Automating Parallel Programming

John Miller
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000

Computerman790@gmail.com

ABSTRACT

The need for parallel programming is becoming more apparent with each passing year. Although research papers and discussions on this topic have been happening since the 1980s, recent discoveries, such as the heat and power-wall, have thrust the once timid discussion into the spotlight as one of the most promising solutions to maintain performance increases.

This article discusses the complexities facing the modern-day parallel programmer and the most prominent solutions to simplifying the unstable task of balancing resource allocation, thread-safe approaches, and hardware / software optimizations.

Topics of discussion include automated pre-compile time optimizations, compile-time and run-time optimizations, and programming structure changes that allow for automatic tuning and optimizing of parallel code.

1. INTRODUCTION

Parallel programming is a complex and difficult subject from many different angles for such a simple and obvious solution in theory. Commerce is not slowing down and the demand for greater speed and broader capabilities in processors will always adjust to the very limits of current technology. For many years to come, it will always be in the interest of processor manufacturers to make room for faster/stronger/better models until a time where either we cannot go any faster or cannot approach the limits of our current capabilities.

Parallel Processing is the current-day market's most promising savior now that we've reached a physical limitation on processors known as the Heat and Power Wall. Unfortunately, the manner of constructing a parallel program to utilize all these new resources is a very complex, very young, and very diverse issue.

This article seeks to unveil the most promising approaches to making parallel programming more accessible and manageable from the perspective of the programmer. The subjects of Parallel Processing and Parallel Programming are assumed to go hand-in-hand but hardware optimizations will not be considered for the sake of brevity.

2. DEFINITIONS

2.1 Heat and Power Wall

The heat and power wall issue mentioned previously is a term used to describe the physical limitations on processor frequency vs processor heat dissipation.

In layman's terms, once a processor is run faster than a certain speed, we're unable to efficiently cool the internal components without affecting performance. As a result, multi-core technologies were developed as a work-around in order to continue performance gains.

2.2 Thread-safe

Thread-safe is a term used to describe a full or partial program designed to run partially/fully independent of other threads and programs in a manner that does not cause unexpected execution states.

In other words, its interactions with system resources, as well as other threads, are well defined and carefully designed so that, no matter what execution state other thread-safe applications are at, it is unlikely the computer will encounter a fatal interaction between them.

A proper parallel program is assumed to be thread-safe

2.3 IDE (Integrated Development Environment)

A programming environment that assists with program development. Used by programmers to create computer programs.

2.4 Deadlock

A state of execution in which two or more processes are waiting for resources the other has obtained. Since the un-obtained resources are required to continue, the processes continually wait on each other, causing execution to halt.

2.5 Distributed Computing

Using multiple computer systems to work collectively towards a specific goal. Used by research labs to solve large / complex problems that require calculation-intense solutions, among other uses.

3. PARALLEL PROGRAMMING LIMITATIONS

3.1 Shared Resources

The greatest hindrance to parallel execution and perhaps the most difficult to solve is ensuring shared resources are thread-safe. Before concurrent thread executions, sequential programing's nature ensured a stable, easy to understand approach to program execution.

Now that parallel programming is taking center stage, we have to ensure that all resources and shared data are being properly allocated.

3.2 Concurrent Usage

The most common source of parallel program bugs arises from shared data not being properly protected while in use.

For instance, if a piece of data is being written to and another thread attempts to access or write to it, the results are undefined, resulting in undesirable behavior or even halting execution. Many languages were not originally developed with robust parallel programming support and have been retrofitted with different access control methods whose behavior must be honored by other sections / threads of code in order to maintain program stability

3.3 Different Systems and Architectures

Another pitfall is varying types of optimizations for different types of environments. It's extremely difficult, if not impossible, for programmers to manually ensure parallel programs are optimized for all the different software / hardware platforms they'll be run on, yet this could be crucial to ensuring the best possible user experience.

4. COMPILE-TIME OPTIMIZATIONS

Compile-time optimizations use algorithms and special techniques to analyze program code and translate, at least in part, sections of code into parallel instructions.

There are several ways to approach this (analyzing byte-code of serially compiled applications, analyzing code directly, etc) that each have their own advantages and disadvantages.

Though a specific example may be given and analyzed in part further on, we'll analyze some of the most common advantages and disadvantages.

4.1 Advantages

One advantage to this approach is that it's a clean, automated way of introducing parallelism that works transparently to the programmer and user.

Recent additions to several key parallelizing compilers have introduced powerful code analysis and significant achievements towards making fully automatic parallelization a closer reality. [6]

They are getting smarter, faster, and more reliable as time marches on. One of the most prominent approaches to compile-time optimizations is thread level speculation.

4.1.1 Thread Level Speculation (TLS)

Using special hardware support, TLS checks for dependencies across tasks executing in parallel. If the tasks are found to have cross-dependencies, they are halted and restarted by the hardware.

TLS compilers work to construct the code in an optimized fashion in order to prevent as many dependencies as possible without having to fully prove complete thread independence [6]

4.2 Disadvantages

The biggest drawback of automatic parallel compilers is their inability to fully recognize parallelizable code. Sequential code that might very easily be converted to parallel might be overlooked if it does not directly conform to predefined patterns the compiler is looking for. [3]

This approach generally encourages programmers to continue to write in serial code, rather than challenging them to evolve their skills.

Another drawback is compile-time support for parallelizing each language wildly varies in terms of functionality, accessibility, and approach. Of all the attempts at these compilers, most are maintained by outside research organizations that may not see the project through to completion.

Finally, compilers that make optimizations based on the computer's environment, taking into account factors such as hardware and software specific optimizations, run-time environment and post-runtime analysis are few and far between. Because of this, some 'optimizations' made by the compiler may, in fact, hurt performance on different systems. [2][5]

For example, different hardware / software configurations have different worst-case and best-case scenarios when it comes to order and execution of instructions. What may be an optimized approach for a Windows system with a large bank of system memory but small L1 & L2 caches may be a worst-case scenario for a Linux system with low system memory but high-capacity processor caches.

Thus, not only do compilers need to make algorithmic-specific optimizations for maximum efficiency, but they also need to make decisions based on architectural properties and resource availability.

4.3 Special Mention: Concurrencer

Concurrencer is a tool for the Java language that attempts to bridge some of the gaps and inconsistencies between programmers and their compilers.

Providing pre-compile time code analysis, Concurrencer works to convert sequential code into parallel sections and objects by using thread-safe classes to replace variables / other classes and rewrite code (which can originate from multiple separate sections of code) into a single, parallelized set of statements. [3]

It achieves this mostly by AtomicIntegers, Concurrent Hashmaps, and utilizing the Java ForkJoinTask framework.

4.3.1 AtomicInteger and Concurrent Hashmap

By converting an integer from a primitive type to a new class type of AtomicInteger, Concurrencer protects against concurrent access and allows loop-based parallelism to be thread-safe. This same approach is used for Hashmaps, replacing Java's HashMap class with a concurrent version.

See Table 1 for an example of the AtomicInteger implementation.

Table 1. Concurrencer replaces accesses to field f with calls to AtomicInteger APIs (e denotes an expression). [3]

Access	int	AtomicInteger
Read	f	f.get()
Write	f = e	f.set(e)
Cond. Write	if (f==e) f=e$_1$	f.compareAndSet(e,e$_1$)
Prefix Inc.	++f	f.incrementAndGet()
Postfix Inc.	f++	f.getAndIncrement()
Infix Add	f = f + e	f.addAndGet(e)
Add	f += e	f.addAndGet(e)
Prefix Dec.	--f	f.decrementAndGet()
Postfix Dec.	f--	f.getAndDecrement()
Infix Sub.	f = f - e	f.addAndGet(-e)
Subtract	f -= e	f.addAndGet(-e)

4.4 Special Mention: Post-Mortem Analysis

Post-compile time analysis is another tool used to assist the programmer in identifying sections of un-optimized code and bottle necks.

Post-Mortem Analysis collects data on a program as it executes and once it exits, presents performance and execution data to the user.

4.4.1 Advantages

Post-Mortem Analysis helps identify major issues in program run time and can lead to practical analysis of program's execution. Its greatest utility is helping the programmer know which sections of their program that need to be optimized.

4.4.2 Disadvantages

The data provided is only for one run and is system specific. The programmer would have to manually analyze information for multiple runs and would have to test the program on each type of architecture to discover any hardware / software related optimization issues.

This approach only helps analyze the program's internal workings and gives a rough idea of its execution sequence for one run. Parallel programs are notorious for having different patterns of execution for every single run.

5. RUN-TIME OPTIMIZATIONS

Dynamic Tuning Environments (DTEs) are becoming more applicable and practical as the field of distributed computing grows. Using real-time analysis, the environment adjusts loads and threads based on expected vs actual time analysis, allowing for long-term systems to respond to current environment limitations and adjust accordingly, rather than optimizing as best as possible while compiling and hoping for the best once the program is run.

The DTE system monitors threads and compares their current processor time to a pre-calculated value of expected time while analyzing each node (distributed computer)'s scheduled tasks and dependencies. Based on this and other data, tasks and threads are swapped between nodes to ensure high-performance and efficient use of resources. [7]

This approach was indirectly created as a response to the limitations post-mortem analysis faces.

5.1 Advantages

Real-time response to current environment conditions is needed when working with distributed computing, especially when those systems are open to public requests.

Load-balancing and high CPU utilization are the greatest advantages over a statically compiled / compiler-optimized parallel program. [7]

In addition, and in contrast to post-mortem analysis, this is an automated solution that works independent of the programmer.

Finally, many research implementations of this approach provide automatic parallelization of the program, although strict guidelines to program structure are required. [7]

5.2 Disadvantages

This solution is mainly for long-term, resource intensive parallel applications and most implementations are (for good reason) specifically made for distributed computing environments.

It also presents more overhead as decisions are constantly being reevaluated as data streams in from all nodes. The model itself is usually implemented on a single dedicated node, preventing multiple nodes from sharing the burden of making decisions.

5.3 Case-Study: MATE

MATE is a diverse, well-developed DTE that overcomes many challenges that other DTEs face.

5.3.1 Monitoring

Many DTEs require special compilers that prepare programs for the tuning environment. MATE, on the other hand, is designed to insert code on the fly, so potential bottle-necks that are identified post-compile time can easily be monitored. [7]

5.3.2 Performance

MATE is primarily focused on resource utilization and preventing bottle-necks, however, its own behavior has the potential to create bottlenecks where there might not have been any.

For example, MATE will attempt to look ahead and preform calculations before they're needed if current performance allows. However, if these calculations require more time than expected and an 'if' statement renders those calculations unnecessary, excess overhead is produced. [7]

6. IDE & DEVELOPMENT OPTIMIZATIONS

One of the most promising solutions for maximizing the balance between automation and performance is updating the coding environment and practices that programmers utilize and combining it with powerful compilers and run-time environments that use the new structure to its utmost potential.

These new environments would likely implement a set list of different types of problems to be solved in parallel (such as divide and conquer or concurrent executions of different iterations of a loop).

6.1 Advantages

Predictable patterns in parallel conversion techniques allow compilers to focus on optimizing a limited set of solvable problems rather than focusing their energy / time on detecting and translating solutions into parallelized and solvable problems.

Environment optimizations would be possible, as optimizations for each type of problem could be easily defined for the rigorous standards this implementation would require.

By its nature, all program elements are easily ensured to be thread safe as long as the model for each algorithm is well defined.

Given a set of specific behavioral boundaries, run-time managers, in theory, would be able to provide features of DTEs on non-distributed environments, allowing for run-time optimizations of shorter-term programs.

6.2 Disadvantages

Having only a limited set of parallel solutions to choose from could present difficulties for those unfamiliar with parallel algorithms and their uses.

Has the potential to discourage or at least complicate program specific optimizations by the programmer.

6.3 Case-Study: Skeleton Programming Model

Of the many proposed solutions, the most developed and, perhaps, the most well-known of all approaches is the Skeleton Programming Model (SPM) proposed in 1989.

SPM has 8 well-defined modules (or skeletons) that are intended to solve specific types of parallel problems and has a specific model devoted to sequential code.

6.3.1 SPM Advantages

Most skeleton implementations provide translation functions which, if required, allows the language / compiler to translate one implemented skeleton model into another. This allows for system-specific optimizations, so even if two systems vary widely in how to optimize programs for their specific environment, each skeleton can be optimized for that environment. [2][4]

Another common feature of skeleton implementations is nested skeletons, which is often implemented alongside translation abilities. This feature is crucial in object oriented languages, which allows for each skeleton to have parent, sibling, or child relationships with other skeletons rather than approaching each problem with only type of parallelism. Specialized optimizations, based upon implementation, are defined between each of these relations and allow for a great deal more of diversity in solving problems in parallel. [4]

Thread safety and isolation is at the core of this model, ensuring that thread interactions are kept to their level of nesting (unless the parent intervenes), thus removing all the hassle of anticipating thread interactions and preventing them.

6.3.2 SPM Disadvantages

As of 2010, there were about 21 well-known implementations of the skeleton framework for 3 major programming languages (C, Java, C++) (See table 2). Requirements and even skeleton behavior vary greatly from language to language. Skeleton semantics have been known to vary widely in some implementations. That is, an implementations will use one of the common 8 skeleton names for a skeleton to solve a completely different type of problem the original skeleton was intended for.

Each implementation also varies in how many and which kind of skeleton models it supports, the degree to which translation and nesting is supported, and the type of optimizations available.

Framing your ideas into a combination of 3-15 different problems can make a simple program much more complex and has the potential to make it less efficient if an in-depth understanding of the skeletal framework is missing.

Table 2. Comparative table of the algorithmic skeleton frameworks considered in [4]

	Programming language	Execution language	Distribution library	Skeleton set
Alt	Java	Java	Java RMI	map, zip, apply, scan, sort, reduce, replicate
ASSIST	Custom control lang.	C++	TCP/IP + ssh/scp	seq, parmod
Calcium	Java	Java	ProActive	seq, pipe, farm, for, while, map, d&c, fork
Eden	Haskell (extension)	C	PVM / MPI	map, d&c, pipe, iterUntil, torus, ring
eSkel	C	C	MPI	pipe, farm, deal, Butterfly, hallowSwap
HDC	Haskell (subset)	C	MPI	map, red, scan, filter, dca, dcB, dcD, dcE, dcF
HOC	Java	Java	Globus	farm, pipe, wavefront
JaSkel	Java	Java	RMI	farm, pipe, heartbeat
Lithium	Java	Java	RMI	pipe, map, farm, reduce
Mallba	C++	C++	NetStream MPI	exact, heuristic, hybrid
Muesli	C++	C++	MPI OpenMP	array, matrix, farm, pipe, parallel comp.
Muskel	Java	Java	RMI	farm, pipe, seq, custom
P^3L	Custom control lang.	C	MPI	map, reduce, seq, comp, pipe, farm, scan, loop
QUAFF	C++	C	MPI	seq, pipe, farm scm, pardo
SAC	Custom	C-like	Threads	genarray, modarray, fold
SCL	Custom control lang.	Fortran	*Ad hoc* Tools	map, scan, farm, fold, SPMD, iterateUntil
Skandium	Java	Java	Threads	seq, pipe, farm, for, while,map, d&c, fork
SKELib	C	C	MPI	farm, pipe
SkeTo	C++	C++	MPI	list, matrix, tree
SkIE	GUI / Custom control lang.	C++	MPI	farm, pipe, map reduce, loop
Skil	C (subset)	C	—	pardata, map, fold
SkiPPER	CAML	C	SynDex	scm, df, tf, intermem

7. SUMMARY/CONCLUSION

The solution to parallel programming complexity is not clear-cut, quite the opposite in fact, and varies depending on programmer experience, complexity, and chosen problem to solve.

Parallel programming adds several degrees of complexity but also promises some of the greatest advantages and solutions to both

speed and diversity in processing improvements. Its greatest hindrance now is its complexity.

8. References

[1] Chen, Michael K. and Kunle Olukotun. "The Jrpm System for Dynamically Parallelizing Java Programs." *Proceedings of ISCA-30, 9-11*. San Diego, 2003.

[2] Darlington, John, et al. "Parallel Programming Using Skeleton Functions." *PARLE'93, 5th International PARLE Conference on Parallel Architectures and Languages Europe* (1993): 146-160.

[3] Dig, Danny, John Marrero and Michael D. Ernst. "Refactoring sequential Java code for concurrency via concurrent libraries." *ICSE '09 Proceedings of the 31st International Conference on Software Engineering* (2009): 397-407.

[4] González-Vélez, H. and M Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers." *Software: Practice and Experience* (2010): 1135-1160. Electronic.

[5] Krishnan, Sanjeev and Laxmikant V. Kale. "Automating parallel runtime optimizations using post-mortem analysis." *ICS '96 Proceedings of the 10th international conference on Supercomputing* (1996): 221-228.

[6] Liu, Wei, et al. "POSH: A TLS compiler that exploits program structure." *In Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York City, 2006.

[7] Scutari, Paola G.C. *Extending the Usability of a Dynamic Tuning Environment*. Barcelona: Departament d'Arquitectura de Computadors i Sistemes Operatius, 2007.