MODELING OF HIGH PERFORMANCE PROGRAMS TO SUPPORT HETEROGENEOUS COMPUTING

by

FEROSH JACOB

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2013

ABSTRACT

In order to harness the power of multicore CPUs and GPUs, HPC (High Performance Computing) programmers and even end-users need new tools and techniques to express their core problem, divide that core problem into sub problems, allocate computational resources for the sub problems, execute the resources, and collect results. HPC users focus more on the problem domain while HPC programmers are concerned with the code or HPC domain. However, in current practice, the distinction of programmers and users is not clearly delineated because most of the end-users (e.g., scientists who have a computational need) must create and write their own HPC code. Moreover, HPC users also have to maintain the HPC source code to keep abreast with the latest advances, techniques and platforms introduced by the HPC programming community.

The specific aim of this dissertation is to introduce new software engineering ideas (e.g., Model-Driven Engineering (MDE) and Domain-Specific Languages (DSLs)) and supporting tools to assist in the evolution of parallel programs used by HPC programmers, as well as HPC users. In this dissertation, we show that tool support can be provided for HPC programs at different levels of abstraction targeted for a specific set of users. These levels of abstraction are: 1) Code-level, 2) Algorithm-level, 3) Program-level, and 4) Sub-domain-level. We designed, implemented, and evaluated DSLs at each abstraction level to support heterogeneous computing.

Code-level abstraction is very general and it can be applied to any C/C++ program, while algorithm-level abstraction is only applicable for programs implementing MapReduce algorithms. Compared to code-level and algorithm-level abstraction, program-level and sub-domain-level abstractions are very specific and are only applicable to specific domains and users (e.g., Signature Discovery Intiative (SDI) project participants and Nbody solution users). We observed that if the domain is specific, less information is required from the user because the DSLs are domain-aware. If the domain is very general (e.g., in the code-level and algorithm-level abstractions), there are more application usage areas for the DSL, but adoption of the DSL at more general levels requires additional information from the end-users.

DEDICATION

To Angie

To Achan, Amma and Chettan

To Simi, Appu and Ammu

To Harley and Tyler

&

To Gandhi who said *"There is more to life than increasing its speed."*

ACKNOWLEDGMENTS

My sincerest gratitude and appreciation goes to my professor, mentor, and adviser, Dr. Jeff Gray. I am truly indebted for Dr. Gray's involvement in my academic, professional, and social lives for the last four years. Not only did Dr. Gray shepherd me in research, teaching, and outreach activities, but also gave me an outstanding example on how to become a computer science researcher while keeping a social awareness of the impact that I could make in my career at many levels (e.g., both academic and in service to the community). During the whole period of my Ph.D. degree, Dr. Gray gave me many opportunities and kept encouraging me to interact with researchers working in related fields. Dr. Gray always helped in identifying, formulating, and solving research problems, and patiently revised and re-revised the publications and presentations to improve the quality of my research results. In the same way, Dr. Gray celebrated with pride and joy my accomplishments as a student, and he had time for me when I went through difficult times. I will be grateful for all the professional exposure Dr. Gray gave me by sending me to many top-rated conferences in different parts of the world, including India. I will always be thankful for the Christmas I spent with my family after many years that Dr. Gray made possible by sending me to a top HPC conference that was back in my home country. Thank you, Dr. Gray, for motivating me and always being available to help me through this process.

To my fiancée, Angie, I am grateful for the love that we share. To my parents, thank you for your love and continuous support. To my brother, who always made me laugh, think and kept me updated about the latest technological advances. Thank you everyone for your love, support and care as I go through another crucial stage of my life.

I would like to express my gratitude to Dr. Purushotham Bangalore. Without Dr. Bangalore's help on understanding and extending the research work he has done on HPC programs, my Ph.D. idea could not have been implemented and realized. Dr. Bangalore introduced me to the HPC programs through his course on parallel programming. Since then, Dr. Bangalore has always been a great model for me on creating new research ideas, conducting high-quality research, and producing exceptional research papers and presentations.

I am also indebted to the help and guidance I gained from Dr. Nicholas A. Kraft. During the last three years, Dr. Kraft served as a technical resource for me when 'google' failed. His helping hand includes, but was not limited to, helping me to connect to Windows servers from my UNIX-based laptop (my first day on the campus), setting up the printer, configuring svn, sharing source code over web, preparing for faculty-related job interviews; even this dissertation is written using the $\LaTeX$ template written by Dr. Kraft. Dr. Kraft– thank you for the time and effort you spent with me, and for your assistance in serving on my committee.

I also want to thank Dr. Susan Vrbksy, for both of her involved roles as a Ph.D. committee member and graduate program director. I really appreciate her advice and direction on each of the key stages in my graduate study, from taking courses, transferring courses, preparing the qualification exam, forming a research proposal, and completing the dissertation defense. None of these can be accomplished without Dr. Vrbsky's support.

To my other committee members, I am grateful for your willingness to take part in the evaluation of my dissertation research. Dr. Jeffrey Carver– I appreciate our discussions and suggestions about my work and our collaborative work on modeling parallel programs. Dr. Yvonne

Coady– I appreciate your willingness to serve on my committee and for the insight that your expertise in modularity and HPC-systems software has brought to my understanding of the concepts in this dissertation. Dr. Brandon Dixon– I thank you for your time and effort in serving on my committee. I thank you for teaching the CUDA programming course, which provided me with the CUDA programming experience that assisted with several areas of this dissertation.

To Dr. Weihua Geng, thank you for introducing me to Nbody problems and helping me with the mathematical background required to understand such problems. Without your sequential FORTRAN programs for treecode algorithm and direct summation, I could not have implemented their parallel versions.

To Dr. Marjan Mernik, thank you for teaching the informative Domain-Specific Languages class and also for our collaborative works. Your course helped to lay the foundation for my Ph.D. dissertation during an early stage of its formation.

To Adam Wynne, thank you for all the opportunities, when I was working as a Ph.D. intern at the Pacific Northwest National Laboratory (PNNL). It was a very good experience for me and I enjoyed every bit of the work at PNNL. Thank you Dr. Yan Liu, without you and Dr. Jeff Gray, my internship idea would not have been realized.

To Dr. Eugene Syriani, thank you for playing badminton with me, which provided a time of relaxation during my studies at UA.

To Kathy DeGraw, Jamie Thompson, Valerie Trull, and Elizabeth Wallace, thank you for your continuous assistance during my studies at the department.

I would like to thank Dr. Robert Tairas and Dr. Yu Sun, who welcomed me to the software engineering group when I first arrived at UAB. Thank you both for all your help and support!

I am also privileged to be associated with the current software engineering group at UA: Blake Bassett, Amiangshu Bosu, Debarshi Chatterji, Hyun Cho, Christoper Corley, Jonathan Corley, Brian Eddy, Dustin Heaton, Mokter Hossain, Amber Krug, Aziz Nanthaamornphong, and Songqing Yue. I also had a great time getting to know the cloud cluster computing group at UA: Michael Galloway, Ashfakul Islam, Gabriel Loewen, Jeffrey Robinson, and Kazi Zunnurhain. I really enjoyed working with you all and will always remember the good times in the lab. Thank you all for your support and help!

I also would like to acknowledge Ph.D. students Ankica Barišić, Rochelle Elva, and Ajay Panyala for our healthy research discussions. Thank you Ankica, for patiently reading my thesis and your valuable suggestions.

To my friends in Tuscaloosa and Birmingham, Chandrahas Narne, Anand Satyam, Priyank Upadhya, Araty Kurup, Archana Sathyasheelan, Calvin Vreeland, Harshan, and many others. Thank you everyone, I enjoyed every minute I spent with you during this journey that occured across several locations in Alabama!

To Linda Evans and Carol Allbritton, thank you for helping me fix some of the language errors.

Finally, I would like to thank the financial support provided by the Department of Computer Science at UA, as well as the funding support from the AirForce/AFRL STTR. Thanks Dr. Nanbor Wang for the interactions over the past year on the STTR research ideas!

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

3D	Three-dimensional space
ACE	Accelerated Cartesian Expansion
ANTLR	Another tool for Language Recognition
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
BFS	Breadth First Search
BLAST	Basic Local Alignment Search Tool
BT	Block Tridiagonal
CDT	C/C++ Development Tooling
CG	Conjugate Gradient
ClModel	Cluster Model
CPL	Call Processing Language
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DLP	Data Level Parallelism
DM	Distributed Memory
DSL	Domain-Specific Language
DSM	Domain-Specific Modeling

DSML	Domain-Specific Modeling Language
EC2	Elastic Compute Cloud
EMF	Eclipse Modeling Framework
ENBF	Extended Backus-Naur Form
EP	Embarrassingly Parallel
FMM	Fast Multipole Method
FFT	Fast Fourier Transform
GMF	Graphical Modeling Framework
GPGPU	General-Purpose GPU programming
GPL	General-purpose Programming Language
GPU	Graphical Processing Unit
GRAPE	Gravity PiPE
GUI	Graphical User Interface
HPC	High Performance Computing
IDE	Integrated Development Environment
ILP	Instruction Level Parallelism
IS	Integer Sorting
JDK	Java Development Kit
JDT	Java Development Tools
KNN	K-Nearest Neighbor
LDA	Linear Discriminant Analysis
LOC	Lines of Code

MAC	Multipole Acceptance Criterion
MAXPARNODE	Maximum Particles or Bodies Per Node
MDE	Model-Driven Engineering
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
ML	Machine Learning
MPI	Message Passing Interface
NAS	NASA Advanced Supercomputing
NPB	NAS Parallel Benchmark
NUMPARS	Number of Particles or Bodies
OpenCL	Open Computing Language
OpenMP	Open Multiprocessing
PB	Protocol Buffer
PNB	Parallel Nbody
PNNL	Pacific Northwest National Laboratory
PPModel	Parallel Program Modeling
PTP	Parallel Tools Platform
SDI	Signature Discovery Initiative
SDK	Software Development Kit
SDL	Service Description Language
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data

SLURM	Simple Linux Utility of Resource Management
SMA	Shared Memory Access
SM	Shared Memory
SOLOMON	Simultaneous Operation Linked Ordinal Modular Network
SPL	Session Processing Language
SPMD	Single Process Multiple Data
SSH	Secure Shell
SVM	Support Vector Machine
TLP	Thread Level Parallelism
VM	Virtual Machine
WDL	Workflow Description Language
WSDL	Web Service Definition Language
XML	Extensible Markup Language

Chapter 1

INTRODUCTION

High Performance Computing (HPC) has emerged in importance over several decades, which has led to parallel software being widely used in multiple domains. However, HPC applications are often developed by augmenting existing sequential applications with parallel programming constructs for synchronization or message passing [Chandra, Dagum, Kohr, Maydan, McDonald, and Menon, 2001; Chapman, Jost, and Pas, 2007; Squyres, 2003]. With the popularity of multicores and multiprocessors, the need to understand parallel programming techniques will continually emerge as a necessary skill needed by future software engineers, not just those who write HPC programs [Jacob, Whittaker, Thapaliya, Bangalore, Mernik, and Gray, 2010]. A survey of general-purpose computation on graphics hardware reveals that General-Purpose GPU (GPGPU) algorithms continue to be developed for a wide range of problems [Owens, Luebke, Govindaraju, Harris, Krüger, Lefohn, and Purcell, 2007]. To use GPGPUs outside of their intended context, much work is required to make such algorithms accessible to a broader range of software developers. High performance, scalable parallel software is recognized to be extremely difficult to design, develop, debug, evolve, and maintain because of accidental complexities of current practice [Jacob, Gray, Bangalore, and Mernik, 2010; McKenney, 2010]. A lack of tool support and poor integration of new software engineering ideas has contributed to the challenges of most HPC software. Abstractions in parallel programming languages and directives or annotations in sequential code have shown initial promise in reducing some of the burdens of parallel programming [Chandra et al., 2001; Han and Abdelrahman, 2009]. However, even with all of these advances, parallel programming still requires skill beyond that possessed by an average programmer [Asanovic, Bodik, Catanzaro, Gebis, Husbands, Keutzer, Patterson, Plishker, Shalf, Williams, and Yelick, 2006].

In this dissertation, HPC programmers refers to programmers whose areas of expertise are parallelization and optimization of the parallelized code. HPC users represent scientists, mathemeticians, and other forms of users who have deep computational needs to solve domain-specific problems. We argue that HPC users should be able to re-use the code written by HPC programmers, thereby allowing them to focus more on their own area of expertise. Both HPC programmers and HPC users design parallel programs at different levels of abstraction, but share similar challenges, as explained in the following sections.

1.1 Which Programming Model to Use?

Parallel programming enables the decomposition of a large computational domain into numerous sub-domains, and employs a large number of processors to compute the solution simultaneously in parallel on these sub-domains. The Message Passing Interface (MPI) [Gropp, Lusk, and Skjellum, 1994] provides a programming model for sending/receiving point-to-point and group communication messages between parallel computing peer processes. MPI is a common, effective, and powerful programming model for multi-computers and clusters, but it has been repeatedly criticized for its low-level abstractions that are challenging to use [Gropp et al., 1994]. Parallel programming libraries and platforms like MPI have been adopted widely by application developers of HPC systems with MPI as the dominant model. In addition to MPI, there are other emerging parallel libraries, such as Open Multi-Processing (OpenMP) [Chandra et al., 2001] and Compute

Unified Device Architecture (CUDA[1]) for Graphical Processing Units (GPUs). Function calls specific to such parallel API libraries are deeply embedded into HPC code, making it very difficult to replace them with another API library without major effort. Furthermore, a developer would have to maintain the same set of code with these flavors of parallel APIs simultaneously, which induces unnecessary redundant efforts that are also very prone to human errors in maintaining and updating the core algorithms [Jablonski and Hou, 2007]. Therefore, HPC development is often focused on a specific parallel library, which is problematic when developing and maintaining several versions of the same applications that may have ties to different parallel libraries and platforms.

In the current state of practice, in order to write a program that will execute a block of code in parallel, a programmer must learn a parallel programming language and supporting libraries to describe the computation. After the program is executed, the programmer must compare the results with some other baseline representation of the computation in order to optimize performance. An execution time comparison plot of MPI and OpenMP versions is explained in Section 3.1.1. As shown in that Section, there is no consistent behavior; hence, a programmer must execute the application to find out which programming paradigm to use or platform to execute for a given size of input. This highlights the need for creating and maintaining multiple versions of the same program for different problem sizes, which in turns leads to code maintenance issues.

A shared memory (OpenMP) solution may perform better for small problem sizes compared to using a GPU (CUDA), which has a high threshold because of the expensive data transfer operations. As the problem size increases, the GPU programs may become faster than shared memory programs. The problem size for which a GPU performs better than a Central Processing Unit (CPU) differs with each application.

[1] NVIDIA's CUDA, `https://developer.nvidia.com/what-cuda`

In the current practice, programmers must manually create the new version, which may share a substantial amount of code with the original version. Usually the programming models require some additional code to setup the execution in addition to specifying the execution. In the case of OpenMP programs, additional code is required for the library declarations; for MPI, it also includes initialization and finalization code for process instance and process size variables; for CUDA, it further includes code for data transfer operations.

1.2 Why are Parallel Programs Long?

The existing programming styles involve creating programs that have parallel and sequential sections. The parallel sections are either platform-specific or architecture-specific; these details often make parallel programming challenging for average programmers. Often, the parallel section is deeply tangled with the sequential section, which can affect the productivity of the programmer as they are required to unravel the coupling the emerges across different modules [Dijkstra, 1976]. Creating a new version for an existing program targeted to a new platform requires copying the sequential section, rewriting the parallel section, and making necessary modifications to bridge the new parallel code with the existing sequential code. A parallel programming style that is void of any machine-specific details, yet can aid programmers in bridging the parallel and sequential sections of code, has the potential to offer much benefit to future software engineers and those who write HPC software. Currently, many long parallel programs have short parallel sections and long sequential sections as revealed by our analysis [Jacob et al., 2010]. By separating the short parallel sections from the long sequential sections, programmers are freed from the additional task of understanding the complete code, and allowed to focus on the core parallel computation.

An analysis was conducted on ten OpenMP programs collected from various domains. In an OpenMP program, a parallel block is defined by a compiler directive starting with `#pragma`

No	Program Name	Total LOC	Parallel LOC	No: of blocks
1	2D Integral with Quadrature rule	601	11 (2%)	1
2	Linear algebra routine	557	28 (5%)	4
3	Random number generator	80	9 (11%)	1
4	Logical circuit satisfiability	157	37 (18%)	1
5	Dijkstra's shortest path	201	37 (18%)	1
6	Fast Fourier Transform	278	51 (18%)	3
7	Integral with Quadrature rule	41	8 (19%)	1
8	Molecular dynamics	215	48 (22%)	4
9	Prime numbers	65	17 (26%)	1
10	Steady state heat equation	98	56 (57%)	3

Table 1.1: Parallel sections in OpenMP programs

`omp parallel`. The details of the analysis are shown in Table 1.1. The second column of the table presents the name of the program, the third column shows the total Lines of Code (LOC), the fourth column shows the total LOC of the parallel block, and the last column shows the number of parallel blocks in each program. The LOC of the parallel blocks to the total LOC of the program ranges from 2% to 57%, with an average of 19% for the selected ten OpenMP programs. To create a different execution environment for any of these programs, more than 50% of the total LOC would need to be rewritten for most of the programs. Currently, programmers manually copy/paste or rewrite the sequential section in the parallel program. To the best of our knowledge, there is no current support for creating or maintaining the sequential section while rewriting the parallel program for a new platform. This dissertation focuses on new technologies to automate this process.

Reducing the accidental complexities associated with designing parallel applications has the potential to decouple the dependence on specific parallel libraries and platforms. A desirable capability is to maintain a single representation of an HPC application so that the adaptation of the application over time does not cause a divergence between the essences of the application from the specific lower-level implementation dependencies in which it is coded. This is a problem prevalent in the design of existing explicit parallel programs (e.g., MPI-based legacy applications). The research described in this dissertation represents a new approach to refactor design concerns between those explicit to applications, and those implicit to achieving performance-portability-productivity when mapping to concurrent architectures.

1.3 What are the Execution Parameters?

To optimize performance in parallel programs, proper allocation and utilization of the resources are very important. Two main execution parameters of a CPU are the number of parallel instances (threads or processes) and memory usage. Program logic decides the number of parallel instances that can be launched and number, size and type of variables used in the program decide the memory usage of the program. Executing an embarrassingly parallel program (i.e., no dependency between parallel tasks) written in OpenMP (with two threads) on a dual-core machine can give a speedup (ratio of execution time of sequential to parallel) that is nearly double the sequential version, but when the same program is re-run in the same machine with more threads, it cannot further improve the performance. In these cases, it is assumed that the program uses negligible memory. In other cases, memory usage is also a key factor in the program execution. Similar rules apply for MPI programming. In general, a multicore machine can deliver the optimum performance when executed with $n1$ number of threads allocating $m1$ KB of memory, $n2$ number of processes allocating $m2$ KB of memory, or $n2$ number of processes with each process having $n2$ number of threads and allocating $m2$ KB of memory for each process. This information is useful in two ways: 1) It can help the programmer to select resources from the pool of available resources, 2) Running the program with execution parameters that can give optimum execution time.

These factors are relevant in the case of Graphical Processing Units (GPUs) also. As an example, CUDA can spawn only 65,536 threads in one dimension and in case of working in an array of size more than 65,536, a kernel program has to be modified such that threads can work on more than one element. Similarly, trying to allocate more memory than is available may throw a memory allocation error. Within a program, the memory usage can be calculated using static code analysis, from which a GPU can be selected from the resource pool to allocate the memory without error. In addition to the number of instances and memory usage, there are other factors for a GPU; such as, memory transfer operations, block size, number of kernel calls. Hence, for the optimum utilization and allocation of resources, these execution parameters have to be determined. There are execution parameters of every resource when executed with a given configuration and also for a program. The execution parameters of the resources can be estimated from the machine or device files, and a program can be estimated by code analysis. With this information, proper allocation can be done automatically or with minimum human interaction.

1.4 How Technical Details Hide the Core Computation?

A domain can be modeled by creating abstractions and configurations [Fowler, 2005]. Even though there are only a few configuration parameters for most of the parallel programming paradigms, their management involves many technical details. An analysis of CUDA and OpenCL[2] programs reveals how these technical details hide the core computations in HPC programs.

1.4.1 Analysis

An analysis was done for both CUDA and OpenCL to explore the abstraction possibilities observed from their common capabilities [Jacob et al., 2010]. In the CUDA analysis, priority was given to the data flow of a GPU program, while OpenCL was analyzed to identify the templates used in an OpenCL program. Code for the analysis was collected from the code samples from

[2] KHRONOS group's OpenCL, `http://www.khronos.org/opencl/`

```
                // Copy vectors from host memory to device memory
                cudaMemcpy(d_A,h_A,size,cudaMemcpyHostToDevice);
                cudaMemcpy(d_B,h_B,size,cudaMemcpyHostToDevice);

                // Invoke kernel
                VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,d_B,d_C,N);

                // Copy result from device memory to host memory
                // h_C contains the result in host memory
                cudaMemcpy(h_C,d_C,size,cudaMemcpyDeviceToHost);
```

Figure 1.1: Level A: Host code to invoke VectorAdd

the NVIDIA CUDA installation package[3], assumed to be written by expert CUDA and OpenCL programmers.

1.4.1.1 Data Flow Analysis using CUDA

Data flow in the current context can be defined as the flow of data from GPU to CPU (or vice versa), the flow of data between multiple threads, and the flow of data within the GPU (e.g., shared to global, or constant). The flow of data within the GPU should be handled inside the kernel and is left to the programmer's preference; hence, our analysis was focused on the other forms of data flow. If thread synchronization is not used in the kernel code, it is assumed that the program has no data flow between the threads. There is a case where this is untrue: warps of 32 threads are synchronized at the instruction issue level by the warp scheduler, and some kernels take advantage of this fact to keep from having to do an expensive barrier while still sharing data within the warp. The use of local memory is a better indication of data sharing or communication between threads in that case [NVIDIA, 2007].

An example code segment is shown in Figure 1.1. While executing the program in the GPU, the variables `h_A` and `h_B` are used as input variables and `h_C` is used as an output variable. For the analysis, only pointer variables are considered because other variables are available in the GPU without explicitly copying (`N` in the Figure 1.1) the variables. The code segment is from the

[3] NVIDIA installation package, `https://developer.nvidia.com/cuda-downloads`

host program in the code samples of the NVIDIA CUDA installation package. As a general rule, for a GPU call from a CPU, the input variables should be copied from CPU to GPU before the GPU execution and output variables should be copied back to the CPU after execution. There can be exceptions as revealed by the following analysis. Instead of explicit copy operations, memory can be mapped between device and host variables. Because this is an implementation detail, we consider variable mapping also as two copy operations.

```
        if(doMultiBlock){

            MonteCarloKernel<<<gridMain, THREAD_N>>>(
                (__TOptionValue *)plan->d_Buffer,
                plan->d_Samples, plan->pathN);

            MonteCarloReduce<<<plan->optionCount,THREAD_N>>>(
                (__TOptionValue *)plan->d_Buffer, accumN);
        }else{
            MonteCarloOneBlockPerOption<<<plan->optionCount,
                THREAD_N>>>
                (plan->d_Samples, plan->pathN);
        }
```

Figure 1.2: Level C: Host code from MonteCarlo program in the NVIDIA CUDA installation package

Based on the data flow, programs are grouped into three levels. If the kernel code does not communicate between threads, input variables are copied to the GPU before execution, and output variables are copied back to main memory after execution of a single kernel. We classify this form of data flow as level A. Missing thread synchronization or the use of local memory in kernel code is an explicit sign of level A. As an example, the code segment shown in Figure 1.1 belongs to level A. Thread synchronization in level A programs are classified as level B. In this form of data flow, the input variables are copied to the GPU and the output variables are copied back to the CPU before and after execution, but there is synchronization between multiple threads. The distinction is made because GPU code for level A programs can be generated directly from the sequential code. In the case of level C programs, the variables are not copied back and forth. This is mainly

Table 1.2: CUDA program analysis

Program name	Kernel name	IOS	L
simpleMultiGPU	reduceKernel	$\surd\surd X$	A
3DFD	stencil_3D_16x16_order8	$\surd\surd\surd$	B
Marching cubes	classifyVoxel	$X\surd X$	C
	compactVoxels	$X\surd X$	C
	generateTriangles	$\surd\surd\surd$	B
	generateTriangles2	$\surd\surd\surd$	B
AsynchAPI	increment_kernel	$\surd\surd X$	A
cudastreams	init_array	$\surd\surd X$	A
matrixMulDrv	matrixMul	$\surd\surd X$	A
BicubicTexture	d_render	$\surd\surd X$	A
	d_renderBicubic	$\surd\surd X$	A
	d_renderFastBicubic	$\surd\surd X$	A
simpleTexture	transformKernel	$\surd\surd X$	A
MersenneTwister	BoxMullerGPU	$X\surd X$	C
	RandomGPU	$X\surd X$	C
Blackscholes	BlackScholesGPU	$\surd\surd X$	A
simpleTextureDrv	transform	$\surd\surd X$	A
MonteCarloMultiGPU	MonteCarloKernel	$\surd XX$	C
	MonteCarloReduce	$\surd\surd X$	A
	MonteCarloOneBlockPerOption	$\surd\surd X$	A
clock	timedReduction	$\surd\surd\surd$	B
simpleZeroCopy	vectorAddGPU	$\surd\surd\surd$	B
oceanFFT	calculateSlopeKernel	$\surd\surd X$	A
	generateSpectrumKernel	$\surd\surd X$	A
convolutionSeparable	convolutionColumnsKernel	$\surd\surd\surd$	B
	convolutionRowsKernel	$\surd\surd\surd$	B
SobelFilter	SobelShared	$\surd\surd\surd$	B
	SobelCopyImage	$\surd\surd\surd$	B
	SobelTex	$\surd\surd\surd$	B
postProcessGL	cudaProcess	$\surd\surd\surd$	B
cppintegration	kernel	$\surd\surd X$	A
	kernel2	$\surd\surd X$	A
sortingNetworks	bitonicSortShared	$\surd\surd\surd$	B
	oddEvenMergeGlobal	$\surd\surd\surd$	B
quasirandomgenerator	quasirandomGeneratorKernel	$\surd\surd X$	A
	inverseCNDKernel	$\surd\surd X$	A
template	testKernel	$\surd\surd\surd$	B
recursiveGaussian	d_transpose	$\surd\surd\surd$	B
	d_simpleRecursive_rgba	$\surd\surd X$	A
	d_recursiveGaussian_rgba	$\surd\surd X$	A
dwtHaar1D	dwtHaar1D	$\surd\surd\surd$	B
scalarProd	scalarProdGPU	$\surd\surd\surd$	B

done to improve the performance of the GPU programs. In level C, the variables are re-used by other kernels, thereby creating a dependency between different kernels. Using Figure 1.2 as an example, if the `doMultiBlock` is set, it executes the `MonteCarloKernel` kernel followed by `MonteCarloReduce`, without making any memory transfer to GPU or from GPU. In this case, `MonteCarloReduce` is clearly level C. In general, for level C programs, all variables inside the kernel code will not be copied either to the GPU or back to the CPU.

For the analysis, 42 kernels were selected from 25 randomly selected programs that are provided as code samples from the installation package of NVIDIA CUDA. A summary of the results is shown in Table 1.2. The column IOS shows whether or not the input variables are copied before the call, the output variables are copied after the call, and thread synchronization appears in the parallel code, respectively. Conclusions drawn from the analysis are listed below:

- **Automatic Code Conversion**: There are a significant amount of programs; i.e., 48% (20), whose code can be automatically generated if the sequential code is available.
- **Copy mismatch**: There exist programs; i.e., 12% (5), where all the variables in the kernel are not copied. Even though the variables are not copied, memory should be allocated in the host code before the first access.

1.4.1.2 Program Analysis of OpenCL

OpenCL supports the execution of programs in heterogeneous platforms (e.g., both GPUs and CPUs) [Munshi, Gaster, Mattson, Fung, and Ginsburg, 2011]. Every OpenCL program includes a considerable amount of code that is used to initialize a program. As an example, in a program `oclMatrVecMul` from the OpenCL installation package of NVIDIA[4], the following three basic steps are achieved with 34 lines of code: 1) creating the OpenCL context, 2) creating a command queue for device 0, and 3) setting up the program. This amount of verbose coding is necessary even though the above steps are common to most OpenCL programs. The situation is similar to the early stages of GUI programming where several dozens of lines were needed to simply create a new window [Angel, 1999]. Our analysis was done with the intention of finding the frequently used steps of an OpenCL GPU program. If the steps for a general OpenCL program

[4] OpenCL Installation, `https://developer.nvidia.com/opencl`

can be found, templates can be provided that can free the programmer from writing much of the common code manually. Furthermore, one or more steps can be abstracted to standard functions to simplify the development process.

As with the data flow analysis, 15 programs were randomly selected from the code samples that are shipped with the NVIDIA OpenCL installation package. Because this analysis was intended to extract the possible abstractions from an OpenCL program, other details like data flow and kernel code were avoided. The steps for the OpenCL programs are shown in Table 1.3. As can be observed from the table, all of the programs execute steps 1, 4, and 5. Step 2 provides an option to the user of the program to specify the devices on which he or she intends to execute the program. Step 3 is for executing the kernel on multiple devices.

The following conclusions were made from the analysis:

- **Default template**: Every OpenCL program consists of creating a context, setting up the program, and cleaning up the OpenCL resources.

- **Device specification**: Each kernel could be specified with a device or multiple devices in which the kernel is meant to be executed.

1.4.2 Discussion

An abstract representation of the three steps consists of:

1. `copyin(in_paramlist)`,

2. `callkernel(original_paramlist)`, and

3. `copyout(out_paramlist)`.

Table 1.3: OpenCL program steps

No	Step Description
1	create the OpenCL context on available GPU devices
2	if user specified GPU/GPUs create command queue for each
3	else find how many available GPUs create command queues for all create a command queue for device0 or a given device
4	program set up load program from file create the program build the program create kernel run calculation on the queues or queue
5	clean OpenCL resources

In the code, `in_paramlist` refers to the list of variables that have to be copied to the GPU before execution, `out_paramlist` refers to the list of variables that have to be copied back to the CPU, and `original_paramlist` refers to the list of original parameters required for the call. From the CUDA examples, any GPU call could be considered as a three-step process: 1) copy or map the variables before the execution, 2) execute on the GPU, and 3) copy back or unmap the variables after the execution.

From the OpenCL examples, to make OpenCL programming easier and faster, the steps identified could be written as functions and included as libraries with the newly written code. If the user is interested only in single device execution, a single method call `initOpenCL(devices, kernel_name)` which returns the `commandQueues` (used for memory transfer and kernel execution) can abstract all the details required for the execution of the kernel. `devices` is the list of devices on which the user intends to execute the kernel `kernel_name`. In the OpenCL programs analyzed, 33% (5) of the programs used multiple devices while 67% (10) used a single device for execution.

1.5 Who can Use HPC Programs?

HPC program users (e.g., chemists, physicists, biologists, and mathematicians) re-use HPC programs with or without modifications in the programs. Most of the time, the programs and the tools available to them are not at the proper level of abstraction. Two scenarios for users working on the improper level of abstractions are explained in the following subsections.

1.5.1 Nbody Problems

Nbody (or more generally, Many-body) problems is a very popular set of problems, which has applications on molecular dynamics, astrophysics, plasma physics, fluid dynamics, quantum chemistry, and quantum chromo-dynamics [Carlson, Kogut, and Pandharipande, 1983; Jastrow, 1955; Sasai and Wolynes, 2003; Watson, 1953]. In most of the cases, these problems can be a represented by a simple equation that can be solved using different algorithms, based on the solutions required for the context. If context demands an accurate solution, slower direct summation can be used, else faster but approximate FMM (Fast Multi-pole Method) [Huang, Jia, and Zhang, 2009] or Treecode [Barnes and Hut, 1986] solutions can be used. For a user who is not an Nbody problem expert, but wants to solve such a problem, it is often hard to: 1) look for programs based on his or her available resources and context, 2) make modifications according to his or her context, and 3) execute the program. More details about the Nbody algorithms and how this can be improved is provided in Chapter 6.

1.5.2 Signature Discovery Project (SDI)

The goal of the SDI project is to enable scientists to develop and deploy Signature Discovery workflows [Baker, 2012]. Many scientists are not familiar with service-oriented software technologies, a popular strategy for developing and deploying workflows. In this case, they will be forced to seek the help of software developers to make the Web services available in the workflow workbench. This technology barrier may degrade the efficiency of sharing signature discovery algorithms, because any changes or bug fixes of an algorithm require a dedicated software developer to navigate through the engineering process. More details about the SDI project and a solution for scientists can be found in Chapter 5.

1.6 Solution: Applying Domain-Specific Modeling Techniques on HPC Programs

Models are often created as a higher level abstraction of some system design. Our research has led us to the realization of the benefits of adopting a modeling approach [Gray, Tolvanen, Kelly, Gokhale, Neema, and Sprinkle, 2007; Schmidt, 2006] to address the challenges of the HPC community. We identified four different levels of applying modeling techniques to such problems: 1) code, 2) algorithm, 3) program, and 4) sub-domain levels. In code-level modeling, a programmer is given flexibility to insert, update, or remove an existing code section; hence, this technique is independent of any language or execution platform. In algorithm-level modeling, we provide support for developing and deploying programs based on a base-algorithm. Program-level modeling is useful for cases when scientists have to create, publish, and distribute a workflow using existing program executables. For users working on problems with similar behavior, sub-domain-level modeling can be valuable to specify the problem hiding all the language-specific details, but providing the optimum solution. Each of the four levels and their research motivations are explained in the following subsections.

1.6.1 PPModel- Code-level Modeling:

As revealed by our study on benchmark programs [Jacob et al., 2010], sequential code is often duplicated in parallel versions of a program. This can affect code comprehensibility and re-usability of the software. In our investigation [Jacob, Gray, Carver, Mernik, and Bangalore, 2012; Jacob, Sun, Gray, and Bangalore, 2011], we designed a framework named PPModel, which is designed and implemented to free programmers from duplicating the sequential code. Using PPModel, a programmer can separate parallel blocks in a program, map these blocks to various platforms, and re-execute the entire program. PPModel is a graphical modeling tool intended for Eclipse[5] users. A Domain-Specific Language (DSL) called tPPModel is available for non-Eclipse users to facilitate the separation, the mapping, and the re-execution. We illustrate our approach with a case study from a benchmark program, which involves re-targeting a parallel block to CUDA and another parallel block to OpenMP. The modified program gave almost 5x performance gain compared to the sequential counterpart, and 1.5x gain compared to the existing OpenMP version. More details about PPModel are presented in Chapter 3.

1.6.2 MapRedoop- Algorithm-level Modeling

A MapReduce implementation can provide faster solutions for a set of HPC programs [Dean and Ghemawat, 2008]. However, the technical details involved in developing, testing, and deploying (Cloud and Local) solutions can make these inaccessible for an average programmer. Our research shows that many such technical details can be hidden from the programmer, which allows him or her to focus on the core computation. In our investigation [Jacob, Wagner, Bahri, Vrbsky, and Gray, 2011], we designed and implemented a DSL to help programmers implement and deploy MapReduce algorithms. The programs written using our DSL can be deployed and executed in a cloud platform such as Eucalyptus[6] or Amazon's Elastic Compute Cloud[7] (EC2). A detailed explanation of our work on algorithm-level modeling can be found in Chapter 4.

[5] Eclipse, http://www.eclipse.org
[6] Eucalyptus Cloud platform, `http://www.eucalyptus.com/why-eucalyptus`
[7] Amazon's EC2, `http://aws.amazon.com/ec2/`

1.6.3 SDL & WDL- Program-level Modeling:

Domain-agnostic signature discovery [Baker, 2012] entails investigation across multiple scientific disciplines. The breadth and cross-disciplinary nature of signature discovery requires that existing executable applications be integrated with new capabilities into workflows, representing a wide range of user tasks. An algorithm may be written in multiple programming languages for various hardware platforms, and so workflow composition requires integrating executables from any number of remote hosts. This raises an engineering issue on how to generate web service wrappers for these heterogeneous executables and to compose them into a scientific workflow environment (e.g., Taverna[8]). In our investigation [Jacob, Wynne, Liu, Baker, and Gray, 2012a], we worked on two simple DSLs that automate these processes. Our Service Description Language (SDL) describes key elements of a signature discovery service and automatically generates its implementation code. The Workflow Description Language (WDL) describes the pipeline of services and generates deployable artifacts for the Taverna workflow management system. We demonstrated our approach with two real-world workflows composed of services wrapping remote executables. This work was performed at Pacific Northwest National Laboratory (PNNL) and is detailed in Chapter 5.

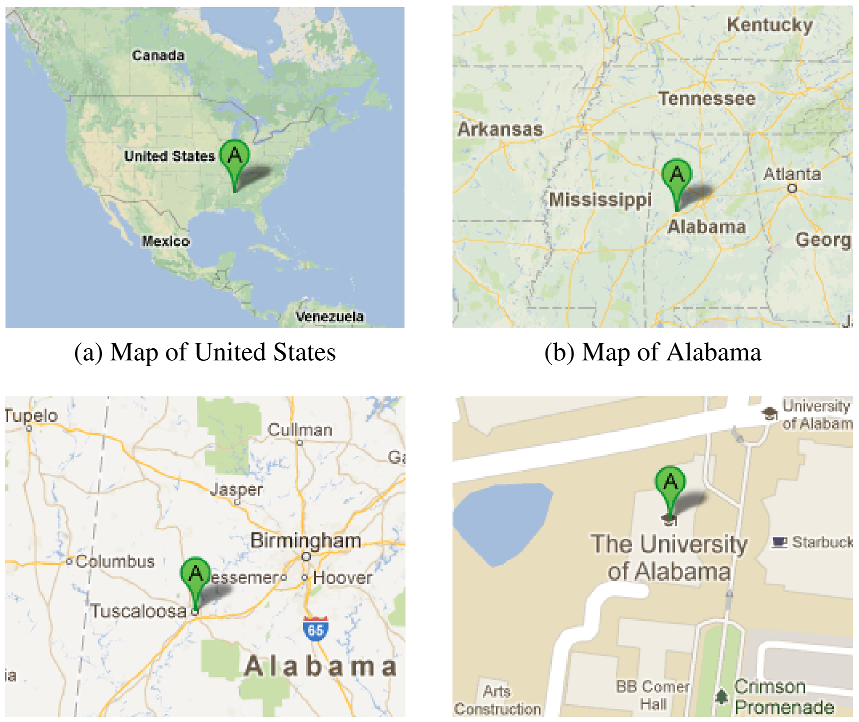
1.6.4 PNBsolver-Sub-domain-level Modeling:

With the advent of multicore processors, parallel computation has become a necessity for next generation applications. It is often a tedious task for domain experts to optimize their programs for a specific platform, algorithm, and problem size. We believe that domain experts should be freed from this task and they should be equipped with tool support to re-use the optimized solutions written by expert parallel programmers. In another investigation, we introduced a two-stage

[8] Taverna Workflow Management System, `http://www.taverna.org.uk/`

modeling approach that allows domain experts to express the problem using domain constructs and re-use the available optimized solutions. This approach has been applied successfully to Nbody problems using a DSL called PNBsolver, which allows domain experts to specify the computations in an Nbody problem without any implementation or platform-specific details. Using PNBsolver, the domain experts are allowed to control the platform and implementation of the generated code. The accuracy and execution time of the generated code can also be fine-tuned based on the parameters provided in PNBsolver. Chapter 6 presents some of the common Nbody interactions and how they can be implemented using PNBsolver.

1.7 Why Different Levels of Abstraction?



(a) Map of United States (b) Map of Alabama

(c) Map of Tuscaloosa (d) Map of The University of Alabama

Figure 1.3: Maps at different levels of abstraction

The need for different levels of abstraction can be explained using a map analogy. Consider the four different maps shown in Figure 1.7. All four maps show the relative location of The

Tool Name	Abstraction Level	Target Users	User Input
PPModel (Chapter 3)	Code	HPC (programmers)	HPC code sections
MapRedoop (Chapter 4)	Algorithm	MapReduce (programmers)	Mapper/Reducer implementations
PPModel (Chapter 5)	Program	SDI project (scientists)	Script meta-data and workflow configuration
PPModel (Chapter 6)	Sub-domain	Nbody solution (users)	Nbody equation

Table 1.4: An overview of abstraction levels in HPC programs

University of Alabama at different resolutions, with each map focusing on a different context. As an example, to a question like “What are your neighboring places?” each map gives a different answer and every answer can be useful based on who is asking the question from a particular context. In a similar way, abstraction levels in HPC programs are intended for different users working on different sets of problems, as shown in Table 1.4.

An overview of the abstraction levels in HPC programs is shown in Table 1.4. The first two columns in the table show the names of the tools we introduce in this dissertation and the abstraction level they represent. The third column indicates the target users and the fourth column shows the required user input for each tool. As an example, PPModel is implemented at code-level abstraction for HPC programmers. At that level, PPModel requires programmers to provide HPC code sections to replace or insert with another code section in the main program. As shown in the table, from PPModel to PNBsolver the Target users are more focused, going from more general HPC users to Nbody solution users. However, the user input required is also reduced from the HPC code sections to the Nbody equation (no code required). This suggests that as the tool is targeted to users in a more refined domain area, we can include more domain information in the tool, which means that the tool requires less input from users. This observation is further substantiated in Chapters 3, 4, 5, and 6.

1.8 Levels of Abstraction in the HPC domain

In this dissertation, four different abstraction levels are introduced for HPC programs. Each abstraction level is explained with a problem scenario, our solution approach, and an evaluation study of the solution on applying to real world applications. All the abstraction levels can be linked to one another using one of the strategies shown in Figure 1.4 and is explained in the following sub-sections.

1.8.1 General to Specific strategy

This dissertation is organized based on this strategy. Code-level abstraction can be applied to any program written in C/C++, hence a code-level solution is applicable to any platform using any algorithm. At the algorithm-level, a restriction is introduced at the algorithm-level. Our solution approach is only applicable to MapReduce algorithms, but it is still general in the sense that many problems from various domains can be solved in optimum time using the MapReduce algorithm. At the program-level, another restriction is introduced in the domain such that it is only applicable to scientists working on a specific project, but this is also general because it is applicable to any executable program within that domain. Finally, at the sub-domain level, the solution is only applicable to a limited set of users working on a very specific problem. Only at the sub-domain level, users have the freedom to specify a problem completely and the tool can generate the optimum solution for the user.

1.8.2 Specific to General strategy

An obvious specific to general strategy to link the four abstraction levels is a bottom up view of the above general to specific strategy. In this alternative strategy, a slightly different ver-

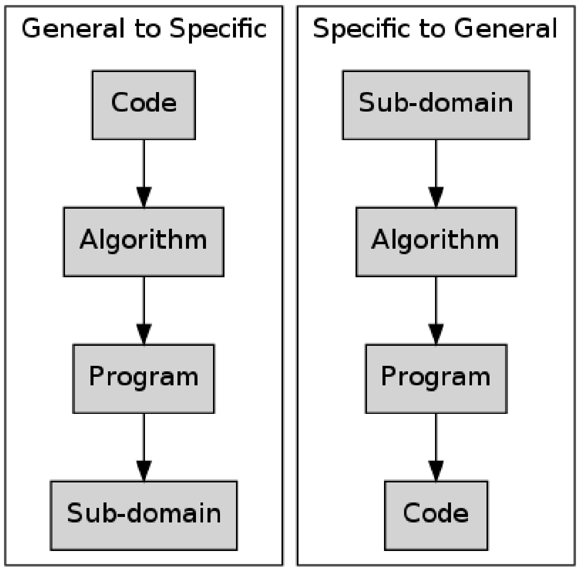


Figure 1.4: Two approaches to link the abstraction levels

sion is introduced. At the sub-domain level, we have used Nbody problems as the case study and it can generate implementations for two different algorithms. In the case of algorithm-level, we generalized this to a more general MapReduce algorithm. At the program-level, we further generalized to any algorithm that is available as an executable program. We provide complete generalization in HPC programs by introducing the code-level abstraction.

1.9 Dissertation Overview

In this chapter, five challenges faced by the HPC community in designing parallel programs are introduced, along with an overview of our solution approach. In Chapter 2, the history and evolution of parallel programs and parallel computers are reviewed. The chapter also includes some related works in the general area of software engineering applications in HPC. Chapters 3, 4, 5, and 6 describes four different levels of abstraction in HPC programs. The future works are summarized in Chapter 7 and this dissertation is concluded in Chapter 8.

Chapter 2

PARALLEL PROGRAMMING: A LITERATURE SURVEY

Multicore processors have gained much popularity recently as semiconductor manufactures battle the “power wall” by introducing chips with two (dual) to four (quad) processors, as well as Graphics Processing Units (GPUs) that have numerous processors per chip. The expectations of increasing clock speed are no longer enough, which is driving the recent trend toward an increase in the number of cores per chip. For a multicore processor with low clock speed to outperform a single core processor with higher clock speed, software must be written in a parallel manner to take advantage of the additional processing capabilities.

In this chapter, Section 2.1 introduces parallel programming by explaining some of the key terms and a brief history. Section 2.2 reviews different types of parallelism and Section 2.3 describes languages used for parallel programming. This chapter concludes with a glossary of parallel terms in Section 2.4.

2.1 Introduction to Parallel Programming

Parallel programming can be defined as the creation of code for computations that can be executed simultaneously. Some of the benefits of parallel programming include improved performance, throughput, and redundancy avoidance.

2.1.1 Parallel Programs and Parallel Computers

S. Gill offered one of the earliest definitions of parallel programming as, “the control of two or more operations which are executed virtually simultaneously, and each of which entails following a series of instructions" [Gill, 1958]. A more recent definition is, "a program in which computations are carried out simultaneously" [Almasi and Gottlieb, 1989]. Parallel programming provides high performance solutions for problems that can be broken into smaller independent problems. For a problem expressed in a traditional programming language like 'C' or Java, a corresponding compiler converts the program to a set of instructions that are executed in the CPU. In the case of a parallel program, instruction sets are generated for the smaller problems and executed simultaneously in different execution units[9]. The execution units can be: 1) a single computer with multiple processors, 2) an arbitrary number of computers connected by a network, or 3) a combination of both.

The first documented approach to make use of parallelism dates back to Charles Babbage [Lovelace, 1843]. The analytical engine designed by Babbage made use of digit-wise parallelism in numerical operations. S. Gill [Gill, 1958] proposed a parallel computer that can be implemented in a single computer either by equipping it with more than one control unit, or by allowing time-sharing of one control unit. Slotnick et. al proposed SOLOMON (Simultaneous Operation Linked Ordinal Modular Network) [Slotnick, Borck, and McReynolds, 1962], a parallel network computer involving interconnections and programming under the supervision of a central control unit. The machine was never built, but the design was used for many later works [Ball, Bollinger, Jeeves, McReynolds, and Shaffer, 1962]. ILLIAC IV [Barnes, Brown, Kato, Kuck, Slotnick, and Stokes, 1968], an earlier SIMD (Single Instruction Multiple Data) parallel-computer with 256 processors.

2.1.2 Flynn's Taxonomy

Flynn's taxonomy is considered one of the earliest classification systems for sequential and parallel programs. Flynn's taxonomy is based on the instruction set and the data in which those

[9] Parallel program definition, `https://computing.llnl.gov/tutorials/parallel_comp`

instructions are executed [Flynn, 1972]. According to this taxonomy, the four classes of programs are: 1) SISD (Single Instruction Single Data) is the traditional uniprocessor with every instruction operating on the same data pool without parallelism; 2) SIMD (Single Instruction Multiple Data) executes the same instruction for a different set of data (e.g., Array processor, Vector computers, GPU); 3) MISD (Multiple Instruction Single Data), a rare architecture (e.g., Space shuttle control); and, 4) MIMD (Multiple Instruction Multiple Data), distributed systems are generally recognized in this category.

SPMD (Single Program Multiple Data or Single Process Multiple Data) [Darema, 2001; Darema, George, Norton, and Pfister, 1988] is considered a sub category of MIMD, where different processors execute different parts of the same program. This is the most common style of parallel programming.

Recent parallel computer memory architectures belong to one of the following categories: 1) Shared Memory [Stenström, Hagersten, Lilja, Martonosi, and Venugopal, 1997] - multiple processors share a common memory and all the processors can access the entire memory as global address space. The first commercial computer to use SMA (Shared Memory Architecture) was IBM PCjr[10]; 2) Distributed Memory (DM) - processors have their own memory and there is no global address space; 3) Hybrid Distributed-Share Memory (DSM)- employ both shared and distributed memory architectures, and can be considered as a set of Shared memory computers distributed over a network. Software DSM systems that extend the underlying virtual memory architecture can be implemented in an operating system, or as a programming library. JIAJIA [Eskicioglu, Marsland, Hu, and Shi, 1999], Kerrighed [Margery, Vallee, Lottiaux, Morin, and yves Berthou,

[10] PCjr, `http://www.adclassix.com/ads/84ibmpcjr.htm`

2003], openMosix[11] [Bilbao, Garate, Olozaga, and del Portillo, 2005], openSSl[12], Strings [Roy and Chaudhary, 1998], Tread marks [Amza, Cox, Dwarkadas, Keleher, Lu, Rajamony, Yu, and Zwaenepoel, 1996] are such software DSM systems. A comparative study can be found in [Lottiaux, Gallard, Vallee, Morin, and Boissinot, 2005].

2.2 Types of Parallelism

Parallelism can be exploited in source code at three levels of abstraction: 1) Instruction, 2) Data, and 3) Thread. Each of these levels of parallelism are explained in the following subsections.

2.2.1 Instruction Level Parallelism (ILP)

In this type of parallelism, the independent instructions are executed simultaneously to keep the execution unit as busy as possible to obtain the complete advantage. The instructions are reordered, grouped and executed in parallel without affecting the program. Recent processors have multi-stage instruction pipelines, where each stage in the pipeline performs a different action to the instruction. A study of ILP compilers can be found in [Schlansker, Conte, Dehnert, Ebcioglu, Fang, and Thompson, 1997].

2.2.2 Data Level Parallelism (DLP)

DLP exploits parallelism in the data. The most common application is image processing; for an operation like contrast, an operation on each pixel is the same and independent of every other pixel. Some operations in a matrix, vector and array can also make use of DLP. There have been efforts [Allen and Kennedy, 1987; Baumstark Jr and Wills, 2002] to extract a data parallel program specification from sequential code and retarget it to data parallel execution mechanisms

[11] openMosix, `http://openmosix.sourceforge.net/`
[12] OpenSSl, `http://openssi.org/index.shtml`

for image processing problems. There have been other efforts to combine the advantages of both ILP and DLP [Espasa and Valero, 1997].

2.2.3 Thread Level Parallelism (TLP)

TLP involves multiple threads (path of execution) simultaneously. TLP can be useful for an application that has independent tasks. TLP is receiving attention with the evolution of multicore and multiprocessor machines that can execute multiple threads in parallel. L. Jack et. al., proposed simultaneous multithreading [Tullsen, Eggers, and Levy, 1998] to convert thread-level parallelism to instruction-level parallelism [Lo, Emer, Levy, Stamm, Tullsen, and Eggers, 1997].

2.3 Parallel Programming Languages

With the realization that complete automatic parallelization of code is not possible at the compiler level [Eigenmann, Hoeflinger, Li, and Padua, 1992], parallel programming languages and libraries were introduced to express parallel algorithms. We classify the programming languages as logical (Prolog[13]), functional (Haskell[14]), imperative (FORTRAN[15]) or applicative (Lisp[16]). This classification is derived from the classification introduced in [Gellerich and Gutzmann, 1996].

2.3.1 Automatic Translation to Parallel Code

Sequential languages are usually based on the Von-Neumann architecture, which is SISD (sequential computer). This makes the conversion of sequential code to parallel code challenging. If the programmer designs and restructures the code to make use of the parallel capabilities of the machine, the speedup of a program can be improved substantially [Karp and Babb II, 1988]. However, there have been efforts to parallelize code automatically [Allen and Kennedy, 1987;

[13] Prolog, `http://www.gprolog.org/`
[14] Haskell, `http://www.haskell.org/`
[15] FORTRAN, `http://gcc.gnu.org/fortran/`
[16] Lisp, `http://clisp.cons.org/`

Wolfe, Shanklin, and Ortega, 1995; Zima and Chapman, 1991]. These works suggest that the involvement of the programmer in the translation can improve the efficiency of the programs.

2.3.2 Computer-Specific Languages

A sequential language can be extended with some constructs to exploit the parallel computer on which it is executed. Some languages provide special syntax to specify arrays that have to be executed in parallel [Reddaway, 1973] and some others even provide syntax for communication between threads or processes. These "high-level assembly languages" [Perrott, 1981] are not portable and can increase the problem complexity [Perrott, 1987].

2.3.3 Architecture-Specific Languages

A more abstract way to define the operational syntax of a language is based on architecture. These languages can provide explicit parallelism, but are not coupled to a machine. MPI[17] [Gropp et al., 1994; Squyres, 2003] targeted to DMs and OpenMP[18] [Chapman et al., 2007] for SMs are examples of this category. Parallaxis (version 2) [Bräunl, 2000] is a structural programming language based on Modula-2[19] for SIMD systems.

2.3.4 Task and Data Parallel Languages

In some cases, a sequential language like 'C' or FORTRAN is combined with a library of communication primitives. As an example, Linda [Gelernter, 1985] uses objects and operations on those objects within a host language to raise the level of abstraction for parallel programming [Deshpande and Schultz, 1992]. PVM [Sunderam, 1990] facilitates concurrent, sequential, or conditional execution of application components in a collection of heterogeneous computing elements connected through a network. Using a data parallel language, a statement sequence can be exe-

[17] MPI, `http://www.mcs.anl.gov/research/projects/mpi/`
[18] OpenMP, `http://openmp.org/wp/`
[19] Modula-2, `http://www.modula2.org/`

cuted in parallel, but on different data. Parallaxis-III [Bräunl, 2000], C-HELP [Dekeyser, Lazure, and Marquet, 1994] are examples of data parallel languages.

2.3.5 Template Languages

Template languages provide a skeleton to execute some specific problem that shares a pattern or similarity with some other well-known algorithm. Programs written using these templates are automatically parallelized and executed. J. Dean proposed a programming model to solve problems in the MapReduce form [Dean and Ghemawat, 2008]. Another related work provides skeletal functions [Darlington, Field, Harrison, Kelly, Sharp, and Wu, 1993] and equation languages [Szymanski and Mueller-Wichards, 1987] are also classified into this category. As the language can provide only a finite set of skeletons or templates, these languages can never be generalized.

2.3.6 Parallel Logic Languages

The mathematical models behind logic programming are first-order logic based on the principle of resolution that allows inferring new propositions from a set of given propositions [de Kergommeaux and Codognet, 1994]. This is achieved with repeated unification of the goals to be proven with facts and rules. P-Prolog [Yang and Aiso, 1986] and BRAVE [Reynolds, Beaumont, Cheng, Delgado-Rannauro, and Spacek, 1988] are two logic languages for parallel computers.

2.3.7 GPU Languages

With the rise in performance, the GPUs originally used for graphics cards have found an application for hosting general-purpose parallel computations, traditionally executed in CPUs. The initial GPU programming languages suffer from portability issues and a steep learning curve [Mark, Glanville, Akeley, and Kilgard, 2003]. CUDA, Microsoft's Direct Compute[20], and OpenCL are the most commonly used frameworks for General-Purpose GPU (GPGPU) programming.

[20] Direct Compute, `http://code.msdn.microsoft.com/directcomputehol`

2.4 Parallel Glossary

2.4.1 Von Neumann Architecture

A computer is said to be a Von Neumann architecture if it uses single memory for both instructions and data [von Neumann, 1993]. It is considered to be one of the initial implementations of a Universal Turing Machine [Turing, 1936].

2.4.2 Speedup Factor and Efficiency

$$S = \frac{T_s}{T_p} \tag{2.1}$$

Speedup [Wilkinson and Allen, 1999] is defined as the ratio of execution time of a sequential program (T_s) compared to the parallel version (T_p) . The maximum speedup is ‘n’ with ‘n’ computation units.

$$E = \frac{T_s}{T_p * n} = \frac{S}{n} \tag{2.2}$$

Efficiency [Wilkinson and Allen, 1999] gives the speedup introduced by each computation unit to the total speedup. The maximum value of efficiency is 1.

2.4.3 Moore’s Law

Moore’s law states that the transistors that can be placed inexpensively on an integrated circuit will double approximately every two years [Moore, 2000]. This has continued for more than 50 years [Mollick, 2006]. However, the physics underlying the semiconductor industry expects several possible barriers to this continued density doubling [Palem and Lingamneni, 2012; Schaller, 1997].

2.4.4 Amdahl's Law

In 1967, G. Amdahl proposed [Amdahl, 1967] that the speedup of a program is limited by the execution time of the sequential fraction in a program. For a program with proportion (p) of the program that can be parallelized and executed with computation units (n), the maximum speedup that can be achieved is

$$S(n)_a = \frac{1}{(1-p)+\frac{p}{n}} \tag{2.3}$$

2.4.5 Gustafson's Law

Amdahl's law does not consider the availability of computation power with the increase of more computation units. It may be possible to solve a larger problem within the same amount of time if there are more resources. The scaled speedup can be calculated using Gustafson's law [Gustafson, 1988]. The scaled speedup, S, is defined as:

$$S(n)_s = \frac{s+n*p}{s+p} \tag{2.4}$$

Where s is the original speedup, p parallel proportion in the program, and n in execution units.

2.4.6 Karp-Flatt Metric

The Karp-Flatt metric [Karp and Flatt, 1990] gives an indication of the extent to which a program can be parallelized. Serial factor (sf), for a program with speedup (s) executed in execution units (n) is defined as:

$$sf(n,s) = \frac{\frac{1}{s}-\frac{1}{n}}{1-\frac{1}{n}} \tag{2.5}$$

Chapter 3

PPMODEL: A CODE-LEVEL ABSTRACTION FOR DEVELOPING AND MAINTAINING HPC PROGRAMS

Parallel programs written in popular parallel programming paradigms have a substantial amount of sequential code mixed with the parallel code. Several such versions supporting different platforms are necessary to find the optimum version of the program for the available resources and problem size. As revealed by our study on benchmark programs [Jacob et al., 2010], sequential code is often duplicated in these versions. This may have the potential to affect code comprehensibility and re-usability of the software.

In this chapter, we discuss a code-level modeling implemented using a framework named PPModel, which is designed and implemented to free programmers from these scenarios. Using PPModel, a programmer can separate parallel sections (hotspots) in a program, map these hotspots to various platforms, and re-execute the entire program. We provide a graphical modeling tool (PPModel) intended for Eclipse users and a Domain-Specific Language (tPPModel) for non-Eclipse users to facilitate the separation, the mapping, and the re-execution. This is illustrated with a case study from a benchmark program, which involves re-targeting a hotspot to CUDA and another hotspot.

3.1 Development and Maintenance of Parallel Programs

Despite the long history of parallel programming, there are no popular editors, tools, or debuggers targeted specifically for parallel programming. The existing tools and editors are extensions of the base programming languages like Java or C. This is mainly due to the nature of the scientific community, the core users of parallel computing.

The unique characteristics of scientific software development places different constraints on the software development process than are present in the more traditional IT domain. Studies show that the scientific community is still not making use of the current solutions offered by the software engineering community [Kelly, 2007]. These constraints affect the applicability of many of the traditional software engineering practices and partially explain why scientists tend not to use them [Carver, 2011; Carver, Kendall, Squires, and Post, 2007; Kendall, Carver, Fisher, Henderson, Mark, Post, Rhoades, and Squires, 2008]. First, the requirements discovery process is difficult because scientific software is often exploring new science. Therefore, requirements cannot be known in advance; they change as knowledge changes [Carver, 2011; Segal and Morris, 2008]. Second, verification and validation is difficult because scientific software is often simulating a phenomenon for which a precise answer is not known, making it difficult or impossible to evaluate the results of the simulation [Carver et al., 2007; Sanders and Kelly, 2008]. Third, because science is the main driver for the software, developers often do not take the time or see the benefit from using best software engineering practices [Carver et al., 2007; Segal, 2007]. Similarly, scientists tend to dislike practices that are process-heavy, preferring more lightweight processes [Carver et al., 2007; Kendall et al., 2008]. Scientists' reluctance to deal with the ever-shifting parallel programming paradigms and platforms, while remaining oblivious to parallel programming models and economic factors, can be some of the other reasons that software engineering practices are not used in the scientific community. These factors also affect traditional software engineers when they parallelize or transform the programs to completely utilize their desktop computation power.

To help these software engineers, we need tools and frameworks that can make the transformation easier.

A survey of general-purpose computation on graphics hardware shows that General-Purpose GPU (GPGPU) algorithms continue to be developed for a wide range of problems [Owens et al., 2007; Rakić, Milašinović, Zivanov, Suvajdin, Nikolić, and Hajduković, 2011]. The existence of abstractions for CUDA, a new GPU programming language, is itself proof that: 1) There exist programmers who write programs in CUDA and are not scientists (some scientists view C as being too high-level to use), 2) There is a need for abstractions in a language like CUDA. Abstractions in parallel programming languages and directives or annotations in sequential code have shown initial promise in reducing some of the burden of parallel programming [Chandra et al., 2001; Fritz, Lucas, and Slusallek, 2004; Han and Abdelrahman, 2009; Jacob et al., 2011; Pike, Dorward, Griesemer, and Quinlan, 2005a]. However, even with all of these advances, parallel programming still requires skill beyond that possessed by an average programmer [Asanovic et al., 2006]. There are several challenges that emerge when programmers design High Performance Computing (HPC) software.

In this chapter, we discuss two aspects of parallel programming that are summarized in the next two sub-sections, which can be improved by applying software engineering techniques.

3.1.1 Code Maintenance

The execution time of a parallel program depends on the platform, program logic, and problem size. An execution time comparison plot of MPI and OpenMP programs from NASA Advanced Supercomputing Parallel Benchmark suite (NPB 3.2) [21] is shown in Figure 3.1. In the graph, the percentage difference of the execution time of an OpenMP version to that of an MPI

[21] NAS Parallel Benchmark (NPB), `http://www.nas.nasa.gov/Resources/Software/npb.html`

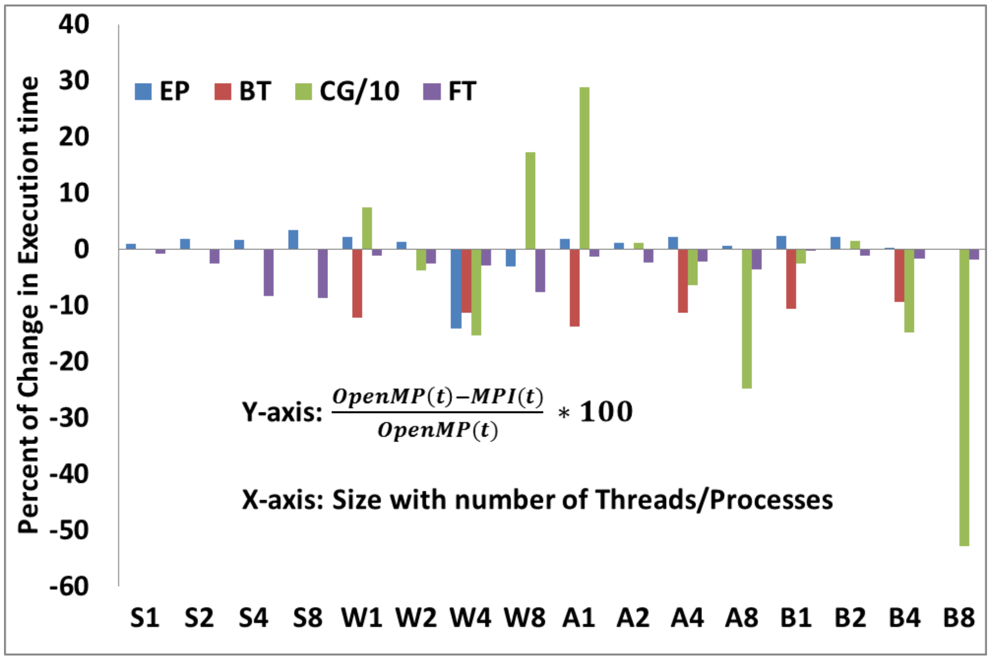


Figure 3.1: Execution time comparison of MPI and OpenMP programs

version is plotted. The benchmarks used are EP (Embarrassingly Parallel), BT (Block Tridiagonal), CG (Conjugate Gradient), and FT (Fourier Transform). Each program was executed for S, W, A, B, C sizes[22] with an equal number of threads (OpenMP) and processes (MPI). A bar in the figure along the positive Y-axis (higher execution time) shows that the MPI version of the program executes faster than the OpenMP version. As an example, for size B, when executed with two instances (B2), benchmarks EP and CG executed faster in MPI. However, with four instances (B4), the version executed faster in OpenMP. Hence, a programmer has to execute the program to find out which programming paradigm to use for optimal performance. This gives rise to the need for creating and maintaining multiple versions of the same program for different problem sizes, which in turn leads to code maintenance issues. For example, to add or update a feature in the program, the programmer has to edit the feature in all of the different versions. The challenges of editing duplicate code manually are described in [Jablonski and Hou, 2007].

[22] Classes used to define size in NAS Parallel Benchmarks (NPB): 1) S (2^{16}), W (2^{20}), A (2^{23}), B (2^{25}), and C (2^{27}).

Popular parallel programming styles involve creating programs that have parallel and sequential sections. We define the sequential section as the code executed by the main thread in shared memory and the master process in a master slave architecture (distributed) as the sequential section, while the remaining code as parallel blocks. Creating a new version of an existing program targeted to a new platform, or creating a parallel version for a sequential version, is usually done in a three-step process: 1) copying the sequential section, 2) rewriting the parallel section, and 3) making necessary modifications to link the newly added code with the existing sequential code. As in the case of NAS parallel benchmarks, OpenMP and MPI versions of the same program share many lines of code. Code duplication is often considered unsafe for the evolution and maintenance of source code [Antoniol, Villano, Merlo, and Penta, 2002; Kamiya, Kusumoto, and Inoue, 2002]. A detailed analysis of one of these programs is included in the case study of Section 6.

3.1.2 Optimum Performance

With the availability of computationally powerful GPUs in desktops, more time efficient parallel programs can be written if programmers can identify parallel blocks well-suited for GPU execution. Because of their highly parallel structure, some parallel blocks on a GPU can deliver a magnitude difference. For the 'random' block explained in the case study in Section 3.6, a Tesla T10 processor provided a 30x increase on performance compared to its sequential version. This may not be the case for all parallel blocks. The performance of a program on a GPU can depend on the type of GPU, implementation, and size of the problem [Pennycook, Hammond, Jarvis, and Mudalige, 2011]. As an example, in the program explained in a future case study (Section 3.6), there are two parallel blocks: 1) a `random` function is embarrassingly parallel (i.e., less communication between threads), and 2) `sort` which involves communication between threads. In this case, a programmer may prefer to execute the embarrassingly parallel section on a GPU and other sections in OpenMP or MPI. To find the optimum execution time of the program, the programmer should have the flexibility to execute any or all of the parallel blocks in platforms of his/her choice.

3.1.2.1 Heterogeneous Computing

In this chapter, heterogeneous computing is discussed in the context of computations in heterogeneous HPC systems. In this context, multiple nodes are used with each node consisting of CPUs with multiple cores and accelerators (e.g., GPUs). In such systems, we need to support not just one programming model, but multiple models simultaneously. As an example, the MPI programming model is used for computation in different nodes, where each process spawned by the MPI model interacts with the accelerators (using a GPU programming language) and the multiple cores (using shared or distributed models).

3.1.3 Solution Approach: Modeling Parallel Programs

We adopted a software modeling and template programming approach to address the challenges of parallel programming as identified in Chapter 1. The result of our work is a modeling tool called PPModel, which is designed to achieve three goals: 1) to separate the parallel sections from the sequential parts of a program, 2) to map and define a new execution strategy for the existing parallel blocks without changing the flow of the program, and 3) to generate code from templates to bridge the parallel and sequential sections. Using PPModel, the parallel part of the program can be separated from the sequential part of the program, re-designed, and then regenerated. With our approach, programmers can switch between technical solution spaces (e.g., MPI, OpenMP, CUDA and OpenCL) without actually changing the base program. We have introduced both graphical and textual implementations of the tool. The graphical model is called PPModel and a Domain-Specific Language (DSL) [Mernik, Heering, and Sloane, 2005] named tPPModel is developed as the textual implementation. Our approach allows a programmer to concentrate more on the essence of the parallelization, rather than focusing on the accidental complexities of language-specific details.

In [Jacob et al., 2011], we first introduced PPModel, which is a graphical modeling tool for Eclipse users and later extended the work to non-Eclipse users using a textual modeling tool [Jacob et al., 2012]. Related works are reviewed in Section 3.2, and PPModel is explained using the Circuit Satisfiability problem in Section 3.3. tPPModel, the DSL for modeling the parallel sections in a program, is introduced in Section 3.4. The implementation of PPModel and tPPModel are described in Section 3.5. A case study showing how tPPModel can be used to improve the execution time of an IS (Integer Sorting) program selected from the NAS parallel benchmarks is shown in Section 3.6 and the chapter is concluded in Section 3.7

3.2 Related Works in HPC Code Modeling

The following research and development efforts associated with parallel programming are surveyed briefly with their objectives, approaches and relation to our effort. In addition to a general survey of related work in parallel programming, this section also summarizes work from software modeling.

3.2.1 Related Works in Parallel Programming

The related works in parallel programming can be classified into two sections: 1) Language transformation, and 2) Raising the level of abstraction. A few examples of each classification are included in the following sub-sections.

3.2.1.1 Language Transformations

There have been various efforts in converting sequential programs to their parallel form [Allen and Kennedy, 1984; Artigas, Gupta, Midkiff, and Moreira, 2000; Di Martino and Keβler, 2000; Kessler, 1995; Martino and Iannello, 1996]. Developing the parallel version of a program from its sequential program with programmer interaction is another field of interest [Appelbe, Smith, and McDowell, 1989; Arora, Bangalore, and Mernik, 2012; Dig, Marrero, and Ernst, 2009; Jacob, Arora, Bangalore, Mernik, and Gray, 2010; Jacob et al., 2010]. Similarly, there are efforts in converting parallel code from one platform to another, with and without programmer interactions. OpenMP to GPGPU [Lee, Min, and Eigenmann, 2009] converts OpenMP programs to CUDA code and [Basumallik and Eigenmann, 2005] converts OpenMP code to MPI.

3.2.1.2 Raising the Level of Abstraction

With the complexities involved in parallel programming, there has been much effort in providing abstraction for parallel programming languages [Han and Abdelrahman, 2009; Pike et al., 2005a; Ueng, Lathara, Baghsorkhi, and Hwu, 2008]. Several approaches [Breitbart, 2009; Fritz et al., 2004; Han and Abdelrahman, 2009; Ueng et al., 2008] were introduced for the abstraction of GPU programs. CGis [Fritz et al., 2004], a data-parallel GPU programming language, allows scientific programmers to specify data-parallel computations at higher level of abstraction. MapReduce [Dean and Ghemawat, 2008] is effective in solving a class of computation intensive problems. For problems that can be solved in Sawzall [Pike et al., 2005a] (a query language written over MapReduce), the resulting code is much simpler and shorter by a factor of ten or more than the corresponding C++ code in MapReduce. MapRedoop [Jacob et al., 2011] assists programmers in developing and deploying MapReduce programs.

However, the goal of our work is to express the parallel part of a program in a way that is separated from the sequential part so as to allow the programmers to focus more on the parallel problem than the program as a whole. Instead of providing abstraction for a language, or tool support for converting from one parallel programming model to another, modeling the parallel sections of the program makes this work unique.

3.2.2 Related Works in Software Modeling

Another category of related works are successful Domain-Specific Modeling (DSM) [Gray et al., 2007] applications [Jiménez, Rosique, Sánchez, Álvarez, and Iborra, 2009; Mathe, Martin, Miller, Ledeczi, Weavind, Nadas, Miller, Maron, and Sztipanovits, 2009], which involve modeling for specific domains and generating low-level software artifacts from the models automatically. Sun et al. presented a tool [Sun, Demirezen, Jouault, Tairas, and Gray, 2008] that uses model-driven engineering techniques to integrate output from different tools into a uniform format. Another work in the same direction involves transforming the Session Processing Language (SPL) code to the Call Processing Language (CPL) using model transformation [Jouault, Bézivin, Consel, Kurtev, and Latry, 2006]. These works focus on performing reverse engineering on text to model the collected data. In PPModel, the essence of the domain is captured through reverse engineering of legacy source code.

There have been a few modeling efforts in the parallel programming domain. The CODE [Browne, Azam, and Sobek, 1989] programming language is based on a generalized dependency graph to express the computation in a unified parallel computation model without any implementation details. GASPARD [Devin, Boulet, Dekeyser, and Marquet, 2002] is another visual parallel programming environment supporting task and data parallelism. In comparison with PPModel, CODE and GASPARD are graphical programming environments, but PPModel is a complete modeling tool to create parallel programs for different platforms without rewriting the entire code.

3.3 Introduction to PPModel

In this section, PPModel is introduced with the Circuit Satisfiability problem example. The problem determines whether a given logical circuit of AND, OR, and NOT gates having `N` variables can be assigned in such a way as to evaluate the circuit to TRUE. The program ('Satisfy.c') is written in OpenMP and has only one parallel block. The goal of the programmer is to rewrite the OpenMP version such that the program can be executed in a cluster using MPI. It is preferred that the goal should be achieved with minimum changes in the base code (OpenMP 'Satisfy.c'). A brief description showing how this can be achieved using PPModel (an online video is available at the project website [Jacob, 2012]) and why we need tPPModel are explained in the following sub-sections.

3.3.1 Circuit Satisfiability Problem in PPModel

The approach used in PPModel is to extract the parallel blocks in a program and re-target the execution such that the rest of the program can be re-used when running in a different platform. This is achieved with a three-stage process. In the first stage, an abstract model is created with only the information about a parallel block; specifications about the platform in which the code will be executed is collected from the programmer in the second stage; and in the final stage, code is generated for linking the parallel blocks with the sequential part of the program and the specific platform. A detailed explanation about each stage for the Circuit Satisfiability problem is provided in the following sub-sections.

3.3.1.1 Model Creation for Circuit Satisfiability Problem

We illustrate our approach using a popular IDE (Integrated Development Environment) - Eclipse. However, the general concept can be implemented with other IDEs. For the Circuit

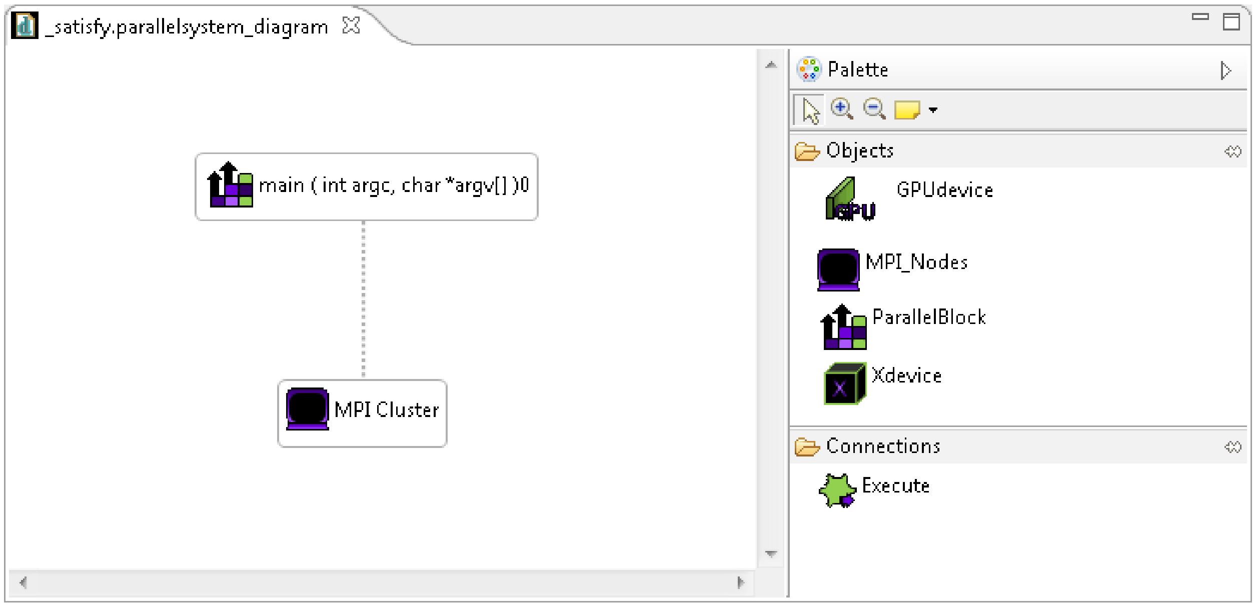


Figure 3.2: Creating models for circuit satisfiability problem

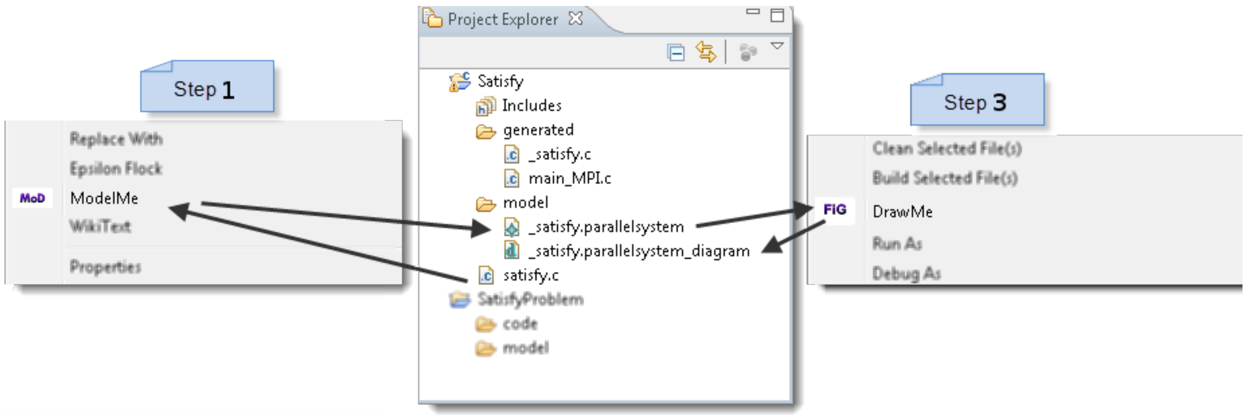


Figure 3.3: Modelling environment for the circuit satisfiability problem

Satisfiability problem, this is achieved in a three-step process as shown in Figure 3.2. In the first step, a model representation of the program is created by right-clicking the program “Satisfy.c” and then selecting “ModelMe.” After this step, a new file is created named “_satisfy.parallelsystem.” This file contains textual information about the parallel blocks in the program. In the second step programmer provides the specification about the platform and the code to be executed in the generated file. In the final step, the newly created file is selected to use the option “DrawMe.” At this stage, another file is created named “_satisfy.parallelsystem_diagram.” This file stores the modeling information of parallel blocks, such as how and where to execute the parallel block. In the future, if the programmer wants to execute the program in an entirely different platform, he or she can still re-use the sequential code, provided the parallel code is written in a language that is a superset of the language in which the sequential code is written. For example, if the programmer desires to rewrite the Circuit Satisfiability program in CUDA or OpenCL (extension of C language), he or she can still re-use the sequential code.

3.3.1.2 Modeling the Circuit Satisfiability Problem

In the current implementation of PPModel, the tool can find all the occurrences of OpenMP parallel blocks. Modeling is designed to assist the programmer in executing the detected parallel blocks in different platforms. The modeling environment of PPModel is shown in Figure 3.3. The Eclipse GMF[23] environment has a palette that consists of Objects and Connectors. Connectors are used to connect different objects. As shown in Figure 3.2, the possible objects are ‘MPI_Nodes,’ ‘GPUdevice,’ ‘ParallelBlock,’ and ‘Xdevice,’ an unknown device. The ‘ParallelBlock’ objects represent parallel blocks in the modeling environment and others represent execution devices. The identifier for the ‘ParallelBlock’ is automatically created from the function name (where the paral-

[23] Graphical Modelling Framework (GMF), `http://www.eclipse.org/gmf`

lel block resides in the sequential program) and a unique identifier. In the example shown in Figure 3.3, a parallel block is found in the `main` function, hence it is named 'main1.' The only defined connector is 'Execute,' which can connect only between a 'ParallelBlock' and an execution device. On creating a link ('Execute') between an execution device and parallel block, a new file is created in the 'generated' folder. The name of the new file is created using the first four letters of the parallel block name and the first three characters of the execution device. The limitation with this strategy is that only one implementation is allowed for a given platform.

3.3.1.3 Code Generation for Circuit Satisfiability Problem

After the programmer has added the required 'Execute' connectors between parallel blocks and the execution devices, control is given to the tool to generate the required code for linking the sequential and parallel files. The programmer right-clicks the 'satisfy.parallelsystem_diagram' and selects the option for code generation. This integrates the code written in 'main_MPI.c' (MPI code) with the program in '_satisfy.c' (base program). More details about this linking is explained in the next sub-section with tPPModel. As an overview, the base program responsible for calling the parallel region is refactored in such a way that the control returns to the sequential code after parallel execution, maintaining the external behavior. The function in the base program that calls the parallel code is unaware of the actual implementation of the parallel section. Hence, parallel sections can be linked with different implementations. The execution plot of the Circuit Satisfiability program for MPI and OpenMP implementations for various sizes of data is shown in Figure 3.4. Both programs are executed on an Intel Quad-Core i5 CPU with 4.0GHz running on Ubuntu 11.04, with four threads in the case of OpenMP and four processes for MPI.

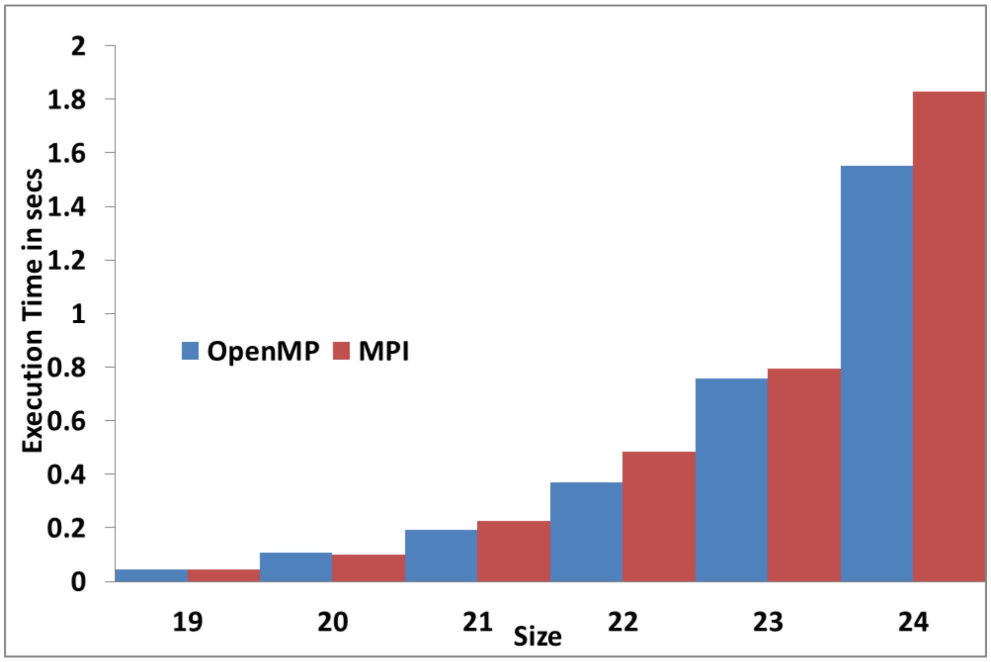


Figure 3.4: Comparison of OpenMP/MPI with problem size

3.3.2 Why tPPModel?

We designed tPPModel (textual PPModel) to solve some of the limitations introduced by PPModel (e.g., PPModel can only work with OpenMP base programs in the Eclipse development environment). These are elaborated in the following sub-sections.

3.3.2.1 OpenMP Base Program

PPModel was introduced to extract the parallel blocks from a parallel program and re-target to another platform. PPModel can automatically detect OpenMP regions in a program and can extract these regions to an abstract function. Hence, PPModel is limited to OpenMP programs and cannot assist programmers who are trying to rewrite a sequential program to parallel, or from a parallel programming paradigm other than OpenMP to another. The flexibility can be improved if programmers can define sections in a program as a parallel section. Because this approach is independent of any language or platform, this can be applied to any parallel program that has parallel and sequential sections.

3.3.2.2 Non-Eclipse Users

Parallel Tools Platform[24] (PTP) is a parallel programming plugin for Eclipse that currently supports only OpenMP and MPI. However, to the best of our knowledge, there is no support available for writing GPU programs in Eclipse. Moreover, it is believed that scientists prefer text editors for software development [Wilson, 2006]. The usability of modeling for some classes of users can be further enhanced if dependency with Eclipse can be removed.

3.4 Using tPPModel

tPPModel is designed to redefine user-specified code sections in a program, such that new code sections can be executed in a different platform. Non-terminal production rules specified as an EBNF grammar for the tPPModel DSL are shown in Figure 3.5 (A simplified version of the grammar showing the core features). tPPModel has three sections: 1) `declare`, 2) `map`, and 3) `execute`. Abstract models are listed in the `declare` section with their concrete implementations specified in the `map` section. Based on the size of the problem or the platform in which the program will be executed, a programmer can specify which implementation to use in the `execute` section.

Refactoring and code generation, performed by tPPModel on each section, is explained using a simple vector addition example. As an overview, the `declare` section refactors the source code, the `map` section generates the templates for different platforms, and the `execute` section generates a 'Makefile' for the current configuration. `declare` and `map` section actions are only executed once for a given configuration. Specifically, the `declare` section refactors the parallel

[24] Parallel Tools Platform (PTP), `http://www.eclipse.org/ptp`

```
grammar tPPModel;

content : declarations mappings execute '.' EOF ;

declarations  : 'declare' namefile '{' 'in' vars 'out' vars '}' ;

mappings  : 'map' ID INTO namefiles ;

execute : 'execute' ID vars ;

namefiles : platform namefile (',' platform namefile)* ;

platform  : 'CUDA'|'OMP'|'MPI' ;

vars  : ID   (',' ID )* ;

namefile  : ID file ;

file  : '[' ID '.' ID']' ;

ID  : LETTER (LETTER|'0'..'9')* ;

INTO  : '<-' ;
```

Figure 3.5: A simplified EBNF grammar for tPPModel

block only if the code is not already refactored, and the map section generates the templates only if the files do not exist.

Assume that there is a serial vector addition program ‘vectoradd.cpp.’ It has three arrays: A, B, and C, each of size SIZE. All of these variables are declared as fields in the program. The goal is to execute the program on a GPU. A programmer first identifies the parallel blocks in the program (a function named “add_vectors”) and marks this function using a #pragma preprocessor statement. This is shown in line 2 (commented line) of Figure 3.6. The original source code in the program is line 2 and lines 14 to 17. The refactored code is shown in the figure and is explained in the following sub-section.

3.4.1 declare Section

The declare section captures more information about each parallel block. The information includes the parallel block name (same name used in the #pragma block), program name, input variables and output variables. With this information, the tPPModel parser can identify the

```
/* A parallel region is defined with name vectorAdd */
//#pragma tppmodel vectorAdd
#pragma TPPMODEL vectorAdd
{
        /* If part is added after refactoring.*/
        #ifdef vectorAdd_DEFINED

        /* The abstract function*/
         abs_vectorAdd(A,B,C,SIZE);

        /* This is the default case or the orginal source code.*/
        #else
          {
            /* core computation in the program */
             add_vectors()
           }
        #endif
 }
```

Figure 3.6: Base program after refactoring

specific parallel block location and extract more information from the program. A symbol table is created from the program. From this table, the type and size (in case of a pointer variable) of each variable can be queried. The size of each variable is calculated using one of two strategies: 1) If the variable is declared as an array, size is the value used in the declaration, and 2) If the variable is declared as a pointer, size is the value used inside the `malloc()` function. While creating symbol tables, the type of a variable is always one of the standard types. The types that are defined through `typedef` and `define` are replaced with their actual types.

3.4.1.1 Why refactor Base Programs?

Software developers prefer their base programs to remain untouched, while at the same time open to extensions [Meyer, 1988]. In a programming language like C, the features like preprocessor directives make parsing hard. This is the same reason the refactoring tools for C are still in the research phase, even though there is more code written in C/C++ than many of the other popular programming languages [Garrido and Johnson, 2002].

As in the case of vector addition, the three arrays are declared each of size, `SIZE` and type `int`. The `SIZE` is defined using a `define` directive. To re-target the implementation to

```

                //Declare section

                declare  vectorAdd [vectoradd.cpp] {

                   //Define input and output variables

                   in A,B,C, SIZE

                   out C

                 }
                // Map section

                map vectorAdd<—CUDA cuda_VA [VAcuda.cu],
                               OMP omp_VA [VAomp.cpp],
                               MPI mpi_VA [VAmpi.cpp]

                //Execution section

                execute VACUDA_EXE cuda_VA.
```

Figure 3.7: tPPModel code for vector addition in CUDA, OMP, and MPI

another platform, we abstract this information to a function so it can be replaced with other implementations. The refactoring introduces a preprocessor conditional statement, through which the execution is routed based on the value of a set of variables. If variable values are not defined in the execution context, the program maintains the original behavior. Each parallel block is given a name such that the programmer can obtain control of the execution of that parallel block. In the case of the vector addition example, `vectorAdd` is the specified parallel block name, `vectorAdd_DEFINED` and `abs_vectorAdd` are generated identifiers from the parallel block name. The variable `vectorAdd_DEFINED` is set through the Makefile and calls the abstract function `abs_vectorAdd`, whose implementation is defined in the `execute` section. The parameters for the abstract function are read from the `declare` section. The complete tPPModel code for the vector addition in three platforms (CUDA, OpenMP, and MPI) is shown in Figure 3.7. In the base program, an `include` statement is also added to include all the abstract functions in that program with the `extern` keyword.

3.4.2 `map` Section

In the `map` section, programmers provide concrete implementations of the abstract function. tPPModel currently supports three platforms: CUDA, OpenMP, and MPI. To provide an implementation, a programmer has to specify the name of the parallel block (`vectorAdd`) and type of implementation (`CUDA, OMP, MPI`). From the map section, the tPPModel parser: 1) generates templates, and 2) identifies the compilation, as well as linking flags. Both of these are based on the type of the mapping used. It is permissible to have more than one implementation for the same type, as explained later in our case study.

For the vector addition example, three mappings are specified for the block `vectorAdd`: 1) Type `CUDA`, identified by `cuda_VA`, and the logic implemented in 'VAcuda.cu;' 2) Type `OMP` (OpenMP), identified by `omp_VA`, and implemented in 'VAomp.cpp;' 3) Type `MPI`, identified by `mpi_VA` in 'VAmpi.cpp'. When the tPPModel file is parsed, if these files do not exist, the parser will generate templates for these programs in the files specified. For OpenMP and MPI, the template is a 'cpp' file with abstract function definitions, `extern` declarations of timers, and `include` statements for the corresponding libraries. MPI templates have initialization of rank and size variables and an empty `if/else` block to distinguish between the master and slave threads. The template details of CUDA are explained in the following sub-section.

Template generation of CUDA programs: CUDA code generated for vector addition using tPPModel is shown in Figure 3.8. Because GPU memory is different from the CPU memory, all the used memory locations from the CPU have to be copied to the GPU before the actual execution, and all the updated GPU memory locations have to copied back to the CPU after execution. In this case, there are three variables: `A, B,` and `C`. Hence, three new variables of the

```
// Device code
__device__ void vectorAdd_main(int* A,int* B,int* C, int opt)
{
}

//This kernel distributes the work irrespective of the size
__global__ void vectorAdd_kernel(int* A,int* B,int* C,int SIZE)
{
    const int       tid = blockDim.x * blockIdx.x + threadIdx.x;
    const int THREAD_N = blockDim.x * gridDim.x;

    for(int opt = tid; opt < SIZE; opt += THREAD_N){
          vectorAdd_main(A,B,C,opt);
    }
}

// Host code
int* abs_vectorAdd(int* A,int* B,int* C,int SIZE)
{
    timerstart("CUDA");
    int* d_A ; int* d_B ; int* d_C ;

    // Allocate vectors in device memory (B and C)
    cutilSafeCall( cudaMalloc((void**)&d_A, sizeof(int)*SIZE)) ;

    // Copy variables from host memory to device memory (B and C)
    cutilSafeCall( cudaMemcpy(d_A,A, sizeof(int)*SIZE,cudaMemcpyHostToDevice) );

    // Kernel call with 480*256 threads
    vectorAdd_kernel<<<480, 256>>>(d_A,d_B,d_C,SIZE);
    cutilCheckMsg("kernel_launch_failure\n");
    cutilSafeCall( cudaThreadSynchronize() );

    // Copy variables from device memory to host memory
    cutilSafeCall( cudaMemcpy(C,d_C, sizeof(int)*SIZE,cudaMemcpyDeviceToHost) );

    // Clean variables, ( B and C )
      if (d_A) cutilSafeCall( cudaFree(d_A));
      timerend();

    return C;

}
```

Figure 3.8: CUDA code generated for vector addition

same size and type are created in the GPU as `d_A,` `d_B`, and `d_C`. GPU execution is performed in these variables and the final result calculated in `d_C` is copied to the CPU variable `C`. As shown in Figure 3.8, function `abs_vectorAdd` calls `vectorAdd_kernel`, a kernel function with 480*256 threads that calls `vectorAdd_main`, a device function, which performs the actual operation. The kernel is implemented with a device function to support larger sizes (CUDA can spawn only 65,536 threads in one dimension). If the `SIZE` is greater than the number of threads, each thread will be working on more than one element. In Figure 3.8, line 24, line 27 and line 38 show memory allocation, memory transfer and clearing of the `d_A` variable, respectively. There are similar statements for `d_B` and `d_C`, but are not shown in the figure. tPPModel can handle scenarios where the size of all pointer variables are not the same. In that case, the last parameter is used to distribute the work between threads.

Template generation for other platforms: The complete template for CUDA can be accessed from the files section at the project website [Jacob, 2012]. A similar approach can be used for generating templates for other platforms. The name, type, and size of input and output variables declared in the `declare` section can be accessed from the template files.

3.4.3 `execute` Section

The `execute` section creates an instance of the given type of abstract function and generates a Makefile to build the project. From the programmer, this section collects the `name` of the executable and its execution path. The execution path is a list of concrete implementations for all the parallel blocks in the project. The concrete implementations used in the execution path should be from the list of function definitions specified in the `map` section.

In the case of the vector addition example, `VACUDA_EXE` is the name of the executable and only one parallel block used in the execution path is `cuda_VA`. Hence, the program builds a binary of name `VACUDA_EXE` using the `cuda_VA` mapping. From the `map` section, `cuda_VA` resolves to a concrete implementation of abstract `vectorAdd`. In the executable, abstract function `vectorAdd` uses the implementation identified by `cuda_VA` and defined in 'VAcuda.cu'. The Makefile generated for the described tPPModel is shown in Figure 3.9. Based on the execution path, all the variables are set to their corresponding values. The Makefile included at the last line of the figure is from the NVIDIA CUDA installation package.

```
#Enable parallel block vectorAdd
COMMONFLAGS +=-DvectorAdd_DEFINED

#Any additional link flags
LINKFLAGS +=

#Executable name
EXECUTABLE  := VACUDA_EXE

# Cuda source files (compiled with cudacc)
CUFILES    := VAcuda.cu

# C/C++ source files (compiled with gcc / c++)
CCFILES    := vectoradd.cpp  pp_timer.cpp

# Rules and targets

include ../../common/common.mk
```

Figure 3.9: Makefile generated for CUDA vector addition

3.5 Implementation Details of PPModel and tPPModel

PPModel and tPPModel are applications of Domain-Specific Modeling (DSM), which is a Model-Driven Engineering (MDE) [Schmidt, 2006] methodology that makes use of a Domain-Specific Modeling Language (DSML) [Lédeczi, Bakay, Maróti, Völgyesi, Nordstrom, Sprinkle, and Karsai, 2001; Sprinkle, Mernik, Tolvanen, and Spinellis, 2009]. A DSML can be used to define a system declaratively using specific domain concepts, directly compute and analyze the domain through model interpreters (i.e., when models are interpreted by an engine written in a generic language like Java), and automatically generate the desired software artifacts by model transformation engines and generators. The metamodel [Atkinson and Kuhne, 2003] specifies the entities, associations and constraints for the specific domain, which can be used to generate a modeling environment. PPModel provides a graphical modeling environment and tPPModel is a textual domain-specific language.

The Graphical Modeling Framework (GMF), a powerful DSM tool in Eclipse, is used to implement PPModel. GMF consists of the three models: 1) Domain model (parallel blocks define the information about the domain); 2) Graphical model (icons) defines the visualization of components in the model editor; 3) Tooling model (actions) defines the editing operations. The separation of these three models makes GMF an extensible modeling editor, which can be used to make changes in one model without affecting other models.

In this case, the structure of parallel programs is the specific application domain and a metamodel defines the formal specification of `declare, map`, and `execute` sections. The models conform to the definition of the metamodel and can be used in computation, analysis and generation of parallel programs.

3.5.1 Implementation of PPModel

The Eclipse C/C++ parser[25] (CDT) has been used to separate the OpenMP preprocessor statements from the rest of the C program. On parsing the programs, a data structure is constructed with the information regarding the symbol table (variable information) and OpenMP preprocessor statements. Each parallel block can be uniquely identified with the function signature and an identifier representing the order of the block in that function. The initialized model can then be edited in order to replace and modify the parallel components. Even though the flow of a program is the same, programmers can provide implementations for the abstract functions.

3.5.2 Implementation of tPPModel

ANTLR[26] is a language tool for developing compilers and interpreters from a grammar containing actions, with support in many languages. ANTLR is used to implement tPPModel. A complete grammar for tPPModel is included in Appendix A.A. StringTemplate[27] is a Java template

[25] C/C++ Development Tooling (CDT), `http://www.eclipse.org/cdt`
[26] ANother Tool for Language Recognition (ANTLR), `http://www.antlr.org`
[27] StringTemplate, `http://www.stringtemplate.org`

engine that strictly enforces model-view separation [Parr, 2004] and generates text output in a specified format.

We use the CDT parser to create the symbol table from the input base program. Using the tPPModel grammar, the input and output variables are identified and more information (size and type) about the variables are fetched from the symbol table. StringTemplate is used to generate the template files for different platforms. Sample template files for a CUDA, OpenMP, and MPI programs are available in Appendix sections A.B.1, A.B.2, and A.B.3. For refactoring the base program, we have implemented a small refactoring library. The refactorings are implemented in this library at the Abstract Syntax Tree (AST) level. Currently, the library supports two refactorings: 1) Inserting an `include` statement (required for inserting the header file of all abstract functions), 2) Replacing a statement with another statement (required to replace the current implementation with an abstract function).

3.6 Case Study: Integer Sorting (IS)

The IS program is taken from the NAS Parallel Benchmarks, version NBP 3.2. The program sorts `N` random integers and can be configured to execute for five different sizes of `N` (S, W, A, B, C). The main issues addressed by our approach are explained in the following sub-sections. All the execution time plots in this case study are plotted with an average value of at least 25 executions.

3.6.1 Code Maintenance by Separating Parallel and Sequential Sections

In the OpenMP, MPI, and Hybrid versions, sequential sections of the program are duplicated. As an example, to convert the serial version of IS to an OpenMP version, the Lines Of Code (LOC) removed is 30 (7% of the total serial LOC) and the number of modified or newly inserted lines is 136 (25% of the total OpenMP code). The new OpenMP version of the program includes a

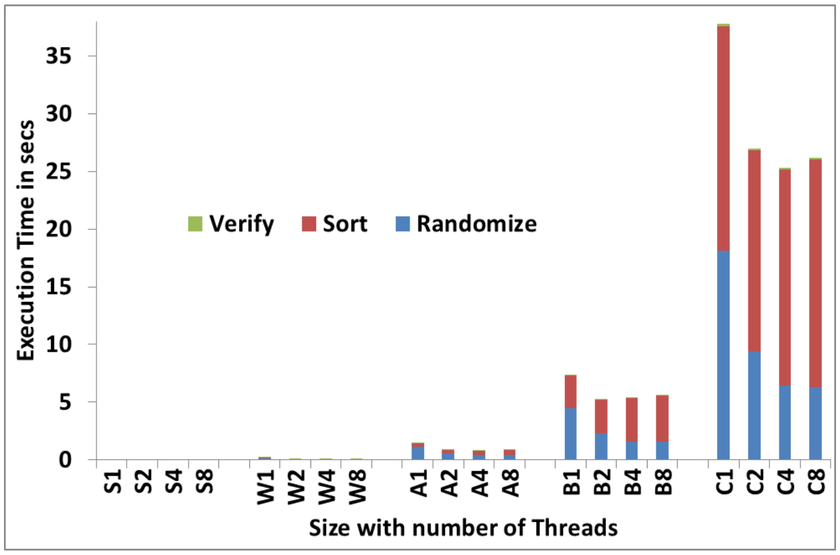


Figure 3.10: Execution time distribution of IS (OpenMP) for different sizes and varying number of threads

method `find_my_seed` which is identical to a function with the same name in the MPI version. More than 50% of the total LOC is duplicated in all the different versions. As shown in Figure 3.1, programs may vary their performance with a change in the size of data. Hence, there is a need to keep all the versions. To add a new feature or change an existing feature, a programmer has to manually edit all the versions, which can lead to maintenance issues (e.g., productivity, correctness). With our approach, the sequential section of the program in stored only in one location.

3.6.2 Optimum Performance through Heterogeneous Computing

The execution time distribution of an IS benchmark for different sizes and a varying number of threads is shown in Figure 3.10. In the plot, B4 refers to execution time of the IS benchmark when executed with four threads and size B. The program has three parallel blocks: 1) Generating random numbers, 2) Sorting the numbers, and 3) Verifying the results. Of these three, the execution time is mostly determined by the random number generation and sorting, as shown in the figure. As clearly seen in the figure, the execution time of the random generation part of the program scales

```

    //Declare section for randomize

    declare  rand_gen [is.c] {

       //Define input and output variables

       in key_array , MAX_KEY , TOTAL_SIZE

       out key_array

     }
     // Declare section for sort

    declare  sort_num [is.c] {

       //Define input and output variables

       in key_array , TOTAL_SIZE

       out key_array

     }
    //Map section

    map rand_gen<-CUDA cuda_rand [RNGcuda.cu],
                    OMP omp_rand [RNGomp.cpp],
                    CUDA cuda_rando [RNGcudao.cu]

    map sort_num<-OMP omp_sort [sortomp.cpp]

      //Execution section

    execute RNGCUDA_SORTOMP_EXE cuda_rando omp_sort.
```

Figure 3.11: tPPModel IS code to use CUDA for random number generation and OMP for sorting

down from C1 to C2 and also C2 from C4, but not from C4 to C8 (i.e., because this is a Quad-core, there is a limited resource). A quick analysis of the source code reveals that the program is well-suited for a platform like a GPU, which can spawn more threads. Hence, the parallel nature of the random generation can be be further utilized for the optimum performance of the program. In the cases of sorting and verification, no such pattern is seen even though it scales well with a large size of data. The goal is to execute the random generation part of the program in a GPU and the rest using the shared memory (OpenMP).

Table 3.1: Speedup for the random function in CUDA

Type	S	W	A	B	C
CUDA	0.087	0.082	2.40	2.61	2.53
CUDA (o)	0.169	2.065	12.6	26.6	38.0

3.6.3 Solution Approach: tPPModel

Because the sequential section of the program is separated from the parallel blocks, it is possible to re-target the parallel block to another platform without modifying the sequential section. For the IS benchmark program, we use CUDA for random number generation and OpenMP for sorting.

The tPPModel code for achieving this is shown in Figure 3.11. The code declares two parallel blocks: 1) `rand_gen`, an abstract function for generating random numbers, and 2) `sort_num`, an abstract function for sorting integers. The parameters of the random number generator include an array, size of the array, and the maximum value of the integer. The parameters of the sorting function are the array and the size of the array. We started from the serial version of the code, marked the `create_sequence` function inside 'is.c' as the `rand_gen` parallel block, and the `rank` function as the `sort_num` parallel block. As shown in Figure 3.11, we tried two different implementations for random number generation in CUDA. The parser generated templates for the OpenMP and CUDA versions. The same code from 'is.c' was added to 'sortomp.cpp'. In the first CUDA implementation (CUDA) we used the same functions as in the OpenMP version by converting them into device functions. In the second implementation (CUDA (o)) we used an optimized random number generator[28] from a molecular dynamics simulation.

[28] Molecular dynamics simulation using CUDA, `http://www-old.amolf.nl/~vanmeel/mdgpu/download2.html`

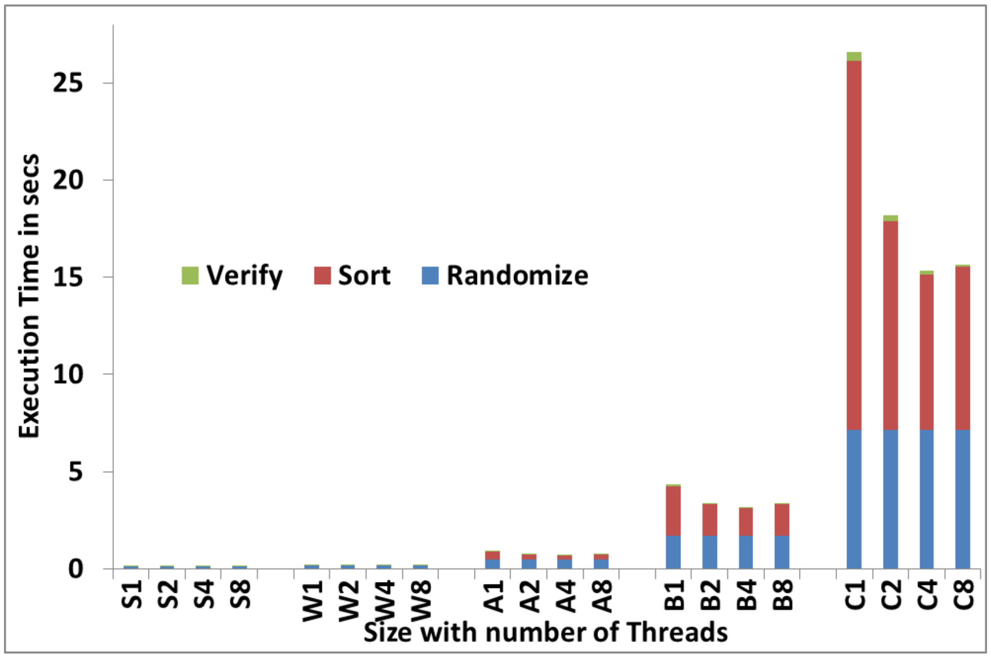


Figure 3.12: Execution time of IS (CUDA + OpenMP) for different sizes and varying number of threads

3.6.3.1 Speedup with tPPModel

The speedup of the random function for two versions of CUDA is shown in Table 3.1. It is shown that the optimized version of CUDA is 38 times faster than the sequential program for data size ‘C.’ As mentioned above, even the optimized version of CUDA is slower than the serial program for problem size ‘S.’

The execution plot for the total IS benchmark for the two versions after using CUDA is shown in Figures 3.12 and 3.13. In Figure 3.10, 50% of the time was spent in generating random numbers while another 50% was spent in sorting. In Figure 3.12, the time spent in random number generation is reduced. It can also be noticed in Figures 3.12 and 3.13 that increasing the threads have no affect in random generation because this part of the program is executed in CUDA and not OpenMP. The threads shown in the figure refer to the OpenMP threads and are not in any

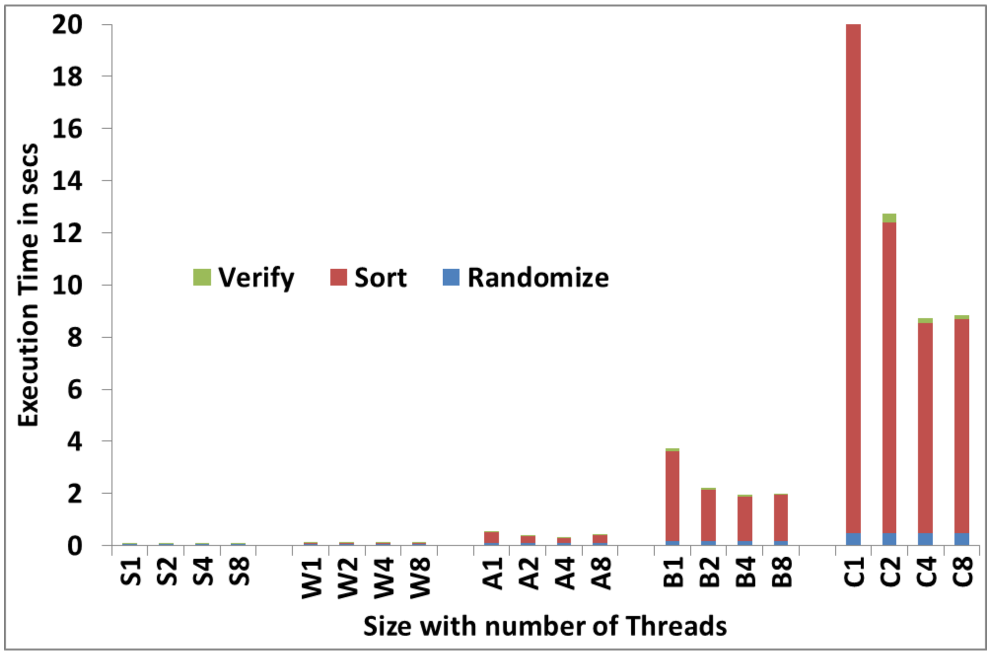


Figure 3.13: Execution time of IS (CUDA (o) + OpenMP) for different sizes and varying number of threads

way related to CUDA threads. As seen from Figure 3.13, the total execution is determined by the sorting time.

A speedup comparison plot for the total IS benchmark with OpenMP (random function implemented in OpenMP), CUDA (random function implemented in CUDA), and CUDA (o) (random function implemented in CUDA (o)) is shown in Figure 3.14. In this program the sorting and verification is implemented using OpenMP. The OpenMP version executed with four threads gives speedup for all sizes (S, W, A, B, and C). The CUDA version gives speedup for only A, B, and C, and CUDA (o) gives speedup for size W, also. The best speedup is observed for size A when executed using CUDA (o). These results further reinforce that the preferred platform for a program is a function of size. The complete source code used for the evaluation is available in the files section of the project website [Jacob, 2012]. Complete example (Program and tPPModel

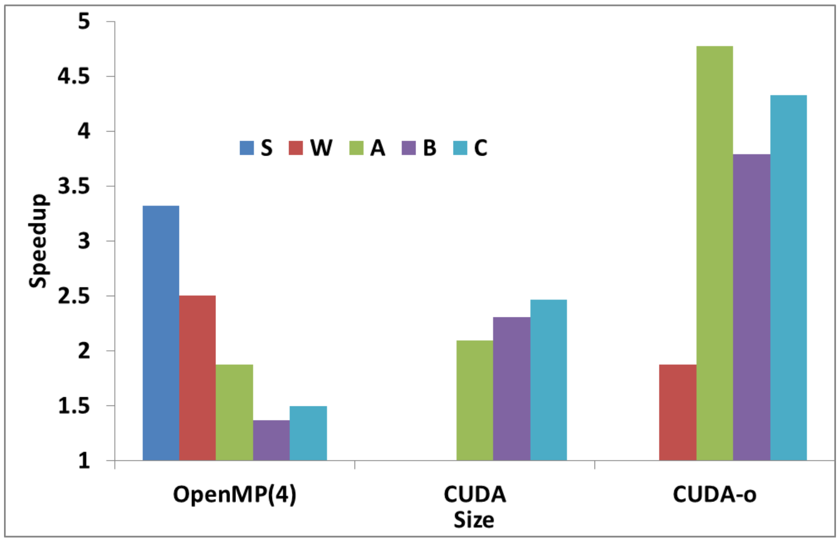


Figure 3.14: Speedup plot for IS, when random function is executed in OpenMP, CUDA, and CUDA (o)

files) for a simple Vector addition using our approach is included in Appendix sections A.C.1 and A.C.2.

3.7 Discussion

In this chapter, we presented a graphical modeling tool named PPModel and a textual representation called tPPModel. The tools are designed to assist programmers while porting a program from a sequential to a parallel version, or from one parallel to another parallel version targeting in a different parallel library. Using PPModel, a programmer can generate OpenMP (shared), MPI (distributed), and CUDA (GPU) templates and can be extended easily by adding more templates for the target paradigm. Our approach is demonstrated with an IS benchmark program through which a programmer can keep the sequential section at one location and can achieve optimized performance by executing one or many parallel sections in a different platform. The benchmark executed 5x faster than the sequential version and 1.5x than the existing OpenMP implementation.

Chapter 4

MAPREDOOP: AN ALGORITHM-LEVEL ABSTRACTION FOR EXECUTING MAPREDUCE ALGORITHMS

In the last chapter, we showed how code-level modeling can help HPC programmers in heterogeneous computing and source code maintenance. In this chapter, we show how algorithm-level modeling can help MapReduce programmers from the accidental complexities involved with existing MapReduce implementations.

This chapter starts with an overview of Cloud computing and MapReduce in Section 4.1 then discusses the motivation for implementing MapRedoop in Section 4.2. The related works are described in Section 4.3 and additional details about MapReduce programs are provided in Section 4.4. A discussion of the benefits of applying MapRedoop can be found in Section 4.5. An evaluation of MapRedoop on several examples is provided in Section 4.6. The final section of the chapter includes a discussion.

4.1 Cloud Computing and MapReduce

Cloud computing provides users with the flexibility of performing high volume computations without the cost of building the required infrastructure. There are several purposes for writing software within a cloud architecture, such as file storage and management [Ghemawat, Gobioff, and Leung, 2003], cloud infrastructure management [Sugiki, Kato, Ishii, Taniguchi, and Hirooka, 2010], and computations of large datasets [Manjunatha, Anderson, Ranabahu, and Sheth, 2011]. The focus of this chapter is on writing MapReduce algorithms, which is a programming model used to solve problems involving large data sets utilizing a cluster of computation nodes where input and output are converted to key/value pairs [Dean and Ghemawat, 2008]. The MapReduce model allows: 1) partitioning the problem into smaller sub-problems, 2) solving the sub-problems, and 3) combining the results from the smaller sub-problems to solve the original issue. The programming model automatically partitions the problems based on the input given (e.g., splitting the input into lines or blocks of lines if given a text file; files if given a directory; and objects if given a list of objects). MapReduce is responsible for solving the sub-problems in parallel and makes the individual results available for the combiner to act upon. From the programmer's perspective, MapReduce involves two main computations:

1. **Map:** implements the computation logic for the sub-problem; and
2. **Reduce:** implements the logic for combining the sub-problems to solve the larger problem.

According to [Dean and Ghemawat, 2008], since its development, "more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day."

Although Google was among the first to implement and utilize MapReduce, Apache (Hadoop[29]) and Stanford (Phoenix) have created open source implementations [Dean and Ghemawat, 2008]. Google identified five areas for the refinement of MapReduce, including customized input and output types and simplified debugging [Dean and Ghemawat, 2008]. Not all MapReduce problems will require the same input type (e.g., text file, associative array, clusters, vectors); therefore, if a programmer wishes to use a different input type, it is the programmer's responsibility to create

[29] Apache's Hadoop, `http://hadoop.apache.org`

a conversion class. Google's solution for customized input and output types is for the programmer to create a custom reader that will convert the input into the required key/value pairs for the mapper, which are called protocol buffers[30] (PB). PBs create data structures automatically, but the programmer must create the appropriate converters. Although this solution did improve Google's MapReduce implementation, the responsibility still lies with the programmer. When debugging a MapReduce algorithm in Google's original implementation, the programmer was forced to debug in the cloud at runtime. Google's solution for simplified debugging is to use a specified flag to run the code locally, at which time the programmer can use the debugging or testing tool of choice [Dean and Ghemawat, 2004]. This gave the programmer the choice to test either in the cloud or locally.

Our solution to the accidental complexity of customized input/output that emerges in MapReduce solutions is to present a DSL, MapRedoop. MapRedoop is our contribution that addresses the second accidental complexity related to simplified debugging of a MapReduce algorithm. MapRedoop is a framework consisting of a DSL and a customized environment within Eclipse. In MapRedoop, the programmer is given the flexibility to implement the map and reduce functions after specifying some of the data structure details, such that the user is oblivious to the setup required to execute the program and any possible type mismatches that might occur. Because MapRedoop is a plug-in for Eclipse, a programmer may develop MapReduce algorithms within an IDE, in which the programmer has the option to execute the code running Hadoop either in a cloud or on a local machine instance. Section 4.4 describes how MapRedoop is used to address the challenges of implementing MapReduce algorithms.

[30] Protocol buffers, `http://code.google.com/apis/protocolbuffers/`

4.2 Motivating Scenario: WordCount Example

As a motivating need for the approach described in this chapter, our first experience with writing a MapReduce algorithm was confusing and frustrating. We wanted to modify the WordCount[31] example algorithm to compute the probabilities of bigrams beginning with a specific word occurring within a given text file. The algorithm itself was straightforward, but the environment, particularly testing, data types, and class interactions presented challenges to our implementation.

In our initial solution using the pseudocode in Algorithm 4.1, we encountered several accidental complexities that contributed multiple challenges during the development process. The primary issue we experienced was not demonstrated until we ran the code: we received zero for each of the probabilities. We knew what the proper data type for each variable should be (int, double, float), but Hadoop did not accept any of these types. Hadoop required the use of internal data types; therefore, we had to change each instance of the `IntWritable` data type to the `FloatWritable` data type. After we corrected the class and attempted to execute the program, we received a type mismatch error ("Type mismatch in key from map"), and we realized we needed to alter the data type in additional classes (`Driver`).

A secondary issue we encountered was that the output of the mapper must match the type of the input of the reducer. Additionally, if a partitioner or combiner were included in our program, the output of the mapper would have to match the input of the partitioner, the output of the partitioner would have to match the input of the combiner, and finally, the output of the combiner would have to match the input of the combiner. This is a simple issue to which a programmer acclimates after writing a few MapReduce programs; however, we feel this is yet another accidental complexity. The situation differs when reading data from a text file versus reading the text file itself; Hadoop

[31] WordCount program, `http://wiki.apache.org/hadoop/WordCount`

```
Input - $R$: Text files and a word $w$
Output - Bigram probability of the word in the files
---
mapper
 if word = $w$
    output ($bigram$, 1)
    output ($word$, 1)
 endIf
---
reducer
 while $morevalues$
    $sum = sum + values$
 output ($sum/total$)
```

Figure 4.1: Bigram probability estimation

reads text files slowly when compared to sequential files. Sequential files support splitting up data for parallel jobs, even if they are compressed, making them a sufficient point of contact for the Hadoop framework. Therefore, the data from a text file must be converted to data in a sequential file to decrease the program's run-time. This experience caused us to ask a few questions:

1. How can the programmer easily identify the required input requirements?
2. Should the programmer need to be concerned with data types?
    - Is there a way the programmer can use familiar data types and then use Hadoop to internally convert these data types appropriately?
3. With what other issues should the programmer not have to be concerned?

We feel there are three primary areas about which the MapReduce programmer should not have to be concerned:

1. **Input structure:** The current frameworks, which claim to address these issues, have not solved the issues entirely. For example, a K-means [Kanungo, Mount, Netanyahu, Piatko,

Silverman, and Wu, 2002] program executed in Mahout[32] (a library for machine learning and data-mining programs) expects a vector as an input; however, if the input structure differs, the programmer has to rewrite the file to match the structure that Mahout supports.

2. **Improper level of abstraction:** Ideally, the programmer should have the ability to focus solely on the map and reduce functions rather than the implementation details. Currently, the MapReduce programmer has to search within the source code to identify the mapper and the reducer (and depending on the program, the partitioner and combiner). After identifying these classes, the programmer has to delve deeper into the code to determine the proper inputs. The key challenge is that there is no central place where the required input values for each of these classes can be identified in order to increase program comprehension.

3. **Improper validation:** Because the input and output for each class (mapper, partitioner, combiner, and reducer) are declared separately, mistakes (such as the data type issue we mentioned previously) are not identified until the entire program is executed. The programmer should have the ability to execute each class separately for validation purposes.

Upon identifying these three primary issues, we built a tool to aid the programmer in creating MapReduce programs. Our tool and its implementation are described in Section 4.5.

4.3 Related Works in MapReduce Algorithms

Recently, there have been many studies about the usefulness of DSLs in Cloud Computing, including work by the following: [Kromer, 2009; Low, Gonzalez, Kyrola, Bickson, Guestrin, and Hellerstein, 2010; Manjunatha et al., 2011; McCullough, 2011; Pike, Dorward, Griesemer, and Quinlan, 2005b; Ranabahu, Sheth, Manjunatha, and Thirunarayan, 2010; Sugiki et al., 2010].

[32] Mahout library, `http://mahout.apache.org/`

Some of the efforts that are most relevant to our work are briefly presented and discussed in this section.

Kumoi is an embedded DSL for virtual data center management. Kumoi's primary goal was to "provide maximum management efficiency for experienced administrators" [Sugiki et al., 2010]. Kumoi utilizes a DSL to allow data center administrators to write complex management scripts while hiding unnecessary details. This tool reduced the number of lines of code required to deploy Virtual Machines (VMs) by 71%, balance VMs by 81%, and shutdown VMs by 98% as compared to the same scripts written in Libvirt[33] (Java-base API). This tool is similar to our MapRedoop in that it simplifies the code necessary for programmers to write; however, Kumoi's application domain is different from MapRedoop.

OptiML [Sujeeth, Lee, Brown, Rompf, Chafi, Wu, Atreya, Odersky, and Olukotun, 2011] and GraphLab [Low et al., 2010] are two DSLs in the machine learning domain. OptiML automatically analyzes and optimizes the domain specifications provided by the user and generates CUDA code. The code generated shows significant improved compared to explicitly parallelized Matlab[34] code. According to Low et al., "GraphLab achieves a balance between low-level (PThreads) and high-level (MapReduce) abstractions" [Low et al., 2010]. The developers of GraphLab report a significant speedup among the various ML (Machine Learning) algorithms tested, and thus, the goal of balancing high-level and low-level abstractions while improving efficiency was met. MapRedoop differs from GraphLab and OptiML in that MapRedoop is based on creating an abstraction for MapReduce and hence MapRedoop is more general compared to these two DSLs.

Manjunatha et al. [Manjunatha et al., 2011] presented a DSL within the Metabolink Toolkit

[33] The virtualization API, `http://libvirt.org/`
[34] MATLAB, `http://www.mathworks.com`

for scientists to analyze Nuclear Magnetic Resonance based metabolomics data. Although the Metabolink Toolkit can be implemented on multiple platforms including Apache's Hadoop (taking advantage of MapReduce) and Microsoft's Azure[35]. Similar to OptiML and GraphLab, this work also has commonalities with MapRedoop, but the primary difference between each of the described tools and the approach presented in this dissertation is the tools described above have been created for a very specific domain. The goal of MapRedoop is to provide an easier method to write MapReduce algorithms in a domain-independent manner.

Three additional tools presented by [Kromer, 2009; McCullough, 2011; Pike et al., 2005b] are very similar because they provide a DSL to write MapReduce algorithms more easily. These tools provide high-level abstractions to simplify the map and reduce functions. In contrast, MapRedoop simplifies the process of creating a MapRedoop program by generating Java template classes and leaving the map and reduce function implementation to the programmer.

4.4 A Study of MapReduce Programs

While implementing a MapReduce solution for a given problem, the programmer has to setup the data to allow the MapReduce framework to process the data efficiently, but the output generated from the framework may need to be converted. Section 4.4.1 explains our classification of MapReduce programs for generalization. Section 4.4.2 describes a data structure analysis we performed on MapReduce programs and the summary of our analysis is provided in Section 4.4.3.

4.4.1 Classification of MapReduce Programs

The number of stages before and after the MapReduce execution differs based on the problem to be solved. As in the case of Hadoop, if the MapReduce implementation requires reading and writing to and from sequential files, more stages are required. Sequential files are flat files

[35] Window's Azure, `http://www.windowsazure.com/`

containing data in the key/value format. The input text files are read as Java objects and are written to sequential files before the MapReduce operation and converted back to text files after (or any other structure as needed).

To get a better understanding of the process, a detailed explanation is given for some of the common types of MapReduce problems. We categorize the MapReduce programs into three classes:

1. **Class 1:** Programs that take text files as the input and read tokens (e.g., WordCount, Bigram, InvertedIndex [Lin and Dyer, 2010]);
2. **Class 2:** Programs that implement a machine learning algorithm (e.g., Clustering algorithms, Classifier algorithms); and
3. **Class 3:** Programs that create a data structure (graph) internally for computation, e.g., PageRanking [Lin and Dyer, 2010], and Breadth First Search (BFS). Please note: although some clustering algorithms take text files as input, we include it in the third class because the text files must first be converted to sequential files before passing to the MapReduce framework.

Programs that cannot be included either in the first or second classes are included in the third class (e.g., matrix multiplication or matrix transposition). The examples to demonstrate the process involved in implementing a MapReduce problem were carefully chosen such that the problems represent a general way of implementing a class of MapReduce programs. We selected InvertedIndex from the tokenizer algorithms (Class 1), the Clustering algorithm using Reuters benchmark[36] from the machine learning class (Class 2), and page ranking Wikipedia articles from the graph algorithms (Class 3). Each of the selected examples are explained in the following subsections.

4.4.1.1 Inverted Index: A Class 1 Example

An InvertedIndex program for text files creates a data structure that maps words in the file to their locations [Zobel and Moffat, 2006]. Another implementation of the InvertedIndex algorithm involves a data structure, which has a field to store the document identifier and a counter for each word in this data structure that is emitted from the mapper. The reducer collects the data structures for each word and combines the data to give the final inverted index of that word. The input is a text file, a data structure that is required to implement the MapReduce block, and a converter that is required to read the data structures to the required output structure (see Figure 4.2).

4.4.1.2 PageRank: A Class 2 Example

PageRank[37] is used by the Google search engine to sort search results [Page, Brin, Motwani, and Winograd, 1998]. The algorithm assigns a weight to the pages based on the incoming and outgoing links in the documents. Pages are mapped to nodes while links are mapped to edges to create a graph structure. The algorithm works on the graph and weights are calculated for graph edges. The implementation[38] we used for our analysis was from Cloud9[39], a MapReduce library implemented using Hadoop for both teaching and data intensive research projects.

The program we used for our analysis creates a PageRank for all of the articles in the current

[36] Reuters benchmark, `http://www.daviddlewis.com/resources/testcollections/reuters21578/`
[37] PageRank algorithm, `http://www.google.com/corporate/tech.html`
[38] PageRank implementation, `http://www.umiacs.umd.edu/ÌČjimmylin/Cloud9/docs/content/pagerank.html`
[39] Cloud9, `http://www.umiacs.umd.edu/ÌČjimmylin/cloud9/docs/`

version of Wikipedia. Wikipedia allows downloading[40] the article contents to a zip file, which can later be extracted to an XML file (e.g., 'date-pages-articles.xml'). As shown in Figure 4.2, the XML file is converted to a sequential file of type PageRankNode using the class RepackWikipedia. The MapReduce framework processes the sequential files and emits the output sequential files of type PageRankNode. Using a converter, the output sequential files are read as Java objects of type PageRankNode to get the data into the necessary final output structure.

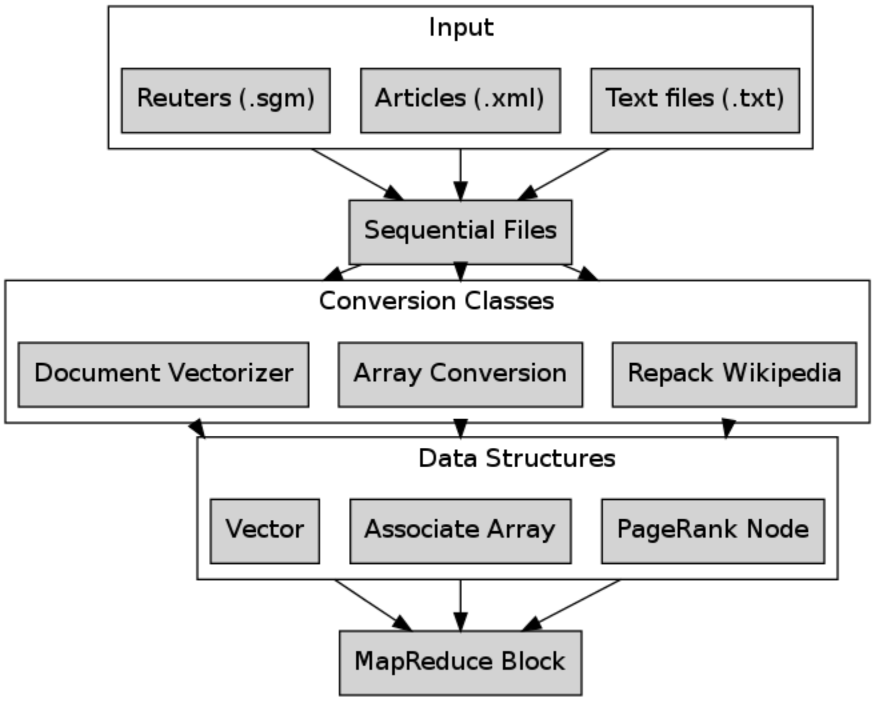


Figure 4.2: Input process overview

4.4.1.3 Clustering: A Class 3 Example

Clustering algorithms assign data into smaller groups based on a similarity factor. We used one of the simplest clustering algorithms, K-means. More details about K-means are presented in Section 4.6.2. We used the same implementation as Mahout. The K-means program is also commonly implemented using Hadoop. The program takes vectors as input and outputs a data

[40] Wikipedia dumps, `http://dumps.wikimedia.org/enwiki/`

structure of type Cluster. Using the data structure, the initial vectors can be clustered into specified groups, and the current center of each cluster can be determined.

The input used for the algorithm testing was Reuters-21578, one of the most widely used text collections in text categorization research. The text files are converted into sequential files of type String and are passed to a DocumentVectorizer, which parses the string to a Vector. The input vectors are then given to the MapReduce framework for clustering and output is generated as sequential files of type Cluster. Using a converter, `ClusterDumper`, the cluster sequential files are converted to text. The different stages of conversion involved in the K-means clustering of Reuters-21578 are shown in Figure 4.2.

4.4.2 Data Structure Analysis of MapReduce Programs

As mentioned earlier, writing a MapReduce solution includes specifying mappers and reducers. In some cases, adding a combiner and partitioner can make the solution more efficient. To define any of these, however, programmers are expected to specify the key type and value type of both the input and output. Hadoop has a small set of predefined key types and value types, which we found to be insufficient for providing solutions for real-world problems. For our analysis, all of the additional key or value types, which must be defined, are referred to as `Writable`. This is the interface Hadoop utilizes for defining new types. Some of the defined types might require a converter to read to and from Java Objects or text files. If they appear in the input of the mapper or output of the reducer, such types are referred to as 'Sequential types' (the types are generally written or read from sequential files). As an observation, if any of the input types of the mapper is a `Writable` object, the input structure for the MapReduce job will be a sequential file structure. The same is true for the output of the reducer.

Table 4.1 lists the various programs we used for our analysis and indicates whether the program has a mapper (M), reducer (R), combiner (C), and/or partitioner (P). The list of writable and sequential writable types for each program are also specified. All of the types shown as sequential writable are also writable; hence, in the table, it is shown as just sequential writable. The sequential writable types can be input (I), output (O), or both (I, O). The programs were collected primarily from Hadoop, Cloud9, and Mahout examples. If a program was collected from another source, the reference is provided within the table itself.

Class Type	Name	M	R	C	P	Writable types	Sequential Writable types
Class 1	WordCount	√	√	√			
	Bigram	√	√	√			
	InvertedIndex	√	√	√		AssociatedArray	
	Collocation matrix	√	√	√		Map	
Class 2	Collocation discovery	√	√	√		Gram	
	LDA model	√	√	√			Vector (I)
	Kmeans	√	√	√		ClusterObservation	Vector (I), Cluster (I,O)
	Dirichlet clustering	√	√			Vector (I)	Cluster (O)
	FuzzyKmeans	√	√	√		ClusterObservation	Vector (I), Cluster (I,O)
Class 3	HITS	√	√		√		HITSNode (O)
	PageRank	√	√	√			PageRankNode (I, O)
	BFS	√	√	√			BFSNode (I, O)
	Matrix multiplication	√	√	√			Vector (O)
	MonteCarlo	√	√	√			GridJobResult (O)
	Image processing	√	√				Image (O,I)

Table 4.1: Data structure analysis of MapReduce programs

4.4.3 Summary of the Analysis

The results of our analysis can be summarized in the context of three key concepts from software engineering based on criteria for well-designed software, such as comprehensibility and reusability [Parnas and Clements, 1986].

4.4.3.1 Code Comprehensibility

Code comprehensibility plays a vital role both in software development and software maintenance [Deimel and Lionel, 1985; Kernighan and Plauger, 1982]. To understand the execution of a given MapReduce program implemented in Hadoop, which is often spread throughout many Java classes, the programmer must determine the Driver class. If proper naming conventions are not followed, the programmer has to find out the extended classes (`AbstractJob`) and implemented interfaces (`Writable`). This is the case for a programmer who is familiar with Hadoop APIs. Even a Java programmer familiar with MapReduce concepts has to read Hadoop documentation to get started with MapReduce programming.

The Driver class, which is the configuration file for Hadoop, is not sufficient in showing relevant information for the code reader in MapReduce programs. As mentioned in the motivation section, there exists a contract between Mapper, Reducer, and Combiner; the output of the mapper should have the same type as the input of reducer, or vice versa.

4.4.3.2 Software Reusability

With the existing framework in Hadoop, MapReduce programs are written for a given input using a specified structure. This situation also occurs for libraries that are built over Hadoop. As an example, a BFS program from Cloud9 takes sequential files of type BFSNode as input. This type can be created from text files, which specify the graph in the following structure: "NodeId AdjacentNodeId1, AdjacenteNodeId2." If a programmer desired to execute the BFS program where the input is specified in a slightly different structure (e.g., "NodeId: AdjacentNodeId, AdjacentNodeId"), the programmer cannot directly use the BFS program from Cloud9.

4.4.3.3 Code Generation

Code generation brings the benefit of automation to software development [Czarnecki and Eisenecker, 2000]. Many programs require custom data structures (writable and sequential writable) to do the necessary computations. Implementing the `Writable` interface can be automated, as well as the corresponding converters, due to the type information about the output of the

`Mapper` being redeclared in the input of the `Reducer`, and the input of `Mapper` being redeclared as parameters in the map method of the Mapper class.

Our analysis concludes that these issues occur due to an improper level of abstraction. Currently, MapReduce is implemented as an API, and these issues can be addressed if we can raise the level of abstraction. Type checking should be done at this new level of abstraction to allow the code to be more precise and readable. In the background, generative programming techniques can be used to run Hadoop while the programmer is presented with a small language targeted solely for MapReduce. Based on this analysis, a tool is presented that illustrates the potential for increasing the extensibility of input/output types with code generation. Our case studies in Section 4.6 show how MapRedoop can improve code comprehensibility and software reusability.

4.5 MapRedoop

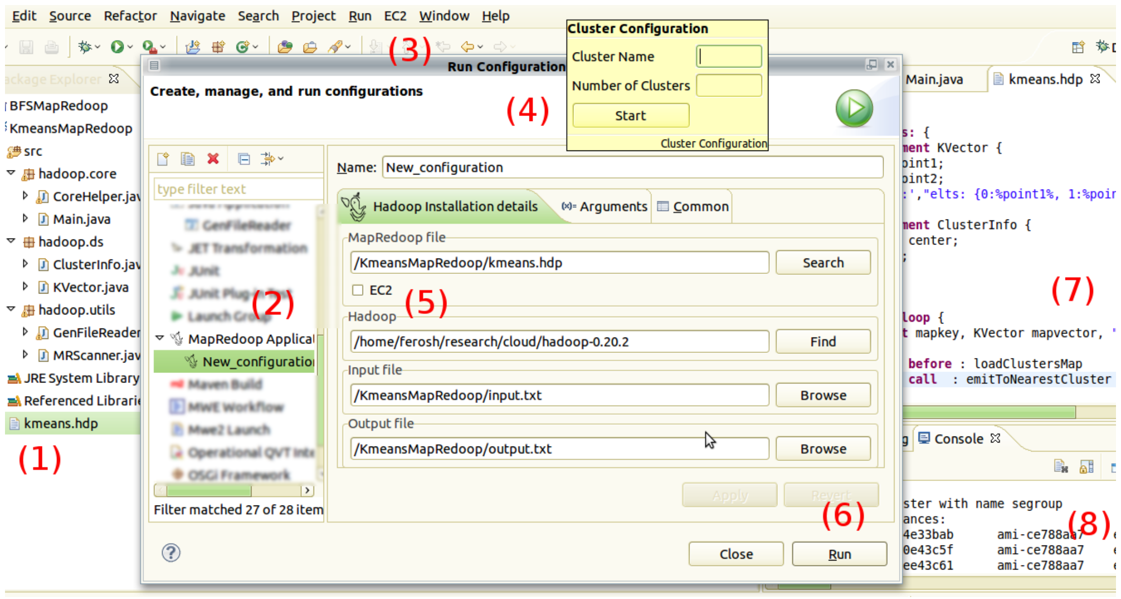
In this section, the MapRedoop framework is explained. MapRedoop has been used interchangeably to denote both the framework and the DSL. Section 4.5.1 explains the implementation of a MapReduce program using MapRedoop from a user's perspective and Section 4.5.2 explains the implementation of the MapRedoop framework. A demo using MapRedoop programs can be found on our project site[41].

4.5.1 Using MapRedoop in the Eclipse IDE

In this section, MapRedoop is explained from a user's perspective. The programmer writes the MapRedoop (DSL) for his/her current problem in a specialized editor (marked '7' in Figure 4.3), which supports syntax highlighting, code completion, validation and quick fixes, and advanced editor features such as bracket matching and outline view.

Development of MapReduce programs in MapRedoop occurs in four stages:

[41] MapRedoop, `https://sites.google.com/site/mapredoop/`



Name: New_configuration

MapRedoop file: /KmeansMapRedoop/kmeans.hdp

☐ EC2

Hadoop: /home/ferosh/research/cloud/hadoop-0.20.2

Input file: /KmeansMapRedoop/input.txt

Output file: /KmeansMapRedoop/output.txt

```
s: {
ment KVector {
oint1;
oint2;
:',"elts: {0:%point1%, 1:%poir

ment ClusterInfo {
 center;
;

loop {
t mapkey, KVector mapvector, '
 before : loadClustersMap
 call   : emitToNearestCluster
```

```
ster with name segroup
ances:
4e33bab       ami-ce788aa7
0e43c5f       ami-ce788aa7
ee43c61       ami-ce788aa7
```

Figure 4.3: Screenshot of MapRedoop in Eclipse IDE

1. **Creating MapRedoop programs:** The programmer completes the MapRedoop program, which is represented by a file having extension ".hdp" (e.g., "kmeans.hdp" marked '1' in Figure 4.3). Within that program, the programmer specifies the required data structures and plugin extension points in the framework.

2. **Code generation:** The programmer generates code by right-clicking the ".hdp" file and using the "Generate code" option. This creates three Java packages: 1) hadoop.core, the main package for executing, setting up, and running the mapper and reducer, 2) hadoop.ds, the package for data structures in the program, and 3) hadoop.utils, the helper classes for converting text files to sequential files, and vice versa.

3. **Implementing MapReduce methods:** The programmer implements the actual MapReduce algorithm. After the code generation, there are empty stub methods inside the "CoreHelper.java" classes. These empty methods give the full flexibility of the Java programming language for the programmer to implement the MapReduce logic.

4. **Execution:** The programs written using MapRedoop can be executed in two modes:

   - *Hadoop standalone version:* Upon right-clicking the ".hdp" file and selecting the "Run as MapRedoop" option, the programmer is presented with a "Run Configuration" dialog as shown in Figure 4.3. If the option "EC2" (marked '5' in Figure 4.3) is unchecked, upon selecting "Run" (marked '6' in Figure 4.3), the program is executed as standalone. Because the run configuration (marked '2' in Figure 4.3) is implemented using the Eclipse run configuration framework[42], the same programs can be executed with different inputs or configurations, and these configurations can be saved.
   - *Hadoop cluster in EC2:* Before executing a program in EC2, a Hadoop cluster should be launched. This assumes that the required EC2 configurations have already been made in the Hadoop installation folder to start a Hadoop cluster in EC2. Upon selecting "EC2" (marked '3' in Figure 4.3), the programmer is presented with a light-weight pop-up (marked '4' in Figure 4.3) to specify the cluster name and number of slaves.

   All communication between the Hadoop server, whether it is standalone or EC2, are shown in the console (marked '8' in Figure 4.3). Video demonstrations of standalone executions and EC2 clusters can be viewed at our project web site.

4.5.2 High-Level Design Diagram

A high-level design diagram of the MapRedoop framework is shown in Figure 4.4.

The configuration and control of this framework is achieved through the MapRedoop DSL explained in Section 4.5.3 As shown in Figure 4.4, the MapRedoop tool has two components:

[42] Eclipse run configuration, `http://www.eclipse.org/articles/Article-Launch-Framework/launch.html`

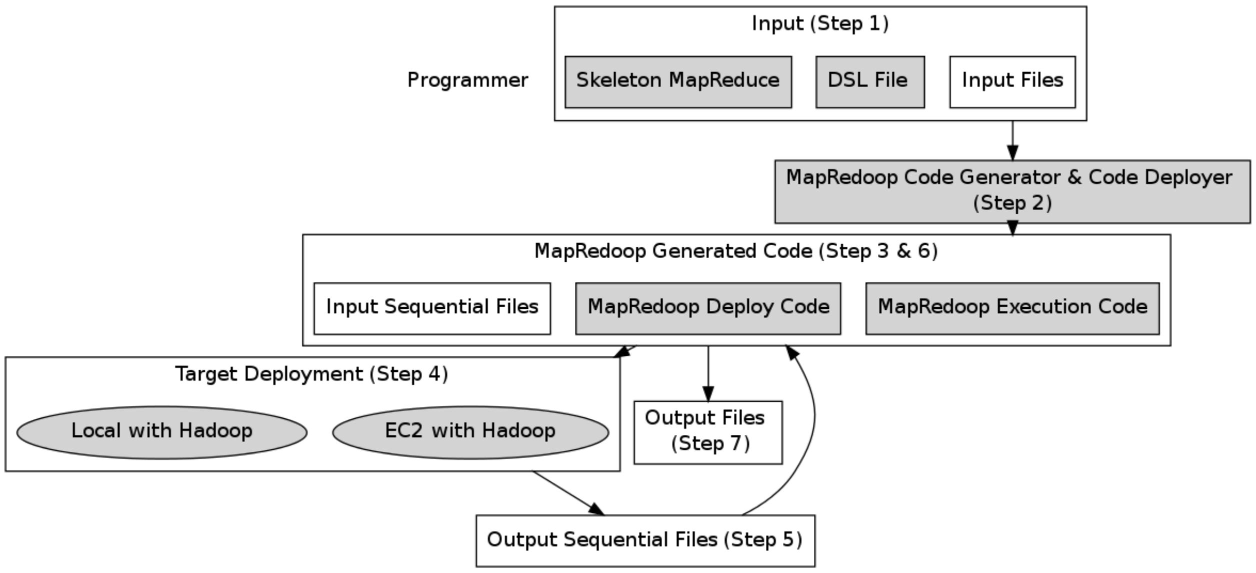


Figure 4.4: An overview of MapRedoop design

1. **Code generator:** This component takes the MapRedoop DSL as input and generates code for three packages, as mentioned previously. These generated classes can be combined into three categories:

   (a) *Core classes:* These include the Mapper, Reducer, Combiner, Partitioner, and the Driver. Other than the Driver class, these classes are only generated if they are mentioned in the MapRedoop DSL. In addition to the `Driver` class, a `CoreHelper` class is generated; this is where the programmer will implement the actual MapReduce algorithm.

   (b) *Data structure classes:* Data structure classes are new types defined in the MapRedoop DSL. There can be two types of data structure classes: 1) Data structure types that occur as the key/value of any of the mappers or reducers, and 2) Data structure types that are only used inside the program. In Hadoop, data structure types that occur as the key/value pairs require Hadoop to implement an interface called `Writable`.

   (c) *File conversion classes:* Programmers can specify the template in which input files should be read. The MapRedoop framework generates classes such that the data can be read while converting the text file to a sequential file and also while converting the sequential file back to a text file. The classes are executed before and after the execution of the MapReduce programs to make the automatic conversion of data possible.

2. **Code deployer:** The deployment is done in four stages:

   (a) *Target environment:* Based on the user's choice, the tool deploys the code either in Hadoop standalone version or in the EC2 Hadoop cluster.

   (b) *Format conversion I:* Text files or the input information has to be converted to sequential files, and in the case of the user selecting the EC2 cluster, the sequential files must be uploaded to the server.

   (c) *Communication:* Collect the results from the server in case of a cluster deployment.

   (d) *Format conversion II:* Convert the results back to the original input structure.

   The key/value types of the program determine the steps. If the key-value types are not composite, as in the case of the simple 'WordCount' program, the deployer just uploads the input files to the server, executes the results and collects the output to the requested folder. Class 1 programs generally skip the conversion steps because the input for the programs are text files.

4.5.3 MapRedoop DSL

A subset of the grammar for the MapRedoop DSL is shown in Figure 4.5 (only the important parts of the grammar are shown for clarity).

```
1  grammar mapredoop;
2
3  MapRedoop: Declaration { (Content)*}
4
5  Declaration: 'program' ID ('extend' ID)?
6
7  Content: ListofEntities | MRBlock
8
9  ListofEntities: 'metaelements': { (Entity)* }
10
11 Entity: 'metaelement' ID ('extend' [Entity])? { (Feature)+ }
12
13 Feature: TypeDeclaration; | ReadWrite;
14
15 ReadWrite: 'read' (STRING, STRING) | 'write' (STRING, STRING)
16
17 MRBlock: 'mapreduce': ('loop')? { Mapper Reducer }
18
19 Mapper : 'map''('Argument, Argument, STRING, STRING ')' Block
20
21 Reducer : 'reduce''('STRING,STRING, STRING , STRING')' Block;
22
23 Block: [(JavaMethodCall)* ]
24
25 JavaMethodCall: TimeOfCall : ID
26
27 TimeOfCall: 'after'| 'call' | 'before'
```

Figure 4.5: MapRedoop DSL grammar

The MapRedoop program starts with a declaration statement (line 2). An optional 'extend' is included to re-use some of the features already declared within a DSL. All of the data structures declared in the parent DSL will be available inside the extending DSL (child DSL). The DSL has two sections: 1) Meta-elements (line 9), and 2) MapReduce (line 17).

1. **Meta-elements:** This block is for generating Java code for the data structures to be used later by file structure conversion classes or the MapReduce framework itself. For every `metaelement` defined in the `metaelements` section, the code generator checks whether that data structure appears as a type in the keys or values of the mapper or reducer. If it does not appear, it is generated as a regular Java class with the fields declared in the `metaelement`, adding mutator and accessor methods. If the types appear in any of the key/value pairs of mapper or reduce, the class is generated as a `Writable` Java class.

2. **Writable classes:** In Hadoop, the classes that are used as a key or value for the mappers or reducers must implement the `Writable` interface. In order to implement this interface, the programmer has to implement two methods, `read` and `write`. An example implementation[43] of these methods is shown in Figure 4.6. The code block is taken from a class having three fields (`x, y`, and `z`), each of type float. As demonstrated in Figure 4.6, it is clear that if we know the field types, the code can be generated. Aggregate relationships, such as a Java List, are implemented using an array of elements and each element's type. In this situation, our implementation adds an additional variable called `fieldname+size` of type `int` to the original list of fields.

   Each meta-element can have two types of features:

   (a) A `TypeDeclaration` (`metaelement` or native type, such as int)

   (b) `ReadWrite`. We first define the fields in a data structure and later link them to the input structure in a text file. A line that is coded as:

   ```
   read (" ", %nodeId%{%distanceFromSource%} :
                       %adjacentlist%");
   ```

   This line specify a meta-element having three fields: `nodeId`, `distanceFromSource`, and `adjacentlist`. It also declares its input structure for reading this object originating from a text file. Given this information, during runtime, the parsers can create Java objects with the values read from a text file. As an example, if a file has the following two lines: "2{3} : 3" and "3{2} : 4 5 6", two Java objects are created:

[43] Writable implementation, `http://developer.yahoo.com/hadoop/tutorial/module5.html`

    (a) The first object created is `nodeId` 2 consisting of `distanceFromSource` 3 and a list `adjacentlist` having the value of 3.

    (b) The second object created is `nodeId` 3 consisting of `distanceFromSource` 2 and a list `adjacentlist` having 4 5 6.

   The second parameter of the read method defines the structure and the first parameter is an optional way of specifying a delimiter while parsing a list of values. For objects of type `adjacentlist`, a space is used as the delimiter.

3. **MapReduce blocks:** The MapReduce blocks were designed to avoid the repeated declaration of types required in the Hadoop implementation. Hence, there is no input declaration for `reducer` and input/output declaration for combiner, because these would be the same as the output of the `mapper`. Therefore, `reducer` has no input declaration, only an output declaration. The `map` function takes two parameters (type followed by variable name) and two arguments (both representing a type). The first two parameters declare two variables, and those variables can be used inside the `mapper` function. The last two parameters are arguments declaring the output type of the `mapper`. Those types are only used inside the `reducer`; hence, those variable names are defined as the first two arguments of the `reducer`. The last two parameters of the `reducer` define the output types of the MapReduce program.

In addition to the above, the MapReduce block allows plug-in Java calls to implement the actual MapReduce program. For `reducer` and `mapper`, these Java calls can be made either during the process function, or before or after the core function call. Plug-in method calls are

```

      public void write(DataOutput out) throws IOException {

        out.writeFloat(x);
        out.writeFloat(y);
        out.writeFloat(z);
      }

      public void readFields(DataInput in) throws IOException {

        x=in.readFloat();
        y=in.readFloat();
        z=in.readFloat();
      }
```

Figure 4.6: Sample `read` and `write` implementation of `Writable`

Algorithm	K-Means			BFS		
Tool	MapRedoop	Mahout	% Reduction	MapRedoop	Cloud9	%Reduction
Lines of Code	$99+23$	493	75%	$94+20$	331	66%

Table 4.2: Lines of code comparison of Hadoop libraries with MapRedoop

generated to support iterations of MapReduce calls by setting the flag loop. Examples of two programs written using MapRedoop are introduced in Section 4.6.

4.6 Two Case Studies for MapRedoop: BFS and K-means

In this section we describe two algorithms implemented using MapRedoop and compare the solutions to other Hadoop libraries. Implementing the K-means clustering algorithm while implementing the Breadth First Search (BFS) algorithm requires Class 3 types (please see Table 4.1). Code-level comparison of the programs is shown in Table 4.2. Lines of code for MapRedoop represent the actual implementation added to the MapRedoop DSL code. The lines of code for the libraries include the code specifically written for implementing the program. The pre-defined data structures (e.g., `Vector` in Mahout) are not considered when counting the total lines of code. As described in the table, MapRedoop requires 75% less lines of code to implement a BFS algorithm as compared to Cloud9, and 66% less lines of code to implement a K-means algorithm as compared to Mahout. A detailed performance comparison of the two case studies along with their MapRedoop solution is given in the following sub-sections. For the performance analysis, we executed the two versions (MapRedoop and Hadoop Library) of the program in a standalone Hadoop installation and also in an EC2 Hadoop cluster. The clusters were implemented with one, two, four, and eight slaves for a given size of data. Every execution in the cluster, as well as the standalone versions, was executed three times and the reading taken was the mean of the three executions.

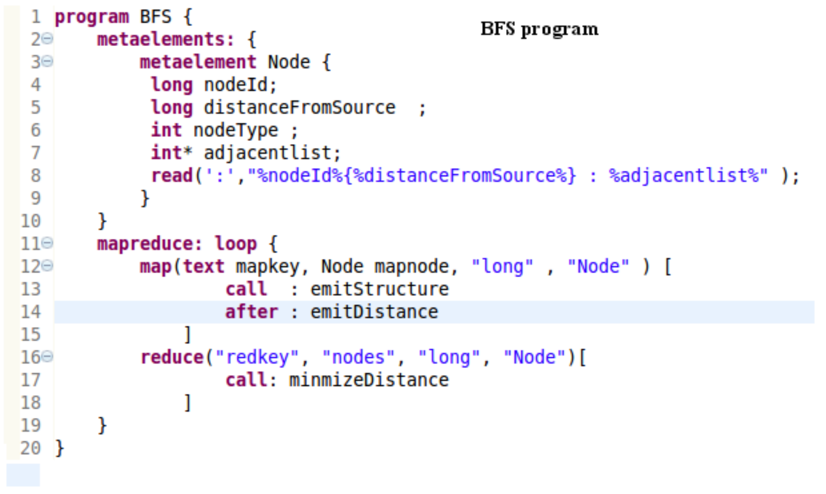
4.6.1 Implementing the Breadth First Search Algorithm in MapRedoop

Breadth First Search (BFS) is a common algorithm to find the distance from the source node to all the reachable nodes. The algorithm begins by finding all of the neighbors for the first node, and for each of the first node's neighbors, the algorithm finds those neighbors. This cycle continues until the algorithm reaches the goal node. Applications of BFS include finding the shortest path and spanning forests.

Implementation of BFS in MapReduce involves finding the distance from the current node to the source node. The mapper is responsible for storing the computed distance to the next node. This node is then passed on to the reducer and emitted. The reducer collects all of the nodes from the mapper, and for each node, the reducer selects the node storing the smallest distance. For every node, the least distant node is selected. Each MapReduce iteration is a hop in the graph. The mapper should emit the structure along with the distance so that the adjacent nodes can be calculated for the next iteration. The MapRedoop DSL for describing these properties is shown in Figure 4.7.

4.6.1.1 MapRedoop DSL

The MapRedoop solution for the BFS algorithm is shown in Figure 4.7. In the meta-elements block, a data structure or meta-element is declared called `Node` with fields: `nodeId`,



```
program BFS {
    metaelements: {
        metaelement Node {
         long nodeId;
         long distanceFromSource  ;
         int nodeType ;
         int* adjacentlist;
         read(':',"%nodeId%{%distanceFromSource%} : %adjacentlist%" );
        }
    }
    mapreduce: loop {
        map(text mapkey, Node mapnode, "long" , "Node" ) [
                call  : emitStructure
                after : emitDistance
            ]
        reduce("redkey", "nodes", "long", "Node")[
                call: minmizeDistance
            ]
    }
}
```

Figure 4.7: MapRedoop DSL for BFS in Eclipse IDE

`distanceFromSource`, `nodeType`, and `adjacentList`. A Java class will be generated from this structure with the name and corresponding fields of each type, as mentioned in the DSL. While reading the structure “2{3}: 3, 4” from a file, the read method creates a `Node` object with `nodeId` of 2, `distanceFromSource` of 3, and `adjacent nodes` 3 and 4. In the MapReduce block, the `map` function accepts `mapkey` of type `Text` as the input key, and the input value `mapnode` of type `Node`. The `mapper` function sets up an output key of type `long`, and creates an output value of type `Node`.

As shown in the BFS example in Figure 4.7, only the type is defined in the `map` function. The name of the variables are defined in the `reduce` function, because reducer uses the variables while the map function defines the type of the variables. The `reduce` function also defines the type of the output key and value.

On code generation, MapRedoop creates `Mapper` and `Reducer` classes, along with the `Driver` class containing all the necessary key value declarations. A class called `CoreHelper` is created with methods `emitStructure`, `emitDistance`, and `minimizeDistance`. The

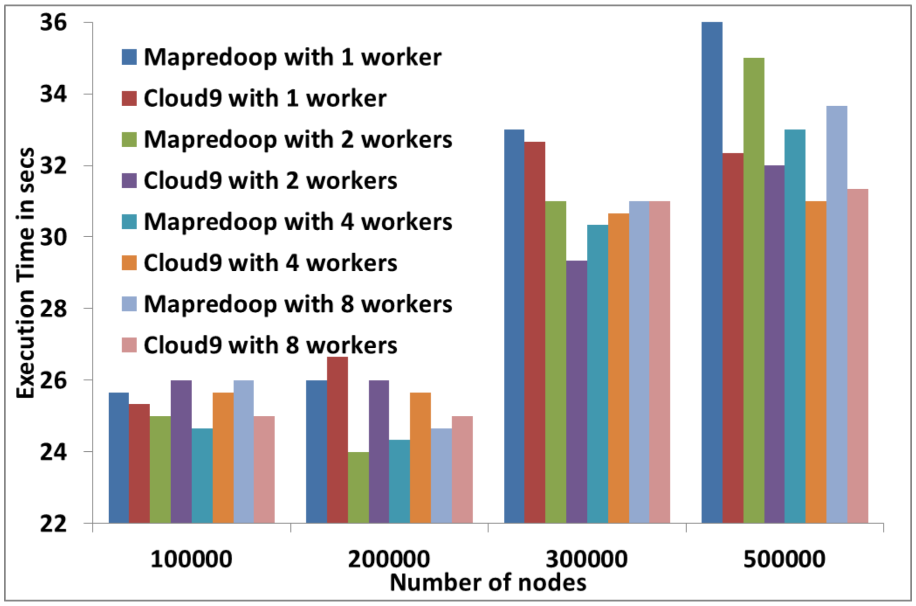


Figure 4.8: Execution time of BFS programs in EC2 Hadoop cluster

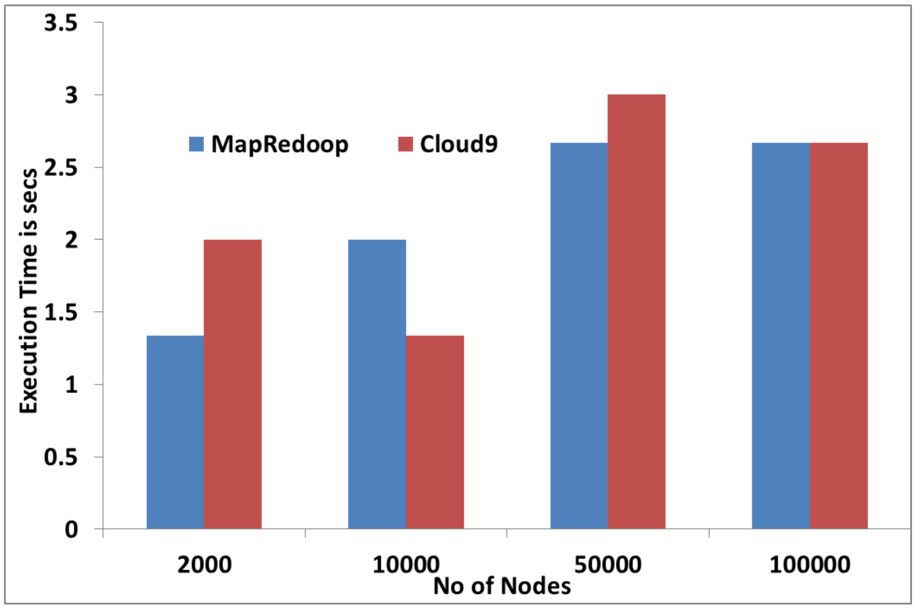


Figure 4.9: Execution time of BFS programs in Hadoop standalone mode

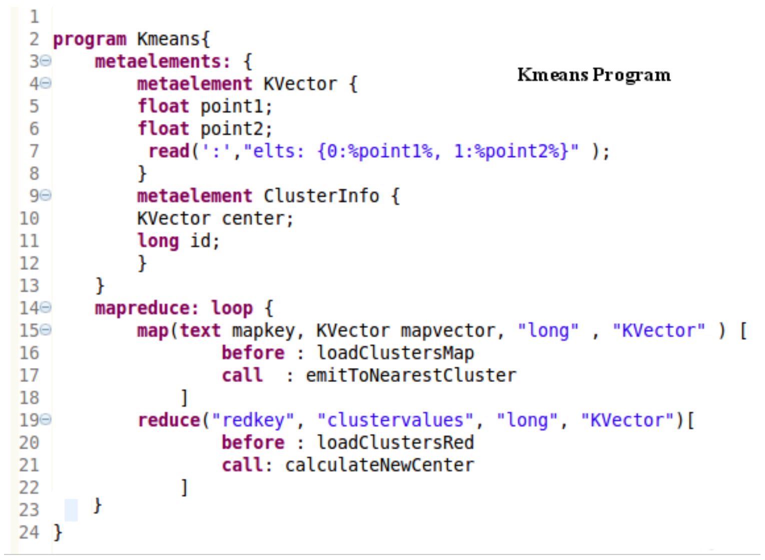
code is configured such that the map method in the `Mapper` class calls the `emitStructure` method, while the `cleanup` method in the `Mapper` class calls the `emitDistance`. Finally, the `reduce` method in the `Reducer` class calls the `minimizeDistance` method.

4.6.1.2 Performance Analysis

The data for the standalone version was classified into four categories: 1) Micro: a graph having 2,000 nodes and a variable number of edges (0-5), 2) Small: graph having 10,000 nodes, each having 5 edges, 3) Medium: a graph having 50,000 nodes, each having 5 edges, 4) Large: a graph having 100,000 nodes, each having 5 edges. For the cluster micro, small, large, medium, large, each had 100,000, 200,000, 300,000, and 500,000 nodes respectively. For performance comparison, we selected a program written by expert Hadoop programmers from Cloud9, which was implemented using the same algorithm. From the figures shown in Figure 4.8 and Figure 4.9, both the MapRedoop and the expert-created program gave comparable performance in both the standalone and cluster implementations. As the graphs illustrate, there are some scenarios where MapRedoop performed more poorly than Cloud9. The bars represent the average of the trials ran, and the error bars represent the max and min values of the trials. In some cases there is a large delta between the max and min illustrating the inconsistency of the Hadoop File System, which is part of the reason for the vast difference in performance from scenario to scenario. Additionally, due to the added flexibility for input/output types provided by MapRedoop, there is a degradation in runtime.

4.6.2 Implementing the K-means Algorithm in MapRedoop

K-means is possibly one of the most commonly used clustering algorithms according to [Kanungo et al., 2002]. The K-means clustering algorithm groups a cluster into 'k' small clusters based on a similarity factor. The similarity factor we used in this implementation is the distance. The algorithm starts with randomly selected 'k' vectors (in our case, user specified vectors). For every input vector, the algorithm calculates the distance from the initial 'k' vectors to the current



```

program Kmeans{
    metaelements: {
        metaelement KVector {
        float point1;
        float point2;
         read(':',"elts: {0:%point1%, 1:%point2%}" );
        }
        metaelement ClusterInfo {
        KVector center;
        long id;
        }
    }
    mapreduce: loop {
        map(text mapkey, KVector mapvector, "long" , "KVector" ) [
                before : loadClustersMap
                call   : emitToNearestCluster
            ]
        reduce("redkey", "clustervalues", "long", "KVector")[
                before : loadClustersRed
                call: calculateNewCenter
            ]
    }
}
```

Figure 4.10: MapRedoop DSL for K-means in Eclipse IDE

vector, and the closest ‘k’ vector is grouped with the current vector. In the MapReduce implementation of K-means, every vector in the mapper part is emitted to the nearest cluster and the reducer part collects the vectors to a given cluster.

4.6.2.1 MapRedoop DSL

The DSL implementation of K-means using MapRedoop is shown in Figure 4.10. The K-means program makes use of meta-element feature of MapRedoop. The `ClusterInfo` meta-element is a special type that does not occur in any of the key/value types of the mapper or reducer. Hence, `ClusterInfo` is generated as an ordinary Java class (not an implementation of `Writable`) and `KVector` is generated as a `Writable` Java class. In this case, the `before` keyword is used both in the `map` and `reduce` blocks. Hence, two additional methods are created in the `CoreHelper` class, which is called from `setup` functions of `Mapper` and `Reducer` classes. In the implementation of K-means, there is an additional input other than the input vectors that represent the current clusters. Before invoking the map and reduce operations, the data from the current clusters has to be loaded using `loadClustersMap`.

4.6.2.2 Performance Analysis

The K-means implementation from the Mahout project was used for a performance comparison. Both programs produce the same output. As input, N-points were used to group the output into three clusters based on their Manhattan distance, which is the distance measured along axes at right angles. The execution plots of these algorithms in EC2 and standalone are presented in Figures 4.12 and 4.11. Four different values were used for N : 1) 100,000, 2) 200,000, 3) 300,000, and 4) 500,000. The MapRedoop version dominated in the standalone version and had comparable results in the cluster implementation. This can be attributed to the Vector data structure in Mahout. Because the K-means solution in Mahout is a case of a generic clustering solution, there are many fields in the data structure that are not relevant to the K-means problem. This can result in more writing and reading during file operations. In the case of MapRedoop, solutions are written for a problem, and hence the programmer needs to define only the fields relevant to the problem. As mentioned in the previous analysis, the inconsistency in the Hadoop File System and the added input/output flexibility contribute to MapRedoop's degraded performance.

4.7 Discussion

After writing several MapReduce programs in Hadoop, we recognized three specific areas of inefficiency resulting from accidental complexities: input structure inflexibility, level of abstraction, and ease of testing. Our solution, MapRedoop, is a framework implemented in Hadoop that combines a DSL and IDE that removes the encountered accidental complexities. To evaluate the performance of our tool, we implemented two commonly described algorithms (BFS and K-means) and compared the execution of MapRedoop to existing methods (Cloud9 and Mahout). With MapRedoop, the programmer only needs to code the DSL and MapReduce algorithms,

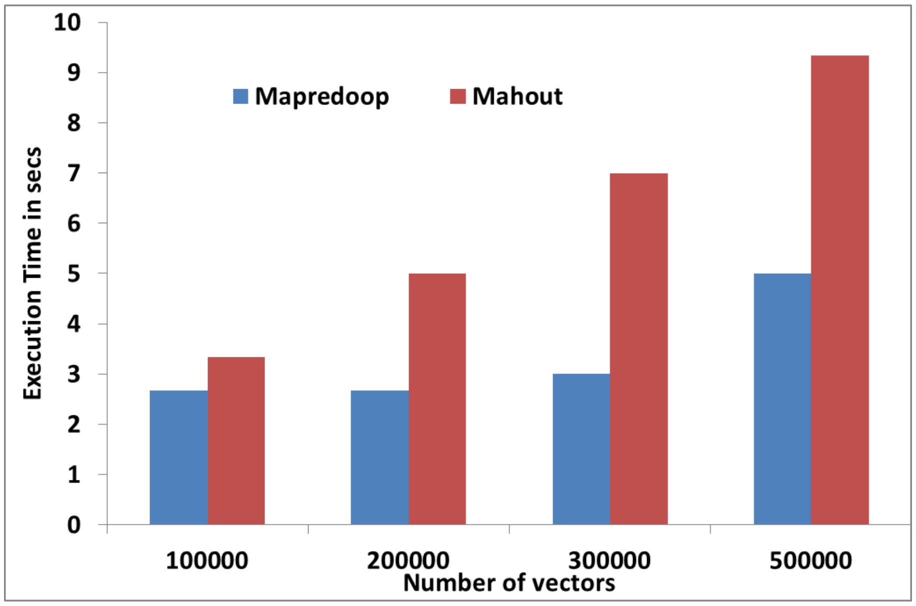


Figure 4.11: Execution time of K-means programs in Hadoop standalone mode

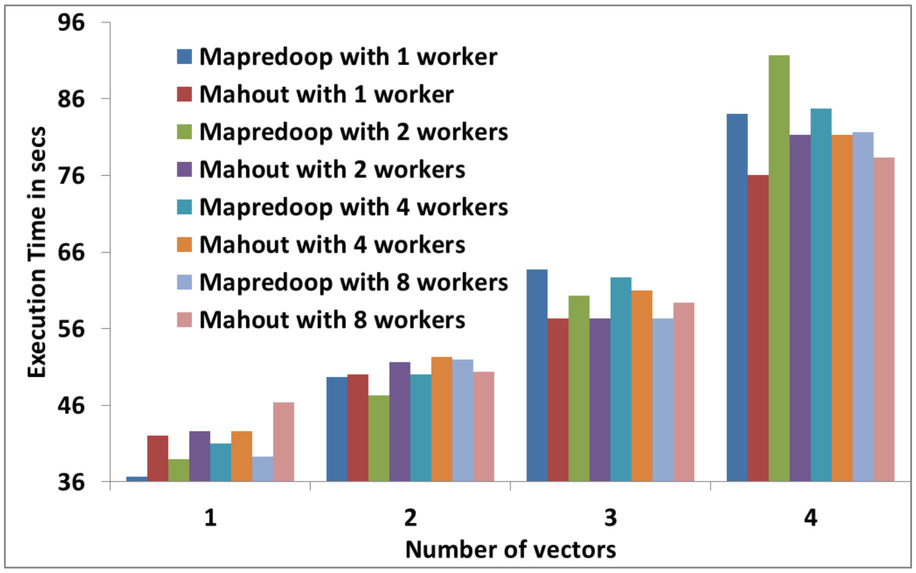


Figure 4.12: Execution time of K-means programs in EC2 Hadoop cluster

whereas Cloud9 and Mahout focused on input/output conversions. The analysis presented in Section 4.6 illustrates that MapRedoop performs comparably to the existing, common methodologies, and in some cases, MapRedoop proved to have better performance due to the programmer being able to focus solely on the specifics of the problem.

Chapter 5

SDL & WDL: A PROGRAM-LEVEL ABSTRACTION FOR CREATING SIGNATURE DISCOVERY WORKFLOWS

Domain-agnostic Signature Discovery entails scientific investigation across multiple domains through the re-use of existing programs into workflows. The existing programs may be written in any programming language for various hardware architectures (e.g., desktops, commodity clusters, and specialized parallel hardware platforms). This raises an

ering issue in generating Web services for heterogeneous programs so that they can be composed into a scientific workflow environment (e.g., Taverna).

In this chapter, we show how program-level modeling can help scientists in generating Web services from programs. This chapter starts with a brief introduction about the SDI project in Section 5.1. Section 5.2 describes two specific engineering issues in the development process and a current solution approach for the two driving issues using an example scenario focused on BLAST[44] execution workflow. A brief discussion of related work is summarized in Section 5.3 and implementation details of the approach are provided in Section 5.4. In Sections 5.5 and 5.6, we introduce case studies for SDL and WDL. The chapter is concluded in Section 5.7.

5.1 Signature Discovery Initiative (SDI) Project

A signature is a unique or distinguishing measurement, pattern, or collection of data that detects, characterizes, or predicts a target phenomenon (object, action, or behavior) of interest .

[44] BLAST (Basic Local Alignment Search Tool), `http://blast.ncbi.nlm.nih.gov/Blast.cgi`

Signatures are valuable to a wide range of application domains (e.g., medicine, network security, and explosives detection) for anticipating future events, diagnosing current conditions, and analyzing past events. However, current approaches suffer from a lack of re-use of existing algorithms, tools, and techniques across application domains and scientific disciplines. The Pacific Northwest National Laboratory (PNNL) has been developing a generalized signature development methodology (SDI) that is applicable to any signature discovery problem [Baker, 2012]. The work of signature discovery entails scientific investigation across multiple disciplines through the re-use of existing algorithms, which may be written in any programming language for various hardware architectures (e.g., desktops, commodity clusters, and specialized parallel hardware platforms). Reusing these algorithms requires a common architecture through which analytical software components created by scientists from different disciplines can be integrated.

Software developers and researchers at PNNL have been working closely with scientists who have developed or applied algorithms using a wide range of programming languages and tools. Software development tasks involve: 1) developing the service-oriented software framework for scientists in specific domains to register and share their algorithms so that they can make those algorithms as reusable service components, and 2) creating new signature discovery workflows in other domains using the created service components. In this chapter, we introduce two DSLs: 1) SDL (Service Description Language), and 2) WDL (Workflow Description Language), through which scientists can achieve these two tasks without dealing with the associated engineering requirements in the Signature Discovery process.

5.2 Example Scenario: BLAST Execution Workflow

Scientists use BLAST to find regions of similarity between biological sequences. From a engineer's (software developer) context, BLAST is a long-running job with input and output. The workflow is usually executed in three steps: 1) Submitting a BLAST job in a cluster using the SLURM[45], job scheduler; 2) Checking the status of the job; and 3) Download the output files upon completion of the job. Note that a UNIX command utility (e.g., sh) and (or) a script file (e.g., SLURM) identifies each step. In the following sub-sections, three engineering challenges when sharing signature discovery workflows with other scientists are explained.

5.2.1 Accidental Complexity of Generating Service Wrappers

To make existing programs available globally, web service wrappers were added for every executable binary. The Legacy Wrapper pattern [Erl, 2009] was used to encapsulate existing algorithms, while providing a standard interface so that they can be orchestrated with other services to create re-usable workflows. Engineers creating wrappers for an existing script or executable typically follow a common set of steps: 1) identify the input files and output files in the program; 2) retrieve the input files from the data management system; 3) execute the program; and 4) upload the output files to the document management system as part of the existing signature discovery software framework. This process for converting a script often results in significant extra code and manual effort.

For BLAST, `checkJob` is a service for checking the job status, with a single input (`JobId`) and output (`status`) requiring a service wrapper with 121 lines of Java code (in four Java classes) and 35 lines of XML code (in two files). The goal of this project is to raise the level of abstraction from these general purpose languages to the signatures domain, such that scientists only have to specify the specifications for the executable binary (signature algorithms, in this case) that has to made public and the language will generate and execute the code required for the process.

[45] SLURM (Simple Linux Utility of Resource Management), `https://computing.llnl.gov/linux/slurm/`

5.2.2 Coupling between Workflows and Services

Even if the remote executable binaries are accessible globally, scientists still need to orchestrate such services to define the Signature Discovery workflow. To make any web service available in a domain-independent workflow engine like Taverna, users have to 1) manually add the operations (three steps in this case) to the workflow designer (called the *workbench* in Taverna) using the web service, and 2) provide XML parsers and generators. In a Web service, the input and output are expressed in XML. Hence, an XML generator is required before passing any workflow parameters to the service. Similarly, an XML parser is required to process the service output after executing the service. Hence, these two processes (i.e., service creation and service orchestration) are coupled together as revealed by our BLAST execution example scenario.

An executable BLAST execution workflow in the Taverna workbench is shown in Figure 5.1. Engineers created three services as workflow deployable service components. In the figure, all boxes (except the light blue, which represents workflow input and output) correspond to processors; processors performing similar functions are identified by the same color. As shown in Figure 5.1 (inside the sub-workflow `checkJob`), `jobID` is wrapped inside an XML descriptor by `jobStatusIn` before passing to the actual service processor `jobStatus`. After executing the service processor, the output XML is passed to the `jobStatusOut` processor, which parses the XML and passes the status to workflow output port status. In case the output of a service is an Object, as in the case of `SubmitBlast`, many XML parser processors (`submitBlastOut`, `submitBlastOut_submitBlastResults`) are required for this conversion. Hence, the type and the number of XML parsers and generators required to make a web service available in a Taverna workspace is dependent on the structure (type and number of the input and the output)

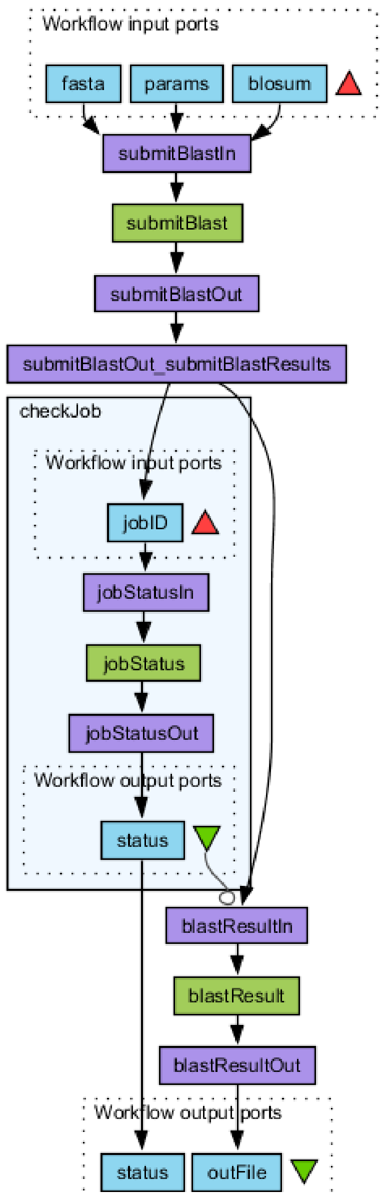


Figure 5.1: BLAST execution in Taverna

of the web service itself. For scientists to use these services in the Taverna workbench, they need to: 1) know the input and the output format of each service, and 2) provide correct parsers and generators for each of them.

5.2.3 Lack of End-User Environment Support

Many scientists are not familiar with service-oriented software technology, which forces them to seek the help of engineers to make web services available in a workflow workbench. On the other hand, engineers find it difficult to work with scientific scripts and tools, which is not usually their area of expertise. This technology barrier may degrade the efficiency of sharing signature discovery algorithms, because any changes or bug fixes of an algorithm require a dedicated engineer and a scientist to navigate through the engineering process.

5.2.3.1 Solution approach using DSLs

In this chapter, a new approach to simplify the integration of signature discovery algorithms into a common architecture is explained. In this approach, two sets of DSLs are defined that contain: 1) A service description language (SDL) that can be used by the end-user to specify the user credentials, executable path, script file, input and output of any script, and 2) A workflow description language (WDL) that specifies the interactions of these services and takes input from one or more service description files. The syntax of the workflow description language is mapped to the Taverna workflow APIs. Thus, the glue code to orchestrate the service components from the service description language is generated.

SDL and WDL files for the BLAST execution workflow are shown in Figure 5.2. In Figure 5.2.a, three services (`submitBlast`, `jobStatus` and `blastResult`) are defined along with their input, resources (script files) and connection details. In Figure 5.2.b, a main workflow is defined with input and output. The code used in these two files are explained later in the implementation section (Section 5.4). The few lines of code shown in Figure 5.2 is powerful enough to: 1) Create three web services, each service is a wrapper for a remote executable, that will upload all the required files to the remote server before execution and after execution the resulting files are downloaded, and 2) Create an executable workflow in a workflow engine like Taverna (Figure 5.1).

Using this approach, scientists can: 1) Design, develop, and deploy new service wrappers

```
//Remote machine configuration
define ssh_oly as "sdi" at "olympus.pnl.gov"
  with-key "/home/jaco181/.ssh/id_rsa"

/*
 *  Creating a service to submit a blastJob
 */
service submitBlast {
  use ssh_oly;
  cmd "sh runJob.sh";
  resource "jobScript.sh", "runJob.sh";
  in doc blossum, params, fasta;
  out jobID, outDir;
  /*
   *Inside run.sh
   *echo "jobID=\$JOBID" > .properties
   *echo "outDir=\$deployDir" >> .properties
   */
}
/*
 *  Creating a service to upload a remote
     ↪file
 */
service blastResult {
  use ssh_oly;
  cmd "cp $outDir$/test_all_v_all_m8.out
      ↪outFile";
  in outDir;
  out doc outFile;
}
/*
 *  Creating a service to check status of a
     ↪job
 */
service jobStatus {
  use ssh_oly;
  cmd "sh checkStatus.sh";
  resource "checkStatus.sh";
  in jobID;
  out status;
}
```

(a) SDL code for BLAST

```
use "SigQuality.sdl"
/*
 *   Main workflow
 */
workflow BlastSearch (in blosum, in params,
  in fasta, out outFile, out status){

  //Passing values to service
  blosum->submitBlast.blossum
  params -> submitBlast.params
  fasta -> submitBlast.fasta

  //Calling sub-workflow
  call checkJob
    till status="Done"
    with  submitBlast.jobID,  status

  //Configuring execution
  submitBlast.outDir ->blastResult.outDir
      ↪after checkJob

  //Passing to workflow output
  blastResult.outFile->outFile

}
/*
 *Sub-workflow checkJob
 */
workflow checkJob (in wf_jobID,  out
    ↪wf_status){
  wf_jobID->jobStatus.jobID
  jobStatus.status->wf_status

}
```

(b) WDL code for BLAST

Figure 5.2: BLAST execution using SDL and WDL

from remote executable scripts or commands and 2) Orchestrate and monitor the execution of such services in a powerful workflow engine like Taverna.

5.3 Related Works in Workflows

In [Jacob et al., 2012a], we presented the initial work on domain-specific modeling for signature discovery workflows Workflow engines like JBPM [Cumberlidge, 2007] also provide graphical user interfaces for designing and deploying workflows. Other related works include

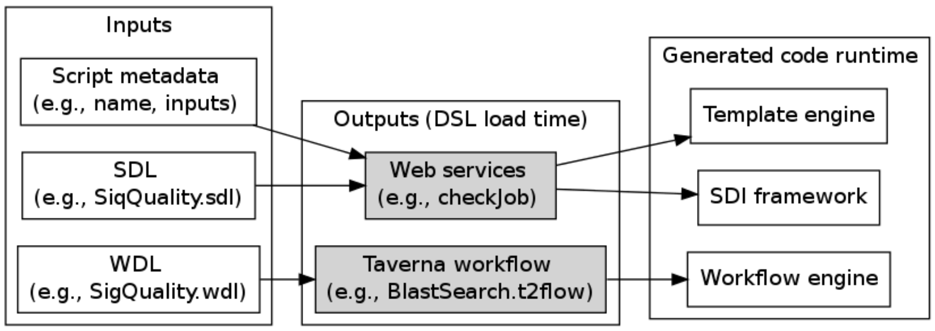


Figure 5.3: Block diagram showing implementation of the approach

languages [Taylor, Deelman, Gannon, and Shields, 2006; Wilde, Hategan, Wozniak, Clifford, Katz, and Foster, 2011] that are designed for composing computation intensive applications. Compared to domain-independent workflows like JBPM and Taverna, the framework has the advantage that it is configured only for scientific signature discovery workflows. In Taverna, a Web service can be of different types and WSDL services are just one example. To keep the uniformity, every time a WSDL processor is created, users have to add transformations to parse the XML input and output. Our framework handles these details automatically through service and workflow definition languages. Moreover, since the output is generated as a Taverna workflow file, it can be viewed, edited and executed in Taverna's full-fledged workflow development environment. There are many tools [Maximilien, Wilkinson, Desai, and Tai, 2007; Pruett, 2007] available for creating applications by composing web services from different vendors. Most of these tools assume that the Web services are available. Our framework configures the workflow definition file that declares how to compose services wrappers created by the framework.

5.4 Implementation Details

Figure 5.3 shows the overall approach to generate both wrappers and workflows. Scripts or commands with template variables, SDL and WDL files are the only inputs. Using these input files, Web service wrappers and a workflow file deployable to a workflow engine (such as Taverna) is generated. Each Web service wrapper is created from scripts and the associated SDL files, and the workflow file is created using both SDL and WDL files. The code generation occurs in two stages:

1. Web application creation. The tool creates Web services from the SDL files that describe the key elements of a script;

2. Workflow creation. The SDL file from the first stage defines a service. It is passed together with the WDL file to create the workflow constructs. These constructs are the basic elements for the Taverna engine to create a workflow as defined in WDL.

At the load time of the SDL file, a web application with Web service wrappers corresponding to each service in the SDL is created. Similarly, a “t2flow” (Taverna workflow executable) file is created during the load time of WDL. When the t2flow is executed, the wrapped script(s) and command(s) are executed in the remote host through an SSH session with the help of existing signature discovery libraries (e.g., SDI). These libraries are responsible for making the input files available before execution and uploading the output files to a dedicated data management system after execution. We make use of a template engine to access runtime values of variables inside services.

5.4.1 SDL Definition and Wrapper Generation

An SDL file has a list of services, where each service has a connection parameter ("SSH" details), an execution command parameter (command to execute), resources (additional scripts required to execute the command), and a set of inputs and outputs required for the execution of the service. If the connection parameter is not specified, a service will be executed in the server in which the application is deployed. SDL supported input and output and their code generation details are described in the following subsections.

5.4.1.1 Input and Output Strings

String is the default parameter type in SDL. Any parameter defined without a modifier other than "in" and "out" are treated as string parameters. The "in" and the "out" modifiers define the directions of the parameters. Each input string parameter is treated as a template variable and is applied to the scripts and commands; any occurrence of the variable will be replaced by its runtime value before execution. If there is an output string variable, code is generated to read a `.properties` file after execution.

5.4.1.2 List of Documents or String

Code is generated to apply the document property or string property for all elements in the list. As shown in Figure 5.5.a, while using the template variables and list, an end-user has to be aware that a single variable may be substituted with many values.

5.4.1.3 Wrapper Generation

Code is generated for a Web service wrapper for each SDL file using the name of the SDL file. For every service, two artifacts are generated; namely, the interface class and the corresponding implementation class. In addition, the framework also generates a helper class, called `SSHHelper`, which automates the connection with remote computers.

5.4.2 SDL Example: BLAST Execution

The code generator creates service wrappers for each step. The job submission service is shown in Figure 5.2a. A BLAST job is submitted using two script files: "jobScript.sh" (a SLURM file) and "runJob.sh" (a BASH file). The script file "runJob.sh" executes the SLURM file and writes `jobID` and `outDir` to the `.properties` file. In Figure 5.2.a, `submitBlast` has two outputs, the execution directory (`outDir`) and the job identifier (`jobID`). Both outputs are declared as the default type; hence, the framework generates code for the string outputs. The generated code downloads and reads their values from a `.properties` file. Other services (`blastResult` and `checkJob`) are command SDL service wrappers (no script files) and are not shown. Service `checkJob` checks the status of a given `jobID` and returns status “Running, Pending, or Done.” Service `blastResult` downloads the files from a given directory.

At runtime, values and fields inside the script files need to be exposed as the input or output of service wrappers. This is achieved by treating the scripts as templates and their runtime values as template variables. The template variables are enclosed inside the “$” character. Before executing any script files in the server, the scripts are passed through a template engine. The template engine substitutes the scripts with runtime values. Template substitution is not necessary for the files, as their runtime names are the same as the load time names. As an example, in Figure 5.2.a (line 25), the `blastResult` service is defined with a template variable `outDir`. When the `blastResult` is executed, the `$outDir$` in the command (line 25) will be replaced by the runtime value of the variable `outDir`.

ANTLR StringTemplate is used for the template implementation. Hence, many advanced features of the template engine can be utilized. As an example, to implement a Web service that can aggregate all of the input files, an SDL file is defined (Figure 5.5.b) with an input as a “list” of documents named `inputs` (line 3). Because the variable type is a list, the template is applied for all the values. Using the StringTemplate “separator” keyword, the template engine separates the individual values with a space.

In the case of a text output, the code is generated to read the `.properties` file and return the specified property value. Hence, a user has to make sure that the output text values are written to the `.properties` if they want the service to return the value. As shown in Figure 5.2.a, after submitting a BLAST job, the job identifier and the output directory is written to the .properties file (commented lines 16 and 17). If there are multiple outputs, a new object return type is created with fields as the outputs specified and returned.

5.4.3 WDL Definition and Workflow Generation

A WDL file creates a Taverna workflow based on the descriptions specified by the user. A WDL workflow involves communication and interactions of various service wrappers among each other and also with other workflows. A WDL file can have many workflows, but the top-most workflow is considered the main workflow with all other workflows treated as sub-workflows. Code is generated for sub-workflows only if it is called inside the main workflow.

Workflows have declarations of elements and connections. The Xtext[46] grammar for WDL is shown in Figure 5.4. Elements can be of any three types: 1) Sub-workflow, 2) Services, and 3) Strings. Strings are required to be initialized along with their declaration. Connections also have three types: 1) Workflow input to service port, 2) Service port to service port, and 3) Service port to workflow output or sub-workflow calls. A sub-workflow call in WDL introduces loops and abstractions (function calls) to another WDL. It can connect a sub-workflow with the connection ports. The sub-workflows, services and strings can be used without explicit declaration in that case; their name itself will be used as the identifier. The main workflow uses a "*call-till-with*" structure for communicating with sub workflows. As shown in Figure 5.4 (lines 13-16), using this structure, the main workflow can iteratively "call" a sub workflow "till" it meets a condition "with" main workflow ports. Hence, the "*call-till-with*" structure is a replacement for function calls and loops in WDL.

[46] Xtext Language tools, `http://www.eclipse.org/Xtext/`

```
grammar gov.pnl.sdi.WDL with org.eclipse.xtext.common.Terminals
generate wDL "http://www.pnl.gov/sdi/WDL"

WorkflowModel:
  'use' servicefile=STRING workflows+=Workflow+;

Workflow:
  'workflow' name=ID '('parameters=Parameters? ')' '{'
        (definitions+=Definition*)
        (stringConstants+=StringConstant*)
        (portlinks+=PortLink|serviceLinks+=ServiceLink|workflowCalls+=WorkflowCall)+'}';

WorkflowCall:
  'call' workflowID=ID
   ('till' criterion=Criterion)?
  'with' argument=Port (',' moreArguments+=Port)*;

Criterion:
  port=ID op=OPERATOR value=STRING ;

OPERATOR:
  '='|'<'|'>';

StringConstant:
  'String'  stringAssignments=StringAssignments;

StringAssignments:
  assignment=StringAssignment(',' moreAssignments+=StringAssignment)*;

StringAssignment:
  name=ID'=' value=STRING;

Definition:
  'workflow'|'service') name=ID services=Services;

Services:
  service=ID (','  moreServices+=ID)*;

ServiceLink:
  service1=ID'|' service2=ID;

PortLink:
  (port1=Port|text=STRING)'->' port2=Port;

Port:
  serviceName=ID('.'portName=ID)? ('after' afterServiceName=ID)? ;

Parameters:
  parameter=Parameter (',' moreParameters+=Parameter)*;

Parameter:
  type=('in' |'out') variable=ID;
```

Figure 5.4: Xtext grammar for WDL

A WDL specification file is used to define workflows. There can be many workflows inside a WDL file. The first workflow in the file is the main workflow. Accordingly, a Taverna workflow file will be generated with the same name as the main workflow with an added “t2flow” extension.

The other workflows used inside the main workflow are sub workflows. The input for the services in a SDL specification file can be defined in two ways:

1. *Direct service specification*: A service name followed by the operator '.' and the input variable name specified in the SDL file (Figure 5.2.b, line 9-11)

2. *Indirect service specification*: An identifier defined as a service, followed by operator '.' and the variable name (Figure 5.6.b, line 18).

The indirect service specification is required when there is more than one occurrence of the same service. The '$->$', operator can be used for connecting any of the three cases, such as connecting a workflow input to a service input, connecting a service output to another service input, or connecting a service output to a workflow output. We introduced the "*call-till-with*" construct to implement loops and abstraction in the WDL. Using the structure, sub-workflows can be included into the main workflow. Using the optional "*till*" construct, the termination condition for a sub-workflow can be specified. The sub-workflow will be executed only once as shown in the landscape classification example in Figure 5.6.b. The support for a conditional expression is restricted by the underlying workflow engine. Hence, for the expression, the first operand has to be a sub-workflow port and the second operand has to be a string. WDL supports three logical operators: '$=$', '$<$', and '$>$'. Using the "*with*" construct, the input and the output of the sub-workflow are specified similar to arguments of a function call. If the arguments are defined as an "out" type in a sub-workflow, the sub-workflow writes to the port after execution. Otherwise, it reads from the port before the execution.

5.4.4 WDL Example: BLAST Execution

The WDL file for the BLAST workflow is shown in Figure 5.2.b. The WDL specification has two workflows: 1) `BlastSearch`, the main workflow, 2) `checkJob`, the sub-workflow. The main workflow `BlastSearch` has the following steps:

1. submits a BLAST job after passing inputs to the `submitBlast` service (lines 9-11)

2. passes the `jobID` to the sub-workflow (line 16) and the output directory to the `blastResult` service and waits for the sub-workflow to continue execution until it meets a criterion on finishing the sub-workflow (lines 14-16);

3. downloads the output file using service `blastResult` and passes it to the workflow output (line 22).

In Figure 5.2.b, a workflow named `checkJob` is defined (lines 28-32). The workflow takes one input (`wf_jobID`) and returns one output (`wf_status`). The workflow input `jobID` is passed to the service, `jobStatus` (line 29), which is defined using SDL to find the status of the specified job. The workflow output status is fetched from the service output `status` (line 30)

The order of execution is determined by the Taverna engine, which executes all the services whose outputs are available. In some cases, scientists might want to control the order as it may not be obvious to the Taverna engine. As an example, for the BLAST execution, the `blastResult` service that downloads the output files needs to wait for the BLAST job to complete. To allow a service to be executed only after the specified workflow or service, the "*after*" keyword is provided. The "*after*" keyword can be added to any service invocation (Figure 5.2.b, line 19). More than one service can also be in the "*after*."

```
1 service class_Training {
2   use  ssh_exe;
3   cmd "R CMD BATCH training.r training.out";
4   resource "training.r";
5   in doc trainXFile , trainYFile;
6   in algorithm;
7   out doc modelFile;
8 }
```

(a) Service executing R script

```
1 service aggregate{
2   use ssh_exe;
3   cmd "cat $inputs;  separator=\" \"  $ >
      ↪aggregatedFile" ;
4   in list doc inputs;
5   out doc aggregatedFile;
6 }
```

(b) Service to add contents of a list of files

Figure 5.5: SDL examples

The language design and the associated software tools are available in Eclipse. We define the syntax of service descriptions and workflows using Eclipse Xtext. The syntax is defined according to Taverna workflow elements and processes. The code generation uses CXF[47] APIs for creating Web service wrappers for script files.

5.5 SDL Service Examples

In this section, some of the services created using SDL for Signature Discovery workflows are described. In Figure 5.5, two examples are shown: 1) Figure 5.5.a shows wrappers for R[48] scripts using SDL, and 2) Figure 5.5.b shows a utility wrapper service that can perform a vertical merge for a list of files. Using the SDL service shown in Figure 5.5.a, a Web service wrapper is created than executes an R script `training.r` with input files, `trainXFile` and `trainYFile`. The R scripts can be executed using the command "`R CMD BATCH scriptfile`" as shown in Figure 5.5.a (line 3). Similarly, the UNIX utility `cat` command is used in the `aggregate` service to aggregate files (line 3 in Figure 5.5.b).

An overview of ten services carefully selected to highlight the core code generation features of the SDL parser is shown in Table 5.1. In the table, Lines of Code (LOC) is defined as the additional LOC generated to include the service. This does not include abstract classes or class

[47] Apache CXF: An Open-Source Services Framework, `http://cxf.apache.org`
[48] R Language for Statistical Computing, `http://www.r-project.org/`

Table 5.1: An overview of SDL code generation

No	Service	Utils/Script	[Inputs (type)] [Outputs(type)]	LOC	Total LOC (files)
1	`echoString`	`echo`	[0][1 (doc)]	10+13+1+6	30(4)
2	`echoFile`	`echo`	[1 (String)] [1 (doc)]	10+14+1+6	31(4)
3	`aggregate`	`cat`	[1(List doc)] [1 (doc)]	10+20+1+7	38(4)
4	`classifier_`Training	R	[2 (doc), 1 (String)] [1 (doc)]	11+24+2+8	45(4)
5	`classifier_Testing`	R	[3 (doc), 1 (String)] [1 (doc)]	12+29+2+8	51(4)
6	`accuracy`	R	[1 (doc)] [1 (doc)]	11+19+1+6	37(4)
7	`submitBlast`	`SLURM, sh`	[3 (doc)] [2 (String)]	17+27+2+8+18	72(5)
8	`jobStatus`	`SLURM`, sh	[1 (String)] [1 (String)]	10+14+1+6	31(4)
9	`blastResult`	`cp`	[1 (String)] [1 (doc)]	10+14+1+6	31(4)
10	`mafft`	`mafft`	[1 (doc)] [1 (doc)]	10+18+1+6	35(4)

definitions, if the class already exists. SDL code generation takes place as: 1) Adding a method in a Java RMI interface, 2) Adding a method in a Java SEI interface, 3) Implementing a method in a Java class through a Helper class, 4) Creating the helper class for the service, and 5) Creating a new Java bean, if there is more than one output. Code generated affecting multiple Java classes (e.g., five, if there are more than one output) for each service wrapper are shown in the LOC column of the table.

5.6 A WDL Case Study: Landscape Classification

In this section, we demonstrate how the current approach can be used to define and deploy signature workflow through a Landscape classification example. In the landscape classification workflow, the goal is to compare accuracies of three landscape classification algorithms. Each landscape classification algorithm is developed as R scripts and involves four stages:

1. *Training stage*: At this stage, image data along with their actual landscape classification values are given as input. A model (a function approximation, generalized from the input training patterns) is created and the model is returned as the output;

2. *Testing stage*: The image data and the estimated model from the training stage are passed as inputs. The estimated classification type is returned as output.

```
1 use "SigAnalysis.sdl"
2 workflow algo_classify (in algo, in trainX,
    ↪in trainY,
3   in testX, in testY, out outFile){
4
5  trainX->classifier_Training.trainXFile
6  trainY->classifier_Training.trainYFile
7
8  algo->classifier_Training.algorithm
9  algo->classifier_Testing.algorithm
10
11 classifier_Training.modelFile ->
      ↪classifier_Testing.modelFile
12
13
14 testX->classifier_Testing.testXFile
15 testY->classifier_Testing.testYFile
16
17 classifier_Testing.outFile ->outFile
18 }
```

(a) Landscape classification using WDL

```
1 workflow Classifier (in trainX, in trainY,
    ↪in testX,
2   in testY, out finalOut) {
3
4  workflow algo_classify lda_class,
      ↪knn_class, svm_class
5
6  service aggregate agg
7
8  call lda_class
9  with "lda", trainX, trainY, testX, testY,
      ↪agg.inputs
10
11 call knn_class
12 with "knn", trainX, trainY, testX, testY,
      ↪agg.inputs
13
14 call svm_class
15 with "svmRadial", trainX, trainY, testX,
      ↪testY, agg.inputs
16
17
18 agg.finalOut->accuracy.inputFile
19
20 accuracy.outputFilePrefix-> finalOut
21
22 }
```

(b) Classification accuracy using WDL

Figure 5.6: WDL examples

3. *Aggregate stage*: We want to perform the first two stages for three different algorithms: LDA (Linear Discriminant Analysis), KNN (K-Nearest Neighbor), and SVM (Support Vector Machine). The result of the algorithms is merged to a single output file. This can be implemented using a simple `cat` command.

4. *Accuracy stage*: Based on the estimated value and actual value, another R script is also available that can calculate the accuracy of a classification algorithm. SDL service definitions for Training and Aggregate stages are shown in Figure 5.5. Similarly, we can define service wrappers for the testing stage and accuracy services.

The next step is to define the workflow. Figure 5.6.a shows a workflow `algo_classify` that imports the service definitions and connects the training and testing stages through their inputs. The inputs of the workflow are passed to the `classifier_Training` (lines 5-8) and `classifier_Testing` (lines 9 14, 15). The output of `classifier_Training` is passed to `classifier_Testing` (line 11), and output of `classifier_Testing` is passed to the workflow output `outFile` (line 17).

The workflow shown in Figure 5.6.a can return the classification results for a set of inputs and an algorithm. In Figure 5.7, a workflow `Classifier` is written with the same inputs as Classifier excluding the `algo` input. Three workflows, `lda_class`, `knn_class`, and `svm_class` are defined as type workflow `algo_classify` (line 4). The `Classifier` workflow is called for values “lda”, “knn”, and “svmRadial” (lines 8-9, 11-12, and 14-15). A service of type aggregate `agg` is also defined (line 6). The output of each workflow `algo_classify` is written to the input of `agg` (lines 9, 12, 15). Hence, the output of `agg (finalOut)` has the aggregated results of all the three classifications. This is given to the input of the accuracy `inputFile` (line 18) and the output of the service is given to workflow output (line 20).

The final output workflow executable generated is shown in Figure 5.7. As seen from the figure, it would be challenging to maintain a workflow in that form. It has 50 processors and a similar count of connections. Using this approach, scientists do not have to deal with the engineering processes to make executables globally available and accessible in a mature workflow engine like Taverna, but the approach provides all of the advantages and features of executing the workflow in the Taverna workbench.

5.7 Discussion

This chapter describes the design and implementation of a framework for converting scripts into scientific workflows. It includes how the domain-specific information required to create the workflows are separated from the accidental complexities introduced by Web services and the Taverna workflow engine, which allows end-users (scientists) to design and develop workflows.

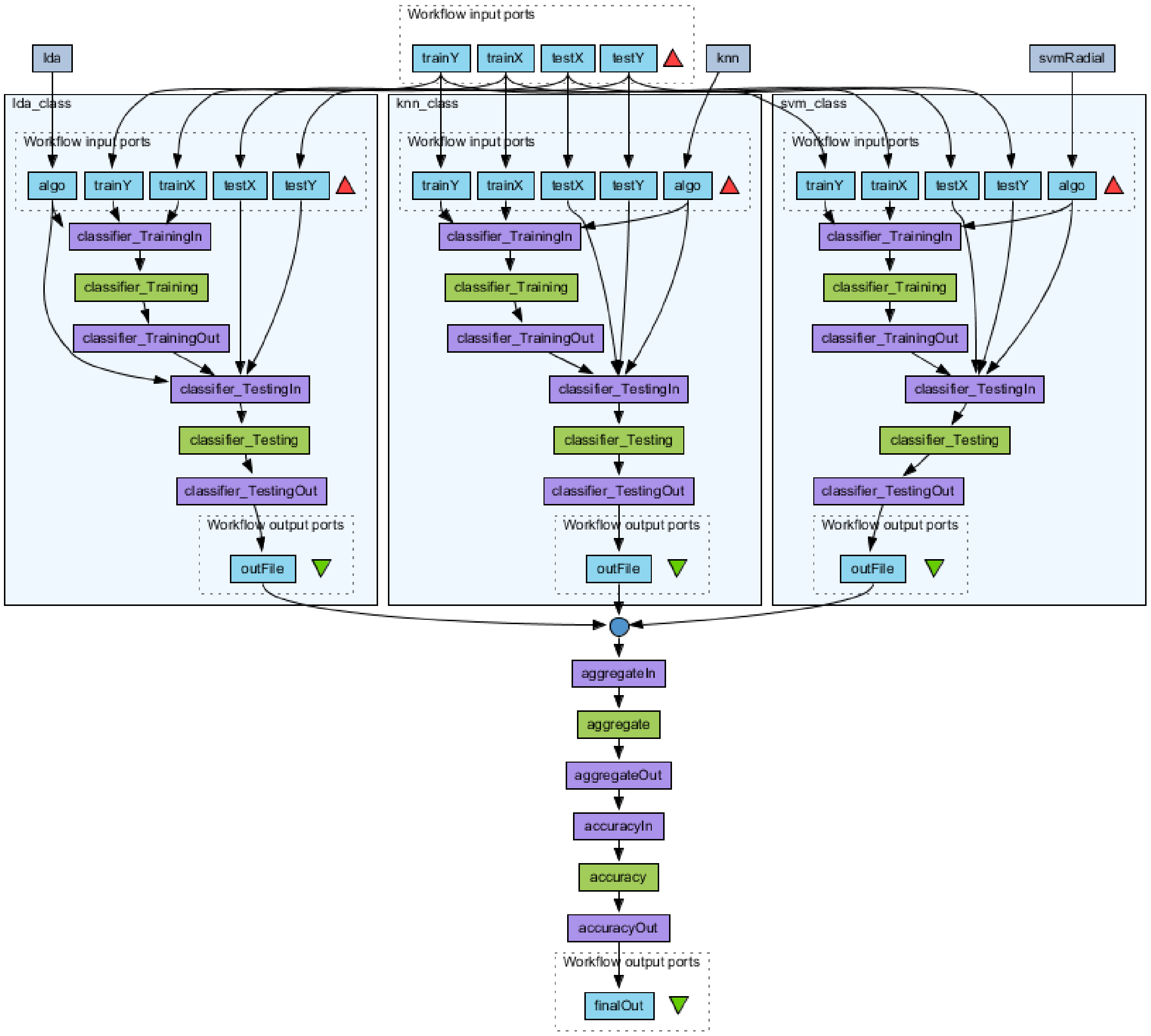


Figure 5.7: Taverna workflow for classification accuracy generated by WDL

The framework was evaluated with two real-world examples that are used to develop and deploy two scientific workflows.

Chapter 6

PNBSOLVER: A SUB-DOMAIN-LEVEL ABSTRACTION FOR PARALLEL N-BODY SOLUTIONS

With the advent of multicore processors, parallel computation has become a necessity for next generation applications. It is often a tedious task for domain users to optimize their programs for a specific platform, algorithm, and problem size. We believe that domain users should be freed from this task and they should be equipped with tool support to reuse the optimized solutions written by expert parallel programmers. In this chapter, we introduce a two-stage modeling approach that allows domain users to express the problem using domain constructs and reuse the available optimized solutions. This approach has been applied successfully to Nbody problems using a DSL called PNBsolver, which allows domain users to specify the computations in an Nbody problem without any implementation or platform-specific details. Using the PNBsolver, the domain users are allowed to control the platform and implementation of the generated code.

In the last three chapters, we described how code, algorithm, and program level modeling can help HPC users. In this chapter, we apply our approach to a very restricted domain: Nbody problems. We introduce PNBsolver, which is a DSL-based implementation of our sub-domain-level modeling.

Section 6.1 introduces two research questions and explains how domain-specific modeling can address these questions. The related works are reviewed in Section 6.2. Section 6.3 reviews Nbody problems and analyzes a commonly used algorithm in such problems. Section 6.4 describes

PNBsolver from a user's perspective and the implementation details are explained in Section 6.5. Section 6.6 gives details of a comparison study of PNBsolver solutions with existing implementations. Additional examples of using PNBsolver are shown in Section 6.7. The chapter is concluded in Section 6.8.

6.1 Domain-Specific Modeling

Popular parallel programming paradigms include: 1) OpenMP (Shared memory), 2) Message Passing Interface (Distributed memory), and 3) CUDA and OpenCL for GPU platforms. All of these parallel programming paradigms have their advantages and disadvantages for a given problem, based on the execution environment, implementation, and even size of the problem. In the current best practice, to find the optimized execution time for a problem of a given size in a specific platform, a programmer must manually write a parallel version, optimize it and compare the execution time with other platforms. It is often hard for programmers to optimize these programs as their domain of expertise is the problem domain and not the High Performance Computing (HPC) domain. To solve this problem, the programmers should be freed from implementation and optimization of parallel versions, but should have the flexibility to express the problem and switch between execution environments (optimal execution time is also a function of problem size). There is no magical tool to convert any sequential program to an optimized parallel program. However, for a restricted domain this can be achieved [DeVito, Joubert, Palacios, Oakley, Medina, Barrientos, Elsen, Ham, Aiken, Duraisamy, Darve, Alonso, and Hanrahan, 2011; Heroux, Bartlett, Howle, Hoekstra, Hu, Kolda, Lehoucq, Long, Pawlowski, Phipps, Salinger, Thornquist, Tuminaro, Willenbring, Williams, and Stanley, 2005; Püschel, Moura, Johnson, Padua, Veloso, Singer, Xiong, Franchetti, Gačić, Voronenko, Chen, Johnson, and Rizzolo, 2005]. In this section, we address two research questions that are summarized in the following subsections.

6.1.1 Q1: Separating Specification and Implementation

Which algorithm on a specific platform can provide the optimized execution for a problem of fixed size? To answer this question, the best way is to execute a program in different platforms implementing various algorithms. This leads to the task of rewriting thousands of lines of code. Can we re-use the implementation for another similar problem? This is possible only if the implementations details are separated from the problem specification. Is there another new algorithm which can give the optimized execution for the specific-problem than any of the existing algorithms? This cannot be linked with the existing code even if the problem specification is separated from the implementation.

6.1.2 Q2: Abstract Problem Specification

How can we maintain and switch between many program versions to provide the optimized execution time? As mentioned earlier, there can be many implementations for a given problem specification. If all such implementations are based on an abstract problem specification, any of these implementations can be used with the given problem specification. If the domain is limited, we can provide optimized solutions in required platforms for each of these implementations.

We used modeling techniques from software engineering to provide separation of specification and implementation, and abstraction in a restricted domain. Our solution approach is explained in the following subsection.

6.1.3 Solution Approach: Modeling Parallel Programs

We developed a two-stage modeling technique for modeling parallel programs. In the first stage, the domain experts create a model (DSL file) specifying the problem (this model has no platform-specific or implementation-specific details) and in the second stage, this model is mapped to different platforms and implementations based on user preferences. The main advantage of this two-step modeling technique is that it can be extended to include additional platforms and implementations and is well-suited for a domain like parallel programming.

6.2 Related Works in Nbody Solvers

The related works are classified into two categories, as explained in the following subsections.

6.2.1 Nbody Solutions

A major advance in Nbody simulation was the introduction of the GRAPE (Gravity PiPE) series of special-purpose computers [Makino and Taiji, 1998]. But recent works show that comparable speedup can be achieved through GPUs [Belleman, Bédorf, and Portegies Zwart, 2008]. AMBER [Pearlman, Case, Ross, Cheatham, DeBolt, Ferguson, Seibel, and Kollman, 1995] is a package of computer programs written in FORTRAN to simulate the structural and energetic properties of molecules. AMBER used a non-bonded cutoff sphere for applying approximation in Nbody energy equations. AMBER 4.0 adds parallelism to modules of the program. NAMD [Kalé, Skeel, Bhandarkar, Brunner, Gursoy, Krawetz, Phillips, Shinozaki, Varadarajan, and Schulten, 1999] is a parallel simulation program written in C++ and used to simulate the behavior of bio-molecular systems. NAMD2 uses Converse (a portable runtime framework) to support interoperability between different parallel paradigms. FMM-Yukawa [Huang et al., 2009] is a FORTRAN program package for fast evaluation of the screened Coulomb interaction of N particles. Programs can execute in linear time for nearly uniform particle distributions. Compared to PNBsolver, these softwares or programs have targeted a specific field and to a specific Nbody problem. Matlab is a high performance language for technical computing where solutions and problems are expressed in mathematical notations. Being a general tool, it is highly improbable to have the expressiveness or performance as compared to the domain-specific PNBsolver, which can be extended to support more algorithms by adding templates and allow programmers to fine-tune these algorithms based on their specific problem.

6.2.2 Tree Code Algorithms

Cheng et. al. [Cheng, Huang, and Leiterman, 2006] developed an adaptive fast solver for a modified Helmholtz equation in two dimensions using an adaptive quad tree structure. Ossmani and Poncet [Ossmani and Poncet, 2010] used a tree based data structure to evaluate efficiency of a multi-scale hybrid grid-particle vortex method. Krasny and Duan [Krasny and Duan, 2002] used a tree code algorithm to compute non-bonded particle-cluster and cluster-cluster interaction. They also used an oct-tree data structure and implemented different tree traversing techniques specific to different boundary conditions. Krasny and Wang [Krasny and Wang, 2011] implemented the tree code in Cartesian coordinates and used a recurrence relation to compute the Taylor coefficient for evaluating sums of multi-quadric radial basis function (RBF). Xu [Xu, 2010] used a tree code algorithm for calculating polarized Coulomb interaction of an Nbody system. Li et. al. [Li, Johnston, and Krasny, 2009] used the algorithm for evaluating electrostatic potential of screened Coulomb interaction using a new recurrence relation. Baczewski and Shanker [Baczewski and Shanker, 2011] presented an Accelerated Cartesian Expansion (ACE) method to evaluate periodic Helmholtz, Coulomb and Yukawa potential using $\mathscr{O}(N)$ operations and $\mathscr{O}(N)$ storage using different approximations in both tree building and tree traversing. All of these works represent different variations of the original tree code algorithm. Because PNBsolver is more focused on the approach than to a specific algorithm, PNBsolver can support any of these.

6.3 Nbody Problems and Tree Code Algorithm

The Nbody problem is a generalized concept of problems related to the physical properties of a system. The system can contain billions of interacting bodies. The classic Nbody problem of celestial mechanics originated from Newton's Principia [Diacu, 1996]. The bodies can be giant celestial objects in a solar system or minuscule particles in an atomic system. The physical properties can be motion, energy, charge, momentum, or force. There are many areas in science that use Nbody problems (e.g., astrophysics, plasma physics, molecular physics, fluid dynamics, quantum chemistry and quantum chromo-dynamics) [Carlson et al., 1983; Jastrow, 1955; Sasai and Wolynes, 2003; Watson, 1953]. Barnes et. al [Barnes and Hut, 1986] introduced a hierarchical $\mathcal{O}(N \log N)$ algorithm to calculate an Nbody force equation and was later used in different Nbody problems [Krasny and Duan, 2002; Krasny and Wang, 2011; Xu, 2010]. In this section, the tree code algorithm is reviewed, explaining how the efficient implementations of this algorithm can be re-used in Nbody computations.

6.3.1 Tree Code Algorithm

The tree code algorithm has proved its efficiency for Nbody problems [Barnes and Hut, 1986]. It reduces the computational cost of these problem from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log(N))$. The tree code algorithm is explained in two perspectives: 1) Mathematical formulation; and 2) Source code. In the mathematical formulation, the algorithm is explained with the help of a 3D cube, which itself is divided in all dimensions into subunits recursively. In the source code, this is represented as a tree structure.

6.3.1.1 *Tree Code Algorithm: A Mathematical Perspective*

$$I_i = \sum_{j=1, j\neq i}^{N} M_j G(x_i, y_j) \tag{6.1}$$

In a system consisting of N bodies, the interaction of body i with other $N-1$ bodies is given in Equation 6.1 [Krasny and Wang, 2011]. According to the tree code algorithm, for bodies within a cell/cluster (one of the subunits of the 3D cube) satisfying the Multipole Acceptance Criterion (MAC, θ) [Salmon and Warren, 1994] in Equation 6.2, their interaction with body i can be calculated more efficiently with controllable errors. In Equation 6.2, r_c represents the location of the cell center and R represents the distance between the ith body and the cell center. On satisfying the MAC, the more efficient particle-cluster interaction between body i and bodies inside a cell is given as $I_{i,c}$ in Equation 6.3 from [Krasny and Duan, 2002; Li et al., 2009; Lindsay and Krasny, 2001]. Note here k, x, and y are vectors with three components.

$$\frac{r_c}{R} < \theta \tag{6.2}$$

$$I_{i,c} \approx \sum_{||k||=0}^{p} \frac{1}{k!} D_x^k G(x_i, y_c) \sum_{y_j \in c} M_j (y_j - y_c)^k \tag{6.3}$$

$$I_{i,c} \approx \sum_{||k||=0}^{p} TCOEFF(k, G) \times MOMENTS(c) \tag{6.4}$$

In Equation 6.3, for each 3D value k, the first term computes the Taylor coefficients for every target body (using recurrence relations) and the second term computes the moment of the cluster, which is only evaluated once for each cluster. In general, Equation 6.3 can be rewritten as Equation 6.4. To implement this algorithm for any interaction, programmers only need to choose one of our included interactions or provide the Taylor coefficients for user-specified interactions. In Equations 6.3 and 6.4, *p* represents the ORDER of the Taylor coefficients.

6.3.1.2 Tree Code Algorithm: A Source Code Perspective

A general implementation of the tree code has three stages: 1) Tree creation, 2) node values calculation, and 3) Tree traversal. Details of these stages are explained as follows:

1. **Tree creation:** At this stage, a tree-based structure is used to maintain a list of bodies. Bodies of the system are added to the root node and if any node has more than a fixed number of bodies (MAXPARNODE), the node is further branched to eight child nodes. This is equivalent to dividing a 3D space recursively until each of the existing subunits has at most MAXPARNODE number of bodies.

2. **Node values calculation:** At this stage (usually implemented along with tree creation), the node properties are calculated. This includes calculating the resultant center of the node and arranging inputs such that bodies in the same node are arranged closer for faster access. Note that each intermediate node of the tree contains the approximate interaction information (the second term, moments, in Equation 6.3) of the sub-units under that intermediate node. Hence, the root node can approximate the entire system.

3. **Tree traversal:** In this stage, the interactions are calculated. The bodies are traversed from the root node. If a node satisfies the MAC, traversal is stopped and a node's approximated value is used as the interaction. If MAC is not satisfied, the traversal is continued until it reaches the leaves. At the leaves, interaction is computed by direct summation.

A summary of four tree code implementations collected from different sources [Ansari, 2012; Barnes, 2012; Johnston, 2012] implementing different interactions is shown in Table 6.1.

Table 6.1: An analysis of existing tree code implementations

Name	Equation	Language	LOC (files)
Total PE	$\sum_{i=1}^{N-1} Q_i \sum_{j=i+1}^{N} \frac{Q_j}{\lvert R_i - R_j \rvert}$	FORTRAN	966 (3)
PE$_i$	$\sum_{j=i,j\neq i}^{N} \frac{Q_j}{\lvert R_i - R_j \rvert}$	FORTRAN	1006 (3)
Screened$_i$	$Q_i \sum_{j=i,j\neq i}^{N} \frac{Q_j e^{-\lvert R_i - R_j \rvert}}{\lvert R_i - R_j \rvert}$	FORTRAN	1023 (3)
Grav. F$_i$	$M_i \sum_{j=i,j\neq i}^{N} \frac{M_j (R_i - R_j)}{\lvert R_i - R_j \rvert^3}$	C	1921 (15)
Grav. F$_i$	$M_i \sum_{j=i,j\neq i}^{N} \frac{M_j (R_i - R_j)}{\lvert R_i - R_j \rvert^3}$	Javascript	908 (6)

We leave out some constants such as dielectric or gravitational constants for simplicity. As shown in the table, all the implementations had roughly 1000 lines of code (LOC).

6.3.2 Analysis Summary

There are parallel implementations of tree code available [Bédorf, Gaburov, and Zwart, 2012; Burtscher and Pingali, 2011; Liu, Duan, Krasny, and Zhu, 2004]. In general, there are two parallel implementations: 1) Parallelizing the node interactions, which are distributed equally among the parallel instances (threads or processes) and every instance creates their own tree (same tree) for computations; 2) Parallelizing tree creation and node interactions, the tree creation, as well as tree walking, is distributed among the instances. In most cases, direct computation is embarrassingly parallel. Hence, when such a computation is executed in a highly parallelized device like a GPU, the direct computation can outperform the tree code algorithm for a range of size. For a larger size problem, the $\mathscr{O}(N\log(N))$ tree code can outperform the $\mathscr{O}(N^2)$ direct summation. This justifies our combination of parallel direct summation implementation in a GPU, parallel implementation of tree code in a CPU, and parallel implementation of tree code in a GPU.

The research questions introduced in Sections 6.1.1 and 6.1.2 are revisited in the context of Nbody problems in the following subsections.

1. **Separating specification and implementation in Nbody problems:** The domain users should be able to specify the Nbody problems without any implementation details (e.g., which algorithm to use). They should also be able to provide the implementation details (e.g., desired accuracy, allowed error, platform). However, the problem specification should be logically separated from the other details.

2. **Abstract problem specification in Nbody problems:** The problem should be represented at the correct abstraction level. It should not be too high so we cannot apply code optimizations (e.g., users should be allowed to configure parameters like MAXPARNODE, ORDER, THETA) and not too low level (e.g., users might not want to specify the bodies as charge or mass).

PNBsolver is designed to address these questions. PNBsolver is introduced in the following section from a user's perspective.

6.4 Working with PNBsolver

The equation to calculate the gravitational force for each body in an Nbody system with masses M_i and positions R_i is shown in Equation 6.5 [Salmon, 1994]. An equivalent representation of the force using PNBsolver is shown in Figure 6.1. In the figure, the `Force` kernel is defined in space `R`, mathematically $\mathbb{R}^3$. A "pnb" (parallel Nbody) file can be logically divided into three sections: 1) Declaration , 2) Calculation, and 3) Generation sections. Each section is explained in the following subsections.

```
1   kernel Force in R
2
3   // Kernel declarations
4     vector F
5     scalar M
6     constant K=1
7
8    /*
9     * Read positions and mass from file
10    * formatted as <x y z>m
11    *
12    */
13    read "<R_1,R_2,R_3>M", "data.dat"
14
15    //Actual computation
16    F=K*M SUM(M*R/(R_*R_*R_))
17
18    // Write force to file
19    write "F_1,F_2,F_3", "out.dat"
20
21  endkernel
22
23  // Generate CUDA code for force kernel
24  generate CUDA ACCURATE Force.
```

Figure 6.1: Gravitational force kernel in space R using PNBsolver

$$F_i = KM_i \sum_{j=1, j\neq i}^{N} \frac{M_j(R_i - R_j)}{|R_i - R_j|^3} \tag{6.5}$$

6.4.1 PNBsolver Declaration Section

Every variable used in the calculation section should be declared before usage. PNBsolver identifies three types of variables: 1) vector, 2) scalar, and 3) constants. Vectors and scalars are linked to every body in the system. The total force that acts on a body or position of a body at an instance are examples of vectors; the mass or charge of a body are example of scalars. The constant data type is used to specify the constants in the calculation and also the global properties of the system. The gravitational constant or the dielectric constant are examples of constant data types. The scalar and the constant variables can be initialized along with their declaration as shown in Figure 6.1 (line 6). In the case of a scalar, all the `N` bodies will be initialized with the given value.

```
//A cluster not satisfying MAC

//Variable declarations
double dx, dy, dz, dist, distsq, temp1;

//Calculating distance variables
dx = R[i] - tarpos[0];
dy = R[i] - tarpos[1];
dz = R[i] - tarpos[2];
distsq = dx * dx + dy * dy + dz * dz;

//Avoiding i==j
if(distsq > 0.0f){
  dist = sqrt(distsq);

  //Calculating specified variables
  temp1 = dist * dist * dist;

  //Actual calculation
  Forcex = Forcex + M[i] * dx / temp1;
  Forcey = Forcey + M[i] * dy / temp1;
  Forcez = Forcez + M[i] * dz / temp1;
}
```

Figure 6.2: Code generated for the tree code algorithm with MPI/OMP

```
//Variable declarations
TYPE dx, dy, dz, distsq, invdist, temp1;
//Calculating distance variables
dx = Ri.x - tarpos.x;
dy = Ri.y - tarpos.y;
dz = Ri.z - tarpos.z;
distsq = dx * dx + dy * dy + dz * dz;

if(distsq > 0.0f){
  //Using CUDA Fast functions
  if(ACCURATE)
    invdist = rsqrt(distsq);
  else
    invdist = rsqrtf(distsq);

  //Calculating specified variables
  temp1   =  invdist * invDist * invDist;

  //Actual calculation
  Force.x += dx * tarpos.w * temp1;
  Force.y += dy * tarpos.w * temp1;
  Force.z += dz * tarpos.w * temp1;
}
```

Figure 6.3: Kernel code generated for direct summation with CUDA

6.4.2 PNBsolver Calculation Section

Actual calculation for the kernel is specified in the Calculation section. As shown in Figure 6.1 (line 16), the calculation involves two expressions. The first expression, $K * M$ corresponds to the KM_i in Equation 6.5. For discussion, this expression is called the outer expression and the second expression is called the inner expression. PNBsolver expects the inner equation to perform the SUM operation. In addition to the variables declared, PNBsolver identifies the position vector in both inner and outer expressions. For the Force kernel, the position vector is R (line 1). In the outer expression, R represents the actual position, and in the inner expression R gives the relative position with the iteration value. For calculations, the magnitude of a vector can be obtained by adding the "_" to the name of the variable (R_ in line 16). PNBsolver achieves separation of calculations within the expression statement with "(" brackets. The expression used in the brackets are computed to a temporary variable before the final computation, and the temporary variable is replaced with the occurrences of the expression in the final expression statement. As an example, for an equation $var1 * (var2 + var3)$, the equation $var2 + var3$ is computed to a temporary variable and all the occurrences of the equation is replaced by the variable name. This is done to optimize the computation.

Read and write statements: This section helps PNBsolver to integrate with existing programs or functions. In this section, the vectors and scalars that are declared are initialized. The read statements read the values from a file to the variables and write statements write the computed results to a file. Both of these commands take two parameters: 1) Format and 2) File name. The format argument specifies how values for each body are formatted in a line. A single kernel can have many read and write statements. As an example, if masses were specified in a different file, another read statement could be added before the calculation statement. For reading and writing formats, the three co-ordinates of vectors can be accessed by adding "_1, _2," and "_3," respectively, to their names (line 13 and line 19).

6.4.3 PNBsolver Generation Section

The Declaration and Calculation sections can express the Nbody problem without any implementation details. The Generation section controls the implementation and code generation. The default code generation language is "C" and the algorithm is selected based on the mode and target parallel programming paradigms. PNBsolver supports CUDA, OMP, and MPI parallel programming paradigms. There are three modes for each paradigm: 1) ACCURATE, 2) AVERAGE, 3) FAST. The program runs faster when a mode is changed from ACCURATE to FAST and programs give more accurate results when the mode is changed from FAST to ACCURATE. More details about how algorithms are selected based on the mode and paradigms are explained in the implementation section.

A section of code generated for the tree code algorithm and CUDA direct implementation is shown in Figures 6.2 and 6.3. This code computes the Force for the body located at R_i due to another body located at `tarpos`. Both implementations have pre-defined variables, `dx, dy, dz, distsq`. The `distsq` calculates the magnitude of the distance. The `if` statement could be avoided if we set $M_i = 0.0$ before the computation, but our current implementation generates the `if` statement. However, if there is a softening factor to the magnitude, the PNBsolver parser identifies this and removes the `if` statement. An example for such a case is explained in Section 6.7.2.

In Figures 6.2 and 6.3, `temp1` (line 4 and line 2, respectively) is a variable created by using brackets in the expression statement. As shown in the figures, the calculation is implemented in two different ways for the two versions (line 17 in Figures 6.2 and 6.3). For the CUDA implementation, the type of the variable is defined based on the mode, for faster implementations `float` is used and for accurate implementations `double` is used. Hence, the type of the variable is defined as a C++ template variable `TYPE` (variable is removed by actual code during compilation). `ACCURATE` is another template variable to tune optimization and speed. For distance, since CUDA has a fast math function, to perform the square root and inverse, we use that function (Figure 6.3), instead of the square root followed by the division operator (Figure 6.2) as in the CPU code. Use of the `float4` data type for the position and mass is another GPU optimization. If the variable `ACCURATE` is set, CUDA fast functions (e.g., `rsqrtf,__expf(x)`) are used.

6.5 Implementation Details of PNBsolver

A block diagram showing the implementation of PNBsolver is illustrated in Figure 6.4. The "pnb" file is passed to the parser and the parser identifies the mode and platform for selecting the proper template. The parser also generates the kernel code which is further optimized and given to the code integrator. The optimizations applied at this stage are very general and it is not coupled to any specific platform. Removing the brackets with new temporary variables, finding expressions that are repeated, and checking whether softening is applied are examples of optimizations at this stage. The code integrator merges the kernel code to the template and fills the template values. This includes updating function declaration and parameters, function calls and arguments, and declaring type variables and initialization. After this, platform-specific optimizations are applied. Rewriting functions with CUDA fast math functions [NVIDIA, 2007] is an example of such an optimization.

The project is implemented in Java and ANTLR is used for implementing the "pnb" file parser. The template store is implemented using StringTemplate. Two important parts of the implementation are "pnb" file parsing and code generation using templates. The code generation varies based on the mode and platform selected in the "pnb" file. However, the code generated will be either a parallel direct summation implementation or a parallel tree code implementation.

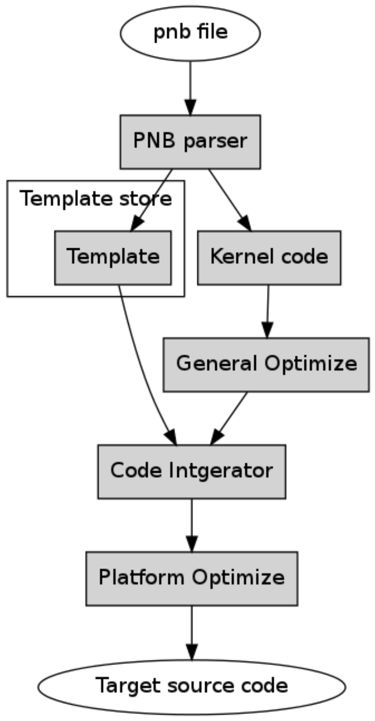


Figure 6.4: Block diagram showing the implementation of PNBsolver

The PNBsolver Parser, PNBsolver code generator, and Modes of operation are explained in the following subsections.

6.5.1 PNBSolver Parser

The simplified EBNF grammar for the PNBsolver language is shown in Figure 6.5. As shown in the figure, the file allows one or more kernel, but code is generated for only one kernel. This is designed to support future extensions, where the output of a kernel can be passed to the input of another kernel. After parsing, the expression statement is captured in two expression objects, the inner expression object and the outer expression object. All expressions in a “pnb” file are made of variables defined in the declaration section. To use a constant inside the expression statement, it should be declared as a constant and use the variable name in the expression statement. As a rule, the read statements should be defined before the expression statement, and write statements after the expression statement. This seems logical as a program usually requires reading before

```

grammar PNBsolver;

content
  : kernels  execute "." EOF
  ;

kernels
  : ("kernel"  ID "in" ID declarations readstmts
            expressionstmt writestmts "endkernel")+
  ;

declarations
  : (type variabledeclaration)+
  ;

execute
  : "generate" platform ("[" parameters "]")? mode ID
  ;

platform
  : "CUDA"|"OMP"|"MPI"|"OCL"
  ;

mode
  : "ACCURATE"|"AVERAGE"|"FAST"
  ;
type
  : "vector"|"scalar"|"constant"
  ;

expressionstmt
  : IDENTIFIER "=" (expression)? "SUM" expression
  ;

expression
  : multidiv( "+" multidiv|   "-" multidiv )*
  ;

multidiv
  : atom ("*" atom |"/" atom)*
  ;

atom
  : IDENTIFIER
  | "(" expression ")"
  | "exp" expression
  | "pow" "("expression "," NUMBER ")"
  ;

readstmts
  : ("read" STRING "," STRING)*
  ;
```

Figure 6.5: Simplified EBNF grammar of PNBsolver

the calculation and writing after the calculation. The EBNF rules for `variabledeclaration`, `writestmts`, `parameters` are not shown in the figure to provide clarity.

```
struct tnode {
    int numbodies, begin, end;
    double x_min, y_min, z_min, x_max, y_max, z_max;
    double x_mid, y_mid, z_mid, radius;
    int level, children, exist_ms;
    double ***moments;
    struct tnode* child[8];
};
```

Figure 6.6: Structure `tnode` in the template

6.5.2 PNBsolver Code Generator

The PNBsolver parser can identity the inputs and outputs for the kernel. The code generator inserts declarations and initialization (if any) into the program. The core computation code is generated as shown in Figures 6.2 and 6.3. These two code sections are inserted into the template identified by mode and platform. To add a new algorithm, new template is defined and configured the parser to route through a different code generator. The same approach can be used to support a language other than C. In Section 6.6, we generate FORTRAN code using a FORTRAN code generator. There are four "C" templates available in the template store and are explained in the following subsections.

6.5.2.1 Parallel CPU (OpenMP and MPI) Tree Code Templates

The tree code used for PNBsolver is adapted from [Krasny and Duan, 2002]. We developed parallel versions of the tree code in MPI and OpenMP. More details about the speedup, error and execution time of the parallel versions are included in Section 6.7. To complete a CPU tree code template, we need: 1) Taylor coefficients, 2) Direct implementation, and 3) Tree code parameters such as MAXPARNODE, ORDER (taylor coefficient order) and THETA (MAC). PNBsolver sets default values for all of these parameters, but this can be customized by the user as shown later in Figure 6.10.

Providing Taylor coefficients: The parameter value TAYLOR is used to find the Taylor coefficients. The PNBsolver code generator reads the file specified as the parameter value of TAYLOR and looks for a function with signature `comptcoeff (struct tnode *t)`, where `tnode` is a structure representing the tree node. In this function, the position of an interacting body can also be accessed. The declaration of structure `tnode` is shown in Figure 6.6. The values of the Taylor coefficients should be set in a 3D array of size ORDER. In addition to the `comptcoeff` function, users can add two more functions: `setup()` and `teardown()`, for declaring values that might be required for the computation. These two functions are executed only once before and after the tree creation, while the `comptcoeff` function is executed for every target that satisfies THETA. All of these functions can access the user-specified parameters and some global parameters that include NUMPARS (number of bodies). If PNBsolver is executed with a TAYLOR parameter, it will look for a file specified as the value. If PNBsolver cannot find the file in the current directory, a file having the above three function signatures will be created. PNBsolver uses Taylor coefficients mentioned in [Krasny and Duan, 2002] if TAYLOR is not specified and it is found effective for Force and Potential calculations.

6.5.2.2 Parallel GPU Template for Direct Computation

For the GPU implementation using the direct summation, we have used two optimization techniques: 1) Tiling (partitioning of the computation domain into smaller tiles) with shared memory, and 2) Loop unrolling (replacing a loop with similar independent statements). The effect of these optimizations for Coulomb Potential is shown in Figure 6.7. In the figure, the direct version is the case when the kernel code generated was executed as a CUDA kernel without any modifications on a Tesla M2070. All kernels in the GPU template can be executed for `double` and `float` mode. In the figure, Speedup is defined as the ratio of execution time of sequential implementation

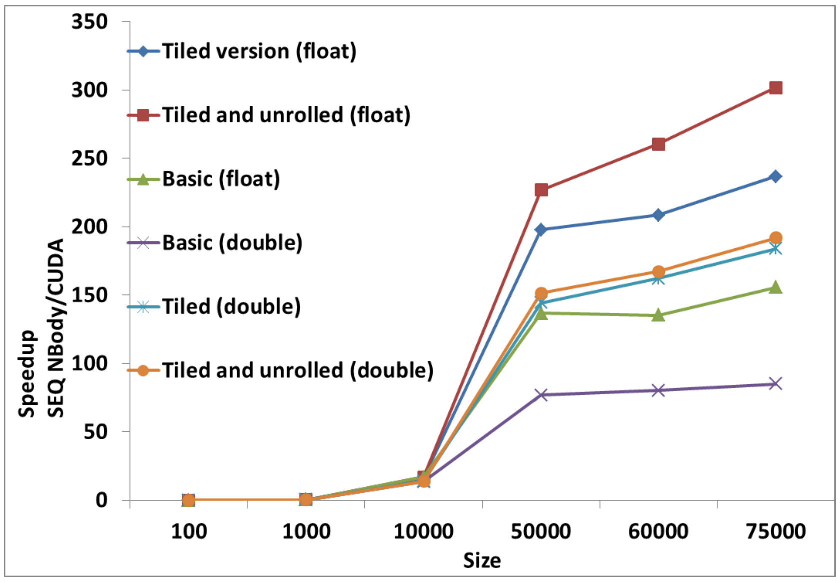


Figure 6.7: Effect of optimization techniques (Tiling and unrolling) for Coulomb potential

of Coulomb potential on a CPU to that of the CUDA implementation. Every reading is an average value of at least three of the same executions.

6.5.2.3 Parallel GPU Tree Code Template

The implementation used for GPU tree code is adapted from [Burtscher and Pingali, 2011] supports zeroth ORDER; hence, there is no need to specify Taylor coefficients for tree code implementation.

6.5.3 Modes of Operation

There are three modes of operation for PNBsolver: 1) ACCURATE, 2) AVERAGE, and 3) FAST. The execution time of the problem increases from ACCURATE to FAST and error decreases from FAST to ACCURATE. This is achieved by varying the parameters ORDER and THETA. The effect of these two parameters for two problems executed for a PNBsolver OpenMP solution for a fixed size is shown in Figures 6.8 and 6.9. As seen from Figure 6.8, as the ORDER increases, error is less but the program takes longer to finish execution. In the case of THETA, the program

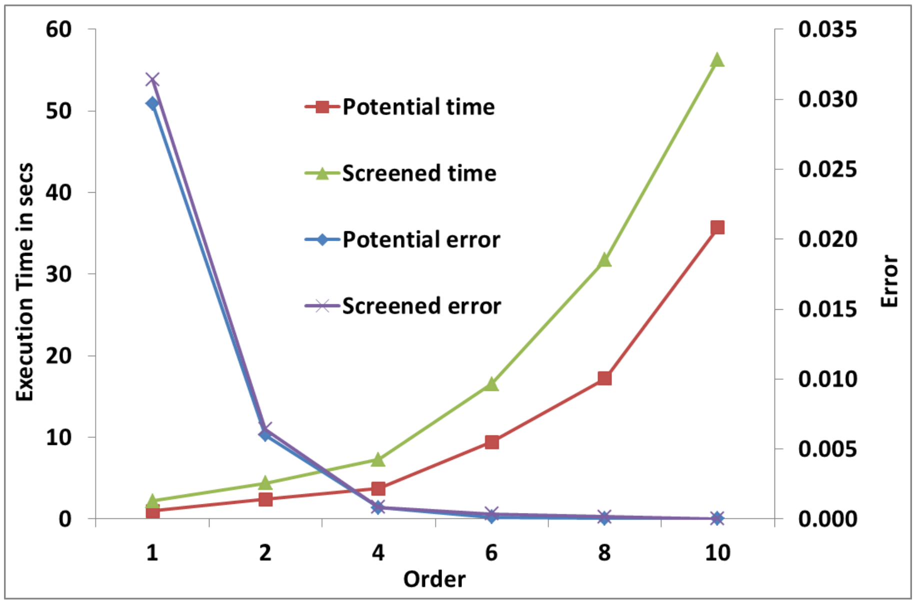


Figure 6.8: Effect of ORDER in error and time

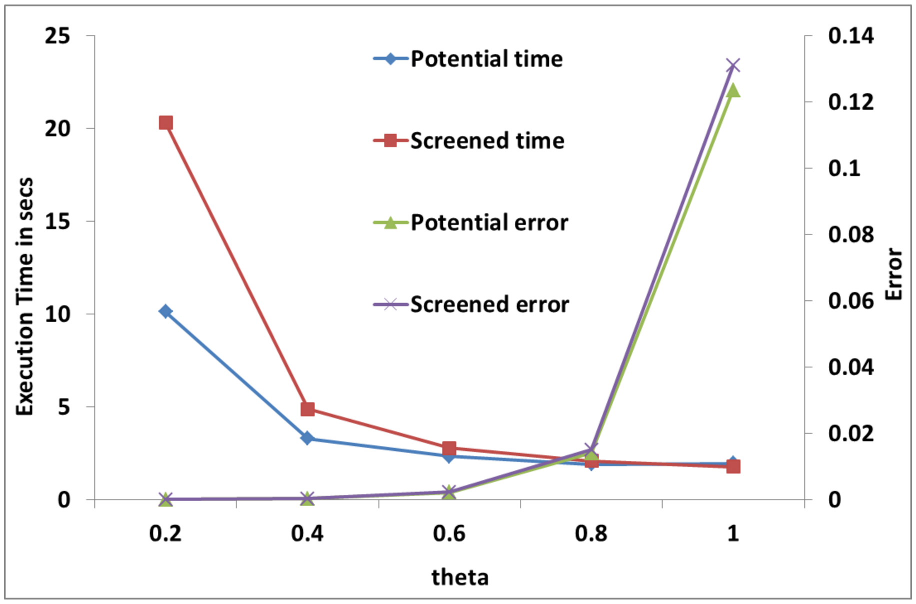


Figure 6.9: Effect of THETA in error and time

finishes faster for higher values of THETA, but has increased errors due to lowered threshold for using particle-cluster interaction. Unless specified, the THETA value is set to 0.5 as a default.

PNBsolver uses these tuning parameters to allow programmers to control the error and execution of generated programs. The parameter values set through a mode can be overridden by

Table 6.2: PNBsolver operation modes

Platform	Mode	Algorithm	Parameters
GPU (CUDA)	ACCURATE AVERAGE FAST	DIRECT DIRECT TREE	FLOAT DOUBLE

CPU (OpenMP & MPI)	ACCURATE
	AVERAGE
	FAST
		TREE
		TREE
		TREE
			ORDER=6
			ORDER=4
			ORDER=1

specifying the parameter in the `generate` clause. The value of relevant parameters for the three modes are shown in Table 6.2.

6.6 PNBsolver and Handwritten Code Comparison

In this section, we compare the execution time of two kernels generated by PNBsolver with that of handwritten code from expert programmers.

6.6.1 Yukawa Potential (Screened Coulomb Potential) Calculation

The electrostatic Yukawa energy potential [Brian Martin, 2008; Li et al., 2009] at the *i*th charge Y_i, due to the N mutually interacting charged particles with coordinates R and partial charges $Q_i(i = 1,N)$, is calculated as shown in Equation 6.6 [Brian Martin, 2008].

$$Y_i = Q_i \sum_{j=1,j!=i}^{N} \frac{Q_j e^{-|R_i - R_j|}}{|R_i - R_j|} \tag{6.6}$$

The PNBsolver representation for Yukawa interactions is shown in Figure 6.10. The code is generated for an MPI version with parameters: TAYLOR and MAXPARNODE (line 21). Since the code is generated for AVERAGE mode, from Table 6.2, ORDER=4 and default value of THETA is 0.5. It is assumed that the Taylor coefficient implementation is available in a file “comptt.c”

Comparison: We compared the execution time of MPI code generated by PNBsolver to [Li et al., 2009]. To make the comparison as accurate as possible, 1) we set the same parameter

```
kernel Yukawa in R

// Kernel declarations
  scalar Y
  scalar Q
  constant KAPPA=1.0

  read "R_1,R_2,R_3,Q", "data.dat"

  //Actual computation
  Y=Q SUM(Q*exp(-KAPPA*R)/R)

  // Write Yukawa potential to file
  write "Y", "out.dat"

endkernel

generate MPI AVERAGE Yukawa
//Parameters
[TAYLOR=comptt.c, MAXPARNODE=100].
```

Figure 6.10: Screened (Yukawa) potential kernel in space R using PNBsolver

values for the version in test, 2) used a FORTRAN template instead of the C template, and 3) MPI code generated by PNBsolver is executed with one process. The execution times are shown in Figure 6.11, where FORTRAN screened [Li et al., 2009] is the version in test and FORTRAN screened (PNBsolver) is the generated version. As seen in Figure 6.11, generated code executed nearly as fast as the sequential hand written code.

6.6.2 Gravitational Acceleration

The equation to calculate gravitational acceleration is similar to Equation 6.5 without the M_i in the outer loop. The PNBsolver code for Force is shown in Figure 6.1. We compared the PNBsolver acceleration with an acceleration implementation from the NVIDIA CUDA installation package[49]. We slightly modified the code to make more precise comparisons.

For all GPU plots, the execution time includes: 1) Time to allocate memory in the GPU, 2) Time to transfer memory from GPU to CPU, 3) GPU execution, 4) Copying results back to the CPU. We added code to pass the input values to the GPU before execution and also code to read

[49] CUDA SDK, `http://developer.nvidia.com/cuda-cc-sdk-code-samples`

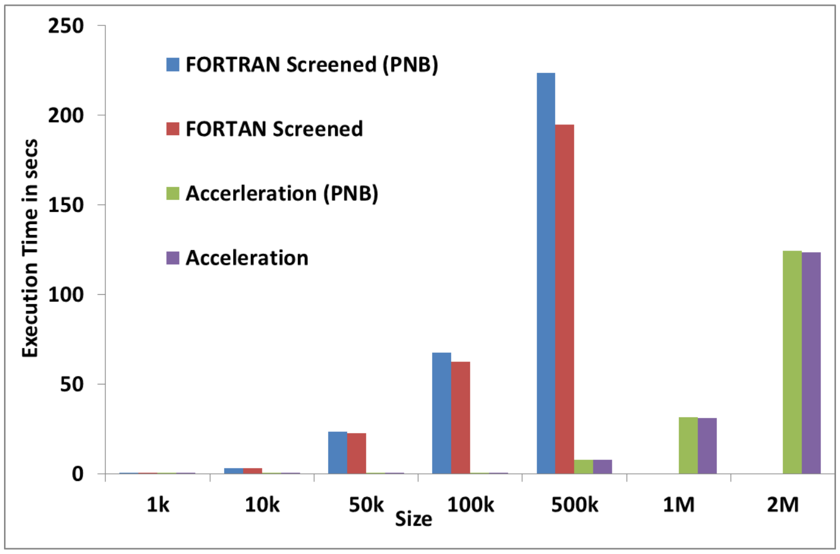


Figure 6.11: Execution time comparison of hand-written code with generated code

back after execution. However, we used the `nbody_kernel.cu` file, which performs the actual implementation for comparison.

Comparison: The execution time of the NVIDIA implementation of gravitational acceleration with the generated code of PNBsolver was compared. The number of bodies were varied from 1k to 2M. For PNBsolver, the CUDA direct summation was used with single point precision. The execution times of both of these versions is shown in Figure 6.11. As seen from the figure, PNBsolver can generate code which is as fast as the CUDA implementation.

6.7 More Nbody Problems

In this section, we explain the speedup and the errors of a few Nbody problems in three different architectures: 1) Shared memory (OpenMP), 2) Distributed memory (MPI) , and 3) GPU (CUDA). The analysis of OpenMP and MPI programs is given in the following subsection and analysis of CUDA programs is provided in subsection 6.7.2.

```
kernel Potential in R

  scalar Y, Q

  read "R_1,R_2,R_3,Q", "data.dat"

  //Actual computation
  V=Q SUM(Q/R)

  // Write Yukawa potential to file
  write "V", "out.dat"

endkernel

generate OMP AVERAGE Potential.
```

Figure 6.12: Coulomb potential kernel in space R using PNBsolver

6.7.1 Analysis of OpenMP and MPI Programs

For the OpenMP and MPI program analysis, in addition to the Force (Equation 6.5) and Yukawa potential (Equation 6.6), we generated code for the Coulomb potential which is calculated as shown in Equation 6.7 [Halliday, Resnick, and Walker, 2008]. The PNBsolver representation of the Coulomb potential is shown in Figure 6.12.

$$V_i = Q_i \sum_{j=1, j \neq i}^{N} \frac{Q_j}{|R_i - R_j|} \tag{6.7}$$

6.7.1.1 Execution Environment

The MPI and OpenMP programs were executed in a cluster with eight nodes. The programs read positions and charges from a file of 5 million records. The generated code was modified to include a direct implementation on every program to verify the accuracy of the implementation. The programs were executed for a number of bodies ranging from 5k to 500k.

6.7.1.2 Speedup of MPI and OpenMP Programs

To evaluate our parallel implementations of the tree code algorithms in OpenMP and MPI platforms, the speedup (ratio of the sequential tree code execution time to that of the parallel tree

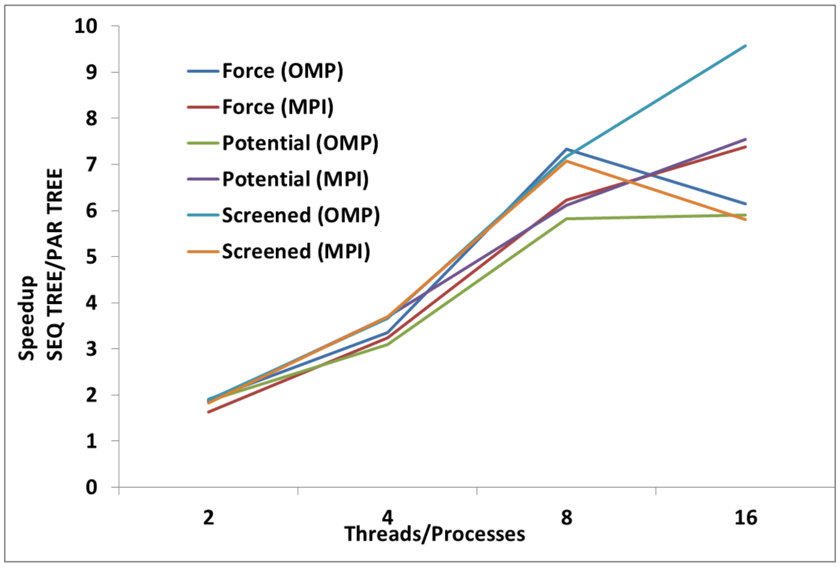


Figure 6.13: Speedup of tree code programs with sequential tree code

code), as shown in Figure 6.13, is plotted against the number of threads for OpenMP and processes for MPI programs. As shown in the graph, our parallel implementation gave a linear speedup until eight (number of nodes in the cluster) threads/processes. For more computation intensive equations, executing with more threads or processes than the available resources can slow down the program to a considerable rate as in the case of Screened Coulomb potential in Figure 6.13. In Figure 6.14, the relative speedup is plotted against the number of bodies, where relative speedup is defined as the ratio of execution time of a sequential program to that of the tree code program. In this case, super linear speedup is observed with the tree code algorithm.

6.7.2 Analysis of CUDA Programs

For CUDA analysis, the expression shown in Equation 6.8 [Lindsay and Krasny, 2001] is used for computing vortex sheet motion in 3D flow. The PNBsolver file is shown in Figure 6.15.

$$v_i = -\sum_{j=1, j\neq i}^{N} \frac{W_j}{(|R_i - R_j|^2 + \delta^2)^3} \tag{6.8}$$

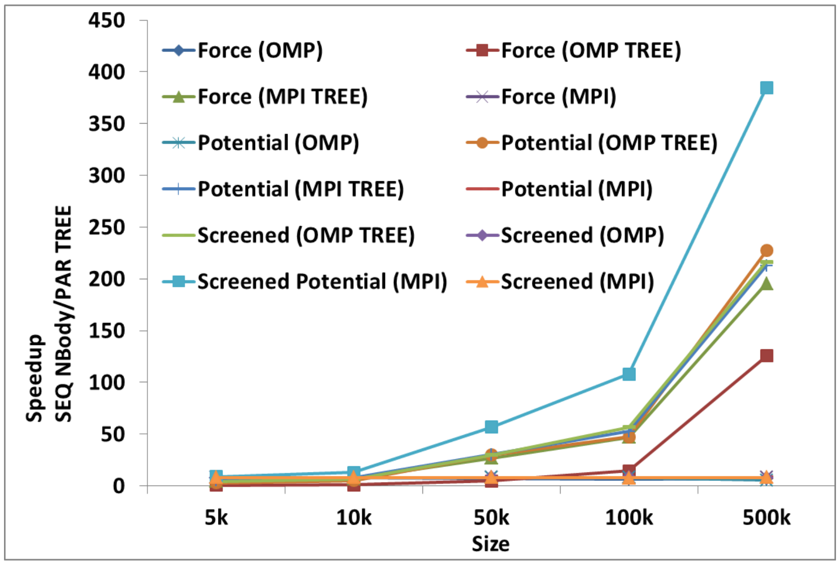


Figure 6.14: Speedup of tree code programs with sequential direct code

```
kernel Vortex in R

  scalar W
  vector V
  constant DELATSQ=0.00125

  read "R_1,R_2,R_3,W", "data.dat"

  V= SUM(W/((R_*R_+DELTASQ)(R_*R_+DELTASQ) (R_*R_+DELTASQ))

  write "V_1,V_2,V_3", "out.dat"

endkernel
generate CUDA AVERAGE Vortex.
```

Figure 6.15: Coulomb potential kernel in space R using PNBsolver

6.7.2.1 Execution Environment

We executed the generated code on a Tesla M2070 GPU card. The generated code was modified to verify the results for accuracy with the direct implementation. Similar to the OpenMP and MPI implementations, the generated code read from a file of five million records and programs were executed from a number of bodies ranging from 10K to 5M values.

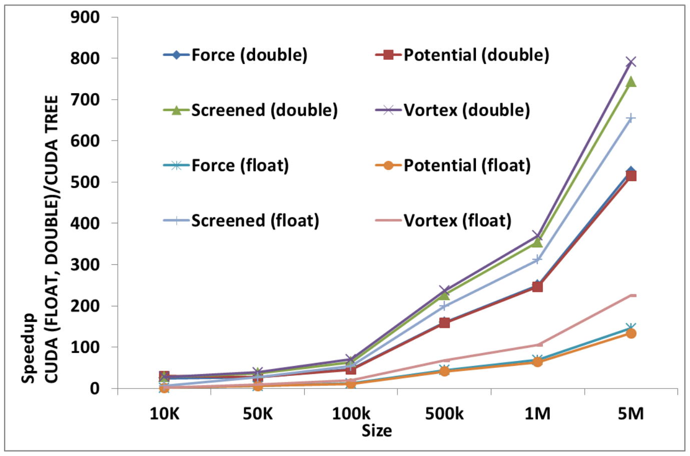


Figure 6.16: Speedup of CUDA tree code with direct CUDA

6.7.2.2 *Speedup of CUDA Programs*

The OpenMP direct implementation of the Coulomb potential for five million bodies executed with eight threads took almost two days to finish execution, while the CUDA FAST version finished in less than a few seconds. For plotting, we defined relative speedup as the ratio of execution time of CUDA direct double or float to that of CUDA tree code. The speedup for the four equations are shown in Figure 6.16 and relative errors recorded in the computation are shown in Figure 6.17. A relative error is defined as the ratio of the highest absolute error to the highest value in direct summation. As shown in Figures 6.16 and 6.17, PNBsolver can generate CUDA programs that are fast and more accurate. From Figure 6.16, it can be seen that the speedup of float is less than double. This is because the execution time of float is less compared to double, thereby the speedup with the tree code is less.

6.8 Discussion

PNBsolver has been implemented to show that our two-stage modeling approach can be applied to parallel programs. In the implementation, tree code algorithms are generated for the specified order for MPI/OpenMP programs, and zero order for CUDA programs. Extending a

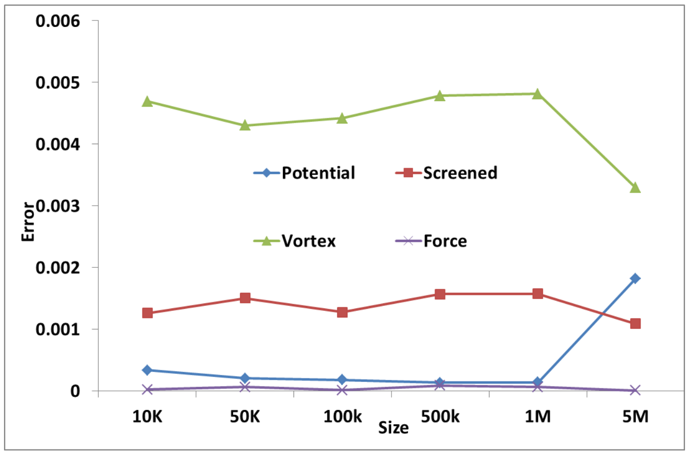


Figure 6.17: Error in CUDA tree code programs

GPU program also for the specified order is a future direction of work. Another possible direction would be adding more templates for supporting other platforms and paradigms like OpenCL.

Nbody problems involve computing interactions of multiple bodies in a 3D space. The existing solutions work on a specific problem, specific algorithm, and specific platform. In this chapter, we introduced a domain-specific language called PNBsolver, which can express the computations in an Nbody problem in a manner that is oblivious to their implementation, platform, and algorithmic details without compromising the execution time. PNBsolver can be executed in three modes: 1) FAST, 2) AVERAGE, and 3) ACCURATE modes in three popular parallel programming paradigms (CUDA, OpenMP and MPI). Modes are defined based on the target platform (e.g., ACCURATE mode on a GPU uses a double precision algorithm in CUDA, and the ACCURATE mode on a CPU uses a 10-order tree code algorithm). The speedup and errors are analyzed for four commonly seen Nbody interactions. We compared the execution time of the generated code with that of handwritten code by expert programmers to show that the execution time is not compromised. PNBsolver was applied successfully to five Nbody problems and their corresponding speedups and errors were plotted to demonstrate the performance of the approach to baseline implementations.

Chapter 7

FUTURE WORKS

This dissertation is focused on the adoption of DSLs to assist in transitioning to a new programming model and/or language. A DSL can provided a higher level of abstraction to specify what changes and where each changes must be performed, with code generators providing the required program transformation associated with mapping the higher abstraction to a specific programming model and/or language. To support program transformation, robust program transformation engines are required. In addition, optimized libraries or templates for the new programming model and/or language will be required so that the new code generated delivers the desired performance. With this intent, we developed PPModel to provide an interface for abstract hotspots in a parallel program. If programmers can express their intent, separate from the target environment, the same code can often be deployed on multiple platforms without requiring any changes to the source code generated.

From our investigation of parallel programs in different platforms, we realized that there are some very frequently used techniques in creating parallel versions for a sequential program as well as converting a parallel program from one platform to another platform. If these techniques can be identified at code-level abstraction, these techniques can provide the design and the use cases from tool support that can help programmers in creating parallel versions for a sequential program and converting from one parallel version to another. In this chapter, we explain three possible extensions of the work described in this dissertation.

7.1 Further Empirical Studies on HPC Applications

Hypothesis: *HPC applications often require systematic and repetitive code changes.*

Systematic and repetitive code changes emerge in two scenarios in the context of HPC applications: 1) while converting sequential code (C or FORTRAN) to parallel code[50], and 2) while converting a parallel implementation to another parallel implementation. In both cases, an HPC programmer follows three steps: 1) search for a popular pattern[51] that is relevant to the context and the target platform; 2) make modifications in the code to apply the selected pattern; and 3) make modifications in the code to apply the pattern.

7.1.1 Study Method/Study Artifacts

The first step of this study will be to identify the patterns and observe if a large number of parallel programs follow a very small number of patterns. Then, the edits made to apply the pattern to the context need to be identified. It is better to conduct this study separately for each parallel implementation because all the parallel patterns may not be relevant to every target platform. The latest NAS parallel benchmark suite has 12 benchmark programs available in serial, MPI, and OpenMP. This benchmark will be a convincing resource in the HPC community, as more benchmark programs can be collected for each parallel paradigm. Research on converting sequential to parallel, and parallel to other parallel implementations, can also provide an insight into the patterns and the edits required for the conversion.

7.1.2 Expected challenges

Two of the expected challenges are listed below:

[50] Parallel code or implementation refers to OpenMP (C or FORTRAN), MPI (C or FORTRAN), and GPU (CUDA or OpenCL) implementations.

[51] In our context, pattern is a code template used by parallel programmers.

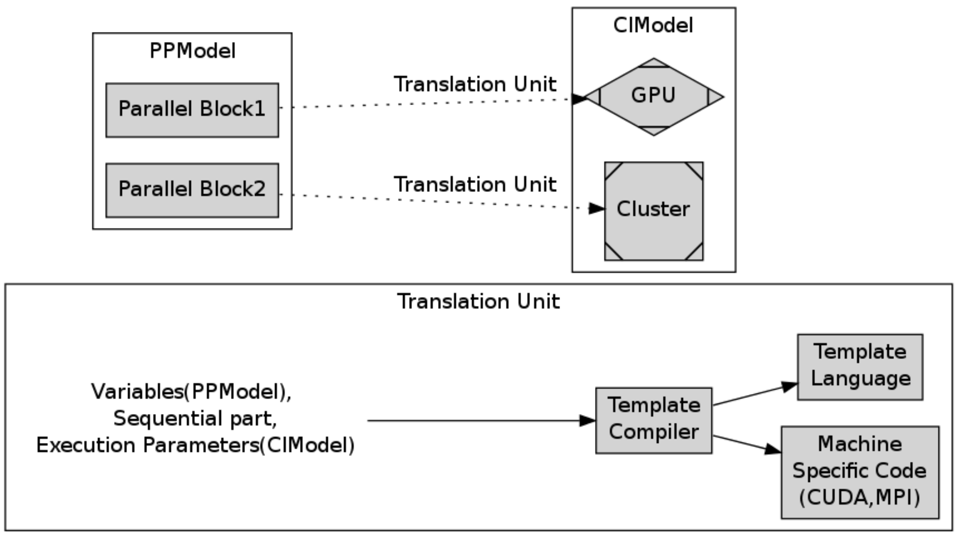


Figure 7.1: High-level design of the proposed two-model approach

- Manual analysis is necessary at least in the early stages to identify the patterns and the edits.
- Occurrences of more than one pattern in a program (pattern composition) can be different from integrating the patterns independently. Hybrid patterns, where two or more patterns for different parallel paradigms occur in a single program, offer another unique challenge. In the approach described above, the objective is to look for patterns in each platform separately. Hence, identifying hybrid patterns is a challenge.

7.2 Extending PPModel to a Two-Model Approach for HPC Programs

To address additional need for separation among the core computation and the particulars of the HPC platform, a new framework will be investigated that has two models (ClModel and PPModel) and a template language, as shown in Figure 7.1. The ClModel includes the machine-specific information for the nodes, and the PPModel includes the core computation of the program. The Translation unit converts the core computation written in the template language to machine-specific code.

7.2.1 Design of the Proposed Approach

The framework has three main components: 1) ClModel, 2) PPModel, and 3) Template language. Each of these components are explained in the following sub-sections.

7.2.1.1 Architecture (Cluster) Model (ClModel)

Using a modeling environment, a cluster can be modeled to include all of the necessary execution parameters of every node in a cluster. A model of a cluster may also include required parameters that describe the interaction between the nodes, such as distance, which measures how fast data can be transferred to a given node. Other resources that are modeled include GPUs, CPUs and memory. A new resource can be added to the modeling environment as a node type. Every resource is added to a cluster model with a set of execution parameters, which determines the execution time of a program executed on that resource. In addition to the execution parameters, any other relevant parameters can be added to the node (e.g., cache size can be linked to the node). The ClModel captures all of the machine-specific details required to generate a program that can give optimum execution time in that cluster. The logic of the program is independent of the hardware configuration and is specified separately (in PPModel, which is described in the next section).

7.2.1.2 Parallel Program Model (PPModel)

PPmodel is a modeling environment for parallel programs, which has two goals: 1) to separate the parallel sections from the sequential parts of a program, which allows a programmer to focus more on the parallelism, and 2) to define a new execution strategy for the computation intensive part of the program without changing the flow of the program. The modeling environment of PPModel is built from ClModel. Using PPmodel, the parallel part of the program can be separated from the sequential part of the program, re-designed, and then regenerated. With our approach, programmers can switch between technical solution spaces (e.g., MPI, OpenMP, TBB, CUDA and OpenCL) without actually changing the core sequential part of a program. This approach will allow a programmer to concentrate more on the essence of the parallelization, rather than focusing on the accidental complexities of language-specific details.

7.2.1.3 Template Language

We propose a bottom-up approach for modeling HPC applications. We start with a benchmark program, and then for each parallel block we try to create templates for OpenMP, TBB, MPI and CUDA architectures. The template will be in the base language with variables defined in the sequential program and also the execution parameters. In addition to the template language, implementations are available for some commonly used programs like Matrix-Vector operations, and Fast Fourier Transformsn (FFTs). Programmers should be free to readily use libraries in their program without re-implementing the entire function. While using a custom function followed by additional code, there may be a better way to perform a specific task (e.g., vector addition followed by finding the highest weight of the two vectors). There should be options to define custom-functions using built-in functions and/or other custom-functions. Such custom functions include code to optimize the interaction between the functions. Adding a new machine involves implementing the templates and defining how to merge with the sequential code.

The five steps involved in developing a parallel version of a program from its sequential version, or rewriting a parallel program for another platform using the framework, are explained in the following steps:

1. Step 1: Creating ClModel- Modeling the execution architecture: A programmer first declares

and defines the resources available and builds the ClModel. The execution parameters are set either by running some benchmark programs or by manual specification.

2. Step 2: Defining parallel sections in the program, which involves: 1) Creating a name for the parallel section, 2) Defining the input variables (i.e., variables that will be used in the parallel section), and 3) Defining the output variables (i.e., variables that will be updated during parallel execution).

3. Step 3: Creating PPModel- Modeling the parallel sections: The parallel sections and the modeled cluster are available in the modeling environment as shown in Figure 2. A programmer can link each of the parallel blocks to a resource of choice. A block not linked with any resource will execute with the default behavior (i.e., the execution is defined in the main program).

4. Step 4: Code generation: After modeling, a programmer can generate code, which creates the needed code templates to execute the program in that resource. It also creates the necessary scripts to run the program with that resource.

5. Step 5: Rewriting computation: Using the generated code templates, a programmer can express the core computation using the variables from the associated sequential section and the execution parameters of that resource.

7.2.2 Advantages of the Two-Model Approach

Some of the advantages we envision in using this approach are explained in the following sub-sections:

7.2.2.1 Optimum Performance

Because the final code executed in the nodes is tuned to a specific machine, optimum execution time can be achieved for the programs. Moreover, this approach gives the flexibility of finding the best execution environment for a program.

7.2.2.2 Code Maintenance

With this approach, only the different versions of a parallel section rather than the entire program itself, is modified, which can help in improved code maintenance and evolution.

7.2.2.3 Heterogeneous Computing

Every parallel section can be modeled to any of the available resources, which provides the advantage of executing one parallel section using one available resource (e.g., OpenMP) and another using a resource (e.g., a GPU) that can give a performance improvement.

7.2.2.4 Small parallel sections

Because the programmer has to rewrite only the parallel section, a problem-oriented approach can be followed to focus more on the problem in hand; the tool support can generate and execute the complete program.

7.2.2.5 Execution Modes

A programmer has the freedom to use different execution strategies for the same program. As an example, a PPModel can be created to execute first the parallel section in a cluster and then in a GPU, or, the PPModel can first execute the parallel section on a GPU and then in a cluster. For each execution, different "modes" can be defined (e.g., timers that are included for every parallel section in a timer mode implementation). A programmer can compare the execution time of two models without writing any code for the timers, by just changing the mode before execution.

7.2.2.6 Static Error Checking

Using the proposed approach, it can be discovered whether a node or cluster in which the program is to be executed can satisfy the execution parameters required by the program. If a node cannot execute the program, that information can be made available to the programmer during the modeling stage itself.

7.3 Human-based Studies to Evaluate Usability

The evaluations discussed in this dissertation are focused on the execution time of the programs. Further studies are required to evaluate the usability of the tools. We plan to conduct human-based empirical studies to evaluate two factors: 1) Development time and 2) Maintenance effort that affect the usability of the tool. These two future areas of assessment are explained in the following sub-sections.

7.3.1 Development Time

We plan to select a small group of test subjects who can be HPC users or HPC programmers based on the tool in test. In the training stage, selected subjects will be provided with a short tutorial explaining the features and the restrictions of the tool. In the testing stage, a problem or a part of a problem will be given to the test subjects to implement using the tool. A comparison study will be performed with a group solving the problem with the tool and another group without the tool.

7.3.2 Maintenance Effort

A very similar study will be conducted to provide an evaluation of the level of maintenance needed on the programs created using the tools discussed in this dissertation. To evaluate the effort in maintaining such programs, we have to understand how well an HPC user can comprehend a program written by another HPC user.

Chapter 8

SUMMARY

With the realization that increasing clock speed is no longer a solution, the semiconductor industry has moved towards increasing the number of cores in a chip, giving rise to multicore processors. For a multicore processor with low clock speed to outperform a single core processor with higher clock speed, software must be written in a parallel manner to take advantage of the additional processing capabilities. Before the prevalence of multicore processors, parallel programming was reserved for a small community like scientists. But now, with the introduction of multicore CPU and GPU enabled desktops, there will be more HPC users. This is due to complexities involved in parallel programming. With all the latest advances in the HPC community, parallel programming is still not an easy task for an average programmer. HPC users (scientists, software programmers, physicists, mathematicians) can have different requirements during the software development process. As an example, one HPC user may not want to compromise on execution time while another HPC user may prefer shorter software development time. There may be cases where both have the same requirements also; e.g., both developers want heterogeneous computing in the software developed. This can be achieved by defining multiple levels of abstraction on HPC programs. The overall goal of this dissertation is to show that a customized tool set for HPC users can be developed at different levels of abstraction.

Our study on parallel programs has revealed some of the common challenges faced by HPC users while designing parallel programs. In this dissertation, we identify four different levels of abstraction for HPC programs. The four different levels of abstractions are: 1) Code, 2) Algorithm, 3) Program, and 4) Sub-domain and each of these abstractions with their motivation, solution approach, and evaluation are explained in the following sections.

8.1 Code-level Modeling

Motivation: To have the optimum execution time for different sizes of problems in different platforms, current parallel programming strategies require programmers to maintain different program versions in popular parallel programming paradigms like OpenMP, MPI, CUDA, or OpenCL. In these versions, the sequential code is often duplicated and the actual computation is tangled with the sequential code. These have the potential to adversely affect program comprehension, evolution, and maintenance.

Solution: PPModel allows programmers to: 1) separate the computation intensive code sections from the program, 2) define new implementations for the pre-defined code sections, and 3) build a new program with user-defined computation code sections. With this approach, the sequential code is maintained only at one location and programmers have the flexibility to define a different execution strategy for different size problems.

Evaluation: A well-known benchmark program, Integer Sorting (IS), was selected from the NAS PBS. For the benchmark, three computation-intensive code sections were identified, two of them were executed in OpenMP and one of them in a GPU. The overall execution time of the program was reduced to one-fifth of the execution time, when compared to the sequential implementation, and to two-thirds when compared to the original benchmark OpenMP implementation.

8.2 Algorithm-level Modeling

Motivation: The programs that were implemented using a MapReduce framework (e.g., Hadoop) suffered from three kinds of problems: 1) the framework was unaware of the input format; hence, a programmer has to provide reader and writer programs to read and write different input formats; 2) there is an improper level of abstraction; the MapReduce logic in submerged inside Java classes and there is no strict separation between the computation part and the framework implementation; and 3) Lack of MapReduce validation, since Hadoop programs express MapReduce algorithms at a GPL (e.g., Java) level, they cannot handle the limitations at the MapReduce level.

Solution: MapRedoop is a framework implemented in Hadoop that combines a DSL and Eclipse Plugin that removes the three accidental complexities mentioned in the previous section. Using MapRedoop, while declaring a data structure, programmers can specify the formats to serialize or de-serialize the data structures. In MapRedoop, MapReduce validations are implemented and hence programmers will be notified as they write the program. The Eclipse plugin allows a programmer to execute the program in a commercial cloud like Amazon's EC2 or local standalone Hadoop installation.

Evaluation: Two commonly used MapReduce programs: BFS and K-means were used to evaluate the performance overhead. The analysis with popular implementations like Cloud9's BFS and Mahout's K-means, showed that MapRedoop implementations finished with comparable execution time.

8.3 Program-level Modeling

Motivation: As a part of the SDI project at PNNL, software developers were employed to implement web service wrappers for remote executable programs, so that scientists can design new scientific workflows using these remote programs. This process introduced the accidental complexity of creating boiler plate code for every web service wrapper. In addition, scientists had to be a part of the development process to provide suggestions on the input and output of web services.

Solution: SDL and WDL are two DSLs designed and implemented to help scientists design signature discovery workflows. Using SDL, scientists can generate web service wrappers for a remote executable program. Using WDL, they can orchestrate the web services to design and develop signature discovery workflow.

Evaluation: SDL was used to convert all of the remote executable programs for two workflows: 1) BLAST search, and 2) Landscape classification. WDL was used to orchestrate the web services created by SDL to design the two workflows. The web services were generated using Apache CXF and workflows were generated using Taverna.

8.4 Sub-domain-level Modeling

Motivation: Nbody solutions can be considered as a sub-domain of HPC programs, as sequential solutions of Nbody problems usually have a very high execution time and these problems are easily parallelizable. Nbody problems use direct summation algorithms, as well as approximation algorithms. For both of these algorithms, parallel versions are available in OpenMP, MPI, and CUDA. In the current practice, a mathematician or a physicist looking for an Nbody solution, 1) has to find an algorithm, 2) modify the algorithm for his specific Nbody equation, 3) parallelize the problem based on his/her resources.

Solution: Using PNBsolver, an Nbody user can specify the equation, resources, and accuracy required, and generate the code required for solving the equation. PNBsolver can generate code for a direct summation algorithm or a tree code approximate algorithm for OpenMP, MPI, and CUDA versions.

Evalution: The speedup and errors are analysed for four commonly seen Nbody interactions. The parallel tree code implementations in OpenMP and MPI with eight nodes were even 400x faster than the sequential direct implementation. The CUDA tree code implementation was 800x faster that CUDA direct summation for Force calculation. We also compared the execution time of the generated code with that of handwritten code by expert programmers to show that the execution time is not compromised.

In this dissertation, we identified four abstraction levels in HPC programs. We used software modeling techniques to provide tool support for users at these four levels of abstraction. Our research suggests that it is possible to support heterogeneous computing, reduce execution time, and improve source code maintenance.

REFERENCES

Allen, E., D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt (2007). The Fortress Language Specification. Technical Report, Sun Microsystems, Inc.

Allen, J. and K. Kennedy (1981). *PFC: A Program to Convert Fortran to Parallel Form*. Rice University, Department of Mathematical Sciences.

Allen, R. and K. Kennedy (1984, June). Automatic loop interchange. In *Proceedings of the Symposium on Compiler Construction*, Montreal, Canada, pp. 233–246.

Allen, R. and K. Kennedy (1987). Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), 491–542.

Almasi, G. S. and A. Gottlieb (1989). *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc.

Amdahl, G. M. (1967, April). Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, Atlantic City, NJ, pp. 483–485.

Amza, C., A. L. Cox, H. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel (1996). Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2), 18–28.

Angel, E. (1999). *Interactive Computer Graphics* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc.

Ansari, S. (2012). Elucidation tree code. `http://www.prism.gatech.edu/~gth716h/BNtree/`. [Online; accessed 19-January-2013].

Antoniol, G., U. Villano, E. Merlo, and M. Penta (2002). Analyzing cloning evolution in the Linux kernel. *Information and Software Technology*, 44(13), 755–765.

Appelbe, W. F., K. Smith, and C. E. McDowell (1989). Start/Pat: A parallel-programming toolkit. *IEEE Software*, 6(4), 29–38.

Arora, R., P. Bangalore, and M. Mernik (2011). A technique for non-invasive application-level checkpointing. *Journal of Supercomputing*, 57(3), 227–255.

Arora, R., P. Bangalore, and M. Mernik (2012). Raising the level of abstraction for developing message passing applications. *The Journal of Supercomputing*, 59(2), 1079–1100.

Artigas, P. V., M. Gupta, S. P. Midkiff, and J. E. Moreira (2000, May). Automatic loop transformations and parallelization for Java. In *Proceedings of the International Conference on Supercomputing*, Santa Fe, NM, pp. 1–10.

Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick (2006, Dec). The landscape of parallel computing research: A view from berkeley. Technical Report, EECS Department, University of California, Berkeley.

Atkinson, C. and T. Kuhne (2003). Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5), 36–41.

Baczewski, A. D. and B. Shanker (2011). An O(n) method for rapidly computing periodic potentials using accelerated cartesian expansions. `http://arxiv.org/abs/1107.3069`. [Online; accessed 19-January-2013].

Baker, N. (2012). `http://signatures.pnnl.gov/`. [Online; accessed 19-January-2013].

Ball, J. R., R. C. Bollinger, T. A. Jeeves, R. C. McReynolds, and D. H. Shaffer (1962, December). On the use of the SOLOMON parallel-processing computer. In *Proceedings of the Fall Joint Computer Conference*, Philadelphia, PA, pp. 137–146.

Barnes, G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes (1968). The ILLIAC IV computer. *IEEE Transactions on Computers*, 17(8), 746–757.

Barnes, J. (2012). Barnes Treecode. `http://www.ifa.hawaii.edu/~barnes/treecode/`. [Online; accessed 19-January-2013].

Barnes, J. and P. Hut (1986, December). A hierarchical O(N log N) force-calculation algorithm. *Nature*, *324*, 446–449.

Basili, V. R., J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz (2008, July). Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Software*, 25(4), 29–36.

Basumallik, A. and R. Eigenmann (2005, June). Towards automatic translation of OpenMP to MPI. In *Proceedings of the International Conference on Supercomputing*, Cambridge, MA, pp. 189–198.

Baumstark Jr, L. and L. Wills (2002, October). Exposing data-level parallelism in sequential image processing algorithms. In *Proceedings of the Working Conference on Reverse Engineering*, Richmond, VA, pp. 245–254.

Bédorf, J., E. Gaburov, and S. P. Zwart (2012). Bonsai: A GPU Tree-Code. `http://arxiv.org/abs/1204.2280`. [Online; accessed 19-January-2013].

Belleman, R. G., J. Bédorf, and S. F. Portegies Zwart (2008). High performance direct gravitational Nbody simulations on graphics processing units ii: An implementation in CUDA. *New Astronomy*, 13(2), 103–112.

Bethe, H. A. (1956, September). Nuclear many-body problem. *Physical Review*, 103(5), 1353–1390.

Bilbao, J., G. Garate, A. Olozaga, and A. del Portillo (2005, July). Easy clustering with openmosix. In *Proceedings of the 9th WSEAS International Conference on Computers*, Athens, Greece, pp. 79:1–79:6.

Bloch, I., J. Dalibard, and W. Zwerger (2008, July). Many-body physics with ultracold gases. *Rev. Mod. Phys.*, 80(3), 885–964.

Bouhamidi, A. and K. Jbilou (2008, August). Meshless thin plate spline methods for the modified helmholtz equation. *Computer Methods in Applied Mechanics and Engineering*, 197(45–48), 3733–3741.

Bräunl, T. (2000). Parallaxis-iii: Architecture-independent data parallel processing. *IEEE Transactions on Software Engineering*, 26(3), 227–243.

Breitbart, J. (2009, May). CuPP - A framework for easy CUDA integration. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Rome, Italy, pp. 1–8.

Brian Martin, G. S. (2008). *Particle Physics* (3rd ed.). John Wiley & Sons.

Brodsky, S. J., H.-C. Pauli, and S. S. Pinsky (1998, August). Quantum chromodynamics and other field theories on the light cone. *Physics Reports*, 301(4–6), 299–486.

Brodtkorb, A. R. and T. R. Hagen (2008, July). A comparison of three commodity-level parallel architectures: Multi-core CPU, Cell BE and GPU. In *Proceedings of the International Conference on Mathematical Methods for Curves and Surfaces*, Tønsberg, Norway, pp. 70–80.

Browne, J., M. Azam, and S. Sobek (1989). CODE: A unified approach to parallel programming. *IEEE Software*, 6(4), 10–18.

Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan (2004). Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3), 777–786.

Burtscher, M. and K. Pingali (2011). An efficient CUDA implementation of the tree-based Barnes Hut Nbody algorithm. In *GPU Computing Gems Emerald Edition*, Chapter 6, pp. 75–92. Elsevier Inc.

Carlson, J., J. Kogut, and V. R. Pandharipande (1983, January). Quark model for baryons based on quantum chromodynamics. *Physical Review D*, 27(1), 233–243.

Carver, J. (2011). Development of a mesh generation code with a graphical front-end: A case study. *Journal of Organizational and End-User Computing*, 23(4), 1–16.

Carver, J., R. Kendall, S. Squires, and D. Post (2007, May). Software development environments for scientific and engineering software: A series of case studies. In *Proceeding of the International Conference on Software Engineering*, Minneapolis, MN, pp. 550–559.

Chafi, H., A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun (2011, February). A domain-specific approach to heterogeneous parallelism. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, pp. 35–46.

Chandra, R., L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon (2001). *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc.

Chapman, B., G. Jost, and R. v. d. Pas (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

Charles, P., C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar (2005, October). X10: An object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10), 519–538.

Cheng, H., J. Huang, and T. J. Leiterman (2006, January). An adaptive fast solver for the modified Helmholtz equation in two dimensions. *Journal of Computational Physics*, 211(2), 616–637.

Cumberlidge, M. (2007). *Business Process Management with JBoss jBPM : A Practical Guide for Business Analysis-Develop Business Process Models for Implementation in a Business Process Management System* (First ed.). PACKT Publishing.

Czarnecki, K. and U. Eisenecker (2000). *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co.

Darema, F. (2001, September). The SPMD model: Past, present and future. In *Proceedings of the European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Santorini/Thera, Greece, pp. 1–1.

Darema, F., D. A. George, V. A. Norton, and G. F. Pfister (1988). A single-program-multiple-data computational model for EPEX/Fortran. *Parallel Computing*, 7(1), 11–24.

Darlington, J., A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu (1993, June). Parallel programming using skeleton functions. In *Proceedings of the International Conference on Parallel Architectures and Languages Europe*, Munich, Germany, pp. 146–160.

de Kergommeaux, J. C. and P. Codognet (1994, September). Parallel logic programming systems. *ACM Computing Surveys*, 26(3), 295–336.

Dean, J. and S. Ghemawat (2004, December). MapReduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on on Operating Systems Design & Implementation*, San Francisco, CA, pp. 137–150.

Dean, J. and S. Ghemawat (2008, January). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107–113.

Deimel, J. and E. Lionel (1985, June). The uses of program reading. *ACM SIGCSE Bulletin*, 17(2), 5–14.

Dekeyser, J.-L., D. Lazure, and P. Marquet (1994). A geometrical data-parallel language. *SIGPLAN Notes*, 29(4), 31–40.

Deshpande, A. and M. Schultz (1992). Efficient parallel programming with Linda. *Scientific Programming*, 1(2), 177–183.

Devin, F., P. Boulet, J.-L. Dekeyser, and P. Marquet (2002, September). GASPARD: A visual parallel programming environment. In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, Warsaw, Poland, pp. 145.

DeVito, Z., N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan (2011, November). Liszt: A domain-specific language for building portable mesh-based pde solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WA, pp. 9:1–9:12.

Di Martino, B. and C. W. Keβler (2000). Two program comprehension tools for automatic parallelization. *IEEE Concurrency*, 8(1), 37–47.

Diaconescu, R. and H. Zima (2007, August). An approach to data distributions in chapel. *International Journal of High Performance Computing Applications*, 21(3), 313–335.

Diacu, F. (1996, November). The solution of the nbody problem. *The Mathematical Intelligencer*, 18(3), 66–70.

Dig, D., J. Marrero, and M. D. Ernst (2009, May). Refactoring sequential Java code for concurrency via concurrent libraries. In *Proceedings of the International Conference on Software Engineering*, Vancouver, Canada, pp. 397–407.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Eigenmann, R., J. Hoeflinger, Z. Li, and D. A. Padua (1992, August). Experience in the automatic parallelization of four perfect-benchmark programs. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, pp. 65–83.

Erl, T. (2009). *SOA Design Patterns* (First ed.). Prentice Hall PTR.

Eskicioglu, M. R., T. A. Marsland, W. Hu, and W. Shi (1999, January). Evaluation of the JIAJIA software DSM system on high performance computer architectures. In *Proceedings of the Hawaii International Conference on System Sciences*, Maui, HI, pp. 8012–8022.

Espasa, R. and M. Valero (1997). Exploiting instruction- and data-level parallelism. *IEEE Micro*, 17(5), 20–27.

Filipponi, A., A. Di Cicco, and C. R. Natoli (1995, December). X-ray-absorption spectroscopy and *n* -body distribution functions in condensed matter. i. theory. *Physical Review B*, 52(21), 15122–15134.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9), 948–960.

Fowler, M. (2005). `http://www.martinfowler.com/articles/languageWorkbench.html`. [Online; accessed 19-January-2013].

Friedrichs, M. S., P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande (2009, February). Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry*, 30(6), 864–872.

Fritz, N., P. Lucas, and P. Slusallek (2004, November). CGiS, a new language for data-parallel GPU programming. In *Proceedings of the International Workshop Vision, Modeling, and Visualization*, Stanford, CA, pp. 241–248.

Garrido, A. and R. Johnson (2002, May). Challenges of refactoring C programs. In *Proceedings of the International Workshop on Principles of Software Evolution*, Orlando, FL, pp. 6–14.

Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), 80–112.

Gellerich, W. and M. M. Gutzmann (1996, September). Massively parallel programming languages – a classification of design approaches. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, Philadelphia, PA, pp. 110–118.

Ghemawat, S., H. Gobioff, and S.-T. Leung (2003, October). The Google file system. In *Proceedings of the Symposium on Operating systems principles*, Bolton Landing, NY, pp. 29–43.

Gill, S. (1958). Parallel programming. *The Computer Journal*, 1(1), 2–10.

Gray, J., J.-P. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle (2007). Domain-specific modeling. In *Handbook of dynamic system modeling*, Chapter 7, pp. 7:1–7:20. CRC Press.

Gropp, W., E. Lusk, and A. Skjellum (1994). *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press.

Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM*, 31(5), 532–533.

Halliday, D., R. Resnick, and J. Walker (2008). *Fundamentals of Physics* (8th ed.). Wiley India Pvt. Limited.

Han, T. D. and T. S. Abdelrahman (2009, March). hiCUDA: A high-level directive-based language for GPU programming. In *Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units*, Washington, D.C., pp. 52–61.

Hannay, J. E., C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson (2009, May). How do scientists develop and use scientific software? In *Proceedings of the Workshop on Software Engineering for Computational Science and Engineering*, Vancouver, Canada, pp. 1–8.

He, B., W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang (2008, October). Mars: A MapReduce framework on graphics processors. In *Proceedings of the Innternational Conference on Parallel Architectures and Compilation Techniques*, Toronto, Ontario, Canada, pp. 260–269.

Heroux, M. A., R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley (2005). An overview of the Trilinos project. *ACM Transactions on Mathematical Software*, 31(3), 397–423.

Huang, J., J. Jia, and B. Zhang (2009, November). Fmm-yukawa: An adaptive fast multipole method for screened coulomb interactions. *Computer Physics Communications*, 180(11), 2331–2338.

Huang, K. and C. N. Yang (1957, February). Quantum-mechanical many-body problem with hard-sphere interaction. *Physical Review*, 105(3), 767–775.

Jablonski, P. and D. Hou (2007, October). CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*, Montreal, Canada, pp. 16–20.

Jacob, F. (2012). `http://sites.google.com/site/tppmodel`. [Online; accessed 19-January-2013].

Jacob, F., R. Arora, P. Bangalore, M. Mernik, and J. Gray (2010, July). Raising the level of abstraction of GPU-programming. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, pp. 339–345.

Jacob, F., J. Gray, P. Bangalore, and M. Mernik (2010, December). Refining high performance Fortran code from programming model dependencies. In *International Conference on High Performance Computing Student Research Symposium*, Goa, India, pp. 1–6.

Jacob, F., J. Gray, J. C. Carver, M. Mernik, and P. Bangalore (2012). PPModel: A modeling tool for source code maintenance and optimization of parallel programs. *The Journal of Supercomputing*, 62(3), 1560–1582.

Jacob, F., Y. Sun, J. Gray, and P. Bangalore (2011, March). A platform-independent tool for modeling parallel programs. In *Proceedings of the ACM Southeast Regional Conference*, Kennesaw, GA, pp. 138–143.

Jacob, F., A. Wagner, P. Bahri, S. Vrbsky, and J. Gray (2011). Simplifying the development and deployment of MapReduce algorithms. *Journal of Next-Generation Computing*, 2(2), 123–142.

Jacob, F., D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray (2010, December). CUDACL : A tool for CUDA and OpenCL programmers. In *Proceedings of the International Conference of High Performance Computing*, Goa, India, pp. 1–11.

Jacob, F., A. Wynne, Y. Liu, N. Baker, and J. Gray (2012a, October). Domain-specific languages for composing signature discovery workflows. In *Proceedings of the Workshop on Domain-Specific Modeling*, Tucson, AZ, pp. 81–83.

Jastrow, R. (1955, June). Many-body problem with strong forces. *Physical Review*, 98(5), 1479–1484.

Jiménez, M., F. Rosique, P. Sánchez, B. Álvarez, and A. Iborra (2009). Habitation: A domain-specific language for home automation. *IEEE Software*, 26(4), 30–38.

Johnston, H. (2012). Johnstone tree code. `https://www.math.umass.edu/~johnston/newtreecode.html`. [Online; accessed 19-January-2013].

Jouault, F., J. Bézivin, C. Consel, I. Kurtev, and F. Latry (2006, July). Building DSLs with AMMA/ATL: A case study on SPL and CPL telephony languages. In *Proceedings of the Workshop on Domain-Specific Program Development*, Nantes, France, pp. 4.

Kalé, L., R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten (1999, May). Namd2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151(1), 283–312.

Kamiya, T., S. Kusumoto, and K. Inoue (2002). CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7), 654–670.

Kanungo, T., D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu (2002, July). An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 24(7), 881–892.

Karp, A. H. and R. G. Babb II (1988). A comparison of 12 parallel FORTRAN dialects. *IEEE Software*, 5(5), 52–67.

Karp, A. H. and H. P. Flatt (1990). Measuring parallel processor performance. *Communications of the ACM*, 33(5), 539–543.

Kelly, D. F. (2007, November). A Software chasm: Software engineering and scientific computing. *IEEE Software*, 24(6), 120–119.

Kendall, R., J. Carver, D. Fisher, D. Henderson, A. Mark, D. Post, C. Rhoades, and S. Squires (2008). Development of a weather forecasting code: A case study. *IEEE Software*, 25(4), 59–65.

Kernighan, B. W. and P. J. Plauger (1982). *The Elements of Programming Style* (2nd ed.). McGraw-Hill, Inc.

Kessler, C. W. (1995, January). Pattern-driven automatic program transformation and parallelization. In *Proceedings of the Workshop on Parallel and Distributed Processing*, San Remo, Italy, pp. 76.

Klepeis, J. L., K. Lindorff-Larsen, R. O. Dror, and D. E. Shaw (2009, Apr). Long-timescale molecular dynamics simulations of protein structure and function. *Current Opinion in Structural Biology*, 19(2), 120–127.

Krasny, R. and Z.-H. Duan (2002). Treecode algorithms for computing nonbonded particle interactions. In *Computational Methods for Macromolecules: Challenges and Applications*, pp. 359–380.

Krasny, R. and L. Wang (2011, September). Fast evaluation of multiquadric RBF sums by a cartesian Treecode. *SIAM Journal on Scientific Computing*, 33(5), 2341–2355.

Kromer, P. (2009). Wukong: Hadoop made so easy. `http://mrflip.github.com/wukong/`. [Online; accessed 19-January-2013].

Lédeczi, A., A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai (2001). Composing domain-specific design environments. *IEEE Computer*, 34(11), 44–51.

Lee, S., S.-J. Min, and R. Eigenmann (2009, February). OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proceedings of the International Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, pp. 101–110.

Lewin, M. (2004, February). Solutions of the multiconfiguration equations in quantum chemistry. *Archive for Rational Mechanics and Analysis*, 171(1), 83–114.

Li, P., H. Johnston, and R. Krasny (2009, June). A cartesian treecode for screened coulomb interactions. *Journal of Computational Physics*, 228(10), 3858–3868.

Lin, J. and C. Dyer (2010). *Data-Intensive Text Processing with MapReduce*. Synthesis lectures on human language technologies. Morgan & Claypool Publishers.

Lindsay, K. and R. Krasny (2001, September). A particle method and adaptive treecode for vortex sheet motion in three-dimensional flow. *Journal of Computational Physics*, 172(2), 879–907.

Liu, D., Z.-H. Duan, R. Krasny, and J. Zhu (2004, April). Parallel implementation of the Treecode Ewald method. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Santa Fe, NM, pp. 236a.

Lo, J. L., J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers (1997). Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3), 322–354.

Lottiaux, R., P. Gallard, G. Vallee, C. Morin, and B. Boissinot (2005, May). OpenMosix, OpenSSI and Kerrighed: A comparative study. In *Proceedings of the International Symposium on Cluster Computing and the Grid*, Cardiff, UK, pp. 1016–1023.

Lovelace, A. (1843). Notes to the translation of sketch of the analytical engine invented by Charles Babbage Esq. by L. F. menabrea, of Turin, officer of the military engineers. *Scientific Memoirs, Selections from The Transactions of Foreign Academies and Learned Societies and from Foreign Journals*, 3(XXIX).

Low, Y., J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein (2010). Graphlab: A new framework for parallel machine learning. *Clinical Orthopaedics and Related Research*, *abs/1006.4990.*

Luksch, P., U. Maier, S. Rathmayer, M. Weidmann, and F. Unger (1997, July). Sempa: Software engineering for parallel scientific computing. *IEEE Parallel & Distributed Technology: Systems & Technology*, 5(3), 64–72.

Makino, J. and M. Taiji (1998). *Scientific Simulations with Special-Purpose Computers - the GRAPE Systems* (1st ed.). John Wiley.

Manjunatha, A., P. Anderson, A. Ranabahu, and A. Sheth (2011, March). Identifying and implementing the underlying operators for nuclear magnetic resonance based metabolomics data analysis. In *Proceedings of the International Conference on Bioinformatics and Computational Biology*, New Orleans, LA, pp. 205–209.

Margery, D., G. Vallee, R. Lottiaux, C. Morin, and J. yves Berthou (2003, September). Kerrighed: A SSI cluster OS running OpenMP. In *Proceedings of the European Workshop on OpenMP*, Aachen, Germany.

Mark, W. R., R. S. Glanville, K. Akeley, and M. J. Kilgard (2003). Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3), 896–907.

Martino, B. D. and G. Iannello (1996, March). PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *Proceedings of the International Workshop on Program Comprehension*, Berlin, Germany, pp. 164.

Mathe, J. L., J. Martin, P. Miller, A. Ledeczi, L. Weavind, A. Nadas, A. Miller, D. Maron, and J. Sztipanovits (2009). A model integrated guideline-driven, clinical decision-support system. *IEEE Software*, 26(4), 54–61.

Maximilien, E. M., H. Wilkinson, N. Desai, and S. Tai (2007, September). A domain-specific language for web apis and services mashups. In *Proceedings of the International Conference on Service-Oriented Computing*, Vienna, Austria, pp. 13–26.

McCool, M. and S. D. Toit (2004). *Metaprogramming GPUs with Sh*. AK Peters Ltd.

McCullough, M. (2011). Cascading through hadoop: A dsl for simpler MapReduce. `http://www.nofluffjuststuff.com/conference/reston/2011/04/session?id=20942`. [Online; accessed 19-January-2013].

McKenney, P. E. (2010). Is parallel programming hard, and, if so, what can you do about it? `http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`. [Online; accessed 19-January-2013].

Menabrea, L. F. (1842, September). Sketch of the analytic engine invented by Charles Babbage. `http://www.fourmilab.ch/babbage/sketch.html`. [Online; accessed 19-January-2013].

Mernik, M., J. Heering, and A. M. Sloane (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4), 316–344.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Inc.

Mollick, E. (2006). Establishing Moore's law. *IEEE Annals of the History of Computing*, 28(3), 62–75.

Moore, G. E. (2000). Cramming more components onto integrated circuits. In *Readings in Computer Architecture*, pp. 56–59. Morgan Kaufmann Publishers Inc.

Munshi, A., B. Gaster, T. Mattson, J. Fung, and D. Ginsburg (2011). *OpenCL Programming Guide*. OpenGL Series. Addison-Wesley.

NVIDIA (2007). Nvidia CUDA compute unified device architecture - Programming guide. `http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`. [Online; accessed 19-January-2013].

Ossmani, M. E. and P. Poncet (2010, September). Efficiency of multiscale hybrid grid-particle vortex methods. *Multiscale modeling & simulation*, 8(5), 1671–1690.

Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell (2007, March). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1), 80–113.

Page, L., S. Brin, R. Motwani, and T. Winograd (1998). The PageRank citation ranking: Bringing order to the web. `http://ilpubs.stanford.edu:8090/422/`. [Online; accessed 19-January-2013].

Palem, K. and A. Lingamneni (2012, June). What to do about the end of Moore's law, probably! In *Proceedings of the 49th Annual Design Automation Conference*, San Francisco, California, pp. 924–929.

Parnas, D. L. and P. C. Clements (1986, February). A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2), 251–257.

Parr, T. J. (2004, May). Enforcing strict model-view separation in template engines. In *Proceedings of the International Conference on World Wide Web*, Manhattan, NY, pp. 224–233.

Pearlman, D. A., J. W. Case, David A. Caldwell, W. S. Ross, T. E. Cheatham, S. DeBolt, D. Ferguson, G. Seibel, and P. Kollman (1995, September). AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules. *Computer Physics Communications*, 91(1–3), 1–41.

Pennycook, S. J., S. D. Hammond, S. A. Jarvis, and G. R. Mudalige (2011). Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark. *SIGMETRICS Performance Evaluation Review*, 38(4), 23–29.

Perrott, R. H. (1981, June). Language design approaches for parallel processors. In *Proceedings of the Conference on Analysing Problem Classes and Programming*, Nurnberg, Germany, pp. 115–126.

Perrott, R. H. (1987). *Parallel Programming*. Addison-Wesley Longman Publishing Co., Inc.

Phillips, J. C., R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten (2005). Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16), 1781–1802.

Pike, R., S. Dorward, R. Griesemer, and S. Quinlan (2005a). Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 277–298.

Pike, R., S. Dorward, R. Griesemer, and S. Quinlan (2005b, October). Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4), 277–298.

Plimpton, S. (1995, March). Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1), 1–19.

Pruett, M. (2007). *Yahoo! pipes* (First ed.). O'Reilly.

Püschel, M., J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo (2005). SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation*, 93(2), 232–275.

Raja, A. and D. Lakshmanan (2010, February). Article: Domain specific languages. *International Journal of Computer Applications*, 1(21), 99–105.

Rakić, P. S., D. D. Milašinović, Z. Zivanov, Z. Suvajdin, M. Nikolić, and M. Hajduković (2011). MPI-CUDA parallelization of a finite-strip program for geometric nonlinear analysis: A hybrid approach. *Advances in Engineering Software*, 42(5), 273–285.

Ranabahu, A., A. Sheth, A. Manjunatha, and K. Thirunarayan (2010, October). Towards cloud mobile hybrid application generation using semantically enriched domain specific languages. In *International Workshop on Mobile Computing and Clouds*, Santa Clara, CA, pp. 349–360.

Ranger, C., R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis (2007, February). Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Phoenix, AZ, pp. 13–24.

Reddaway, S. F. (1973). DAP-a distributed array processor. *ACM SIGARCH Computer Architecture News*, 2(4), 61–65.

Rendleman, C. A., V. E. Beckner, M. Lijewski, W. Crutchfield, and J. B. Bell (2000, August). Parallelization of structured, hierarchical adaptive mesh refinement algorithms. *Computing and Visualization in Science*, 3(3), 147–157.

Reynolds, T. J., A. J. Beaumont, A. S. K. Cheng, S. A. Delgado-Rannauro, and L. A. Spacek (1988). Brave:A parallel logic language for artificial intelligence. *Future Generation Computer Systems*, 4(1), 69–75.

Roy, S. and V. Chaudhary (1998, July). Strings: A high-performance distributed shared memory for symmetrical multiprocessor clusters. In *Proceedings of the International Symposium on High Performance Distributed Computing*, Chicago, IL, pp. 90.

Sagui, C. and T. A. Darden (1999, June). Molecular dynamics simulations of biomolecules: Long-range electrostatic effects. *Annual Review of Biophysics and Biomolecular Structure*, 28(1), 155–179.

Salmon, J. K. (1994, June). Fast parallel tree codes for gravitational and fluid dynamical n-body problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(2), 129–133.

Salmon, J. K. and M. S. Warren (1994). Skeletons from the Treecode closet. *Journal of Computational Physics*, *111*, 136–155.

Sanders, R. and D. Kelly (2008). Dealing with risk in scientific software development. *IEEE Software*, 25(4), 18–20.

Sapirstein, J. (1998, January). Theoretical methods for the relativistic atomic many-body problem. *Reviews of Modern Physics*, 70(1), 55–76.

Sasai, M. and P. G. Wolynes (2003, March). Stochastic gene expression as a many-body problem. *Proceedings of the National Academy of Science, PNAS*, 100(5), 2374–2379.

Schaller, R. R. (1997). Moore's law: Past, present, and future. *IEEE Spectrum*, 34(6), 52–59.

Schlansker, M., T. M. Conte, J. Dehnert, K. Ebcioglu, J. Z. Fang, and C. L. Thompson (1997). Compilers for instruction-level parallelism. *IEEE Computer*, 30(12), 63–69.

Schmidt, D. (2006). Model-driven engineering. *IEEE Computer*, 39(2), 25–32.

Segal, J. (2007, September). Some problems of professional end-user developers. In *Proceedings of the IEEE Symposium on Visual Languages and HumanCentric Computing*, Coeur dÁléne, ID, pp. 111–118.

Segal, J. and C. Morris (2008). Developing scientific software. *IEEE Software*, 25(4), 18–20.

Shull, F., J. Carver, L. Hochstein, and V. R. Basili (2005, November). Empirical study design in the area of high-performance computing (hpc). In *Proceedings of the International Symposium on Empirical Software Engineering*, Noosa Heads, Australia, pp. 305–314.

Slotnick, D. L., W. C. Borck, and R. C. McReynolds (1962, December). The SOLOMON computer. In *Proceedings of the Fall Joint Computer Conference*, Philadelphia, PA, pp. 97–107.

Sprinkle, J., M. Mernik, J.-P. Tolvanen, and D. Spinellis (2009). Guest Editors' Introduction: What kinds of nails need a Domain-Specific hammer? *IEEE Software*, 26(4), 15–18.

Squyres, J. M. (2003). Definitions and fundamentals – The Message Passing Interface (MPI). *ClusterWorld Magazine, MPI Mechanic Column*, 1(1), 26–29.

Stenström, P., E. Hagersten, D. J. Lilja, M. Martonosi, and M. Venugopal (1997). Trends in shared memory multiprocessing. *IEEE Computer*, 30(12), 44–50.

Stone, J. E., D. Gohara, and G. Shi (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Design and Test*, 12(3), 66–73.

Stone, J. E., J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten (2007, September). Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16), 2618–2640.

Sugiki, A., K. Kato, Y. Ishii, H. Taniguchi, and N. Hirooka (2010, December). Kumoi: A high-level scripting environment for collective virtual machines. In *International Conference on Parallel and Distributed Systems*, Volume 0, Shanghai, China, pp. 322–329.

Sujeeth, A. K., H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun (2011, June). OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the International Conference on Machine Learning*, Haifa, Israel, pp. 609–616.

Sun, Y., Z. Demirezen, F. Jouault, R. Tairas, and J. Gray (2008, September). A model engineering approach to tool interoperability. In *Proceedings of the International Conference on Software Language Engineering, Tool Demonstration*, Toulouse, France, pp. 178–187.

Sunderam, V. S. (1990). Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), 315–339.

Szymanski, B. K. and D. Mueller-Wichards (1987). Parallel programming with recurrent equations. *International Journal of Supercomputer Applications and High Performance Engineering*, 1(2), 44–74.

Taylor, I. J., E. Deelman, D. B. Gannon, and M. Shields (2006). *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag Inc.

Tullsen, D. M., S. J. Eggers, and H. M. Levy (1998, June). Simultaneous multithreading: Maximizing on-chip parallelism. In *25 Years of the International Symposia on Computer architecture (selected papers)*, Barcelona, Spain, pp. 533–544.

Turing, A. M. (1936). On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42), 230–265.

Ueng, S.-Z., M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu (2008, July). CUDA-Lite: Reducing GPU programming complexity. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, Edmonton, Canada, pp. 1–15.

von Neumann, J. (1993). First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4), 27–75.

Watson, K. M. (1953, February). Multiple scattering and the many-body problem - applications to photomeson production in complex nuclei. *Physical Review*, 89(3), 575–587.

Wilde, M., M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster (2011). Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9), 633–652.

Wilkinson, B. and M. Allen (1999). *Parallel programming: Techniques and Applications using Networked workstations and parallel computers*. Prentice-Hall, Inc.

Wilson, G. (2006). Where's the real bottleneck in scientific computing? *American Scientist*, 94(1), 4.

Wolfe, M. J., C. Shanklin, and L. Ortega (1995). *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc.

Wu, H. and J. Gray (2005, October). Testing domain-specific languages in Eclipse. In *Companion to the 20th International Conference on Object-oriented Programming, Systems, Languages, and Applications*, San Diego, CA, pp. 173–174.

Xu, Z. (2010, February). Treecode algorithm for pairwise electrostatic interactions with solvent-solute polarization. *Physical Review E*, 81(2), 020902.

Yang, R. and H. Aiso (1986, July). P-Prolog: A parallel logic programming language based on exclusive relation. In *Proceedings on Third International Conference on Logic Programming*, London, United Kingdom, pp. 255–269.

Zhang, B., J. Huang, N. Pitsianis, and X. Sun (2010, December). Revision of fmm-yukawa: An adaptive fast multipole method for screened Coulomb interactions. *Computer Physics Communications*, 181(12), 2206–2207.

Zima, H. and B. Chapman (1991). *Supercompilers for Parallel and Vector Computers*. ACM Press.

Zobel, J. and A. Moffat (2006, July). Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 1–56.

Appendices

Appendix A
PPMODEL

In this Appendix, the grammar, template files, and an example of PPModel are shown in the following sections.

A.A PPModel ANTLR Grammar

```
grammar TPPModel;
options {
    backtrack=true;
    memoize=true;
    k=3;
}
@header {
package grmr;
}
@lexer::header {package grmr;}

content returns [ Content cont]
    :  l_decls=declarations mps=mappings vrs=execute '.' EOF
    {$cont= new Content($l_decls.listofdecls ,$mps.maps,$vrs.execution);}
    ;

declarations returns [ List<Declaration> listofdecls]
@init
    {
      $listofdecls = new ArrayList<Declaration >();
    }
    : ('declare'  nf=nameandfile '{' 'in' in=variables 'out' out=variables'}'
    {listofdecls.add(new Declaration($nf.namefile ,$in.variables ,$out.variables));})+
    ;

mappings returns [ Map<String , List<NameTypeAndFile>> maps]
@init
    {
      $maps = new HashMap<String , List<NameTypeAndFile>>();
    }
    : ('map' id1=IDENTIFIER INTO  lfs=listofnameTfile
    {maps.put($id1.text ,$lfs.namefiles);})+
    ;

execute returns [ Execution execution]
    : 'execute'id=IDENTIFIER vars=variables
    {$execution=new Execution($id.text ,$vars.variables);}
    ;

listofnameTfile returns [ List<NameTypeAndFile> namefiles]
@init
    {
      $namefiles = new ArrayList<NameTypeAndFile >();
    }
  :p_t1=platform nf1=nameandfile
  {$namefiles.add(new NameTypeAndFile($p_t1.platName , $nf1.namefile));}
  (',' p_t2=platform nf2=nameandfile{$namefiles.add(new NameTypeAndFile($p_t2.platName , $nf2.
    ↪namefile));})*
  ;

variables returns [ List<String> variables]
@init
    {
      $variables = new ArrayList<String >();
    }
  : id1=IDENTIFIER {$variables.add($id1.text);}  (',' id2=IDENTIFIER {$variables.add($id2.text)
      ↪;})*
  ;


nameandfile returns [NameAndFile namefile]
  : id=IDENTIFIER f1=file {$namefile=new NameAndFile($id.text,$f1.fileName);}
  ;

file returns [ String fileName]
  : '[' id1=IDENTIFIER '.' id2=IDENTIFIER']' {$fileName=$id1.text +"."+ $id2.text;}
  ;

platform returns [ String platName ]
  : 'CUDA'{$platName="CUDA";}|'OMP'{$platName="OMP";}|'MPI'{$platName="MPI";}|'OCL'{$platName="
      ↪OCL";}
  ;

IDENTIFIER
  : LETTER (LETTER|'0'..'9')*
  ;

INTO
  : '<-'
  ;

fragment
LETTER
  : '$'
  | 'A'..'Z'
  | 'a'..'z'
  | '_'
  ;

WS  :  (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;}
    ;

COMMENT
    :   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;

LINE_COMMENT
    : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    ;

// ignore #line info for now
LINE_COMMAND
    : '#' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    ;
```

A.B PPModel Templates

In this section, the CUDA, MPI and OpenMP templates are shown.

A.B.1 PPModel CUDA Template

```
template(name,_vars,_ptrs, o_ptr, i_ptr_ws,size) ::=<<#include \<stdio.h\>
#include \<cutil_inline.h\>


void extern timerstart(char* name);
void extern timerend();

// Device code
__device__ void <name>_main(<i_ptr_ws:{s|<s.parameter>}; separator=",">, int opt)
{
/* Sample Vector add code
 C[opt]=A[opt]+ B[opt];
*/



}

//This kernel distributes the work irrespective of the size
__global__ void <name>_kernel(<_ptrs:{s|<s.parameter>}; separator=",">,<_vars:{s|<s.parameter>};
    ↪separator=",">)
{
    const int      tid = blockDim.x * blockIdx.x + threadIdx.x;
    const int THREAD_N = blockDim.x * gridDim.x;

    for(int opt = tid; opt \< <size>; opt += THREAD_N){
          <name>_main(<i_ptr_ws:{s|<s.name>}; separator=",">,opt);
  }
}

// Host code
<o_ptr.tType>* abstract_<name>(<_ptrs:{s|<s.parameter>}; separator=",">,<_vars:{s|<s.parameter>};
    ↪ separator=",">)
{
      timerstart("CUDA");
<_ptrs:{s| <s.type>* d_<s.name> ; }; separator="\n">

    // Allocate vectors in device memory
    <_ptrs:{s|cutilSafeCall( cudaMalloc((void**)&d_<s.name>, sizeof(<s.type>)*<s.size>)) ; };
        ↪separator="\n">

    // Copy variables from host memory to device memory

    <_ptrs:{s| cutilSafeCall( cudaMemcpy(d_<s.name>,<s.name>, sizeof(<s.type>)*<s.size>,
        ↪cudaMemcpyHostToDevice) ); }; separator="\n">

    // Kernel call with 480*256 threads



    <name>_kernel\<\<\<480, 256\>\>\>(<_ptrs:{s|d_<s.name>}; separator=",">,<_vars:{s|<s.name>};
        ↪separator=",">);


    cutilCheckMsg("kernel launch failure\n");
    cutilSafeCall( cudaThreadSynchronize() );

    // Copy variables from device memory to host memory

    <o_ptr:{s| cutilSafeCall( cudaMemcpy(<s.name>,d_<s.name>, sizeof(<s.type>)*<s.size>,
        ↪cudaMemcpyDeviceToHost) ); }; separator="\n">

      <_ptrs:{s| if (d_<s.name>) cutilSafeCall( cudaFree(d_<s.name>)) ; }; separator="\n">
      timerend();

    return <o_ptr.name>;

}
>>
```

A.B.2 PPModel OpenMP Template

```
template(name,_vars,_ptrs, o_ptr, i_ptr_ws,size) ::=<<#include \<stdio.h\>
#include \<omp.h\>

void extern timerstart(char* name);
void extern timerend();

// Host code
<o_ptr.tType>* abstract_<name>(<_ptrs:{s|<s.parameter>}; separator=",">,<_vars:{s|<s.parameter>};
    ↪ separator=",">)
```

```
{

  timerstart("OMP");
  /************
   SAMPLE OMP CODE GOES BELOW

  int i, chunk = 100;
#pragma omp parallel shared(A,B,C,chunk) private(i)
  {
#pragma omp for schedule(dynamic,chunk) nowait
    for (i = 0; i \< TOTAL_SIZE; i++)
      C[i] = A[i] + B[i];

  }
   *************/
  timerend();
        return <o_ptr.name>;
}
>>
```

A.B.3 PPModel MPI Template

```
template(name,_vars,_ptrs, o_ptr, i_ptr_ws,size) ::=<</*MPI Template */
#include \<stdio.h\>
#include \<mpi.h\>

void extern timerstart(char* name);
void extern timerend();

// Host code
<o_ptr.tType>* abstract_<name>(<_ptrs:{s|<s.parameter>}; separator=",">,<_vars:{s|<s.parameter>};
    ↪ separator=",">)
{

  timerstart("MPI");
  int rank, size;

  MPI_Init (&argc, &argv);  /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);  /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);  /* get number of processes */

/********************
 Implementation goes here

*********************/

  MPI_Finalize();

  timerstart("MPI");
        return <o_ptr.name>;

}
>>
```

A.C PPModel Example: Integer Sorting

A.C.1 Main Program File: Vectoradd

```
#include <stdio.h>
#define TOTAL_SIZE 1000
        int A[1000], B[1000],C[1000];

void vectoradd()
{
        int i;
        for(i=0;i<TOTAL_SIZE;i++)
        {
        C[i]= A[i]+B[i];
        }
}
void verify()
{
        int i;
        for(i=0;i<TOTAL_SIZE;i++)
        {
        if(C[i]!=2)
        {
                printf("Failed");
                return;
        }
        }

        printf("Passed");
}
int main()
{
        int i;


        for( i=0; i<TOTAL_SIZE;i++)
        {
                A[i]=1;
                B[i]=1;
        }
        #pragma tppmodel vectorAdd
        {
                vectoradd();
        }


        verify();
        return 0;
}
```

A.C.2 tPPModel File: Vectoradd

```
        declare  vectorAdd [vectoradd.cpp] {
        in A,B,C, TOTAL_SIZE
        out C
        }
        map vectorAdd<-CUDA cuda_VA [VAcuda.cu], OMP omp_VA [VAomp.cpp], MPI omp_VA [VAmpi.cpp]

        execute VACUDA_EXE cuda_VA.
```

Appendix B
PNBSOLVER

In this Appendix, the grammar and two examples of PNBsolver are shown in the following sections.

B.A PNBsolver ANTLR Grammar

```
grammar PNBsolver;
options {
    backtrack=true;
    memoize=true;
    k=3;
}
@header {
package grmr;
}
@lexer::header {package grmr;}

content returns [ Content cont]
    :  l_decls=kernels  vrs=execute '.' EOF
    {$cont= new Content($l_decls.listofkernels ,$vrs.execution);}
    ;

kernels returns [ List<Kernel> listofkernels]
@init
    {
      $listofkernels = new ArrayList<Kernel >();
    }
    : ('kernel'  nf=IDENTIFIER 'in' id2=IDENTIFIER decls=declarations stmts=readstmts stmt=
       ↪expressionstmt wstmts=writestmts 'endkernel'
    {listofkernels.add(new Kernel($nf.text ,$stmts.stmts ,stmt ,$decls.stmts));})+
    ;

declarations returns [ List<Statement> stmts]
@init
    {
      $stmts = new ArrayList<Statement >();
    }
    : (op=vectorscalarnumber vars=variables {Statement stmt= new Statement($op.vec_sca_num ,$vars.
       ↪variables);stmts.add(stmt);})+
    ;

expressionstmt returns [ Statement stmt]
    : id1=IDENTIFIER '=' (expr1=expression)? 'SUM' expr2=expression {stmt= new Statement($id1.
       ↪text ,$expr1.value ,$expr2.value);}
    ;

expression returns [Expression value]
@init
    {
          List<Expression> exprlist = new ArrayList<Expression >();
          List<String> opslist = new ArrayList<String >();
    }
    :   e1=multidiv
        (   '+' e2=multidiv {exprlist.add($e2.value);opslist.add("+");}
        |   '-' e2=multidiv {exprlist.add($e2.value);opslist.add("-");}
        )* {if(exprlist.size()<1)$value=$e1.value; else $value = new Expression($e1.value,
            ↪exprlist ,opslist); }
    ;

multidiv returns [Expression value]
```

```
@init
    {
         List<Expression> exprlist = new ArrayList<Expression>();
         List<String> opslist = new ArrayList<String>();
    }
    :   e1=atom
    ('*' e2=atom {exprlist.add($e2.value);opslist.add("*");}
    |'/' e2=atom {exprlist.add($e2.value);opslist.add("/");})* {if(exprlist.size()<1)$value=$e1.
        ↪value; else $value = new Expression($e1.value,exprlist,opslist); }
    ;

atom returns [Expression value]
    :   id=IDENTIFIER{value=new Expression($id.text);}
    |   '(' expr=expression ')' {$value = $expr.value;}
    |   'exp' expr=expression {$value= new Expression(expr,"exp");}
    |    'pow' '('expr=expression ',' num=NUMBER ')' {int n= Integer.parseInt($num.text);$value=
        ↪new Expression(expr,n);}
    ;

readstmts returns [ List<Statement> stmts]
@init
    {
      $stmts = new ArrayList<Statement>();
    }
    : ('read' id1=STRING ',' id2=STRING {Statement stmt= new Statement($id1.text,$id2.text,0);
        ↪stmts.add(stmt);})*
    ;

writestmts returns [ List<Statement> stmts]
@init
    {
      $stmts = new ArrayList<Statement>();
    }
    : ('write' id1=STRING ',' id2=STRING {Statement stmt= new Statement($id1.text,$id2.text,1);
        ↪stmts.add(stmt);})*
    ;

execute returns [ Execution execution]
    : 'generate' p_t1=platform ('[' parameters ']')? md=mode id=IDENTIFIER
    {$execution=new Execution($id.text,$p_t1.platName,$md.md);}
    ;

variables returns [ List<String> variables]
@init
    {
      $variables = new ArrayList<String>();
    }
  : id1=IDENTIFIER {$variables.add($id1.text);} (initialization)? (',' id2=IDENTIFIER {$variables
      ↪.add($id2.text);}(initialization)?)*
  ;

initialization
  : '=' NUMBER ('.' NUMBER)?
  ;

platform returns [ int platName ]
  : 'CUDA'{$platName=0;}|'OMP' {$platName=1;}|'MPI'{$platName=2;}|'OCL'{$platName=3;}
  ;

parameters
    : IDENTIFIER '=' parameter  (',' IDENTIFIER '=' parameter)*
    ;

parameter
  :IDENTIFIER
  |NUMBER ('.' NUMBER)?
  ;

```

```
mode returns [ int md ]
  : 'ACCURATE'{$md=0;}|'AVERAGE'{$md=1;}|'FAST'{$md=2;}
  ;


vectorscalarnumber returns [ int vec_sca_num ]
  : 'vector'{$vec_sca_num=0;}|'scalar'{$vec_sca_num=1;}|'constant'{$vec_sca_num=2;}
  ;

NUMBER
  : '0'..'9'+
  ;

IDENTIFIER
  : LETTER (LETTER|'0'..'9')*
  ;

INTO
  : '<-'
  ;
STRING
    :  '"' ( EscapeSequence | ~('\\'|'"') )* '"'
    ;
fragment
EscapeSequence
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UnicodeEscape
    |   OctalEscape
    ;

fragment
OctalEscape
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UnicodeEscape
    :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    ;

fragment
HexDigit : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
LETTER
  : '$'
  | 'A'..'Z'
  | 'a'..'z'
  | '_'
  ;

WS  :  (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;}
    ;

COMMENT
    :   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;

LINE_COMMENT
    : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    ;

// ignore #line info for now
LINE_COMMAND
    : '#' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    ;
```

B.B PNBsolver Example 1: Force Calculation

```
kernel Force in R
// Kernel declarations
  vector F
  scalar M
  constant K=1

 /*
  * Read positions and mass from file
  * Data is formatted as shown
  *
  */
  read "<R_1 R_2 R_3>M","data.dat"

//Actual computation
  F=K*M SUM(M*R/(R_*R_*R_))

// Write force to file
  write "F_1 F_2 F_3", "out.dat"

endkernel

// Generate CUDA code for force kernel
generate OMP[MAXPARNODE=1] ACCURATE Force.
```

B.C PNBsolver Example 2: Potential Energy Calculation

```
kernel Potential in R

// Kernel declarations
        scalar PE

 /*
  * Read positions and mass from file
  * Data is formatted as shown
  *
  */
        read "R_1 R_2 R_3","data.dat"

//Actual computation
        PE=Q SUM(Q/R)

// Write force to file
        write "PE", "out.dat"
endkernel
generate OMP[MAXPARNODE=1] ACCURATE Potential.
```

Appendix C
MAPREDOOP

In this Appendix, the Xtext grammar of MapRedoop and two example MapRedoop applications: 1) BFS and 2) K-means are shown.

C.A MapRedoop Xtext Grammar

```
grammar cs.ua.edu.se.MapRedoop with org.eclipse.xtext.common.Terminals

generate mapRedoop "http://www.ua.cs/edu/se/MapRedoop"

MapRedoop
    :declaration=Declaration'{'(content+=Content)*'}'
    ;

Content
    : entities=ListofEntities|mrBlocks=MRBlock
    ;

ListofEntities
    : {ListofEntities}'metaelements:''{'(entities+=Entity)*'}'
    ;

MRBlock
    :'mapreduce:'loop='loop'? '{'mapper=Mapper reducer=Reducer '}'
    ;

Declaration
    :'program' name=ID ('extend' superName=ID)?
    ;

Mapper
    :'map''('inkey=Argument','invalue=Argument ',' outKeyType=STRING ','outValueType=STRING')'
        ↪text=Block
    ;

Reducer
    : 'reduce''('inkey=STRING','invalue=STRING ',' outKeyType=STRING ','outValueType=STRING')'
        ↪text=Block
    ;

Entity
    :'metaelement' name=ID ('extend' superType=[Entity])? '{'(features+=Feature)+ '}'
    ;

Feature
    : type=TypeRef name=ID';'| readorWrite=ReadorWrite';'
    ;

ReadorWrite
    : 'read''(' token=STRING',' readval=STRING')'| 'write''(' token=STRING ','writeval=STRING')'
    ;

TypeRef
    : referenced=Type(multi?='*')?
    ;

Block
    :  block='[' (javafunctions+=JavaFunction)* ']'
    ;

```

```
JavaFunction
    : timeofCall=TimeOfCall ':' paramname=ID
    ;

TimeOfCall
    :'after' | 'call' | 'before'
    ;


Type
    : type='int'|type='float'|type='double'|type='text'|type='long' | name=ID
    ;

Argument
    :type=Type argname=ID
    ;

ListofArgument
    :','type=Type argname=ID
    ;
```

C.B MapRedoop Example 1: BFS

In this example, MapRedoop code is shown with the relevant input and output formats.

C.B.1 Sample Input

```
0(0) : 1 3
1(10) : 2 3
2(10) : 4
3(10) : 1 2 4
4(10) : 2 0
```

C.B.2 Sample Output

```
0{0} :  1  3
1{1} :  2  3
2{2} :  4
3{1} :  1  2  4
4{2} :  2  0
```

C.B.3 MapRedoop Code

```
program BFS {
  metaelements: {
    metaelement Node {
     long nodeId;
     long distanceFromSource  ;
     int nodeType ;
     int* adjacentlist;
     read(':',"%nodeId%{%distanceFromSource%} : %adjacentlist%" );
    }
  }
  mapreduce: loop {
    map( text mapkey, Node mapnode, "long" , "Node" ) [
         call  : emitStructure
         after : emitDistance
      ]
    reduce("redkey", "nodes", "long", "Node")[
         call: minmizeDistance
      ]
  }
}
```

C.C MapRedoop Example 2: K-means

In this example, MapRedoop code and the code generated from the MapRedoop file is shown (only two classes are shown).

C.C.1 MapRedoop Code

```
program Kmeans{
  metaelements: {
    metaelement KVector {
    float point1;
    float point2;
     read(':',"elts: {0:%point1%, 1:%point2%}" );
    }
    metaelement ClusterInfo {
    KVector center;
    long id;
    }
  }
  mapreduce: loop {
    map( text mapkey, KVector mapvector, "long" , "KVector" ) [
        before : loadClustersMap
        call   : emitToNearestCluster
      ]
    reduce("redkey", "clustervalues", "long", "KVector")[
        before : loadClustersRed
        call: calculateNewCenter
      ]
  }
}
```

C.C.2 KVector Class Generated

```
package hadoop.ds;


import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.WritableComparable;

public class KVector implements WritableComparable<KVector> {

  public KVector() {
    // TODO Auto-generated constructor stub
  }
  private float point1;
  public void setpoint1(float point1) {
    this.point1 = point1;
  }
  public float getpoint1() {
    return point1;
  }
  private float point2;
  public void setpoint2(float point2) {
    this.point2 = point2;
  }
  public float getpoint2() {
    return point2;
  }

    public void readFields(DataInput arg0) throws IOException {
      point1 = arg0.readFloat();
      point2 = arg0.readFloat();
  }
  public void write(DataOutput arg0) throws IOException {
      arg0.writeFloat(point1);
      arg0.writeFloat(point2);

  }
  public int compareTo(KVector arg0) {

        return 0;
  }
  public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("elts: {0:");
        buffer.append(point1);
        buffer.append(", 1:");
        buffer.append(point2);
        buffer.append("}");
        return buffer.toString();
  }

}
```

C.C.3 Main Class Generated

```
package hadoop.core;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import hadoop.ds.KVector;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
public class Main extends Configured implements Tool {
  public static class MapClass extends Mapper<Text, KVector, LongWritable, KVector> {

    protected void map(Text mapkey, KVector mapvector, Context context)
    throws IOException, InterruptedException{
        new CoreHelper().emitToNearestCluster( mapkey,mapvector,context);


    }
    protected void setup(org.apache.hadoop.mapreduce.Mapper<org.apache.hadoop.io.Text,KVector,
        ↪LongWritable,KVector>.Context context)
    throws IOException ,InterruptedException {
        new CoreHelper().loadClustersMap( context);
    }

  }
  public static class Reduce extends Reducer<LongWritable, KVector, LongWritable, KVector> {
    public void reduce(LongWritable redkey, Iterable<KVector> clustervalues, Context context)
        ↪throws IOException, InterruptedException {
        new CoreHelper().calculateNewCenter( redkey,clustervalues,context);



    }
  }
  public int run(String[] args) throws Exception {

    Path finaloutput = new Path(args[1]);
    Path input = new Path(args[0]);
      Configuration configuration = new Configuration();
      FileSystem  fs1 = FileSystem.get(finaloutput.toUri(), configuration);
        if (fs1.exists(finaloutput))     fs1.delete(finaloutput, true);
      Job job = new Job(configuration, "Kmeans");
```

```java
      job.setOutputKeyClass(LongWritable.class);
      job.setOutputValueClass(KVector.class);

      job.setMapperClass(MapClass.class);
      job.setReducerClass(Reduce.class);
      job.setInputFormatClass(SequenceFileInputFormat.class);
      job.setOutputFormatClass(SequenceFileOutputFormat.class);
      FileInputFormat.addInputPath(job, input);
      FileOutputFormat.setOutputPath(job,finaloutput);
      job.setJarByClass(Main.class);

      if (job.waitForCompletion(true) == false) {
        throw new InterruptedException("Kmeans Iteration failed processing " );
      }

    return 0;

  }
  public static void main(String[] args) throws Exception {
  //  Setup.begin();
    int res = ToolRunner
    .run(new Configuration(), new Main(), args);
  //  Setup.end();
    System.exit(res);

  }

}
```

Appendix D

SDL & WDL

In this Appendix, the SDL and the WDL grammars are shown along with some sample usages and code generated.

D.A SDL Xtext Grammar

```
grammar gov.pnl.sdi.SDL with org.eclipse.xtext.common.Terminals

generate sDL "http://www.pnl.gov/sdi/SDL"

Model
    :(defintions+=Definition|service+=Service)+
    ;

Definition
    :'define' executable=Executable
    ;

Executable
    :name=ID 'as' path=STRING  'at' machine=STRING ('with-key' key=STRING)?
    ;

Service
    :'service' name=ID ("extends" super=ID)? "{"
                                (connectStmt=ConnectStmt";")?
                                (cmdStmt=CmdStmt";")?
                                (resourceStmt=ResourceStmt";")?
                                (ioStmts+=IOStmt*)
                               "}"
    ;

CmdStmt
    :"cmd" cmd=STRING
    ;

ConnectStmt
    :"use" exe_ref=ID
    ;

IOStmt
    :(type="in"|type="out") (modifierList="list")? modifier="doc"? variables=Variables ";"
    ;

ResourceStmt
    :"resource" resources=Resources
    ;

Resources
    :variable1=STRING ("," variable2+=STRING)*
    ;

Variables
    :variable1=ID ("," variable2+=ID)*
    ;
```

D.B WDL Xtext Grammar

```

grammar gov.pnl.sdi.WDL with org.eclipse.xtext.common.Terminals

generate wDL "http://www.pnl.gov/sdi/WDL"

WorkflowModel
    :'use' servicefile=STRING workflows+=Workflow+
    ;

Workflow
    :'workflow' name=ID '('parameters=Parameters? ')' '{'
      (definitions+=Definition*)
      (stringConstants+=StringConstant*)
      (portlinks+=PortLink|serviceLinks+=ServiceLink|workflowCalls+=WorkflowCall)+'}'
    ;


WorkflowCall
    :'call' workflowID=ID
    ('till' criterion=Criterion)?
     'with' argument=Port (',' moreArguments+=Port)*
    ;

Criterion
    :port=ID op=OPERATOR value=STRING
    ;

OPERATOR
    :"="|"<"|">"
    ;

StringConstant
    :'String' stringAssignments=StringAssignments
    ;

StringAssignments
    :assignment=StringAssignment(',' moreAssignments+=StringAssignment)*
    ;

StringAssignment
    :name=ID'=' value=STRING
    ;

Definition
    :'define' name=ID services=Services
    ;

Services
    : service1=ID (',' service2+=ID)*
    ;

ServiceLink
    :service1=ID'|' service2=ID
    ;

PortLink
    :(port1=Port|text=STRING)'->' port2=Port
    ;

Port
    :serviceName=ID('.'portName=ID)? ('after' afterServiceName=ID)?
    ;

Parameters
    :parameter=Parameter (',' moreParameters+=Parameter)*
    ;
```

```

Parameter
   :type=('in' |'out') variable=ID
   ;
```

D.C SDL Examples

```
//Declaring a connection with security
 define ssh_oly as "sdi" at "olympus.pnl.gov"
   with-key "C:\\cygwin\\home\\jaco181\\.ssh\\id_rsa"

//Declaring a connection without security
 define ssh_exe as "jaco181" at "sdi.pnl.gov"

 define ssh_exe as "jaco181" at "sdi.pnl.gov"
   with-key "C:\\cygwin\\home\\jaco181\\.ssh\\id_rsa"

//A simple echo service
 service  echo {
   env ssh_exe;
   cmd "cat input > output";
   in input;
 }

//A service with only output
 service sayHi {
 use ssh_exe;
   cmd "echo 'someText' > outFile" ;
   out doc outFile;
 }

//A service with two outputs (first document as 1 and second has 2 in it)
 service sayHello {
   use ssh_exe;
   cmd "echo 'text1=1' > .properties; echo 'text2=2' >> .properties; echo 'let us' > file1;echo
       ↪'see' > file2;" ;
   out  text1,text2;
   out doc file1,file2;
 }

//A service for collecting all the inputs and output as one file
 service aggregate{
   use ssh_exe;
   cmd 'cat $inputs;  separator=\" \"  $ > finalOut' ;
   in list doc inputs;
   out doc finalOut;
 }

//Classifier training service
 service classifier_Training {
   use  ssh_exe;
   cmd "R CMD BATCH training.r training.out";
   resource "training.r";
   in doc trainXFile, trainYFile;
   in algorithm;
   out doc modelFile;
 }

//Classifier testing service
 service classifier_Testing {
   use  ssh_exe;
   cmd "R CMD BATCH testing.r testing.out";
   resource "testing.r";
   in doc testXFile, modelFile, testYFile;
   in algorithm;
   out doc outFile;
 }

```

```
//Accuracy service
 service accuracy {
   use ssh_exe;
   cmd "R CMD BATCH accuracy.r accuracy.out";
   resource "accuracy.r";
   in doc inputFile;
   out doc outputFilePrefix;
 }

//A service to submit BLAST job
 service submitBlast {
   use ssh_oly;
   cmd "sh runJob.sh";
   resource "jobScript.sh", "runJob.sh";
   in doc blossum, params, fasta;
   out jobID, outDir;
 }

//A service to check job status
 service jobStatus {
   use ssh_oly;
   cmd "sh checkStatus.sh";
   resource "checkStatus.sh";
   in jobID;
   out status;
 }

//A service to collect the BLAST output file
 service blastResult {
   use ssh_oly;
   cmd "cp $outDir$/test_all_v_all_m8.out outFile";
   in outDir;
   out doc outFile;
 }
```

D.D Generated Classes for Accuracy Service

SDL can create or modify four classes in four packages (`rmi`, `ws`, `services` and `hepers`) for every new service added. The code generated for the Accuracy service in each package is shown in the following sub-sections.

D.D.1 Method Generation- `rmi` Package

```
/**
 *
 * @param inputFile
 * @param
 * @param
 * @param
 * @return
 * @throws Exception
 */
public  int accuracy(
int inputFile
  ) throws Exception ;
```

D.D.2 Method Generation- `ws` Package

```
@RequestWrapper(partName = "accuracyIn")
@ResponseWrapper(partName = "accuracyOut")
@WebResult(name = "accuracyResults")
@WebMethod
public  int accuracy(@WebParam(name =  "inputFile") int inputFile   )
  throws Exception;
```

D.D.3 Method Generation- `services` Package

```java
@Secured(value = { "ROLE_AUTHENTICATED" })
public    int accuracy(
int inputFile
) throws Exception {
   AccuracyHelper accuracyHelper = new AccuracyHelper("jaco181","sdi.pnl.gov","C:\\cygwin\\home
       ↪\\jaco181\\.ssh\\id_rsa");
   File outputFilePrefix = AfFileUtils.createTemporaryFile("outputFilePrefix");

   accuracyHelper.accuracy(
   getUri(inputFile)  ,
   outputFilePrefix.getPath()
   );
       accuracyHelper.close();
return  uploadFile(1, 1 ,outputFilePrefix.toURI().toString());
  }
```

D.D.4 Class Generation- `helpers` Package

```java
package  utils.helpers;
import  utils.AfFileUtils;
import java.net.URI;

public class AccuracyHelper extends AbstractSSHHelper {
public AccuracyHelper(String userName, String host,String keyLocation) {
  super( userName, host, keyLocation);
}
public void accuracy(
    String inputFile  ,
    String outputFilePrefixPath
    ) throws Exception {

    URI inputFileUri = new URI(inputFile) ;
    String inputFilePath = AfFileUtils.resolveFile(inputFileUri);
    uploadFile("inputFile", inputFilePath );
    templateVariables.put("inputFile", "inputFile");

    templateVariables.put("outputFilePrefix", "outputFilePrefix");
    uploadClassPathFile("accuracy.r");
    executeScript("R CMD BATCH accuracy.r accuracy.out");
    downloadFile("outputFilePrefix", outputFilePrefixPath);
  }
}
```