Bringing the HPC Programmer's IDE into the 21st Century through Refactoring

Fredrik Berg Kjolstad Danny Dig Marc Snir
University of Illinois at Urbana-Champaign
{kjolsta1,dig,snir}@illinois.edu

Abstract

Programming tools for High Performance Computing are lagging behind the tools that have improved the productivity of desktop programmers. The increasing complexity of HPC codes, the growing number of cores that they must utilize, their long life-span, and the plethora of desirable source code optimizations and hardware platforms make HPC codes hard to maintain. Refactoring tools can enable HPC programmers to explore the space of performance optimizations and parallel constructs safely and efficiently.

This position paper presents our view on how HPC programming tools should evolve, a growing catalog of refactorings for HPC programmers, and reports on our initial effort to automate some of these refactorings.

Keywords Parallelism, Refactoring, Performance, HPC

1. Introduction

In the last ten years great strides have been made in increasing the productivity of the desktop application programmer. Powerful IDEs such as Eclipse, Microsoft Visual Studio, NetBeans, and IntelliJ IDEA have become the norm and programmers expect automated support for code refactoring.

Before refactoring tools appeared, programmers often over-designed, because it was expensive and error-prone to change the design of large systems once they were implemented. Refactoring tools allow programmers to *continuously* explore the design space of multi-million line codebases, without the fear of introducing unintended behavioral changes. Modern IDEs incorporate refactoring in their top menu, and often compete on the basis of refactoring support.

So far, automated refactoring support in mainstream IDEs has mainly been geared towards improving code design and readability. However, refactoring is the process of changing source code without changing its semantics, in order to improve some non-functional aspect. Aside from readability and maintainability, there are many such aspects that can benefit from refactoring technology.

Recently we have used refactoring to retrofit sequential Java applications to use parallelism for performance [3, 4, 9]. The goal is to decrease the running time of an application, or to increase the amount of data that can be processed in the same amount of time, *without* changing the functionality. It is therefore a good fit for refactoring technology.

The HPC (High Performance Computing) programmer is, of course, concerned with getting the most performance he can from the available hardware. However, there are other concerns that are equally, and often more, important and that conflict with this goal. Examples include numerical stability, scalability, maintainability, and portability [1]. Codes must be correct and they typically have a life-span of decades. The latter implies that they must be maintained for a long time and that they will run on machines that are wildly different, and that may have orders of magnitudes more processor cores than the machine they were originally designed for.

The HPC programmer must carefully balance all of these concerns and has very little effective development tool support to do so. While the domain of application programming is ripe with powerful IDEs, HPC programming typically involves a programmer with VIM, or an X-forwarded text browser, and a command line shell. While HPC programmers have created many great codes using these trusted tools, we believe it is time to bring their tool support into the 21st Century through a powerful IDE with support for the kinds of refactorings that *they* need.

One promising effort at providing an IDE for parallel and HPC programmers is the Eclipse PTP project [14]. PTP is a set of plugins that extend Eclipse with tools to develop parallel software for remote machines and clusters. It simulates a local development environment even though the code may be run on, and even located at, a remote cluster. Furthermore, it provides an integrated environment in which to interact with the plethora of debuggers, profilers, and job launchers that exist for these systems.

PTP is built on top of CDT, which is a set of Eclipse plugins for C and C++ development. It therefore provides some refactorings such as Rename and Extract Method. However, PTP does not provide the other kinds of refactorings we believe HPC programmers need in order to achieve higher productivity in the face of increasing complexity and a plethora of parallel programming models.

2. Towards A Refactoring Catalog for the 21st Century HPC Programmer

In this section we begin a catalog of refactorings for the HPC programmer. The catalogue is incomplete and we expect it to grow and mature over time. Some of the refactorings in the catalog introduce parallelism, others prepare the code for parallelism, while others still improve the performance of already parallel code.

As the number of cores per processor continues to increase, more cores are contending for the same shared memory resources. It is therefore increasingly important to manage memory resources so that they do not become bottlenecks that limit an application's performance and prevent it from scaling. Consider for example an application where threads frequently update memory on the same cache line. The threads have to get ownership of this cache line before writing to it, thus forcing them to write to it one at a time.

Furthermore, a parallel application consists of several units of execution that *sequentially* manipulate a chunk of data or execute a flow of control. As such, a parallel application can be viewed as an array of cooperating sequential components. If the sequential components suffer from poor performance then the parallel application will too. Also, due to the effect demonstrated by Amdahl's law, the inherently sequential parts of an application will dominate when the number of parallel execution units becomes sufficiently large. The HPC programmer must therefore not only extract enough parallelism to keep available cores busy, but must also ensure that the sequential parts perform adequately.

We therefore also include refactorings that are intended to help the HPC programmer speed up the sequential components of the parallel application. Since structured programming is still dominant in the HPC community [1], the sequential refactorings we propose are for the C language.

2.1 Parallel MPI Refactorings

Make Communication Asynchronous This refactoring replaces a synchronous MPI_Send or MPI_Recv with an asynchronous MPI_Isend or MPI_Irecv and an MPI_Wait. It then attempts to move the MPI_Wait as far apart from the send as possible with the constraint that it is not moved past code that writes to memory areas that are being sent, or reads from memory areas that are being received. This allows useful computation to be performed while messages are sent, which decreases running time and makes the application less sensitive to communication jitter.

Split Computation Into Communication Dependent and Independent Parts This refactoring finds the subset of the iterations of a loop that compute results that are subsequently sent to another process, and peals these off from the rest of the loop. This enables the programmer to overlap communication with computation by asynchronously transferring communication-dependent results while the rest are computed. Many iterative algorithms follow the pattern of repeatedly updating chunks of a data structure on different nodes and then having each node exchange the borders of its chunk with those of its neighbors. Iterative structured grid computations are examples of this, where an n-dimensional grid is split into chunks and divided amongst the compute nodes. Since only the borders of the chunks are exchanged between each iteration it is useful to compute this part first so that the exchange can be overlapped with the computation of the interior [10].

Remove Superfluous Barriers This refactoring analyzes an application and attempts to remove superfluous barriers. That is, if it can detect that other MPI functions provide sufficient synchronization then it can remove redundant barriers.

Replace Barrier with Local Synchronization This refactoring attempts to replace an MPI Barrier with synchronous calls to the `MPI_Ssend` and `MPI_Recv` functions, so that only some processes are synchronized in cases where this is safe. This can be useful to relax synchronization in MPI programs where a barrier is used to force all processes to synchronize, when in reality only some of the processes need to wait for each other.

Gather Data This refactoring helps the programmer create communication code to retrieve the whole or parts of a shared data structure maintained on a master process. This can be useful when porting a sequential or shared-memory parallel program that uses a shared data structure to MPI. As a first step it is reasonable to maintain this data structure on a master process, before considering whether to distribute it between the processes.

Create MPI Datatype from Struct This refactoring creates an MPI Datatype for a C struct or an array of structs. Previous work on marshaling data structures as MPI Types include Tansey and Tilevich's MPI Serializer [13].

Replace Flattening Loop with MPI Datatype This refactoring replaces a flattening loop with the construction of an MPI Datatype that describes the mapping between the local data structures and a message. A flattening loop is a loop that copies the parts of the local data structures that should be sent to another process into a buffer. This buffer is then sent using a simple MPI datatype. Flattening loops are very common in MPI applications and replacing them with the use of MPI datatypes yields code that is simpler to understand, and faster if a copy pass can be avoided [6].

2.2 Sequential C Refactorings

Restrict Pointer The ISO C99 standard added support for a new type qualifier, called `restrict`, that "[...] allows programs to be written so that translators can produce significantly faster executables" [7]. Intuitively, `restrict`, when applied to a pointer p, specifies that the object pointed to by p will not be accessed through another pointer in a block where it is updated through p. The Restrict Pointer refactoring performs a whole-program analysis of the source code and restricts a given pointer if it is safe. If there are dependencies preventing the pointer from being restricted the refactoring provides information about the dependency to the programmer so that he can break it.

Split Struct into Hot and Cold Fields It is common for some fields in a struct to be accessed more often than others. An example is a linked data structure where the key and next fields are frequently accessed when searching for a key, while the value fields are only accessed when the key is found [5]. Furthermore, some data structures such as heaps and k-d trees are often stored in an array. This refactoring splits a struct in two, where one part contains the fields that are accessed often and the other contains the fields that are accessed less often. The structs are then stored in different arrays so that the hot array can be traversed quickly in cache.

Organize Block as Load/Compute/Store This refactoring organizes a code block so that all input data are read into local variables at the beginning, and the results stored back at the end. This can improve the compiler's ability to analyze dependencies, optimize code and schedule code and can be useful in kernels that need heavy optimization.

Split Loop This refactoring finds independent code blocks in a loop and lets the programmer decide which code blocks to split of into new loops. This serves as an enabling step when the programmer wants to incrementally parallelize a large loop using OpenMP or CUDA. It is also helpful when he wants to split of parts of a loop that does not contain loop carried dependencies and that can therefore be parallelized.

Unroll Loop This refactoring checks whether it is safe to unroll a loop and then unrolls (or re-rolls) it by a given factor. Unrolling can be useful to reduce branch overhead and to improve the compiler's ability to schedule code, but excessive unrolling can end up causing ICache misses (and even hurt scheduling). This refactoring allows the programmer to quickly test different unroll degrees to find the one that yields the best performance. See Section 3 for a discussion on how to maintain readability when expressing performance transformations like Unroll Loop in the source code.

Block Loop This refactoring establishes whether it is safe to convert a linear loop to a blocked/tiled loop and then, with input from the programmer, performs the transformations. This can drastically improve the cache hit ratio since the program only operates on a small block of data at a time.

3. Annotations and Views

We realize that some of the loop-related refactorings in the previous section, such as Unroll Loop and Block Loop, are traditionally performed by optimizing compilers. However, the compilers knowledge about the performance of the final code is very limited, due to its static nature and the complexity of modern hardware. The former means that it can not know which paths the program will take at runtime. The latter means that even if it could, it would need a full model of the hardware to predict the performance. It is therefore forced to make educated guesses and it often guesses wrong.

The programmer, on the other hand, does not need to guess. He can use a profiler, or manually instrument the code with timers, and then explore the space of optimizations effectively using his intuition. For this reason, performance programmers still manually implement these optimizations when they need near-optimal performance. The refactorings in the previous section help them do this safely.

Furthermore, compilers often cannot perform loop transformations due to dependencies that break transformation preconditions. Sometimes these dependencies are real, but in many cases they are artifacts of the conservative nature of compiler analysis. In fact, compilers are forced to be overly conservative, due to their static nature, the NP-Completeness of the dependency testing problem, and the tradeoff between compile time and powerful pointer analysis.

With the programmer in the loop more transformations are possible, as demonstrated by the Parascope project [2, 8]. The programmer can tell the refactoring engine to ignore dependencies he knows to be incorrect, or rewrite the code once the refactoring environment has pointed them out.

However, it would be better if the performance programmer had the best of both worlds. He keeps his source code readable *and* can take control of loop optimization when needed. The programmer decides when he should be in the driver seat and has the tools to help him drive fast and safe.

To aid readability, we therefore propose that refactoring engines insert source annotations instead of transforming loops directly. The refactoring engines will also analyze the code to determine that transformations are safe. By keeping the source code free of performance constructs that are unrelated to the algorithm the programmer wants to express, he can more easily focus on solving the problem at hand.

To further tune performance, the programmer can expand the annotations in a performance view to understand the performance characteristics of the code, and use this understanding to improve the annotations.

The expanded source code lets the programmer reason about the performance of the code, while the original source code allows him to reason about its meaning. For example, when profiling he would want to use the performance view, while the normal view would be more suitable for debugging. Finally, the IDE tracks the annotations and indicates an error if the programmer later changes the source in a way

Project	Target Lang.	Impl. Lang.	Features
Cetus	C	Java	AST, CFG, Callgraph, Inter-Procedural Pointer Analysis (Steensgaard), Loop Dependence Analysis
clang/LLVM	C, C++, Objective-C	C++	AST, CFG, Incremental Compilation, SSA IR, Callgraph, Inter-Procedural Pointer Analysis (DSA, Anderson, Steensgaard), Loop Dependence Analysis
Elsa/Pork/ Dehydra/Treehydra	C, C++,	C	AST, Scriptable AST Analysis
LLNL Rose	C, C++, Fortran, OpenMP, UPC	C++	AST, CFG, Callgraph, Inter-Procedural Pointer Analysis (Steensgaard), Loop Dependence Analysis

Table 1. Candidates for a Refactoring Analysis Backend

that is inconsistent with the annotation preconditions.

When the programmer wants to build a project, the IDE first passes the code to an internal source-to-source annotation compiler. This compiler expands annotations before it passes the code to the target compiler. The annotation compiler is also available as a command line application for programmers who need to build the project outside of the IDE.

The annotations are in the form of pragma directives that are ignored by C preprocessors. Therefore, the program will still work without the annotation compiler.

4. A Framework for Advanced Refactoring

Users have come to expect certain features from a modern refactoring framework. This includes invoking refactorings directly from the editor, previewing changes before they are applied, and the ability to undo and redo changes. Eclipse provides a rich infrastructure that has these features, in addition to an AST program representation, visitors, and rewrite capability. CDT and PTP extend Eclipse with support for C, C++ and MPI. Moreover, PTP provides numerous other useful features for HPC programmers. We therefore believe the best option is to base the performance refactoring infrastructure we envision on Eclipse with CDT and PTP.

However, the refactorings we discussed in the previous section require more sophisticated analysis than traditional refactorings. It is therefore necessary to augment the Eclipse infrastructure with a powerful analysis framework. To demonstrate this we discuss two refactorings and the analyses that are necessary to implement them.

Consider the Restrict Pointer refactoring. To safely add the restrict qualifier to a pointer the refactoring must verify that no other pointers accessed in the same block refer to any of the memory areas that are modified through the restricted pointer. This requires inter-procedural alias analysis to check for aliases and array disambiguation to find out if array accesses through pointers refer to the same parts of an array.

Now consider the Split Computation Into Communication Dependent and Independent Parts refactoring. It requires dataflow analysis, array disambiguation and possibly shape analysis to find out which parts of a data structure are sent to other processes. It must also do loop dependence analysis and pointer analysis to find if there are loop carried dependencies that prevent it from peeling of loop iterations.

Table 1 contains four analysis framework candidates that we surveyed. Given the need for a mature framework with powerful analysis data structures we believe LLNL Rose [12] and clang/LLVM [11] are the most promising candidates. In order to investigate their suitability we have created an Eclipse plugin that allows refactoring plugins to access the Rose compiler APIs. We have also started to implement the Restrict Pointer refactoring on top of this plugin. When this has been completed, we are planning to do the same using clang/LLVM.

5. Conclusions

Not long ago, refactoring to improve the design of code was impractical and only done by a select group of hero programmers. Refactoring tools empowered the average programmer to explore the design space like a pro.

A similar situation exists today for performance refactoring. However, these require more in-depth analysis than design refactorings, which means that they can be even more useful to automate. In the next decade, performance refactorings can become as transformative as design refactorings.

Acknowledgments

This work was funded by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by Intel Corporation and Microsoft Corporation.

References

[1] V. R. Basili, et. al. *Understanding the High-Performance-Computing Community: A Software Engineer's Perspective*. IEEE Software, July/August 2008.

[2] Cooper, K.D., et. al. *The ParaScope parallel programming environment*. Proceedings of the IEEE, Vol. 81, 1993.

[3] D. Dig, et. al. *Refactoring sequential Java code for concurrency via concurrent libraries*. ICSE, 2009.

[4] D. Dig, et. al. *ReLooper: Refactoring for Loop Parallelism*. OOPSLA, 2009.

[5] C. Ericson. *Real-time Collision Detection*. Elsevier, 2005.

[6] T. Hoefler and S. Gottlieb. *Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes*. EuroMPI, 2010.

[7] ISO JTC 1 Working Group. *Rationale for International Standard — Programming Languages — C*. 2003.

[8] K. Kennedy and K. S. McKinley and C. W. Tseng. *Interactive Parallel Programming using the ParaScope Editor*. IEEE Transactions on Parallel and Distributed Systems, Vol. 2, 1991.

[9] F. Kjolstad, et. al. *Refactoring for Immutability*. UIUC TechReport, 2010, www.ideals.illinois.edu/handle/2142/16399

[10] F. Kjolstad and M. Snir. *Ghost Cell Pattern*. ParaPLoP, 2010.

[11] C. Lattner and V. Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. CGO, 2004.

[12] D. Quinlan. *ROSE: Compiler Support for Object-Oriented Frameworks*. CPC, 2000.

[13] W. Tansey and E. Tilevich. *Efficient Automated Marshaling of C++ Data Structures for MPI Applications*. IPDPS, 2008.

[14] G. Watson. *The Parallel Tools Platform: A Development Environment For High Performance Computing*. EclipseCon'10.