A SIMILARITY-BASED ANALYSIS TOOL FOR SCIENTIFIC APPLICATION PORTING

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Wei Ding

December 2013

A SIMILARITY-BASED ANALYSIS TOOL FOR SCIENTIFIC APPLICATION PORTING

Wei Ding

APPROVED:

Dr. Barbara M. Chapman
Dept. of Computer Science

Dr. Edgar Gabriel
Dept. of Computer Science

Dr. Marc Garbey
Dept. of Computer Science

Dr. Zhigang Deng
Dept. of Computer Science

Dr. Kevin E. Bassler
Department of Physics

Dean, College of Natural Sciences and Mathematics

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Barbara Chapman, for her invaluable advice and guidance in my Ph.D. study. I really appreciate all her contribution of time, discussion, and trust for giving me a lot research opportunities and funding to make my Ph.D. study productive.

Special thanks to all my committee members: Dr. Edgar Gabriel, Dr. Marc Garbey, Dr. Zhigang Deng, and Dr. Kevin E. Bassler for their time, insightful comments and help.

Specifically, I am very grateful to Dr. Oscar R. Hernandez from the Oak Ridge National Laboratory for giving me generous help during my summer internship at ORNL. Additionally, I very much appreciate his guidance during my research.

I would also like to thank my fellow lab-mates of the HPCTools group: Dr. Yonghong Yan, Dr. Sunita Chandrasekaran, Deepak Eachempati, Tony Curtis, Rita Hazlewood, Xiaonan Tian, and Jingxin Yang for their valuable discussion and friendships. I really enjoy working with them.

Last, but not least, my dissertation is dedicated to my wife, Xiaoxi Man and my daughter, Jennifer Yuqi Ding. Also, I cannot express my appreciation for endless supports from my parents, Changle Ding and Hanqun Yi. Their love and support provided me the driving force, courage, and perseverance. Thank you all for supporting me along the way.

A SIMILARITY-BASED ANALYSIS TOOL FOR SCIENTIFIC APPLICATION PORTING

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Wei Ding

December 2013

Abstract

Porting applications to a new system is a nontrivial job in the HPC field. It is a very time-consuming, labor-intensive process, and the quality of the results will depend critically on the experience of the experts involved. In order to ease the porting process, we propose a methodology to address an important aspect of software porting that receives little attention, namely planning support. When a scientific application consisting of many subroutines is to be ported, the selection of key subroutines greatly impacts the productivity and overall porting strategy, because these subroutines may represent a significant feature of the code in terms of functionality, code structure, or performance. They may also serve as indicators of the difficulty and amount of effort involved in porting a code to a new platform. The proposed methodology is based on the idea that a set of similar subroutines can be ported with similar strategies and result in a similar-quality porting. By viewing subroutines as data and operator sequences, analogous to DNA sequences, we are able to use various bio-informatics techniques to conduct the similarity analysis of subroutines while avoiding NP-complete complexities of other approaches. To further improve accuracy for porting, we also merged some other code metrics and cost-model metrics for similarity analysis to capture the internal code characteristics. In this dissertation, we describe our methodology, which includes presentation of a tool called *Klonos*. To evaluate the effectiveness of *Klonos*, we used it to conduct experiments to find strategies for porting of several scientific benchmarks and a large scientific application. Our experiment shows *Klonos* is very effective for providing a systematic porting plan to guide the users during their porting process of reusing similar porting strategies for similar code regions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The High Performance Computing (HPC) community is heading toward the era of exascale computing. Although the final form of an exascale machine is yet unknown, this system is expected to exhibit a hitherto unprecedented level of complexity and size, and this architectural innovation comes with a high cost to the users of the system. Today, computer architects are already building systems that employ Graphical Processing Units (GPUs) in order to provide more favorable power/performance ratios. The new Titan supercomputer [36] at Oak Ridge National Laboratory (ORNL) exemplifies this trend. Hardware platforms are currently undergoing drastic changes that require significant code reorganization in order to provide much higher data locality, and to map the computations and data to different kinds of devices configured within the node. To deploy a highly efficient code on such systems, which we need

to explore different levels of parallelism from the inter and intra nodes by using the becoming more and more complicated programming models.

In order to migrate codes to Titan, scientists will need to know how to exploit not only the large number of CPU cores, but also the GPUs that are configured on the nodes. They will need to create new computational kernels with suitable granularity to exploit the GPUs, while minimizing costly data movements, exploiting complex memory subsystems, and mapping the work to balance the overall load. They may need to use a hybrid programming model, such as adding OpenMP [9, 25] and accelerator directives [35, 41] to MPI applications [84], or they may introduce Pthreads [22] or an API designed for accelerators [61, 65]

Unfortunately, it is not possible to fully automate the task of restructuring codes to exploit the capabilities of heterogeneous node architectures. In our early evaluation of porting two real scientific kernels to GPUs, we found the porting process is very time-consuming. Users have to rely on compiler and other profiling tools to locate bottlenecks and manually to do the optimization in order to get desired performance. Also according to our experiments, we found that many subroutines have very similar porting strategies which suggest that many aspects of the process are highly repetitive. In other words, a successful restructuring strategy, once identified, can typically be re-used in multiple code regions. Yet until now, nobody has studied or quantified the degree of re-usability of porting strategies with respect to real scientific applications porting in the HPC field.

Besides these challenges faced by the application developers, many scientific applications tend to be long lived and comprise large code bases developed by a team

of people, and often outlive the involvement of any single developer. To maintain those applications is not a easy task, let alone to port them to the new system while with guaranteed efficiency and correctness. Rewriting the entire application is too expensive and seems unrealistic. In contrast, simulation platforms emerge at a frequency much higher than the lifetime of such application code bases, at times fairly intrusive changes required to fully benefit from the capabilities such platforms provide, which take years to integrate. These changes are made *manually* due to the lack of tool support. Automation of the process is rather impractical given its complexity, the level of tool support, and the lack of tool integration that automates user-driven porting strategies. Thus, the porting of scientific applications is both time consuming, labor intensive, and erroneous. Even an "expert team" needs to start somewhere. This team will initiate the effort to restructure the code in order to exploit the new systems features, a very challenging error-prone process; the quality of the results will depend critically on the experience of the experts involved.

In order to reduce the effort for the software porting, we create a tool called *Klonos* which is based on a methodology to address an important aspect of software porting that receives little attention, called tool-based *planning* support [44, 82, 83], in this thesis. Given a scientific application with many subroutines, the planning problem is defined as finding a good *order* on which subroutine to port first and which one next so that the porting process is productive and maximizes the knowledge gained from previous subroutine porting experiences. In this work, we broaden the similarity concept, only not exploring source code structure similarity, but also other metrics to define if two source codes could be be optimized similarly. We can divide subroutines

into different groups based on their syntactic and code feature similarity and only concentrate on doing the best porting of a few subroutines from each group. By narrowing down the focus, we believe that we are able to provide a good planning support for the porting of scientific applications, and give an overall view of the porting effort involved in porting a code to a new platform.

1.2 Research Goals and Contributions

Emerging parallel architectures require software porting, but few tools are available for supporting software porting. Worse still, most of the tools are outdated for porting code to the emerging architectures such as GPUs, MICs, etc. For performance sensitive scientific applications, the porting includes not only the same functionality but also the same or better execution efficiency. In this dissertation, we create a tool called *Klonos* used for software porting, which addresses an important aspect of the software porting process called planning. We proposed a novel similarity-based methodology for the planning problem. Unlike previous similarity-based approaches, our work adopts a bio-inspired view of the program. We describe in detail the methodology developed and present a tool called *Klonos* to facilitate the process of porting applications to a new system. As a proof of concept, we conducted experiments on several scientific benchmarks and one real scientific application called HOMME with respect to porting to the OpenMP programming model and High-Level GPU directives. We showed that the methodology is effective and scalable in providing the planning support for the porting of scientific applications.

In fact, we were able to use the methodology to identify a possible optimization that the programmer missed for one of the codes, which is beyond the scope of planning. We also found out that similarity needs to be stricter than pure syntactic measures in some cases, which should contain more parallel information about the codes. We are currently investigating how to develop a better similarity metric. In particular, we are looking into various ways to combine syntactic similarity with cost models and parallel properties of the source code. We also want to extend our test base. We have planned to employ the methodology to test the behavior of existing compilers. In addition, we want to evaluate the effect of different score models and different views of similarity on the effectiveness of the methodology.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 explores the directive-based approach for porting two large scientific application kernels to GPU, which motivates our creation of a software porting tool. It describes several popular directives for accelerators to support porting serial code to GPUs from the optimization point of view and make a comparison of the performance by comparing with hand-written CUDA code. In addition, we evaluate how much effort is needed for restructuring the code during the porting process and how we can use the compiler to help us perform the optimization in order to get desired performance in a heterogeneous programming environment.

Chapter 3 provides an overview of the software porting challenges faced by the

High Performance Computing (HPC) community and related tools, methods used to help software porting.

Chapter 4 introduces the main framework of the *Klonos* tool. Detailed implementation for each key module is described.

Chapter 5 shows the similarity analysis we have by using *Klonos* with the NPB NAS Benchmark 3.3. We used OpenMP as an example to show how we can use *Klonos* to port serial code a shard memory environment. By referring to the optimized NPB NAS OpenMP codes, we have also verified the correctness of suggested porting plan produced by *Klonos*.

Chapter 6 explores different code feature metrics which could be used to increase the accuracy of using similar optimization strategies for similar code. We also explain why those metrics matter in our similarity analysis.

In Chapter 7, we evaluate the *Klonos* tool by applying *Klonos* to port serial code to a shared memory programming environment by using OpenMP. We then explain the steps for making a porting plan and demonstrate the effectiveness of using the proposed porting plan by *Klonos*, and propose a method for validating the porting plan.

Chapter 8 summarizes the whole work and describe the improvement we are going to make for *Klonos*, and other future work.

Chapter 2

Heterogeneous Programming for the GPU

This chapter describes our experience of porting serial code to GPU. We explore popular GPU programming models and compare the use of two sets of accelerator directives in two real-world application kernel studies. We explain the porting challenges and limitations encountered, and based on the lessons learned, reach initial conclusions on how to transform code to take advantage of the accelerator directive from the optimization point-of-view. We also compare the performance of running the codes on the GPU versus the CPU, and found that in all the cases the GPU yielded significantly better performance. In order to use the accelerator directives efficiently, it is necessary to perform code transformations to close the gap in performance to native CUDA implementations.

2.1 Heterogeneous Computing Today

Heterogeneous computing is not new, but the range of devices and the extent of their deployment is. Heterogeneous platforms may include Digital Signal Processors (DSPs), Field Programmable Gate Arrays (FPGAs), Application Specific Instruction Processors (ASIPs), stream processors, SIMD (Single Instruction Multiple Data) processors, and high-end floating point accelerators. Hybrid systems such as the IBM's Cell Broadband Engine have been used both for games as well as for technical computing. Low-cost GPUs are widely deployed. Intel and AMD have designed general-purpose chips that are heterogeneous.

Today's accelerators are typically GPUs, massively parallel processors that can be programmed to perform a range of computations including, but not limited to, their original graphics domain. The raw computing power of modern GPUs such as the NVIDIA Tesla series, Quadro and GeForce series, and ATI Radeon, ATI FirePro, and AMD FireStream have far outstripped that of the conventional CPU. A typical GPU contains a number of different SIMD engines, each of which may execute different code. Code running on a single SIMD engine is executed by its thread processors, each of which will process the same instruction sequence but on an independent data stream. For example, the ATI Radeon 3870 GPU has 4 SIMD engines, each with 16 thread processors; these, in turn, have five stream cores. The individual thread processors in an NVIDIA GeForce 8-series GPU can manage 96 concurrent threads, so that the GPU can potentially execute many thousands of threads simultaneously. A typical GPU computation will have a large number

of threads whose instructions are executed by a thread processor in an interleaved manner to hide memory access and instruction latencies. This implies that each such thread requires a high operation count if its processing power is to be exploited. The optimal thread count is highly application dependent. Communication between the host and GPU is typically provided by the PCI Express channel, with a data transfer performance that depends on the CPU and chipset. Prefetching can help to avoid penalties for data access.

The Nvidia Fermi presents many significant advances over its predecessors, although the basic architectural ideas remain the same. Fermi supports higher double precision floating point performance, error-correcting code (ECC), 64-bit unified addressing, and an extensive cache and memory architecture and hierarchy. Fermi has a 384-bit GDDR5 memory interface and 512 cores that give performance gain up to 8x. The 512 cores are arranged into 16 streaming multiprocessors(SM), where each SM (also known as a cluster) has 32 processors cores. The shared memory and L1 data cache are used for explicit and implicit communication between threads respectively. The 64 KB shared memory can be configured with a 16 KB or 48 KB shared memory, with the remainder used for L1 cache. An L2 cache is shared by all SMs.

2.2 Programming Models for Heterogeneous Systems

Several vendors have provided programming interfaces for accelerators. Most have adapted the C programming language to fit the strict requirements of applications on their platform. GPUs were originally programmed using OpenGL. Domain-specific languages for graphics programming like GLSL (OpenGL Shading Language), HLSL (high level shader language), and Cg (C for graphics) from NVIDIA are also available. With their growing usefulness for compute-intensive functions in general-purpose applications, a number of programming interfaces (mostly based on C) have been provided to facilitate the development of application kernels for them. Rather than being fully fledged languages, most of them are based upon C. These include StreamIt [14], Sh [57], Brook [21], CUDA [64], and OpenCL. Thrust [70] is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). UC Berkeley developed Copperhead [23] together with NVIDIA, a high-level data parallel language embedded in Python. In addition, other languages wrapper around CUDA, like PyCUDA and sgc-ruby-cuda, are also available. Matlab and Mathematica also have GPU CUDA plug-in for GPUs support. Compared with those programming languages, CUDA in particular has become popular for general-purpose programming on NVIDIA GPUs. The OpenCL [4] specification has been released by a group of vendors led by the Khronos organization [1].

Moreover, a variety of high level programming directives for accelerators are

available or are undergoing development. CAPS HMPP [35], PGI accelerator directives [5], and HiCUDA [43] target CUDA and OpenCL [4]. RapidMind [6] defines C++ extensions that allow its users to describe how data in a C++ application should be mapped between GPUs, cell processors, and cores. However their approach requires redefinition of data types and the creation of kernels with a special syntax language. It is important to note that while these approaches provide portability at the language / directive level, the program optimizations and porting strategies required to apply them depend heavily on the target architecture and the application input set. The OpenACC Application Program Interface, which is released on 2011, describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs, and accelerators [66].

2.3 A Comparison of Directive-based Programming Approaches

CUDA (and OpenCL) require the application developer to carefully study all the salient details of the target architecture. The process of code adaptation and tuning may be lengthy, involving significant reorganization of code and data, and is moreover error-prone. Porting the resulting code to another GPU (e.g. a successor model) may require non-trivial modification. High-level programming models have the potential to simplify the program creation and maintenance effort and may potentially result in

portable code. There have been few studies [58] that compare different vendors GPU directives implementations, their overall features, compare what sort of application changes are needed to get good performance, and the benefits of using directives versus native CUDA/OpenCL or OpenMP.

2.3.1 Overview of the HMPP and PGI Accelerator Directives

HMPP is a directive-based programming interface for hybrid multicore parallel programming that aims to free the application developer from the need to code in a hardware-dependent manner. It is implemented by a source-to-source compiler designed to extract maximal data parallelism from C and FORTRAN kernels and translate them into Nvidia CUDA or OpenCL. The main concepts of HMPP are the codelet and the callsite. A function that can be executed remotely on an accelerator is identified by the codelet directive; the callsite is the place for launching the codelet (kernel) function call. HMPP has both synchronous and asynchronous modes for the codelet remote procedure calls (RPCs). The asynchronous mode enables the overlapping of data transfers between the host and accelerators with other work. The programmer specifies targets for the execution of codelets. If the desired accelerator is present and available, it will run there. Otherwise the native host version is run.

PGI's accelerator directives may be incrementally inserted into a code to designate a portion of C or Fortran code to be run on CUDA-enabled Nvidia GPUs. They enable the application developer to specify regions for potential acceleration,

to manage the necessary data transfers between the host and accelerator, and to initialize and shut down the accelerator. They further provide guidance to the implementation to help it perform data scoping, mapping of loops, and transformations for performance. The directives assume that it is the host that handles the memory allocation on the device, initiates data transfers, sends the kernel to the device, waits for completion, and transfers the results back from the device. The host is also responsible for queuing kernels for execution on the device.

The PGI directives include the kernel region declaration *#pragma acc* with *copyin*, *local*, and *copyout* clauses to specify the input data, local data, and output data of the kernel. PGI also supports the autoscoping of these data automatically. Directives *#pragma acc for parallel(M)* and *#pragma acc for vector(N)* are used to help compiler identify parallel loops and how they should be mapped to the GPU. The compiler also attempts to associate a loop nest's iterations to grid and block-sizes that map to the GPU. The grid sizes will depend on the amount of work launched in the kernel, while the thread-block size remains constant. PGI has been developing new directives such as the *#pragma acc region* and *#pragma acc* declaration directive to scope variables that should shared among kernels and reside in the GPU or CPU or both.

HMPP uses the concept of groups, *#pragma hmpp group <groupid >, target=CUDA*, to specify codelets that will run in the same accelerator and that share data. HMPP provides the codelet *#pragma hmpp <groupid ><codeletid >codelet* and callsite directive *#pragma hmpp <groupid ><codeletid >* to specify codelets and where to invoke them. In addition, these directive have clauses to specify the input and output

data and their sizes in the format *args[A1]=size* and *args[A1].io=in, args[An].io=inout*. HMPP provides asynchronous advanced load and delegate store directive to control when to move data to and from the accelerator. The directive *#pragma hmppcg parallel* indicates that the following loop is parallel and can be mapped to the GPU, while *#pragma hmppcg noParallel* indicates that a loop can not be parallelized. HMPP allows the user to explicitly define the thread-block size of a loop nest by using the *#pragma hmppcg grid blocksize NxN* directive. The *#pragma hmpp <groupid >resident* specifies data that should be allocated in the GPU and that may be shared among codelets.

2.4 Adapting Programs for GPUs: Two Case Studies

In this section, we present our experimental results that consists of adapting two applications kernels to run on GPU-based platform. For both kernels, we use the PGI and HMPP accelerator directive-based programming to accelerate the kernels. Then we describe the transformation we used to get good performance when comparing it against CUDA and OpenMP. Our experiments were run on an Nvidia Tesla C2070 GPU with 448 cores in 14 Streaming Multiprocessors with frequency of 1.15 GHz. The GPU has 6GB DDR5 global memory shared by all threads. The local memory is 64K in size, and can be split 16K/48K or 48K/16K between L1 cache and shared memory. Shared memory for each Streaming Multiprocessor is accessible only within a thread block. The Tesla C2070 is also equipped with an L2 cache (768KB in size for

a 512-core chip). The L2 cache covers GPU local DRAM as well as system memory.

2.4.1 S3D Thermodynamics Kernel

S3D is a parallel combustion flow solver for the direct numerical simulation of turbulent combustion. S3D [27] solves fully compressible Navier-Stokes, total energy, species, and mass conservation equations coupled with detailed chemistry. The governing equations are supplemented with additional constitutive relations, such as the ideal gas equation of state, models for chemical reaction rates, molecular transport, and thermodynamic properties. These relations and detailed chemical properties are implemented as kernels or community-standard libraries that are amenable to acceleration through GPU computing. For this work, we chose the thermodynamics kernel that evaluates the mixture-specific heat, enthalpy, and Gibbs functions as a temperature polynomial. The coefficients of the thermodynamic polynomials and their relevant temperature ranges are obtained from thermodynamic databases following the conventions used in the NASA Chemical Equilibrium code. The thermodynamic kernel with small variations is applicable across a wide range of reacting flow applications.

Figure 2.1(a) shows the most time consuming portion of the serial kernel, where a double nested loop contains an *if* statement. The serial version takes about 22 seconds to execute in a CPU core. We decided to parallelize the outerloop with OpenMP as shown in Figure 2.1(b). We noticed that the inner loop was not being vectorized

because of the *if* conditional. To further optimize the code, we hoisted the *if* conditional by precomputing the branch values in a separate loop that was also parallelized with OpenMP. As a result, we merge the if and else computations into single statements that were masked with the precomputed branch result. Figure 2.1(c) shows the transformation applied. By doing so, we were able to parallelize and vectorize the computational loop which yielded a good speed up 3.8x when running the code on four codes. When running the code on twelve cores, the original OpenMP version in Figure 2.1(b) yielded the best of performance 9.5x because because it does not have any shared memory contention on the masked branch variable.

Our first attempt to accelerate the code with PGI and HMPP directives, by inserting a *!$ acc region* directive and creating a HMPP codelet for the main computational loopnest yielded very little performance for PGI and HMPP directives (2x speedup for PGI and 1.2 speedup for HMPP). By using the CUDA Profiler from Nvidia, we observed that most of the time spent in the accelerated kernels was on data transfer between the CPU and GPU. Since the kernel contains read-only arrays, we optimized the accelerated kernels by allocating and initializing the read-only variables inside the GPU. To do so, we had to inline the main computational kernel loop (loop i) to the procedure that was invoking it within its loop j. Figure 2.2(a) shows the corresponding code transformation implemented in PGI, which uses a data region data region to define the data that resides in the GPU. For HMPP, we used the *group* and *resident* directive to allocate data in the GPU and share data among the codelets of the same group. Figure 2.2(b) shows the HMPP implementation, where codelets *s3d_mixenth* and *s3d_mixcp* belong the same group, named cudagroup. Arrays *Rsp,*

midtemp, coeffhig, and *coefflow* are declared as resident variables, which are used to make them accessible by the two codelets defined in the HMPP group. In order to optimize the data transfers, we used the *advancedload* directive to transfer the data used to initialize the read only GPU variables one time before the first codelet *calc_mixenth* is executed. We also used the *advancedload* clause of the HMPP *callsite* directive to notify HMPP that the read only data is available in the GPU for the second codelet.

S3D Thermodynamics Timings (Seconds)	
SERIAL	21.926
HMPP	0.363
HMPP Kernel	0.3192948
HMPP Data Transfer	0.042834
PGI	0.346305
PGI Kernel	0.320225
PGI Data Transfer	0.02608
CUDA	0.29
CUDA Kernel	0.269265
CUDA Data Transfer	0.019952
OpenMP 12 Threads (best)	2.274

Table 2.1: S3D thermodynamics timing table

By comparing the results of the different parallelization and acceleration methods, we found that the HMPP and PGI implementations produced a 60 and 63 times speedup, respectively. The native CUDA implementation produced a speedup of 76 times that amount, while the OpenMP version using twelve threads produced a speedup of 10, as shown in Figure 2.3. The timings of our experiments are shown in Table 2.1. Our results show that by managing the data correctly, we were able to produce good speed-ups with the PGI and HMPP accelerator directives, within 80%

of the native CUDA performance.

2.4.2 HOMME/SE Application

The **H**igh-**O**rder **M**ulti-scale **M**odelling **E**nvironment application, or HOMME, is one of the highly promising frameworks for integrating the atmospheric primitive equations in spherical geometry. HOMME applies a spectral element method to conserve both mass and energy using a isotropic hyper-viscosity term. To discretize the horizontal dimension, it uses a cubed-sphere grid and in the radial direction a vertical dimension. The HOMME application consists of several hundred Fortran 90 subroutines where the computations are spread evenly across them and whose relevance depends on the input problem.

For each of the spherical elements in the grid, HOMME maintains a global data structure that stores the state of elements, including velocity, temperature, pressure, divergence, and geo-potential. Figure 2.4(a) shows a code fragment of the subroutine *compute_and_apply_rhs* which is one of the routines that computes the divergence for each of the cubed elements. The *ie* loop iterates over the spherical elements, the q loop over the advected physics, the k loop iterates over the vertical radial grid points, and the j and i loops iterate over the horizontal plane grid points.

In HOMME, coarse-grain parallelism is implemented via MPI by distributing the spherical elements across nodes, whereby one or more elements can be assigned to an MPI process (see *ie* loop). In our case we assumed that each node will be assigned twelve elements to provide enough work for all the cores for in-node optimization or

acceleration (i.e one element per core). The in-node problem size used was: $ie = 12$, $qsize\_d = 101$, $nlev = 26$ and $nv,np = 4$. We optimized several version of the kernel for OpenMP, PGI Accelerator directives, and HMPP. We then compared them against the original serial version and the CUDA implementation tuned by NVIDIA and ORNL.

For the OpenMP version, see Figure 2.4(b). We parallelized the *ie* loop with *OpenMP parallel do* to assign spherical elements to OpenMP threads and take advantage of the node's shared memory. One of the challenges when porting the code to OpenMP is to make sure memory access are consistent, by always accessing the same spherical element with the same thread including the data initialization loops. This improves locality by placing element's data in the core's local memory. We also must determine whether to privatize variables such as *gradQ*, temporary variable that gathers data that is passed to the procedure or inline the procedure to avoid unnecessary data copies. For the inner loops we need to make sure loops get vectorized, if possible. When running the OpenMP kernel we noticed that using 4 threads gives the best performance with 510 milliseconds and a speed-up of 2.67. The sequential version takes 1366 milliseconds.

For the PGI and HMPP implementations of the kernel, it was necessary to inline the procedure *divergence_sphere* to map data parallel loops to a GPU and provide sufficient work. Also, this step is necessary if we want to accelerate the kernel at the *ie* loop, using the same approach adopted by the OpenMP implementation.

With the PGI accelerator directives, we needed to do some code restructuring to achieve good performance. We inlined the procedure *divergence_sphere* and inserted

a *!$acc region* to accelerate the *ie* loop. This was necessary for the PGI directives, since it cannot handle function calls inside an accelerated region. The next step was to specify how to parallelize the loop nest iterations across the GPU Symmetric Multi-processors (SM) and within the SMs efficiently. We use the parallel and vector clause to specify the vector size and grid size: in this case we specified a block size of nvxnvxnv. To avoid non-coalesced memory accesses, we eliminated a temporary array *gv* and fused the inner loops. We also allocated and initialized all the data inside the GPU by using the PGI data region directive and *copyout* directive to obtain the results of the kernel. To achieve good performance, we allocated and initialized the twelve spectral elements state on the GPU. Figure 2.5(a) shows the implementation of the kernel using the PGI accelerator directives.

HOMME/SE Timings (Miliseconds)	
SERIAL	1366.37
HMPP Kernel	224.73
PGI Kernel	137.43
CUDA	70.00
OpenMP 4 Threads (best)	510.62

Table 2.2: HOMME/SE timing table

We used a similar code transformation to implement the kernel with HMPP directives. With HMPP we had to outline the *ie* loop to a separate procedure to create a *codelet*. We also had to transform the loops by collapsing the *ie* and *q* loops with the *l* and the *e* loop respectively, to provide enough work for a two-dimensional thread block (At the time of writing, HMPP 2.5.0 only supported two dimensional thread blocks). Figure 2.5(b) shows the HMPP implementation of the kernel. Table 2.2 lists the timing for using OpenMP(with 4 threads), HMPP, PGI,

and CUDA compared with the serial version. According to Figure 2.6, it shows that the OpenMP version (with 4 threads) achieves a speed-up of 2.67 (over the serial version). Without counting the data transfer time, the GPU implementations achieve a speed up of 9.9x for the (ACC) PGI directives, 12.92x for HMPP, and 19.5x for the CUDA implementation.

In order to get good performance when porting those two scientific kernels to GPU, we used Open64, HMPP, and PGI compilers to help us analyze the code. The compiler is able to dump all the optimization information, and that information is quite helpful, as it is able to tell us the loop parallelization and vectorization information of the code. This is illustrated in Figure 2.7, which lists the dumped information by using PGI compiler. With this information, we can properly set up the parallelism among the threads blocks and also parallelism for threads inside each thread block. Additionally, the compiler is able to help us check the data transfer between CPU and GPU to make sure to data transferred to the device properly.

2.5 Summary of Code Porting to GPUs

In this porting experience, we explore different GPU programming models and compare the use of two sets of accelerator directives in two real-world application kernel studies. We explain the porting challenges and limitations encountered, and demonstrate how we could use available tools and compilers to help us to analyze the code for optimization. Specifically, we show how much effort is needed for restructure the code in order to make the optimization meet the GPU architecture needs.

Most importantly, we evaluated the porting difficulty for porting code to the heterogeneous environment. During the porting process, we found there are a lot of scientific application kernels that are quite structurally similar to each other; the optimization experience we get can be reused for similar kernels. In our experiment, We compare the performance of running the codes on the GPU versus the CPU, and found that in all the cases the GPU yielded significantly better performance. In order to use the accelerator directives efficiently, it is necessary to perform code transformations to close the gap in performance to native CUDA implementations. Purely replying on manual directive insertion would make the porting very challenging for a large scientific application porting. There is an urgent need of a porting tool which could give the user some guidance to automate the porting process.

```
do i = 1, np
  enth(i) = 0.0
  do m = 1, nslvs
  if(temp(i)<midtemp(m)) then
    enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coefflow(6, m)+temp(i)*(&
                  . . .
            coefflow(5, m)*rp05)))) )
  else
    enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coeffhig(6, m)+temp(i)*(&
                  . . .
            coeffhig(5, m)*rp05)))) )
    end if
  end do
end do
```

(a)S3D Thermodynamics Serial

```
!$OMP parallel do private(i, m, enth)
do i = 1, np
  enth(i) = 0.0
  do m = 1, nslvs
    if(temp(i)<midtemp(m))then
     enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coefflow(6,m)+temp(i)* (&
                . . .
            coefflow(5,m)*rp05))))))
    else
      enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
            coeffhig(6,m)+temp(i)*(&
                . . .
            coeffhig(4,m)*rp04+temp(i)*(&
            coeffhig(5,m)*rp05))))))
    end if
  end do
end do
!$OMP end parallel do
```

(b)S3D Thermodynamics OpenMP

```
 . . .

!$OMP parallel private(i,m,flag_hig,flag_low)
do m = 1, nslvs
  !$OMP do
  do i = 1, np
    if(temp(i)<midtemp(m)) then
         flag_low(i, m)=1
         flag_hig(i, m)=0
    else
         flag_low(i, m)=0
         flag_hig(i, m)=1
    endif
  enddo
!$OMP end do nowait
enddo
```

```
!$OMP do
do i = 1, np
enth(i) = 0.0
  do m = 1, nslvs
   enth(i)=flag_low(i,m)*(enth(i)+yspec(i,m)*&
           Rsp(m)*(coefflow(6,m)+ temp(i)*(&
                   . . .
           coefflow(5, m)*rp05)))) ))+&
           flag_hig(i,m)*(enth(i)+yspec(i,m)*&
           Rsp(m)*(coeffhig(6,m)+temp(i)*(&
                   . . .
           coeffhig(5, m)*rp05)))) ))
  end do
end do
!$OMP end do nowait
!$OMP end parallel

 . . .
```

(c)Optimized S3D Thermodynamics with OpenMP

Figure 2.1: S3D thermodynamics kernel OpenMP code snippets

```
!$acc data region copyin(temp,...),&
   copyout(enth)
do j = 1, MR
!$acc region
!$acc do parallel(np)
  do i = 1, np
    enth(i) = 0.0
    do m = 1, nslvs
      if(temp(i)<midtemp(m)) then
        enth(i)=enth(i)+yspec(i,m)*&
          Rsp(m)*(&
          ...
        coefflow(5, m)*rp05)))))
      else
        enth(i)=enth(i)+yspec(i,m)*&
          Rsp(m)*(&
          ...
        coeffhig(5, m)*rp05))))))
      end if
    end do
  end do
!$acc end region
!$acc region
!$acc do parallel(np)
  do i = 1, np
    cp(i) = 0.0
    do m = 1, nslvs
      ...
    end do
  end do
!$acc end region
end do
!$acc end data region
```

(a) PGI

```
!$hmpp <cudagroup> group, target=CUDA
!$hmpp <cudagroup> resident, args[Rsp].io=in
real,parameter::Rsp(1:nstts)=Ru/molwgt(1:nstts)
!$hmpp <cudagroup> resident, args[midtemp].io=in
real,parameter::midtemp(68)=(/ ... /)
!$hmpp <cudagroup> resident,args[coeffhig].io=in
real,parameter::coeffhig(7,68)=reshape(/.../)

subroutine calc_mixenth(np, ... ,cp)
implicit none
. . .
!$hmpp <cudagroup> allocate
!$hmpp <cudagroup> s3d_mixenth advancedload,&
       args[::Rsp; ...; ::coeffhig]
!$hmpp <cudagroup> s3d_mixenth callsite
call hmpp_kernel1(np, ... , coeffhig)
!$hmpp <cudagroup> s3d_mixcp callsite, &
       arg[::Rsp; ...].advancedload=true
call hmpp_kernel2(np, temp, ... , coeffhig)
!$hmpp <cudagroup> release
end subroutine calc_mixenth

!$hmpp <cudagroup> s3d_mixenth codelet, &
    args[np;...;yspec].io=in,args[enth].io=out
subroutine hmpp_kernel1(np,temp,...,coeffhig)
    ...
end subroutine hmpp_kernel1
!$hmpp <cudagroup> s3d_mixcp codelet, &
      args[np;...;yspec].io=in,args[cp].io=out
subroutine hmpp_kernel2(np,...,coeffhig)
    ...
end subroutine hmpp_kernel2
```

(b) HMPP

Figure 2.2: S3D thermodynamics kernel PGI and HMPP code snippets

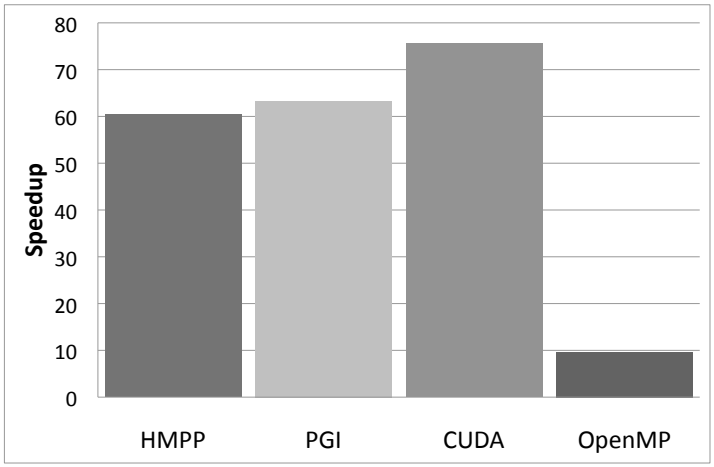


Figure 2.3: S3D thermodynamics speedup

```
  ...

do ie=nets, nete
  do q=1,qsize
    do k=1,nlev
      gradQ5d(:,:,k,q,1)=
      elem(ie)%state%v(:,:,1,k,n0)*&
      elem(ie)%state%Qdp(:,:,k,q,n0)

      gradQ5d(:,:,k,q,2)=
      elem(ie)%state%v(:,:,2,k,n0)* &
      elem(ie)%state%Qdp(:,:,k,q,n0)
    end do
  end do

  divdp4d(:,:,:,:) =
         divergence_sphere5d( &
         gradQ5d(:,:,:,:,:), &
         deriv, elem(ie))

    ...

end do
```

(a) Serial code

```
 ...
!$omp parallel private(ie,j,i,k,q,m,l,&
!$omp& gradQ5d,divdp4d,deriv)
 ...
!$omp do
  do ie=nets, nete
   do q=1,qsize
     do k=1,nlev
       gradQ5d(:,:,k,q,1)=
       elem(ie)%state%v(:,:,1,k,n0)*&
        elem(ie)%state%Qdp(:,:,k,q,n0)
        gradQ5d(:,:,k,q,2)=
        elem(ie)%state%v(:,:,2,k,n0)* &
        elem(ie)%state%Qdp(:,:,k,q,n0)
      end do
    end do

    divdp4d(:,:,:,:) =
           divergence_sphere5d( &
           gradQ5d(:,:,:,:,:), &
           deriv, elem(ie) )
   ...
  end do
!$omp enddo nowait
!$omp end parallel region
```

(b) OpenMP code

Figure 2.4: The original serial and OpenMP divergence_sphere code snippets

```
!$acc region
!$acc do parallel(nete)
  do ie=nets, nete
!$acc do parallel(qsize)
   do q=1,qsize
!$acc do vector(32)
   do k=1,nlev
!$acc do vector(nv)
    do j=1,nv
!$acc do vector(nv) private(dudx00,dvdy00i)
     do l=1,nv
      dudx00=0.0d0
      dvdy00i=0.0d0
      do i=1,nv
       dudx00 = dudx00 + Dvv(i,l  ) * &
       (metdet(i,j,ie)*(Dinv(1,1,i,j,ie)* &
        gradQ5da(i,j,k,q,1,ie) + &
        Dinv(1,2,i,j,ie)*gradQ5da(i,j,k,q,2,ie)))

       dvdy00i = dvdy00i + Dvv(i,j  ) * &
       (metdet(l,i,ie)*(Dinv(2,1,l,i,ie)* &
       gradQ5da(l,i,k,q,1,ie) + &
       Dinv(2,2,l,i,ie)*gradQ5da(l,i,k,q,2,ie)))
      end do
      divdp4da(l,j,k,q,ie)= &
               rmetdetp(l,j,ie) * &
              (rdx(ie))*dudx00+(rdy(ie))*dvdy00i
       end do
      end do
     end do
    end do
   enddo
!$acc end region
```

(a) PGI Accelerator Directives

```
!$hmppcg grid blocksize 4x4
!$hmppcg parallel
  do i2=nets, nete*nlev ! ie, q
!$hmppcg parallel
   do i1=1, qsize*nv  ! q, j
!$hmppcg set b2 = BlockId(i2)
!$hmppcg set t2 = RankInBlock(i2)
!$hmppcg set b1 = BlockId(i1)
!$hmppcg set t1 = RankInBlock(i1)
   ie=b2+1
   q = b1+1
   k =t2 +1
   j =t1+1
     do l=1, nv
       dudx00=0.0d0
       dvdy00i=0.0d0
       do i=1,nv
          dudx00 = dudx00 + Dvv(i,l  ) * &
          (metdet(i,j,ie)*(Dinv(1,1,i,j,ie)* &
          gradQ5da(i,j,k,q,1,ie) + &
          Dinv(1,2,i,j,ie)*gradQ5da(i,j,k,q,2,ie)))

          dvdy00i = dvdy00i + Dvv(i,j  ) * &
          (metdet(l,i,ie)*(Dinv(2,1,l,i,ie)* &
          gradQ5da(l,i,k,q,1,ie) + &
          Dinv(2,2,l,i,ie)*gradQ5da(l,i,k,q,2,ie)))
        end do
      divdp4da(l,j,k,q,ie)= rmetdetp(l,j,ie)
             * (rdx(ie)*dudx00+rdy(ie)*dvdy00i)
     enddo
    enddo
   enddo
end subroutine
```

(b) HMPP Implementation

Figure 2.5: The inlined and accelerated divergence_sphere code snippet

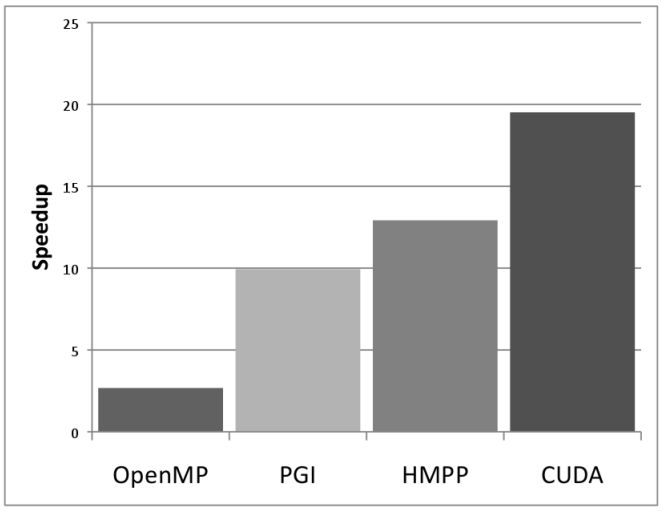


Figure 2.6: HOMME/SE divergence sphere speedup

S3D: Thermo Dynamics Kernel (PGI)

```
$acc data region copyin(temp,...) copyout(enth)
…
!$acc region do parallel
 do i = 1, np
  enth(i) = 0.0
  do m = 1, nslvs
   if(temp(i)<midtemp(m)) then
    enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
      ...
    coefflow(5, m)*rp05)))))
   else
    enth(i)=enth(i)+yspec(i,m)*Rsp(m)*(&
      ...
    coeffhig(5, m)*rp05))))))
   end if
  end do
 end do
!$acc end region do
…
!$acc end data region
```

Compiler Output:

```
calc_mixenth:
   38, Generating copyout(cp(:))
       Generating copyout(enth(:))
       Generating copyin(yspec(:,:))
       Generating copyin(temp(:))
   39, Loop not vectorized/parallelized: contains call
   40, Generating copyin(midtemp$ac(1:52))
       Generating copyin(coefflow$ac(1:6,1:52))
       Generating copyin(rsp$ac(1:52))
       Generating copyin(coeffhig$ac(1:6,1:52))
   42, Loop is parallelizable
       Accelerator kernel generated
       42, !$acc do parallel, vector(256)
           Using register for 'temp'
           Using register for 'enth'
   44, Loop carried reuse of 'enth' prevents parallelization
       Complex loop carried dependence of 'enth' prevents'
       parallelization
       Inner sequential loop scheduled on accelerator
```

Figure 2.7: S3D PGI compiler dumped information

Chapter 3

Related Work for Tool Support in Software Porting

The hands-on experience on porting two real large scientific application kernels to GPUs motivates us to create a porting tool. This chapter will describe the application porting scenarios in the HPC field in the past decade, and current porting challenges faced by scientists for porting code to some emerging architectures, such as GPUs, MICs, etc. We will describe available tools which can be used for software porting and show how we build on previous similarity research for creating our software porting tool.

3.1 Traditional Application Porting

In the past decade, the process of adapting large-scale DOE applications to a new architecture focused primarily on adapting and tuning MPI-based parallel codes to clusters with new processors and communication characteristics, as well as increasing the amount of exploitable parallelism and reducing bottlenecks. A variety of strategies were employed to support this porting process. Microbenchmarks [40, 60, 42, 75] enabled low-level evaluation of MPI performance on new target platforms. Some systems, such as the first generation Blue Gene/L supercomputer [13], resulted in reduced memory per process necessitating reduction of replicated data and careful memory management in applications. One of the most common approaches used during the porting process is to use profiling tools [69, 33, 79, 38] to analyze the application under the new environment, then to use the profiling information to aid program optimization by removing computation bottleneck.

There has been a long history of research and development in techniques and tools for automatic or semi-automatic parallelization [17, 45, 49, 47]. However, few of these efforts have been helpful for porting HPC codes to new parallel architectures. There are a few reasons why parallelization tools often fail to generate efficient parallel code. They are unable to reproduce expert knowledge about the available parallelism that may be exploited within the code. As a result, the parallelization strategy they apply is often too conservative to meet the performance requirements, necessitating human effort to manually restructure and tune the codes. In addition, users may program at a lower level, such as assembly language or accelerator-specific languages

for better control of code generation (this can be useful, as evidenced by projects such as Goto BLAS [29]). However, these codes are typically too low-level and architecture-specific, making them difficult to port to new architectures. Because of the limitations of parallelization tools, typically an iterative manual approach has been used, where the user parallelizes the code using explicit (e.g. Pthreads) or directive-based (e.g. OpenMP) threading APIs , evaluates the performance of the resulting code, and repeats until a satisfactory performance is reached for the target.

In addition to the issue of parallelization, the challenge of porting HPC codes to new parallel architectures also depends on identifying a suitable compiler optimization strategy. Compilation flags can be used to tune performance for a target platform; however, because the optimizations selected by these flags are typically globally applied, this can result in uncertainy over why they may or may not be effective. Some compilers support directives for applying optimizations to specified code sections [39], giving more fine-grained control to the user. But they provide only limited tuning options and are not portable across compilers. Source-to-source translators [72, 30, 86, 68, 71, 2] can apply local restructuring to source code, but they fail to give access to many optimizations available within the compilation process, and there is no guarantee that the generated code restructuring will not be undone by the target compiler.

3.2 State-of-the-art

In the HPC field nowadays, hardware platforms are currently undergoing drastic changes, as scientists introduced Graphic Process Units (GPUs), MICs, and other co-processors for computation. The traditional homogeneous system is evolving to heterogeneous architectures. The architecture changes require significant code reorganization in order to provide much higher data locality and to map the computations and data to different kinds of devices configured within the node. It is a challenge to make changes to the code and port the scientific applications to these newer hardware platforms. Newer platforms have features that include faster CPUs with multi-cores and improved larger space DRAM access, bigger hard drives, etc. For example, many platforms have GPUs enabled for their computing nodes. Since the GPU can easily generate hundreds or thousands of light-weighted threads due to its intrinsic hardware structure, more and more applications are being ported to it. Porting and optimizing codes for these newer and more powerful parallel platforms require restructuring code, identifying parallelism, optimizing loops and data structures, managing data locality, adding or rewriting code in new languages, etc.

For example, the 20 Petaflop (peak) Titan supercomputer, which has been deployed at Oak Ridge National Laboratory, is composed of Cray XK6 nodes with 18,000 Nvidia Tesla GPUs. CPUs and GPUs have their own local memories instead of a physically shared memory, requiring shared data to be explicitly transferred between the two. Porting an MPI or hybrid MPI+OpenMP application to such heterogeneous platforms requires identifying multiple levels of parallelism, optimizing

loops and data structures, and potentially reorganizing data accesses across wide areas of code in order to avoid large transfers of data between the CPU and GPU. Moreover, the portions of code that will be accelerated on a GPU must be rewritten: there are several programming interfaces that may be used to create GPU code, and the appropriate choice may not be easy. Some researchers summarize their early experience on using Titan [53, 54], but they only describe how to use OpenACC [66], a directive-based programming model, to port code to GPU and how the hybridization technique is able to convert an application with a single level of parallelism to an application with multiple levels of parallelism. No porting tools were used during their porting process. They mainly rely on profiling tools to locate computation hotspots and then manually insert directives for porting code to the GPU. Finally they manually tune the code based on the profiling information provided by the Nvidia Visual Profiler [28].

Hardware platforms may undergo dramatic changes, requiring significant code reorganization in order to provide much higher data locality and to ensure that efficiency is guaranteed after porting to the newer platform. Obviously, originally developed tools and methodologies used for homogeneous environments can not meet the new porting requirements today. Purely relying on scientists to manually migrate applications to the heterogeneous system is expensive and error-prone. The computer science research community recognizes these challenges and have been working towards providing comprehensive solutions. New tools are needed in order to ease the porting process.

3.3 Tools Support for Application Porting

Although the computer science research community has recognized these porting challenges, little attention has been given to deriving a detailed process for software porting to new platforms. Tools such as CHiLL [26] and POET [90] can be used to apply optimizing transformations, but these tools depend on manually derived transformation scripts which still need user involvement. Also, they lack features for identifying code regions to which these transformations can be applied. Considering the size of scientific applications, it is a non-trivial job to locate those code regions and derive scripts for each of them. Other available tools used for application porting can be generally classified into two groups, software maintenance and similarity analysis, according to their attributes. We will discuss the relevant ones and discuss the differences between those studies and our work.

Software porting is one of the most common maintenance activities, but little attention has been given to deriving a detailed process and planning support for it until recently. Varma et al. described how eXtreme Programming (XP) practices could provide a detailed process for quality porting [83, 82]. XP is a software development methodology which advocates frequent releases of a working software in short development cycles to improve productivity in response to changing customer requirements. Much of the work in this area begins to focus on various ways to automate the iterative and incremental process, with some emphasis on acceptance testing [12]. Hennessy's and Power's work in [44] is the most relevant to ours. In their porting strategy, one C++ class is ported at a time. An order for classes is

defined based on an object relation diagram, whereas our methodology is based on similarity.

Some of the work in this area addresses the problem through pattern matching, by which we mean matching code either by coming up with syntactic measures or by giving guidance via identification of methods or underlying computation (e.g. matrix multiplication [56, 19]). TSF [18] and other work [51] also developed a notion of similarity in order to detect related code regions for the purpose of applying loop transformations. Such analysis typically examines loop nests, their nesting depth, and certain details of the data usage patterns they contained. They rely on a defined "pattern abstractor" to generalize patterns according to a criterion and then to do a pattern matching which operates on the abstract syntax tree (AST). This requires the user to be familiar with the pattern description language. Additionally, there is a no way to measure the degree of similarity. Compared with our methodology, we not only define a metric for the level of similarity between subroutines, but also provide a way to visualize the similarity between all subroutines inside a large application. Additionally, we can trace the code changes in terms of optimization from an evolutionary point of view. Some other problems for using TSF include: 1) it requires users to write transformation script, and TSF only could be applied for Fortran code engineering which cannot be used for C/C++; 2) their implementation is based on FORESYS system, which cannot be extended to other systems; and 3) there are no recipes described or explored for accelerators such as GPUs. The similarity methodology used in our "*Klonos*" tool can be quickly implemented by other compilers, such as GCC, without the need for users to write transformation recipes.

Hercules [50] is another code transformation tool which is an on-going project at ORNL. This tool relies on a transformation recipe and a compiler plug-in infrastructure to apply transformations at compile time. Although early evaluations of Hercules suggest that the pattern matching approach is feasible on today's computer resources, the task of defining patterns may become daunting to the users. Therefore, a tool for assisting the user on this pattern creation based on "similar code" is needed. Also, automatic software porting via pattern-matching requires us to solve a *search* problem. Thus, a pattern language is needed to specify a query [16, 74]. Our similarity-based methodology does not require that the pattern be specified explicitly.

Our work is also related to studies on software evolution [89]. These studies target different releases of an individual subroutine (or program), whereas our family distance tree represents similarity among all subroutines in a specific release. Compared with this work, we applied the similarity analysis more broadly for each subroutine of a large scientific application. Most importantly, our similarity is stricter than pure syntactic similarity since our similarity also captures the data access pattern.

Similarity is used to compare programs in many contexts. Initially, similarity work was conducted to identify students who copied code from others in a programming assignment, but later this method was adapted for software engineering. However, there was no *precise* definition of similarity between programs [85]. In addition, the definition of similarity often depends on the context in which it is being used [80]. As a result, different types of similarity are defined in the literature. For example, Walenstein et al. [85] considered *representational* similarity and *behaviorial*

similarity. Roy et al. [77] proposed four categories of clones: identical code fragments (Type I), syntactically identical code fragments with variation in identifiers, literals (Type II), code fragments that have been further modified, improved or changed (Type III), and code fragments that have the same functionality but that are implemented differently (Type IV). Current research focuses more on representational similarity (Type I–III) because behavioral similarity (Type IV) is more difficult to detect. Our similarity metric falls under representational similarity (Type I–III). Other researchers [52, 76] classified similarity based on strings, tokens, parse trees, program dependence graphs, or a mix of different metrics.

In representational similarity, the term "program" refers to the representation, commonly viewed as a sequence of characters forming a more complex text structure. Similarity can be defined in terms of the form, properties, or characteristics of this representation. The community generally distinguishes between textual and syntactic similarity. Textual similarity considers the program source to be text and analyzes it in the way that general text documents are analyzed [20, 34, 48]. Therefore, it is (programming) language independent. Smith and Horwitz [80] proposed an approach that is largely language-independent, requiring only a language lexer to detect similar codes called clones. Funaro et al. [37] combined both textual and syntactic approaches in clone detection.

Although some of these prior efforts have focused on helping restructure applications by detecting code clones at the syntactic level, few have aimed to detect clones, or similar code regions, that can potentially be optimized in the same way for a given architecture. Given a code portion that has benefited from a specific

optimization strategy, our goal is to determine other parts of the code that exhibit not just a certain level of syntactic similarity but also similarity with respect to code optimization characteristics. We ultimately also need to develop effective porting strategies to automate the task of restructuring codes to exploit the capabilities of emerging architectures.

What applications people really need are strategies that can automatically identify code pieces that have benefited from a previous specific optimization/transformation technique. Then, once these have been identified, they need a method to easily apply the same strategies to other code subroutines that exhibit certain levels of similarity. They also need effective porting strategies to automate the task of restructuring codes to exploit the capabilities of emerging architectures. We note that the cost of adapting these strategies to a new environment strategies should be less than the cost of re-development.

Chapter 4

Design of *Klonos* Tool

After exploring the directive-based programming models for porting code to a heterogeneous programming environment, we realized the software porting is a time-consuming and error-prone process. State-of-the-art shows relevant tools for software porting are rare and outdated, and cannot meet the current need in the HPC field. In order to ease the porting process and increase porting productivity, we have designed and created a tool, called "*Klonos*", which is used for software porting. In this chapter, we will describe the framework and key components of the *Klonos* tool. This tool adapts an existing compiler, OpenUH, to detect similar codes based on collected static code information and other code analysis. The users only need to submit an application source code to *Klonos*. The tool then does the code pattern extraction first and then analyzes the code sequences. After that, it will provide a plan for porting the application to a target system.

4.1 *Klonos* Framework

In this section, we formally introduce the similarity-based methodology and present a prototype implementation of *Klonos* to carry out the similarity analysis. For better understanding and clarity, we have defined some terms in Table 4.1 that are used in the rest of this dissertation.

Table 4.1: Terminology used in *Klonos*

Term	**Definition**
Similarity score	A score of percentage which is used to describe percent of identities of a pair of sequences.
Distance	A metric used for evaluating the dissimilarities of a pair of sequences. Distance = 100 - (similarity_score) * 100
Similarity distance matrix	A matrix (two-dimensional array) containing the distances, taken pairwise, of a set of subroutines. This matrix has a size of N×N, where N is the number of subroutines.
Family distance tree	A tree structure which is constructed based on the similarity distance matrix. Inside the tree, subroutines with similar code structure will be grouped into one sub-tree.
Porting strategy	A solution for adapting a program to a different or new platform while guaranteeing program correctness and efficiency.
Similar porting group	A set of subroutines, which fall into the same static and dynamic clustering, can be ported with the same porting strategy. Generally, one subroutine will be selected as a representative porting example, the porting strategy used for the rest of the subroutines inside the same group will refer to the representative subroutine.
Porting clustering	A group of clusters with subroutines in each cluster share with the same static and dynamic clustering.
Porting planning	A process of making plans for deciding the porting orders among the porting groups to a new platform in order to reuse porting strategies as much as possible.

We formulate the problem of *planning* in a software porting process as follows: a scientific application, consisting of many subroutines, is selected to be ported and optimized to a new architecture. The personnel responsible for carrying out this porting will need to make a plan on *which* subroutine in the application to port first and which subroutine is next. In other words, it is an optimization problem that attempts to create an order for all subroutines in a scientific application to facilitate

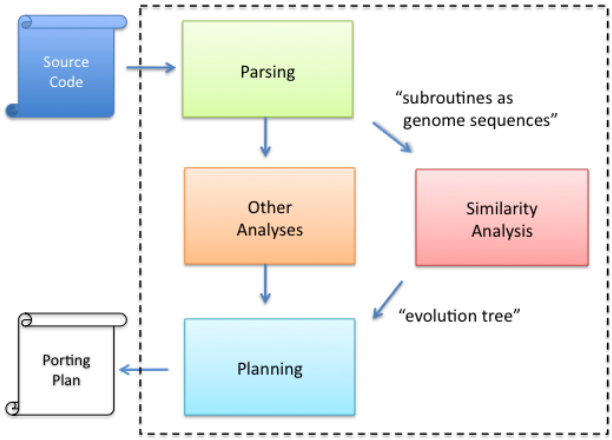


Figure 4.1: The proposed methodology.

productive porting.

The proposed methodology addresses the planning problem. It is based on a hypothesis that similar subroutines can be ported and optimized with similar strategies and result in a similar-quality porting. At a high level, the methodology aims to create a tree with subroutines as leaves based on similarity. The tree can provide additional, structural information that enables us to suggest a good porting order (detailed explanation described in Section 5.1). To keep it general, the methodology is not tied with any specific way to generate that order. In some cases, the exposure of the tree by itself already provides valuable information for the end user.

Figure 4.1 shows the four phases of the methodology: parsing, similarity analysis, code analyses, and planning. The first step is parsing. We basically extract individual subroutines from the application and transform them into a representation suitable for similarity analysis and other analyses. In the similarity analysis phase, the subroutines are analyzed based on similarity analysis, and a family distance tree

Operator	Character Map
STORE	E
MULTIPLY	M
IF	I
ADD	A
CALL	C
DO_LOOP	L
ARRAY ACCESS	Y
SUBTRACTION	S
DIVISION	D
FUNCTION_ENTRY	F
WHILE_DO	O
DO_WHILE	W
SWITCH	H
MINUS	R
PARALLEL REGION	P
PARALLEL DO	Q
CONSTANTS 0—8	B,G,J,K,N,T,U,V,X
>=9	Z

Table 4.2: The character map in *Klonos*.

is created at the end of the phase. Other analyses may be required to annotate the tree for further refinements. Finally, we make (or suggest) a porting plan, and that concludes the process of the proposed methodology.

4.2 Code Sequence Extraction

The phase of parsing requires identifying and transforming individual subroutines into representations suitable for similarity and other analyses. We select OpenUH, an open source research compiler suite for C, C++, and Fortran 95 programs, to

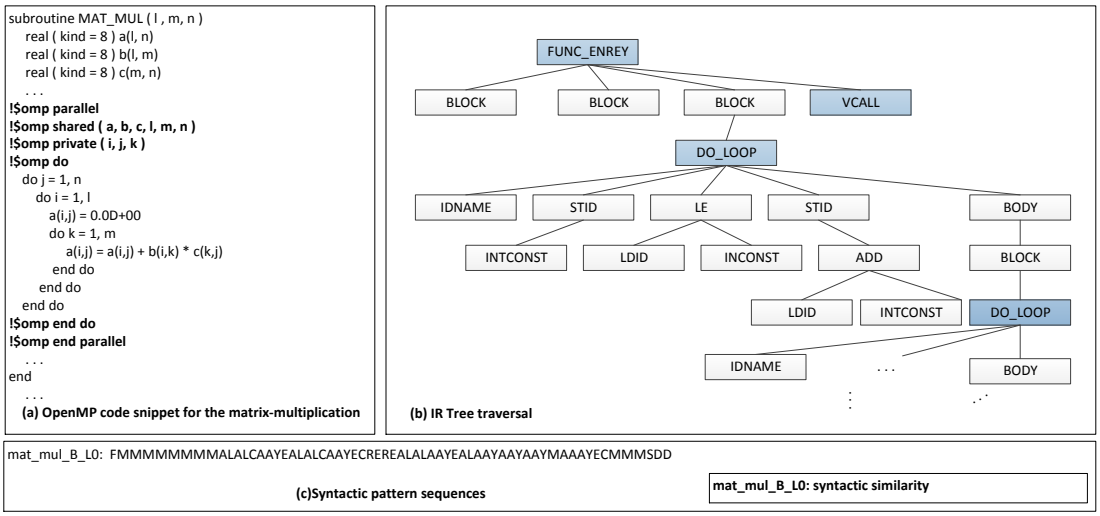


Figure 4.2: The illustration of parsing: An OpenMP code snippet shown in (a) is parsed into an AST shown in (b). The AST is then traversed to generate a string shown in (c).

implement this phase. We use the front end of OpenUH to help encode a subroutine into a text-based string. Specifically, the front end parses the source code of the subroutine into an abstract syntax tree (AST) and also provides a mechanism to traverse tree in post-order. We use the mechanism to traverse the AST for each subroutine. During the traversal, each node visited is mapped to a unique character, denoting the type of node such as branch, data access, or mathematical calculation based on a character map as listed in Table 4.2. During the denoting process, we quickly found that we were running out of the alphabetical letters. In order to keep the sequence short and make each letter inclusively stands for corresponding node, we denote any numbers greater than 9 as 'Z' in the node map. Figure 4.2 illustrates the parsing phase.

4.3 Code Sequence Analysis

Once the parsing is done, each subroutine has an associated character string. Viewing the string as a DNA-like sequence, we can leverage tools in bio-informatics for similarity analysis. In particular, we use Jalview to generate the visual representation of the (global) alignment, Geneious[8], to create the family distance tree, and EMBOSS[7] to calculate pair-wise global alignments and calculate the similarity score. Note that Jalview is also able to construct distant trees, but the visual representation of the tree is not as scalable as that from Geneious. Also note that the tree used in our analysis was based on UPGMA (Unweighted Pair Group Method with Arithmetic Mean), which is used to calculate genetic distance from multiple sequence alignments and is a simple agglomerative or hierarchical clustering method used in bio-informatics for the creation of phonograms. This method does not use an evolutionary model and it is safe to apply to our program sequences.

The similarity score that is calculated for the pair-wise alignment of two program sequences is based on optimal sequence alignment and the scoring system is parametric. The substitution matrix used for the alignment uses the score of 1 for a character match, -1 for a mismatch on all characters used to encode the program AST. For the alignments we used a score of -6 and -.5 for the gaps and subsequent gap penalties respectively. Once sequences are aligned, we calculate a pair-wise percentage of identity based on the percentage of non-gap positions in the aligned sequences divided by the total length of the aligned sequence. This percentage defines our similarity score. Note that the score is normalized.

We could have compared two ASTs involved directly, but we decided not to because of the complexities. This is because graph comparison is generally a difficult problem to solve and NP-complete. In other words, we also avoided the approaches based on program dependence graphs. By casting each AST into a DNA-like sequence, we take advantage of years of research on dynamic programming which have proved to be fast and scalable. In our experiments, we were able to deal with sequences of an average length of 256 characters and 5,000 characters maximum. In bio-informatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [59]. In our research, we used the sequence alignment to calculate the similarity of a pair of subroutines. There are two types of methods for the sequence alignment: local and global sequence alignment.

4.3.1 Local Sequence Alignment

Local sequence alignment is a technique mainly used to detect local regions of similarity of sequences within their larger sequence context. The Smith-Waterman algorithm is a popular local alignment method based on dynamic programming. This algorithm is normally more biologically useful. Figure 4.3 shows an example of two sequences, initialize_B and initialize_C. Figure 4.4 is the result after using local alignment Smith-Waterman algorithm. In our analysis, we mainly used the global sequence alignment, since the granularity in our analysis is at subroutine level.

```
initialize__B:  FMMADADEGGALGGALRGAMAE

initialize__C:  FMMADALGGALAYEMSMMSMMSMSMMAAYE
```

Figure 4.3: Original sequences initialize_B and initialize_C

```
initialize__B      1 FMMADA      6
                     ||||||
initialize__C      1 FMMADA      6
```

Figure 4.4: Local sequences alignment example when setting gap penalty=6.0, gapextend=0.5

Input: Two sequences x and y of length M and N, respectively; scoring matrix σ(a,b); linear gap cost A.

Output: Dynamic programming matrix F.

Initialization:

F(0,0) =0

for i =1 to M

F(i, 0) =0;

for j =1 to N

F(0, j) =0;

Recursion:

for i =1 to M

for j =1 to N

$$F(i,j) = \max \begin{cases} 0, \\ F(i-1, j-1) + \sigma(x_i, x_j) \\ F(i-1, j) - A \\ F(i, j-1) - A \end{cases}$$

Figure 4.5: Smith-Waterman algorithm pseudo code

Assume we have two sequences, X and Y with length of M and N respectively, and a scoring matrix which is a symmetric diagonal matrix with elements on diagonal equal to 1, and the rest of elements are 0. In other words, we set perfect match score to 1, and value of -1 for mismatch. We also set the gap penalty score to -0.5. In order to align the sequences, we build a dynamic programming matrix with size of M*N. M and N are the length of input sequences X and Y as mentioned before. This algorithm first initializes the scoring matrix to zero, and then keeps updating the scoring matrix based on the calculation of its right, down, and diagonal neighboring points for each letter alignment calculation. Once the scoring matrix is ready, Figure 4.5 lists the Smith-Waterman local sequence alignment pseudo code. Based on the algorithm, we can calculate the dynamic programming matrix at first. To find the optimal local sequence alignment, this algorithm starts from choosing the cell which has the highest score, and backtracks until reaching a cell with score 0.

4.3.2 Global Sequence Alignment

A general global alignment technique is the Needleman-Wunsch dynamic programming algorithm [63], which is used in our sequence alignment. Assume there are two input sequences A and B used for alignment:

```
String A: A B C D
String B: A A C D
```

In order to find the alignment with the highest score, a size of (M+1)×(N+1) two-dimensional array (or matrix) is allocated. M is the length of sequence A, and

N is the length of sequence B. There is one column for each character in sequence A, and one row for each character in sequence B.

$$InitialMatrix = \begin{bmatrix} & - & A & B & C & D \\ - & 0.0 & -0.5 & -1.0 & -1.5 & -2.0 \\ A & -0.5 & 0.0 & 0.0 & 0.0 & 0.0 \\ A & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ B & -1.5 & 0.0 & 0.0 & 0.0 & 0.0 \\ C & -2.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

As illustrated in Figure 4.6, the diagonal edges correspond to character matches or identities. A down edge corresponds to a gap in string B, and an across edge corresponds to a gap in string A.

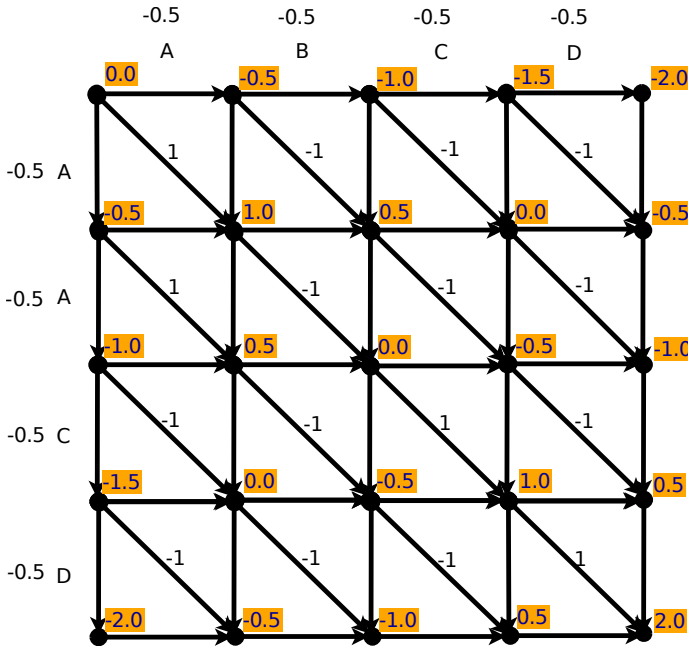


Figure 4.6: Needleman-Wunsch global sequences alignment algorithm

$$AlignmentMatrix = \begin{bmatrix} & - & A & B & C & D \\ - & -0.0 & -0.5 & -1.0 & -1.5 & -2.0 \\ A & -0.5 & 1.0 & 0.5 & 0.0 & -0.5 \\ A & -1.0 & 0.5 & 0.0 & -0.5 & -1.0 \\ B & -1.5 & 0.0 & -0.5 & 1.0 & 0.5 \\ C & -2.0 & -0.5 & -1.0 & 0.5 & 2.0 \end{bmatrix}$$

Global sequence alignment is used to align every residue in every sequence. A gap will be inserted between residues if a mismatch is detected so that identical or similar characters are aligned in the successive columns. A vertical bar between a pair of sequences represents an identity. Figure 4.7 demonstrates an example of a pair of sequence alignments by using the global sequence alignment method. *Alignment_matrix* is the corresponding scoring matrix we have. Global sequence alignment is very similar to the local sequence alignment for the scoring matrix generation, but it will backtrack the scoring matrix until all letters in the sequence have been processed.

```
initialize__A      1 FMMADAD-EGGALGGALRGAMARGAABABABDGAIRGAYEMSMMSMMSMS     49
                     ||||||| ||||||||||||||||   |.|..||||||||||||||||||
initialize__B      1 FMMADADAEGGALGGALRGAMARG---AMAJRGAIRGAYEMSMMSMMSMS     47
```

Figure 4.7: Global sequences alignment example when setting gap penalty=6.0, gapextend=0.5

4.4 Building the Distance Matrix

After extracting the sequences, we performed a pair-wise global alignment to compare the degree of syntactic similarity between the subroutines. For this, we used the Needleman-Wunsch algorithm [62] using the identity substitution matrix. A score is generated for each pair-wise alignment using the value for their percent of identity. A percent of identity of 100% means the sequences are identical. We used the pair-wise comparison similarity score for each pair of sequences to calculate a distance matrix by subtracting the percent of identity with 1, and then times 100.

4.5 Constructing the Family Distance Tree

There are several algorithms [73, 78, 87] that can be used to classify sequences into distance trees. In our case, we used the Neighbor-joining [78] to build our syntactic distance tree for its simplicity and because it is distance matrix based. This algorithm aims to minimize the sum of all branch lengths. It starts by generating the distance matrix from the input of multiple sequence. Then, it contiguously selects two nodes which have the least distance, and replaces them with another new node until all nodes have been consumed. It initially starts from a star-like digram, and gradually expands to a tree while continuously reducing the total length of the tree. This algorithm is heuristic greedy algorithm based. In our experiment, we used a tool called Clearcut [55] to help us generate the family distance tree by using the NJ algorithm, then we used Geneious [8] to read the generated tree file and draw the

family distance tree.

4.6 Building the Optimization Planning Tree

According to the generated family distance tree, sequences with less distance have been clustered into groups. In other words, similar subroutines have been grouped together based on their syntactic similarity. The generated family distance tree can precisely give us the overall code structure relationship over the all subroutines. However relying solely on the syntactic code structure does not enable capture of the similar code optimization similarity. More metrics are needed to determine if two codes can be applied for a given optimization strategy (in our case OpenMP parallelization).

Since loop parallelization information and data accesses are important factors for codes with OpenMP, we extracted this information from the compiler and gathered hardware counter data when we executed the serial version of the code on the processor. As our experiments were conducted in a hex-core Opteron processor, so we gathered the following hardware counters using AMD CodeAnalyst: "DC accesses", "DC misses", "DTLB L1M L2M", "CPU clocks","Ret branch", and "Ret inst". Once we got those metrics, we used Weka [15] to help us cluster the subroutines based on these code features, using the K-means algorithm based on the calculation of the Euclidean distance for each pair of subroutine. After that, we appended the Euclidean distance clustering back to the family syntactic distance tree, which serves as the optimization guidance. *Klonos* predicts that if two subroutines fall in

the same subtree which are also in the same performance cluster, then there is a high probability that those two subroutines should be optimized the same way.

Chapter 5

Evaluation of Syntactic Similarity Analysis

After the introduction of the *Klonos* tool framework, we will use a CFD benchmark BT as an example to provide details on how to use similarity methodology for porting code to a shared memory environment by using OpenMP. In particular, we will discuss what is gained by viewing the benchmark as a set of DNA-like sequences. We will also use the benchmark to test our similarity hypothesis. In addition, we will explore different code metrics, such as syntactic similarity, parallel information, application sampling, cost model, etc. to capture the code characteristics in order to make an accurate porting planning. Also we will show our methodology could be extended to other programming models such as HMPP, PGI, OpenACC, etc. for different programming architectures.

5.1 Similarity Analysis Metrics Used for BT Benchmark

BT is a simulated CFD application, meaning that it reproduces much of the data movement and computation found in the full-scale code. BT solves a system of three-dimensional compressible Navier-Stokes equations by factorizing the system into block tridiagonal matrices, followed by solving the block tridiagonal system along x, y, and z dimensions successively. Figure 5.1 shows the overall structure of the benchmark. A major portion of the execution time is spent in the three subroutines `x_solve`, `y_solve`, and `z_solve`.

```
program BT
    . . .
  do step = 1, niter
    call compute_rhs
    call x_solve
    call y_solve
    call z_solve
    call add
  enddo
    . . .
end
```

Figure 5.1: The overall structure of BT.

The three subroutines are the target when the optimized serial version of BT was ported to the multi-core platform using OpenMP [46]. By examining the source code for these subroutines, we found that they are similar to each other structurally. This may or may not be detected by a given similarity detection tool, depending on the tool's capability. For example, Unix `diff` cannot detect such similarity. Other

text-based comparison tools may not capture correctly the structure of the code or differentiate the data accesses of the code (which plays an important role for performance). In contrast, if we translate the code for each subroutine to a DNA-like sequence, then we can see a high degree of similarity among these subroutines (details follow), because each character in the sequences encodes important code structures and data accesses.

Specifically, we represent the three subroutines by a sequence of characters encoding the operators and operands of the source code. This is done by mapping each node in the abstract syntax tree (AST) representation of the code into a character based on a character map that represents the type of the node. Each of the characters in the sequence represent operators and data operands that a typical compiler will translate to machine instructions. For example, we encode a loop as a 'L' and a subroutine call as 'C'. (The details of the encoding process will be described in Section 4.2.) Then we can use any similarity analysis method for DNA sequences to evaluate the similarity among the subroutines, and the similarity we measure is a type of syntactic (or representational) similarity. To calculate syntactic similarity, we align the sequences to identify functionally or structurally similar regions of code. Sequence alignment gives us a set of transformations (such as character substitution, deletions, insertions, and gaps) which can be scored and used as a metric for code similarity.

Figure 5.2 shows a graphical view of the similarity encoding result for the three original subroutines after we apply multiple sequence alignment. The result is visualized by using a tool called Jalview. Jalview [11] uses the ClustalW multiple

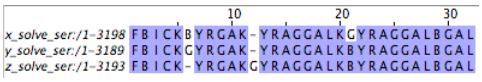


Figure 5.2: The results of a similarity encoding for BT when the subroutines are viewed as a set of DNA-like sequences. The figure shows a local view of the sequences.

alignment algorithm [24] to optimally align the three sequences with respect to a substitution matrix. For simplicity, we use the identity substitution matrix where matches have a score of 1 and mismatches with a score of -1. The shaded area in the figure indicates where the sequences are identical after the alignment. Therefore, the larger the shaded area, the more similar the sequences are. As we can see, the three subroutines are very similar to each other.

In our experiment, we also used another tool called *needle* from the European Molecular Biology Open Software Suite (EMBOSS) to calculate a similarity score for a pair of sequences, which uses the Needleman-Wunsch [62] pair-wise global alignment algorithm. With *needle* we also used the identity substitution matrix. The output of the score is based on the percent of identity (described later), which is correlated to the percentage of the shaded area in the overall area. A larger score indicates more resemblance between two subroutines. The similarity scores for the three subroutines, when compared to each other, are all around 87%. In other words, these subroutines are similar to each other according to the similarity definition. Note that the figure only shows the first 32 characters of the sequences. The length of the sequences are around 3,200 characters before the alignment and 3,250 after the alignment. Figure 5.3 shows the overall percent of identity of all the routines of BT when compared to each other in a 2D plot. The *hotter* colors means the higher similarity (percent of identity) between two routines. We can observe several regions

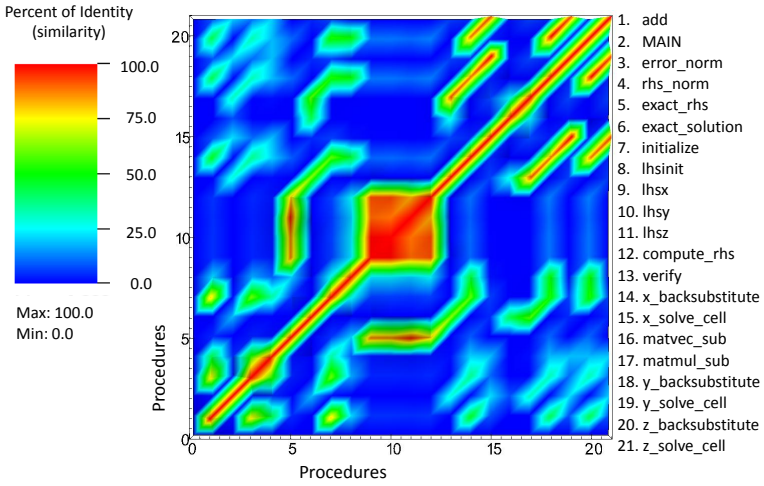


Figure 5.3: The percent of identity among the different routines of BT benchmark.

in the plot with a high degree of similarity that need further classification with a bio-inspired view.

We have intentionally used two tools, Jalview and EMBOSS, to evaluate the similarity of subroutines. We wanted to demonstrate that there is a variety of tools available for similarity analysis of DNA sequences, and we should leverage them. Another motivation for viewing the source code of a scientific application as a set of DNA-like sequences is that a"family distance tree" can be constructed. The tree is constructed based on the alignment scores of the sequences, and provides valuable structural information that other types of similarity analysis would not be able to provide. For example, Figure 5.4 shows the family distance tree for all the subroutines in BT, created by the third tool called Geneious [8]. The tree not only provides the similarity information between two subroutines but also imposes a hierarchical structure and can be used to order them for processing.

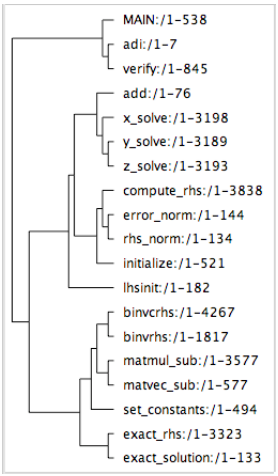


Figure 5.4: A "family distance tree" of all subroutines in BT based on syntactic similarity.

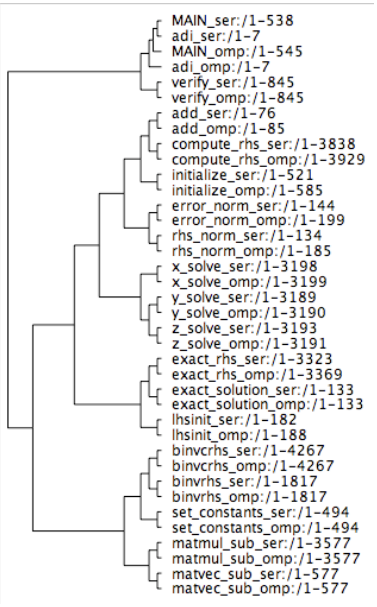


Figure 5.5: The results of a similarity analysis for the OpenMP version of BT as an evolution tree construction problem.

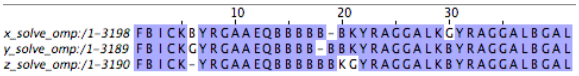


Figure 5.6: The results of a similarity analysis for the OpenMP version of BT as a sequence alignment problem.

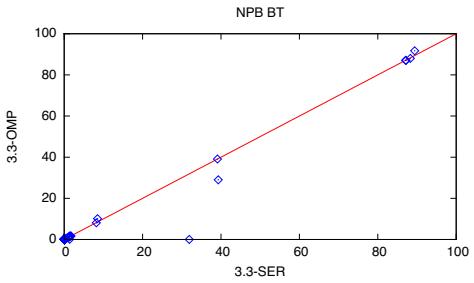


Figure 5.7: The results of a similarity analysis for the OpenMP version of BT through correlation analysis.

Since the group of `y_solve` and `z_solve` is more similar to `x_solve` than to `add`, it may be more productive if we port `x_solve` before porting `add` once both `y_solve` and `z_solve` are ported, making the porting plan familiarity oriented. Subroutine `y_solve` and `z_solve` are more similar to each other because their code structures and data access patterns are similar, and our sequence encoding scheme can capture both.

In addition, the family distance tree constructed is balanced rather than skewed. A balanced distance tree is desirable because it helps minimize the number of subroutines we will need to learn how to best port. Specifically, if we divide subroutines into groups based on similarity, then we will only need to focus on finding the best porting for one representative subroutine from each group. The rest of the subroutines are relatively easy to port according to our similarity analysis. Given that the number of groups is an indicator of how complex the porting process will be, a plan with a small number of groups is easier to porting. For example, there are 19 leaves

in the tree for BT. The perfectly balanced tree has the height of 5 whereas the most skewed tree has the height of 18. The height of the tree for BT is 7. Therefore the tree is more balanced than skewed. In comparison with `compute_rhs`, subroutines `x_solve`, `y_solve`, `z_solve`, and `add` are much more similar to each other, and can be put in the same group while `compute_rhs` is in another group. That other group may also include `error_norm`, `rhs_norm`, and `initialize` due to the balanced feature of the tree. In other words, we can use a similar porting strategy for those 8 subroutines by only concentrating on two subroutines. This is certainly a productive porting plan. We have demonstrated a few benefits we have gained with the bio-inspired view of program, namely, the abundance of tool support and the additional structural information. We believe that there are more benefits that the bio-inspired view can provide, such as evolutionary trees.

5.1.1 Verification of Similar Optimization

Now we need to verify whether similarity measure works in practice for a good-quality porting. First, we need to find a good-quality port of BT. Since BT is part of NAS Parallel Benchmarks, and these benchmarks have been ported to the OpenMP version [46] with the goal of comparing the performance of parallel computers, we assume that the OpenMP version of BT is a good-quality port. Figures 5.5, 5.7, and 5.6 show the results of various tests. As we can see from Figure 5.6, subroutines `x_solve`, `y_solve`, and `z_solve` are similar to each other in the ported version. The similarity scores for the ported subroutines are all around 87%. Since they are similar in the original serial version, the hypothesis is true for these three subroutines.

Figure 5.7 shows the correlation between the original and ported versions for all pairs of subroutines in BT with respect to similarity. For the hypothesis to be true, we need to see the linear correlation exhibited (as indicated by the diagonal line). The figure clearly shows the linear correlation. We can conduct statistical tests if we want the analysis to be more rigorous. We omit them due to space constraints. We can also test whether the two family distance trees are identical or not. Figure 5.5 shows a different test based on the tree view. We construct a family distance tree for the union of two versions of code. From the figure we see that not only are the ported versions of `x_solve`, `y_solve` and `z_solve` similar to each other but also the original of each subroutine are similar as well. In other words, this particular view provides additional insight that other views such as Figures 5.6 and 5.7 cannot provide. The high similarity between the original and ported versions of a code indicates that the code is not dramatically changed. This explains why our similarity analysis is correct for those three solvers.

Finally, we note a difference between syntactic similarity and performance similarity. It has been observed that `z_solve` spends more execution time than `x_solve` and `y_solve` because the data is accessed non-contiguously, thereby producing more cache misses [46]. As a result, we expect to see less similarity between `z_solve` and the other two solvers in the ported version because different data locality optimizations will need to be employed. Instead, `z_solve` is actually more similar to `y_solve` than between `x_solve` and `y_solve`. This is because our similarity metric groups them together due to their non-contiguous memory access pattern whereas `x_solve` has the non-contiguous memory access pattern.

5.2 Using *Klonos* for the NPB

In this section we present the experimental setup and results for NAS Parallel Benchmarks (NPB)3.3. We pay particular attention to the verification of the similar optimization strategy or directives and to the shape of the family distance tree for each benchmark.

5.2.1 Experiment Setup

NPB consist of 10 CFD benchmarks implemented and optimized serially as well as with OpenMP. Table 5.8 shows some properties of these benchmarks. In our experiments we omit EP since it has no subroutines. We also omit two benchmarks written in C, DC, and IS, because *Klonos* is currently only verified for Fortran programs. C programs require the sequence encoding to be extended to include pointers. BT is omitted because we have presented its experimental results in Section 5.1. Thus, we will only collect results for CG, FT, LU, MG, SP, and UA.

We use *Klonos* to collect the results. We use it to create the family distance tree and pair-wise similarity scores for each of the six benchmarks, like what we did for BT in Section 5.1. We assume that the OpenMP version of NPB's 3.3 is a good-quality port. Thus, we run *Klonos* for both serial and OpenMP versions to collect trees and scores.

Figure 5.8: Benchmarks properties of NPB's 3.3

Benchmark Name	Programming Language	Number of Subroutines	Lines of Code
BT	Fortran	19	5059
CG	Fortran	7	1034
DC	C		2719
EP	Fortran	1	272
FT	Fortran	12	790
IS	C		804
LU	Fortran	18	5119
MG	Fortran	15	1376
SP	Fortran	20	3144
UA	Fortran	67	6751

Figure 5.9: NPB family distance trees statistics.

Benchmark Name	Height Range	Tree Height
BT	[5,18]	7
CG	[3,6]	4
FT	[4,11]	6
LU	[5,17]	7
MG	[4,14]	7
SP	[5,19]	8
UA	[7,66]	26

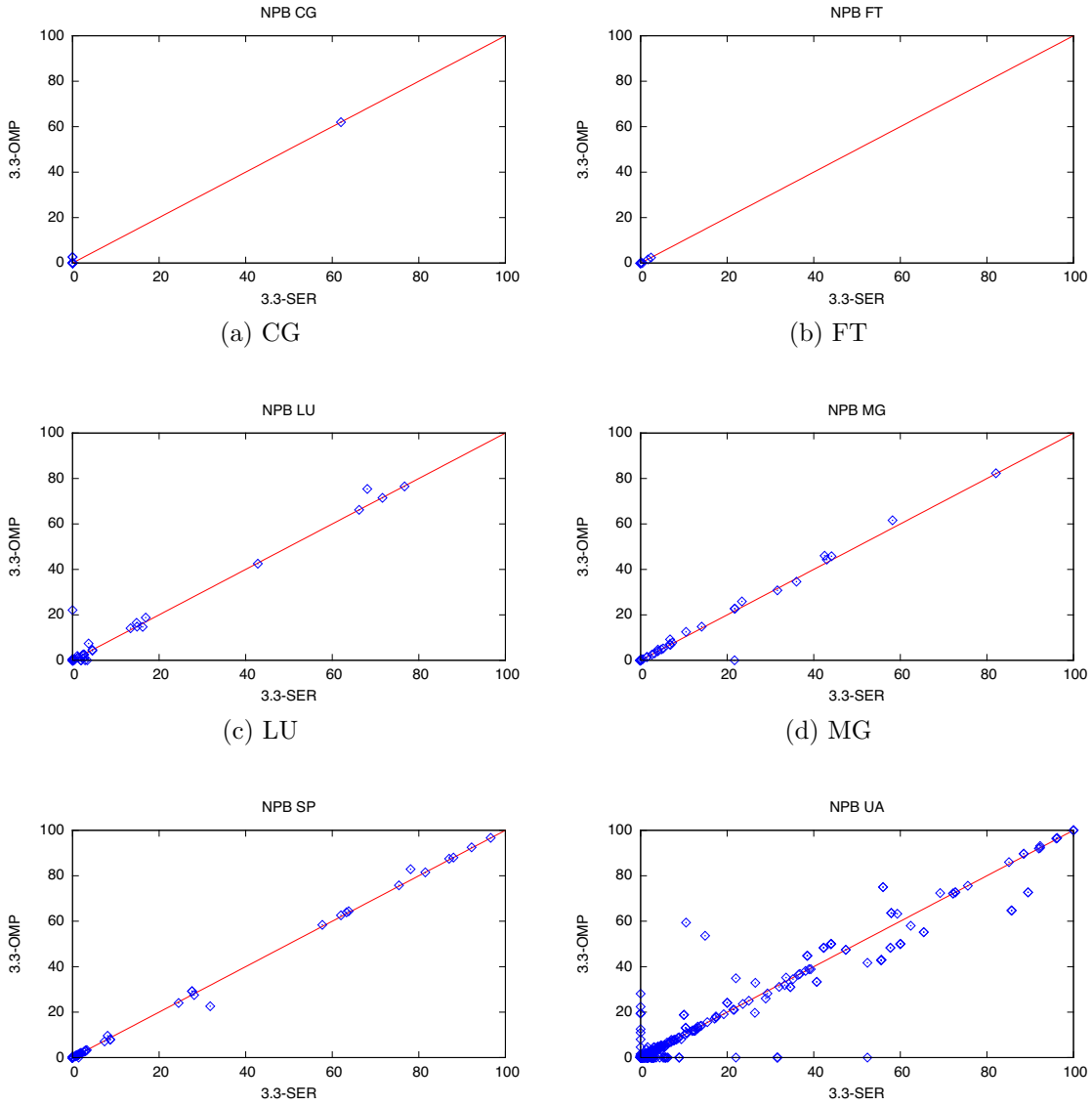
5.2.2 The Verification of Optimization

We first check the similarity of optimization. The hypothesis states that the similar subroutines in the serial version should be similar to each other in the OpenMP version. We will show that this is generally true for the tested benchmarks.

Figure 5.10 shows six scatter plots, one for each NPB. A dot in the plot represents a pair of similarity scores for the serial and OpenMP versions with respect to a pair of subroutines. The diagonal line indicates results that would indicate that our hypothesis is true. (In fact, the line shows a much stronger hypothesis.) If all the dots are located near the line, then the hypothesis is generally true. As we can see from all the plots, this is indeed the case.

Figure 5.10 also provides other insights. First, we observe that there exists more similarity chances between subroutines when the application consists of more subroutines due to large number of subroutines. UA and SP are two largest applications among all NPB's. Both of their plots show more dots on the right hand side of the plots, meaning that there are more subroutines similar to each other with a high degree. This is promising as our methodology desires an application with such a feature.

Second, the plot for UA has *outliers*. One type of outlier shows a case where two subroutines are highly similar in the serial version but less similar in the ported version. Looking into these two subroutines, we found out that one subroutine is parallelized with the OpenMP directive whereas the other is not parallelizable, therefore leading to a lower similarity in the ported version of the code. Conceptually,



(a) CG (b) FT

(c) LU (d) MG

(e) SP (f) UA

Figure 5.10: The visual check of the hypothesis.

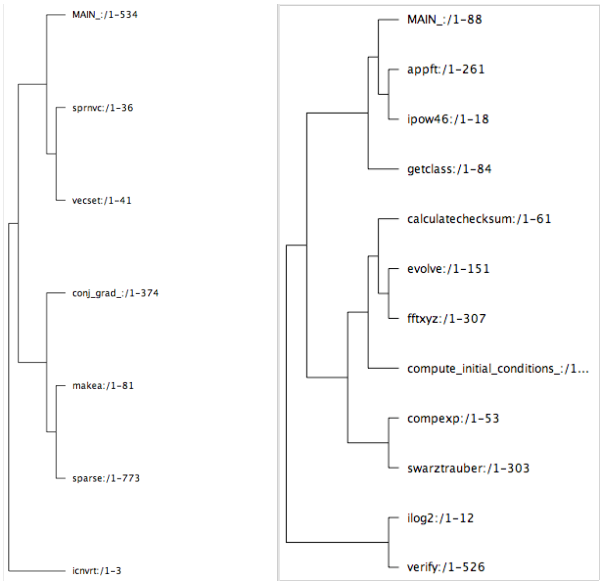
this action causes the dots to go "up" in the scatter plot. This case shows that our similarity-based methodology enables us to do a cross checking after the porting is done, i.e., to see if there exists any possible code transformation that the user neglected or intentionally chose to ignore.

Another type of outlier shows a case where two subroutines are reasonably similar in the serial version but less similar in the ported version. We attribute this to the imperfectness of the syntactic similarity metric we use. Specifically, two syntactically similar subroutines may not be suitable for the same set of optimization strategy in some cases. Conceptually, a better similarity metric will move these dots "left" in the scatter plot. Note that our similarity still holds as it only concerns highly similar subroutines. This experiment shows that the capability of *Klonos* to detect missing possible optimization strategies missed by the developer.

5.2.3 The Family Distance Tree for Each NPB

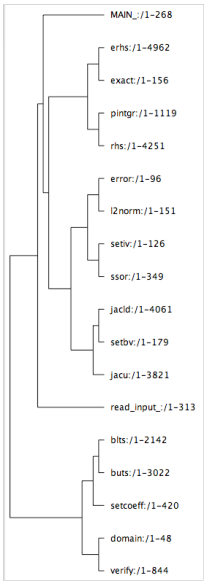
Now we turn to the shape of the family distance trees for all six NPB's. Figure 5.11 shows these trees. As we have observed in the BT case, the balanced tree is desirable. Table 5.9 shows that the six trees are more balanced than skewed, using the tree height as a measure of tree balance.

Although Geneious generates the tree with a better layout, the construction is still based on similarity. We also believe that it is not coincidental for the following reason: the subroutines in a scientific application often provide different functionalities for the same data domain. In CFD codes, the data domain is the fluid substances while

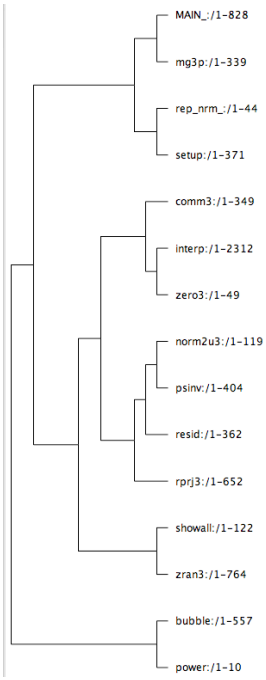


(a) CG

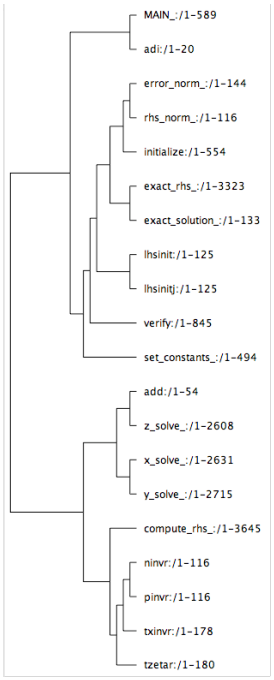
(b) FT



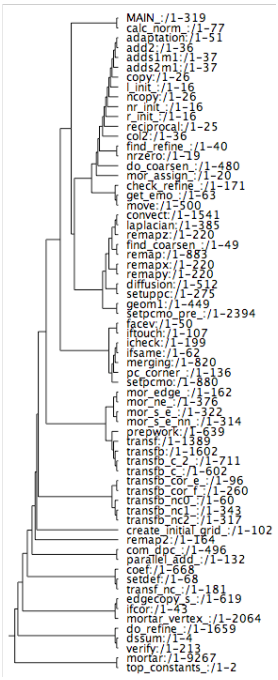
(c) LU



(d) MG



(e) SP



(f) UA

Figure 5.11: The family distance trees of all serial NPB's.

in cosmological simulation, the data domain is the discretized universe. We can observe that domain information is also encoded in the syntactic structure of the subroutines (same type of solvers belong to the same group) which are translated into syntactic similarity. As a result, the distance tree is balanced.

5.3 Scalability of *Klonos*

Normally the size of a scientific application is large, containing several hundreds or thousands of subroutines. In order to test the scalability of *Klonos* tool, we applied this tool for a large scientific application called "HOMME" which is introduced in Section 2.4.2.

5.3.1 HOMME Application

The HOMME application consists of several hundred Fortran90 procedures where the computations are spread evenly across them, and whose relevance depends on the input problem. There is a certain degree of syntactic similarity between the solvers that are implemented with minor changes in their algorithms to deal different physical properties. Figure 5.12 shows the overall similarity among the different routines in HOMME when doing a pair-wise comparison. The *hotter* colors (dots) represent pair-wise routines with a high percent of identity. By inspecting the figure, we can see areas with a high degree of similarity that need to be classified and further inspected for porting planning purposes.

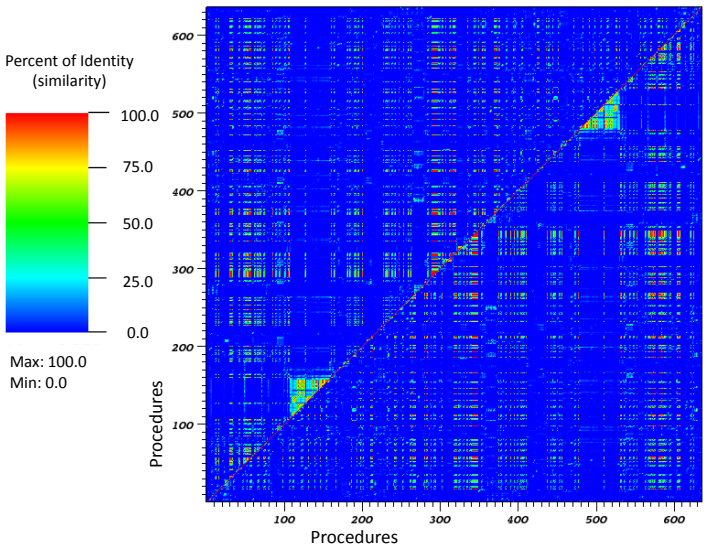


Figure 5.12: The percent of identity among the different routines of HOMME.

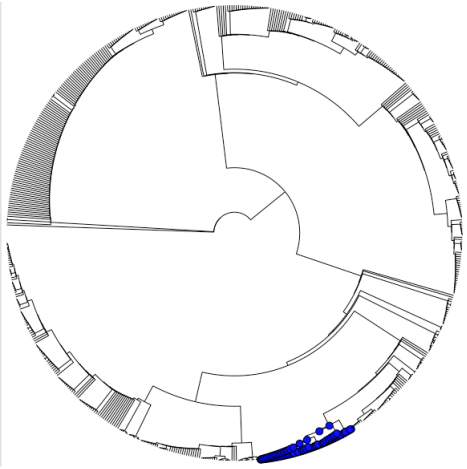


Figure 5.13: The "family distance tree" of all subroutines in HOMME based on syntactic similarity.

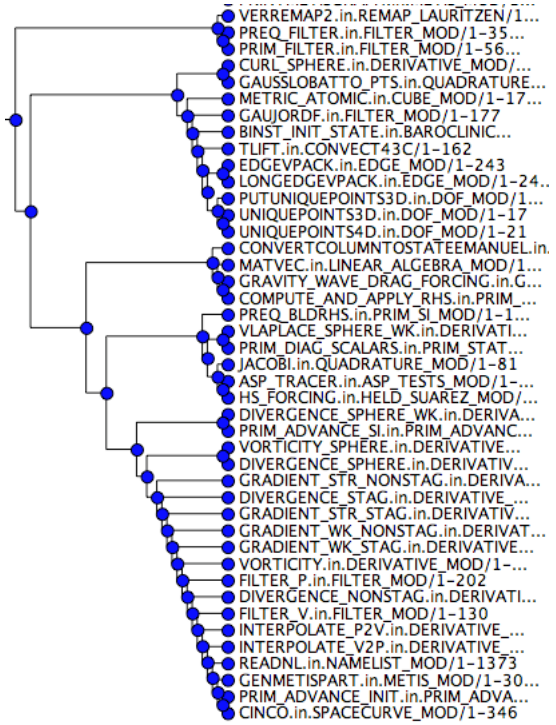


Figure 5.14: A subtree from the "family distance tree" in HOMME that belong to similar implicit solver

Figure 5.13 shows the overall distance tree among the different routines of HOMME. Because of space issues, we just present the overall view of the shape of the distance tree. There is a large portion of the tree that is unbalanced, which consists of routines with small degrees of similarity among them. However, there are several balanced subtrees where routines have very similar structures. We highlight a portion of a subtree in blue to show one of those regions. It is a subtree that contains a range of implicit solver with similar structure in the application. The distance tree was able to cluster these type of routines together, which shows that our approach can also encode some domain knowledge of the application. This is something that a scientist might need to validate or annotate as useful domain information to the subtree. Figure 5.14 shows a zoomed in version of the subtree selected in Figure 5.13. We can see that some implicit solvers routines have a high degree of similarity. For example, *divergence_sphere* and *vorticity_sphere* have a similarity of 91.8%. Other solvers, *curl_sphere* and *gradient_sphere* have a similarity of 90.8%, but have an average similarity of 80.68% with *divergence_sphere*. The procedures *interpolate_V2P* and *interpolate_P2V* have a degree of similarity of 100% because they are interpolating between velocity and pressure, which have the same data structures and interpolation algorithm. A similar porting strategy can be applied these procedures based on our manually porting observation mentioned in Chapter 2.

For evaluation purposes we evaluated how scientists ported the routines *divergence_sphere* and *vorticity_sphere* to a GPU-based system using a high-level accelerator directives. Because of the high degree of similarity of the routines, we noticed that scientists ported these routines using the same strategy, while achieving good

speed-ups. Both routines were parallelized over the physics and the vertical dimension loops of the cube-sphere and mapped them to the GPU multi-processors, while the cubed-sphere planes were parallelized and mapped to the GPU thread-blocks. Our similarity approach can be very promising in this case, as long as we can use other analysis such as parallel, data scoping, and alias information to validate if two routines can be ported the same way.

Chapter 6

Adding Code Metrics Similarity in Porting

Although syntactic similarity could be used to detect possible missing optimizations neglected by developers, we still need other code feature metrics to increase the accuracy of deciding whether a similar optimization or porting strategy could be applied for similar subroutine. Also, for different architectures, different metrics have to be used in order to meet different porting environment needs, since most optimizations or porting strategies are target specific. In this chapter, we will show other possible metrics that we have explored for *Klonos* tools. Those metrics include cost-model-related code feature information and relations of syntactic similarity maintained at different compilation phases.

6.1 Dynamic Code Feature Similarity

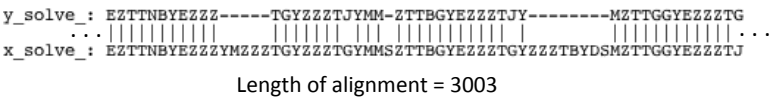


Figure 6.1: NAS BT OpenMP benchmark x_solve, y_solve sequences alignment

Although syntactic similarity can help users to locate structurally similar code regions, this is still not sufficient to decide whether similar porting strategies could be used for similar subroutines or code regions. Other code information and metrics need to be taken into consideration. To illustrate this, we use the BT serial NAS benchmark as an example to demonstrate how our notion of similarity can be extended to other code information analyses to help the user parallelize this code with OpenMP. This technique can also be used for other directive-based programming models. BT belongs to a family of CFD code, and it uses a multi-partition scheme to solve three sets of uncoupled systems of equations in a block tridiagonal of 5x5 blocks. The direction of the solvers are in the x, y, and z dimensions over several disjoint sub-blocks of the grid. The solvers use the same algorithm along the different directions before the results are exchanged among the different blocks. Because of this algorithmic property, BT is a good candidate to illustrate how our notion of similarity can help to analyze and parallelize this code.

One way to quantify the source code similarity is to convert the intermediate representations of subroutines of a program into sequences of symbols (in our case, we used characters) that can be aligned using a global sequence alignment algorithm. The result can then be used to calculate their pair-wise syntactic distance based on a

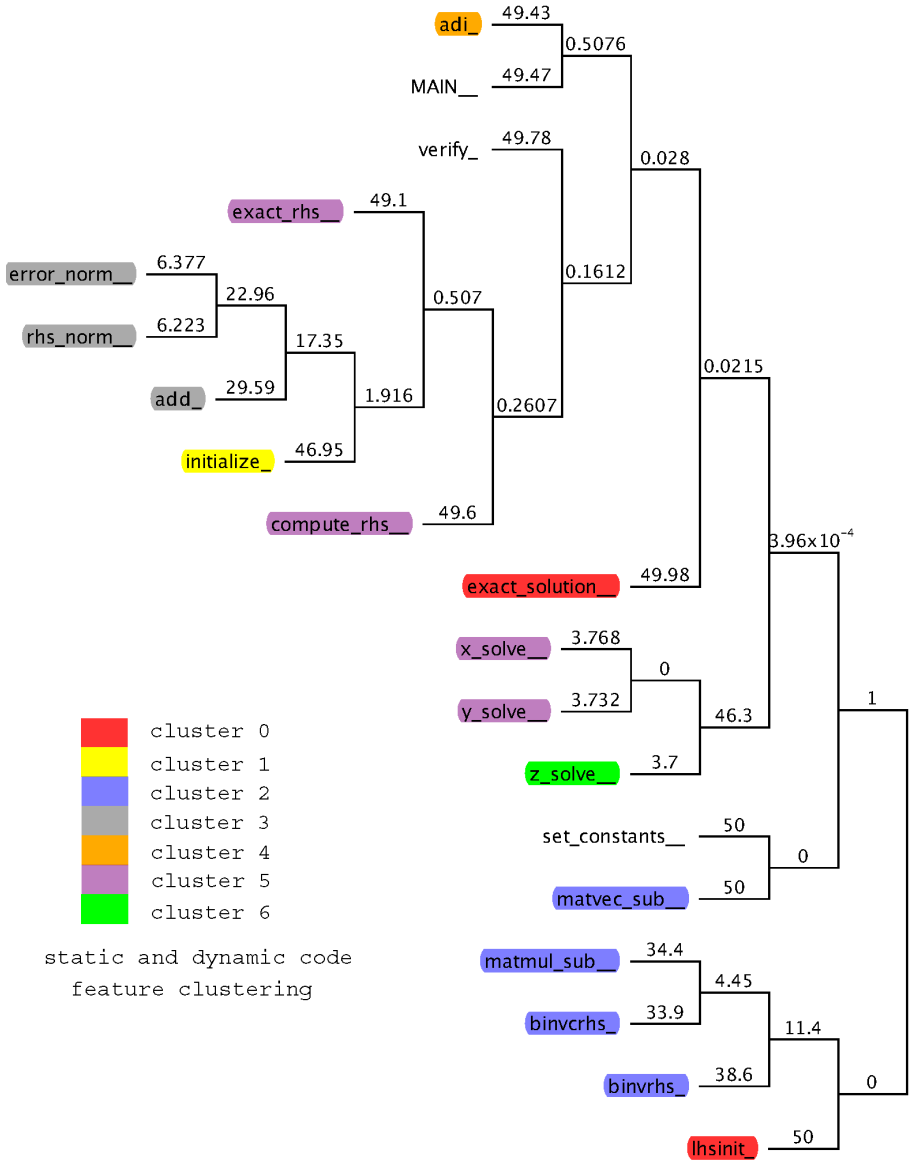


Figure 6.2: NAS BT benchmark porting planning tree

percentage score reflecting how similar they are (for example 100% means identical). Figure 6.1 shows one portion of the pair-wise sequences alignment of subroutines *x_solve* and *y_solve*. The length of the alignment is of 3003 characters. The vertical lines indicate the portions of the sequences that are identical and the portions of the subroutines that have identical operators.

We can then use the Neighbor-Joining algorithm to create a family distance tree based on the pair-wise distance of the subroutines. Figure 6.2 shows the family distance tree for the BT serial benchmark. Each edge of the tree is annotated with a distance score, which represents the degree of their syntactic code differences. The distance between two subroutines can be calculated by adding the distance value of the edges between them.

By looking at the tree, we find that *x_solve*, *y_solve*, and *z_solve* are siblings. *x_solve* and *y_solve* are grouped into one subtree, and their parent node is grouped with *z_solve* into another subtree. By calculating the distance between these three subroutines, we get *x_solve* $\sim$ *y_solve*=7.5, *x_solve* $\sim$ *z_solve*=7.468, and *y_solve* $\sim$ *z_solve*=7.432. These subroutines have small distances between them because they have a high degree of similarity in their source code, which is consistent with the algorithm of BT. Although the source code of the subroutines *x_solve, y_solve*, and *z_solver* look very similar, their data accesses are different. The subroutine *x_solve* has contiguous memory accesses but *y_solve* and *z_solve* have dis-contiguous memory accesses. This may impact the optimization strategy for these subroutines on a cache-based system. For example, the OpenUH [67] compiler optimizes the serial versions of *x_solve and y_solve* similarly (when inspecting their intermediate representations

Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
lhsinit	initialize	matvec_sub	add	adi	exact_rhs	z_solve
exact_solution		matmul_sub	error_norm		compute_rhs	
		binvcrhs	rhs_norm		x_solve	
		binvrhs			y_solve	

Table 6.1: Subroutine clusters for the serial BT benchmark based on the code features

Attributes	Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
DC accesses	0.0004	452.5	0.0007	84.058	0	1747.5666	1629.2438
DC misses	0.0004	25.5	0	10.2338	0.005	420.6692	858.1244
DTLB L1M L2M	0	0.5	0	0.6202	0	4.5561	37.9751
CPU clocks	0.0008	623.5	0.0012	194.0514	0.005	2718.1974	2913.2836
Ret branch	0.0001	113.5	0.0001	9.2769	0	210.1474	139.8806
Ret inst	0.0005	1227	0.0015	125.8657	0	3485.2936	3125.7264
# parallel loops	1.5	21	0	3.3333	0	22.5	15

Table 6.2: Cluster center point for the serial BT benchmark based on the code features

after optimizations) but uses a different strategy for *z_solve*.

Based on this information, if we want to parallelize these subroutines using OpenMP, we cannot rely on the syntactic similarity analysis, because other code features need to be taken into consideration. Since we use OpenMP for porting serial code to a shared memory environment, some important code features for porting with OpenMP have been taken into consideration: parallel information is used for detection of parallelism, amount of work, data access pattern, etc. These might be, for example, the number of parallel loops in the subroutine, the values of hardware counters that characterize the data access, and the amount of work done for each subroutine. We can define a set of program features (analyses) that are relevant to OpenMP optimizations and cluster them to further classify the subroutines. For the

BT Benchmark, we clustered its subroutines based on the number of parallel loops, data cache accesses, and cache misses, total number of cycles, TLB misses, and total number of instructions. The subroutines were classified into seven clusters (the number of families in level three of the distance tree), using the K nearest neighbor (K-NN). The k-means method is favored when the number of data points is small. The result of the cluster is shown in Table 6.1. The dynamic code features were calculated by running the BT serial benchmark with class B on a hex-core Opteron 2435 processor. Each cluster consists of a set of subroutines with the closest Euclidean distance among their feature vectors. Table 6.2 shows the list of code features of each subroutine that are used for the clustering. It also shows the average values of the code features per cluster. In our experiment, we use the Instruction-Based Sampling (IBS) events, since those events are the key factors which can summarize the memory access pattern (or internal application behavior) of each subroutine. Those memory events have a direct link with the optimization that contributes to the final performance. Although IBS is a statistical method, the sampling technique delivers precise event information and eliminates inaccuracies due to skid [10] for the AMD Family 10h processors. Using the clustering results we can annotate families of codes that share important code features for OpenMP optimizations. Figure 6.2 shows the resulting annotated serial BT benchmark porting planning tree we achieved based on the collected static and dynamic information of the syntactic similarity of the code and its features. The subroutines marked with the same color have a greater potential to be optimized similarly if they are syntactically close enough to each other in the same subtree with small syntactic distance. Our sampling performance tool was

not able to collect hardware counter information for the subroutines *MAIN*, *verify*, and *set_constants*, because their execution time was too short. We excluded these subroutines from further similarity analysis.

After collecting this information, the next step is porting planning. We notice that *x_solve*, *y_solve*, and *z_solve* fall into two different code feature clusters, although their syntactic distance is small, with *x_solve* and *y_solve* in the same code feature cluster. So we can predict that *x_solve* and *y_solve* can be optimized using the same OpenMP strategy, while *z_solve* might need a different one since it falls into "cluster 6" based on the code feature clustering. This result suggests that the user should first attempt to parallelize *x_solve* with OpenMP, then based on this experience develop a porting strategy that can be applied to *y_solve*. For the case of *z_solve*, a different porting strategy is needed. When we inspected the corresponding OpenMP version of these solvers, we noticed that the user inserted an OpenMP *do* directive at the same loop level of the main computation loop. The user also used the same privatization and data scoping strategy for the data. The user chose not to optimize the data access of *z_solve* and left this job to the compiler. This is perfectly captured by the planning scheme supplied by our tool. The user applied the same parallelization strategy for the subroutines *error_norm* and *rhs_norm* that fall under the same syntactic distance family and the code feature clusters. Our tool predicted that these two subroutines may be parallelized by using a similar OpenMP strategy. Figure 6.3 shows a partial code listing of those two subroutines after being parallelized with OpenMP (from the OpenMP version of the benchmark). We observed that the programmer used exactly the same OpenMP strategy as suggested by our similarity tool.

```
!$omp parallel default(shared)
!$omp& private(i,j,k,m,...,u_exact,rms_local)
!$omp&        shared(rms)
  do m = 1, 5
    rms_local(m) = 0.0d0
  enddo
!$omp do
  do k = 0,grid_points(3)-1
    zeta=dble(k)*dnzm1
    do j=0,grid_points(2)-1
      eta=dble(j)*dnym1
      do i=0,grid_points(1)-1
        xi=dble(i)*dnxm1
        call exact_solution(xi,...u_exact)

        do m = 1, 5
          add=u(m,i,j,k)-u_exact(m)
          rms_local(m)=rms_local(m)+add*add
        enddo
      enddo
    enddo
  enddo
!$omp end do nowait
  do m = 1, 5
!$omp atomic
    rms(m)=rms(m)+rms_local(m)
  enddo
!$omp end parallel
```

(a) *error_norm*

```
       .  .  .

!$omp parallel default(shared) private(i,...)
!$omp&        shared(rms)
  do m = 1, 5
    rms_local(m) = 0.0d0
  enddo
!$omp do
  do k=1,grid_points(3)-2
    do j=1,grid_points(2)-2
      do i=1,grid_points(1)-2
        do m=1,5
          add=rhs(m,i,j,k)
          rms_local(m)=rms_local(m)+add*add
        enddo
       enddo
     enddo
  enddo
!$omp end do nowait
  do m = 1, 5
!$omp atomic
    rms(m)=rms(m)+rms_local(m)
  enddo
!$omp end parallel

       .  .  .
```

(b) *rhs_norm*

Figure 6.3: Subroutines *rhs_norm* and *error_norm* code snippets of the NAS BT benchmark

Based on these findings, we believe that the experiences gained when porting a subroutine using OpenMP can be used for similar subroutines and benefit from previous optimizations/transformation strategies. This dissertation includes a new approach to define the code similarity based on syntactic structure of the codes and code features that are relevant for the OpenMP parallelization. If two codes are similar, there is a high probability that these codes can be ported using the same OpenMP strategy.

6.1.1 Experiment for NPB-3.3 Benchmarks

The OpenUH compiler first translates different languages to a high level intermediate representation (IR), called WHIRL [3]. For each subroutine, we summarize the intermediate representation (IR) into character sequences by traversing the IR in post-order. The characters in the sequences represent operators and operands based on a "node-map" described in [32].

In order to further verify this methodology, we use a similar concept for extracting the code sequence. To generate the optimization distance, we first trace the functions in charge of the OpenMP transformation. We used a unique letter to denote each function called during the OpenMP translation phase. Those functions are responsible for translating a given OpenMP construct. The optimization process has been converted to a flattened sequence for a comparison. Similarly, we can derive the optimization distance from the OpenMP version of NPB based on the optimization similarity score described above. The generated optimization distance is then used to check if two codes were lowered and optimized in the same way.

In Section 5.1, we explained how we used the similarity technique to find a similar porting strategy which could can be applied to the similar subroutines inside the BT benchmark. Due to space limitations, we are not able to the list all the family distance trees and code feature clusters for the rest of the five benchmarks. However, in Figure 6.4, all the pair-wise subroutines comparison from the six benchmarks including the BT are shown. Each dot inside this diagram refers to a pair of subroutines with respect to their syntactic and optimization distance. The *x axis*

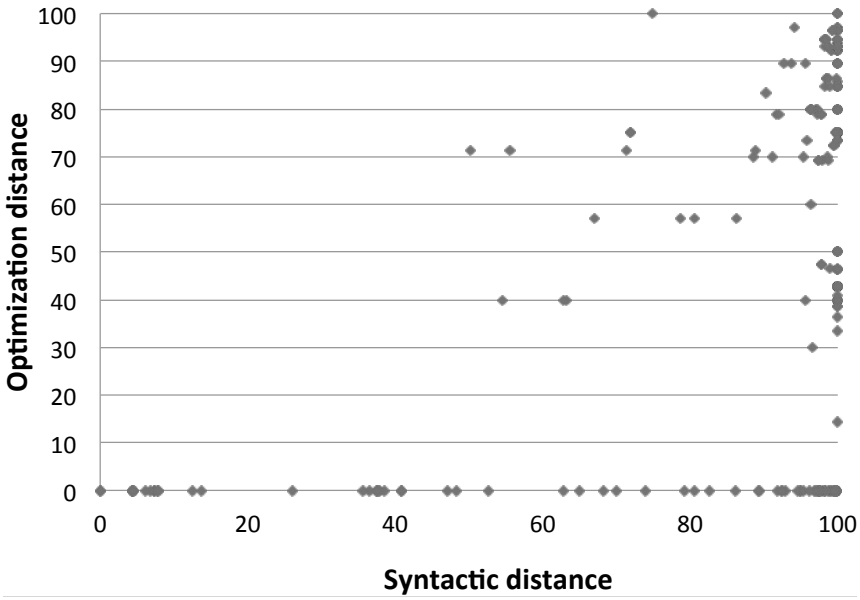


Figure 6.4: Syntactic and optimization distance for a pair of subroutines

shows the syntactic distance between a pair of subroutines, and the y-axis represents a optimization similarity as a distance value. for comparing of two subroutines.

According to Figure 6.4, we observe that a syntactic distance of 50 is an appropriate threshold for the NAS benchmark.

For the subroutine pairs with a syntactic distance less than 50, they are more likely to have an identical OpenMP optimization strategy. Figure 6.5 shows the percentage of subroutine pairs that have optimization distance is always zero when their syntactic distances are less than 50, 60, 70, and 80 respectively. When the subroutine pairs have syntactic distance less than 50 (similarity score is greater than 50), we found that they can all be optimized in the same way with OpenMP. However only 90 percent of the subroutine pairs are optimized in the same way when their syntactic distance is less than 60. The optimization strategy starts to

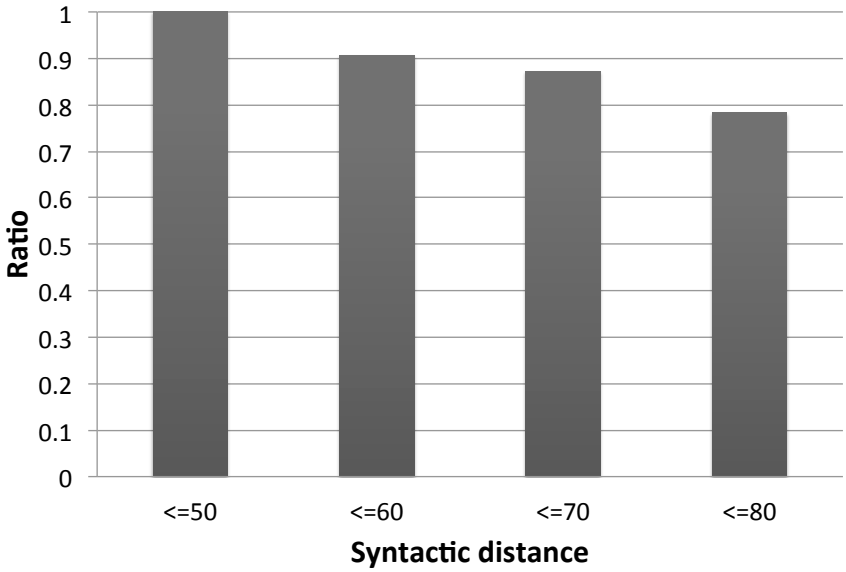


Figure 6.5: Percentage diagram for the syntactic distance

change when the syntactic distance goes beyond 50. As the code structure further diverges, the optimization similarity continues to decrease. This is generally true that different subroutines are optimized differently. Referring back to the generated family distance tree 6.2, we verified that all the subroutine pairs that classified into the same syntactic and code feature clusters, used the same optimization strategy with OpenMP as long as their syntactic distance is less than 50. We also observe that for some cases they use the same optimization strategies although they diverge dramatically from each other.

6.2 Cost-Model Metrics-based Similarity

The aggregated structural information provided by the static metrics, and the aggregated behavioral information provided by the dynamic metrics combine to identify

a viable porting plan for an application. However, it is still impractical to run an entire application to collect the performance sampling data, especially for a large application that consists of millions lines of code. To analyze a very large data set generated by such a run is difficult and time-consuming. Even if it is possible to collect this kind of performance data, the output is sensitive to the content of the input data and sampling information varies significantly between different execution phases. It is also evident that many optimization or code restructuring techniques used during porting are target specific, and lead to variations in performance on different platforms. Different cost-models will be used for different target systems. A cost model is a performance estimation without regard to specific input data, and is used by the compiler to select different optimization algorithms. OpenUH uses a shared memory processor cost model to evaluate different combinations of optimizations and to decide if there is enough work (in processor cycles) to benefit from automatic parallelization of a loop. The cost model is essential to evaluate whether it is worth applying static optimizations to loops and consists of three major components: processor, cache, and parallel overhead [88]. The similarity of code is measured by analyzing the similarity of cost-model-based metrics. Sections of code that exhibit the same metrics are likely to benefit from similar optimization and porting strategies.

6.3 Similarity Analysis for Other Proposed Metrics

After successfully testing the scalability of *Klonos* using a real application, we found that relying solely on syntactic similarity will help us locate similar code regions. However, if we want to accurately find similar optimization strategies that can be applied for similar code regions, we require more metrics that can be used to capture the internal code optimization characteristics. In our experiments, we used the parallel information and cost model as two metrics for code analysis.

6.3.1 Parallel Information Analysis

Parallelization analysis can play an important role in code optimization. The compiler first examines the source code, and then analyses any possible data dependencies that impede the parallelization. For each loop inside a subroutine, we rely on the compiler to help us analyze whether it (1) is parallelizable, or (2) might be parallelizable, or (3) is not parallelizable. Like the code sequence extraction we conducted before, we encode loop parallelization information into a flattened sequence. Based on the parallelization characteristic of loops, we denote a loop that is parallelizable with the letter "P", a loop that might be parallelizable with the letter "M" and a loop that is not parallelizable with the letter "N". The length of the loop parallel information sequence represents the number of loops of a subroutine.

A cost model is a model that is used to evaluate the possible computing workload

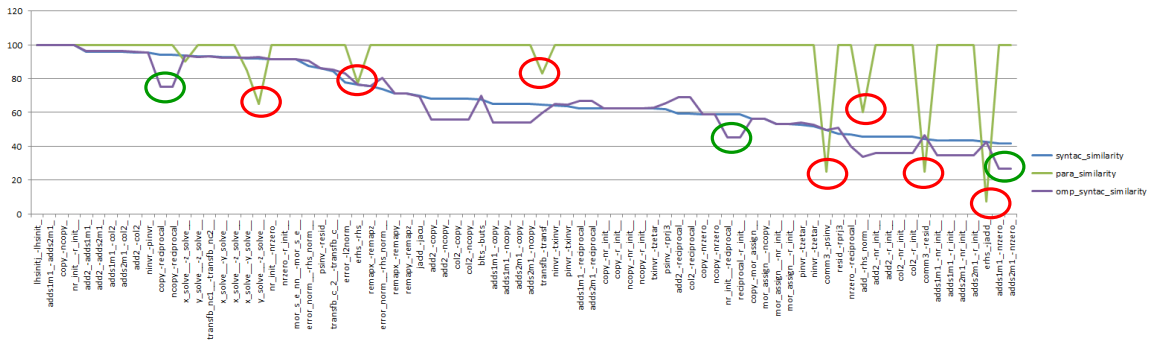


Figure 6.6: Syntactic, parallel information and cost model similarities for the NPB benchmarks

and performance outcomes for possible combinations (search space) and select the best overall transformations. It is used to help the compiler select optimal choices about loop transformations (loop permutation, outer unrolling factor, tiling size, fission, and fusion) that cannot be made individually. After the parallelization analysis, OpenUH applies cost model analysis for each possible parallelizable loop for a performance estimation, an optimized solution will finally be selected and applied for the code optimization. Using the OpenUH cost model, we are able to calculate a value to estimate the computing workload. By comparing the cost value of two subroutines using MIN{*cost_value_a*/*cost value_b*, *cost_value_b*/*cost_value_a*}, we are able to get a score which can serve as an indicator to evaluate the workload similarity of a pair of subroutines.

Figure 6.6 lists the syntactic similarity, parallel information similarity, and cost model similarity for the different subroutines of the NAS NPB benchmarks. Next, we will explore new methods to find the relationships among the different similarity metrics. These methods include probabilistic and transductive models, k-means, and Euclidean distances.

6.3.2 Similarities Comparison at Different Compiler Lowering Phases

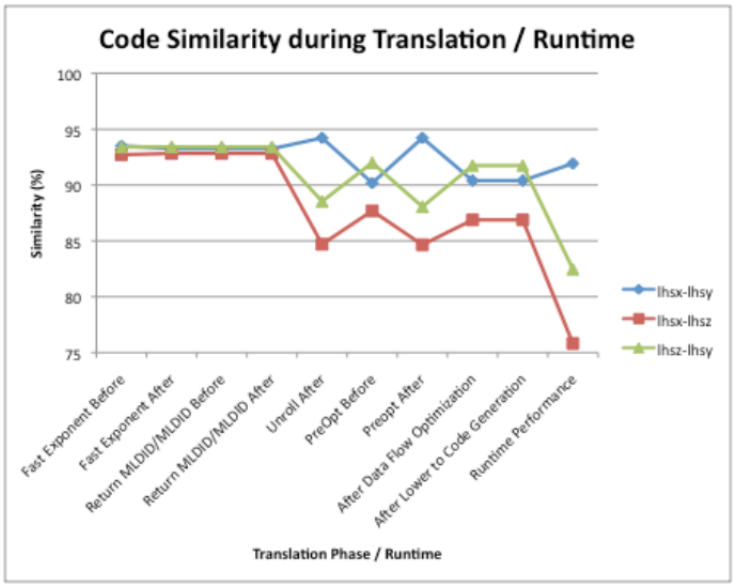


Figure 6.7: Similarity between three procedures in BT as they are translated by the compiler

Subroutines *lhsx*, *lhsy*, and *lhsz* inside the NAS BT benchmark, are siblings in the family distance tree because they have a high degree of similarity. However, these syntactically similar solvers might not perform the same way when translated by the compiler, since they perform calculations on different dimensions of various data structures. We used our similarity analysis to track down how these three solvers were translated and optimized within the OpenUH compiler. Some of their syntactic similarity was lost as the compiler used different strategies to exploit data locality, pre-fetching, and loop optimizations (fusion, unrolling). What we notice is that procedures *lhsx* and *lhsy* maintain a high degree of similarity throughout the compiler translation. Before the code translation, they have 94% syntactically similarity. After

the code translation, they only have 90% similar score. We learn that the compiler applied a similar strategy to them. Figure 6.7 shows how their similarity is affected by the compiler translation. The compiler is optimizing the codes similarly because the cost of the data accesses in x and y still remain close enough. When we execute these two solvers, they have a performance ratio of 93%, indicating that their performance characteristics are close to each other. We can advise the user to apply a similar code transformation strategy to these two procedures when optimizing for this platform and compiler. In contrast, when we compare these routines with *lhsz*, we discover that the syntactic similarity of the code is lost during the translation and in its performance. A different strategy is needed when optimizing the procedure *lhsz*.

Chapter 7

Combined Approach to Detect Similarity in Porting

In this chapter, we evaluate *Klonos* by applying it to a real scientific application porting to a shared memory environment using OpenMP. We adapt cost-model-provided metrics to capture code similarity in terms of optimization or porting, which saves the trouble of running the application for profiling information collection. For GenIDLEST application, we used the cost-model-based similarity analysis together with syntactic similarity, which could very accurately find similar subroutines to which we can apply similar optimization strategies. According to our experimental result, *Klonos* is very accurate in detecting similar codes which can be ported similarly.

7.1 GenIDLEST Similarity Analysis

For better evaluation, we have applied the "*Klonos*" tool to a real application called GenIDLEST. GenIDLEST is a Fortran program that simulates transitional and turbulent flows in complex geometries [81]. This application features both shared memory (OpenMP) and distributed memory (MPI) parallelism, which leads to a high degree of portability between computer architectures. This application is thus ideal for the porting planning strategy verification that we propose to perform with the *Klonos* tool. First, we use *Klonos* to analyze the serial version of GenIDLEST, and then generate a porting plan for a parallel version of the code using OpenMP. By referring to the optimized GenIDLEST OpenMP code, we are able to verify the accuracy of the proposed OpenMP porting plan with *Klonos*.

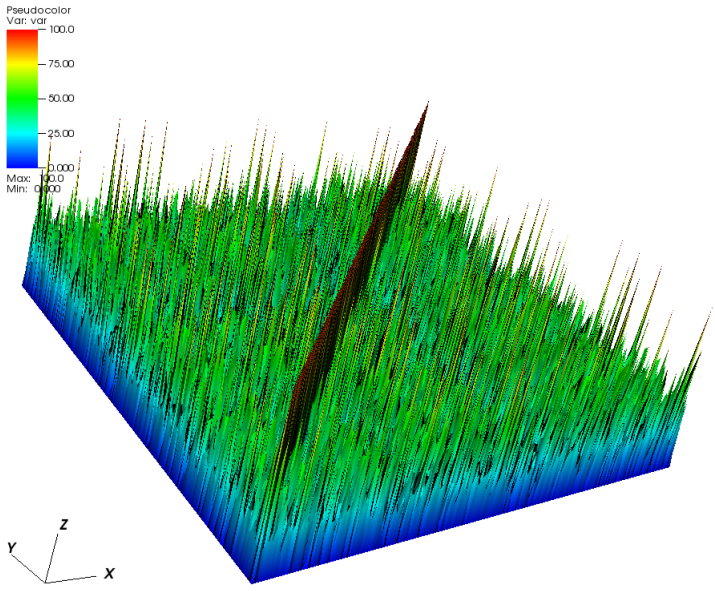


Figure 7.1: The subroutine similarities of the GenIDLEST application

GenIDLEST has a total of 264 subroutines. Before we perform the syntactic

similarity analysis, we pre-process the generated sequence pattern files by excluding subroutines with only one function invocation inside them, since those files only contribute noise through many highly syntactically similar pairs. After the preprocessing steps, we generate a similarity square matrix by comparing each pair of subroutine sequences until all the subroutines have been consumed. Figure 7.1 shows the 3D visualization of GenIDLEST subroutines. It lists the overall similarities among all the subroutines. Axes X and Y are subroutines, the Z axis represents the similarity score for each pair of subroutines. The node map legend shows the level of similarity. Red means high similarity and blue means low, or no, similarity. The diagonal shows subroutine self-similarity. Figure 7.2 is a circular family distance tree with height of 31. It shows the overall relationship of syntactic similarity for GenIDLEST subroutines after pre-processing. The family distance tree lists similarity relationship of 254 subroutines, which the total number of subroutines after preprocessing that excludes subroutines with only one function call inside.

Table 7.1 summarizes the statistics of the similarities of subroutines after preprocessing. GenIDLEST has 1327 subroutine pairs that maintain syntactic similarity of greater than 50%, which means a majority of subroutines look similar structurally.

7.1.1 Syntactic Cluster Analysis

Figure 7.3 shows the relationship of the number of correct porting similar subroutine pairs with setting different number of clusters based on code syntactic similarity. In

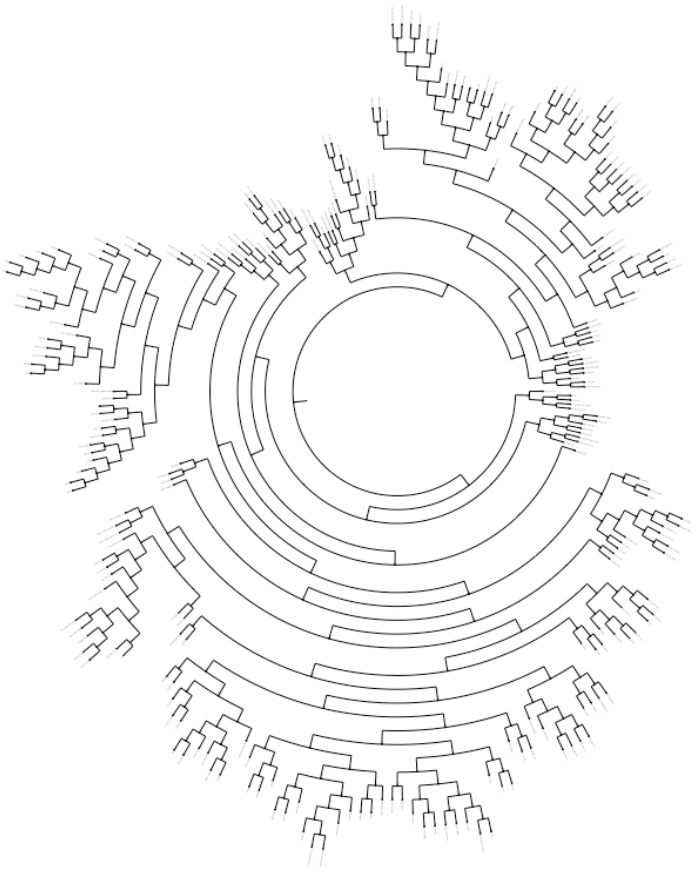


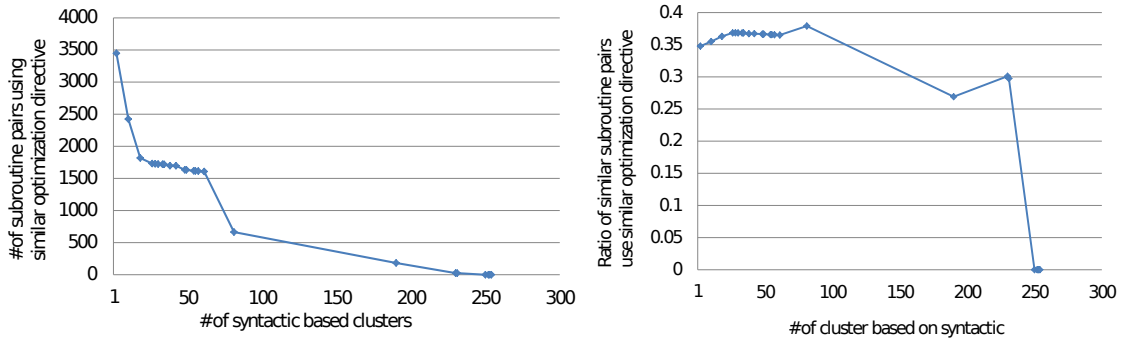
Figure 7.2: The overall family distance tree for GenIDLEST

Figure 7.3(a), we can the see the number of similar subroutine pairs using similar porting directives decreases gradually as the syntactic based cluster number increases. Figure 7.3(b) shows the ratio of similar subroutine pairs using similar porting directives over the total number of similar subroutine pairs from the "syntactic cluster". As we can see, the ratio is less than 40%, which means the porting accuracy is very low by only using cost-model-provided metrics for porting clustering.

To further divide hierarchical clustering into fine-grained *syntactic groups*, we propose three methods to cluster the tree, based on: a) the user inputs the tree

Table 7.1: GenIDLEST subroutines similarity statistics

Similarity Range	Number of subroutine pairs
Similarity $\geqslant 90$	47
Similarity $\geqslant 80$	43
Similarity $\geqslant 70$	44
Similarity $\geqslant 60$	208
Similarity $\geqslant 50$	985
Similarity < 50	30804

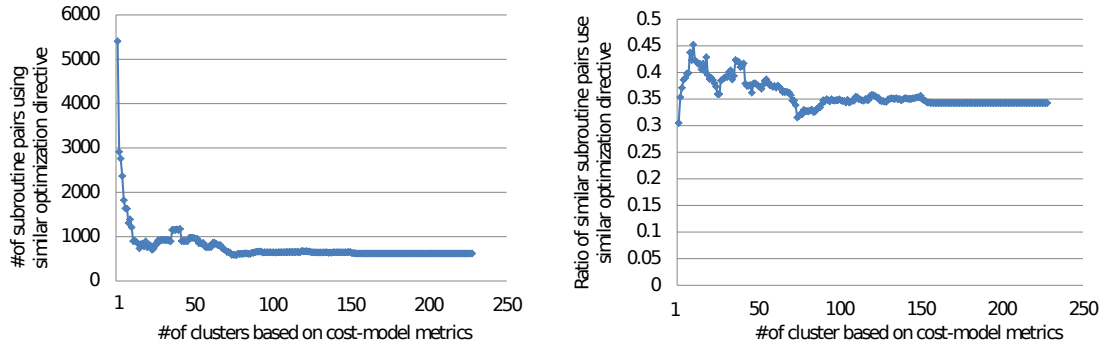


(a) (b)

Figure 7.3: Syntactic-based cluster for GenIDLEST application

depth value, which is used to divide the tree. b) a similarity distance value serves as a threshold to divide the tree: if the distance between current the node and its parent is greater than the distance threshold, then the current node and its descendants will be separated into a subtree. c) a combination of the first two methods; this combination method clusters the tree based on user input of tree depth and similarity distance. Our goal is to find a cluster number that is able to put syntactically similar subroutine pairs into groups as much as possible while maintaining a moderate group size. Based on previous empirical experience, a syntactic value of 50% is a suitable threshold [31], so in our experiment we use that threshold value and the input depth of the tree for clustering.

7.2 Cost-model Metrics Similarity Analysis



(a) (b)

Figure 7.4: Cost-model metrics-based cluster analysis for GenIDLEST

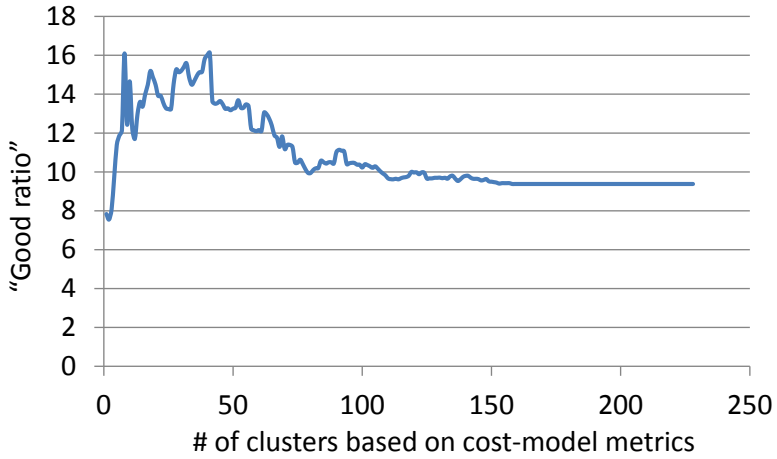


Figure 7.5: Cost-model metric-based "Good-ratios" diagram for GenIDLEST

The cost-model metrics used are estimated number of iterations, suggested parallelization, loop parallelizable attribute, loop vectorizable attribute, loop vectorized number, loop align peeled, work estimate, and loop depth. These metrics are key factors used in the cost model of OpenUH for optimization strategy selection and performance prediction, which can accurately capture the internal code optimization characteristics. To better understand the relationship between those metrics

and similar optimization or porting strategy, we only used the cost-model metrics to cluster subroutines and then check the number of subroutine pairs which use the same optimization directives or strategies. Figure 7.4(a) depicts the relationship between the number of subroutine pairs that use similar directives and the number of clusters, which is set manually based on the cost-model metrics. When manually setting the number of clusters based on the cost-model metrics, we can see the number of subroutine pairs using similar directive strategies decreases gradually until it reaches a constant. Figure 7.4(b) shows the ratio of subroutine pairs using a similar porting strategy over the total number of subroutine pairs that have been clustered with respect to different numbers of clusters. According to this result, we find that relying purely on cost-model-provided metrics for clustering subroutines results in low accuracy (below 46%) for detecting subroutine pairs that can be ported or optimized in the same way. To obtain a reasonable number of clusters for cost-model metrics, we define and use a "good ratio" to set the number of clusters. "good ratio" is a percentage score of the number of subroutine pairs with syntactic similarity greater than 50% over the total number of subroutine pairs in the clusters. We select a cluster number with the highest "good ratio" to make structurally similar subroutines aggregated as much as possible for similar porting experience reuse.

In Figure 7.5, the Y-axis is the percentage of the number of subroutine pairs with syntactic similarity greater than 50% over the total number of subroutine pairs in the clusters. We use the term "good ratio" to define this percentage score in the next text. The x-axis is the number of clusters manually set for clustering subroutines based on cost-model metrics In this diagram, we can see that the "good ratio" is around 16%

when the number of clusters is set to 8 and 41 respectively. When setting up the number of cluster based on cost-model-provided metrics, we want choose a cluster number which could result in "good ratio" while maintaining a moderate group size to void a scenario of generating too many combined clusters. Considering this, we set the number of cluster for cost-model metrics to 8 in our experiment.

7.3 Combination of Syntactic and Cost-model-based Clusters

Relying solely on either syntactic or cost-model-provided metrics results in low accuracy when detecting similar subroutine pairs that could be optimized or ported similarly. By merging these two metrics we can greatly increase the accuracy of the process of detecting subroutine pairs to be ported in the same way.

In Section 7.2, we found that we can get a "good ratio" by setting cost-model-provided cluster number to 8. To discover the relationship between those two clusters, we tried different combinations of numbers of clusters for the syntactic and cost-model-based clusters. To control the size of combined clusters, we set cost-model-provided clusters from 1 to 9 in the relationship of syntactic and cost-model cluster analysis. Our goal is to accurately aggregate similar subroutines into groups as much as possible, which provides the opportunity to find subroutines that can be optimized in the same way. In Figure 7.6, the X-axis is the ratio of the number of subroutine pairs with syntactic similarity more than 50% over the total number of subroutine

pairs based on current combined clustering methods. The Y-axis is the number of clusters obtained by using different distance values from 0 to 100. Inside each cluster, we vary the number of clusters based on the cost-model metrics, resulting in the "heart-beat" shape diagram. We observe that the ratio reaches a peak in this diagram when setting the cost-model metrics-based cluster to 8, which is exactly the number of cluster we can get peak "good ratio" value in our cost-model metrics analysis described in section 7.2.

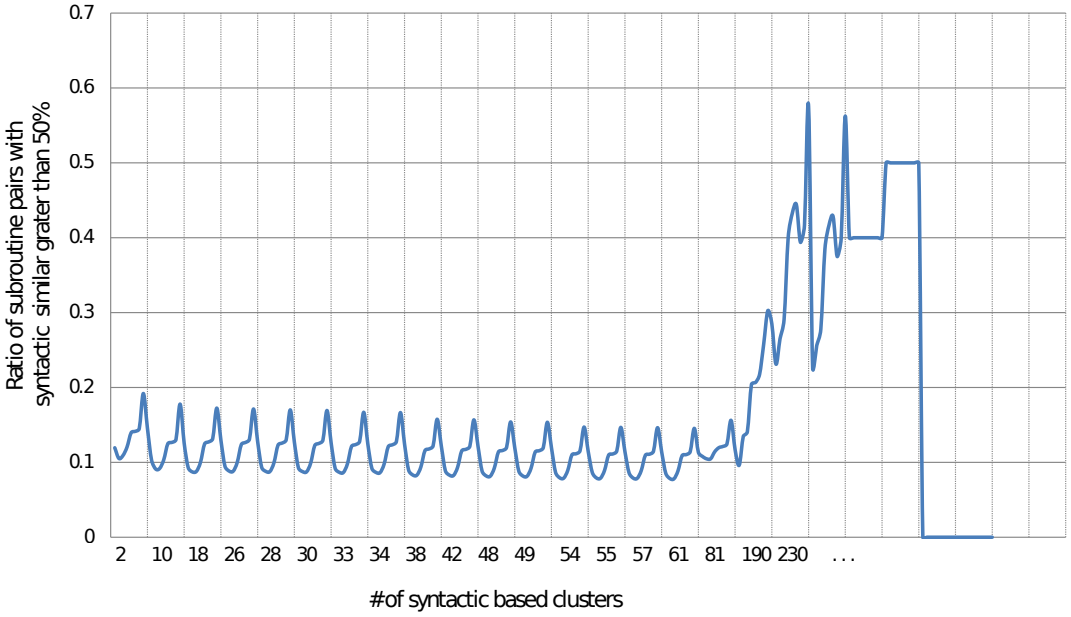


Figure 7.6: Combined syntax and cost-model metric clusters for GenIDLEST

7.4 Improved Verification Methodology

To increase the accuracy of testing the suggested plan for porting, our improved methodology focuses on the syntax of OpenMP directive comparison directly. We

add functions into the phase of code sequence extraction to extract the OpenMP directives to see if a subroutine pair is ported in the same way by checking the ported versions (described in section 4.2). If any OpenMP directive is detected in a subroutine, a separate ".opt" file will be generated; this is used to record a loop position index from its corresponding subroutine code sequence and optimization sequences by encoding OpenMP directives into sequences according to the code map defined in table 7.2.

Table 7.2: OpenMP Directive Encoding Code Map

Directives	**Character Map**
$!OMP PARALLEL	P
$!OMP DO	D
$!OMP PARALLEL DO	PD

Assume we have subroutines A and B in a combined cluster group. There are three cases that can be classified when comparing their similar optimization or porting strategy: 1) Neither A nor B have corresponding ".opt" files. We treat A and B in the same way, meaning neither of them could be optimized. 2) Only one of A and B has a ".opt" file, which means one was optimized and the other was not. Therefore A and B do not count as similar for optimization and do not use similar directives for porting. 3) Both A and B have ".opt" files. In this case, we perform code sequence alignments first. We are able to see which loops have been aligned by referring the loop index obtained from a code sequence back to the corresponding ".opt" file. For aligned loops, we check the OpenMP encoded directive sequences directly to check if two similar subroutines can have similar optimization directives applied to them for porting purpose.

7.5 Porting Strategy Verification

Based on analysis of clustering using syntactic and cost-model metrics listed in Figures 7.3 and 7.4, we found that either using syntactic distance or cost-model metrics alone as a clustering method will results in inaccurate clustering for the purposes of porting planning. Our goal is to minimize the number of clusters while at the same time to make sure accuracy for clustering similar subroutines using similar porting directives. Figure 7.6 shows that we can have maximum similar pairs ratios for subroutine pairs fall into the same syntactic and cost-model metrics-based clustering. So we set up the number of clusters based on cost-model metrics (or for short, cost-model cluster) to 8 in our experiment and then make a comparison of the accuracy of porting. By setting distance value to 50 and depth to 5 based on the shape of the tree, we are then able to divide the tree into 9 clusters for syntactic clustering. By merging the syntactic and cost-model metrics-based clustering, we divide the 254 subroutines into 25 groups. Subroutines within each group fall into the same syntactic and cost-model cluster. After combining syntactic and cost-model metric-based clusters or combined clusters, the next step is verifying the correctness of similar directives used for subroutines falling into combined clusters. The ratio of all subroutine pairs using similar optimization reaches 49.51% in our experiment. Figure 7.7(a) shows the relationship of similar optimization ratio over the 254 subroutines with respect to the syntactic similarity for subroutine pairs which fall into the same syntactic and cost-model cluster, when setting the cost-model metrics-based cluster number to 8. As the figure shows, the correctness of using similar directives for parallelizing the code is almost 80% for subroutine pairs which fall into the same syntactic and

cost-model cluster with syntactic similarity greater than 50%. Figure 7.7(b) shows the number of pairs using a similar porting strategy in detail.

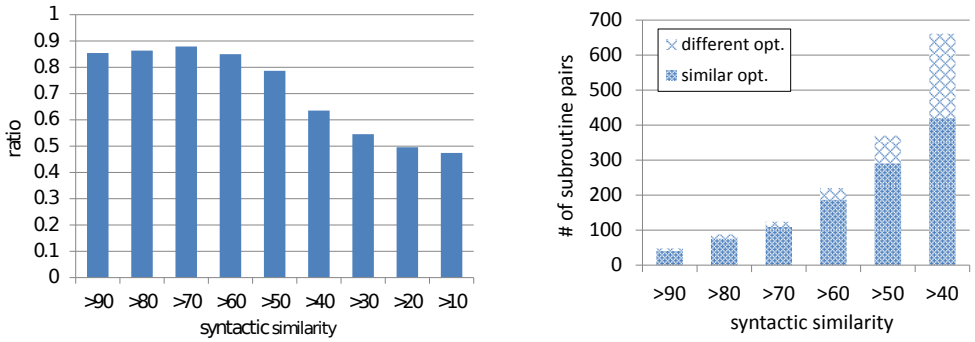


Figure 7.7: Verification of GenIDLEST porting planning analysis

Higher syntactic similarity will result in using a similar directive-based parallelization strategy for subroutine pairs with the same syntactic and code feature cluster. This result proves that our similarity based methodology is very effective and accurate in detecting similar subroutines which could use similar porting or optimization strategies. Cost-model-based metrics are accurate for capturing code similarity in terms of optimization or porting, which saves the trouble of running applications to collect profiling information.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

One novelty of our methodology is that our "similarity" is based on key program constructs derived from the program abstract syntax tree (AST), consisting of operators and data, which have an impact on the porting strategy beyond pure simple textual comparison of the code. The methodology is based on a hypothesis that we can port and tune *similar* subroutines of an application in a similar way while maintaining a good porting quality and improving programmer's productivity. In this dissertation, we show this hypothesis is generally true and verify this hypothesis by using NAS benchmarks and a real scientific application. Also, we extended the notion of similarity to other metrics than pure syntactic language constructs. We show that subroutines in scientific applications tend to be similar to each other syntactically due to practices used in the scientific community, such as reoccurring implementation

choices and code replication used for multi-versioning, and as a result of incrementally adapting codes to new platforms. In fact, we found out that when we view each subroutine as a bio-inspired DNA-like sequence, we are able to construct and define relationships among the sequences using bio-informatic techniques. For example, this allows us to build a "family distance tree" among the groups of subroutines based on their similarity. However, adaptation to traditional sequence alignment methods are needed, such as defining new scoring parameters and substitution matrices (i.e. replacing BLOSUM80, with observed software trends) suitable for software porting purposes. The tree provides valuable structural information that allows us to determine the porting order at the group level based on similarity and their distances. To the best of our knowledge, we are the first to explore this bio-inspired view of the source code. This is the second novelty of the work presented in this dissertation.

The similarity-based methodology is quite powerful. In this dissertation, we demonstrate how this methodology can be used not only for detecting similar code but as well as a "debugger" for newly ported codes. If we detect dissimilarities in the ported code of similar routines, this may indicate that some optimizations were missed by the user. We will then be able to check if it is simply due to an error or some other factors. Methodologically, we expect to see high similarity in ported subroutines of the same group. Other potential applications of the proposed methodology include allowing typical programmers to benefit from prior successful porting experiences of the heroic programmers, and allowing them to focus on critical areas of the code. The shape of the tree can also serve as an indicator of how difficult the porting process might be, and the diversity of the code involved in the application,

which could help to quantify the process in terms of resources.

In summary, the main contribution of this thesis is a bio-inspired similarity-based methodology for the planning support of scientific applications. The innovative aspects of the methodology are: (1) the similarity-based analysis for similar code porting that can lead to productive porting, (2) the casting of source code into DNA-like sequences for powerful similarity comparison, and (3) cost-model provided code feature information metrics are useful for detecting similar optimization or porting strategy and save the trouble of running applications. We also identify the advantages and limitations of this similarity analysis. To validate our similar porting strategy and evaluate the feasibility of the methodology, we implemented a software tool and used it for the entire National Aeronautics and Space (NSA) parallel benchmark suite (i.e., version 3.3 with 10 computational fluid dynamics (CFD) mini-applications). In order to test the scalability and adaptability of our methodology, we have also applied our tool to a real application called High-Order Multiscale Modelling Environment (HOMME) climate application which contains more than 700 subroutines. Finally we have validated *Klonos* by applying it to GenIDLEST, a real scientific application, that was originally written as serial code and then parallelized for a shared memory environment using OpenMP. By referring to the optimized OpenMP GenIDLEST code, we discovered that the OpenMP directives proposed by *Klonos* are both accurate and effective. This porting approach is quite easily extended to other directive based approaches for code migration to different architectures (e.g. PGI, OpenACC, and HMPP etc.). Future work will include exploring cost-models for porting code to other accelerators. We will also use data mining techniques to create a framework

which can automatically find combinations of syntactic and cost-model clusters to increase porting accuracy.

8.2 Future Work

In the future, we will modify the front-end of the OpenUH compiler to make it able to parse the OpenACC directives. Then we will apply the *Klonos* tool for code porting to other accelerators for the heterogeneous programming environment. Future work will include exploring cost-models for porting code to other accelerators. We will also use data mining techniques to create a framework which can automatically find combinations of syntactic and cost-model clusters to increase porting accuracy.

Also we are planning to build a database to store code sequences and its corresponding optimization strategies. A website will be provided to users to submit their code and search for similar optimization from the database, which could give users guidance for their code optimization and porting.

Bibliography

[1] Khronos Group. `http://www.khronos.org/`.

[2] Looptool:controlled loop fusion. `http://www.hipersoft.rice.edu/looptool/index.html`.

[3] The open64 compiler - whirl intermediate representation. `http://www.mcs.anl.gov/OpenAD/open64A.pdf`.

[4] OpenCL 1.0 Specification. `http://www.khronos.org/opencl/`.

[5] PGI Fortran & C Accelerator Compilers and Programming Model. `http://www.pgroup.com/lit/pgi_whitepaper_accpre.pdf`.

[6] Rapidmind. `http://www.rapidmind.net/`.

[7] Emboss: The european molecular biology open software suite (2000), 2000.

[8] Geneious: Inspirational software for biologists, 2005-2011.

[9] Openmp.org, 2008-2011.

[10] Codeanalyst user's manual, 2010.

[11] Jalview version 2 - a multiple sequence alignment editor and analysis workbench, 2011.

[12] F. Abbattista, A. Bianchi, and F. Lanubile. A storytest-driven approach to the migration of legacy systems. In *Agile Processes in Software Engineering and Extreme Programming*, volume 31 of *Lecture Notes in Business Information Processing*, pages 149–154. Springer Berlin Heidelberg, 2009.

[13] N. Adiga, G. Almasi, and G. Almasi. An overview of the bluegene/l supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–22, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[14] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies. Language and compiler design for streaming applications. *Int. J. Parallel Program.*, 33(2):261–278, 2005.

[15] M. L. G. at University of Waikato. Weka 3: Data mining software in java. `http://www.cs.waikato.ac.nz/ml/weka/`.

[16] D. Atkinson and W. Griswold. Effective pattern matching of source code using abstract syntax patterns. *Software: Practice and Experience*, 36(4):413–447, Apr. 2006.

[17] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The parascope editor: an interactive parallel programming tool. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 540–550, New York, NY, USA, 1989. ACM Press.

[18] F. Bodin, Y. Mével, and R. Quiniou. A user level program transformation tool. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 180–187, New York, NY, USA, 1998. ACM.

[19] M. Boekhold, I. Karkowski, and H. Corporaal. Transforming and parallelizing ansi c programs using pattern recognition. In *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes in Computer Science*, pages 673–682. Springer Berlin Heidelberg, 1999.

[20] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 77–86. IEEE, 2010.

[21] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[22] D. Buttlar, B. Nichols, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., 1996.

[23] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 47–56. ACM, 2011.

[24] K. Chaichoompu, S. Kittitornkun, and S. Tongsima. Mt-clustalw: multithreading multiple sequence alignment. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 254–254, Washington, DC, USA, 2006. IEEE Computer Society.

[25] B. Chapman and R. v. d. P. Gabriele Jost. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Cambridge Massachusetts London, England, 2007.

[26] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, June 2008.

[27] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. Hawkes, S. Klasky, W. Liao, K. Ma, J. Mellor-Crummey, N. Podhorszki, et al. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery*, 2(1):015001, 2009.

[28] N. Corporation. Nvidia visual profiler, 2013.

[29] P. D'Alberto and A. Nicolau. Adaptive strassen's matrix multiplication. In *ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing*, pages 284–292, New York, NY, USA, 2007. ACM.

[30] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42:36–42, 2009.

[31] W. Ding, O. Hernandez, and B. Chapman. A similarity-based analysis tool for porting openmp applications. In *Facing the Multicore-Challenge III*, pages 13–24. Springer, 2013.

[32] W. Ding, C.-H. Hsu, O. Hernandez, R. Graham, and B. M. Chapman. Bioinspired similarity-based planning support for the porting of scientific applications. In *4th Workshop on Parallel Architectures and Bioinspired Algorithms*, page n.pag, Galveston Island, Texas, USA, 2011.

[33] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *Proc. of Workshop on Parallel and Distributed Systems: Testing and Debugging (at IPDPS)*, page 289, 2003.

[34] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance*, August/September 1999.

[35] C. Enterprise. Hmpp: A hybrid multicore parallel programming platform. http://www.caps-entreprise.com/en/documentation/caps_hmpp_product_brief.pdf.

[36] O. R. L. C. Facility. Oak ridge national laboratory titan supercomputer. `http://www.olcf.ornl.gov/titan/`.

[37] M. Funaro, D. Braga, A. Campi, and C. Ghezzi. A hybrid approach (syntactic and textual) to clone detection. In *Proceedings of the 4th International Workshop on Software Clones*, pages 79–80. ACM, 2010.

[38] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[39] P. Grogono. Comments, assertions and pragmas. *SIGPLAN Not.*, 24(3):79–84, 1989.

[40] W. D. Gropp and E. Lusk. Reproducible measurements of MPI performance characteristics. In J. Dongarra, E. Luque, and T. Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 11–18. Springer Verlag, 1999. 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 1999.

[41] T. P. Group. PGI accelerator compilers, 2010. http://www.pgroup.com/resources/accel.htm.

[42] D. Grove and P. Coddington. Precise mpi performance measurement using mpibench. In *Proceedings of HPC Asia*, pages 24–28. Citeseer, 2001.

[43] T. D. Han and T. S. Abdelrahman. /hi/cuda: a high-level directive-based language for gpu programming. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61, New York, NY, USA, 2009. ACM.

[44] M. Hennessy and J. F. Power. Ensuring behavioural equivalence in test-driven porting. In *CASCON*, page 377, 2006.

[45] M. Ishihara, H. Honda, and M. Sato. Development and implementation of an interactive parallelization assistance tool for openmp: ipat/omp. *IEICE - Trans. Inf. Syst.*, E89-D(2):399–407, 2006.

[46] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS Parallel Bbenchmarks and its performance. Technical Report NAS-99-011, NAS System Division, NASA Ames Research Center, Oct. 1999.

[47] H. Jin, M. A. Frumkin, and J. Yan. Automatic generation of OpenMP directives and its application to computational fluid dynamics codes. In *High Performance Computing, Third International Symposium, ISHPC 2000, Tokyo, Japan*, volume 1940 of *Lecture Notes in Computer Science*, pages 440–456. Springer, October 2000.

[48] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of International Conference on Software Maintenance*, pages 120–126. IEEE, 1994.

[49] S. P. Johnson, E. Evans, H. Jin, and C. S. Ierotheou. The parawise expert assistant - widening accessibility to efficient and scalable tool generated OpenMP code. In *WOMPAT*, pages 67–82, 2004.

[50] C. Kartsaklis, O. Hernandez, C.-H. Hsu, T. Ilsche, W. Joubert, and R. L. Graham. Hercules: A pattern driven code transformation system. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 574–583. IEEE, 2012.

[51] C. Keler and W. Paul. Automatic parallelization by pattern-matching. In *Parallel Computation*, volume 734 of *Lecture Notes in Computer Science*, pages 166–181. Springer Berlin Heidelberg, 1993.

[52] R. Koschke. Frontiers of software clone management. In *Frontiers of Software Maintenance*, pages 119–128. IEEE, 2008.

[53] S. Lee and J. S. Vetter. Early evaluation of directive-based gpu programming models for productive exascale computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 23:1–23:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[54] J. M. Levesque, R. Sankaran, and R. Grout. Hybridizing s3d into an exascale application using openacc: an approach for moving to multi-petaflops and beyond. In *Proceedings of the International Conference on High Performance*

Computing, Networking, Storage and Analysis, SC '12, pages 15:1–15:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[55] J. E. Luke Sheneman and J. A. Foster. Clearcut: a fast implementation of relaxed neighbor joining. *Bioinformatics*, 22(22):2823 – 2824, 2006.

[56] B. D. Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *International Workshop on Program Comprehension*, May 1996.

[57] M. D. McCool and S. Du Toit. *Metaprogramming GPUs with Sh*. AK Peters Wellesley, 2004.

[58] R. Membarth, F. Hannig, J. Teich, M. Korner, and W. Eckert. Frameworks for gpu accelerators: A comprehensive evaluation using 2d/3d image registration. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 78–81. IEEE, 2011.

[59] D. W. Mount. *Bioinformatics: sequence and genome analysis*. CSHL Press, 2004.

[60] P. Mucci and K. London. The MPBench report, 1998.

[61] A. Munshi. *The OpenCL Specification Version 1.0, Document Revision 48*, 6 Oct. 2009.

[62] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[63] S. B. Needleman, C. D. Wunsch, et al. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[64] NVIDIA. CUDA. `http://www.nvidia.com/object/cuda_home.html`.

[65] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide version 3.0. http://developer.nvidia.com/cuda, Mar. 2010.

[66] OpenACC. Openacc: Directives for accelerators, 2011.

[67] The OpenUH compiler project, 2005.

[68] I. Park, M. Voss, B. Armstrong, and R. Eigenmann. Parallel programming and performance evaluation with the URSA tool family. *Int. J. Parallel Program.*, 26(5):541–561, 1998.

[69] Perfsuite. `http://perfsuite.ncsa.uiuc.edu/`.

[70] G. C. Project. Thrust: Thrust Quick Start Guide. `http://code.google.com/p/thrust/`.

[71] A. Qasem, G. Jin, and J. Mellor-Crummey. Improving Performance with Integrated Program Transformations. Technical Report TR03-419, Rice University, Oct. 2003.

[72] D. Quinlan. Rose: Compiler support for object-oriented frameworks. In *Proceedings of Conference on Parallel Compilers (CPC2000)*, 2000.

[73] S. R and M. C. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin 38*, pages 1409 – 1438, 1958.

[74] S. P. Reiss. Specifying what to search for. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 41–44, Washington, DC, USA, 2009. IEEE Computer Society.

[75] R. Reussner, P. Sanders, L. Prechelt, and M. Muller. SKaMPI: A detailed, accurate MPI benchmark. In *PVM/MPI*, pages 52–59, 1998.

[76] C. Roy and J. Cordy. A survey on software clone detection research. Technical Report 2007–541, Queen's University at Kingston Ontario, Canada, Sept. 2007.

[77] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 81–90, Washington, DC, USA, 2008. IEEE Computer Society.

[78] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987.

[79] S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[80] R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *International Workshop on Software Clones*, page n.pag, Mar. 2009.

[81] D. Tafti. Genidlest a parallel high performance computational infrastructure for simulating complex turbulent flow and heat transfer. In *APS Division of Fluid Dynamics Meeting Abstracts*, volume 1, 2002.

[82] P. Varma. Process and planning support for iterative porting. *Journal of Information Science and Engineering*, 22(2):337–356, Mar. 2006.

[83] P. Varma, A. Anand, D. P. Pazel, and B. R. Tibbitts. Nextgen extreme porting: structured by automation. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1511–1517. ACM, 2005.

[84] S. Vetter, Y. Aoyama, and J. Nakano. *RS/6000 SP: Practical MPI Programming*, volume SG24-5380-00 of *0738413658*. vervante, August 1999.

[85] A. Walenstein, M. El-Ramly, J. Cordy, W. Evans, K. Mahdavi, M. Pizka, G. Ramalingam, J. W. von Gudenberg, and T. Kamiya. Similarity in programs. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Apr. 2007.

[86] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J.-A. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[87] F. WM and M. E. Construction of phylogenetic trees. *Science*, 155(3760):279 – 284, 1967.

[88] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.

[89] G. Xie, J. Chen, and I. Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 51–60. IEEE, 2009.

[90] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.