A Study of the Usability of Multicore Threading Tools

Ami Marowka
Department of Computer Science
Bar-Ilan University, Israel
amimar2@yahoo.com

Abstract
The primary consequence of the transition to multicore processors is that applications will increasingly need to be parallelized to fully exploit the throughput gains now becoming available. Unfortunately, parallel programming is no doubt much more tedious and error-prone than serial programming. Software vendors have begun efforts to educate developers and provide them with better tools for multicore programming. But many of the tools available are still works in progress.

This paper presents a preliminary study of the usability of the state-of-the-art threading tools for multicore programming. We report on our evaluation of the usability of two Intel threading tools: Intel Thread Profiler - a visual profiling tool for monitoring the parallelism gain of parallel applications and Intel Thread Checker - a sophisticated tool for finding potential deadlocks and data race conditions in a parallel code.

Keywords: Multicore, Usability, Threading Tools, Profiler, Debugger.

1. Introduction

Parallel computing is rapidly entering mainstream computing, and multicore processors can now be found in the heart of supercomputers, desktop computers, and laptops. Consequently, applications will increasingly need to be parallelized to fully exploit the multicore processor throughput gains that are becoming available. However, the promising potential of future many-core processors will be greatly diminished if the industry cannot overcome certain programming challenges [1, 2, 4].

Parallel software developers must contend with problems not encountered during sequential program development. Non-determinism, communication, synchronization, heterogeneity, load balancing, deadlock and race conditions present new challenges [3, 5]. System builders have begun efforts to educate developers and provide them with better tools for multicore programming. But many of the tools available are still works in progress. Currently, the responsibility to bridge the gap between hardware and software to write better parallel programs may ultimately lie with developers. Many programmers are not up to speed on the latest developments in hardware design. They should study chip architectures to understand how their code can perform better. This is not a desirable situation.

Usability is the most important factor of multicore system since it directly influences the productivity of programmers. Given the extra complexity of debugging and testing parallel applications, it is essential that a multicore system eliminate, reduce, or at least mask the complexity.
Debugging and profiling a parallel program is a highly tedious and difficult task. Although parallel debuggers and visual profiling analyzers exist and are improving all the time, finding a bug in a parallel program is like finding a needle in a haystack. The complexity of parallel debugging is due to the invisible problems and to the timing complexity of parallel program flow that harden on finding temporary bugs, whose appearance cannot be predicted.

The aim of our usability study is to learn how easy is to design, develop, writing, benchmarking and debugging application for multicore systems. The contribution of this work is to present a preliminary report of a long-term study of the usability of current state-of-the-art debuggers; profilers and analyzers tools for multicore programming that are available in the market today. We report on our evaluation of the usability of two Intel threading tools: Intel Thread Profiler [7] - a visual profiling tool for monitoring the parallelism gain of parallel applications and Intel Thread Checker [6] - a sophisticated tool for finding potential deadlocks and data race conditions in a parallel code.

The rest of this paper is organized as follows. In Section 2, we present the profiling abilities of Intel Threading Profiler. Section 3 presents the functionality of Intel Threading Checker. Section 4 analyses the usability of the tools presented in the previous sections, and Section 5 concludes the paper.

2. Code Profiling Tools

Profiling is a technique for measuring where software programs consume resources, including CPU time and memory. Code profiling tools are visual instruments that help to expose performance bottlenecks and hotspots that inhibit to achieve the desired code parallelism. The instrumentation is done by gathering information, such as execution times and number of calls, during execution of program's entities such as functions and loops. The outcome is visualized graphically on the screen and enables the programmer to detect, for example, the imbalance in either computation or communication that is present in an algorithm.

Intel Thread Profiler is a profiling tool that identifies bottlenecks that limit parallelism of threaded applications and locates overheads due to synchronization, stalled threads and long blocking times. Thread Profiler supports applications threaded with Windows threads, POSIX threads and OpenMP [8, 9].

Thread Profiler creates two kinds of views for analyzing the behavior of threaded application: Profile view and Timeline view. While the Timeline view is more intuitive to the way a sequential programmer think, the Profile view demands to think in parallel which is a different way of thinking.

The Profile view can be grouped by Concurrency Level, Thread, Object, Object Type, or Source Stack. A detailed colorful legend helps to identify immediately which objects incur the hotspots and the bottlenecks and by clicking on the suspicious object's bar a detailed timing information is displayed. For certain types of data, it is also possible to "drill down" to the source code locations as shown in Figure 2.
The Profile view displays the time where the application spent on the *critical path*. The *critical path* is the longest flow of execution. Figure 1 illustrates a typical Profile view of a program called Primes.exe. In this view the results are grouped by objects and each object is grouped by *concurrency level*. The concurrency level is the number of active threads executing at the same time on the critical path. For example, the object FindPrimes in Figure 1 spent a neglect time (yellow bar that means overhead) on concurrency level 0 (CL-0) when all threads in the application are either sleeping or blocked. Half of the time the object FindPrimes was used by one thread (CL-1, red bar that means under utilization). In the rest of time it was used by two threads simultaneously (CL-2, green bar that means fully utilized).

The Timeline view displays the behavior of the program over time and across threads. Figure 3 demonstrates a scenario where at the beginning of the program a master thread is invoked and after some time the master thread spawns two child-threads that are running in parallel till the end of the program. The colorful legend represents time categories and enables to identify the status of each thread (Active – dark green, wait – light green), the critical path data (Under utilization – red, fully utilization – green) and transitions (Fork-join – purple, transition – yellow). Moreover, it is possible to zoom in on any time section along the Timeline graph by simply dragging the mouse over the desired section. Another useful feature is the possibility to hover the mouse over anyone of the elements in the Timeline view’s legend and the corresponding element flash in the graph, helping the programmer to locate or identify different elements in the graph. Likewise, the Timeline view enables to drill-down to source-view for locating the places where an event is occurring inside the code.
3. Deadlocks and Data Race Conditions

Multiple threads accessing shared data simultaneously may lead to a timing dependent error known as data race condition. Data races may be hidden in the code without interfering or harming the program execution until the moment when threads are scheduled in a scenario (the condition) that break the program execution. Moreover, usually data races cannot be detected just by inspecting the source code.

To avoid unpredictable situations such as race conditions, mutual exclusion techniques are used to impose constraints on the order that threads access a shared-memory location. An exclusive access to a shared object can be guaranteed by using locks, also called mutexes.

Improper use of locks may cause many hard problems that are very difficult to detect without a sophisticated debugging tool. One such situation is known as deadlock. A deadlock
is a situation in which a task A is waiting to acquire a lock on a shared object \( r1 \) locked by a task B, while locking a shared object \( r2 \) requested by task B. Since both tasks are blocked and waiting for the release of the object held by the other task, and none of them volunteers to be the first to release its object, the program execution is stuck. There are four conditions that lead to a deadlock situation:

1. **Exclusiveness**: Exclusive assignment of an object to a task.
2. **Multilock**: Allowing a task to acquire a lock on one object while locking another object.
3. **Ownership**: A locked object can be released only by the task that holds it.
4. **Cycling**: A task is willing to acquire a lock on an object held by another task that willing to acquire a lock on an object held by him.

Deadlocks can be avoided by breaking any one of these conditions. One way to avoid deadlocks is to impose an ordering (eliminating cycles in the resource acquisition graph) on the locks and demand that all threads acquire their locks in the same order. Other techniques to prevent deadlock are **timer-attached mutex** and **exception-aware mutex**. In a **timer-attached mutex** a timer is attached to the mutex, thus guaranteeing that the mutex will be released after a predetermined time if a release operation has not been invoked before. An **exception-aware mutex** is a technique that ensures that a mutex gets released when an exception occurs.

<table>
<thead>
<tr>
<th>Type</th>
<th>ID</th>
<th>Short Description</th>
<th>Severity</th>
<th>Description</th>
<th>Count</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write -&gt; Read</td>
<td>1</td>
<td>Data-race</td>
<td>X</td>
<td>Memory read at &quot;primes.cpp&quot;:43 conflicts with a prior memory write at &quot;primes.cpp&quot;:44 (flow...)</td>
<td>69</td>
<td>False</td>
</tr>
<tr>
<td>Write -&gt; Write</td>
<td>2</td>
<td>Data-race</td>
<td>X</td>
<td>Memory read at &quot;primes.cpp&quot;:44 conflicts with a prior memory write at &quot;primes.cpp&quot;:44 (flow...)</td>
<td>89</td>
<td>False</td>
</tr>
<tr>
<td>Write -&gt; Write</td>
<td>3</td>
<td>Data-race</td>
<td>X</td>
<td>Memory write at &quot;primes.cpp&quot;:44 conflicts with a prior memory write at &quot;primes.cpp&quot;:44 (output dependence)</td>
<td>89</td>
<td>False</td>
</tr>
<tr>
<td>Write -&gt; Write</td>
<td>4</td>
<td>Data-race</td>
<td>X</td>
<td>Memory write at &quot;primes.cpp&quot;:43 conflicts with a prior memory write at &quot;primes.cpp&quot;:43 (output dependence)</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>Read -&gt; Write</td>
<td>5</td>
<td>Data-race</td>
<td>X</td>
<td>Memory write at &quot;primes.cpp&quot;:44 conflicts with a prior memory read at &quot;primes.cpp&quot;:44 (anti dependence)</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>Thread termination</td>
<td>6</td>
<td></td>
<td></td>
<td>Thread termination at &quot;primes.cpp&quot;:60 - includes stack allocation of 1 KB and use of 4 KB</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>Thread termination</td>
<td>7</td>
<td></td>
<td></td>
<td>Thread termination at &quot;primes.cpp&quot;:50 - includes stack allocation of 1 KB and use of 4 KB</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>Thread termination</td>
<td>8</td>
<td></td>
<td></td>
<td>Thread termination at &quot;primes.cpp&quot;:50 - includes stack allocation of 1 KB and use of 4 KB</td>
<td>1</td>
<td>False</td>
</tr>
</tbody>
</table>

Figure 4. A snapshot of the diagnostic view of Intel Thread Checker

Intel Thread Checker is a tool that detects data races and deadlocks. Moreover, Thread Checker is able to detect potential deadlocks and data races that may occur in the future. However, **Thread Checker cannot detect every data race condition or a deadlock because they are intermittent, non-deterministic, non-repeatable and timing dependent problems where the number of the schedule permutations of threads is huge**. Figure 4 shows an example of a diagnostic view created by Thread Checker. In this example, five data races were detected of type's write-after-read, read-after-write and write-after-write. A short description explains the meaning of the error and reports where it occurred in the code.
including the line number in the source code. Thread Checker enables the programmer to drill-down to locate the conflict in the source code as shown in Figure 5.

4. Usability Analysis

Intel threading tools, Thread Checker and Thread Profiler, are easy to use, user friendly and have rich visualization features that help to make the debugging process more communicative. However, we found a few issues that call for improvements:

**Learning curve** – it takes a long time for novices to learn these tools and even experienced parallel programmer need time to understand their functionality. There are no books that can help. The User Guides of the products contain brief tutorials with very simple examples. More examples are needed to demonstrate more hard cases. Moreover, the tools use a lot of terms that the programmer need to study. This terminology and visualization features are changing from one version to another and demand more study with each new version. The on-line Help systems of the tools are helpful but it is not enough.

**Integration with other tools** – Each Intel Threading tools is coming separately. It should either come with a complete suite of software development tools including facilities for debugging, monitoring and performance evaluation.

**Portability** – Intel Threading tools are good for debugging and profiling programs that are running on top of Intel processors. These tools cannot be used, for an example, for AMD processors. Therefore, organization that is using many types of processors needs as many tools. Moreover, these tools support POSIX threads, Windows threads and OpenMP but not Solaris threads or .NET threads.

**Skills and knowledge** – The programmer that is using these tools must have the appropriate skills and knowledge. It is impossible to find and resolve deadlocks and data-races without understanding these issues from the theoretical and practical points of views. The user must to be an expert in parallel computing and programming. In other words, these tools are not self-contained that can be studied on-the-fly.

5. Conclusions

Parallel programming is not only hard and tedious demanding different thinking and creativity, but also a very error-prone one. The lack of multicore programming tools for mainstream developers is perhaps the biggest challenge the industry faces today. Debugging tools are improving all the time but lake sophistication. They demand from the programmer a broad knowledge about the details of the underlying architecture and familiarity with the hardware counters. Debugging tools should be able to analyze the code and report on bugs and potential bugs while freeing the programmer from the need to be hardware experts.

Parallel programming should be as simple and productive as sequential programming.
6. References


Authors

Ami Marowka obtained his PhD in Computer Science from the School of Computer Science and Engineering at the Hebrew University of Jerusalem, Israel. He was a research associate at the University of Houston, an Assistant Professor in Shenkar College of Engineering and Design and currently is an Adjunct Assistant Professor in the Department of Computer Science at the Bar-Ilan University, Israel.