**КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2015 Т. 7 № 3 С. 485−491**

Ки&М

СЕКЦИОННЫЕ ДОКЛАДЫ

УДК: 004.43

Performance of the OpenMP and MPI implementations on ultrasparc system

A. Bogdanov[1,a], P. Sone K. Ko[2,b], K. Zaya[2,c]

[1] Institute for High-performance computing and the integrated systems, St. Petersburg, 199397, Russia
2 St.Petersburg State Marine Technical University, 3 Lotsmanskaya Str., St. Petersburg, 190008 Russia

E-mail: [a] bogdanov@csa.ru, [b] pyaesonekoko@gmail.com, [c] kyawzaya4436@gmail.com

Received December 4, 2014

This paper targets programmers and developers interested in utilizing parallel programming techniques to enhance application performance. The Oracle Solaris Studio software provides state-of-the-art optimizing and parallelizing compilers for C, C++ and Fortran, an advanced debugger, and optimized mathematical and performance libraries. Also included are an extremely powerful performance analysis tool for profiling serial and parallel applications, a thread analysis tool to detect data races and deadlock in memory parallel programs, and an Integrated Development Environment (IDE). The Oracle Message Passing Toolkit software provides the high-performance MPI libraries and associated run-time environment needed for message passing applications that can run on a single system or across multiple compute systems connected with high performance networking, including Gigabit Ethernet, 10 Gigabit Ethernet, InfiniBand and Myrinet. Examples of OpenMP and MPI are provided throughout the paper, including their usage via the Oracle Solaris Studio and Oracle Message Passing Toolkit products for development and deployment of both serial and parallel applications on SPARC and x86/x64 based systems. Throughout this paper it is demonstrated how to develop and deploy an application parallelized with OpenMP and/or MPI.

Keywords: OpenMP, Parallel Programming, MPI (Message Passing Interface), SPARC System

**КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2015 Т. 7 № 3 С. 485−491**

Ки&М

Производительность OpenMP и реализация MPI на системе ultrasparc

А. В. Богданов, Пуае Сон Ко Ко, Кьяв Зайя

*[1] Института высокопроизводительных вычислений и Информационных Систем,
Россия, 199397, г. Санкт-Петербург
[2] Санкт-Петербургский государственный морской технический университет,
Россия, 190008, г. Санкт-Петербург, ул. Лоцманская, д. 3*

Данная работа нацелена на программистов и разработчики, заинтересованных в использовании технологии параллельного программирования для увеличения производительности приложений. Программное обеспечение Oracle Solaris Studio обеспечивает современную оптимизацию и распараллеливание компиляторов для языков C, C ++ и ФОРТРАН, продвинутый отладчик, и оптимизированные математические и быстродействующие библиотеки. Также включены чрезвычайно мощный инструмент анализа производительности для профилирования последовательных и параллельных приложений, инструмент анализа для обнаружения состязания при передаче данных и блокировки в памяти параллельных программ и IDE. Программное обеспечение Oracle Message Passing Toolkit обеспечивает высокопроизводительные MPI библиотеки и сопряжённую среду во время работы программы, необходимую для приложений передачи сообщений, которые могут работать на одной системе или по всему множеству вычислительных систем с высокопроизводительным сетевым оснащением, включая Gigabit Ethernet, 10 Gigabit Ethernet, InfiniBand и Myrinet. Примеры OpenMP и MPI представлены по всему тексту работы, включая их использование через программные продукты Oracle Solaris Studio и Oracle Message Passing Toolkit для развития и развертывания последовательных и параллельных приложений на основе систем SPARC и x86/x64. В работе продемонстрировано, как развивать и развертывать приложение, распараллеленное с OpenMP и/или MPI.

Ключевые слова: OpenMP, параллельное программирование, MPI (Message Passing Interface), система SPARC

Multicore Processor Technology

In a multicore processor architecture there are multiple independent processing units available to execute an instruction stream. Such a unit is generally referred to as a *core*. A processor might consist of multiple cores, with each core capable of executing an instruction stream. Since each core can operate independently, different instruction streams can be executed simultaneously. Nowadays all major chip vendors offer various types of multicore processors. A block diagram of a generic multicore architecture is shown in Figure 1.

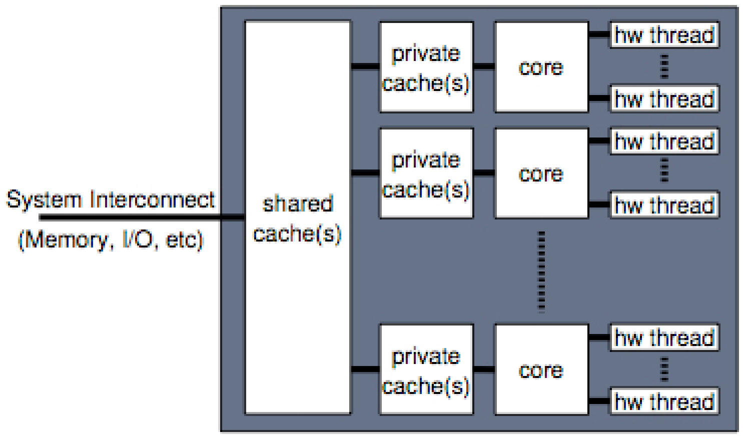

Fig. 1. Block diagram of a generic multicore architecture

In some architectures, each core has additional hardware support to efficiently execute multiple independent instruction streams in an interleaved way. For example, while one instruction stream waits for data to come from memory, another stream may be able to continue execution. This is transparent to the application and reduces, or even entirely avoids, processor cycles being wasted while waiting. It also adds a second level of parallelism to the architecture. Although a very important feature to improve both the throughput and single application parallel performance, we will not make this distinction in the remainder.

On the memory side, multiple levels of fast buffer memory can be found. These are generally referred to as *cache memory* or cache(s) for short. Today first level caches are typically local to the core. Higher-level caches can be local, but may also be shared across the cores. Typically at least the highest level of cache often is shared.

The instruction streams can be completely unrelated. For example, one might watch a video on a laptop, while having an email client open at the same time. This gives rise to (at least) two instruction streams. We say "at least" because each of these applications could be internally parallelized. If so, they might each execute more than one instruction stream.

On a dual-core processor, one core can handle the application showing the video, while the other core executes the email client. This type of parallel execution is often referred to as *throughput computing*. A multicore architecture greatly improves throughput capacity.

What is a Thread?

A thread consists of a sequence of instructions. A thread is the software vehicle to implement parallelism in an application. A thread has its own state information and can execute independently of the other threads in an application. The creation, execution and scheduling of threads onto the cores is the responsibility of the operating system. This is illustrated in Figure 2.

In general it is best for performance to make sure the hardware resources used are not overloaded and do not exceed their capacity. In case a resource is overloaded, the common phrase is to say that

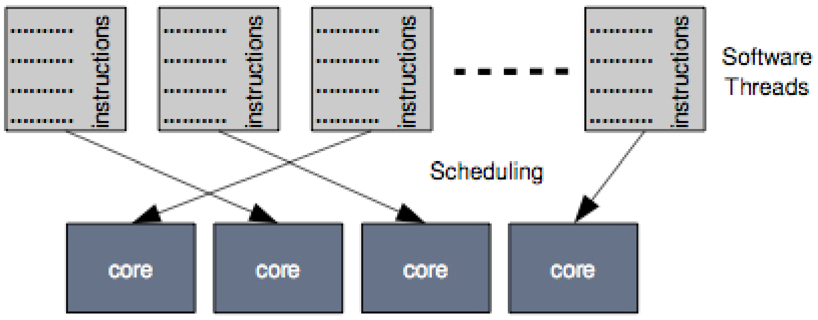

Fig. 2. Software threads scheduled onto the cores

this resource is oversubscribed. For example, when executing more than one application on a single core, the operating system has to switch between these programs. This not only takes time, but information in the various caches might be flushed back to main memory as well. In that respect, one should see the operating system itself as an application too. Its various daemons have to run in conjunction with the user level programs. This is why it is often most efficient to not use more software threads than cores available in the system, or perhaps even leave some room for these daemons to execute as well.

The exception is if a core has hardware support for multiple threads. In this case, some level of oversubscription of a core could be beneficial for performance. The number of software threads to use depends on the workload and the hardware implementation details.

On current operating systems, the user can have explicit control over the placement of threads onto the cores. Optimally assigning work to cores requires an understanding of the processor and core topology of the system. This is fairly low-level information, but it can be very beneficial to exploit this feature and improve the performance by carefully placing the threads.

To improve cache affinity, one can also pin the threads down onto the cores. This is called binding and essentially bypasses the operating system scheduler. It could work well in a very controlled environment without oversubscription, but in a time-shared environment it is often best to leave the scheduling decisions up to the operating system.

Why Parallelization?

Parallelization is another optimization technique to further enhance the performance. The goal is to reduce the total execution time proportionally to the number of cores used. If the serial execution time is 20 seconds for example, executing the parallel version on a quad core system ideally reduces this to 20/4 = 5 seconds. This is illustrated in Figure 3.

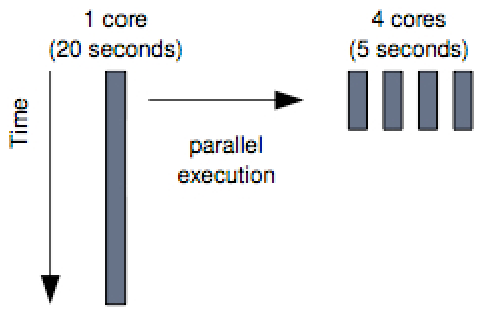

Fig. 3. Parellelization reduces the execution time

Parallelization attempts to identify those portions of work in a sequential program that can be executed independently. At run time this work is then distributed over the cores available. These units of work are encapsulated in threads.

The programmer relies on a programming model that will express parallelism inherent in an application. Such a parallel programming model specifies how the parallelism is implemented, and the parallel execution managed.

An Application Programming Interface (API) consists of a library of functions available to the developer. POSIX Threads (or *Pthreads*), Java Threads, Windows Threads and the Message Passing Interface (MPI) are all examples of programming models that rely on explicit calls to library functions to implement parallelism.

Another approach might utilize compiler directives such as #pragma constructs in C/C++ to identify and manage the parallel portions of an application's source code. OpenMP is probably the most well known example of such a model.

Parallel Architectures

In this section an overview of various types of parallel systems is given. These are generic descriptions without any specific information on systems available today.

The Symmetric Multiprocessor (SMP) Architecture
The Non-Uniform Memory Access (NUMA) Architecture
The Hybrid Architecture
The Cache Coherent Non-Uniform Memory Access (cc-NUMA) Architecture

Parallel Programming Models

There are many choices when it comes to selecting a programming model for a parallel system.

Automatic Parallelization
The OpenMP Parallel Programming Model
The Message Passing Interface (MPI) Parallel Programming Model
The Hybrid Parallel Programming Model

Performance Results

The results were obtained on a Sun SPARC Enterprise T5120 server from Oracle. The system had a single UltraSPARC T2 processor with 8 cores and 8 hardware threads per core. In Figure the elapsed times in seconds for the Automatically Parallelized and OpenMP implementations are plotted as a function of the number of threads used. Note that a log scale is used on the vertical axis.

For up to 8 threads, both versions perform equal. For 16 threads the Automatically Parallelized version performs about 9% faster than the OpenMP version.

Both versions scale very well for up to 8 threads. When using 32 threads, the performance deviation compared to the Automatically Parallelized version is about 30% for the OpenMP version. For 64 threads, the elapsed time is about twice as high. This difference is caused by the parallel overheads increasing as more threads are used. If more computational work was performed, this overhead would not be as dominant.

In Figure5 the performance of the MPI implementation is plotted as a function of the number of processes. Both the computational time (solid line) as well as the total time spent in the MPI functions (bar chart) are shown.

The time spent in the MPI functions is obviously zero if only one process is used. When using two processes it is just below one second, going up to 2.4 seconds on 64 threads. The computational work is very small. As a result, the cost of message passing is relatively dominant and no overall performance gain is achieved when running in parallel. If more work were performed, this would be different.

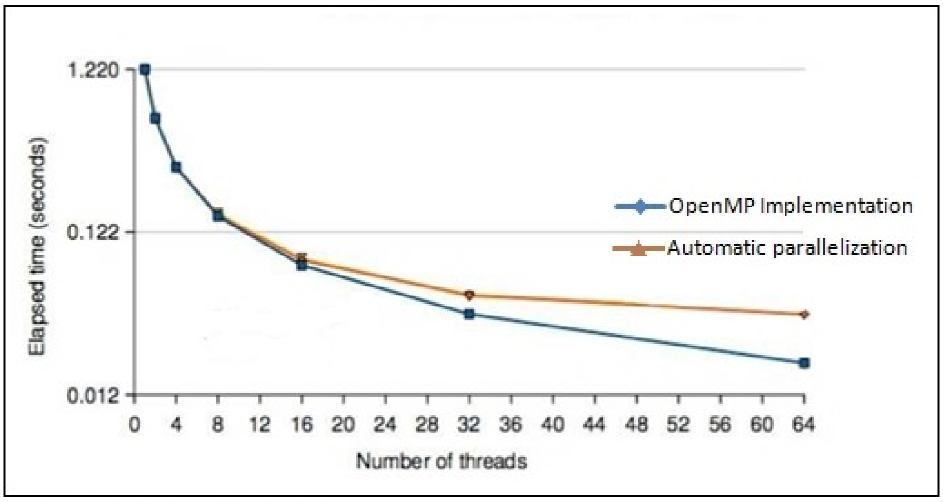

Fig. 4. Performance of the Automatically Parallelized and OpenMP implementations

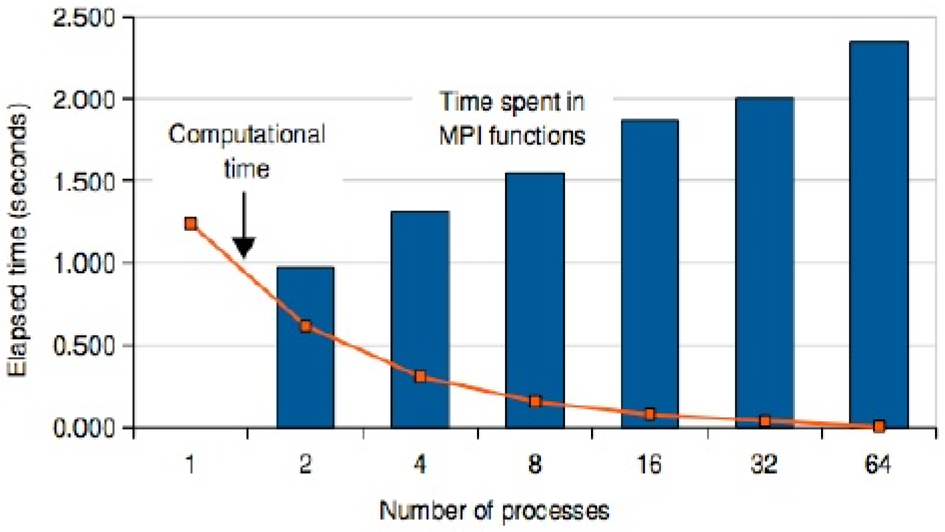

Fig. 5. Performance of the MPI implementation

Conclusion

The goal of parallel computing is to reduce the elapsed time of an application. To this end, multiple processors, or cores, are used to execute the application. The expected speed up depends on the number of threads used, but also on the fraction of the execution time that can be parallelized. Only if this fraction is very high, scalable performance to a very high number of cores can be expected. MPI is used to distribute the work over the nodes, as well as handle the communication between the nodes. More fine-grained portions of work are then further parallelized using Automatic Parallelization and/or OpenMP. Together with the Oracle Message Passing Toolkit, the Oracle Solaris Studio compilers can be used to develop and deploy these kinds of applications. OpenMP is a de-facto standard to explicitly implement parallelism. Like Automatic Parallelization, it is suitable for multicore and bigger types of shared memory systems. It is a directive based model, augmented with run time functions and environment variables. The Oracle Solaris Studio compilers fully support OpenMP, as well as additional features to assist with the development of applications using this programming model. The choice of the programming model has substantial consequences regarding the implementation, execution and maintenance of the application. We strongly recommend to carefully consider these before making a choice.

References

Barbara Chapman, Gabriele Jost, Ruud van der Pas, "Using OpenMP", The MIT Press, 2008.

Darryl Gove, "Solaris Application Programming", Prentice Hall, 2008.

High Performance Computing and Communications Glossary, http://wotug.kent.ac.uk/parallel/acronyms/hpccgloss

MPI Forum, http://www.mpi-forum.org

Open MPI Home Page, http://www.open-mpi.org

OpenMP Specifications, http://openmp.org/wp/openmp-specifications

Oracle Message Passing Toolkit Home Page, http://www.sun.com/software/products/clustertools

Oracle Solaris Studio Documentation, http://developers.sun.com/sunstudio/documentation

Oracle Solaris Studio Numerical Computation Guide, http://docs.sun.com/app/docs/doc/819-3693

Oracle Solaris Studio Performance Analyzer Reference Manual, http://docs.sun.com/app/docs/doc/821- 0304.