implemented model in the benchmark test suite is taken from [31]. The model computes all the eigenvalues of a tridiagonal symmetric matrix which lies in the Gerschgorin interval. According to the corollary in [31], for a symmetric tridiagonal matrix $A$, all the eigenvalues are bound in Gerschgorin interval, see Equation 3.6.

$$\lambda(A) \subseteq \bigcup_{i=1}^{n} [a_i - r_i, a_i + r_i]$$

$$r_1 = b_1$$

$$r_i = b_i + b_{i-1} \quad 2 \leq i \leq (n - 1)$$

$$r_n = b_{n-1}$$

In Equation 3.6, $a_i$ and $b_i$ are the diagonal and off-diagonal elements of $A$ respectively. This can be implemented as seen in Listing 3.4.

**Listing 3.4**: Modelica function for computing the Gerschgorin interval.

```
function computeGerschgorinInterval
    input Integer length;
    input Real diagonal[:];
    input Real offDiagonal[:];
    output Real lowerBound;
    output Real upperBound;

    Real r;

    algorithm
        lowerBound:=diagonal[1]-abs(offDiagonal[1]);
        upperBound:=diagonal[1]+abs(offDiagonal[1]);

        for i in 2:length-1 loop
            r:=abs(offDiagonal[i-1])+abs(offDiagonal[i]);
            if (lowerBound > (diagonal[i]-r)) then
                lowerBound:=diagonal[i]-r;
            end if;
            if (upperBound < (diagonal[i]+r)) then
                upperBound:=diagonal[i]+r;
            end if;
        end for;
```
if (lowerBound > (diagonal[length]-abs(offDiagonal[length-1]))) then
    lowerBound:=diagonal[length]-abs(offDiagonal[length-1]);
end if;

if (upperBound < (diagonal[length]+abs(offDiagonal[length-1]))) then
    upperBound:=diagonal[length]+abs(offDiagonal[length-1]);
end if;

end computeGerschgorinInterval;

Once the Gerschgorin interval is computed by the algorithm in Listing 3.4, then the number of eigenvalue in this interval will be calculated by the function in Listing 3.5. After that, as it can be seen in Listing 3.6 the interval will be divided into a number of sub-interval such that,

- if the interval has no eigenvalue, it will be discarded.
- if the number of eigenvalues is more than 1. The interval will be divided into equal sub-intervals.
- if the number of eigenvalues is 1, then if the interval length is less than the given tolerance, the upper bound or lower bound of this interval will be considered as an eigenvalue. Otherwise, the interval will be divided into 2 sub-intervals, and the one that contains eigenvalue will be considered.

This procedure will be repeated iteratively until the length of all the intervals is below the given tolerance (a desired accuracy). As a result, at the end of this procedure all the eigenvalue within an acceptable tolerance are obtained. For more details regarding the above algorithm, see [31].

Listing 3.5: Modelica function for calculating the number of eigenvalues within an interval.

```modelica
function calNumEigenValuesLessThan
    input Integer length;
    input Real diagonal[:];
    input Real offDiagonal[:];
    input Real x;
    output Integer count;

    Real prev_diff,diff;

algorithm
    count:=0;
    prev_diff:=diagonal[1]-x;
    if(prev_diff < 0) then
```
count := count + 1;
end if;

for i in 2:length loop
diff := (diagonal[i] - x) -
((offDiagonal[i-1] * offDiagonal[i-1]) / prev_diff);
if (diff < 0) then
count := count + 1;
end if;
prev_diff := diff;
end for;
end calNumEigenValuesLessThan;

Listing 3.6: Modelica function for calculating the eigenvalues of a tridiagonal symmetric matrix.

function caleigenvalue
input Real tolerance;
input Integer length;
input Real diagonal[:];
input Real offDiagonal[:];
input Real eigenIntervals[:];
output Real newEigenIntervals[2 * length];

Integer lid, uid, newLid, newUid, numSubIntervals, offset, n;
Integer eigenValuesLessLowerBound, eigenValuesLessUpperBound;
Real avgSubIntervalWidth, lowerBound, upperBound, mid;

algorithm
offset := 0;
for i in 1:length loop
lid := (2 * i) - 1;
uid := lid + 1;

// compute the number of eigenvalue in an interval
lowerBound := eigenIntervals[lid];
upperBound := eigenIntervals[uid];

eigenValuesLessLowerBound := calNumEigenValuesLessThan(length, diagonal, offDiagonal, lowerBound);
eigenValuesLessUpperBound := calNumEigenValuesLessThan(length, diagonal, offDiagonal, upperBound);
numSubIntervals := eigenValuesLessUpperBound -
    eigenValuesLessLowerBound;

// if the number of eigenvalues in an interval is 0
// discards the intervals.

// if the number of eigenvalues in an interval is 1, it splits
//(bisects) that interval into two half and considers the one
// in which the eigenvalue exists.
// no bisection is done when the interval length is less than
// a given tolerance (a desired accuracy)
if(numSubIntervals == 1) then
  mid := (lowerBound + upperBound) / 2;
  n := calNumEigenValuesLessThan(length, diagonal, offDiagonal, mid) - eigenValuesLessLowerBound;
newLid := (2 * offset) + 1;
newUid := newLid + 1;

// check if the interval size is less than tolerance levels
if (upperBound - lowerBound < tolerance) then
  newEigenIntervals[newLid] := lowerBound;
  newEigenIntervals[newUid] := upperBound;

// the eigenvalue lies in the right
// half of the interval
elseif (n == 0) then
  newEigenIntervals[newLid] := mid;
  newEigenIntervals[newUid] := upperBound;

// the eigenvalue lies in the left half
// of the interval
else
  newEigenIntervals[newLid] := lowerBound;
  newEigenIntervals[newUid] := mid;
end if;

// if the number of eigenvalues is more than 1
// the interval is split into equal intervals
// of size divisionWidth
elseif (numSubIntervals > 1) then
  divisionWidth := (eigenIntervals[uid] - eigenIntervals[lid]) / numSubIntervals;

  for j in 1..numSubIntervals loop
    newLid := (2 * (offset + j) - 1);
    newUid := newLid + 1;
    newEigenIntervals[newLid] := eigenIntervals[lid] + (j - 1) * divisionWidth;
    newEigenIntervals[newUid] := newEigenIntervals[newLid] + divisionWidth;
  end for;
end if;

offset := offset + numSubIntervals;
end for;
end caleigenvalue;
3.1.4 Heat Conduction

When objects are in a physical contact with each other different forms of energy are transferred between them. Heat conduction is a case where thermal energy is transferred from one object with higher temperature to another object with lower temperature, until the temperature difference between the objects become equilibrated [32]. This can also happen in one object where its surfaces have different temperatures. Heat conduction can be transient or steady-state. Conduction is known as transient when the temperature of the conducting object varies regularly with the time and position. However, when the temperature is constant and is not changing after equilibration, conduction is called steady-state or stationary.

3.1.4.1 Stationary Heat Conduction

The model that has been implemented in this thesis mainly deals with stationary heat conduction. The problem is to find temperature distribution in a surface of size $n \times n$. According to [32, 33] in a two-dimensional stationary situation such as a square $[0, 1] \times [0, 1]$ with specified boundary conditions, heat conduction is represented by the differential equation in 3.7.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0, \quad 0 < x, y < 1 \quad (3.7)$$

To calculate heat conduction in a $n \times n$ surface with specified boundary conditions, we can define an equidistant grid $(x_i, y_j)$ with $1 \leq i, j \leq n + 2$ as shown in Figure 3.1, and use finite differences approximation methods [35] to discretize the differential equation 3.7. Therefore, Equation 3.7 is replaced by the following system of linear equations in 3.8.

$$-4T_{i,j} + T_{i+1,j} + T_{i-1,j} + T_{i,j-1} + T_{i,j+1} = 0,$$

$$T_{i,j} = (T_{i-1,j} + T_{i+1,j} + T_{i,j+1} + T_{i,j-1})/4, \quad 1 \leq i, j \leq N \quad (3.8)$$

In Equation 3.8, $T_{i,j} = T(x_i, y_j)$ refers to an approximate temperature at grid point $(x_i, y_j)$.

Direct numerical methods such as Gaussian Elimination can be used to solve the system of linear equations 3.8. However, when the number of grid points is large, iterative methods such as Jacobi method are instead more appropriate than direct methods for solving such large system of linear equations. The idea behind the Jacobi method is to find the next solution of the equation using the obtained solution in the previous iteration. Therefore, if the approximate solution (temperature) $T_{i,j}^{k}$ for grid point $(x_i, y_j)$ at the $k$th
As shown in Listing 3.7, to calculate the temperature distribution in a surface of size $n \times n$, an $(n+2) \times (n+2)$ matrix is needed to store the boundary values and temperatures at grid points $(x_i, y_j)$, where $2 \geq i, j \leq n+1$. The algorithm starts by setting the boundary conditions and initial values, and then iteratively from top to bottom modify each row from left to right. As it can be seen from the equation 3.9, the temperature of grid point $(x_i, y_j)$ is calculated by using the values of grid points around it i.e, top, bottom, right, and left. Therefore, before modifying each element it is necessary to store the value of that element, so that it can be used for calculating the temperature of the grid points located at right and below it. More details can be found in [33].

**Listing 3.7: Stationary heat conduction.**

```plaintext
function laplsolve
  input Integer n;
  input Integer maxiter;
  output Real T[n+2,n+2];

  Real toprow[n];
  Real tmp, left;

algorithm

  // Set boundary conditions and initial values
  for i in 1:n+2 loop
    for j in 2:n+1 loop
      T[i,j]:=0.0;
    end for;
  end for;
```

Figure 3.1: Equidistant grid with boundary conditions.
3.2 Parallel Implementation

As mentioned earlier, the preliminary aim of the MPAR benchmark suite is on evaluating the feasibility of the new Modelica language extension presented in [17]. As a result of this work new parallel language constructs including parallel variables (parglobal, parlocal), parallel function, kernel function (parkernel), and parallel for-loop (parfor) have been added to the OpenModelica compiler to efficiently generate parallel simulation code for algorithmic parts of Modelica with respect to execution on OpenCL-enabled platforms. However, to be able to use this extension, the end user i.e, the modeler has to explicitly state parallelism in models using these language constructs. For this reason, the MPAR benchmark suite has been extended with the parallel implementations of the models using these new language constructs.

3.2.1 Matrix-Matrix Multiplication

As was shown in the sequential implementation (Listing 3.1), since there is no dependencies between the iterations of the for-loops, therefore it is possible to calculate the elements of the matrix C in parallel by simply converting the most outer loop to a parfor-loop. In this way the iterations of the parfor-loop will be distributed equally among the available number of
threads supported by the parallel device, so that each thread will calculate one row of the matrix C independently from other threads. However, if the number of threads is less than the number of iterations some threads may execute more than one iteration [17]. This implementation is shown in Listing 3.8.

**Listing 3.8:** Parallel implementation of matrix-matrix multiplication using parfor-loop.

```plaintext
function matrixmultiply
  input Integer m;
  input Integer n;
  input Integer k;
  //input and output parallel variable used on device
  parglobal input Integer pn;
  parglobal input Integer pk;
  parglobal input Real pA[m,n];
  parglobal input Real pB[n,k];
  parglobal output Real pC[m,k];

  parglobal Real plocaltmp;

algorithm
  //Iterations of the parfor-loop will be distributed among the total number of
  //threads supported by the device, and each thread will calculate one row of
  //the matrix pC.
  for i in 1:m loop
    for j in 1:pk loop
      plocaltmp := 0;
      for h in 1:pn loop
        plocaltmp := plocaltmp + (pA[i,h] * pB[h,j]);
      end for;
      pC[i,j] := plocaltmp;
    end for;
  end parfor;
end matrixmultiply;
```

In addition to using parfor-loop for parallelizing the matrix multiplication algorithm, another implementation using kernel function which is much faster is also presented in this suite (Listing 3.9). A kernel function is declared using keyword `parkernel`. As discussed earlier in Chapter 2, each instance of the kernel function will be mapped to a single thread and all instances will be executed in parallel. Therefore, in this case, each element of the matrix C i.e, $c_{ij}$ will be calculated independently from the other elements by a separate thread. `oclGetGlobalId(1)` and `oclGetGlobalId(2)` functions here are used to access each global thread Id and accordingly each element of the matrix C in first and second dimensions respectively.
Listing 3.9: Parallel implementation of matrix-matrix multiplication using kernel function.

```plaintext
parkernel function matrixmultiply
  parglobal input Integer pn;
parglobal input Real pA[:,:];
parglobal input Real pB[:,:];
parglobal output Real pC[size(pA,1),size(pB,2)];

  Real plocaltmp;
  Integer i,j;

algorithm

  // Returns the unique global thread Id value
  // for first and second dimension
  i := oclGetGlobalId(1);
  j := oclGetGlobalId(2);

  plocaltmp := 0;
  for h in 1:pn loop
    plocaltmp := plocaltmp + (pA[i,h] * pB[h,j]);
  end for;
  pC[i,j] := plocaltmp;
end matrixmultiply;
```

3.2.2 Reduction

As mentioned in Section 3.1.2, performing the reduction over an array or a matrix can be done by using for-loops. However, since the result of each iteration is dependent on the obtained result from previous iterations, therefore it is not possible to directly convert the loop to a parallel for-loop (parfor-loop). To parallelize such dependent iterations, we need to explicitly specify a desired number of threads and work-groups and define on what range of the array threads should do accumulation. This can be done by implementing a kernel function as shown in Listing 3.10. The idea to parallelize the algorithm is to divide the input array into a number of blocks (work-group) and map each block to a separate compute unit on parallel device. In this way, the summation of each block will be calculated separately from other block in parallel, and then the result of all blocks will be added together sequentially on the host to obtain the final result. It is also worth to note that this kernel has been implemented using shared memory, so that threads within each work-group can access local and shared memory rather than global memory. Accessing to shared memory by threads within each work-group also should be explicitly synchronized by using the function `oclLocalBarrier(0)`, so that we can ensure that the obtained result is correct.
Listing 3.10: Parallel implementation of reduction using kernel function.

```
parkernel function reduce
| parglobal input Real pA[:];
| parglobal output Real pC[size(pA,1)];
| parglobal output Integer tmp;

// declare shared memory
parlocal Real sdata[oclGetLocalSize(1)];

Integer s, localSize;
Integer localThreadId, workGroupId, globalThreadId;

algorithm
// returns the size of the block
localSize := oclGetLocalSize(1);

// returns the unique global thread Id value
// for first dimension
globalThreadId := oclGetGlobalId(1);

// returns the current work-group Id
workGroupId := oclGetGroupId(1);

// returns the unique local thread Id value
// for first dimension
localThreadId := oclGetLocalId(1);

// copy from global memory to shared memory
sdata[localThreadId] := pA[globalThreadId];

// synchronize shared memory
oclLocalBarrier(0);

s := integer(localSize/2);
while (s > 0) loop
    // perform reduction in shared memory
    if ((localThreadId <= s)) then
        sdata[localThreadId] := sdata[localThreadId] + sdata[localThreadId + s];
    end if;

    // synchronize shared memory
    oclLocalBarrier(0);

    s := integer(s / 2);
end while;

// copy the result of the block to global memory
if (localThreadId == 1) then
    pC[workGroupId] := sdata[localThreadId];
end if;
end reduce;
```
As shown in the kernel function and Figure 3.2, in each work-group first the data are copied from global memory to shared memory, and then the block of data will be divided into two halves. Afterward, half of the threads will update the values in one half by adding the corresponding values in another half. This division will be done iteratively until the block is not dividable anymore. The result of each block will be then copied from the shared memory to the global memory, and then on the host the final result will be calculated sequentially.

3.2.3 Computation of Eigenvalues

This model can be parallelized using two kernel functions `calcNumEigenValueInterval()` and `recalculateEigenIntervals()` shown in Listing 3.11 and Listing 3.12 respectively. The idea is to map each interval to a separate thread and perform computation for each interval independently from other in parallel. To do so, for each interval the number of eigenvalues should
first be calculated by the kernel function \texttt{calNumEigenValueInterval()} and be stored for each thread. Once, the number of eigenvalues for each interval have been calculated then based on this number each thread will recalculate its own interval using kernel function \texttt{recalculateEigenIntervals()}. It should also be noted that, according to the implementation of the OpenModelica compiler presented in [17], it is not possible to call a normal function from a kernel function. Thus, to be able to use the \texttt{numEigenValuesLessThan()} function it is required to define this function as a parallel function. This can be done by declaring the function using the \textit{parallel} keyword as shown in Listing 3.13.

**Listing 3.11:** Kernel function that calculates the number of eigenvalues with-in an interval.

```plaintext
parker kernel function calNumEigenValueInterval
  parglobal input Integer length;
  parglobal input Real diagonal[:];
  parglobal input Real offDiagonal[:];
  parglobal input Real eigenIntervals[:];
  parglobal output Integer numEigenIntervals[oclGetGlobalSize(1)];
  parglobal output Integer tmp;

  Integer globalThreadId, lowerId, upperId;
  Integer eigenValuesLessLowerBound, eigenValuesLessUpperBound;
  Real lowerBound, upperBound;

  algorithm
  // Returns the unique global thread Id value
  // for first dimension
  globalThreadId := oclGetGlobalId(1);

  lowerId := (2*globalThreadId) - 1;
  upperId := lowerId + 1;

  lowerBound := eigenIntervals[lowerId];
  upperBound := eigenIntervals[upperId];

  eigenValuesLessLowerBound := numEigenValuesLessThan(length,
      diagonal, offDiagonal, lowerBound);

  eigenValuesLessUpperBound := numEigenValuesLessThan(length,
      diagonal, offDiagonal, upperBound);

  // store the number of eigenvalue of each interval
  // for each thread
  numEigenIntervals[globalThreadId] := eigenValuesLessUpperBound -
      eigenValuesLessLowerBound;

  end calNumEigenValueInterval;
```

38
Listing 3.12: Kernel function that recalculates and divides the eigenvalue intervals.

```c
parkernel function recalculateEigenIntervals
parglobal input Real tolerance;
parglobal input Integer length;
parglobal input Real diagonal[];
parglobal input Real offDiagonal[];
parglobal input Real eigenIntervals[];
parglobal input Integer numEigenIntervals[];
parglobal output Real newEigenIntervals[size(eigenIntervals,1)];
parglobal output Integer tmp;

Integer globalThreadId, currentindx, indx, lowerId, upperId, lId, uId;
Real divisionWidth, midValue, lowerBound, upperBound, n;

algorithm

// Returns the unique global thread Id value
// for first dimension
globalThreadId := oclGetGlobalId(1);
currentindx := globalThreadId - 1;
lowerId := (2*globalThreadId) - 1;
upperId := lowerId + 1;

indx := 1;
while (currentindx >= numEigenIntervals[indx]) loop
  currentindx := currentindx - numEigenIntervals[indx];
  indx := indx + 1;
end while;

lId := (2*indx) - 1;
uId := lId + 1;
lowerBound := eigenIntervals[lId];
upperBound := eigenIntervals[uId];

if (numEigenIntervals[indx] == 1) then
  midValue := (upperBound + lowerBound) / 2;
  n := numEigenValuesLessThan(length, diagonal, offDiagonal, midValue) -
      numEigenValuesLessThan(length, diagonal, offDiagonal, lowerBound);
  if (upperBound - lowerBound < tolerance) then
    newEigenIntervals[lowerId] := lowerBound;
    newEigenIntervals[upperId] := upperBound;
  elseif (n == 0) then
    newEigenIntervals[lowerId] := midValue;
    newEigenIntervals[upperId] := upperBound;
  else
    newEigenIntervals[lowerId] := lowerBound;
  end if;
end if;
```

39
newEigenIntervals[upperId] := midValue;

end if;

// split the intervals into equal intervals
// of size divisionWidth
elseif (numEigenIntervals[indx] > 1) then
    divisionWidth := (upperBound - lowerBound) / numEigenIntervals[indx];
    newEigenIntervals[lowerId] := lowerBound + divisionWidth * currentindx;
    newEigenIntervals[upperId] := newEigenIntervals[lowerId] + divisionWidth;
end if;
end recalculateEigenIntervals;

Listing 3.13: Parallel function that calculates the number of eigenvalues less than x.

parallel function numEigenValuesLessThan
parglobal input Integer length;
parglobal input Real diagonal[:];
parglobal input Real offDiagonal[:];
parglobal input Real x;
parglobal output Integer count;

Real prev_diff, diff;
Integer dummy=1;

algorithm

    count := 0;
    prev_diff := diagonal[dummy] - x;
    if (prev_diff < 0) then
        count := count + 1;
    end if;

    for i in 2:length loop
        diff := (diagonal[i]-x) - ((offDiagonal[i-1] * (offDiagonal[i-1]))/prev_diff);
        if (diff < 0) then
            count := count + 1;
        end if;
        prev_diff := diff;
    end for;
end numEigenValuesLessThan;
3.2.4 Stationary Heat Conduction

Calculating the temperature at each grid point \((x_i, y_j)\) is dependent on the values of grid points around it before they are modified. Therefore, because of this dependency it is not possible to directly convert for-loops to parfor-loops without any thread management. If we do so, all iterations will be distributed automatically among available threads and they will be executed independently from each other at the same time without any specific order. As a result, the values of elements around a specific grid point which should be used for calculating the temperature at that point may have been already modified by other threads. Therefore, the result of the calculations will not be correct. Thus, to be able to perform calculations in parallel, it is required to explicitly control and manage the execution of the threads which are run in parallel. To do so, two different implementations are presented in Listing 3.14 and Listing 3.15. In the former implementation, one work-group of specific number of threads will be defined and then the equidistant grid will be divided among those threads within that work-group. However, the latter divides the equidistant grid among different work-groups with specific number of threads and uses kernel function to calculate the temperature at each grid point.

Listing 3.14: Parallel implementation of stationary heat conduction using parfor-loop.

```plaintext
defunction laplsolve
    input Integer n;
    input Integer maxiter;
    input Integer num_threads;
    output Real T[n+2,n+2];

    // the total number of global threads which execute
    // the kernel in parallel
    Integer global_size[1] = {num_threads};

    // the total number of local threads which
    // run in parallel in each work-group
    Integer local_size[1] = {num_threads};

    Integer block_size;

    parglobal Real ptoprow[n];
    parglobal Real pT[n+2,n+2];
    parglobal Real ptmp, pleft, prightmost;
    parglobal Integer pn,pk,pmaxiter;
    parglobal Integer pstart, pend, pblock_size,pk;

    algorithm
    pn := n;
    pmaxiter := maxiter;
```
// Set boundary conditions and initial values
// iterations will be distributed among the
// number of threads supported by device and
// will be executed in parallel, each thread
// will initializes one row of the matrix pT.
parfor i in 1:n+2 loop
    for j in 2:pn+1 loop
        pT[i,j] := 0.0;
    end for;
    pT[i,1] := 1.0;
    pT[i,pn+2] := 1.0;
end parfor;
parfor j in 1:n+2 loop
    pT[pn+2,j] := 2.0;
end parfor;

// specify the number of threads and work-groups
// to be used for a kernel function execution
oclSetNumThreads(global_size, local_size);

// divide the work-group into a number of blocks
block_size := integer(n / num_threads);
if( (block_size * num_threads) < n ) then
    block_size := block_size + 1;
end if;

// copy from host to device
pblock_size := block_size;

// each thread will execute separate iteration,
// and each will modify its own portion (block)
parfor id in 1:num_threads loop
    for pk in 1:pmaxiter loop

        // specify the lower and upper boundary of
        // each portion for each thread
        pstart := (id - 1) * pblock_size + 2;
        pend := id * pblock_size + 1;
        if(pend > pn+1) then
            pend := pn+1;
        end if;

        // each thread saves it's portion of the first rows.
        if(pstart <= (pn +1) ) then
            for j in pstart:pend loop
                ptoprow[j-1] := pT[1,j];
            end for;

            // synchronization
            oclGlobalBarrier(0);

            for i in 2:pn+1 loop
                pleft := pT[i,pstart-1];
            end for;
    end for;
end parfor;
As shown in Listing 3.14, since the iterations for initializing and setting boundary conditions are independent from each other, therefore it is possible to parallelize them by using parfor-loops. In this way the iterations will be distributed automatically among the total number of threads supported by the parallel devices, and each will execute one iteration. To parallelize the rest of the code we can map the equidistant grid to one work-group of threads. Then, we can divide the work-group into blocks of threads and use parfor-loop to execute the all blocks of threads in parallel. In this way, each block can calculate the temperature at grid points of its own portion independently from other blocks in parallel. Also, since all threads are executed within the same work-group, it is possible to manage and synchronize their execution using the `oclGlobalBarrier(0)` function. The reason for specifying only one work-group is likely because in OpenCL synchronization can only be done among threads within each work-group and there is no synchronization between work-items in two different work-group [18, 19]. In addition,
if we do not explicitly specify one work-group, then the parfor-loop will implicitly define a number of work-groups and distribute iterations threads within those work-groups. In this way, since some blocks of threads may be executed in different work-groups, therefore the synchronization functions will not work properly. Consequently, the result of the computation will be incorrect. However, another implementation using kernel function is presented in 3.15 which uses all available compute units in the parallel devices and provides much better performance and faster execution speed compared to the implementation using parfor.

Listing 3.15: Parallel implementation of stationary heat conduction using kernel function.

```cpp
parkernel function laplsolve
    parglobal input Integer n;
    parglobal input Real T[:, :];
    parglobal output Real T2[size(T,1), size(T,2)];
    parglobal output Integer tmp;
    Integer i, j;

    algorithm
    // Returns the unique global thread Id value
    // for first and second dimension
    i := oclGetGlobalId(1) + 1;
    j := oclGetGlobalId(2) + 1;

    // Set boundary conditions and initial values
    // the first row and column will be initialized
    // by one thread, and the last row and column
    // will be initialized by another thread.

    // fill columns 2 up to n+1 of first row
    if(i == 2) then
        T2[1, j] := 0.0;
    end if;

    // fill columns 2 up to n+1 of last row
    if(i == n + 1) then
        T2[n+2, j] := 2.0;
    end if;

    // fill rows 2 up to n+1 of first column
    if(j == 2) then
        T2[i, 1] := 1.0;
    end if;

    // fill rows 2 up to n+1 of last column
    if(j == n + 1) then
        T2[i, n+2] := 1.0;
    end if;
```
Each instance of this kernel function will be executed by separate threads which are mapped to the elements of the matrix T2. And, as it can be seen each element is accessed by the `oclGetGlobalId(1)` function, where parameter 1 and 2 in this function denote the row and column indexes of the matrix respectively.
Chapter 4

Measurements

In this section the simulation measurements of all the models in the benchmark test suite from Chapter 3 (both sequential and parallel with the new added parallel language constructs) are presented. The models have been simulated on different hardware configurations five times each with the duration of 0.001 seconds and time step of 0.001 seconds (1 time step in total), and then the average simulation time is measured. The reason for performing simulation for only 1 time step is because the models are time-invariant, so that the only thing that affects the performance is the problem size. Detailed measurements data can be found in Appendix A. The simulation time is calculated by using the `clock_gettime()` function from the C time standard library. This function is integrated in the body of the main simulation function of the OpenModelica runtime system (Listing 4.1), and is called once when the simulation starts and once when the simulation finishes (reading and writing to the result file will be also included in the measurements). The difference between the returned values then gives the simulation time.

Listing 4.1: The main simulation function of the OpenModelica runtime system.

```c
int _main_SimulationRuntime(int argc, char** argv)
{
    double simulationtime;
    timespec starttime, endtime;

    // timing starts
    clock_gettime(CLOCK_REALTIME, &starttime);

    int retVal = -1;

    // initRuntimeAndSimulation returns 1 if an error occurs
    if (initRuntimeAndSimulation(argc, argv))
        return 1;

    if (interactiveSimulation)
        {
```
cout << "startInteractiveSimulation: " << version << endl;
retVal = startInteractiveSimulation(argc, argv);
}
else {
    cout << "startNonInteractiveSimulation: " << version << endl;
    retVal = startNonInteractiveSimulation(argc, argv);
}
deInitializeDataStruc(globalData, ALL);
free(globalData);
fflush(NULL);

// timing finishes
clock_gettime(CLOCK_REALTIME, &endtime);
simulationtime = (endtime.tv_sec - starttime.tv_sec) +
(double)(endtime.tv_nsec-starttime.tv_nsec) /
(double)1000000000L;

cout <<"Simulation time is: " << simulationtime
    << " second" << endl;
EXIT(retVal);
}

4.1 Hardware

4.1.1 CPU Configuration

For executing sequential and parallel code generated by the old OpenModelica compiler as well as the new one with the new language extensions, an Intel Xeon E5520 CPU [36] has been used. It has 16 cores each with 2.27 GHz clock frequency. The maximum number of threads in each work-group (work-group size) supported by this CPU is 1024.

4.1.2 GPU Configuration

In addition of using a multi-core CPU, a NVIDIA Fermi-Tesla M2050 GPU [7] has also been used to execute parallel simulations. This GPU supports both single precision and double precision floating point, with peak performance of 1.03 TFlops and 515 Gflops respectively. It also supports maximum 1024 threads per work-group.
4.2 Simulation Results

4.2.1 Matrix-Matrix Multiplication

As discussed in Chapter 3, the model has been parallelized with two different implementations one using kernel function and another one using parfor-loop. As it is illustrated in Figure 4.2 and Figure 4.4, for the maximum matrix size (512×512) the gained speedup with former implementation is considerably greater than the latter implementation. The speedups with kernel implementation are approximately 115 and 27 on GPU and multi-core CPU respectively. With parfor-loop implementation on GPU, the gained speedup is around 6 and on multi-core CPU it is around 12. The reason is because in kernel function each element of the matrix C is calculated with a separate thread, while in the parfor-loop each thread computes each row of the matrix. In addition, with the kernel implementation specifying the total number of global threads as well as local threads is done explicitly. Therefore, it is possible to run the model with a desired number of threads and work-groups until the best performance is achieved on GPU and multi-core CPU. However, using parfor-loop only a limited number of threads will be launched (maximum work-group size supported by the parallel device). It is also notable from Figure 4.1 and Figure 4.3, parfor-loop simulation on multi-core CPU has better performance compared to the GPU. This is also because of the limitation in parfor-loop regarding the implicit division of
4.2.2 Reduction

As might be expected, it is clear from the trends in Figure 4.5 that the serial execution does not take that much time even when the model is simulated with very large problem size. The reason is likely because the reduction algorithm is not very complex and can be easily implemented with a simple for-loop. However, as mentioned earlier reduction operations may be used in several places in a complex and big algorithm, so that parallelizing them will increase the performance of that algorithm. For the largest problem size the gained speedup for this model is approximately around 6 for GPU and 2 for multi-core CPU (Figure 4.6).

4.2.3 Computation of Eigenvalues

As discussed in Chapter 3, this model deals with the computation of all eigenvalues of a tridiagonal symmetric matrix of size n×n. Because the matrix is symmetric and tridiagonal, therefore the inputs for this model are the main diagonal and off-diagonal elements of the matrix. The main diagonal elements are stored in an array of size n and off-diagonal elements
Figure 4.3: Simulation time plot as function of the parameter $M$ (matrix size $M \times M$), for matrix multiplication model using parfor.

Figure 4.4: Speedup graph as function of the parameter $M$ (matrix size $M \times M$), for matrix multiplication model using parfor.
Figure 4.5: Simulation time plot as function of the parameter N (array size) for reduction model.

Figure 4.6: Speedup graph as function of the parameter N (array size) for reduction model.
are stored in an array of size $n-1$. Since in this model all the computations are performed on these two arrays, the model has been simulated for different values of parameters $n$. The minimum value was chosen to be 128, then it is doubled until the value becomes 8192. Figure 4.7 shows as the problem size increases from $n=128$ to $n=8192$, how the simulation time for serial code increases in comparison with the simulation time of parallel code. As it is illustrated, for the lowest bound ($n=128$) it took only 1.543 seconds to simulate the serial code on the CPU, while on the multi-core CPU and GPU the simulation was completed after 3.049 and 7.188 seconds respectively. The simulation time for the sequential code remained less than the parallel code until the beginning of the $n=256$, but then increased dramatically from 5.116 to 574.057 seconds over the larger problem size. From $n=512$ onward, the performance on GPU grew steadily to just over 47.71 for $n=8192$. While on multi-core CPU it is clearly seen that the gained speedup was relatively less (approximately 3). And, the simulation time was quite a lot (208.789 seconds) compared to the simulation time on GPU (12.032 seconds). The reason is likely because of the data-parallel computational power of GPUs compared to the CPUs. In addition, since the problem size is large and parallelism has been exploited sufficiently in the algorithm, it was expected to achieve a better performance on GPU compared to CPU.
4.2.4 Stationary Heat Conduction

The model was simulated with different sizes of the parameter n which refers to the size of the surface and the equidistant grid \((x_i, y_j)\) with \(1 \leq i, j \leq n+2\). It can be seen from the Figure 4.9 and Figure 4.10 that the gained speedup is not very noticeable when the model is parallelized using parfor-loop. The reason is obvious: the model is explicitly defined to perform the parallel computations in only one compute unit of the device. This is necessary, otherwise the parfor-loop will distribute the iterations into different workgroups, so that thread synchronization will not work properly. In contrast, as can be seen in Figure 4.11 and Figure 4.12, the implementation using kernel function has provided a considerable speedup over large problem size. Starting from the CPUs, at the beginning the differences are not very noticeable since for \(n=128\) the simulation time is 1.958 seconds for serial execution and 0.959 seconds for parallel execution on multi-core CPU. As the problem size increases the differences among the trends also slightly increase, in which at the point that the matrix size is \(2048 \times 2048\) the serial simulation takes a very long time (487.342 seconds), while the parallel simulation on CPU takes 76.077 seconds (around 7x faster). On the other hand, at the beginning, the simulation time on GPU was quite a lot (8.704 seconds) in comparison with simulation times on CPUs. However, as the value of n became larger simulation on GPU also got better performances, where at point \(n=2048\) the amount of gained speedup was around 23.
Figure 4.9: Simulation time plot as function of the parameter $N$ (matrix size $N \times N$) for stationary heat conduction model using parfor-loop.

Figure 4.10: Speedup graph as function of the parameter $N$ (matrix size $N \times N$) for stationary heat conduction model using parfor-loop.
Figure 4.11: Simulation time plot function of the parameter $N$ (matrix size $N \times N$) for stationary heat conduction model using kernel function.

<table>
<thead>
<tr>
<th></th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU E5520 (Serial)</td>
<td>1.958</td>
<td>7.903</td>
<td>32.104</td>
<td>122.754</td>
<td>487.342</td>
</tr>
<tr>
<td>CPU E5520 (Parallel)</td>
<td>0.959</td>
<td>1.875</td>
<td>5.488</td>
<td>19.711</td>
<td>76.077</td>
</tr>
</tbody>
</table>

Figure 4.12: Speedup graph function of the parameter $N$ (matrix size $N \times N$) for stationary heat conduction model using kernel function.
Chapter 5

Discussion and Conclusions

5.1 Guidelines to Use the New Parallel Language Constructs

The most important task in all approaches regarding parallel code generation is to provide an appropriate way for analyzing and finding parallelism in sequential codes. In automatic parallelization approaches, the whole burden of this task is on the compiler and tool developer. While in explicit parallelization approaches, it is the responsibility of the modeler to analyze the source code and define which parts of the code are more appropriate to be explicitly parallelized. This requires a good understanding of the concepts of parallelism to avoid inefficient and incorrect generated code. In addition, it is necessary to know the constraints and limitations involved with using explicit parallel language constructs to avoid compile time errors. For this purpose, in this section several suggestions on how to use the current Modelica language extension to parallelize Modelica models even more efficiently are given.

- Try to declare parallel variables as well as copy assignments among normal and parallel variables as less as possible since the costs of data transfers from host to devices and vice versa are very expensive.

- In order to minimize the number of parallel variables as well as data transfers between host and devices, it is better not to convert for-loops with few iterations over simple operations to parallel for-loops (parfor-loops).

- It is not always useful to have parallel variables and parfor-loops in the body of a normal for-loop which has many iterations. Especially in cases where there are many copy assignments among normal and parallel variables.
• Although it is possible to declare parallel variables and also parfor-loops in a function, there are no advantages when there are many calls to the function (especially in the body of a big for-loop). This will increase the number of memory allocations for parallel variables as well as the number of expensive copies required to transfer data between host and devices.

• Do not convert directly a for-loop to a parfor-loop when the result of each iteration depends on other iterations. In this case, although the compiler will correctly generate parallel code for the loop, the result of the computation may not be correct.

• Use parfor-loop in situations where the loop has many independent iterations and each iteration takes a long time to be completed.

• Try to parallelize models using kernel functions as much as possible rather than using parfor-loops. This will enable you to explicitly specify a desired number of threads and work-group to get the best performance.

• If the global work size (total number of threads to be run in parallel) and the local work size (total number of threads in each work-group) need to be specified explicitly, then the following points should be considered. First, the work-group size (local size) should not be zero, and also it should not exceed the maximum work-group size supported by the parallel device. Second, the local size should be less or equal than the global-size. Third, the global size should be evenly divisible by the local size.

• The current implementation of OpenCL does not support recursive functions, therefore it is not possible to declare a recursive function as a parallel function.

5.2 Result Interpretation

According to the measurements presented in Chapter 4, it can clearly be seen that in overall a considerable speedup has been achieved for some models as a result of running parallel code on OpenCl-enabled platforms including multi-core CPUs as well as GPUs. It is also notable that as the problem size increases, it is expected to achieve even better performance for very large problem sizes. Another conclusion is that, when the problem size is not very large the sequential execution has better performance than the parallel executions and the speedup factor is less or equal than 1. This is not surprising since for executing even a simple code on OpenCL devices it is required to create an OpenCL context within those devices, allocate
OpenCL memory objects, transfer input data from host to those memory objects, perform computations, and finally transfer back the result to the host. Consequently, performing all these operations normally takes more time compared to the sequential execution when the problem size is small. It can also be seen that, as the sizes of the models increase, the simulations get better relative performance on GPU compared to multi-core CPU. Thus, to fully utilize the power of parallelism using GPUs it is required to have large regular data structures parts which can be operated on simultaneously by being decomposed to all blocks and threads available on GPU. Otherwise, performing parallel codes on a multi-core CPU would be a better choice to achieve more efficiency and speedup.

5.3 Future Work

As mentioned earlier the implemented benchmark suite in this thesis only involves the algorithmic parts of the Modelica language. However, the main use of the Modelica language is for declarative equation-based modelings using mathematical equations directly in Modelica models. Therefore, it would be worth extending the current benchmark suite with time-variant models that cover the equation part of the language as well. Another future consideration regarding the benchmarking of code generated from the OpenModelica compiler with respect to the implemented benchmark suite is to compare the performance of OpenModelica with other Modelica based environments such as Dymola [37]. In addition, it might be worth evaluating critical performance factors and relative strength of the generated OpenCL code compared to manual written OpenCL and CUDA implementation of the same algorithms.

5.4 Conclusions

In this thesis a suitable benchmark test suite for validating the feasibility and performance of the new Modelica language extensions was presented. In addition, several measurements of this benchmark suite on different hardware configurations such as single-core and multi-core CPUs as well as GPUs were provided. According to the results of the measurements the following points can be concluded. First, parallelization of Modelica models is a goal worth pursuing considering the complexity of many Modelica models and the corresponding long simulation times associated with the simulation of these models. Second, the parallel constructs introduced in the algorithmic parts of the Modelica language provides a novel way in Modelica toward simulating computationally heavy Modelica models on OpenCL-enabled platforms. Finally, the MPAR benchmark suite offers an appropriate way of measuring the performance of computationally intensive Modelica models that make
use of the novel parallel Modelica constructs. In addition, the parallel algorithms in the benchmark suite can be a starting point for programmer to start implementing parallel Modelica models when using the new Modelica language extensions.
Appendix A

Simulation Time Measurements

A.1 Matrix-Matrix Multiplication

<table>
<thead>
<tr>
<th>M, N, K</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.094</td>
<td>0.092</td>
<td>0.093</td>
<td>0.094</td>
<td>0.093</td>
<td>0.093</td>
</tr>
<tr>
<td>64</td>
<td>0.735</td>
<td>0.744</td>
<td>0.74</td>
<td>0.743</td>
<td>0.743</td>
<td>0.741</td>
</tr>
<tr>
<td>128</td>
<td>5.865</td>
<td>5.889</td>
<td>5.87</td>
<td>5.879</td>
<td>5.869</td>
<td>5.875</td>
</tr>
<tr>
<td>256</td>
<td>58.451</td>
<td>58.252</td>
<td>58.47</td>
<td>58.474</td>
<td>58.466</td>
<td>58.426</td>
</tr>
</tbody>
</table>

Table A.1: Matrix-matrix multiplication simulation times for the Intel Xeon E5520 CPU (serial).

<table>
<thead>
<tr>
<th>M, N, K</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.177</td>
<td>0.175</td>
<td>0.177</td>
<td>0.188</td>
<td>0.18</td>
<td>0.179</td>
</tr>
<tr>
<td>64</td>
<td>0.354</td>
<td>0.357</td>
<td>0.367</td>
<td>0.364</td>
<td>0.373</td>
<td>0.363</td>
</tr>
<tr>
<td>128</td>
<td>1.286</td>
<td>1.283</td>
<td>1.28</td>
<td>1.301</td>
<td>1.287</td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: Matrix-matrix multiplication simulation times for the Intel Xeon E5520 CPU (parallel using parfor).

<table>
<thead>
<tr>
<th>M, N, K</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1.325</td>
<td>1.267</td>
<td>1.279</td>
<td>1.288</td>
<td>1.28</td>
<td>1.287</td>
</tr>
<tr>
<td>64</td>
<td>1.475</td>
<td>1.47</td>
<td>1.479</td>
<td>1.467</td>
<td>1.529</td>
<td>1.484</td>
</tr>
<tr>
<td>128</td>
<td>2.679</td>
<td>2.662</td>
<td>2.659</td>
<td>2.675</td>
<td>2.645</td>
<td>2.664</td>
</tr>
<tr>
<td>512</td>
<td>86.6</td>
<td>86.385</td>
<td>86.386</td>
<td>86.418</td>
<td>86.419</td>
<td>86.441</td>
</tr>
</tbody>
</table>

60
Table A.3: Matrix-matrix multiplication simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel using parfor).

<table>
<thead>
<tr>
<th>M, N, K</th>
<th>Global-Size</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>M*K</td>
<td>8*8</td>
<td>0.133</td>
<td>0.138</td>
<td>0.136</td>
<td>0.129</td>
<td>0.153</td>
<td>0.137</td>
</tr>
<tr>
<td>64</td>
<td>M*K</td>
<td>8*8</td>
<td>0.169</td>
<td>0.173</td>
<td>0.17</td>
<td>0.172</td>
<td>0.167</td>
<td>0.170</td>
</tr>
<tr>
<td>128</td>
<td>M*K</td>
<td>8*8</td>
<td>0.429</td>
<td>0.445</td>
<td>0.436</td>
<td>0.435</td>
<td>0.446</td>
<td>0.438</td>
</tr>
<tr>
<td>256</td>
<td>M*K</td>
<td>8*8</td>
<td>2.355</td>
<td>2.367</td>
<td>2.366</td>
<td>2.372</td>
<td>2.385</td>
<td>2.36</td>
</tr>
<tr>
<td>512</td>
<td>M*K</td>
<td>8*8</td>
<td>17.794</td>
<td>17.581</td>
<td>17.795</td>
<td>17.551</td>
<td>17.609</td>
<td>17.66</td>
</tr>
</tbody>
</table>

Table A.4: Matrix-matrix multiplication simulation times for the Intel Xeon E5520 CPU (parallel using kernel function).

<table>
<thead>
<tr>
<th>M, N, K</th>
<th>Global-Size</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>M*K</td>
<td>16*16</td>
<td>1.219</td>
<td>1.211</td>
<td>1.207</td>
<td>1.224</td>
<td>1.214</td>
<td>1.215</td>
</tr>
<tr>
<td>64</td>
<td>M*K</td>
<td>16*16</td>
<td>1.214</td>
<td>1.22</td>
<td>1.219</td>
<td>1.218</td>
<td>1.214</td>
<td>1.217</td>
</tr>
<tr>
<td>128</td>
<td>M*K</td>
<td>16*16</td>
<td>1.282</td>
<td>1.273</td>
<td>1.267</td>
<td>1.263</td>
<td>1.285</td>
<td>1.274</td>
</tr>
<tr>
<td>256</td>
<td>M*K</td>
<td>16*16</td>
<td>1.633</td>
<td>1.619</td>
<td>1.626</td>
<td>1.629</td>
<td>1.619</td>
<td>1.625</td>
</tr>
<tr>
<td>512</td>
<td>M*K</td>
<td>16*16</td>
<td>4.044</td>
<td>4.04</td>
<td>4.043</td>
<td>4.1</td>
<td>4.062</td>
<td>4.057</td>
</tr>
</tbody>
</table>

Table A.5: Matrix-matrix multiplication simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel using kernel function).

A.2 Reduction

<table>
<thead>
<tr>
<th>N</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^17</td>
<td>0.601</td>
<td>0.531</td>
<td>0.530</td>
<td>0.530</td>
<td>0.530</td>
<td>0.544</td>
</tr>
<tr>
<td>2^18</td>
<td>1.065</td>
<td>1.059</td>
<td>1.058</td>
<td>1.061</td>
<td>1.058</td>
<td>1.06</td>
</tr>
<tr>
<td>2^19</td>
<td>2.141</td>
<td>2.131</td>
<td>2.139</td>
<td>2.221</td>
<td>2.131</td>
<td>2.152</td>
</tr>
<tr>
<td>2^21</td>
<td>8.713</td>
<td>8.578</td>
<td>8.531</td>
<td>8.581</td>
<td>8.583</td>
<td>8.597</td>
</tr>
<tr>
<td>2^22</td>
<td>17.429</td>
<td>17.166</td>
<td>17.132</td>
<td>16.949</td>
<td>17.156</td>
<td>17.166</td>
</tr>
</tbody>
</table>

Table A.6: Reduction simulation times for the Intel Xeon E5520 CPU (serial).

<table>
<thead>
<tr>
<th>N</th>
<th>Global-Size</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^17</td>
<td>N</td>
<td>8</td>
<td>0.368</td>
<td>0.380</td>
<td>0.374</td>
<td>0.379</td>
<td>0.374</td>
<td>0.375</td>
</tr>
<tr>
<td>2^18</td>
<td>N</td>
<td>8</td>
<td>0.668</td>
<td>0.682</td>
<td>0.656</td>
<td>0.687</td>
<td>0.682</td>
<td>0.675</td>
</tr>
<tr>
<td>2^19</td>
<td>N</td>
<td>8</td>
<td>1.245</td>
<td>1.222</td>
<td>1.213</td>
<td>1.233</td>
<td>1.23</td>
<td>1.228</td>
</tr>
<tr>
<td>2^20</td>
<td>N</td>
<td>8</td>
<td>2.253</td>
<td>2.288</td>
<td>2.246</td>
<td>2.243</td>
<td>2.297</td>
<td>2.265</td>
</tr>
<tr>
<td>2^22</td>
<td>N</td>
<td>8</td>
<td>8.715</td>
<td>8.626</td>
<td>8.571</td>
<td>8.626</td>
<td>8.659</td>
<td>8.639</td>
</tr>
</tbody>
</table>
Table A.7: Reduction simulation times for the Intel Xeon E5520 CPU (parallel).

<table>
<thead>
<tr>
<th>N</th>
<th>Global-Size</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{17}$</td>
<td>N</td>
<td>256</td>
<td>0.927</td>
<td>0.921</td>
<td>0.934</td>
<td>0.959</td>
<td>0.924</td>
<td>0.933</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>N</td>
<td>256</td>
<td>0.988</td>
<td>0.979</td>
<td>0.984</td>
<td>0.978</td>
<td>0.98</td>
<td>0.981</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>N</td>
<td>256</td>
<td>1.132</td>
<td>1.102</td>
<td>1.136</td>
<td>1.103</td>
<td>1.104</td>
<td>1.115</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>N</td>
<td>256</td>
<td>1.364</td>
<td>1.37</td>
<td>1.358</td>
<td>1.375</td>
<td>1.37</td>
<td>1.367</td>
</tr>
<tr>
<td>$2^{21}$</td>
<td>N</td>
<td>256</td>
<td>1.915</td>
<td>1.869</td>
<td>1.867</td>
<td>1.864</td>
<td>1.875</td>
<td>1.87</td>
</tr>
<tr>
<td>$2^{22}$</td>
<td>N</td>
<td>256</td>
<td>2.927</td>
<td>2.92</td>
<td>2.859</td>
<td>2.912</td>
<td>2.921</td>
<td>2.907</td>
</tr>
</tbody>
</table>

Table A.8: Reduction simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel).

A.3 Computation of Eigenvalues

<table>
<thead>
<tr>
<th>N</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.508</td>
<td>1.474</td>
<td>1.480</td>
<td>1.620</td>
<td>1.637</td>
<td>1.543</td>
</tr>
<tr>
<td>256</td>
<td>5.097</td>
<td>5.164</td>
<td>4.978</td>
<td>5.166</td>
<td>5.179</td>
<td>5.116</td>
</tr>
<tr>
<td>1024</td>
<td>52.746</td>
<td>52.047</td>
<td>52.635</td>
<td>52.258</td>
<td>52.625</td>
<td>52.462</td>
</tr>
<tr>
<td>2048</td>
<td>145.066</td>
<td>147.187</td>
<td>146.793</td>
<td>148.246</td>
<td>149.765</td>
<td>147.411</td>
</tr>
<tr>
<td>4096</td>
<td>363.939</td>
<td>362.812</td>
<td>363.266</td>
<td>362.581</td>
<td>362.976</td>
<td>363.114</td>
</tr>
<tr>
<td>8192</td>
<td>574.112</td>
<td>573.543</td>
<td>574.631</td>
<td>574.128</td>
<td>573.872</td>
<td>574.057</td>
</tr>
</tbody>
</table>

Table A.9: Computation of eigenvalues simulation times for the Intel Xeon E5520 CPU (serial).

<table>
<thead>
<tr>
<th>N</th>
<th>Global-Size</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>N</td>
<td>32</td>
<td>3.177</td>
<td>3.017</td>
<td>2.978</td>
<td>3.024</td>
<td>3.042</td>
<td>3.049</td>
</tr>
<tr>
<td>256</td>
<td>N</td>
<td>32</td>
<td>5.012</td>
<td>5.027</td>
<td>5.025</td>
<td>5.022</td>
<td>5.086</td>
<td>5.034</td>
</tr>
<tr>
<td>512</td>
<td>N</td>
<td>32</td>
<td>8.316</td>
<td>8.418</td>
<td>8</td>
<td>8.412</td>
<td>8.483</td>
<td>8.385</td>
</tr>
<tr>
<td>1024</td>
<td>N</td>
<td>32</td>
<td>23.299</td>
<td>23.385</td>
<td>23.398</td>
<td>23.612</td>
<td>23.374</td>
<td>23.413</td>
</tr>
<tr>
<td>2048</td>
<td>N</td>
<td>32</td>
<td>63.379</td>
<td>63.472</td>
<td>63.529</td>
<td>63.389</td>
<td>63.329</td>
<td>63.419</td>
</tr>
<tr>
<td>4096</td>
<td>N</td>
<td>32</td>
<td>144.588</td>
<td>144.887</td>
<td>144.783</td>
<td>144.619</td>
<td>144.862</td>
<td>144.747</td>
</tr>
<tr>
<td>8192</td>
<td>N</td>
<td>32</td>
<td>208.878</td>
<td>208.635</td>
<td>208.73</td>
<td>208.911</td>
<td>208.795</td>
<td>208.789</td>
</tr>
</tbody>
</table>

Table A.10: Computation of eigenvalues simulation times for the Intel Xeon E5520 CPU (parallel).
<table>
<thead>
<tr>
<th>N</th>
<th>Global-Size</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>N</td>
<td>128</td>
<td>7.185</td>
<td>7.224</td>
<td>7.172</td>
<td>7.189</td>
<td>7.171</td>
<td>7.188</td>
</tr>
<tr>
<td>512</td>
<td>N</td>
<td>256</td>
<td>7.319</td>
<td>7.434</td>
<td>7.374</td>
<td>7.405</td>
<td>7.334</td>
<td>7.373</td>
</tr>
<tr>
<td>1024</td>
<td>N</td>
<td>256</td>
<td>7.85</td>
<td>7.848</td>
<td>7.867</td>
<td>7.831</td>
<td>7.872</td>
<td>7.853</td>
</tr>
<tr>
<td>2048</td>
<td>N</td>
<td>256</td>
<td>8.721</td>
<td>8.670</td>
<td>8.676</td>
<td>8.678</td>
<td>8.73</td>
<td>8.695</td>
</tr>
<tr>
<td>4096</td>
<td>N</td>
<td>256</td>
<td>10.938</td>
<td>10.954</td>
<td>10.935</td>
<td>10.895</td>
<td>10.89</td>
<td>10.922</td>
</tr>
</tbody>
</table>

Table A.11: Computation of eigenvalues simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel).

---

### A.4 Stationary Heat Conduction

<table>
<thead>
<tr>
<th>N</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.896</td>
<td>2.045</td>
<td>1.929</td>
<td>1.898</td>
<td>2.025</td>
<td>1.958</td>
</tr>
<tr>
<td>256</td>
<td>7.613</td>
<td>7.904</td>
<td>7.947</td>
<td>8.173</td>
<td>7.881</td>
<td>7.903</td>
</tr>
<tr>
<td>512</td>
<td>31.566</td>
<td>32.494</td>
<td>32.489</td>
<td>32.436</td>
<td>31.537</td>
<td>32.104</td>
</tr>
<tr>
<td>1024</td>
<td>122.631</td>
<td>123.418</td>
<td>122.517</td>
<td>122.792</td>
<td>122.413</td>
<td>122.754</td>
</tr>
<tr>
<td>2048</td>
<td>486.696</td>
<td>488.541</td>
<td>487.732</td>
<td>487.390</td>
<td>486.352</td>
<td>487.342</td>
</tr>
</tbody>
</table>

Table A.12: Stationary heat conduction simulation times for the Intel Xeon E5520 CPU (serial).

<table>
<thead>
<tr>
<th>N</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>8</td>
<td>1.582</td>
<td>1.575</td>
<td>1.558</td>
<td>1.594</td>
<td>1.567</td>
<td>1.575</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>5.361</td>
<td>5.443</td>
<td>5.472</td>
<td>5.387</td>
<td>5.506</td>
<td>5.433</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
<td>20.73</td>
<td>20.914</td>
<td>19.92</td>
<td>20.418</td>
<td>20.204</td>
<td>20.437</td>
</tr>
<tr>
<td>1024</td>
<td>8</td>
<td>81.684</td>
<td>81.311</td>
<td>78.910</td>
<td>79.652</td>
<td>81.116</td>
<td>80.534</td>
</tr>
<tr>
<td>2048</td>
<td>8</td>
<td>319.238</td>
<td>318.915</td>
<td>319.527</td>
<td>319.681</td>
<td>319.531</td>
<td>319.378</td>
</tr>
</tbody>
</table>

Table A.13: Stationary heat conduction simulation times for the Intel Xeon E5520 CPU (parallel using parfor).

<table>
<thead>
<tr>
<th>N</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>512</td>
<td>2.827</td>
<td>2.818</td>
<td>2.814</td>
<td>2.824</td>
<td>2.813</td>
<td>2.819</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>11.103</td>
<td>11.082</td>
<td>11.068</td>
<td>11.07</td>
<td>11.076</td>
<td>11.079</td>
</tr>
<tr>
<td>2048</td>
<td>512</td>
<td>168.011</td>
<td>168.128</td>
<td>168.048</td>
<td>168.210</td>
<td>168.256</td>
<td>168.130</td>
</tr>
</tbody>
</table>

Table A.14: Stationary heat conduction simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel using parfor).
<table>
<thead>
<tr>
<th>N</th>
<th>Global-Size</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>N*N</td>
<td>4*4</td>
<td>0.966</td>
<td>0.955</td>
<td>0.957</td>
<td>0.964</td>
<td>0.954</td>
<td>0.959</td>
</tr>
<tr>
<td>256</td>
<td>N*N</td>
<td>4*4</td>
<td>1.874</td>
<td>1.877</td>
<td>1.865</td>
<td>1.893</td>
<td>1.867</td>
<td>1.875</td>
</tr>
<tr>
<td>512</td>
<td>N*N</td>
<td>4*4</td>
<td>5.455</td>
<td>5.496</td>
<td>5.503</td>
<td>5.488</td>
<td>5.501</td>
<td>5.488</td>
</tr>
<tr>
<td>2048</td>
<td>N*N</td>
<td>4*4</td>
<td>76.020</td>
<td>76.204</td>
<td>76.008</td>
<td>76.147</td>
<td>76.008</td>
<td>76.077</td>
</tr>
</tbody>
</table>

**Table A.15:** Stationary heat conduction simulation times for the Intel Xeon E5520 CPU (parallel using kernel function).

<table>
<thead>
<tr>
<th>N</th>
<th>Global-Size</th>
<th>Local-Size</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Run 5</th>
<th>Average</th>
</tr>
</thead>
</table>

**Table A.16:** Stationary heat conduction simulation times for the NVIDIA Fermi-Tesla M2050 GPU (parallel using kernel function).
Appendix B

Modelica Models Source Code

B.1 Sequential Implementation

B.1.1 Matrix-Matrix Multiplication

Listing B.1: Matrix-matrix multiplication.

```modelica
function initialize
  input Integer m;
  input Integer n;
  input Integer k;
  output Real A[m,n];
  output Real B[n,k];

algorithm
  for i in 1:m loop
    for j in 1:n loop
      A[i,j] := j;
    end for;
  end for;
  for j in 1:n loop
    for h in 1:k loop
      B[j,h] := h;
    end for;
  end for;
end initialize;

function matrixMultiply
  input Integer m;
  input Integer p;
  input Integer n;
  input Real A[m,p];
```
input Real B[p,n];
output Real C[m,n];

Real localtmp;

algorithm
for i in 1:m loop
  for j in 1:n loop
    localtmp := 0;
    for k in 1:p loop
      localtmp := localtmp + (A[i,k] * B[k,j]);
    end for;
    C[i,j] := localtmp;
  end for;
end for;
end matrixMultiply;

//---------------------------
function mainF
input Integer m;
input Integer n;
input Integer k;
output Real result;

Real A[m,n];
Real B[n,k];
Real C[m,k];

algorithm
  //initialize matrix A, and B
  (A,B) := initialize(m,n,k);
  //multiply matrices A and B
  C := matrixMultiply(m,n,k,A,B);
  //only one item is returned to speed up
  //the compile time
  result := C[m,k];
end mainF;
//---------------------------
model MatrixMultiplication
parameter Integer m=256,n=256,k=256;
Real result;

algorithm
  result := mainF(m,n,k);
end MatrixMultiplication;
B.1.2 Reduction

Listing B.2: Reduction.

```plaintext
function initialize
    input Integer n;
    output Real A[n];
    Integer init;
algorithm
    for i in 1:n loop
        init := integer((3125 * i)/65536);
        A[i] := (init - 32768.0)/16384.0;
    end for;
end initialize;

// ---------------
function reduce
    input Integer n;
    input Real A[n];
    output Real result;
algorithm
    result := 0;
    for i in 1:n loop
        result := result + A[i];
    end for;
end reduce;

// --------------
function mainF
    input Integer n;
    output Real result;
    Real A[n];
algorithm
    //initialize array A
    A := initialize(n);
    //compute the sum of array
    result := reduce(n,A);
end mainF;

// ----------------------
model Reduction
    parameter Integer n=2048;
    Real result;
algorithm
    result := mainF(n);
end Reduction;
```
B.1.3 Computation of Eigenvalues

Listing B.3: Computation of eigenvalues.

function initialize
  input Integer length;
  output Real diagonal[length];
  output Real offDiagonal[length-1];

  Real init;

algorithm
  for i in 1:length loop
    init := integer((3125 * i)/65536);
    diagonal[i] := (init - 32768.0)/16384.0;
  end for;

  for i in 1:length-1 loop
    init := integer((3125 * i)/65536);
    offDiagonal[i] := (init - 20423.0)/16384.0;
  end for;
end initialize;

// -----------------
function isComplete
  input Integer length;
  input Real tolerance;
  input Real eigenIntervals[:];
  output Integer result=0;

  Integer lid, uid;

algorithm
  for i in 1:length loop
    lid := (2*i) - 1;
    uid := lid + 1;
    if(eigenIntervals[uid] - eigenIntervals[lid] >= tolerance) then
      result := 1;
      break;
    end if;
  end for;
end isComplete;

// -----------------------------------
function computeGerschgorinInterval
  input Integer length;
  input Real diagonal[:];
  input Real offDiagonal[:];
  output Real lowerBound;
  output Real upperBound;

  Real r;
Algorithm:

\[
\text{lowerBound} := \text{diagonal}[1] - \text{abs} (\text{offDiagonal}[1]);
\]
\[
\text{upperBound} := \text{diagonal}[1] + \text{abs} (\text{offDiagonal}[1]);
\]

for \( i \) in 2: length -1 loop

\[
r := \text{abs} (\text{offDiagonal}[i-1]) + \text{abs} (\text{offDiagonal}[i]);
\]
if ( \( \text{lowerBound} > (\text{diagonal}[i] - r) \)) then

\[
\text{lowerBound} := \text{diagonal}[i] - r;
\]
end if;

if ( \( \text{upperBound} < (\text{diagonal}[i] + r) \)) then

\[
\text{upperBound} := \text{diagonal}[i] + r;
\]
end if;
end for;

if ( \( \text{lowerBound} > (\text{diagonal[length]} - \text{abs} (\text{offDiagonal[length-1]})) \)) then

\[
\text{lowerBound} := \text{diagonal[length]} - \text{abs} (\text{offDiagonal[length-1]});
\]
end if;

if ( \( \text{upperBound} < (\text{diagonal[length]} + \text{abs} (\text{offDiagonal[length-1]})) \)) then

\[
\text{upperBound} := \text{diagonal[length]} + \text{abs} (\text{offDiagonal[length-1]});
\]
end if;

end computeGerschgorinInterval;

// -----------------------------
function caleigenvalue

input Real tolerance;
input Integer length;
input Real diagonal[:];
input Real offDiagonal[:];
input Real eigenIntervals[:];
output Real newEigenIntervals[2*length];

algorithm

offset := 0;
for \( i \) in 1: length loop

\[
lid := (2*i) - 1;
\]
\[
uid := lid + 1;
\]

// compute the number of eigenvalue in an interval
\[
\text{lowerBound} := \text{eigenIntervals}[lid];
\]
\[
\text{upperBound} := \text{eigenIntervals}[uid];
\]
\[
\text{eigenValuesLessLowerBound} := \text{calNumEigenValuesLessThan}(\text{length}, \text{diagonal}, \text{offDiagonal}, \text{lowerBound});
\]
\[
\text{eigenValuesLessUpperBound} := \text{calNumEigenValuesLessThan}(\text{length}, \text{diagonal}, \text{offDiagonal}, \text{upperBound});
\]

end caleigenvalue;
numSubIntervals := eigenValuesLessUpperBound -
eigenValuesLessLowerBound;

// if the number of eigenvalues in an interval is 0
// Discards the intervals.

// if the number of eigenvalues in an interval is 1, it splits
//(bisects) that interval into two half and considers the one
// in which the eigenvalue exists.
// No bisection is done when the interval length is less than
// a given tolerance (a desired accuracy)
if (numSubIntervals == 1) then
    mid := (lowerBound + upperBound) / 2;
    n := calNumEigenValuesLessThan(length, diagonal, offDiagonal, mid) -
        eigenValuesLessLowerBound;
    newLid := (2 * offset) + 1;
    newUid := newLid + 1;

    // check if the interval size is less than tolerance levels
    if (upperBound - lowerBound < tolerance) then
        newEigenIntervals[newLid] := lowerBound;
        newEigenIntervals[newUid] := upperBound;
    // the eigenvalue lies in the right
    // half of the interval
    elseif (n == 0) then
        newEigenIntervals[newLid] := mid;
        newEigenIntervals[newUid] := upperBound;
    // the eigenvalue lies in the left half
    // of the interval
    else
        newEigenIntervals[newLid] := lowerBound;
        newEigenIntervals[newUid] := mid;
    end if;

// if the number of eigenvalues is more than 1
// the interval is split into equal intervals
// of size divisionWidth
elseif (numSubIntervals > 1) then
    divisionWidth := (eigenIntervals[uid] -
        eigenIntervals[lid]) / numSubIntervals;

    for j in 1..numSubIntervals loop
        newLid := (2 * (offset + j)) - 1;
        newUid := newLid + 1;
        newEigenIntervals[newLid] := eigenIntervals[lid] +
            (j - 1) * divisionWidth;
        newEigenIntervals[newUid] := newEigenIntervals[newLid] +
            divisionWidth;
    end for;
offset := offset + numSubIntervals;
end for;
end caleigenvalue;

// --------------------------------
function calNumEigenValuesLessThan
input Integer length;
input Real diagonal[:];
input Real offDiagonal[:];
input Real x;
output Integer count;

Real prev_diff , diff;
algorithm
count :=0;
prev_diff := diagonal[1] - x;
if( prev_diff < 0) then
  count := count +1;
end if;
for i in 2:length loop
  diff :=(diagonal[i]-x) - ((offDiagonal[i-1]* offDiagonal[i-1])/ prev_diff);
  if( diff < 0) then
    count := count +1;
  end if;
  prev_diff := diff;
end for;
end calNumEigenValuesLessThan;

// ---------------------------
function mainF
input Integer length;
input Real tolerance;
output Real result;

Real diagonal[length];
Real offDiagonal[length-1];
Real EigenIntervals1[2*length];
Real EigenIntervals2[2*length];
Real EigenIntervals3[2*length];
Real lowerBound, upperBound;
Integer In;

algorithm
// initialize diagonal and offdiagonal
(diagonal, offDiagonal) := initialize(length);

// calculate the upperbound and the lowerbound
// of the eigenvalues of the matrix
(lowerBound, upperBound) := computeGerschgorinInterval(length,
diagonal,
offDiagonal);

//initialize the eigenvalue intervals
EigenIntervals1[1] := lowerBound;

//the following intervals have no eigenvalues in first run
//but afterwards the initial Gerschgorin Interval will be
//divided into some sub-intervals which may have eigenvalues.
for i in 3:(2*length) loop
  EigenIntervals1[i] := upperBound;
end for;

In := 0;
EigenIntervals3 := EigenIntervals1;

//Checks if the difference between lowerBound and upperBound
//of all intervals is below tolerance levels(desired accuracy),
//if all are below tolerance then all eigevalue based on the
//desired accuracy are obtained.
while (isComplete(length,tolerance,EigenIntervals3) == 1) loop
  if(In == 0) then
    EigenIntervals2 := calEigenvalue(tolerance,length,
                                   diagonal,offDiagonal,
                                   EigenIntervals1);
    EigenIntervals3 := EigenIntervals2;
  elseif (In == 1) then
    EigenIntervals1 := calEigenvalue(tolerance,length,
                                    diagonal,offDiagonal,
                                    EigenIntervals2);
    EigenIntervals3 := EigenIntervals1;
  end if;
  In := 1 - In;
end while;

//only one element is returned to speed up the compile time
result := EigenIntervals3[length];

end mainF;

---------
model Eigenvalue
parameter Integer length=2048;//should be power of 2
Real tolerance = 0.001;
Real result;

algorithm
result := mainF(length,tolerance);
end Eigenvalue;
B.1.4 Stationary Heat Conduction

Listing B.4: Stationary heat conduction.

```plaintext
function laplsolve
  input Integer n;
  input Integer maxiter;
  output Real T[n+2,n+2];

  Real toprow[n];
  Real tmp, left;

algorithm

  // Set boundary conditions and initial values
  for i in 1:n+2 loop
    for j in 2:n+1 loop
      T[i,j] := 0.0;
    end for;
    T[i,1] := 1.0;
    T[i,n+2] := 1.0;
  end for;
  for j in 1:n+2 loop
    T[n+2,j] := 2.0;
  end for;

  // Solve the linear system of equations
  // using the Jacobi method
  for k in 1:maxiter loop
    for j in 2:n+1 loop
      toprow[j-1] := T[1,j];
    end for;
    for i in 2:n+1 loop
      left := T[i,1];
      for j in 2:n+1 loop
        tmp := T[i,j];
        T[i,j] := (toprow[j-1] + T[i+1,j] + T[i,j+1] + left)/4.0;
        left := tmp;
        toprow[j-1] := tmp;
      end for;
    end for;
  end for;
end laplsolve;

function mainF
  input Integer n;
  output Real result = 0;
  Integer maxiter;
  Real T[n+2,n+2];

algorithm
```

73
maxiter := 20;
T := laplsolve(n,maxiter);
result := T[2,2];

end mainF;
//----------------------
model SHeatConduction
  parameter Integer n=512;
  Real result;
algorithm
  result := mainF(n);
end SHeatConduction;

B.2 Parallel Implementation

B.2.1 Matrix-Matrix Multiplication

Listing B.5: Matrix-matrix multiplication.

function initialize
  input Integer m;
  input Integer n;
  input Integer k;
  parglobal input Integer pn;
  parglobal input Integer pk;
  parglobal output Real pA[m,n];
  parglobal output Real pB[n,k];

  Real A[m,n];
  Real B[n,k];
algorithm
  parfor i in 1:m loop
    for j in 1:pn loop
      pA[i,j] := j;
    end for;
    end parfor;

  parfor j in 1:n loop
    for h in 1:pk loop
      pB[j,h] := h;
    end for;
    end parfor;
end initialize;
//--------------------------
parkernel function matrixmultiply
  parglobal input Integer pn;
parGlobal input Real pA[:,];
parGlobal input Real pB[:,];
parGlobal output Real pC[size(pA,1),size(pB,2)];

Real plocaltmp;
Integer i,j;

algorithm
// Returns the unique global thread Id value
// for first and second dimension
i := oclGetGlobalId(1);
j := oclGetGlobalId(2);
plocaltmp := 0;
for h in 1:pn loop
  plocaltmp := plocaltmp + (pA[i,h] * pB[h,j]);
end for;
pC[i,j] := plocaltmp;
end matrixmultiply;
//-------------------

function mainF
input Integer m;
input Integer n;
input Integer k;
output Real result ;

Real C[m,k];

parGlobal Real pA[m,n];
parGlobal Real pB[n,k];
parGlobal Real pC[m,k];
parGlobal Integer pm;
parGlobal Integer pn;
parGlobal Integer pk;

//the total number of global threads which execute
//the kernel in parallel
Integer globalSize[2] = {m,k};

//the total number of local threads which run
//in parallel in each work-group
Integer localSize[2] = {16,16};

algorithm
//copy from host to device
pm := m;
pn := n;
pk := k;
(pA,pB) := initialize(m,n,k,pn,pk);

//specify the number of threads and work-groups
// to be used for a kernel function execution
oclSetNumThreads(globalSize, localSize);

pC := matrixMultiply(pn, pA, pB);

// copy from device to host
C := pC;

result := C[m, k];

// set the number of threads to the available number
// supported by device
oclSetNumThreads(0);

d end mainF;

// -------------------------
model MatrixMultiplicationP
parameter Integer m=32, n=32, k=32;
Real result;
algorithm
result := mainF(m, n, k);
ed MatrixMultiplicationP;

B.2.2 Reduction

Listing B.6: Reduction.

function initialize
input Integer m;
parglobal output Real pA[m];
parglobal Integer pinit;
algorithm
parfor i in 1:m loop
pinit := integer((3125 * i) / 65536);
pA[i] := (pinit - 32768.0) / 16384.0;
end parfor;
end initialize;

// ------------------------
parkernel function reduce
parglobal input Real pA[:];
parglobal output Real pC[size(pA, 1)];
parglobal output Integer tmp;

// declare shared memory
parlocal Real sdata[oclGetLocalSize(1)];
Integer s, localSize;
```plaintext
Integer localThreadId, workGroupId, globalThreadId;

algorithm

// returns the size of the block
localSize := oclGetLocalSize(1);

// returns the unique global thread Id value
// for first dimension
globalThreadId := oclGetGlobalId(1);

// returns the current work-group Id
workGroupId := oclGetGroupId(1);

// returns the unique local thread Id value
// for first dimension
localThreadId := oclGetLocalId(1);

// copy from global memory to shared memory
sdata[localThreadId] := pA[globalThreadId];

// synchronize shared memory
oclLocalBarrier(0);

s := integer(localSize/2);
while(s > 0) loop
   // perform reduction in shared memory
   if ((localThreadId <= s)) then
      sdata[localThreadId] := sdata[localThreadId] +
      sdata[localThreadId + s];
   end if;

   // synchronize shared memory
   oclLocalBarrier(0);
   s := integer(s / 2);
end while;

// copy the result of the block to global memory
if (localThreadId == 1) then
   pC[workGroupId] := sdata[localThreadId];
end if;

end reduce;

// --------------

function mainF
input Integer m;
output Real result;

// the total number of global threads which execute
// the kernel in parallel
Integer globalSize[1] = {m};
```

// the total number of local threads which run
// in parallel in each work-group
Integer localSize[1] = 128;

Real A[m];
parallel Real pA[m];

algorithm
    pA := initialize(m);
    while (globalSize[1] > localSize[1]) loop
        oclSetNumThreads(globalSize, localSize);
        pA := reduce(pA);
        globalSize[1] := integer(globalSize[1]/localSize[1]);
    end while;
    A := pA;
    result := 0;
    for i in 1:globalSize[1] loop
        result := result + A[i];
    end for;
    oclSetNumThreads(0);
end mainF;

// ---------------
model ReductionP
    parameter Integer m=131072; // should be power of 2
    Real result;
    algorithm
    result := mainF(m);
end ReductionP;

B.2.3 Computation of Eigenvalue

Listing B.7: Computation of eigenvalue.

function initialize
    input Integer length;
    parglobal output Real pdiagonal[length];
    parglobal output Real poffDiagonal[length-1];
    parglobal Real pinit;
    algorithm
parfor i in 1:length loop
  pinit := integer((3125 * i)/65536);
pdiagonal[i] := (pinit - 32768.0)/16384.0;
end parfor;
parfor i in 1:length-1 loop
  pinit := integer((3125 * i)/65536);
poffDiagonal[i] := (pinit - 20423.0)/16384.0;
end parfor;
end initialize;

// ---------------
function isComplete
  input Integer length;
  input Real tolerance;
  input Real eigenIntervals[:];
  output Integer result =0;
Integer lid, uid;
algorithm
  for i in 1:length loop
    lid := (2*i) - 1;
    uid := lid + 1;
    if(eigenIntervals[uid] -
eigenIntervals[lid] >= tolerance) then
      result := 1;
      break;
    end if;
  end for;
end isComplete;

-------------------
function computeGerschgorinInterval
  input Integer length;
  input Real diagonal[:];
  input Real offDiagonal[:];
  output Real lowerBound;
  output Real upperBound;
Real r;
algorithm
  lowerBound := diagonal[1] - abs(offDiagonal[1]);
  upperBound := diagonal[1] + abs(offDiagonal[1]);
  for i in 2:length-1 loop
    r := abs(offDiagonal[i-1]) + abs(offDiagonal[i]);
    if (lowerBound > (diagonal[i] - r)) then
      lowerBound := diagonal[i] - r;
    end if;
    if (upperBound < (diagonal[i] + r)) then
      upperBound := diagonal[i] + r;
    end if;
  end for;
upperBound := diagonal[i] + r;
end if;
end for;

if (lowerBound > (diagonal[length] - abs(offDiagonal[length-1]))) then
    lowerBound := diagonal[length] - abs(offDiagonal[length-1]);
end if;

if (upperBound < (diagonal[length] + abs(offDiagonal[length-1]))) then
    upperBound := diagonal[length] + abs(offDiagonal[length-1]);
end if;

end computeGerschgorinInterval;

parkernel function calNumEigenValueInterval
parglobal input Integer length;
parglobal input Real diagonal[:];
parglobal input Real offDiagonal[:];
parglobal input Real eigenIntervals[:];
parglobal output Integer numEigenIntervals[oclGetGlobalSize(1)];
parglobal output Integer tmp;

Integer globalThreadId, lowerId, upperId;
Integer eigenValuesLessLowerBound, eigenValuesLessUpperBound;
Real lowerBound, upperBound;

algorithm

// Returns the unique global thread Id value
// for first dimension
globalThreadId := oclGetGlobalId(1);
lowerId := (2*globalThreadId) -1;
upperId := lowerId + 1;
lowerBound := eigenIntervals[lowerId];
upperBound := eigenIntervals[upperId];
eigenValuesLessLowerBound := numEigenValuesLessThan(length, diagonal, offDiagonal, lowerBound);
eigenValuesLessUpperBound := numEigenValuesLessThan(length, diagonal, offDiagonal, upperBound);

// Store the number of eigenvalue of each interval
// for each thread
numEigenIntervals[globalThreadId] := eigenValuesLessUpperBound - eigenValuesLessLowerBound;

end calNumEigenValueInterval;

parkernel function recalculateEigenIntervals
parglobal input Real tolerance;

80
parglobal input Integer length;
parglobal input Real diagonal[:];
parglobal input Real offDiagonal[:];
parglobal input Real eigenIntervals[:];
parglobal input Integer numEigenIntervals[:];
parglobal output Real newEigenIntervals[size(eigenIntervals,1)];
parglobal output Integer tmp;

Integer globalThreadId, currentindx, indx, lowerId, upperId, lId, uId;
Real divisionWidth, midValue, lowerBound, upperBound, n;

algorithm

// Returns the unique global thread Id value
// for first dimension
globalThreadId := oclGetGlobalId(1);
currentindx := globalThreadId - 1;
lowerId := (2 * globalThreadId) - 1;
upperId := lowerId + 1;

indx := 1;
while (currentindx >= numEigenIntervals[indx]) loop
  currentindx := currentindx - numEigenIntervals[indx];
  indx := indx + 1;
end while;

lId := (2 * indx) - 1;
uId := lId + 1;

lowerBound := eigenIntervals[lId];
upperBound := eigenIntervals[uId];

if (numEigenIntervals[indx] == 1) then
  midValue := (upperBound + lowerBound) / 2;
  n := numEigenValuesLessThan(length, diagonal, offDiagonal, midValue) -
      numEigenValuesLessThan(length, diagonal, offDiagonal, lowerBound);
  if (upperBound - lowerBound < tolerance) then
    newEigenIntervals[lowerId] := lowerBound;
    newEigenIntervals[upperId] := upperBound;
  elseif (n == 0) then
    newEigenIntervals[lowerId] := midValue;
    newEigenIntervals[upperId] := upperBound;
  else
    newEigenIntervals[lowerId] := lowerBound;
    newEigenIntervals[upperId] := midValue;
  end if;

// split the intervals into equal intervals
// of size divisionWidth
elseif (numEigenIntervals[indx] > 1) then

\[
\text{divisionWidth} := \frac{\text{upperBound} - \text{lowerBound}}{\text{numEigenIntervals}[\text{indx}]};
\]

newEigenIntervals[lowerId] := lowerBound + \text{divisionWidth}*\text{currentindx};

newEigenIntervals[upperId] := newEigenIntervals[lowerId] + divisionWidth;

end if;

end recalculateEigenIntervals;

// ---------------------------------------
parallel function numEigenValuesLessThan
parglobal input Integer length;
parglobal input Real diagonal[:];
parglobal input Real offDiagonal[:];
parglobal input Real x;
parglobal output Integer count;

Real prev_diff, diff;
Integer dummy = 1;

algorithm
  count := 0;
  prev_diff := diagonal[dummy] - x;
  if(prev_diff < 0) then
    count := count + 1;
  end if;

for i in 2:length loop
  diff := (diagonal[i]-x) - ((offDiagonal[i-1] * (offDiagonal[i-1]))/prev_diff);
  if(diff < 0) then
    count := count + 1;
  end if;
  prev_diff := diff;
end for;

end numEigenValuesLessThan;

// ---------------------------------------
function mainF
input Integer length;
input Real tolerance;
output Real result;

Real diagonal[length];
Real offDiagonal[length-1];
Real EigenIntervals1[2*length];
Real EigenIntervals2[2*length];
Real EigenIntervals3[2*length];
Real lowerBound, upperBound;
Integer In;

// the total number of global threads which execute
// the kernel in parallel
Integer globalSize[1] = {length};

// the total number of local threads which run
// in parallel in each work-group
Integer localSize[1] = {128};

parglobal Integer plength;
parglobal Real ptolerance,pupperBound;
parglobal Integer pnumEigenIntervals[length];
parglobal Real pdiagonal[length];
parglobal Real poffDiagonal[length-1];
parglobal Real pEigenIntervals1[2*length];
parglobal Real pEigenIntervals2[2*length];
parglobal Real pEigenIntervals3[2*length];

algorithm

plength := length;
ptolerance := tolerance;

// initialize diagonal and offdiagonal
(pdiagonal, poffDiagonal) := fillrandom(length);
diagonal := pdiagonal;
offDiagonal := poffDiagonal;

(lowerBound,upperBound) := computeGerschgorinInterval(length, diagonal, offDiagonal);

// specify the number of threads and work-groups
// to be used for a kernel function execution
oclSetNumThreads(globalSize,localSize);

In := 0;
EigenIntervals1[1] := lowerBound;

for i in 3:(2*length) loop
    EigenIntervals1[i] := upperBound;
end for;

EigenIntervals3 := EigenIntervals1;
pEigenIntervals3 := EigenIntervals1;
pEigenIntervals1 := EigenIntervals1;

while (isComplete(length,tolerance,EigenIntervals3) == 1) loop

    // The kernel will be executed by all threads, so that the
    // number of eigenvalues in each interval which is mapped to
    // a specific thread will be calculated

end while;
pnumEigenIntervals := calNumEigenValueInterval(plength,
pdiagonal,
poffDiagonal,
pEigenIntervals3);

if(In == 0) then
    //The kernel will be executed by all threads, and each will
    //recalculate its own interval
    pEigenIntervals2 := recalculateEigenIntervals(ptolerance,
            plength,
pdiagonal,
poffDiagonal,
pEigenIntervals1,
pnumEigenIntervals);

    pEigenIntervals3 := pEigenIntervals2;
elseif(In == 1) then
    //The kernel will be executed by all threads, and each will
    //recalculate its own interval
    pEigenIntervals1 := recalculateEigenIntervals(ptolerance,
            plength,
pdiagonal,
poffDiagonal,
pEigenIntervals2,
pnumEigenIntervals);

    pEigenIntervals3 := pEigenIntervals1;
end if;

EigenIntervals3 := pEigenIntervals3;
In := 1 - In;
end while;

result := EigenIntervals3[length];

//set the number of threads to the available number
//supported by device
oclSetNumThreads(0);
end mainF;

//--------------------
model EigenvalueP
    parameter Integer length=128;
    Real tolerance = 0.001;
    Real result;

algorithm
    result := mainF(length,tolerance);
end EigenvalueP;
B.2.4 Stationary Heat conduction

Listing B.8: Stationary heat conduction.

```c
parkernel function heatPlate2D
    parglobal input Integer n;
    parglobal input Real T[:,,:];
    parglobal output Real T2[size(T,1), size(T,2)];
    parglobal output Integer tmp;

    Integer i,j;

algorithm
    // Returns the unique global thread Id value
    // for first and second dimension
    i := oclGetGlobalId(1) + 1;
    j := oclGetGlobalId(2) + 1;

    // Set boundary conditions and initial values
    // the first row and column will be initialized
    // by one thread, and the last row and column
    // will be initialized by another thread.

    // fill columns 2 up to n+1 of first row
    if(i == 2) then
        T2[1,j]:=0.0;
    end if;

    // fill columns 2 up to n+1 of last row
    if(i == n+1) then
        T2[n+2,j]:=2.0;
    end if;

    // fill rows 2 up to n+1 of first column
    if(j == 2) then
        T2[i,1]:=1.0;
    end if;

    // fill rows 2 up to n+1 of last column
    if(j == n+1) then
        T2[i,n+2]:=1.0;
    end if;

    // synchronization
    oclGlobalBarrier(0);

    // each element will be calculated in each instance
    // of the kernel by a specific thread
    T2[i,j]:=(T[i-1,j]+T[i+1,j]+T[i,j+1]+T[i,j-1])/4.0;
end heatPlate2D;
```

// ----------------

function laplsolve
input Integer n;
input Integer maxiter;
input Integer globalSize[:];
input Integer localSize[:];
output Real T[n+2,n+2];

parglobal Real pT[n+2,n+2];
parglobal Integer pmaxiter,pn;

algorithm
pn := n;
pmaxiter := maxiter;

parfor i in 1:n+2 loop
for j in 2:pn+1 loop
pT[i,j]:=0.0;
end for;
pT[i,1] := 1.0;
pT[i,pn+2] := 1.0;
end parfor;

parfor i in 1:n+2 loop
pT[pn+2,i] := 2.0;
end parfor;

// specify the number of threads and work-groups
// to be used for a kernel function execution
oclSetNumThreads (globalSize, localSize);

for k in 1:maxiter loop
pT := heatPlate2D(pn,pT);
end for;
T := pT;

// set the number of threads to the available number
// supported by device
oclSetNumThreads (0);
end laplsolve;

// ---------------
function mainF
input Integer n;
output Real result=0;

// the total number of global threads which execute
// the kernel in parallel
Integer globalSize[2] = {n,n};

// the total number of local threads which run
// in parallel in each work-group
Integer localSize[2] = {16,16};

Integer maxiter;
Real T[n+2,n+2];

algorithm
maxiter := 20;
T := laplsolve(n,maxiter,globalSize,localSize);
result := T[2,2];
end mainF;

model HeatEquationSolverP
parameter Integer n=2048;
Real result;

algorithm
result := mainF(n);
end HeatEquationSolverP;
Bibliography


På svenska

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

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

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

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

In English

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

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

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Afshin Hemmati Moghadam