High Performance Cortical Simulator

Konstantin Kudryavtsev

August 19, 2011

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2011
Abstract

Neural simulators are an extremely computationally intensive class of software applications. Most such applications have been initially developed as serial code with some of them later parallelised using various HPC libraries and techniques to allow faster, larger and hence more realistic simulations. Topographica is one such application. The software, initially serial, has been developed in Python. Parts of the simulator have been recently parallelised with MPI, but the initial parallel version of the code had poor scaling properties. Here we find the most widely used and computationally intensive components of the simulator and demonstrate what performance should be possible to achieve using MPI by modelling these components with highly efficient C code. Using this performance benchmark we optimised Topographica to improve its scaling properties.
Contents

Table of Contents

Chapter 1: Introduction........................................................................................................... 1

Chapter 2: Background........................................................................................................... 3
  2.1 The Topographica Project............................................................................................... 3
    2.1.1 The Glossary of terms............................................................................................. 3
    2.1.2 Components of the simulator................................................................................ 5
  2.2 Tools............................................................................................................................... 6
    2.2.1 Mpi4py..................................................................................................................... 6
    2.2.2 PMI.......................................................................................................................... 6
  2.3 The First Parallel Version............................................................................................. 7
  2.4 The Approach............................................................................................................... 8
    2.4.1 Divide and Conquer............................................................................................... 8
    2.4.2 Abstraction............................................................................................................. 9
    2.4.3 Hand-Crafted Profiling.......................................................................................... 9
  2.5 Communication............................................................................................................ 10

Chapter 3: Systems............................................................................................................. 11
  3.1 Jupiter.......................................................................................................................... 11
    3.1.1 Hardware............................................................................................................... 11
    3.1.2 Software................................................................................................................ 11
  3.2 HECToR....................................................................................................................... 12
3.2.1 HECToR Phase 2b hardware characteristics .................................................. 12
3.2.2 Software ........................................................................................................... 12
3.2.3 Modules used in this project ........................................................................... 12
Chapter 4: Implementation .................................................................................... 14
  4.1 Bare C .................................................................................................................. 14
  4.2 Matchmodel ...................................................................................................... 16
  4.3 Plotting and Timing .......................................................................................... 16
  4.4 Optimised PMI .................................................................................................. 17
  4.5 Scripting ............................................................................................................ 17
Chapter 5: Bare C Performance Analysis ............................................................... 18
  5.1 Jupiter ................................................................................................................ 18
  5.2 HECToR ............................................................................................................ 21
    5.2.1 Shared Memory Run ...................................................................................... 21
    5.2.2 Distributed Memory Run: 4 nodes ............................................................... 24
    5.2.3 Job Submission Overhead ........................................................................... 26
    5.2.4 Distributed Memory Run, 7 nodes ............................................................... 29
    5.2.5 Distributed Memory Run on 15 Nodes ....................................................... 31
  5.3 Conclusion ........................................................................................................ 34
Chapter 6: Performance Analysis of Topographica ............................................... 35
  6.1 HECToR, Single Node SMP Run, C VS Python ................................................ 35
  6.2 HECToR, Multiple Node Run, C VS Python ..................................................... 37
  6.3 Detailed Topographica Timings ........................................................................ 38
  6.4 Performance of the Optimised Topographica .................................................. 42
  6.5 Conclusion ........................................................................................................ 43
Chapter 7: Development Issues and Discussion ................................................... 44
  7.1 First Steps ......................................................................................................... 44
  7.2 Bare C: Matching the Topographica Simulations in Bare C ............................ 47
7.3 Timing..............................................................................................................................................48
7.4 Building and Running Topographica on HECToR.................................................................49
7.5 Problems with Eddie ECDF........................................................................................................50
7.6 Optimising Weights Distribution.............................................................................................50
7.7 PMI Optimisations.....................................................................................................................51
Chapter 8: Conclusions..................................................................................................................54
  8.1 Future Work................................................................................................................................55

Appendix A: Output for the Mpi4py Collective Routines Test.....................................................57
Appendix B: Serial Topographica Profiling Results.....................................................................59
References ..........................................................................................................................................60
List of Tables

Index of Tables

Table 1: Bare C parallel efficiency values, 15-node MPP, subscribing 4 cores per node .................................................................34

Table 2: Bare C and Topographica parallel efficiency comparison, SMP....................37

Table 3: Optimised and non-optimised Topographica parallel efficiency comparison..44
List of Figures

Components' scaling on Jupiter. Optimised weights distribution.................................19
Components' scaling on Jupiter, optimised weights distribution.................................20
Simulation components' scaling on HECToR, SMP....................................................21
High-level application components' scaling on HECToR, SMP....................................22
Detailed parallel activation components' scaling on HECToR, SMP..............................23
Parallel Activate speed-up on HECToR, SMP.........................................................23
High-level run components' scaling on HECToR, 4-node MPP, high density simulation .................................................................24
Detailed simulation components' scaling on HECToR, 4-node MPP, high density simulation ........................................................................................................25
Parallel activate speed-up on HECToR, 4-node MPP, high density simulation............26
Job submission overhead on HECToR, minimal problem size simulation..................27
Job submission overhead on HECToR, scaling the number of iterations......................28
Parallel activate speed-up on HECToR, 7-node MPP.................................................29
Detailed simulation components' scaling on HECToR, 7-node MPP............................30
Detailed simulation components' scaling on HECToR, 15-node MPP subscribing only 4 cores per node...............................................................................................31
Parallel activate speed-up on HECToR, 15-node MPP subscribing only 4 cores per node........................................................................................................32
Topographica speed-up, SMP......................................................................................35
Bare C speed-up, SMP..............................................................................................36
Topographica speed-up, 13-node MPP subscribing only 4 cores per node...............37
Bare C speed-up, 13-node MPP subscribing only 4 cores per node..........................38
Topographica, comparing the total simulation time to the node activation time, SMP. 40
Detailed simulation components' timings for Bare C on HECToR, SMP..................42
Optimised Topographica speed-up, SMP.................................................................43
Optimised Topographica, comparing the total simulation time to the node activation
time..................................................................................................................................44
Acknowledgements

I would like to thank my supervisors Dr James Bednar and Mr Adrian Jackson for this great project, their advice and support. Many thanks to the CSNG student Christopher Ball for all the help he had offered throughout the entire course of this project. Finally, big thank you to the Edinburgh Parallel Computing Centre for this brilliant MSc in HPC programme and for letting us use HECToR.
Chapter 1

Introduction

Computational Neuroscience is the area of Science that uses computational resources to study the function of brain. There is a large number of research projects and simulators in the field running around the world. Some of the most notable are NEURON, Genesis, Brian, NEST and many others.

The Institute for Adaptive and Neural Computation at the University of Edinburgh and the Neural Networks Research Group at the University of Texas have developed a software package for computational modelling of neural maps named Topographica. This simulator focuses on high-level modelling of interactions of thousands of neurons at the same time. Currently Topographica allows simulating the Primary Visual Cortex (V1) which is an area located at the back part of the brain and is responsible for visual processing. The goal of the project is to help scientist get complete understanding of brain function at the level of topographic maps by modelling and running complex simulations of such maps. That distinguishes Topographica from other simulators.[1]

Even such small area of our brain as V1 contains hundreds of millions of interconnected neurons. Simulating the work of the entire patch would require enormous amount of computational resources and has not been feasible up to date. Initially Topographica was developed as a serial application and for a while was taking advantage of constantly increasing computational capabilities of single-core processors which allowed bigger and more complex simulations. However as this trend stopped some six years ago the Group started looking for new ways to allow running bigger and more accurate simulations.

The first approach taken was applying serial optimisations to the most computationally-heavy parts of the software: some of core components of Topographica were replaced with hand-crafted C code. Great results have been achieved this way. However, there is always a certain limit to the amount of serial optimisations that can be applied to a single piece of software. Once this limit has been reached the Group proposed three new projects: optimising the core components for efficient cache use and introducing OpenMP into the code, accelerating computations with GPGPUs and, finally, distributing computations with MPI. So far the GPGPU approach failed to produce any
Introduction  High Performance Cortical Simulator
reasonable performance improvements. Certain level of success was achieved with
OpenMP, although ultimately limited by the possible number of CPU cores in a multi-
core (multi-processor) chip and memory bandwidth. Introducing MPI has allowed
成功fully distributing computations across any MPP system but so far failed to
achieve the speed-up level that the Group was hoping for.

The goal of the project *High Performance Cortical Simulator* was to analyse what
possible performance can be achieved by using MPI with Topographica, test it by
running on massively parallel systems such as ECDF or HECToR, investigate the
reasons why the initial parallelisation failed to produce the anticipated results, pinpoint
the problems and fix them, if possible.

Being able to fix the problems with the current MPI implementation of Topographica
and achieving good parallel scaling properties would allow users of the software to run
simulations with densities of neurons much higher than the ones that are currently
feasible to use. Experiments of a bigger scale would produce more accurate results and
could possibly lead to new discoveries in the field. Such discoveries would give the
researchers better understanding of the brain function and, apart from expanding the
pool of human knowledge, could potentially help fighting mental illnesses.
Chapter 2

Background

In this chapter you can find some essential information on the project background, technologies used and some discussion for the approach used over the course of the project.

2.1 The Topographica Project

The Topographica software has been continuously developed over the past ten years by a number of different scientists and engineers which has led to a very complex structure. The application originates from another neural simulator called Lissom which was entirely written in C++. To reduce complexity and allow scientists to be able to easily add new components the majority of Topographica code, similarly to most other top neural simulators, has been written in Python with some core computation components implemented in C.[2] The following sections describe the main components of the simulator and provide definitions for some important terms used throughout this report.

2.1.1 The Glossary of terms

**Simulation Script:** Key to each simulation is the simulation script. This script describes the simulation parameters such as the density of neurons, their connectivity, the configuration of inputs and outputs and many others. It is the simulation script that defines the architecture of each individual simulation. These scripts are contained in the examples/ subdirectory of the root Topographica directory and have the .ty extension despite being ordinary Python scripts.

**Sheet:** Another key definition is the the sheet. There can be several types of sheets, each, however, representing a layer of neurons or photosensitive receptors. The most important ones for this project are the input sheet and the projection sheet.
Background  High Performance Cortical Simulator

Density: in this report, the number of elements contained in a sheet. Can be defined either in terms of $M$ and $N$ in an $M \times N$ matrix or as a single number corresponding to the total number of elements contained in a sheet.

Input Sheet (further in this report it may also be referred to as retina) represents the visual input received by the retina of a simulated eye. It would be safe to imagine the input sheet as an object containing a two-dimensional matrix of floating-point values (input activity, see below) – usually an image or some generated pattern. Each element in this case would represent the frequency of the signals sent by the corresponding area of retina to the the brain. For the sake of simplicity each element can be seen as the brightness value of one pixel in the grayscale input image.

Projection Sheet or the sheet of neurons represents a layer of neurons. Again, this can be seen as an $M \times N$ matrix where each element corresponds to a certain neuron in the sheet. Some of the most important components of any projection sheet are its activity and connection fields. These terms are described below.

Activity: each neuron in our brain is connected to thousands of other neurons sending and receiving millions of electric impulses every moment of time. Some of these connections intensify the electric impulse (excitatory connections) while others weaken (inhibitory connections). Topographica focuses on high-level interactions on the level of topographic maps and thus instead of simulating each impulse sent by any particular neuron uses the notion of firing rate, or frequency with which the signals are sent by one neuron to another. Such frequency can be represented in form of a single floating-point number. Activity, in turn, is a matrix where each element corresponds to the firing rate of one neuron or photosensitive element. Activity is not an object on its own but rather a property of any sheet object and can be one of the two types: input or output.

Input Activity: the matrix of frequencies of electronic impulse signals transmitted from one sheet onto another. Common to input sheets.

Output Activity: the matrix of electronic impulse frequencies on one sheet upon receiving signals from another. Common to any projection sheet.

Connection Field: normally represents connections of one neuron to a set of neurons either on a different sheet (Afferent connection) or on the same one (Lateral connection). Connection field is a class of objects one of the key components of which is the weights matrix – this matrix defines which neurons the element is connected to and the strength of each connection.

Weight: (as in the weights matrix of a connection field): the strength of one individual connection between two neurons represented as a floating point value. The larger the value of a weight the stronger the connection is between the two neurons.
Background  High Performance Cortical Simulator

**Iteration:** one of the most important parameters of any simulation is the number of iterations to be performed until it completes. One iteration is a moment of time during which the sheets exchange signals and the new activity values are computed.

**Activation:** the process of computing the new activity values through the response and activity output functions.

**Response Function:** the function for computing the output activity values. The most common response function, which is also one of the most computationally intensive components of the simulator, is the matrix-matrix dot-product of connection fields' weights and the input activity.

**Learning:** the process of updating the connection fields' weights using a certain learning algorithm. In all experiments carried out over the course of this project the learning function was switched off.

**Output Function:** the class of functions applied to connection fields' weights and the output activity.

**Weights Output Function:** the output function applied to connection fields' weights. Normalisation function which divides each weight value of one connection field by the sum of all weights in that connection field was used in all experiments carried out throughout the course of this project. Normalisation was done only once per simulation during the initialisation stage.

**Activity Output Function:** the output function applied to the output activity produced by the response function. The Hysteresis function was used in most of the experiments discussed in this paper.

### 2.1.2 Components of the simulator

Topographica is a highly modular application: it is built of multiple components each serving their own purpose. Only the ones directly relevant to this project's optimisation effort will be discussed. These are the fundamental components used in almost all of the Group's models

**topo/base/cf.py:**

- Class **ConnectionField:** a building block of a projection sheet. Contains the weights matrix
- Class **CFProjection:** set of all connection fields for one projection
- Class **MPI_CFProjection:** the proxy-class (or *serial-side* implementation, see section 2.2.2 PMI) of the parallelised CFProjection class. Objects of this class are only created on the master node in parallel mode

**topo/base/mpi Cf.py**
2.2 Tools

In this section you may familiarise yourself with the main tools used over the course of the project.

2.2.1 Mpi4py

MPI for Python, or simply Mpi4py is a library that is a thin wrapper around MPI routines that makes them available for use in the Python programming language. The library provides an object oriented approach to message passing, grounding on the standard MPI-2 C++ bindings.[3] Mpi4py is a part of the SciPy project and was chosen from the list of other libraries with similar functionalities such as pyMPI or Pydusa, to be used in Topographica mainly due to being a very mature Python package and extremely responsive support service available at the Mpi4Py Google group at:

http://groups.google.com/group/mpi4py

2.2.2 PMI

Parallel Method Invocation (PMI) is a Python module developed by Olaf Lenz at the Max Planck Institute for Polymer Research. This module is implemented on top of Mpi4Py (or boostmpi) and introduces data-parallel programming model into Python by combining message passing with OpenMP fork-join model. The purpose of the PMI module was to make the parallel code easier to write and understand.

PMI allows calling the arbitrary python functions to run in parallel from a serial script. Once called the different invocations of the function can communicate over MPI. When the function returns the results are gathered on the master node and control flow is returned to the serial script.

Thus PMI introduces the concepts of the **serial-side** implementation, or the controller, and the **parallel-side** implementation, or the workers. On the controller side PMI functions `call()`, `invoke()` and `reduce()` once called execute the given function on all workers.[4]

In the following example, taken from the “PMI – Parallel Invocation” paper by Olaf Lenz, we can see a function that computes the Mandelbrot fractal in parallel:

```python
import pmi

# import the module on all workers
```
Background  High Performance Cortical Simulator

```python
pmi.exec_('import mandelbrot_pmi')

# This is the parallel function that is
called from mandelbrot()
def mandelbrot_parallel((x1, y1), (x2, y2),
(w, h), maxit):
    '''Compute the local slice of the
    mandelbrot fractal in parallel.'''
    # Here we can use any MPI function.

# This is the serial function that can be
called from a (serial) user script
def mandelbrot(c1, c2, size, maxit):
    return pmi.call(
        'mandelbrot_pmi.mandelbrot_parallel',
        c1, c2, size, maxit)
```

Following is a sample serial script that calls the parallelised Mandelbrot function:

```python
import pmi, mandelbrot_pmi
# Setup pmi
pmi.setup()

# Call the parallel function
M = mandelbrot_pmi.mandelbrot((
    (-2.0, -1.0), (1.0, 1.0),
    (300, 200), 127
)

PMI was used for developing the parallelised version of Topographica.

2.3 The First Parallel Version

The first parallel version introducing MPI into the previously serial code of
Topographica has been developed in year 2010. Although most of the computationally
heaviest components have been successfully distributed, only a limited performance
improvement has been achieved. For example, using the Lissom simulation model with
the parameters of cortex and retina densities of 142 and 24 respectively, used in most of
the experiments described in the “Computational Maps in the Visual Cortex” book that
served as a guidance for the project, the best speed-up factor achieved with any number
of processors of the Jupiter cluster (up to 24 cores maximum) was 3.2.[5] Despite the
arguably poor performance improvement the project was successful in terms of
analysing the range of available MPI tools and libraries and finding the most suitable
Background  High Performance Cortical Simulator ones, discovering the areas for potential improvement and creating a solid base for future parallelisation work.[6]

The first parallel version of Topographica and the research carried out during its implementation served as a basis for further development and analysis discussed in this document.

2.4 The Approach

2.4.1 Divide and Conquer

When dealing with very complex problems in research it is often useful to break it down into smaller problems, then narrow down the scope of investigations and focus on the ones that seem to be the most important by isolating them from other problems and interrelations. This approach of breaking the problem down into smaller components and solving them independently is called “Divide and Conquer” and is an essential part of any analysis process.[7] It was decided to use this approach in Topographica the following way:

1. Identify the most computationally intensive process in the simulator using profiling
2. Identify the components of the simulator that are directly responsible for managing that process
3. Carry out a series of experiments in order to find out how the performance of these components could be improved
4. If possible, conduct all the necessary optimisation work
5. Once the optimisation effort is complete analyse the performance of the new code and state why no or very little improvement could be achieved by continuing the optimisation work
6. Go back to stage 1 unless all of the processes in the simulator have been covered using this approach already in which case the simulator can be considered fully optimised

The most computationally intensive process has been determined to be the activation, or computing the neural activity values. The following profiling listing for the serial version of Topographica produced by the standard Python profiler demonstrates this (edited to leave only the most important bits, the full output can be found in Appendix B):
In the listing above `cumtime` is the cumulative time taken by the routine one line below the timing. The first numerical value in bold corresponds to the total simulation time: 22.508 seconds. Out of that time 18.838 seconds have been consumed by doing computations in the inlined C code. That includes computing new activity values, doing learning and applying various output functions. Finally, 16.230 seconds, or 72% of the total simulation time were spent performing the `responsefn` routine of producing new activity values using the matrix-matrix dot-product function, or simply activation. This routine has been picked as the most important part of Topographica for the optimisation efforts discussed in this paper to be focused on.

### 2.4.2 Abstraction

Another approach for problem solving employed in this project was the Abstraction. In philosophy, abstraction is a thought process based on distancing ideas from objects.[8] Applied to science and engineering this concept can be described as solving a problem in a model of a system before applying the solution to the real system. This approach served as the motivation for building the Bare C application and the Matchmodel simulation script discussed further in this report.

The Bare C code was entirely written in C in order to exclude any serial overheads common to interpreted and object-oriented languages. It was decided to implement some of the core functionalities of the simulator and use this highly efficient model as performance benchmark for Topographica.

The Matchmodel script was designed to match Bare C in terms of the operations and the results produced given the same set of inputs. Both models are described in greater detail in the Implementation chapter of this document.

### 2.4.3 Hand-Crafted Profiling

Previous research into optimising Topographica for shared memory architectures has shown that due to the complex object-oriented nature of the simulator and the inlined C code implementing some of the core components, off-the-shelf profiling and tracing
Background  High Performance Cortical Simulator tools often give contradicting and unreliable results. The only measurements that could be trusted were wall-clock timings – system time (/usr/bin/time) and basic timers placed into the code.[9] Therefore it was decided to reserve to only using these tools for the current project.

2.5 Communication

The reporting on progress and communication with the project stakeholders from the CSNG group of the School of Informatics and EPCC was done through:

- **Weekly meetings** for discussing the status of the project and making decisions on further development
- **E-mail**
- **Project blog** where all experiment results, progress updates and issues were posted and discussed in the comments. The blog is public and can be found at [http://www.hyper-graphica.blogspot.com/](http://www.hyper-graphica.blogspot.com/)
Chapter 3

Systems

In the following sections you can find some technical characteristics of the systems used for running the experiments discussed in further chapters of this paper.

3.1 Jupiter

Jupiter is a cluster consisting of three nodes provided by the Computational Neuroscience department of the School of Informatics.

3.1.1 Hardware

• Three 8-core nodes
• Each node consists of two hyper-threaded Intel 3.00 GHz Xeon X5450 processors
• Bus frequency of 1333 MHz
• Data width of 64 bit
• Cache:
  ○ 6 MB shared L2 cache
  ○ 32 KB dedicated L1 cache
• 32 GB of RAM per node

3.1.2 Software

• GNU/Linux operating system with x86_64 Kernel 2.6.18
• GNU compiler version 4.1.2
3.2 HECToR

“High End Computing Terascale Resource (HECToR) is the UK’s high-end computing resource, funded by the UK Research Councils. It is available for use by academia and industry in the UK and Europe.”

3.2.1 HECToR Phase 2b hardware characteristics

- Cray XE6 parallel processing system consisting of 1856 Magny Cour compute nodes which contain two AMD 2.1 GHz 12-core Opteron processors
- Each 12-core processors consists of two hex-core dies connected via the same socket. Within each die:
  - 6 MB shared L3 cache
  - 512 KB dedicated L2 cache
  - 64 KB dedicated L1 instruction cache
- Theoretical peak performance of 373 Tflops
- 32 GB main memory per node
- 85.3 GB/s of bandwidth to local memory across 2 processors
- Gemini interconnect [10]

3.2.2 Software

HECToR is running the CLE operating system consisting of a full-featured Linux distribution on the service nodes, including user login nodes, and Compute Node Linux (CLE) designed to reduce the demands on compute time and memory.[11]

Job scheduling is managed by Portable Batch System Professional (PBS Pro).

3.2.3 Modules used in this project

- GNU compiler (PrgEnv-gnu)
- PGI compiler (PrgEnv-pgi)
- Cray MPT (xt-mpt)
Systems  High Performance Cortical Simulator

- Cray MPICH2 (xt-mpich2)
Chapter 4

Implementation

This chapter covers the details of implementation for the code developed over the course of this project.

4.1 Bare C

Bare C is a largely simplified model of the core Topographica functionality written in C. Bare C was capable of

- Running simulations for a specified number of iterations and with a given problem size passed as command line arguments
- Using rectangular sheets (activity and projection) defined by the width and height parameters passed as command line arguments
- Simulating one projection sheet fully-connected to the input activity sheet and producing one output activity sheet per iteration, all of the same height and width
- Running the simulations in two modes: serial and parallel, and then comparing the results of both runs to ensure correctness of the parallel version
- Timing the components of the code and then printing the timings to the command line and into an XML file

The code base consisted of

- `main()` function responsible for
  - MPI initialisation and finalisation
  - Array declarations and memory allocation
  - I/O
Implementation of the High Performance Cortical Simulator

- Parsing the command line arguments

- Starting the serial and/or parallel simulations through calls to `activateSeq()` and `activateMPI_1()` functions

- Running output correctness tests

- Printing the component timings measured with `MPI_Wtime()` to the command line and an XML file

- `activateSeq()` function, the main body of the serial simulation code. Consists of
  - Array declarations and memory allocation
  - The simulation loop that runs for a user-specified number of iterations, each iteration
    - Generating a new input activity matrix using the Moving Dot function (described in section 7.2)
    - Producing the output activity matrix by computing the matrix-matrix dot-product of the input activity and connection fields' weights matrices, one operation per element of the output activity
    - Applying the Hysteresis output function to the output activity matrix

- `activateMPI_1()`, the parallel version of `activateSeq()`. In addition to the functionality of the serial version
  - Determines using the MPI rank and size values the number of connection fields and their absolute position in the global array (the projection sheet) for each MPI node
  - Initialises connection field weights matrices locally on each MPI node
  - Broadcasts the input activity generated with the Moving Dot function on MPI node 0 to all other nodes
  - Produces the new values of the output activity on each node the same way as in the serial version but only for a chunk of the matrix that was assigned to that node
  - Gathers the chunks of the output activity from all nodes into one matrix using the `MPI_Gatherv()` routine
  - Applies the Hysteresis function to the gathered output activity matrix, only on the node with rank 0
4.2 Matchmodel

The Matchmodel script (*matchmodel.js*) was created to match the Bare C application in terms of the functionality and the outputs produced given the same inputs. The script was capable of:

- Running a simulation in serial and parallel modes depending on the value of the *mpi* command line argument supplied (True or False)
- Using arbitrary projection and activity sheet densities set through the command line as width and height parameters (*m* and *n*)
- Running for a certain number of iterations set through the command line
- Initialising and generating the connection field weights
- Generating the input activity matrices using the Moving Dot function

4.3 Plotting and Timing

For plotting a Python application based on the Matplotlib library was created. The code of this application evolved throughout the course of this project and went from a primitive script that could read text files with entries sorted by the user and create only component scaling graphs with a minimal number of labels to a full-scale plotting application capable of parsing XML files and working in several modes.

The final version of the plotting script was capable of:

- Taking inputs from the command line
- Using the inputs given selecting which mode to work on
- Parsing XML files
- Sorting entries of the XML file according to the arguments supplied
- Identifying and omitting duplicate entries in the XML file
- Generating individual timer scaling graphs
- Generating complex coloured component scaling graphs
- Generating speed-up graphs with “ideal speed-up” indicated and printing the parallel efficiency values

Implementation  High Performance Cortical Simulator

- Utility functions
Implementation  High Performance Cortical Simulator

- Allowing users to select which attributes or entries of the XML file to use as X and Y axes
- Generating multiple graphs from a single XML file if Z parameter was supplied
- Labelling graphs

This application was used to create most of the graphs presented in this report and was proven to be of great value despite having taken a considerable amount of time and effort to be taken through several evolution steps before it was completed.

Another Python script designed to make it easier to time the highly modular code of Topographica was created at a later stage of the project and was called topotimers.py. This script could be imported in any Python file as a module and allowed placing startTimer() and stopTimer() calls around any piece of code also setting a name and a tag for each timer. After the execution of the timed program was finished it combined all individual timers into one XML document, set simulation run parameters, if any, and printed into a file. This tool was also heavily used throughout the later stage of the project.

4.4 Optimised PMI

The PMI Python module pmi.py was improved by adding a new method invoke_opt that unlike the standard invoke was using the non-pickling Gatherv communication routine of the Mpi4py module. The details of implementation and development can be found in chapter 7.

4.5 Scripting

A number of scripts were created to manage

- PBS job submission on HECToR in different modes (p.pbs, cp.pbs, activation.pbs, lissom.pbs, timed_topographica.pbs)
- Running correctness tests (c_vs_python.sh, timed_topographica.sh)
- Timing support (parsetime.py)
- Benchmarking collective communication routines of Mpi4py (bench_collectives.py)
Chapter 5

Bare C Performance Analysis

In this chapter you will find the performance analysis for the Bare C application created to identify the ideal performance that could be achieved with some of the core components of Topographica. The graphs and analysis presented in this chapter were aimed to demonstrate the scaling properties of the parallel Bare C code, the potential speed-up that could be achieved with different numbers of CPU cores and parallel nodes and how the simulation time is distributed across different components of the application.

5.1 Jupiter

The Jupiter cluster was picked for the first series of experiments. There is no batch processing system on Jupiter which means zero turn-around times. The downside of using Jupiter was that its very limited, compared to ECDF or HECToR, number of CPU cores and memory bandwidth made it unsuitable for studying the scaling properties of Topographica and Bare C which are very intensive applications in terms of memory accessing. However, the ability to execute parallel applications on Jupiter immediately made testing the timing approach and identifying any potential problems very convenient. For example, the first series of graphs produced demonstrated that there was a large overhead of inefficient weights distribution.
On the following figure you can see the Bare C top-level set of timings from Jupiter:

**Figure 1: Components’ scaling on Jupiter. Optimised weights distribution**

This figure is an example of a component scaling graph where different components of the simulation are stacked one on top of the other. The top edge of topmost coloured field represents the sum of all components. The black line represents the capping timer to which all components should ideally add up.
The higher-detail timings for parallel activation:

![Graph showing components' scaling on Jupiter, optimised weights distribution](image)

**Figure 2: Components' scaling on Jupiter, optimised weights distribution**

Legend:

- `act_mpi_tot` – the total simulation time in parallel
- `act_mpi_of` – activity output functions
- `act_mpi_gather` – gathering output activities
- `act_mpi_dp` – computing dot-product for new activity values
- `act_mpi_bc` – broadcasting the inputs
- `act_mpi_distr_weights` – distributing the initial set of connection field weights
- `act_mpi_utils` - utility functions

The most important values of MPI.size to pay attention to on this graph are 1,2 and 3 as only these correspond to purely distributed-memory runs where the simulation was not affected by the limitations of the Jupiter memory system. The total simulation time (`act_mpi_tot`) demonstrated very close to linear scaling behaviour with these numbers of CPU cores subscribed.
5.2 HECToR

The massively parallel HECToR system with high memory bandwidth was used for obtaining reliable timings and studying the scaling properties of Bare C.

5.2.1 Shared Memory Run

Following is the Bare C components breakdown graph from the first SMP run on HECToR (Time along the Y axis, MPI size along the X axis) with problem size of 200 by 200 elements for 100 iterations:

![Diagram showing simulation components' scaling on HECToR, SMP]

**Figure 3: Simulation components’ scaling on HECToR, SMP**

Here we can see that the simulation time (act_mpi_tot) is dominated by the dot-product computation on any number of cores between 1 and 24. The weights distribution despite adding a certain overhead on small number of cores becomes insignificant on a larger number of cores.

The speed-ups and parallel efficiency values calculated as speed-up divided by MPI size for this run:

- 2 cores: x1.99 (E=0.99)
- 4 cores: x3.91 (E=0.9775)
- 8 cores: x6.40 (E=0.8)
Bare C Performance Analysis  High Performance Cortical Simulator

- 16 cores: x11.87 (E=0.74)
- 24 cores: x20.94 (E=0.87)

On HECToR Bare C seemed to be demonstrating a relatively good scaling behaviour with slightly sub-linear speed-up. Notice the parallel efficiency improvement on 24 cores – 13% higher than on 16 cores. An assumption was made that at this number of CPU cores subscribed or higher the Bare C data structures were hitting the cache and it would be possible to achieve linear or even super-linear speed-up by varying the number of processors used and the problem size. In order to test this assumption it was decided to make a smaller problem size run with the same number of CPU cores and then try a bigger problem size with a larger number of CPU cores, using several MPP nodes.

In the following figure we can see the overall timing graph capped by the wall-clock timer (/usr/bin/time) for a 100 by a 100 density run for 100 iterations:

![Figure 4: High-level application components' scaling on HECToR, SMP](image)

Notice how the gap between the wall-clock time (/usr/bin/time) and the the total application run time seems to increase with the number of cores. This issue has been investigated and discussed in section 5.2.3 of this document. More detailed break-down of main: Parallel Activate, the total parallel simulation time:

22
Hector, 1 node, TIME (activate), density = 10000

Figure 5: Detailed parallel activation components’ scaling on HECToR, SMP

Finally, the speed-up graph (Parallel Activate is the same timer as parallel: Total on the previous figure):

Hector, 1 node, SPEEDUP, density = 10000

Figure 6: Parallel Activate speed-up on HECToR, SMP
Bare C Performance Analysis  High Performance Cortical Simulator
Notice that while still demonstrating good scaling properties (e.g. parallel efficiency of about 88% on 24 cores), unfortunately the speed-up of Bare C remained sub-linear.

5.2.2 Distributed Memory Run: 4 nodes
The following graphs were obtained in a hybrid SMP-MPP run using 4 nodes (up to 96 CPU cores) with a problem size of 200 by 200 elements for 100 iterations. Top-level component break-down:

Figure 7: High-level run components' scaling on HECToR, 4-node
MPP, high density simulation
Bare C Performance Analysis  High Performance Cortical Simulator
Detailed simulation break-down:

Figure 8: Detailed simulation components' scaling on HECToR, 4-node MPP, high density simulation

Same as with the previous runs, the simulation time is, as expected, largely dominated by the computation (dot-product).
Bare C Performance Analysis  High Performance Cortical Simulator

The speed-up graph (again, Parallel Activate is the same timer as parallel: Total on the previous graph):

![Graph showing speed-up for Hector, 4 nodes, SPEEDUP, density = 40000](image)

**Figure 9: Parallel activate speed-up on HECToR, 4-node MPP, high density simulation**

Again, the speed-up never turns linear or super-linear on any number of processors. Nevertheless, achieving anything even close to the speed-up of 76 times observed in Bare C on any number of CPU cores with Topographica in normal simulations would mean a broad range of new opportunities for simulations that have never been feasible before.

### 5.2.3 Job Submission Overhead

One of the first set of timing results produced by running Bare C for a relatively small number of iterations on one node with up to 24 cores subscribed uncovered an interesting problem: there was significant difference between the total execution time measured externally with `usr/bin/time` and internally with `MPI_Wtime()` calls placed right after the entry point of the main function of the program and right before the function return. Judging by the performance graph displayed in the figure 4, there was an overhead to the execution time that almost doubled the program run time measured externally and seemed to scale up with the number of CPU cores. Obviously if this had been the case then the internal timings obtained in previous simulations would have been misleading and could not be relied on. They would have simply been not representing the actual time it took to complete a single simulation. In order to test
Bare C Performance Analysis  High Performance Cortical Simulator

whether the overhead was indeed scaling with the number of CPU cores the following experiment has been conducted: running Bare C with the smallest density of just one element for just one iteration on one node subscribing from 1 up to 24 cores. On the following graph you can observe the results of this experiment:

![Hector, 1 node, TIME, density = 1](image)

**Figure 10: Job submission overhead on HECToR, minimal problem size simulation**

Conclusions:

- The overhead is to a certain degree unpredictable
- The overhead does not depend on the number of cores used in a single simulation run
- Finally, the overhead can be anywhere between 0.5 and 4 seconds

Next it was decided to check whether this overhead would be affected by the length of the program execution time by running series of experiments of the same problem size and varying number of iterations. The density was chosen to be 10,000 elements (100x100) and the number of iterations was running from 50 to 500. The following graph demonstrates the results of this experiment:
Conclusions:

- Although somewhat deviating randomly from run to run by 1-2 seconds, the overhead is static, meaning it does not increase for longer simulations
- By running simulations that are longer in terms of the number of iterations the ratio of the overhead to the total run time can be reduced to some insignificant value and therefore ignored
5.2.4 Distributed Memory Run, 7 nodes

10,000 elements problem size (100x100 density), 1,000 iterations. The speed-up graph:

![Speed-up Graph](image)

Figure 12: Parallel activate speed-up on HECToR, 7-node MPP

Parallel efficiency:

- cpu\_cores: 1  E = 99%
- cpu\_cores: 7  E = 99%
- cpu\_cores: 14  E = 97%
- cpu\_cores: 28  E = 94%
- cpu\_cores: 56  E = 81%
- cpu\_cores: 112  E = 68%
- cpu\_cores: 168  E = 37%

According to these results good scaling with the density of 10,000 elements can only be achieved up to a certain number of CPU cores subscribed and this number is between 112 and 168. It would not make sense to add any more computational power above that number since it would only negatively affect the performance. Another conclusion made was that the best efficiency can be achieved if MPP nodes were undersubscribed. Up to four cores per node provide very close to linear performance scaling. This
Bare C Performance Analysis  High Performance Cortical Simulator
number of CPU cores utilised per node was used to produce the timing results analysed
in section 5.3.5.

Simulation break-down timings (the sum of all timers, or the top edge of the topmost
part of the graph filled with colour represents the same timer as main: Parallel Activate
on the previous graph):

![Graph showing performance analysis](image)

**Figure 13: Detailed simulation components' scaling on HECToR,**
**7-node MPP**

It is clear from the graph that starting from four cores subscribed per node processes
start under-performing and create visible load imbalance. However, there is no actual
load imbalance in any of the simulations: all nodes have exactly the same amount of
work to do. Neverthless due to the memory accessing bottleneck some processors
wait longer to get their data from memory while others have to wait in the collective
routine for the former to complete their computations. In addition to that, the increasing
time spent in the MPI_Gather collective is definitely not representing the real
communication overheads: if it was then the broadcasting time would be increasing as
well because the amount of data broadcast by node 0 on each iteration is greater than
the amount of data it has to gather by the corresponding collective routine. The only
explanation for why broadcasting time is very small is that on each iteration except the
first one it happens after another collective - MPI_Gather, which implicitly
synchronises all nodes, and the actual communication time resulting from network
bandwidth and latency limitations for this routine is very small.
5.2.5 Distributed Memory Run on 15 Nodes

In this experiment the simulation was run for 1,000 iterations with a 100x100 density on up to 15 nodes subscribing four cores per node. The remaining 20 cores on each node were left unused. Thus, in the first run only one core on one node was used, in the second – four cores on one node, in the third – two cores using four cores per node and so on. Timings:

![Diagram](image)

**Figure 14: Detailed simulation components' scaling on HECToR, 15-node MPP subscribing only 4 cores per node**

Even with 60 cores the simulation is still largely dominated by computation, although communication overhead becomes important too.
Bare C Performance Analysis  High Performance Cortical Simulator

The speed-up graph (main: Parallel Activate timer equals to the sum of all coloured components from the previous graph):

Figure 15: Parallel activate speed-up on HECToR, 15-node MPP subscribing only 4 cores per node
Finally, the parallel efficiency values:

<table>
<thead>
<tr>
<th>CPU cores</th>
<th>Parallel Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>E = 97.5%</td>
</tr>
<tr>
<td>12</td>
<td>E = 96.7%</td>
</tr>
<tr>
<td>20</td>
<td>E = 96.0%</td>
</tr>
<tr>
<td>28</td>
<td>E = 94.7%</td>
</tr>
<tr>
<td>36</td>
<td>E = 86.9%</td>
</tr>
<tr>
<td>44</td>
<td>E = 90.2%</td>
</tr>
<tr>
<td>52</td>
<td>E = 87.4%</td>
</tr>
<tr>
<td>56</td>
<td>E = 80.5%</td>
</tr>
<tr>
<td>60</td>
<td>E = 76.4%</td>
</tr>
</tbody>
</table>

Table 1: Bare C parallel efficiency values, 15-node MPP, subscribing 4 cores per node

The performance of Bare C scales almost linearly with close to 90% parallel efficiency on up to 52 cores (13 nodes) after which point adding any more computing nodes creates communication overhead that prevents the speed-up factor from increasing any further.

Despite the higher parallel efficiency that can be achieved by under-subscribing the nodes, the speed-up gained by fully (or almost fully) subscribing fewer parallel nodes can be a lot higher. Recall the results of section 5.2.4 where using 112 CPU cores across 7 nodes gave the speed-up of above 76 times the serial performance. The parallel efficiency in that case was at 68%, almost 10% lower than the lowest value achieved in the experiment described in the current section. However, the speed-up of 76 times was over one and a half times higher than the highest speed-up achieved in this experiment and that – using fewer parallel nodes, hence less CPU budget. This means that given some fixed budget, more science can be done by fully or almost fully subscribing the nodes even despite the poorer parallel efficiency than it would have been if nodes were under-subscribed.
5.3 Conclusion

In this chapter we observed that given a large enough problem size and number of iterations and up to a certain number of CPU cores dependent on the problem size most of the simulation time was spent doing the computations. This was the highly desired behaviour. Apart from that, it was demonstrated that despite the good scaling property the application can only achieve sub-linear speed-up on any number of CPU cores or parallel nodes. This gives the limit to what would be possible to achieve with Topographica in terms of performance, which is investigated in the next chapter.

We were able to observe that the best parallel efficiency could be achieved if nodes were under-subscribed. However, despite the lower parallel efficiency, fully subscribing the nodes can provide higher speed-ups with a smaller number of nodes used and therefore less CPU budget consumed to do the same amount of science.
Chapter 6

Performance Analysis of Topographica

In this chapter you will find the comparative analysis of Topographica and Bare X and also how optimising certain routines of Topographica has affected its performance.

6.1 HECToR, Single Node SMP Run, C VS Python

In this experiment the performance of Topographica executing the Matchmodel script was compared to that of Bare C with the 100x100 elements problem size used in both applications, running for 1,000 iterations. In the following figure you can see the speed-up graph for Topographica running on one SMP node using from 1 to 24 CPU cores:

![Graph showing speed-up for Topographica](image)

**Figure 16: Topographica speed-up, SMP**
Performance Analysis of Topographica  High Performance Cortical Simulator
On the next graph you can see the speed-up of Bare C on the same range of CPU cores and with the same simulation settings:

![Graph showing speed-up of Bare C on HECToR, 1 node, S-U, density = 10000]

**Figure 17: Bare C speed-up, SMP**

The difference in performance is quite significant: for example, on 24 cores Bare C shows more than one and a half better speed-up than Topographica. The difference in performance becomes even clearer once we compare the parallel efficiency of the two versions of the code:

<table>
<thead>
<tr>
<th>CPU cores</th>
<th>Topographica</th>
<th>Bare C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E = 99.0%</td>
<td>E = 99.5%</td>
</tr>
<tr>
<td>2</td>
<td>E = 95.4%</td>
<td>E = 98.3%</td>
</tr>
<tr>
<td>4</td>
<td>E = 89.3%</td>
<td>E = 96.3%</td>
</tr>
<tr>
<td>8</td>
<td>E = 78.4%</td>
<td>E = 91.7%</td>
</tr>
<tr>
<td>16</td>
<td>E = 66.1%</td>
<td>E = 90.6%</td>
</tr>
<tr>
<td>24</td>
<td>E = 55.4%</td>
<td>E = 86.2%</td>
</tr>
</tbody>
</table>

**Table 2: Bare C and Topographica parallel efficiency comparison, SMP**
Performance Analysis of Topographica

High Performance Cortical Simulator

Such relative difference in performance not favouring Topographica signified an important goal achieved in this project: an application matching the simulator in terms of the core functionality has been built and given the same simulation parameters was demonstrating far better performance on any number of CPU cores.

6.2 HECToR, Multiple Node Run, C VS Python

Prior to investigating the reasons for the Topographica performance being worse than that of Bare C it was necessary to check whether the difference would hold in a distributed memory run. In the experiment discussed in this section both applications were run on a number of HECToR nodes ranging from 1 to 13, subscribing only 4 CPU cores per node which was determined previously to be the optimal number of cores to be used for best memory system utilisation. The problem size was slightly higher than usually – 14,400 elements per sheet (120 by 120 density) and the number of iteration was 1,000. The following graph displays the speed-up factor scaling of Topographica:

![Graph showing speed-up factor scaling of Topographica](image)

*Figure 18: Topographica speed-up, 13-node MPP subscribing only 4 cores per node*
Bare C speed-up:

![Graph showing speed-up for Bare C, S-U, density = 14400](image)

**Figure 19:** Bare C speed-up, 13-node MPP subscribing only 4 cores per node

It can be easily seen that the performance of Topographica is still significantly worse: for instance on 48 cores the speed-up of Bare C is about 1.7 times higher than that of Topographica.

### 6.3 Detailed Topographica Timings

Following are the key component timings taken from a single-node 24-core run for 1,000 iterations with a density of 100x100:

- **"Simulation total"**: 16.3 s – the total amount of time elapsed between the beginning of the first iteration and the end of the last one

- **"MPI_CFP.activate()"**: 13.6 s – the activate function of *MPI_CFPProjection* class. This class contains serial code and through PMI calls functions of *MPI_CFPProjection_node*

- **"MPI_CFP_n.activate()"**: 11.2 s – the activate function of *MPI_CFPProjection_node* (parallel CFProjection), consists only of computing the response and output functions on the nodes (does not include any communication)
Performance Analysis of Topographica  High Performance Cortical Simulator

- "MPI_CFP.nactivate():response_fn": 11.1 s – response function call (response_fn mainly consists of the dot-product implementation) inside the 
  MPI_CFPProjection_node.activate() method

Judging by these timings we can see that most of the parallel activation time (MPI_CFP.nactivate()) was spent computing dot-products (11.15 s out of 11.19). This makes it safe to disregard the output function computation done by the parallel-side activate which could only account for as little as 0.3%. Parallel activation took 11.19 seconds out of 13.57 seconds (or 82%) of serial-side activation call (MPI_CFP.activate()) which consisted only of the PMI call serving as a bridge between the serial and parallel implementations and re-combining the data. Following is the source code for this function:

```python
def activate(self, input_activity):
    activity_list = pmi.invoke(self.pmiobj,'activate',
                               input_activity)
    self.activity = numpy.array([])
    for activity_row in activity_list:
        self.activity = numpy.append(self.activity,
                                      activity_row)
    self.activity=self.activity.reshape(
                              self.activity_shape[0],
                              self.activity_shape[1])
```

As we can see, the serial-side activation consists of calling the PMI invoke method which distributes the input activity array and triggers the activation on the nodes and reconstructing the output activity using the chunks gathered from the nodes and returned in a list by PMI invoke. The timings presented in the beginning of this section indicate that 18% of of the time spent inside the serial-side activation function were consumed by the array reconstruction and the PMI internals.

The serial-side activation according to the same timings contributed to 83% of the total simulation time which means that only 68% of the simulation was spent doing computations while the remaining 32% were wasted due to various overheads. If we compare this to the performance of Bare C we will see that this gap between the simulation time and the node activation indicates a severe inefficiency on the serial side of the Topographica code:

- 'main: Parallel Activate' 10.8 s – the total simulation time (called “Simulation Total” in Topographica)
Performance Analysis of Topographica  High Performance Cortical Simulator

- **parallel: Dot Product** 10.0 s – the computation time. Same as the response function which takes more than 99% of the parallel-side activation in Topographica.

9.997 out of 10.81 seconds of the total simulation time was spent computing the dot-products for the new activity values. That is 92%, as opposed to Topographica's 68% which more than 20% less efficient. The difference can be also observed by looking at the following two graphs where the total simulation time (thick black line) and the computation time were plotted for Topographica and then for Bare C. Again, we assume that the dot-product computation in Bare C and node activation in Topographica are the same concepts by disregarding the output functions also computed by the node activation method which account for an insignificantly small amount of time:

![Graph](image)

**Figure 20: Topographica, comparing the total simulation time to the node activation time, SMP**

And now, the timings for Bare C. The indicators most important for this discussion are **main: Parallel Activate** (the total simulation time, shown as a thick black line) and **parallel: Dot-product** (the area filled with grey colour):
Figure 21: Detailed simulation components' timings for Bare C on HECToR, SMP

It seemed clear that the activation time in Bare C

1. is completely dominated by the dot-product computation

2. takes about as much time as the total simulation

However, that was not the case with Topographica. Notice on the first figure that the time difference between the simulation total and the node activation increased with the number of cores used: there was almost none on 2 cores and it became clearly visible on 24. This was an indication of a scaling overhead which could be caused by two things:

- PMI internals such as communication routines and data operations

- Serial-side Topographica code (anything outside the node activation and PMI functions)

This assumption has led to discovering the roots of the inefficiency in Topographica and improving the PMI invoke method with a new communication mechanism. The details of the development can be found in the Development Issues and Discussion chapter of this document.
6.4 Performance of the Optimised Topographica

The following graph displays the speed-up factor scaling with the number of CPU cores for the optimised version of Topographica and with the improved PMI module:

![Graph showing speed-up factor scaling with number of CPU cores](image)

**Figure 22: Optimised Topographica speed-up, SMP**

The speed-up factor on 24 cores after the optimisations have been applied was 16.22, as opposed to the previously observed 13.29. The comparison can be also made using the parallel efficiency measurements:

<table>
<thead>
<tr>
<th>CPU Cores</th>
<th>Optimised Version</th>
<th>Non-optimised Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E = 99.5%</td>
<td>E = 99.0%</td>
</tr>
<tr>
<td>2</td>
<td>E = 96.6%</td>
<td>E = 95.4%</td>
</tr>
<tr>
<td>4</td>
<td>E = 91.7%</td>
<td>E = 89.3%</td>
</tr>
<tr>
<td>8</td>
<td>E = 79.7%</td>
<td>E = 78.4%</td>
</tr>
<tr>
<td>16</td>
<td>E = 75.3%</td>
<td>E = 66.1%</td>
</tr>
<tr>
<td>24</td>
<td>E = 67.6%</td>
<td>E = 55.4%</td>
</tr>
</tbody>
</table>

**Table 3: Optimised and non-optimised Topographica parallel efficiency comparison**
Performance Analysis of Topographica  High Performance Cortical Simulator

Notice that the difference in performance favouring the optimised version increases with the number of cores which means the scaling of Topographica has been improved. We can also find that the gap between the total simulation and the node activation time has significantly decreased and became fairly constant, which means that most of the scaling overheads have been eliminated:

![Graph showing performance improvement](image)

\[ \text{New PMI,TOPO,1 node,TIME, density = 10000} \]

\[ \text{Figure 23: Optimised Topographica, comparing the total simulation time to the node activation time} \]

6.5 Conclusion

In this chapter we have seen that initially the performance of Topographica was a lot poorer than that of Bare C and that the difference was getting worse with the amount of computational power added. Studying this behaviour has helped to make assumptions as for what could be causing the overheads and to identify which parts of the simulator had to be investigated in search of the inefficient routines.

The new version of Topographica together with the optimised PMI module still failed to match Bare C in terms of performance but it was shown that in the new version the overheads were fairly static with only a small amount of scaling with the number of CPU cores which means that these overheads are most likely to be caused by the Python code rather than some inefficiencies in the parallel code.
Chapter 7

Development Issues and Discussion

7.1 First Steps

The project started off with the first meeting where the first task was set to re-familiarise with the Topographica code and the parallel implementation developed before. The first system to base development on was Jupiter. Topographica installation went seamlessly, and after the build process has been completed both the serial and the parallel versions worked perfectly fine. Later the new versions of Mpi4py and OpenMPI have been downloaded and successfully replaced the outdated packages.

In order to re-familiarise with the Mpi4py interface while also hoping to find a potential source of performance improvement it was decided to test one assumption that there was a way to remove some of the python overheads by changing the communication routines in Python. The issue was once raised at the Mpi4py Google group that it provides several versions of the same collective communication routines: one category, the one that was employed in Topographica, was using pickling (compressing Python objects into strings) and then using the underlying message passing to communicate the objects while the other was using only the underlying message passing functions without any compression involved. The difference was in syntax (”pickling” routines started with a lower-case and “non-pickling” ones – with a capital) and in the fact that the latter category, same as in C or Fortran, could only communicate contiguous chunks of memory or derived datatypes.

It was decided to write a simple piece of code that gathered and scattered large arbitrary python arrays in a loop (later on broadcasting was added too). The following routines were tested: scatter/Scatter, gather/Gather and bcast/Bcast. Jupiter cluster was used for running the test. Series of experiments was conducted with different matrix sizes and in different modes: 3-core SMP and 3 core MPP, subscribing only one core per CPU per machine. The full output can be found in the Appendix A: “Output for the Mpi4py Collective Routines Test”. Following are the results of the last two experiments with the largest sets of data to be communicated:
Development Issues and Discussion  High Performance Cortical Simulator

N=200,000

SMP, 3 nodes
scatter time: 1.79250884056
Scatter time: 0.383674144745
bcast time: 2.98570203781
Bcast time: 0.743601083755
gather time: 1.25536489487
Gather time: 0.288822889328

MPP, 3 nodes
scatter time: 3.97047996521
Scatter time: 2.73933315277
bcast time: 6.40952301025
Bcast time: 5.25424790382
gather time: 3.54560399055
Gather time: 3.84726691246

As we can see, SMP benefits from “non-pickling” routines the most. According to the full output set of results, the difference in performance seems to increase with the amount of data sent in each packet. On the contrary, MPP had not shown any significant difference between the performance of the two categories of routines, except in the Scatter/scatter case. It was decided not to introduce any changes into the core Topographica code at this point. However, later on, if it turned out that indeed the pickling overheads were significantly slowing down the simulations and the observations made in this experiment were used to improve the simulator's performance.

Another interesting observation was made over the course of the experiment, in particular in the SMP mode: in the main loop of the testing program between the message passing routine calls there were computations placed to simulate work of a normal parallel application; when the timings output was produced for all MPI nodes, some nodes were showing significantly more time spent doing computations while others – communication, despite the perfectly balanced amount of computations assigned to each node. With that in mind and also the fact that the same behaviour could not be observed in MPP mode, the only assumption that could be made at this point was that the insufficient memory bandwidth of the Jupiter system was causing
Development Issues and Discussion  High Performance Cortical Simulator some of the cores to perform their computations slower than others, while others having completed their portion of work had to wait inside the collective communication routines to synchronise with the rest of the communicator. It was decided to test whether this behaviour would be observed in Bare C and on a larger set of nodes. Again, the work in terms of the number of elements to compute was perfectly balanced. The application was running a loop for a certain number of iterations, broadcasting data from the master node, computing the results and gathering them on the master node. Following is the output for an SMP run, subscribing 6 nodes out of the 8 available in the system; only the results from node 0 and node 1 where the difference is the most apparent are provided here: [12]

########## P0 ##########
dot product: 7.45656
  Broadcasting: 0.00737047
  Dot Product: 4.6
  Gathering: 2.84918

########## P1 ##########
dot product: 7.45768
  Broadcasting: 0.0125265
  Dot Product: 7.44423
  Gathering: 0.000930071

Processes 0 and 5 were dominating the run in that the computations on these have taken 30%-40% less time than on the other nodes. And on the same two processes gathering was taking as much as about 50% of total computation time, a lot more than on the other nodes (notice on process 1 the total Gathering time is less than a millisecond). This behaviour is sometimes referred to as the Early Reduce Problem: some nodes finish their computations quicker than others and have to wait inside the collective routine. Why was broadcasting not taking as much time as gathering to complete? This collective operation was called on each iteration right after gathering and with no work preformed in between. MPI gather was synchronising all processes so that they were entering the broadcasting routine roughly at the same time, hence no significant overhead observed for this operation. Again, this pattern was not observed in MPP mode where each node was using an independent memory system. This could also be confirming the Early Reduce problem.

The results of the early investigations made in this project indicated that the large communication overheads previously observed in Topographica could possibly be attributed to the memory bandwidth problems rather than the actual inter-node communication speed.
Development Issues and Discussion  High Performance Cortical Simulator

7.2 Bare C: Matching the Topographica Simulations in Bare C

In the initial versions of the Matchmodel script and Bare C it was decided to use the lateral-only square all-to-all simulation model. In this model the output activity matrix produced during the first iteration was fed back as the input activity matrix for the next iteration. All-to-all meant that each neuron in the projection sheet was connected to each neuron in the input sheet, which in lateral model was the same projection sheet. In order to simplify the model, only MxM sheets were used for the simulations and no rectangular forms were allowed.

At this stage in order to create a reliable model it was decided to use only serial implementations. After both the Matchmodel script and Bare C were built to match simulations a bash shell script called c_vs_python.sh was created to automate the testing process. The script also measured the total simulation time for both applications and printed into the command line. Many other scripts, including the PBS job submission ones, were based on c_vs_python.sh.

The first problem spotted with the simulation model created at that stage was that due to the lack of external inputs the outputs were very quickly, after only as few as four or five iterations, converging to some static values and never changed after. This made it particularly hard to test whether Bare C and Topographica were still doing the same thing after a number of iterations. In order to fix that problem the purely Lateral model had to be abandoned and the simulations were switched to a less reserved Afferent model. In this model a new input matrix was generated on each iteration and used to produce outputs. Clearly, the inputs at each step of the simulation in Bare C and Topographica had to match and therefore could not be randomised. A simple algorithm called Moving Dot was implemented in both applications to generate input matrices. According to this algorithm the value of 1.0 was moving across a matrix of zeros, starting from the first element (at [0,0]) and making one step forward on each iteration:

\[ [0,0] \rightarrow [0,1] \rightarrow [0,2] \]

and so on. In a similar manner but backwards was moving the value of 0.25. For instance, the input matrix of size 3 by 3 elements produced on the second iteration would be:

\[
\begin{bmatrix}
0.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 \\
0.0 & 0.25 & 0.0
\end{bmatrix}
\]

on the fourth:

\[
\begin{bmatrix}
0.0 & 0.0 & 0.0 \\
1.0 & 0.0 & 0.25 \\
0.0 & 0.0 & 0.0
\end{bmatrix}
\]

and so on.
Development Issues and Discussion  High Performance Cortical Simulator

Despite now using external inputs, the outputs of previous iterations were still taken into account: after the new activity matrix has been computed a hysteresis function was applied to it the following way (TC is a pre-defined time constant):

\[
new\_activity = old\_activity + (new\_activity - old\_activity) \times TC
\]

This was done to make sure that there could be no discrepancies between the outputs of Topographica and Bare C that would only show up if accumulated over time. Initially, the results produced by c_vs_python.sh were hardly readable and contained only a bare minimum of information, but at the same time indicated that a milestone had been reached: both Matchmodel and Bare C were working and given the same inputs producing the same results. Apart from that both simulator were also improved to make use of rectangular sheets. [13]

[jupiter3]$ XXXXXXX: ./c_vs_python.sh -m 170 -i 40
Serial: 59.0316541195
=== Sequential Time ===
Dot Product: 53.9727
------------------------
Total: 53.9792

Here the simulations were run with the input and projection sheet sizes of 170 by 170 elements for 40 iterations. The total simulation time for Topographica was printed in the first line where it says “Serial” and for Bare C on the line where it says “Total”.

Although the C implementation was always faster than the Python code of Topographica, the percentage difference was tolerable and decreased for larger problem sizes. For example, with the sheet density of 50 by 50 elements the difference in timings was about 50% while for the density of 170 by 170 elements it was only about 10%.

7.3 Timing

After the matching simulation code was built the next step was to produce detailed timings for every significant bit of code inside Bare C with MPI wall-clock timers (MPI_Wtime). The new version of the code was printing out into the terminal where and how much of the application time was spent inside each function in the following format:

========== main() ==========
Total: 10.7808

-----------------------------

48
These are the results for the main() function of Bare C. The full output also contained
the timings for activateSeq() (serial activation run) and activateMPI() (parallel
activation run). Apart from that, the timings were also printed into a text file in
machine-readable format where each timer was placed on a new line and separated by a
space with the time value.[14]

Later in the project the code was improved to create XML files with timings. This
format allowed setting simulation parameters such as the sheet density and the number
of CPU cores for each run in form of attributes, providing names for each individual
timing entry and allowed smarter graph plotting. In Bare C XML files were created
using only standard file printing fprintf() routine while in Topographica the
xml.dom.minidom was employed.

It is important to note that Topographica and Bare C were not only timed from the
inside (with MPI_Wtime) but also from the outside – with /usr/bin/time. This was done
to make sure at any stage that the timings produced on the inside reflected the realistic
run time for both applications.

7.4 Building and Running Topographica on HECToR

Building and running Topographica on HECToR unexpectedly turned out to be a
challenging task. The first problem was with building Mpi4py: Portland Group
Compiler (pgcc) is the default one used on this system. Trying to compile Mpi4py with
pgcc resulted in a failure due to unrecognised compiler flags set in the package’s
makefile. The library was designed to be built with GNU compiler (gcc) and although
some of the material on the Internet suggested there was a work-around for building
Mpi4py with pgcc, none of it suggested a working solution.

After swapping the PrgEnv-pgi module with PrgEnv-gnu the build process again
resulted in a failure until experimentally it was found that the xt-mpt module also had to
be replaced by xt-mpich2. This solved the compilation problem and allowed Mpi4py
installing successfully. However, whenever the module was imported in any Python
code it exited with the following error message:

Traceback (most recent call last):
File "mpitest.py", line 1, in <module>
  from mpi4py import MPI
Development Issues and Discussion

High Performance Cortical Simulator

```
ImportError: libgfortran.so.3: cannot open shared object file: No such file or directory
```

The problem was that the file it could not find was located in one of the gcc sub-directories (/opt/gcc/4.5.2/sno/lib) which was not added to the `LD_LIBRARY_PATH` by the module swap command. The export statement adding a new directory to the library path had to be added to the Topographica's PBS job submission script.

The next issue was with Topographica unable to locate its modules. This was fixed by exporting another environment variable:

```
export PYTHONPATH=$PBS_O_WORKDIR/topo:$PYTHONPATH
```

Finally, the last thing that prevented Topographica from running on HECToR was the fact that some of the modules that had to write to disk (in particular, Matplotlib and Weave) could not create their files and directories anywhere inside or outside the project directory. It turned out that the default home directory is not visible for the HECToR back-end system and anything that has to interact with the file system and uses absolute paths must be launched from the `/work` sub-directory of system root. Submitting batch jobs from this directory and adding the following line to the PBS script fixed that problem:

```
export HOME=/work/d04/d04/s0676011 #instead of /home/d04/d04/s0676011
```

More information on these issues and full error print-outs can be found in the project blog.

### 7.5 Problems with Eddie ECDF

Building and running Topographica on ECDF was a lot easier than on HECToR. However, the extremely long turn-around time which sometimes reached several hours for a 10-second long run made any further use of the system for the project unreasonable. Because of this most of the timing results presented in this paper were obtained on HECToR.

### 7.6 Optimising Weights Distribution

After the first timing results have been obtained on Jupiter it became obvious that the connection field weights distribution process was largely inefficient and had to be optimised. In the original model all weights were generated on the master node and then broadcast across the communicator which caused the initialisation time to account for the largest part of the simulation time, especially since the number of iterations used on average in all simulations carried out over the course of this project was a lot smaller than it would be in real-life Topographica runs.

Therefore it was decided to replace in both Topographica and Bare C the old weights generation and distribution mechanism with a new one, where each node was using its
Development Issues and Discussion  High Performance Cortical Simulator
rank value and a simple stepping algorithm to produce the value of each individual
weights of the connection fields allocated to that node. Following is the formula used to
obtain the weights values:

\[
1 + I + \text{start} + (n^*j + k) / \text{tot\_size}
\]

- I – the index of the position of the connection field relative to the \textit{start}
- \text{start} – the index of the first connection field assigned to the MPI node that it
would have if all connection fields were stored on one node. Calculated as rank
multiplied by the number of connection fields assigned per node, which in turn
is the total size of the connection field matrix across all MPI nodes divided by
the number of nodes.
- n – the width of the weights matrix of a connection field
- j – the Y-index of the weight in the weights matrix
- k – the X-index of the weight in the weights matrix
- \text{tot\_size} – the total number of elements in the weights matrix of the connection
field

While the new weights distribution feature was not something that could be committed
as a permanent change to the current parallel implementation of Topographica for
reasons going far beyond the scope of this project, this small optimisation has helped to
save a lot of time while running simulations on HECToR and produce far more
meaningful and readable timing results than it would have been possible to obtain
otherwise.

7.7 PMI Optimisations

The first set of timings produced with Topographica on HECToR when compared with
the results obtained with Bare C helped discovering a few inefficiencies related to the
way the data was distributed and collected on each iteration. In particular, one thing
that attracted attention was the gap between the total simulation (the amount of time
elapsed between the start of the first iteration and the end of the last one) and node
activation time (the amount of simulation time spent computing new activity values on
the nodes, i.e. the total computation time). Given an ideal case of zero communication
or other overheads the simulation total and node activation time should be identical.
Node activation time, in turn, given no hardware-related overheads should scale
linearly. Thus, the bigger the gap between the total simulation and total computation
time, means lower overall parallel efficiency of the simulator. In Bare C, computing
new activity values on the nodes accounted for 93% of the total simulation time, while
in Topographica for only 68%. Using Bare C as performance benchmark for
Topographica it was decided to investigate what was causing the difference to be so
Development Issues and Discussion: High Performance Cortical Simulator severe and if possible eliminate the inefficiency. Because of the fact that everything apart from computing new activity happens on the serial side it was decided to pay most attention to the activation method of the MPI_CProjection proxy-class and the PMI methods which manage the interaction between the serial and parallel sides of the code.

One problem discovered that way was with reconstruction of the output activity array on the master node once parts of it were computed on each individual node and communicated back to the activation method of the parallel projection proxy-class. The reconstruction involved creating a new Python list object and appending the new computed chunks of the output activity at the end of this list. Under the hood it meant re-allocating memory for each activity chunk received from each computing node, thus creating an overhead proportional to the number of MPI nodes used for each simulation. This inefficiency was easily removed by creating an array to hold the output activity once during the run initialisation stage and then once all output activity chunks were computed, placing them between the corresponding to each node start and end indexes. This fix helped reducing the simulation time on 24 cores by half a second.

Another problem was discovered while analysing the PMI invoke() method. This method acts as a bridge between the serial and parallel sides of a PMI-based application by taking care of data distribution through a broadcast, calling the supplied function to be executed in parallel and then gathering the results from all nodes and returning them together with the control flow back to the serial side of the application. After placing timers around all three of these routines and producing the breakdown timing results for PMI invoke it turned out that the collective communication routines, namely bcast and gather were taking twice as much time as the corresponding collectives in Bare C. It was decided to apply the knowledge obtained during the early investigations discussed in section 4.1: the reason for poorer than in C performance was that PMI was using the general-purpose versions of Mpi4py collective routines which have a significantly simpler interface at the cost of some performance loss. The assumption was that creating a new PMI routine called invoke_opt with collectives replaced by more efficient versions of broadcasting and gathering would give near-C communication speed and further reduce the overlap gap between the simulation total and node activation time thus improving the parallel efficiency of Topographica. First it was decided to optimise the gathering operation of the new invoke_opt method: MPI.gather call had to be replaced with MPI.Gatherv. The difference between the interface of these two routines is that while MPI.gather needs only the send buffer, receive buffer and the root to be supplied as arguments, MPI.Gatherv having a very similar syntax to the corresponding MPI routine in C in addition to those three parameters needs to know the displacements (an array of integer values corresponding to the displacements relative to the starting address of the receive buffer at which to place the incoming data from each node), receive counts (an array of integer values corresponding to the number of elements expected to be received from each node) and datatypes of the receive and send buffers. In order to supply these parameters the code of the Topographica functions interacting with PMI also had to be changed. Unfortunately, because of the
Development Issues and Discussion  High Performance Cortical Simulator
way PMI communicates with the nodes which involves sending a pickled function to
be executed in parallel, broadcasting could not be changed in a similar manner.
However, broadcasting accounted for less than 3% of the total simulation time and
improving its performance was not worth the effort of completely redesigning and
reimplementing the operation of PMI invoke routine. Gathering, on the other hand,
before the optimisation was taking twice as much time as the broadcasting and did not
involve communicating anything apart from the arrays of results computed by each
node. After the new PMI invoke_opt function was built and tested with Topographica
the new timings showed that the total amount of simulation time doing gathering
halved making it identical to the respective operation in Bare C in terms of
performance.

After the optimisation work discussed above has been completed the gap between the
total simulation and node activation time went down from 32% to only as much as
14%. Further analysis of the serial-side code and the new set of timing results showed
that there were no more significantly scaling overheads and any difference between the
simulation total and node activation time was mostly caused by Python itself being an
interpreted language. For instance, with only one node used in a 100x100 density run
for 1,000 iterations the total simulation time is on average 1.34 seconds greater than
node activation while on 24 cores with the same run parameters the difference is at 1.82
seconds which can in the most part be attributed to the increased communication costs.
Chapter 8

Conclusions

One of the main challenges of optimising an application by distributing its computations with MPI is to balance good performance with code maintainability. With the current range of HPC technologies the code parallelisation process can be seen as a constant battle between the good engineering practices and the efficiency of the parallel code.

Topographica's optimisation is a long series of efforts aimed to bring the performance of the simulator to the point where it would be possible to simulate parts of the human brain in their entirety, without any approximations or simplifications. The project discussed in this paper has brought Topographica one step closer to that aim.

The approach exercised in this project consisted of identifying the slowest part of the simulator and isolating it from other components (Divide and Conquer), then modelling that part with highly efficient C code to see the ideal performance that could be achieved (Abstraction) and then optimising the relevant components of the simulator to match the performance of the C code up until the point where further optimisations would significantly compromise the engineering practices maintained in the Topographica project.

By optimising the activation function, one of the core components of the simulator, it has been proven over the course of this project that it is possible to achieve the performance scaling comparable to that of C code while still using the highly flexible and easily maintainable Python code.

As a result of this project a strong code base has been created for future work. Apart from that the approach taken in this project has given positive results and should be recommended to anyone who will be further developing the parallel version of Topographica.
8.1 Future Work

The activation function is one of the most computationally intensive components of Topographica and certainly is the most common operation across all its simulation models. However, it is not the only part of the simulator where performance plays the critical role. Other routines that need to be improved are learning and output functions. Depending on the simulation model executed by Topographica, these could be doing significant amounts of computation. Therefore it is important to focus optimisation efforts on improving the performance of these routines.

HECToR was used as a supercomputing resource for this project. However, it could be more practical for CSNG to know how the parallel Topographica code scales on Eddie ECDF which the Group has access to. In future some performance testing should be done on that system.

Finally, although PMI invoke and gathering turned out to be the most important routines for this project’s optimisation effort, there are other functions of this module that could potentially be used a lot heavier in some simulation models. Therefore these routines need to be improved too. Specifically, attention should be paid to:

- The PMI call routine which is still using the less efficient pickling gather
- Broadcasting routine inside PMI invoke and call. This function was not optimised as part of this project and it could potentially become a source of significant scaling overhead depending on the simulation model used. A suggested way of optimising this routine would be breaking it down into two calls: one that would be broadcasting the function to be executed in parallel using the pickling bcast and another that would broadcast the data for computation. The latter call should be replaced with the non-pickling version of bcast same way as it was done in this project for gathering.
High Performance Cortical Simulator
## Appendix A

### Output for the Mpi4py Collective Routines Test

<table>
<thead>
<tr>
<th>N</th>
<th>SMP, 3 nodes:</th>
</tr>
</thead>
</table>
| 10,000 | scatter time: 0.0482320785522  
Scatter time: 0.0166721343994  
bcast time: 0.0553240776062  
Bcast time: 0.041867017746  
gather time: 0.0940809249878  
Gather time: 0.0110039710999 |
|     | MPP, 3 nodes:                                                                 |
|     | scatter time: 0.177076816559  
Scatter time: 0.176827907562  
bcast time: 0.211786031723  
Bcast time: 0.213672876358  
gather time: 0.22430896759  
Gather time: 0.168394088745 |
| 50,000 | SMP, 3 nodes:                                                                 |
|       | scatter time: 0.220118045807  
Scatter time: 0.0812599658966  
bcast time: 0.321494102478  
Bcast time: 0.180558919907  
gather time: 0.190404176712  
Gather time: 0.0511260032654 |
|       | MPP, 3 nodes:                                                                 |
|       | scatter time: 3.98046803474  
Scatter time: 0.681133985519  
bcast time: 1.60409212112  
Bcast time: 1.39482522011  
gather time: 0.849080085754 |
High Performance Cortical Simulator

<table>
<thead>
<tr>
<th></th>
<th>SMP, 3 nodes</th>
<th>MPP, 3 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather time</td>
<td>0.734710931778</td>
<td>0.288822889328</td>
</tr>
<tr>
<td>N=200,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scatter time</td>
<td>1.79250884056</td>
<td>3.97047996521</td>
</tr>
<tr>
<td>Scatter</td>
<td>0.383674144745</td>
<td>2.73933315277</td>
</tr>
<tr>
<td>Bcast time</td>
<td>2.98570203781</td>
<td>6.40952301025</td>
</tr>
<tr>
<td>Bcast</td>
<td>0.743601083755</td>
<td>5.25424790382</td>
</tr>
<tr>
<td>Gather time</td>
<td>1.25536489487</td>
<td>3.54560399055</td>
</tr>
<tr>
<td>Gather time</td>
<td>0.288822889328</td>
<td>3.84726691246</td>
</tr>
</tbody>
</table>
## Appendix B

### Serial Topographica Profiling Results

```
ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
    1     0.000    0.000    22.508    22.508 <string>:1(<module>)
    1     0.135    0.135    22.508    22.508 /disk/scratch/vlcball/topographica-
         dev/topo/base/simulation.py:1132(run)
      6200    0.067    0.000   18.838    0.003 /disk/scratch/vlcball/topographica-
         dev/topo/misc/inlinec.py:137(inline_weave)
      6200   0.146    0.000   18.719    0.003 /disk/scratch/vlcball/topographica-dev/lib/python2.6/site-packages/weave/inline
         e_tools.py:130(inline)
      6200   18.091    0.003   18.567    0.003  {apply}
      4400    0.020    0.000   16.931    0.004 /disk/scratch/vlcball/topographica-
         dev/topo/base/simulation.py:440(__call__)  
      4400    0.063    0.000   16.848    0.004 /disk/scratch/vlcball/topographica-
         dev/topo/base/projection.py:393(input_event)
      4400    0.014    0.000   16.701    0.004 /disk/scratch/vlcball/topographica-
         dev/topo/base/projection.py:521(present_input)
      4400    0.210    0.000   16.686    0.004 /disk/scratch/vlcball/topographica-
         dev/topo/base/cf.py:696(activate)
      4000    0.042    0.000   16.591    0.004 /disk/scratch/vlcball/topographica-
         dev/topo/sheet/lissom.py:89(input_event)
      4400    0.103    0.000   16.230    0.004 /disk/scratch/vlcball/topographica-
         dev/topo/responsefn/optimized.py:36(__call__)
```
References

Bibliography


3. Lisandro Dalcin (2010) MPI for Python

4. Olaf Lenz (2009) PMI - Parallel Method Invocation


8. Suzanne K. Langer (1953) Feeling and Form: a theory of art developed from Philosophy in a New Key


10. Elena Breitmoser, Joachim Hein, Christopher Johnson (2011) HECToR Hardware, Parallel Scaling on Modern Architectures