A Similarity-Based Analysis Tool for Porting OpenMP Applications *

Wei Ding, Oscar Hernandez, and Barbara Chapman

Dept. of Computer Science, University of Houston
Oak Ridge National Laboratory
{wding3,chapman}@cs.uh.edu,
{oscar}@ornl.gov

Abstract. Exascale computers are expected to exhibit an unprecedented level of complexity, thereby posing significant challenges for porting applications to these new systems. One of the ways to support this transition is to create tools that allow their users to benefit from prior successful porting experiences. The key to such an approach is the manner in which we define source code similarity, and whether similar codes can be ported in the same way to a given system. In this paper, we propose a novel approach based on the notion of similarity that uses static and dynamic code features to check if two serial subroutines can be ported with the same OpenMP strategy. Our approach creates an annotated family distance tree based on the syntactic structure of subroutines, where subroutines that belong to the same syntactic family and share the similar code features have a greater potential to be optimized in the same way. We describe the design and implementation of a tool, based upon a compiler and performance tool, that is used to gather the data to build this porting planning tree. We then validate our approach by analyzing the similarity in subroutines of the serial version of the NAS benchmarks and comparing how they were ported in the OpenMP version of the suite.

1 Introduction

The High Performance Computing (HPC) community is heading toward the era of exascale computing. An exascale machine and the programming models deployed on the machine are expected to exhibit a hitherto unprecedented level of complexity. New tools will be needed to help the application scientist restructure programs in order to exploit these emerging architectures, improve the quality of their code and enhance its modularization to facilitate maintenance. As scientists port their application codes to new systems, they must restructure their applications to exploit the multi-cores available in each node. OpenMP [13], a de-facto standard for shared memory programming, is widely used due to its

* This work was funded by the ORAU/ORNL HPC grant and NSF grant CCF-0917285. This research used resources of the Leadership Computing Facility at Oak Ridge National Laboratory and NICS Nautilus supercomputer for the data analysis.

simplicity, incremental parallelism, and wide availability. However this simplicity comes at a performance cost. For example, programmers need to take care of false sharing issues and apply aggressive data privatization to achieve good performance.

During the porting process of parallelizing serial codes with OpenMP, we observed that, in some applications, many of the porting strategies are highly repetitive, although not necessarily easily detected. For example, many similar computational kernels that need to be parallelized with OpenMP may appear in different parts of the application, where code is slightly modified to meet specific computational needs. Locating and porting these code regions is a time-consuming and error prone process that needs to be systematically addressed.

One way to improve the productivity and reduce the potential problems is to create tools that allow users to benefit from prior successful porting experiences that can be applied to multiple code regions in an application. In this paper, we explore different code characteristics similarities and make a lot of improvement based on a tool called KLONOS [6] that can better assist user to accurately port their applications by combining the notion of syntactic code similarity. An adaptation strategy that is successful for one code region could then also be applied to similar regions. We use this tool to demonstrate how the concept of similarity can be used to parallelize serial codes with OpenMP more easily and productively. With this goal in mind, we have developed a tool that is able to classify subroutines in similar families of codes that may follow the same optimization strategy. Our tool uses the concept of code syntactic distance and a k-means clustering approach to classify codes based on their static and dynamic features (such as parallelization and hardware counters profiling information). Our hypothesis is that we can classify the subroutines of an application with a family distance tree, where subroutines that share similar syntactic structure and program features will have a high likelihood to be ported in the same way. In this paper we focus specifically on finding similar codes that can be ported to the shared memory systems using OpenMP with the same strategy.

This paper is organized as follows: Section 2 helps further explain the motivation of our work. Section 3 summarizes the related work on similarity research and current practice for the software porting. Section 4 describes the functionality and design of the tool we build to detect similar fragments of code. The implementation, including the compiler infrastructure that it is based upon, is then introduced in section 5. Section 6 describes our evaluation of the KLONOS tool using the serial and OpenMP version of the NAS benchmarks. Finally, Section 7 briefly discusses our conclusions and future work.

2 A Motivating Example

A byproduct of the efforts to enhance and port applications, particularly as a result of incremental improvements, is an increasing likelihood of code similarity in an application. Developers may implement similar versions of an algorithm, redevelop parts of it, copy and paste code multiple times and adapt it for specific

```
y_solve_: EZTTNBYEZZZ-----TGYZZZTJYMM-ZTTBGYEZZZTJY--------MZTTGGYEZZZTG
          ...||||||||||     |||||||  |||  ||||||||||| |        ||||||||||||...
x_solve_: EZTTNBYEZZZYMZZZTGYZZZTGYMMSZTTBGYEZZZTGYZZZTBYDSMZTTGGYEZZZTJ
```

Length of alignment = 3003

Fig. 1. NAS BT OpenMP benchmark x_sove, y_sove sequences alignment

needs, or use a code transformation tool that creates many instances of similar code. Therefore, there is a possibility that we may be able to reuse given porting strategies on similar codes and benefit from prior porting experiences. To illustrate our motivation, we use the BT serial NAS benchmark to demonstrate how our notion of similarity can be used to help the user parallelize this code with OpenMP. BT belongs to a family of CFD code that uses a multi-partiton scheme

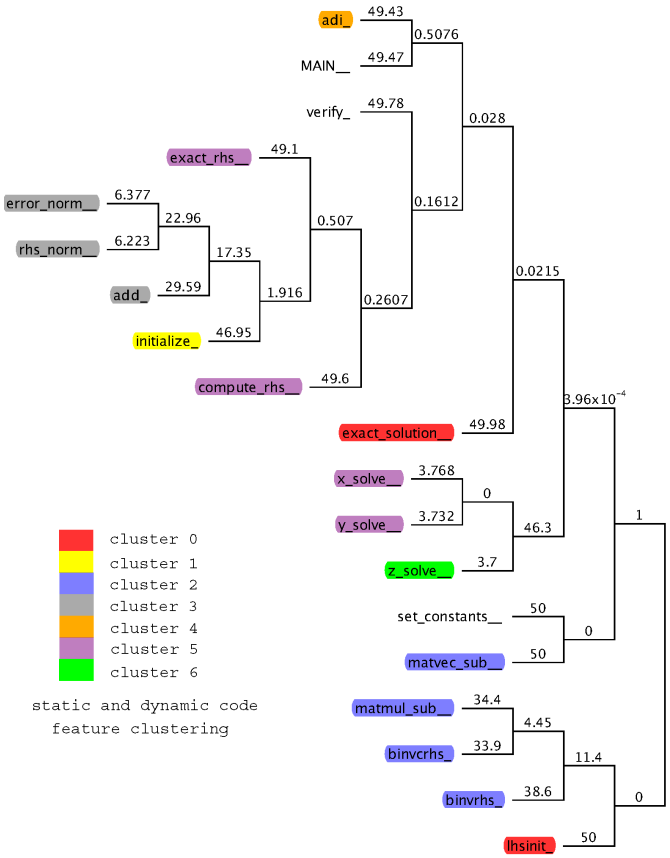


Fig. 2. NAS BT benchmark porting planning tree

to solve three sets of uncouple systems of equations in a block triagonal 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 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 their percent of identity. Figure 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 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 in the tree. *x_solve*, *y_solve* are grouped into one subtree, and their parent node is grouped with *z_solve* into another subtree. By calculating the distance among these three subroutines, we get *x_solve* $\sim$ *y_solve*=7.5, *x_solve* $\sim$ *z_solve*=7.468, *y_solve* $\sim$ *z_solve*=7.432. These subroutines have small distances among 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 discontinguous memory accesses. This may impact the optimization strategy for these subroutines on a cache based system. For example, the OpenUH [14] compiler optimizes the serial version of *x_solve, y_solve* similarly (when inspecting their intermediate representations 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 (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 the them to further classify the subroutines. For the BT Benchmark, we clustered its subrotuines based on the number of parallel loops, data cache accesses and 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

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 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 2. Cluster center point for the serial BT benchmark based on the code features

points is small. The result of the cluster is shown in Table 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 2 shows the list of code features of each subroutine that is 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. Besides, those memory events have direct link with the optimization which contributes to the final performance. For the AMD Family 10h processors, although IBS is a statistical method, the sampling technique delivers precise event information and eliminates inaccuracies due to skid [2]. Using the clustering results we can annotate families of codes that share important code features for OpenMP optimizations. Figure 2 shows the resulting annotated serial BT benchmark porting planning tree we get 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 features clusters although their syntactic distance is small, with *x_solve*, *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 used the same parallelization strategy applies for the subroutines

```
!$omp parallel default(shared)
!$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

Fig. 3. subroutine rhs_norm and error_norm code snippet of the NAS BT benchmark

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 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.

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 paper 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 with the same OpenMP strategy.

3 Related Work

Similarity analysis is one of the techniques that is used to identify the code regions that are similar in a given program. One early use was to identify students who copied code from others in a programming assignment; later some researchers applied this technique in the software engineering to detect the redundant code for the code maintenance. This technique proved very effective detecting code clones, especially for a large legacy application developed by a team of people over a long period, which makes the code maintenance very difficulty. Besides code maintenance, TSF [4] and other work [10] also developed a notion of similarity in order to detect related code regions for the purpose of applying transformations. Such analysis typically examined loop nests, their nesting depth and certain details of the data usage patterns they contained. There is no *precise* definition of similarity between programs [19], in part because the appropriateness of any given metric depends on the context in which it is being used [18]. In addition to syntactic approaches based upon the source text, graph-based approaches that use data flow and control flow information have been employed to detect sections of code that are similar. Duplix [11] is a tool that identified "similar" code regions in programs, the idea is based on finding subgraphs that are similar stemming from duplicated code in fine-grained program dependency graphs. In this work, identified subgraphs could be directly mapped back onto the program code and presented to the user. Walenstein et al. [19] considered *representational* similarity and *behaviorial* similarity. Roy et al. [16] proposed four categories of clones to differentiate the level of code clones. I [9] presented a scalable clone detection algorithm that helped in reducing a complex graph similarity problem to a simpler tree similarity problem.

Although some of these prior efforts have focused on helping to restructure applications by detecting code clones at the syntactic level, few have attempted to detect 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 the code optimization characteristics. We ultimately need to develop 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. The Milepost/Collective tuning project [8] proposed 56 metrics for each subroutine. It used probabilistic and transductive machine learning models to search for similar compiling flags from similar subroutines

based on previous experience on which flags yield good performance. Their proposed method is able to predict the performance, and normally the suggested flags are good for contributing good performance. However, their approach does not consider syntactic similarity and code features that are relevant for a specific optimization or porting goal (i.e. OpenMP parallelization, etc.). In their automated framework, the user has little input on what optimizations, and the exact code transformations / optimizations that lead to good performance, other than compiler passes and flags, are not clear.

4 Design of Our System

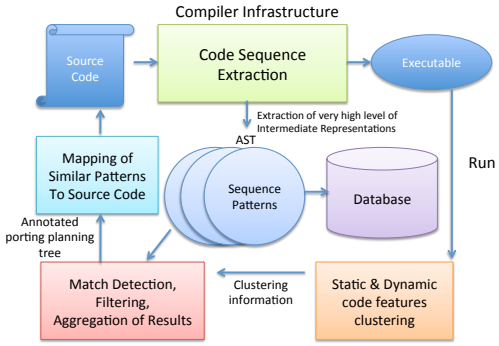


Fig. 4. The KLONOS Porting Planning System

In order to automate the application porting analysis, we designed a porting planning system. Figure 4 shows the overall design of "KLONOS", our system. We have adapted an existing compiler, OpenUH, to create a tool that is capable of detecting similar codes based on static and dynamic information. The user submits a code to the modified compiler for the code pattern extraction, then `KLONOS` extracts "sequence patterns" at a very high level just after the internal WHIRL tree structure has been generated by the compiler. The compiler then produces an executable which is run, and performance metrics are collected. The generated sequences are subsequently input to an alignment engine, which returns the a syntactic similarity score that is used to construct a family distance. Then the user extracts other code features such as the number of parallel loops in the subroutines and hardware counter information which is fed into the "code feature similarity matching engine" to cluster the subroutines based on these code features. As a last step, the system analyzes and processes the family distance tree together with the code feature clustering, and outputs an annotated distance tree that can be used to develop a porting plan to incrementally add OpenMP in an application. In the pattern extraction phase, we perform this operation by analyzing the code at this first level of representation, which is closely related to the source program form called the Abstract Syntax

Tree (AST). The AST is then "lowered" into a representation that is language independent and may be used to optimize codes written in multiple languages. Once the extraction is completed, our sequence pattern representation is input to a sequence alignment program called EMBOSS [1] to calculate the similarity score which is used to evaluate the degree of syntactic similarity.

In the design of our system we leverage the bioinformatic techniques for multiple sequence alignments. Compared with other approaches that generate program graphs for graph comparison, this gives us the freedom to compare large code without being constrained by graph size and graph complexity.

5 Implementation

Our implementation is based on OpenUH, an open source research compiler suite for C, C++ and Fortran 95, OpenMP 3.0 and the IA-64 Linux ABI and API standards. The compiler first translates different languages to a high level intermediate format (IR), called WHIRL [5]. 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 [7].

Building the distance matrix: After extracting the sequences, we perform a pair-wise global alignment to compare the degree of syntactic similarity between the subroutines. For this, we used the Needleman-Wunsch algorithm [12] 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 substracting the percent of identity with 1 and then times 100.

Constructing the family distance tree: There are several algorithms [15, 17, 20] that can be used to classify sequences into distance trees. In our case, we used the Neighbor-joining [17] 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.

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 extract this information from the compiler and

gather hardware counter data when we execute 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 get those metrics, we use Weka [3] 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 append 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.

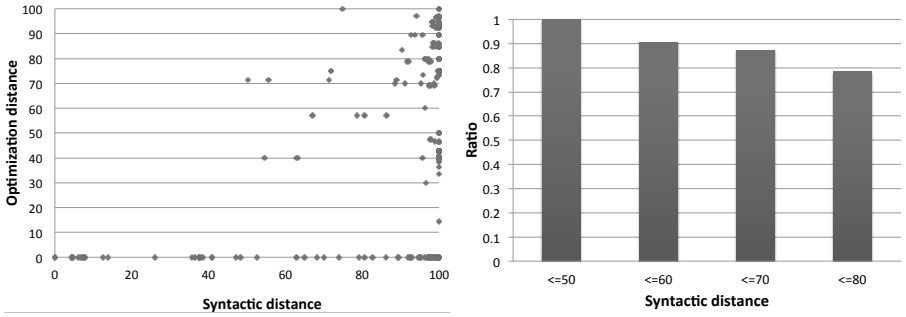
Verification of optimization strategy: In order to further verify this hypothesis, we use a similar concept for extracting the code sequence. To generate the optimization distance, we first trace the functions in charge of the OpenMP transformation. We used an unique letter to denote each function called during the OpenMP translation phase. Those functions are responsible for translating a given OpenMP construct. So, 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 NAS based on the optimization similarity score as the steps described above. The generated optimization distance is then used to check if two codes were lowered and optimized in the same way.

6 Experiments

NAS Benchmark-3.3 has nine benchmarks. We focused on the BT, CG, FT, MG and SP, UA because they are written in Fortran. KLONOS is able to support codes C/C++ and Fortran, but we have only verified it with Fortran at this stage in our experiment. As described in Section 5, we first extract all the subroutine sequence patterns and collect the static and dynamic program features which include hardware counter information. We then use this information to build the annotated distance tree for optimization planning support.

In Section 2, we explained how we used the similarity technique to find the similar porting strategy which could can be applied to the similar subroutines inside the BT benchmark. Due to the 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 5, 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* is the syntactic distance which shows the syntatic distance between a pair of subroutines, and the y-axis is the optimization distance used to represent the optimization similarity of the comparison of two subroutines.

According to Figure 5(a), we observe that a syntactic distance of 50 is an appropriate threshold for the NAS benchmark. For the subroutine pairs with a



(a) Syntactic and optimization distance for a pair of subroutines

(b) Percentage diagram for the syntactic distance

Fig. 5. NAS Benchmark-3.3 Experimental result

syntactic distance less than 50, they are more likely to have an identical OpenMP optimization strategy. Figure 5(b) shows the percentage of subroutine pair with their optimization distance equals to zero when their syntactic distance are less than 50, 60, 70 and 80 respectively. When the subroutine pairs have syntactic distance less than 50 (or a percent of similarity score is greater than 50), we found that they all 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 change when the syntactic distance goes beyond 50. And this trend continues, as the code structure further diverges, the optimization similarity keeps decreasing. This is generally true that different subroutines are optimized differently. Back to the generated family distance tree, we verified that all the subroutine pairs, which 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 diverges dramatically from each other.

7 Conclusions and Future Work

We have further expanded the notion of code similarity by exploring different dynamic code metrics for similarity in the context of accurately helping users port their codes to a multicore system using OpenMP. We use the concept of syntactic distance to check codes for the degree of similarity in their syntactic structure. We then extract static and dynamic program features to describe the parallelism and data accesses of the codes. Finally, we cluster the codes based on these code features, and annotate the family distance tree, where codes that belong to the same family and cluster can be ported in the same way with OpenMP. We validate our results with the NAS parallel benchmarks.

Future work will include exploring whether two similar codes maintain their syntactic similarity during critical compilation phases inside a compiler. If this

new measure is utilized, we can guarantee that two similar codes can be optimized in the same way by the user or compiler on a target platform. We will also explore the use of code feature sets that are relevant to a given optimization goal (i.e. porting by using OpenMP or GPU directives.).

References

1. Emboss: The european molecular biology open software suite (2000), 2000.
2. Codeanalyst user's manual, 2010.
3. Machine Learning Group at University of Waikato. Weka 3: Data mining software in java. `http://www.cs.waikato.ac.nz/ml/weka/`.
4. F. Bodin, Y. Mével, and R. Quiniou. A user level program transformation tool. In *International Conference on Supercomputing*, July 1998.
5. Open64 Compiler. Open64 compileri, whirl intermediate representation. `http://www.mcs.anl.gov/OpenAD/open64A.pdf`.
6. Wei Ding, Chung-Hsing Hsu, Oscar Hernandez, Barbara Chapman, and Richard Graham. Klonos: Similarity-based planning tool support for porting scientific applications. *Concurrency and Computation: Practice and Experience*, 2012.
7. Wei Ding, Chung-Hsing Hsu, Oscar Hernandez, Richard Graham, and Barbara M. Chapman. Bioinspired similarity-based planning support for the porting of scientific applications. In *4th Workshop on Parallel Architectures and Bioinspired Algorithms*, Galveston Island, Texas, USA, 2011.
8. Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization*.
9. M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330. ACM, 2008.
10. Christoph Kessler. Parallelize automatically by pattern matching, 2000.
11. J. Krinke. Identifying similar code with program dependence graphs. In *wcre*, page 301. Published by the IEEE Computer Society, 2001.
12. Saul B. Needleman and Christian 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.
13. OpenMP: Simple, portable, scalable SMP programming. `http://www.openmp.org`, 2006.
14. The OpenUH compiler project. `http://www.cs.uh.edu/~openuh`, 2005.
15. Sokal R and Michener C. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin 38*, pages 1409 – 1438, 1958.
16. Chanchal K. Roy and James 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.
17. 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.
18. R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *International Workshop on Software Clones*, March 2009.
19. A. Walenstein and M. El-Ramly et al. Similarity in programs. In *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, April 2007.
20. Fitch WM and Margoliash E. Construction of phylogenetic trees. *Science*, 155(3760):279 – 284, 1967.