International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Impact Factor (2012): 3.358

Literature Survey on a New Advanced Refactoring Based Approach for Parallelism Using Heterogeneous Parallel Architectures

Shanthi Makka[1], Bharat Bushan Sagar[2]

[1]Assistant Professor, JRE Group of Institutes and Ph.D. Scholar at BITs, Mesra (Noida Campus), Greater Noida, India-201308

[2]Assistant Professor at BITs, Mesra (Noida Campus), Sector-1, near Metro Station ,Noida, India

Abstract: ***Refactoring is the process of changing the structure of a program without changing its behavior. Refactoring has so far only really been deployed effectively for sequential programs. However, with the increased availability of multi core systems, refactoring can play an important role in helping both expert and non-expert parallel programmers structure and implement their parallel programs. This paper describes benefits or advantages of a refactoring approach for parallel programs using heterogeneous parallel architectures such as GPUs and CPUs. A refactoring based methodology gives many advantages over unaided parallel programming: it helps identify general patterns of parallelism; it guides the programmers through the process of refining a parallel program, whether new or existing; it enforces separation of concerns between application programmers and system programmers; and it reduces time to deployment. All of these advantages help programmers understand how to write parallel programs.***

Keywords: refactoring, parallelism, CPU, GPU, refactoring tool

1. Introduction

Despite Moore's "law" [24], uniprocessor clock speeds have now stalled. Rather than using single processors running at ever higher clock speeds, and drawing ever increasing amounts of power, even consumer laptops, tablets and desktops now have dual, quad or hexa core processors. Haswell, Intel's next multi core architecture, will have eight cores by default. Future hardware is likely to have even more cores, with many cores and perhaps even mega core systems becoming main stream. This means that programmers need to start thinking parallel, moving away from traditional programming models where parallelism is a bolted-on afterthought towards new models where parallelism is an intrinsic part of the software development process. One means of developing parallel programs that is attracting increasing interest is to employ parallel patterns, that is, sets of basic, pre-defined building blocks that each model and embed a frequently recurring pattern of parallel computation.

In the multi core era [14], a major programming task will be to make programs more parallel. This is tedious because it requires changing many lines of code, and it is error-prone and non-trivial because programmers need to ensure non-interference of parallel operations. Fortunately, refactoring tools can help reduce the analysis and transformation burden. This paper discuss how refactoring tools can improve programmer productivity, program performance, and program portability and also present the current incarnation of this vision: a toolset that supports several refactoring for (i) making programs thread-safe, (ii) threading sequential programs for throughput, and (iii) improving scalability of parallel programs.

The strong need for increased computational performance in science and engineering has led to the use of heterogeneous computing, with GPUs and other accelerators acting as co-processors for arithmetic intensive data parallel workloads [20–23]. The trend towards heterogeneous computing and highly parallel architectures has created a strong need for software development infrastructure in the form of parallel programming languages and subroutine libraries supporting heterogeneous computing on hardware platforms produced by multiple vendors. Many existing science and engineering applications have been adapted to make effective use of multi-core CPUs and massively parallel GPUs.

2. Refactoring

The term refactoring was originally introduced by William Opdyke in his PhD dissertation [2]. Refactoring is basically the object oriented variant of restructuring: "the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" [1]. The key idea here is to redistribute classes, variables and methods across the class hierarchy in order to facilitate future adaptations and extensions. In the context of software evolution, restructuring and refactoring are used to improve the quality of the soft- ware (e.g., extensibility, modularity, reusability, complexity, maintainability, efficiency). Refactoring and restructuring are also used in the context of reengineering, which is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. In this context, re-structuring is needed to convert legacy code or deteriorated code into a more modular or structured form [1], or even to migrate code to a different programming language or even language paradigm.

Licensed Under Creative Commons Attribution CC BY

The key defining aspect of refactoring is its focus on purely structural changes rather than on changes in program functionality. Some advantages of refactoring are as follows:

(i) Refactoring aims to improve software design. Without refactoring, a program design will naturally decay: as code is changed, it progressively loses its structure, especially when this is done without fully understanding the original design. Regular refactoring helps tidy the code and retain its structure.
(ii) Refactoring makes software easier to understand. Refactoring helps improve readability, and so makes code easier to change. A small amount of time spent refactoring means that the program better communicates its purpose.
(iii) Refactoring helps the programmer to program more rapidly. Refactoring encourages good program design, which allows a development team to better understand their code. A good design is essential to maintaining rapid, but correct, software development.

Refactoring activities: - The refactoring process consists of a number of distinct activities:

1) Identify where the software should be refactored.
2) Determine which refactoring(s) should be applied to the identified places.
3) Guarantee that the applied refactoring preserves behavior.
4) Apply the refactoring.
5) Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).
6) Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests and so on).

In the past, refactoring has been traditionally associated with improving the structure of the code, thus making the code more *readable* and more *reusable*, even across different platforms.

3. Refactoring Approach to Parallelism

For decades, programmers relied on Moore's Law [3] to improve the performance of their applications. With the advent of multicores, programmers are forced to exploit parallelism [2] if they want to improve the performance of their applications, or when they want to enable new applications and services that were not possible earlier. One approach for parallelization of a program is to rewrite it from a scratch, however the most common way to parallelize a program consider one piece at a time and each small step can be considered as a behavior preserving transformation, i.e., a refactoring. Every programmer prefers this approach because it is safer: they prefer to maintain a working, deployable version of the program. Also, the incremental approach is more economical than rewriting. However, the refactoring approach is still *tedious* because it requires changing many lines of code is *error-prone* and is *non-trivial* because programmers need to ensure noninterference of parallel operations.

To reduce the programmer's burden when converting sequential to parallel programs, several tools have been proposed. They come in two distinct flavors: (i) fully automatic tools or non interactive tools (e.g., automatic parallelizing compilers [4]–[7]) and (ii) interactive tools (e.g., refactoring tools [8]–[15]). The fundamental difference between these tools is the role of the programmer. A non-interactive tool creates a parallel program automatically, without any help from the programmer. When this works it gives great results. Unfortunately, without programmer's domain knowledge, the compiler has limited applicability. To date, the only compiler successes have been in programs involving dense matrix operations and stencil computations. Even though compilers have improved a lot, programmers still parallelize by hand most of the code. Interactive tools take a completely different approach: sometimes, less automation is better! They let the programmer be in the driver's seat. The programmer is the expert on the problem domain, and so understands the domain concepts amenable to parallelism. The programmer also understands the current sequential implementation: the program invariants that must be preserved during parallelization, along with the data and control flow relationships between parts of the program, and the algorithms and data structures used in the current implementation. Thus, the interactive approach combines the strengths of the programmer (domain knowledge, seeing the big picture) and the computers (fast search, remember, and compute). The programmer does the creative part: selects code and targets it with a transformation. The tool does the tedious job: checks the safety (this involves searching in many files, by traversing through many functions and through aliased variables), and modifies the program. When the tool cannot apply a transformation, it provides information integrated within the visual interface of an Integrated Development Environment (IDE), thus allowing a programmer to pinpoint the problematic code.

A refactoring toolset for parallelism has several points of interaction with the programmer, shown in below algorithm.

1. Start
2. Select code and a target refactoring
3. Apply tool, which can analyze the safety of the transformation.
   - If it is safe then apply the changes what you want to make.
   - If it is not safe then tool raises some warnings. The programmer can decide to cancel the refactoring, fix the code, then re-run the refactoring, or he can decide to proceed against warnings.
4. Stop

We found that parallelizing transformations are not random, but they fell into four categories.

1. Transformations that improve the *latency* (i.e., an application feels more responsive).
2. Transformations that improve the *throughput* (i.e., more computational tasks executed per unit of time).

Licensed Under Creative Commons Attribution CC BY

3. Transformations that improve the *scalability* (i.e., the performance scales up when adding more cores), and
4. Transformations that improve *thread safety* (i.e., application behaves according to its specification even when executed under multiple threads).

4. Refactoring tools for parallelism

When parallelizing a sequential program, a programmer needs to

(i) Make the code thread-safe by protecting accesses to mutable shared data,
(ii) Make the code run on multiple threads of execution, and
(iii) Make the performance scalable when adding more cores.

Several authors advocate to first make the code right (i.e., thread-safe), then make it fast (i.e., multi-threaded), then make it scalable. Our growing toolset currently automates six refactorings, that fall into three categories. Refactorings for thread-safety make a program thread-safe but do not introduce multithreading yet. Refactorings for throughput add multi-threading. Refactorings for scalability replace existing data structures with highly scalable ones.

i. Refactorings for Thread-Safety

Before introducing multi-threading, the programmer needs to prepare or enable the program for parallel execution. This involves finding the *mutable* data that will be *shared* across parallel executions. The programmer can decide to (i) synchronize accesses to such data, or (ii) remove either its mutability or sharedness. Below I present the refactoring for converting a mutable into an immutable class.

How to make Class Immutable?
One way to make a whole class thread-safe is to make it immutable. An immutable class is thread-safe by default, because its state cannot be mutated once an object is properly constructed. Thus, an immutable class can be shared among several threads, with no need for synchronization. Our refactoring enables the programmer to convert a mutable class into an immutable class. To do so, the tool makes the class and all its fields final, so that they cannot be assigned outside constructors and field initializers. The tool finds all mutator methods in the class, i.e., methods that directly or indirectly mutate the internal state (as given by its fields). The tool converts these mutator methods into factory methods that return a new object whose state is the old state plus the mutation.

Next, the tool finds the objects that are entering from outside (e.g., as method parameters) and become part of the state, or objects that are part of the state and are escaping (e.g., through return statements). It clones these objects, so that the class state cannot be mutated by a client class who holds a reference to these state objects. Lastly, the tool updates the client code to use the class in an immutable fashion. For example, when the client invokes a factory method, the tool reassigns the reference to the immutable class to the object returned by the factory method. Our comparison with open-source classes that were manually refactored for immutability shows that the tool is much safer: it finds subtle mutations and entering/escaping objects that programmers overlooked. However, not all classes can be made immutable. For example, if a mutator method already returns an object, the tool cannot convert it into a factory method. Also, due to the extra overhead of copying state, using this refactoring is advisable only when mutations are not frequent.

ii. Refactorings for Throughput

Once a program is threadsafe, multi-threading can be used to improve its performance. The programmer could manage himself a raw thread (e.g., create, spawn, wait for results), or he could use a programmer-friendlier construct, a *lightweight task*, managed automatically by a framework. Our toolset supports two such refactorings. One refactoring converts a sequential divide-and-conquer algorithm into an algorithm which solves the recursive subproblems in parallel. Another refactoring parallelizes loops over arrays.

How to Parallelize a Loop?
This refactoring parallelizes loop iterations over an array via ParallelArray [14], a parallel library upcoming in Java. ParallelArray is an array data structure that supports parallel operations over the array elements. For example, one can apply a procedure to each element, or can reduce all elements to a new element in parallel. The library balances the load among the cores it finds at runtime. The refactoring changes the data type of the array, and it replaces loops over the array elements with the equivalent parallel operations from ParallelArray. At the heart of the tool lies a data-flow analysis that determines objects that are shared among loop iterations, and detects writes to the shared objects. The analysis works with both programs in source code and in byte code. When the analysis finds writes to shared objects, it presents the user a stack of code statements that resulted in the objects being shared. These statements are hyper-linked to the original source code, thus helping the developer to find the problematic code.

Although we were able to refactor several real programs and the analysis was fast and effective, not all loops can be refactored. For example, a loop must (i) iterate over all the array elements, (ii) not contain blocking I/O calls, and (iii)not contain writes to shared objects.

iii. Refactorings for Scalability

One must not sacrifice thread-safety and correctness in the name of performance. However, a naive synchronization scheme can lead to serializing an application, thus drastically reducing its scalability. This usually happens when working with low-level synchronization constructs like locks. Locks are the goto statements of parallel programming: they are tedious to work with, and error prone. Too many locks slow down or deadlock a program, while too few lead to data races.
When possible, a better alternative is to use a highly scalable data-structure provided by parallel libraries. However, this refactoring is not always applicable, for example when an

Licensed Under Creative Commons Attribution CC BY

application needs to lock the entire map for exclusive access (e.g., for a whole traversal). Building this refactoring toolset taught us several lessons:

i. Programmers often use parallel libraries, thus refactoring tools need to support such libraries.
ii. To keep the programmer engaged, refactoring tools need to finish in less than thirty seconds. Thus, they must use efficient, on-demand program analyses.
iii. Program analysis libraries and IDEs with excellent AST rewriting capabilities are essential for building refactoring tools.
iv. Once a program is parallel, it must remain maintainable, i.e., readable and portable.
v. Refactoring tools must interact with other tools in the parallel toolbox.

5. Heterogeneous Parallel Architectures

Key issues include dealing with advanced *heterogeneous* parallel architectures, involving combinations of GPUs and CPUs; providing good hygienic abstractions that cleanly separate components written in a variety of programming languages; identifying new high level *patterns* of parallelism; developing new rule based mechanisms for rewriting (refactoring) source-level programs based on those patterns etc. Why heterogeneous parallel architectures such GPUs and CPUs has been chosen?

GPU:-

I. GPUs were designed in a highly parallel structure [16] that allows large blocks of data to be processed at one time similar computations are being made on data at the same time (rather than in order). If you assigned the task of rendering a 3D environment to a CPU, it would slow to a crawl and handles requests more linearly, because GPUs are better at performing repetitive tasks on large blocks of data than CPUs, you start see the benefit of enlisting a GPU in a server environment.
II. The GPU has emerged as a computational accelerator [17] that dramatically reduces the time to discovery in High End Computing (HEC). However, while today's state of the art GPU can easily reduce the execution time of a parallel code by many orders of magnitude, it arguably comes at the expense of significant power and energy consumption.

Even though GPU has more benefits but why do we need to use CPU, always it may not be the requirement to make all segments of program needed to be parallelized, in those situations better to use CPU because using of GPU is very expensive as compare to CPU.

5.1 GPUs as Storage System Accelerators:

Massively multicore processors [18], such as graphics processing units (GPUs), provide, at a comparable price, a one order of magnitude higher peak performance than traditional CPUs. This drop in the cost of computation, as any order of magnitude drop in the cost per unit of performance for a class of system components, triggers the opportunity to redesign systems and to explore new ways to engineer them to recalibrate the cost-to-performance relation.

5.2 Multi-core CPUs

Modern CPUs [19] are typically composed of a small number of high-frequency processor cores with advanced features such as out-of-order execution and branch prediction. CPUs are generalists that perform well for a wide variety of applications including latency-sensitive sequential workloads, and coarse-grained task-parallel or data-parallel workloads. Since they are typically used for latency sensitive workloads with minimal parallelism, CPUs make extensive use of large caches to hide main memory latency. Many CPUs also incorporate small scale use of single-instruction multiple-data (SIMD) arithmetic units to boost the performance of dense arithmetic and multimedia workloads. These SIMD units are not directly exposed by conventional programming languages like C and Fortran, so their use requires calling vectorized subroutine libraries or proprietary vector intrinsic functions, or trial-and-error source level restructuring and auto vectorizing compilers.

5.3 Graphics Processing Units

Contemporary GPUs are composed of hundreds of processing units running at a low to moderate frequency, designed for throughput-oriented latency insensitive workloads. In order to hide global memory latency, GPUs contain small or moderate sized on-chip caches, and they make extensive use of hardware multithreading, executing tens of thousands of threads concurrently across the pool of processing units. The GPU processing units are typically organized in SIMD clusters controlled by a single instruction decoder, with shared access to fast on-chip caches and shared memories. The SIMD clusters execute machine instructions in lock-step, and branch divergence is handled by executing both paths of the branch and masking off results from inactive processing units as necessary. The use of SIMD architecture and in-order execution of instructions allows GPUs to contain a larger number of arithmetic units in the same area as compared to traditional CPUs.

Although GPUs are powerful computing devices in their own right, they must currently be managed by the host CPUs. GPUs are typically attached to the host by a PCI-Express bus, and in most cases have their own independent on-board memory system. In order to exchange input and output data with the GPU, the host CPU schedules DMA transfers between the host and GPU memory systems.

Licensed Under Creative Commons Attribution CC BY

6. Dynamic Mapping

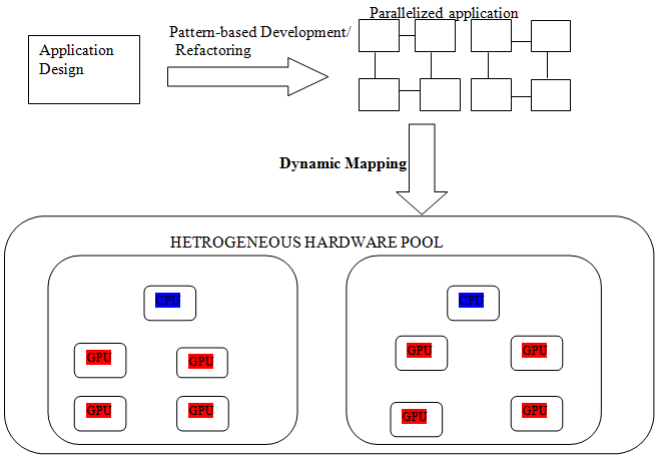


Figure 1: vision of refactoring for parallelism using heterogeneous architectures

Aim is to produce a new structured design and implementation process for heterogeneous parallel architectures, where developers exploit a variety of parallel patterns to develop component-based applications that can be mapped to the available hardware resources, and which may then be dynamically re-mapped to meet application needs and hardware availability (Figure 1). We will exploit new developments in the implementation of parallel patterns that will allow us to express a variety of parallel algorithms as compositions of lightweight software components forming a collection of virtual parallel tasks. Components from multiple applications will be instantiated and dynamically allocated to the available hardware resources through a simple and efficient software virtualization layer. In this way, we will promote adaptivity, not only at an application level, but also at a system level. Finally, virtualization abstractions will be provided across the hardware boundaries, allowing components to be dynamically re-mapped to either CPU or GPU resources on the basis of suitability and availability.

7. Conclusion

This Paper has described advantages of parallel programs over sequential programs. Programmers are forced to exploit parallelism if they want to improve the performance of their applications, or when they want to enable new applications and services that were not possible earlier. One approach for parallelization of a program is to rewrite it from a scratch, however the most common way to parallelize a program consider one piece at a time and each small step can be considered as a behavior preserving transformation, i.e., a refactoring. It also described benefits of refactoring towards parallelism and how the refactoring tools can be used to achieve parallelism. The strong need for increased computational performance in science and engineering has led to the use of heterogeneous computing, This paper also describes the how a refactoring approach can be used for sequential programs and parallel programs using homogeneous and heterogeneous parallel architectures such as GPUs and CPUs.

References

[1] Mens, Tom and Tourwe, Tom (2004) *A Survey of Software Refactoring*, IEEE Transactions on Software Engineering, February 2004 (vol. 30 no. 2), pp. 126–139.
[2] W. F. Opdyke, Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks Ph.D. thesis,University of Illinois at Urbana Champaign, 1992.
[3] G. E. Moore. Readings in Computer Architecture. Chapter Cramming more components On to integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
[4] Netflix Prize Forum / Grand Prize Award. http://www.netflixprize.com/ community/viewtopic.php?id=1537, Sept. 2009.
[5] M. Aldinucci and M. Danelutto. Stream Parallel Skeleton Optimization. In Proc. Of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems, pages 955–962, Cambridge, Massachusetts, USA, Nov. 1999. IASTED, ACTA press.
[6] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating Code on Multi-cores with FastFlow. In Euro-Par, pages 170–181, 2011.
[7] J. Backus. Can Programming be Liberated from the von Neumann Style? Communications of the ACM, 21(8):613–641, 1978.
[8] S. Benkner, S. Pllana, J. L. Tr¨aff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. W. Kessler, D. Moloney, and V. Osipov. PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. IEEE Micro, 31(5):28–41, 2011.
[9] R. S. Bird. Lectures on Constructive Functional Programming. In M. Broy, editor, Constructive Methods in Computer Science, pages 151–218. Springer-Verlag, 1988.
[10] NATO ASI Series F Volume 55. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.
[11] C. Brown, H. Li, and S. Thompson. An Expression Processor: A Case Study in Refactoring Haskell Programs. In R. Page, editor, Eleventh Symposium on Trends in Functional Programming, page 15, May 2010.
[12] C. Brown, H. Loidl, and K. Hammond. Paraforming: Forming Haskell Programs using Novel Refactoring Techniques. In Twelth Symposium on Trends in Functional Programming, Madrid, Spain, May 2011.
[13] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. J. ACM, 24(1):44–67, 1977.
[14] F. Cesarini and S. Thompson. ERLANG Programming. O'ReillyMedia, Inc., 1st edition, 2009.
[15] D. Dig. A Refactoring Approach to Parallelism. IEEE Softw., 28:17–22, 2011.
[16] R. Gemulla, P. J. Haas, Y. Sismanis, C. Teflioudi, and F. Makari. Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. In NIPS2011 Workshop on Big Learning. Sierra Nevada, Spain, Dec. 2011.

Licensed Under Creative Commons Attribution CC BY

[17] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M Sarrafzadeh. Energy-Aware High Performance Computing with Graphic Processing Units. In Workshop on Power Aware Computing and System, December 2008.
[18] Huang, Song, Shucai Xiao, and Wu-chun Feng. "On the energy efficiency of graphics processing units for scientific computing." Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, 2009.
[19] Al-Kiswany, Samer. "Embracing diversity: optimizing distributed storage systems for diverse deployment environments." (2013).
[20] Stone, John E., David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." Computing in science & engineering 12.3 (2010): 66.
[21] Shi Guochun, Kindratenko Volodymyr, Pratas Frederico, Trancoso Pedro, Gschwind Michael. Application acceleration with the Cell broadband engine. Computing in Science and Engineering. 2010;12(1):76–81.
[22] Cohen Jonathan, Garland Michael. Solving computational problems with GPU computing. Computing in Science and Engineering. 2009;11(5):58–63.
[23] Bayoumi Amr, Chu Michael, Hanafy Yasser, Harrell Patricia, Refai-Ahmed Gamal. Scientific and engineering computing using ATI stream technology. Computing in Science and Engineering. 2009;11(6):92–97.
[24] Barker Kevin J, Davis Kei, Hoisie Adolfy, Kerbyson Darren J, Lang Mike, Pakin Scott, Sancho Jose C. SC' 08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Piscataway, NJ, USA: IEEE Press; 2008. Entering the petaflop era: the architecture and performance of Roadrunner; pp. 1–11.
[25] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Sch¨oner, and Tino Breddin, Paraphrasing: Generating Parallel Programs using Refactoring, Springer Berlin Heidelberg pp. 237-256,2013.

Licensed Under Creative Commons Attribution CC BY