

Porting Applications with OpenMP Using Similarity Analysis

Wei Ding^{1,2(✉)}, Oscar Hernandez^{1,2}, Tony Curtis^{1,2},
and Barbara Chapman^{1,2}

¹ Department of Computer Science, University of Houston, Houston, USA

² Oak Ridge National Laboratory, Oak Ridge, USA

{wding3,tonyc,chapman}@cs.uh.edu,
oscar@ornl.gov

Abstract. Computer architecture has undergone dramatic changes due to technology innovation. Some emerging architectures, such as GPUs and MICs also have been successfully used for parallel computation in the today's HPC field. Nowadays, people frequently have to port application to a new architecture or system and to expand its functionality for a better performance while in the meantime to meet the new hardware environment need. However, many scientific application legacy codes have a relative large size and long development cycle, so it's a very challenging job to port legacy codes to a new environment. And current codes porting process is a manual, time-consuming, expensive and error-prone process, which requires a team of people work together. Barely any useful tools can be used to ease the porting process in High Performance Computing (HPC). In this paper, we present a tool called *Klonos*, which is designed for assisting scientific application porting. Based on similarity analysis of code syntax and cost-model provided metrics, we are able to find codes which can be optimized similarly without the need of profiling the codes. The proposed porting plan can systematically guide users for selecting subroutines in a way which maximizes the reuse of similar porting strategy. We evaluate *Klonos* by applying it to a real scientific application porting to a shared memory environment using OpenMP. According to our experiment result, which shows that *Klonos* is very accurate to detect similar codes which can be ported similarly.

1 Introduction

HPC systems have been continually evolving, driven by technology innovation in computer hardware, operating systems, network protocols, and system libraries. As a result, applications that have been developed and tuned for older systems often require significant code changes to utilize the capabilities of the newer systems. The process of code changes for a new system is called *software porting*.

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

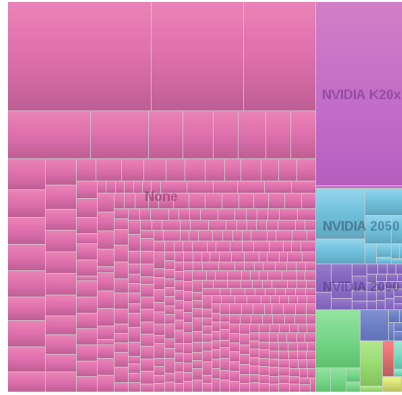


Fig. 1. Accelerator/Co-Processor Treemap of Top500 released in November 2012

In software engineering, *software porting* refers to the process of adapting software originally designed for one computing environment so that an executable program can be created for a different computing environment [28]. This process is a task that frequently arises in HPC, and it poses particular challenges in this domain.

On the roadmap toward exascale computing, a major challenge with respect to supercomputer design is the need to provide higher levels of computational power at dramatically lower rates of power consumption. The use of GPUs as co-processors is rapidly becoming a popular and powerful way to perform parallel computation. In Fig. 1, according to the Treemap (Accelerator/Co-Processor) of the Top500 List [19], released in November 2012, we see that the emerging co-processor such as GPUs and MICs are playing a greatly increased role in the supercomputers listed in the Top500. On the one hand, these co-processor- (or vector processor-) based heterogeneous systems provide more computation and power balance, but on the other hand, complicate programming models, which makes application porting more challenging than ever.

The Titan supercomputer at Oak Ridge National Laboratory is equipped with NVIDIA graphics processing units (Kepler K20x GPUs). In order to migrate codes to Titan, scientists 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 movement, exploiting complex memory subsystems, and mapping work to balance the overall load. They may need to use a hybrid programming model, such as adding OpenMP [1,3] and accelerator directives [12,13] to MPI applications [27], or they may introduce Pthreads [4] or an API designed for accelerators [22,23].

CUDA and OpenCL are two popular programming APIs specifically used for GPU programming, however programmers often have to restructure and write kernels for running regions of code on GPU. Worse still, the syntax of CUDA or

OpenCL code is quite different from the traditional C and Fortran languages, which makes it almost impossible for programmers with no CUDA or OpenCL background to understand and maintain the code. Directives-based programming models like OpenMP, HMPP, PGI and OpenACC for GPU programming that can greatly increase programming productivity, have been proposed to face such challenges. Reference [14] have explored and compared popular programming models used for GPUs, showing that a directives-based approach is able to achieve similar or even better performance compared to CUDA and OpenCL. By raising the level of abstraction, directives-based models will enable incremental development and increase programming productivity, fast prototyping and retargetability for future new development environment and hardware. Reference [18] summarizes the authors' experience of porting a simulation of turbulent combustion application to a GPU by using OpenACC. Although directives-based programming models to some extent ease the programmability burden, the whole porting process is still manual, time-consuming and error-prone.

Profiling tools are used to find computationally intensive code regions and then offload them to GPUs, followed by either a manual or compiler-driven restructuring for performance tuning. The quality of code porting relies solely on the user's programming experience. There are very few tools that can assist the porting process. The whole process requires a lot of work, and worse still, neither programmers or compilers can reuse previous experience for structurally similar code. In order to ease the process of porting software to a new system, we have created a tool called *Klonos*, which is able to provide a porting plan based on similarity analysis. This tool allows programmers and compilers to reuse porting experience as much as possible during the porting process. In this paper, we use the OpenMP programming model as an example to show how we can apply *Klonos* for porting serial code to a shared-memory programming environment. The main contributions of this paper are that: (1) 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; (2) we propose a method for combining syntactic and cost-model-provided metrics clusters which aggregate similar subroutines that can be ported similarly. (3) we validate the *Klonos* tool by applying it to a large scientific application that is in production use. Our experiments shows that *Klonos* is an accurate tool for detecting subroutines that can benefit from similar porting strategies, and which reuse the programmers' or compiler's porting experience as much as possible. For clarity, Table 1 explains terms used in this paper.

This paper is organized as follows: Sect. 2 summarizes related work for the software porting. Section 3 describes the framework of *Klonos* and the cost model metrics which we introduced for detecting subroutine similarity in terms of porting or optimization. Section 4 evaluates *Klonos* tool for porting a real application called GenIDLEST to a shared programming environment by using OpenMP. Section 5 is the conclusion and future work.

Table 1. Terminology used in the Klonos tool

Term	Description
Similarity	A percentage score used to describe the match between a pair of sequences
Similarity distance matrix	A matrix (two-dimensional array) containing the distances, taken pairwise, of a set of subroutines. Matrix size is $N \times N$, 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
Porting cluster	A group of clusters with subroutines in each cluster share the same syntactic and cost-model provided metrics clusters
Porting plan	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

2 Related Work

Various techniques are used to port software from one environment to another. Two of these techniques used in the evolution of legacy codes are software refactoring and re-engineering. Additionally, a directives-based approach is used to guide the compiler while minimizing code changes and retaining the original syntax. With the help of code transformation tools, such restructuring work can be carried out (semi-)automatically, greatly improving work productivity.

Software refactoring is an important technique for the development and evolution of complex software systems. This technology saves development time and effort by reusing much of the existing design and code. Reference [20] uses some design tactics that assist users when evolving code from an existing software system, rather than starting from scratch. However based on their proposed approach, the onus is on the programmer to build a case model and object-oriented design model. The development team must also go manually through a discovery process to determine the structure of the code [8]. The discovery process is difficult and time-consuming, and it also not trivial to determine architecture features from the source code. Software re-engineering is the examination, analysis and alternation of an existing software system to reconstitute it for a new system. But this technique usually comes with extremely high manual re-engineering costs, and it's hard to get a global view for code, data, process re-engineering. Additionally

the re-engineered system might perform inadequately. We still need a tool to accurately provide us a code review.

A directives-based programming approach can be used to increase programmability and keep code concise. OpenMP serves as the de facto directives-based standard for parallel programming on shared memory systems, and is now deployed beyond pure HPC to include embedded systems, real time systems, and accelerators [6]. This directives-based approach greatly increases programming productivity, although it is not easy to write highly efficient code simply by adding directives, as unexpected overheads or side-effects may be introduced. In order to remedy this, several tools have been proposed: [16] develops an environment integrated with a tool called CAPO [5] where the user can navigate through both the program structure and performance data information in order to make efficient optimization decisions during the process of porting sequential applications to parallel computer architectures. ParaWise [15] parallelizes applications, including the automatic insertion of message passing communications and/or OpenMP directives. However, all of those tools are limited to the compiler for the code analysis, no optimization strategies can be reused, and the analysis capability is inaccurate in some cases.

A code transformation technique, [25] describes the porting strategy for translating from COBOL to C/C++ based. However this tool is outdated since COBOL is not used in the HPC field. There are some other tools such as CHiLL [7], POET [30] which provide code transformation for a target system. But the code transformation relies on users to manually write transformation scripts, also it's lack of capability of finding code regions which could apply code transformation. TSF [21] is a pattern matching based code transformation tool only for Fortran code engineering, but it's lack of capability to find how similar of the code regions could be applied for code transformation. Hercules [17] is another code transformation tool that could be used to apply optimizing transformations, but we still need a tool to identify code regions in which these transformations could be applied. The Hercules project from Oak Ridge relies on a transformation recipe and a compiler plug-in infrastructure to apply the transformation processes at compile time. Although early evaluations of Hercules suggest that the pattern matching approach is feasible on current computer resources, the task of defining patterns may become daunting to the programmer, and a tool to assist with the creation of this pattern creation based on similar code is needed.

3 Klonos Framework

Klonos [11] is the tool we designed for assisting software porting. This tool is based on the similarity analysis with the help of the OpenUH [24] compiler. As Fig. 2 shows, the main framework of *Klonos* is comprised of static, dynamic and cost-model metrics collection and porting planning analysis.

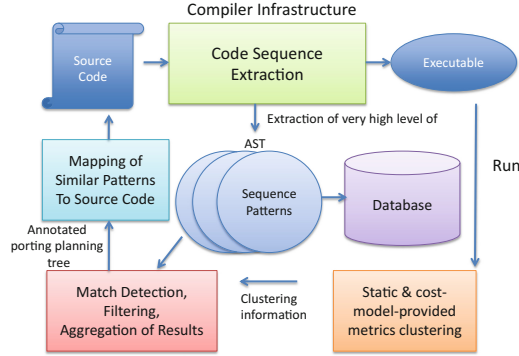


Fig. 2. The KILONOS porting planning system

3.1 Static Metrics

For static metrics collection, we rely on the compiler to collect code syntactic information. Additional functionality to track pattern information in the generated parse tree has been added to OpenUH as part of this work. During the traversal process, each key operator or operand visited by the compiler will be decoded into a unique character, which is defined by a node map that was defined in advance. This maps the hierarchical source code information into a flattened sequence. Next, a sequence alignment is constructed to evaluate the subroutine syntactic similarity for each pair of subroutines. Based on the syntactic similarity score for each pair of subroutines, a family distance tree is built, showing the aggregation of structurally similar subroutines. This aggregation is called a “*syntactic cluster*”. This method is easy to use and quite scalable compared to the graph comparison method. Reference [10] describes these steps in more detail.

3.2 Dynamic Metrics

Syntactic analysis is able to help find similar code quickly, which provides the ability to apply similar optimization strategies to that code. However, similar code structure does not guarantee that similar optimization techniques will apply. For example, the parallelization strategy for a particular loop structure would be totally different if we alternate an array name which might introduce loop-carried dependence. In order to ensure a particular optimization or parallelization strategy can be safely applied for similar codes, code feature metrics such as parallelization, vectorization, and memory access pattern related metrics need to be taken into consideration. In selecting dynamic metrics, we choose metrics that are able to reflect and capture the memory behaviors of an application. In [9], the following hardware counters were collected using AMD Code-Analyst: “DC accesses”, “DC misses”, “DTLB L1M L2M”, “CPU clocks”, “Ret branch” and “Ret inst”. Once those metrics are available, Weka [2] is used to

create “*dynamic clusters*” for subroutines based on these code features, using the K-means algorithm to calculate the Euclidean distance for each pair of subroutines.

3.3 Cost-Model Metrics

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 which consists of millions lines of code. To analyse 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. So 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 gain 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: the processor, cache, and parallel overhead [29]. The similarity of code is measured by analysing 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. The cost model provided metrics used are: estimated number of iterations, suggested parallelization, loop parallelizable attribute, loop vectorizable attribute, loop vectorized number, loop align peeled, work estimate, loop depth. These metrics are key factors used in the cost model for optimization strategy selection, which can accurately capture the internal code optimization characteristics.

4 Experiments

GenIDLEST is a Fortran program that simulates transitional and turbulent flows in complex geometries [26]. 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*.

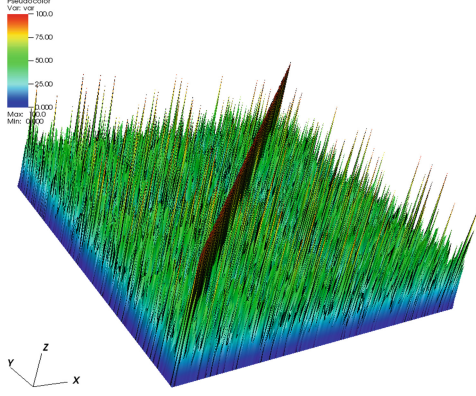


Fig. 3. The subroutine similarities of the GenIDLEST application

4.1 GenIDLEST Similarity Analysis

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 pre-processing steps, next we generate a similarity square matrix by comparing each pair of subroutine sequences until all the subroutines have been consumed. Figure 3 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 4 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 relationships of 254 subroutines, which the total number of subroutines after preprocessing that excludes subroutines with only one function call inside.

Table 2 summarizes the statistics of the similarities of subroutines after pre-processing. GenIDLEST has 1327 subroutine pairs that maintain syntactic similarity of greater than 50 %, which means a majority of subroutines look similar structurally.

4.2 Syntactic Clustering Analysis

Figure 5 shows the relationship of the number of correct porting similar subroutine pairs with setting different number of clusters based on code syntactic similarity. In Fig. 5(a), we can see the number of similar subroutine pairs using similar porting directives decreases gradually as the syntactic based cluster number increases. Figure 5(b) shows the ratio of similar subroutine pairs using

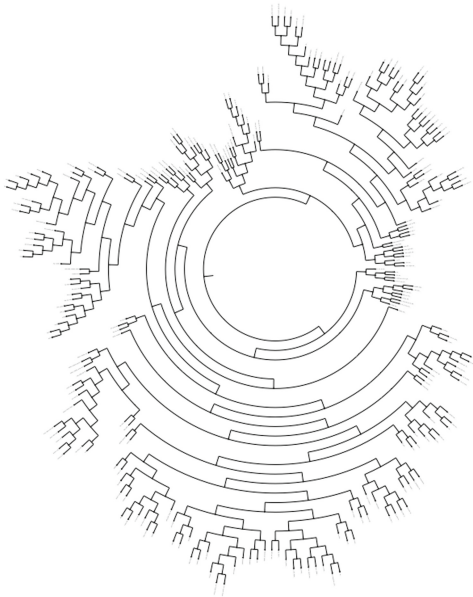


Fig. 4. The overall family distance tree for GenIDLEST

Table 2. GenIDLEST subroutines similarity statistics

Similarity range	# of subroutine pairs
Similarity ≥ 90	47
Similarity ≥ 80	43
Similarity ≥ 70	44
Similarity ≥ 60	208
Similarity ≥ 50	985
Similarity < 50	30804

Table 3. OpenMP directive encoding code map

Directives	Character map
<code>\$!OMP PARALLEL</code>	P
<code>\$!OMP DO</code>	D
<code>\$!OMP PARALLEL DO</code>	PD

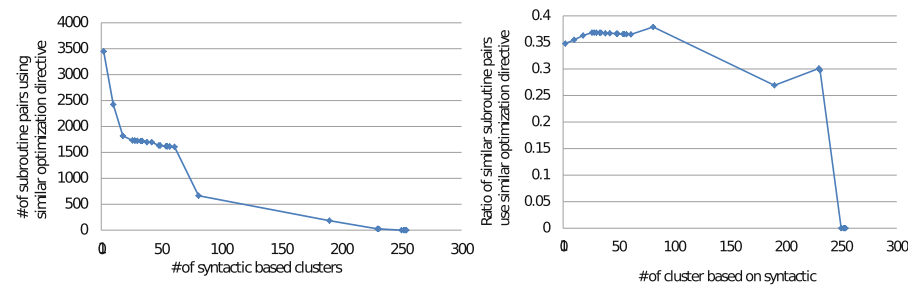


Fig. 5. Syntactic-based cluster for GenIDLEST application

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 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 [9], so in our experiment we use that threshold value and the input depth of the tree for clustering.

4.3 Cost-Model Metrics Clustering Analysis

To better understand the relationship between the cost-model-provided metrics and similar optimization or porting strategy, we only used the cost-model metrics to cluster the subroutines and then check the number of subroutine pairs which use the same optimization directives or strategies. Figure 6(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 changing 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 6(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 clus-

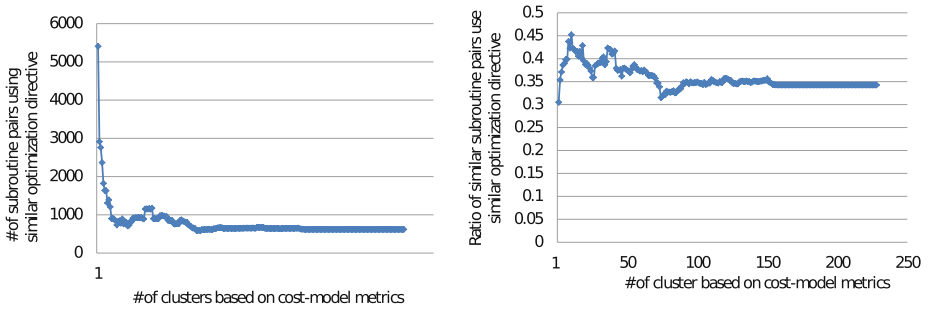


Fig. 6. Cost-model metrics based cluster for GenIDLEST application

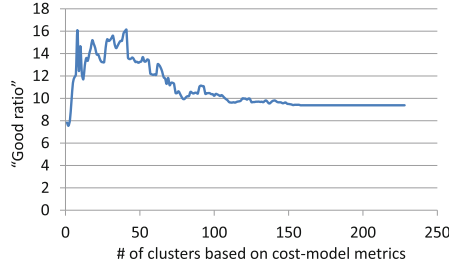


Fig. 7. Cost-model metric based clusters for GenIDLEST

tering 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” 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 highest “Good ratio” to make structurally similar subroutines aggregated as many as possible for similar porting experience reuse.

In Fig. 7, 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 to 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.

4.4 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 incorporating these two metrics we can greatly increase the accuracy of the process of detecting subroutine pairs to be ported in the same way.

In Sect. 4.3, 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

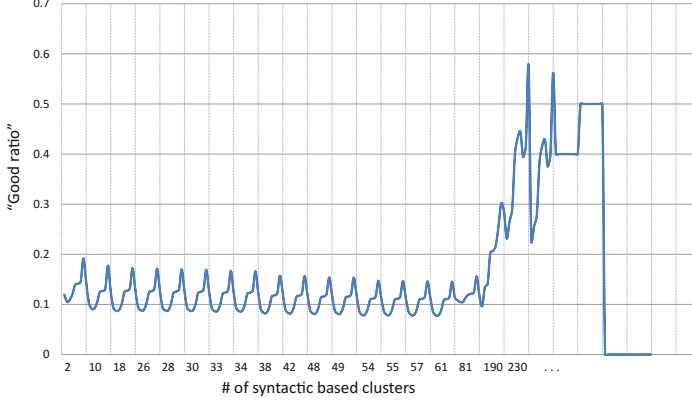


Fig. 8. Combined syntax and cost-model metric clusters for GenIDLEST

subroutines that can be optimized in the same way. In Fig. 8, 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 Sect. 4.3.

4.5 Improved Verification Methodology

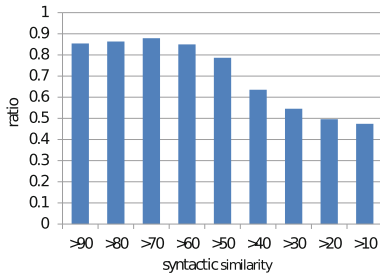
To increase the accuracy of verification, our improved methodology focuses on the syntax of OpenMP directive comparison directly. We add functions into the phase of code sequence extraction (described in Sect. 3.1). If any OpenMP directive is detected in a subroutine, a separate “.opt” file will be generated: 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 3.

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

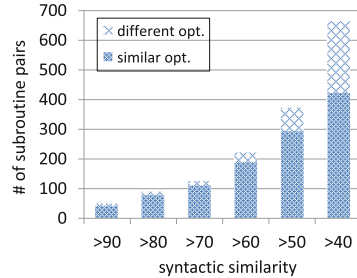
back to the corresponding “.opt” file. For aligned loops, we check the OpenMP encoded directive sequences directly to check if two similar subroutines can have similar optimization directives applied to them for porting purposes.

4.6 Porting Strategy Verification

Based on analysis of clustering using syntactic and cost-model metrics listed in Figs. 5 and 6, we found that either using syntactic distance or cost-metric metrics only as cluster method will results in inaccurate clustering for making porting planning. Our goal is to minimize the number of clusters while in the meantime to make sure accuracy for clustering similar subroutines using similar porting directives. According to Fig. 8 shows that we can have maximum similar pairs ratios for subroutine pairs fall into the same syntactic and cost-model metrics based cluster. So we set up the number of code-model provided metrics cluster (or short for 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, then we are able to divide the tree into 9 clusters for syntactic cluster. By merging the syntactic and code feature clustering, we divide the 254 subroutines into 25 groups. Subroutines within each group fall into the same syntactic and code-model cluster. After combining syntactic and code-most metric based clusters or combined cluster, next comes to step of verifying the correctness of similar directives used for subroutines fall into combined cluster. The ratio of all subroutine pairs using similar optimization reaches 49.51 % in our experiment. Figure 9(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 code-model cluster when setting the cost-metrics based cluster number to 8. As the figure shows, the correctness of using similar directive for parallelizing the code is almost 80 % for subroutine pairs who fall into the same syntactic and code-model cluster with syntactic similarity is greater than 50 %. Figure 9(b) shows the number of pairs using similar porting strategy in detail.



(a) Similar optimization directive verification for GenIDLEST



(b) Similar optimization directive verification for GenIDLEST

Fig. 9. Verification of GenIDLEST porting planning analysis

Higher syntactic similarity will result in using similar directive 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 strategy. Using 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.

5 Conclusions and Future Work

In this paper, we have expanded the notion of code similarity analysis to cost-model-provided metrics for detecting similar porting strategies for similar subroutine pairs, thus avoiding the burden of running applications to gather profiling information.

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, 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. Additionally, we will implement a GUI to visualize the process of generating porting plans.

References

1. OpenMP ARB. Openmp arb. <http://openmp.org/wp/about-openmp/>
2. Machine Learning Group at University of Waikato. Weka 3: Data mining software in java. <http://www.cs.waikato.ac.nz/ml/weka/>
3. Jost, G., Chapman, B.M., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, Cambridge (2007)
4. Buttlar, D., Nichols, B., Farrell, J.P.: Pthreads Programming. O'Reilly & Associates Inc., Sebastopol (1996)
5. NASA Ames Research Center. Capo (computer-aided parallelizer and optimizer). <http://people.nas.nasa.gov/~hjin/CAPO/index.html>
6. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming, vol. 10. MIT Press, Cambridge (2007)
7. Chen, C., Chame, J., Hall, M.: CHiLL: a framework for composing high-level loop transformations. Technical report, Technical Report 08-897, USC Computer Science Technical Report (2008)
8. Davison, J., Mancl, D., Opdyke, W.: Understanding and addressing the essential costs of evolving systems. Bell Labs Tech. J. **5**, 44-54 (2000)

9. Ding, W., Hernandez, O., Chapman, B.: A similarity-based analysis tool for porting OpenMP applications. In: Keller, R., Kramer, D., Weiss, J.-P. (eds.) *Facing the Multicore-Challenge III*. LNCS, vol. 7686, pp. 13–24. Springer, Heidelberg (2013)
10. Ding, W., Hsu, C.-H., Hernandez, O., Chapman, B., Graham, R.: Klonos: similarity-based planning tool support for porting scientific applications. *Concurrency Comput. Pract. Experience* **25**, 1072–1088 (2013)
11. Ding, W., Hsu, C.-H., Hernandez, O., Graham, R., Chapman, B.M.: 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)
12. CAPS Enterprise. HMPP: A Hybrid Multicore Parallel Programming Platform. http://www.caps-entreprise.com/en/documentation/caps_hmpp_product_brief.pdf
13. The Portland Group. PGI accelerator compilers (2010). <http://www.pgroup.com/resources/accel.htm>
14. Hernandez, O., Ding, W., Chapman, B., Kartsaklis, C., Sankaran, R., Graham, R.: Experiences with high-level programming directives for porting applications to GPUs. In: Keller, R., Kramer, D., Weiss, J.-P. (eds.) *Facing the Multicore - Challenge II*. LNCS, vol. 7174, pp. 96–107. Springer, Heidelberg (2012)
15. Johnson, S., Evans, E., Jin, H., Ierotheou, C.: The ParaWise expert assistant - widening accessibility to efficient and scalable tool generated OpenMP code. In: Chapman, B.M. (ed.) *WOMPAT 2004*. LNCS, vol. 3349, pp. 67–82. Springer, Heidelberg (2005)
16. Jost, G., Jin, H., Labarta, J., Gimenez, J.: Interfacing computer aided parallelization and performance analysis. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J., Zomaya, A.Y. (eds.) *ICCS 2003, Part IV*. LNCS, vol. 2660, pp. 181–190. Springer, Heidelberg (2003)
17. Kartsaklis, C., Hernandez, O., Hsu, C.H., Ilsche, T., Joubert, W., Graham, R.L.: Hercules: a pattern driven code transformation system. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pp. 574–583. IEEE (2012)
18. Levesque, J., Sankaran, R., et al.: 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*, p. 15. IEEE Computer Society Press (2012)
19. Top500 List. Treemap - november 2012 (accelerator/co-processor). <http://www.top500.org/statistics/treemaps/>
20. Mancl, D.: Refactoring for software migration. *IEEE Commun. Mag.* **39**(10), 88–93 (2001)
21. Mével, Y.: Tsf: an environment for program transformations
22. Munshi, A.: *The OpenCL Specification*, October 2009
23. NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 3.0*, March 2010. <http://developer.nvidia.com/cuda>
24. The OpenUH compiler project (2005). <http://www.cs.uh.edu/~openuh>
25. Sampaio do Prado Leite, J.C., Sant’Anna, M., Francisco do Prado, A.: Porting cobol programs using a transformational approach. *J. Softw. Maintenance: Res. Pract.* **9**(1), 3–31 (1997)
26. Tafti, D.: Genidlest a parallel high performance computational infrastructure for simulating complex turbulent flow and heat transfer. *APS Division of Fluid Dynamics Meeting Abstracts*, vol. 1 (2002)

27. Vetter, S., Aoyama, Y., Nakano, J.: RS/6000 SP: practical MPI programming, vol. SG24-5380-00 of 0738413658. vervante, August 1999
28. The Wikipedia. Software porting. <http://en.wikipedia.org/wiki/Porting>
29. Wolf, M.E., Maydan, D.E., Chen, D.-K.: Combining loop transformations considering caches and scheduling. In: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 274–286. IEEE Computer Society, Washington, DC (1996)
30. Yi, Q., Seymour, K., You, H., Vuduc, R., Quinlan, D.J.: POET: parameterized optimizations for empirical tuning. In: Workshop on Performance Optimization for High-Level Languages and Libraries, March 2007

Languages and Compilers for Parallel Computing
26th International Workshop, LCPC 2013, San Jose, CA,
USA, September 25--27, 2013. Revised Selected Papers
Caşcaval, C.; Montesinos, P. (Eds.)
2014, XXIV, 357 p. 160 illus., Softcover
ISBN: 978-3-319-09966-8