ORACLE

SPARC SERVERS

An Oracle Technical White Paper
August 2013

READ_ME_FIRST: What Do I Do with All of Those SPARC Threads?

ORACLE

Executive Overview

With an amazing 1,536 threads in a SPARC M5-32 system from Oracle, the number of threads in a single system has never been so high. This offers a tremendous processing capacity, but one may wonder how to make optimal use of all these resources.

In this technical white paper, we explain how Oracle's heavily threaded SPARC T5 and M5 servers can be deployed to efficiently consolidate and manage workloads using virtualization through Oracle Solaris Zones, Oracle VM Server for SPARC, and Oracle Enterprise Manager Ops Center, as well as how to improve the performance of a single application through multithreading.

Introduction

Recently, Oracle raised the bar by introducing the SPARC T5 and M5 servers. With up to 1,024 and 1,536 threads respectively, these systems provide unprecedented processing power within a single system.

Although in some cases customers will use all of these resources to run a single workload or perhaps even one single application, we expect that in some situations these servers will be used to consolidate multiple application workloads. Some, or even all, of these applications may be multithreaded.

Oracle provides several solutions to address both needs. That is the topic of this paper.

The first part gives a brief overview of important concepts related to programming for multithreaded systems; the next part covers the SPARC S3 core from Oracle, which supports 8 hardware threads and is at the heart of all Oracle's SPARC T4, T5, and M5 systems.

Next, we show how the SPARC S3 core is used to build shared memory servers through a highly optimized cache coherent interconnect that scales with the size of the system. The topologies of the SPARC T4-4, T5-8, and M5-32 servers are also shown and discussed.

The performance of a single application can be improved through multithreading. This topic is quite important and is covered quite extensively, with emphasis on using the Oracle Solaris Studio compiler and analysis tools. We use two different parallel programming models to illustrate how this can be achieved. After an overview of the OpenMP parallel programming model, the POSIX Threads way to parallelize an application is shown.

The goal of multithreading is performance and scalability, but this is easier said than done. This is why we give recommendations on how to write scalable applications. Following these rules is a big step towards good parallel performance.

As mentioned above, these new systems have so much capacity that customers should consider using one such system to consolidate and manage workloads.

Oracle offers a variety of solutions to match the specific consolidation needs. Oracle Solaris Zones and Oracle VM Server for SPARC are two different virtualization technologies that address this. Oracle Enterprise Manager Ops Center provides an integrated and cost-effective solution for complete physical and virtual server lifecycle management. All of these are discussed in detail.

One other important topic that is sometimes overlooked by performance analysts is the impact of the operating system. A serious and ongoing engineering effort is required to support these very heavily threaded systems. This is the topic of the last section.

The white paper concludes with a glossary that explains the terminology used.

Exploiting Machine Parallelism

The era of massively multithreaded servers is here. This is due to modern processor technology's success at tracking Moore's Law [1]. Oracle's recently introduced Oracle Solaris-based servers with the SPARC T5 and M5 processors [2, 3]—which are based on the highly successful SPARC S3 core first seen in the SPARC T4 processor—feature scalability exceeding 1,000 high-performance hardware threads (HWTs) for processing. This trend—ever more cores per processor and more processors per server—is expected to continue. The challenge now is finding methods to exploit the increasing power of these systems. With this embarrassment of riches, SPARC customers will welcome some guidance in efficiently using all those HWTs. This document provides that guidance.

Thinking in Parallel and the Advantages of Parallelism

To speed up the execution of any program and to improve overall system performance, the key issue is identifying independent software components and executing them "in parallel," that is, *simultaneously*, on multiple resources. In nearly all areas of a computer system, exploiting parallelism is essential. Not just at the processor level but also I/O and data communication benefit from overlapping and simultaneous activities. Therefore, both application developers and IT architects have a need to understand where parallelism can be used to enhance the performance, scalability, and reliability of their applications and system configurations.

While this might seem obvious, history shows that both programmers *and* system architects often overlook the potential to exploit parallelism in all its forms. Scalable parallel processing must be used in order to realize the potential of cloud computing and big data implementations.

Additionally, with so much processing power concentrated in modern multiprocessor multicore servers, these systems provide a compelling platform for server consolidation through the use of virtualization technologies to provide deep savings in financial cost, server count, floor space, software licensing, electricity, cooling, and maintenance effort. In particular, Oracle's latest SPARC servers can be easily subdivided into many hypervisor-based VM domains, and can also use Oracle Solaris' built-in OS virtualization technology—Oracle Solaris Zones.

Oracle's Approach: Products and Technologies

Oracle's SPARC processors, operating systems, application software, engineered systems (Oracle Exadata, Oracle Exalogic, SPARC SuperCluster, Big Data Appliance, and others) [4], and optimized solutions [5], are all prime examples of Oracle's adoption and integration of parallelization concepts throughout its entire product line.

Just a few examples include

- The SPARC S3 multicore processor design supporting 8 threads per core
- SPARC T4, T5, and M5 shared memory servers with up to 1,536 hardware threads
- Oracle Real Application Clusters (Oracle RAC)
- The extensively multithreaded Oracle Solaris operating system kernel
- Oracle Solaris 11 Integrated Load Balancer

- ZFS (RAID-Z, separate disk for ZIL)
- SPARC and x86 virtualization
- Oracle Solaris Studio compilers and tools with full support for multithreading
- The DTrace [6, 7] system profiling tool with full support for multithreading

These products and technologies incorporate multiple methods for increasing overall system performance, including

- Improving Oracle Solaris' use of parallelism and caching
- Providing multithreading APIs and optimizing compilers for application developers
- Accelerating processors through intelligent caching, faster clock rates, and instruction parallelism
- Providing tools that discover opportunities for parallelism, such as DTrace [6, 7] and Oracle Solaris Studio [8]

This document helps developers, system architects, performance analysts, and system administrators recognize opportunities to exploit parallelism and it explains how Oracle's extensively threaded systems can be configured for server consolidation.

It does so by providing information on various underlying parallelism concepts, and it also covers how Oracle's products and technologies can be used to address a variety of use cases. There is a special emphasis on the recently introduced SPARC multicore servers.

The SPARC T4, T5, and M5 Shared Memory Systems

In this section we give an overview of the SPARC S3 core as well as the SPARC T4, T5, and M5 systems. All these servers use the same SPARC S3 core as a basic building block, ensuring binary compatibility throughout the system family and with earlier SPARC systems. It should be noted that only a small selection of key features is highlighted here, such as out-of-order execution and instruction level parallelism. For much more information on the SPARC S3 core and the SPARC servers it is used in, refer to [9, 10, 11, 12].

Out-of-Order Execution

In a conventional "in order" processor design, instructions are fetched from an instruction cache, decoded and executed in exactly the same order as they appear in the code. The downside of this approach is that an "expensive" instruction such as a divide, which takes many clock cycles to complete, blocks the execution of instructions following, even if they are independent. This easily leads to delays, or "stalls," in the execution flow that could be avoided if there were a way to bypass the expensive instruction with instructions that do not depend on it.

That is exactly what an out-of-order (OOO) architecture does. With OOO execution, the processor allows instructions issued later to execute before instructions issued earlier have completed. This is allowed because the resulting value will be correct, as long as the instructions are independent. With OOO execution, such runtime dependencies are automatically tracked and transparently handled in the processor.

We illustrate this with a simple example. Below we list a floating point instruction sequence using pseudo instructions. We use `r<n>` to denote register <n>. The instructions below are executed from top to bottom, starting with the first instruction.

```
fadd    r1,r2,r3      # r3 = r1+r2
fsub    r1,r2,r4      # r4 = r1-r2
load    @a,r1         # load the value at memory address "a" into register r1
load    @b,r2         # load the value at memory address "b" into register r2
fdiv    r3,r4,r5      # r5 = r3/r4 = (r1+r2)/(r1-r2)

fmul    r1,r2,r6      # r6 = r1*r2
```

The above instruction schedule is non-optimal because independent instructions block each other. The instructions causing these stalls are in this case also (potentially) expensive.

On an in-order architecture, the `fdiv` instruction performing the division must wait for the two memory load instructions to finish, even though it does not need those values. Even if this data is cached nearby, it still takes a few cycles to get it. This results in a pipeline stall. Although its input values do not depend on it, the `fmul` instruction still needs to wait for the long latency `fdiv` instruction to finish. Note that the delays are introduced because of the instruction schedule, not because of true dependencies.

Also, the use of `r1` and `r2` in the load instructions creates an artificial dependence on the `fadd` and `fsub` instruction, since they also use `r1` and `r2`. This duplicate use is not needed, however. Without affecting correctness, the results of the load instructions could go into `r7` and `r8`. This is, therefore, a correct, but non-optimal schedule and has stalls that could be reduced or avoided.

On an OOO architecture, the execution pattern is more dynamic and eliminates these two drawbacks. The `fdiv` instruction only needs to wait for the `fadd` and `fsub` instructions to finish. Likewise, the `fmul` instruction has no dependency on the `fdiv` instruction and starts as soon as the results from the two memory loads are available.

Note that another feature is needed when implementing OOO execution. In order for the two memory load instructions to be issued before the `fadd` and `fsub` instructions have completed we need to consider the multiple use of registers `r1` and `r2`. As just mentioned, this re-use is not needed for correctness and should not slow down the execution. The "register renaming" part of the processor handles this situation.

OOO execution not only improves the performance of many applications, but it also makes the performance smoother. It is less dependent on the precise details of the order in which instructions are scheduled and therefore more tolerant regarding non-optimal instruction scheduling. Another advantage is that instructions with a variable latency, such as a memory load instruction that may or may not miss in cache, can be overlapped with unrelated work.

Thanks to OOO execution these dependencies are handled in hardware and the instruction stalls are reduced or even eliminated, depending on the number of cycles needed by these instructions.

The SPARC S3 Core

The SPARC S3 core [9, 10, 11, 12] implements OOO execution. It is the first SPARC processor to implement this feature and is used in in the SPARC T4, T5, and M5 processors. Figure 1 shows a block diagram of this core.

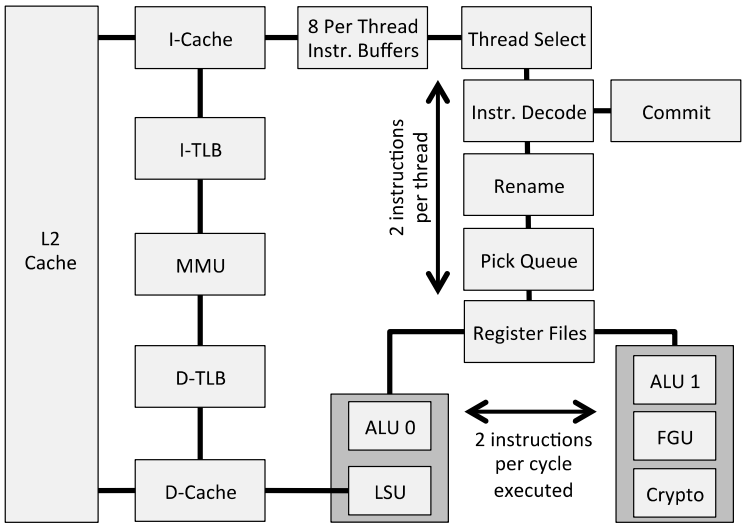

Figure 1. Block diagram of the SPARC S3 core.

Please note this is the core only. The SPARC T4, T5, and M5 processors not only include multiple cores, they also have additional on-chip functionality like the memory controller, I/O, coherence interfaces, etc.

In the SPARC S3 core, OOO execution is combined with instruction level parallelism (ILP). Transparent to the application, the hardware issues up to *two* instructions to the various execution pipelines in the core. For example, two integer add instructions, or an integer add and a memory load instruction, can be issued simultaneously.

This mechanism is also referred to as "2-way superscalar." At first sight it seems tempting to allow many more instructions to be issued in parallel, but it has been found that many real world applications only exhibit a modest degree of instruction level parallelism. Supporting the ability to issue two instructions per clock cycle has proven to accelerate single thread performance on a wide variety of applications.

Other Processor Features

Another important performance feature is the very efficient use of eight hardware threads (HWTs, sometimes called "strands") within the SPARC S3 core.

The basic idea of hardware threads is to avoid wasting processor cycles in case a running application experiences a delay in the pipeline. This delay could be a long latency instruction or a cache miss. In a more conventional design, all such events cause the core to waste cycles because execution cannot proceed until the delay has been resolved.

In the SPARC S3 core, idle cycles are automatically used by other hardware threads. This is transparent to the application, to Solaris, and to the user.

This thread selection mechanism is very fine grained and is performed by the hardware. Each processor cycle, a selection of one out of eight candidate threads is made. Only threads that are ready to run are considered. The selection is based upon a modified least recently used algorithm that ensures that each thread gets a fair share of the pipeline(s) and avoids thread starvation.

Great care has been taken to make this very efficient. For example, each thread has its own instruction buffer. The instructions for the selected thread are fetched from this buffer and sent to the OOO part of the core.

The implementation of hardware threading in the SPARC S3 core provides an efficient, adaptive and fine-grained usage of the available cycles with the goal of improving system throughput and parallel application performance.

The combination of these features results in a very powerful and flexible core. In the next section we shall see how this core is used as a fundamental building block in SPARC T4, T5, and M5 shared memory systems.

SPARC T4, T5, and M5 Systems: Connecting the Dots

As we have just seen, at the core level there are already two levels of parallelism. At the lowest level, the 2-way superscalar feature issues up to two instructions at the same time. This is combined with the hardware support for eight hardware threads per core.

In this section, we see how specific features of the SPARC T4, T5, and M5 systems are used to add a *third* level of parallelism.

As mentioned earlier, these systems all use the same SPARC S3 core with eight hardware threads per core. However, there are differences between the SPARC T4, T5 and M5 *processors* regarding core clock speed, the number of cores within one processor, and the size of the L3 cache[1]. These processors are used in SPARC T4, T5, and M5 systems. The name of the system is based on the processor name appended with the number of processors/sockets supported. For example, a SPARC T5-4 system has four SPARC T5 processors.

[1] There are some other low level differences that are beyond the scope of this paper.

Table 1 gives an overview of the various processor features.

TABLE 1. VARIOUS PROCESSOR LEVEL FEATURES OF SPARC T4, T5 AND M5 SYSTEMS

SYSTEM	MAX NUMBER OF PROCESSORS	NUMBER OF CORES/PROCESSOR	HWTS/CORE	TOTAL HWTS	CLOCK SPEED (MHZ)	L3 CACHE (MB)
SPARC T4	4	8	8	256	2998	4
SPARC T5	8	16	8	1024	3600	8
SPARC M5	32	6	8	1536	3600	48

Since the same core is used in all three systems, any optimization performed at the core level benefits all three systems.

All of the systems use a cache coherent network that connects all of the processors, memory and I/O in the system. The difference lies in the topology of the network but, as we shall see shortly, the details of the network will not impact application development and performance tuning.

The SPARC T4 and T5 systems use a single hop directly connected network. Diagrams of the interconnect networks used in the SPARC T4-4 and T5-8 are shown in Figures 2 and 3 respectively.

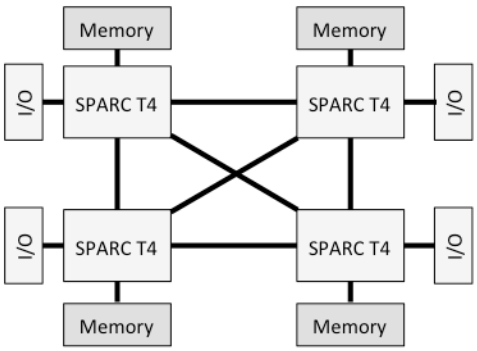

Figure 2. The interconnect network of the SPARC T4-4 system.

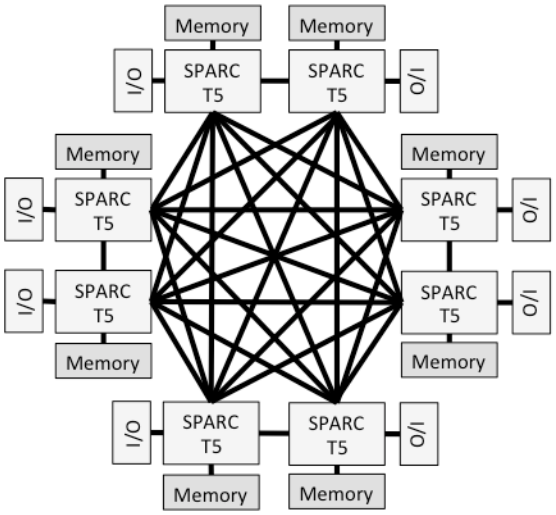

Figure 3. The interconnect network of the SPARC T5-8 system.

As seen in Figures 2 and 3, processors, memory and I/O are physically distributed across the server, but thanks to the cache coherence circuitry, the server has a single system image architecture. In other words, it is not a cluster. There is one address space for the entire system and each memory location and I/O device is accessible from anywhere in the system.

This is a very advantageous property because unlike a cluster, resources like memory and I/O are not fragmented. An application has no restrictions regarding the usage of these resources. For example, a single application can use all of the shared memory, even when using one core only, while a parallel application can as easily use all memory and all cores. The same transparency holds for I/O.

This kind of architecture also provides scalable memory bandwidth. Each processor connects to a portion of the shared memory and has its own direct connection to that memory. Therefore, adding processors also adds memory bandwidth. This is a huge improvement over older bus based designs with a fixed bandwidth. In such a case, adding processors reduces the memory bandwidth available to each processor.

Each core on any processor has transparent access to all of the memory connected to other processors. The only difference is that access to "remote" memory takes longer compared to accessing the "local" memory connected to the processor on which the core physically resides.

This difference in memory access time is called NUMA (Non-Uniform Memory Access). Since the system also supports cache coherence, it is, therefore, said to have a cc-NUMA architecture.

Similar to the SPARC T5 memory architecture, the interconnect network of the SPARC M5-32 system is shown in Figure 4.

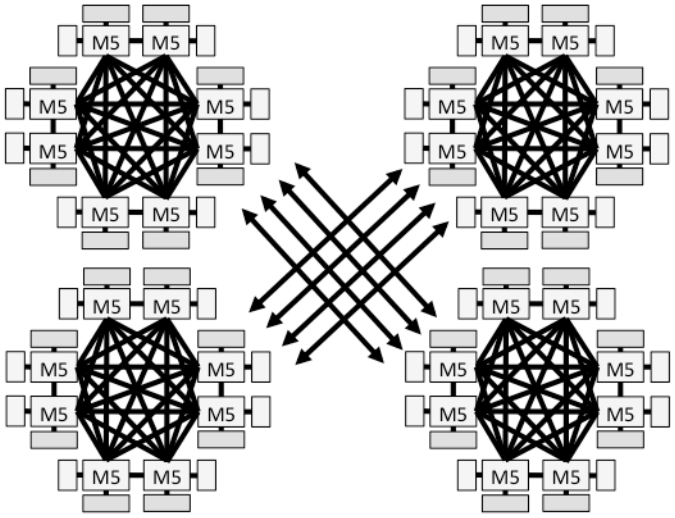

Figure 4. The interconnect network of the SPARC M5-32 system.

From an application development and deployment point of view a cc-NUMA architecture is very easy to use. There are no restrictions on the choice of cores to use, nor on the memory access. An application can transparently access any memory location to read or write data.

In addition to the considerations discussed so far, the section "Optimizing for Data Locality" describes the ability to use local memory preferentially, either automatically or programmatically.

Developing Applications for Multicore Systems

In this section we cover how to take advantage of the shared memory parallelism that multicore based hardware offers. All SPARC T4, T5, and M5 based servers are included in this and everything covered in this section is therefore immediately applicable to these systems.

Finding the Parallelism

Before diving into more details on the "how," we would like to spend some time on the "where."

Unfortunately there is no silver bullet when it comes to identifying the parallelism in an application. This is where knowledge of the application and its functionality are a big help.[2]

Having said that, what can one do then?

[2] The Oracle Solaris Studio compilers support automatic parallelization through the `-xautopar` option the compiler performs a dependence analysis at the loop level. If the work associated with the loop iterations is found to be independent, the compiler automatically generates the parallel code for the user.

In the ideal case one has the luxury of developing a new application. If so, we highly recommend considering parallelization from the start. This need not be implemented yet, but one could write the code such that it is easier to parallelize later on.

In the case of a newly developed program it is necessary to identify opportunities for parallelization. Whether it is an on-line banking system, a scientific simulation, an e-commerce application, a device driver, or anything else, it is important to understand the level of parallelism.

Ideally, this understanding is upfront and could drive some software design choices in order to facilitate the parallelization either directly, or at a later stage.

For example, if there is parallelism at a fairly high level in the application one may consider implementing the independent parts that a function calls. Unless these functions are extremely simple, one need not worry about the overhead such a function call introduces. The gain is noteworthy, because one can simply execute these functions in parallel. When using OpenMP, for example, this requires only n+1 pragmas for "n" functions to be executed simultaneously.

Another design choice could be to carefully decide where to use global data. In general it is a good performance principle to use localized data as much as possible and refrain from global data unless really needed, but in a parallel application global data that is modified in general requires extra care.

In many cases, however, one does not have the luxury of writing an application from scratch. There is no reason to worry though. It is possible to develop an efficient parallel application in this case as well.

The first key step is to use a profiling tool like the Oracle Solaris Studio Performance Analyzer to find where the execution time is spent. Generally this is an eye opener because more often than not this turns out to be in a part of the application that is not expected to be expensive.

Once it is known where most of the time is spent, the key question to answer is whether the work performed in this part has a certain level of independence. If so, there is an opportunity for parallel execution.

For example, maybe there are several function calls that can be executed in any order. If so, and if they have no side effects like modifying a shared data structure without the proper locking, they can also be executed simultaneously. Perhaps there is a `for` loop and the work performed across the various iterations is independent. If that is the case, the loop can be executed in parallel.

Of course one needs to be careful there are no gotchas somewhere. Unless the target code fragment is quite straightforward, more analysis may be needed and knowledge of the application can be a big help then.

Once the most time consuming part has been parallelized, the second most expensive part needs to be tackled in a similar way. In this way one works through the performance profile, gradually parallelizing the application.

Don't stop too soon with this process, and always remember Amdahl's Law [18].

Parallel Programming Models for Shared Memory

There are many shared memory programming models available to express parallelism in an application that targets a multicore system.[3]

Traditionally, developers have used explicit threading models like POSIX Threads (or Pthreads for short) [13] in C/C++ and Java Threads for Java applications. Such models provide a relatively modest set of powerful, yet fairly low level functions to implement parallelism in a shared memory environment.

Given the conceptual similarities between these models and the scope of this paper we have chosen to only cover Pthreads as an example of an explicit threading model, but many of the topics discussed here carry over to other such models.

On the other side of this spectrum is OpenMP [14, 15]. At a high level, the developer uses *pragmas* to express the parallelism in the application. The compiler then translates this into the appropriate underlying infrastructure to create a parallel application.

Both OpenMP and Pthreads are covered in the remainder of this section.

The OpenMP Parallel Programming Model

The first OpenMP 1.0 specification was released in 1997. Since then it has evolved significantly, and the 4.0 specifications that were released in July 2013 are another major step forward.

So what is OpenMP then?

It is a shared memory parallel programming model available for C, C++ and Fortran. The user specifies the parallelism by adding directives in the source. In C/C++ these are pragmas, while in Fortran specific language comment syntax is used. In both cases the advantage is portability. A compiler that does not support OpenMP simply ignores the pragmas and special comment lines.

A simple example of a parallel version of "Hello World" is shown below. The source code changes specific to OpenMP have been color coded blue.

[3] We refer to a program as being "parallel" or "multithreaded" if there are multiple independent instruction streams that could execute simultaneously. These streams are also loosely defined to be a "thread."

```
#include <stdio.h>                                                    (1)
#include <stdlib.h>                                                   (2)

#ifdef _OPENMP                                                        (3)
#include <omp.h>                                                      (4)
#endif                                                                (5)

int main()                                                            (6)
{                                                                     (7)
 #pragma omp parallel                                                 (8)
 {                                                                    (9)
  printf ("Hello OpenMP World, I am thread %d\n",omp_get_thread_num()); (10)
 } // End of parallel region                                         (11)
}                                                                     (12)
```

As simple as it is, this code already has several of the key features to be found in every OpenMP application.

The first new element can be found at lines 3–5. Here we include an OpenMP specific file (`omp.h`) that defines all the OpenMP runtime functions. If no such functions are used this file need not be included. Note that the inclusion is under the condition that `_OPENMP` is defined. This is guaranteed to be the case if an application is compiled for OpenMP. In this way, one can also compile the program without enabling OpenMP because then the file will not be included.

Lines 8–11 form the core part of the parallel code. They define what is called a parallel region. This is the key construct in OpenMP. It is defined by the `#pragma omp parallel` keywords. All threads execute the code in the parallel region. Outside parallel regions the main "master" thread executes the serial portions of the code.

In this case the parallel region starts with the opening curly brace at line 9 and ends with the closing curly brace at line 11. Although not required, we prefer to mark it with a comment string to underline the end of the parallelism.

Since the `printf` statement at line 10 is part of the parallel region, each thread executes it. To distinguish who prints the line we include the unique thread number in the output. This value is returned by the `omp_get_thread_num()` runtime function.

Now that we have our parallel code we need to compile it. This is easily achieved by using a specific compiler option, which is `–xopenmp` for the Oracle Solaris Studio C, C++ and Fortran compilers. Assuming we stored our program in file `hello.omp.c` we simply compile it as follows:

```
$ cc -o hello -fast -xopenmp hello.omp.c
```

The `–fast` option activates many advanced serial optimizations. As just mentioned, the `-xopenmp` option activates OpenMP support in the Oracle Solaris Studio compilers.

It is worth spending a little bit of time on what actually happens under the hood of the compiler. Thanks to the `–xopenmp` option, the compiler recognizes the pragmas (or directives in Fortran) with the appropriate syntax, which is `#pragma omp <keyword(s)>` in C/C++.

A simple pragma as shown on line 8 translates to a quite complicated library call structure to generate the code that executes in parallel at runtime. All thread creation, management and execution is handled transparently to the user. All the developer needs to do is to use the pragmas to tell the compiler where the parallelism is and how to distribute the work.

As we will see later, OpenMP provides a much richer functionality than just this, but the general philosophy remains. The level of abstraction is very high, relieving the developer from worrying about all sorts of low level details.

The program we just compiled and linked can now be run as usual. The only difference is that we use an OpenMP specific environment variable called `OMP_NUM_THREADS` to inform the runtime system how many threads we like to use. If this variable is not set, a system dependent value is used.

Here's how to run this code using 2 and 4 threads:

```
$ export OMP_NUM_THREADS=2                          (1)
$ ./hello                                           (2)
Hello OpenMP World, I am thread 0                   (3)
Hello OpenMP World, I am thread 1                   (4)
$ export OMP_NUM_THREADS=4                          (5)
$ ./hello                                           (6)
Hello OpenMP World, I am thread 0                   (7)
Hello OpenMP World, I am thread 3                   (8)
Hello OpenMP World, I am thread 2                   (9)
Hello OpenMP World, I am thread 1                   (10)
$
```

At line 1 we set the number of threads to 2 and run the program as usual (line 2). Lines 3 and 4 show the output for each thread. In OpenMP it is guaranteed that the main "master" thread has a thread ID of zero. The other threads are integers, numbered consecutively from one.

At line 5 we change the number of threads to be 4 and run the same program in exactly the same way, but now four lines are printed, one for each thread (lines 7–10).

Note that the output is not sorted by thread ID. This is an aspect of parallel computing. Various threads execute at different times and most likely not in the same order across runs. If a code block is truly parallel this should not affect correctness of the results.

There is one thing we have not mentioned yet. In any shared memory parallel program there is not only a need to have data shared by all threads, but each thread needs private data too. Private data is only visible to the thread that owns it and cannot be read or written by other threads. The loop counter in a `for` loop is an example of this. It would be really bad if one thread can modify the loop variable used by another thread.

In OpenMP, the `shared` and `private` clauses are used to label data accordingly.

Additionally, in C/C++ any variables local to a code block are automatically privatized. A simple example of this is a re-written version of our example program:

```
#pragma omp parallel                                                (1)
 {                                                                  (2)
  int thread_id = omp_get_thread_num();                             (3)
  printf ("Hello OpenMP World, I am thread %d\n",thread_id);        (4)
 } // End of parallel region                                       (5)
```

Variable `thread_id` is declared locally at line 3 and, therefore, is automatically a private variable in OpenMP.

Now that we have seen a first and admittedly simple example of OpenMP, it is time to look at the bigger picture and cover the main functionality available. There are three major components and we have already seen an example of each.

- **Directives** – The fundamental concept is the parallel region. An application can have an arbitrary number of these. Within each region the developer can select how to distribute the work over the threads.

  For example by using the `#pragma omp for` construct the iterations of the loop to which it applies are executed in parallel where each thread operates on a subset of all iterations. However, there is much more.

  Through parallel sections (`#pragma omp sections`), multiple independent code blocks can be identified. These blocks can contain any type of code and are executed in parallel. This for example can be used to set up a pipeline overlapping I/O and processing.

  As of OpenMP 3.0, tasks have been introduced. Through the `#pragma omp task` construct one identifies tasks in the source. The assumption is that code blocks identified as tasks can execute independently. The compiler and runtime system handle the creation and parallel execution of the tasks. Tasks are ideally suited for more dynamic and irregular types of parallelism.

- **Runtime functions** – An extensive set of runtime functions are available to query and modify the runtime environment. The `omp_get_thread_num()` function we saw earlier is an example. It returns the thread ID. There are many more functions though, providing great flexibility to adapt or change the execution to the specific situation.

- **Environment variables** – Several environment variables are supported. These can be used to define the initial environment. We already saw one. Variable `OMP_NUM_THREADS` can be used to set the initial number of threads to be used in a parallel region. All environment variables take a default. Unless one prefers to change a setting, there is no need to set them.

Aside from the high level of abstraction and ease of use, OpenMP has another major advantage over a programming model that relies on explicit library calls. It is possible to write the parallel application such that the original, serial version is still "built in" and can be used by not compiling with the option to invoke OpenMP (e.g. `–xopenmp` on the Oracle Solaris Studio compilers).

This is a good safety net that can be used in the development phase, as well as when encountering a last minute regression. The code will still run serially if the `–xopenmp` option is omitted. Those parts not compiled with `–xopenmp` obviously execute serially, but at least the application will run.

Another important feature of OpenMP is that the specifications continue to adapt to the rapid changes in the architecture of microprocessors and computer systems. A case in point is the release of OpenMP 4.0 in July 2013. Major new functionality added includes support for heterogeneous systems (e.g. attached accelerators), cc-NUMA architectures, thread cancellation and an extension of the tasking model.

In this section we only touched upon OpenMP. For much more in-depth information we refer to [16]. The *Oracle Solaris Studio OpenMP User's Guide* can be found at [17].

POSIX Threads

For many years, POSIX Threads (also referred to as Pthreads) provided the only portable, standardized parallel programming model for shared memory systems.

Thanks to its portability and flexibility it has become a very popular way to parallelize applications.

It is an explicit programming model. The developer has to handle all of the details. The Pthreads API provides functions that can be used to create and manage threads, synchronize threads, and so on, but there is no higher-level functionality.

A question that arises often is whether one should use OpenMP or Pthreads.

While OpenMP provides an easy to use and flexible way to develop parallel programs, control over the threads and their behavior is not always explicit. This is by design and sufficient for many, but sometimes a developer wants to control the threading model in a very detailed manner. In this case, Pthreads provide a natural and portable solution.

The Pthreads API contains a large number of functions and features—too many to discuss in this paper. Instead, the remainder of this section will cover the core subset of thread creation and mutual exclusion.

The Pthreads way to create a new thread is to call function `pthread_create()`. The key parameter it takes is a pointer to the routine that the new thread will execute. When a thread completes its work, it waits to be "joined" so that it can pass back a return value.

The API call `pthread_join()` waits for a particular thread to complete, and makes the return value from the child thread available to the calling thread. The code below shows a thread being created, executing, and returning a value. The Pthread specific code fragments are color coded blue.

```
#include <pthread.h>
#include <stdio.h>

void * work(void * param)
{
   printf("In child thread\n");
   return (void*)8;
}

int main()
{
   pthread_t childthread;
   pthread_create(&childthread, 0, work, 0);

   printf("In parent thread\n");

   long value;
   pthread_join(childthread,(void*)&value);

   printf("Child returned %i\n",value);
}
```

A disadvantage of using Pthreads is that the developer is responsible for managing all aspects of the interactions between the various threads. For example, if there are two threads that manipulate the same variable, then one needs to ensure that these two threads cannot update it at the same time. The easiest (but also least efficient) way of doing this is to place the updates to the variable under the control of a mutually exclusive lock, or "mutex" (lock).

This mutex lock can only be acquired by a single thread at a time. To update the shared variable, a thread needs to acquire the mutex lock, update the variable, and then release the mutex lock. The API provides for the functionality to acquire and release the lock. It is guaranteed that only one thread at a time can do so.

A mutex lock has a lifecycle. It needs to be initialized before it can be used, and destroyed once it is no longer needed. The example below demonstrates how two threads increment a shared variable using a mutex lock. Pthreads specific code fragments are color coded in blue.

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t mutex;
volatile int value=0;

void *work(void * param)
{
   for(int i=0;i<200; i++)
   {
      pthread_mutex_lock(&mutex);
         value++;
      pthread_mutex_unlock(&mutex);
   }
}

int main()
{
   pthread_t childthread[2];

   pthread_mutex_init(&mutex,0);

   for (int i=0;i<2; i++)
   {
      pthread_create(&childthread[i], 0, &work, 0);
   }
   for (int i=0;i<2; i++)
   {
      pthread_join(childthread[i],0);
   }

   printf("Value of shared variable = %i\n",value);

   pthread_mutex_destroy(&mutex);
}
```

In the example, the threads increment a shared variable. This type of update occurs in quite a few situations. For example, if statistics are being kept for the amount of work completed, or the time taken to complete the work.

Rather than use a mutex lock for these situations, it may be more efficient to use atomic operations. These are typically very efficient, but non-portable. On Oracle Solaris they are declared in the `atomic.h` header file. So the previous example can be recoded to use atomics as shown next. The code specific to the use of the atomic operation(s) is color coded in blue.

```
#include <pthread.h>
#include <stdio.h>
#include <atomic.h>

volatile unsigned int value=0;

void *work(void * param)
{
   for(int i=0;i<200; i++)
   {
      atomic_add_int(&value,1);
   }
}

int main()
{
   pthread_t childthread[2];
   for (int i=0;i<2; i++)
   {
      pthread_create(&childthread[i], 0, &work, 0);
   }
   for (int i=0;i<2; i++)
   {
      pthread_join(childthread[i],0);
   }
   printf("Value of shared variable = %i\n",value);
}
```

In this example, we have declared the variable `value` to be *volatile*. This tells the compiler that the variable should not be cached in a register. Instead, it should be fetched from memory whenever it is used, and immediately stored back to memory after it has been modified. The volatile keyword is not strictly necessary in this situation—the fact that we are making function calls in the loop will ensure that the variable is stored back to memory before the mutex lock is released, but in other situations it may be necessary for shared variables to be declared this way.

Consider the code below:

```
#include <pthread.h>
#include <stdio.h>

int ready = 0;

void *work(void * param)
{
   while (!ready) {}
}

int main()
{
   pthread_t childthread;

   printf("Started\n");

   pthread_create(&childthread, 0, &work, 0);
   printf("Continue\n");

   ready=1;

   pthread_join(childthread,0);

   printf("Finished\n");
}
```

The thread executing the routine `work()` waits for variable `ready` to become non-zero. Since variable `ready` is not declared as volatile, the compiler is free to optimize the loop so that the variable is only read once.

This is a perfectly valid optimization, but in a parallel program it has an unwanted and unpleasant side effect.

If the variable is read before it is set to one, then the child thread will never read it again, and so will loop forever. If the variable is set to one by the time the child thread starts, it will immediately return.

The difficulty with the code is that most of the time the variable `ready` is set correctly, and the code behaves as expected; however, on rare occasions, if the variable is not set at the "right" time, the program will get stuck in an infinite loop. It is this kind of unpredictability that makes multithreaded applications hard to debug.[4]

[4] This is an area where OpenMP has more safety nets. The "flush" concept provides a well-defined and portable way to force consistency of variable values throughout the system. Since the flush construct is built-in with all major constructs (e.g. the barrier), there is usually no need to declare shared variables as volatile.

Other Synchronization Mechanisms

In addition to mutex and atomic operations, there are a number of other mechanisms for coordinating between threads.

The readers-writer lock is a variant of a mutex where multiple reader threads can hold the lock and read the protected variables, or one writer can hold the lock to modify the protected variables.

Semaphores are counters where a "post" increments the counter, and a "wait" waits until the counter is greater than zero. This mechanism can be used to make a thread wait until there is work ready for it.

Condition variables are a mechanism similar to semaphores in that they allow a thread to signal other threads, and the other threads wait until they are signaled. However, condition variables also allow protected access to a shared variable. So it is easy to communicate data, as well as indicate that there is data available.

How to Develop Scalable Applications

Many misconceptions surround writing a well performing application, both for serial as well as multithreaded execution.

This is actually no surprise, because the hardware continues to evolve and compilers are increasingly getting more advanced. As a result, many of the guidelines that were valid until recently are no longer needed, or can even backfire, resulting in non-optimal code.

This is why in this section we prefer to focus on coding techniques with a long lasting performance benefit. In some cases, it may not only be challenging for compilers to fix these performance issues, but sometimes it is even fundamentally impossible to address them. This is especially true for data layout and cc-NUMA related performance bottlenecks.

Serial Performance

It seems strange to write about this in a paper on massively threaded systems, but that is exactly why we put it in here. All too often, developers take serial performance for granted and immediately jump to the parallelization. Although understandable, skipping serial performance may inhibit scalability and often sooner rather than later.

The reason is quite simple. If an application does not perform well using a single thread, what would you expect when that code runs in parallel? It would be rather naïve to assume things would then go very well.

Instead, scalability most likely suffers. This is because poor serial performance is almost always caused by non-optimal use of the cache(s) and memory system. This results in more memory transactions than necessary. Not only do these take extra time, the bandwidth (and sometimes the latency) requirements become stricter as well because more data than needed has to be fetched or written back.

In a parallel version of the same application this quickly puts additional strain on the bandwidth since multiple threads now use the interconnect at the same time. The more threads that are used, the more pronounced this problem will be and the bandwidth saturates sooner than needed.

This is why we highly recommend first ensuring that the serial application performs reasonably well. Very often this effort does not go beyond exploring some additional compiler options.

Modern compilers such as the Oracle Solaris Studio compilers can perform magic when tuning an application. The key to this lies in the ability for the compiler to determine the intent of the code. What do we mean by that? It is quite easy to write very cryptic code that is not only hard to read for humans, but also difficult for compilers to find out what is really going on. As a result, it may fail to generate optimal code.

Another reason compilers may find it hard to deliver the best performance is in the area of data structures. It is beyond the scope of this paper to elaborate on this, but we can summarize the problem here.

Any data structure (e.g. an array of structures) is stored in a certain way in memory. The best performance is obtained if data access smoothly follows the storage order in memory. This is when caches are used in an optimal way and techniques like data prefetch can hide cache and memory latencies very well.

If this is not the case, however, performance can quickly take a turn for the worse. This results in longer memory access times and increased bandwidth requirements.

A standard example is accessing an array. The code snippet below reads 2-dimensional array `a` one row after the other. This is the correct way for performance in C/C++ since the array is stored row by row in memory.

```
for (int i=0; i<nRows; i++)
  for (int j=0; j<nColumns; j++)
    sum += a[i][j];
```

If these two loops were reversed, the performance would be much worse. Especially for large matrices there will be many more data cache and data Translation Look-Aside Buffer (TLB) misses.

In general, an optimizing compiler will try to restructure the code to improve memory access, by interchanging the order of the loops in the above-shown nested loop, for example. It may, however, be limited in what it can do because it lacks the information needed to validate a specific optimization. For example, what if certain elements in an array of records are not used? This loads more data in the cache than needed, but fixing this requires re-arranging the storage of the data structure(s) in memory, perhaps by setting them up differently. Unless the compiler has full visibility of this throughout the entire code, it would stretch limits too far to expect the compiler to handle this.

This is where the developer can help by reconsidering the data layout and data structure definitions.

In summary, there are many opportunities to optimize serial performance of your code, and the compiler's options are a great place to start. Once the program performs sufficiently well serially, it is time to consider the parallel performance. This is the topic of the remainder of this section.

Amdahl's Law

There is a rule that can be used to predict the speedup of a parallel application. It has been named after Gene Amdahl, a famous computer architect, and is generally referred to as Amdahl's Law [18].

What this law basically says is that any serial, that is, non-parallel, part of a program dominates the performance sooner than later.

Expressed in a formula, this law gives the expected speed-up as a function of two parameters: `P` and `f`:

$$S_P(f) = 1/(1-f + f/P) \qquad \text{(Amdahl's Law)}$$

In this formula, $S_P(f)$ is the parallel speedup as a function of `P` and `f`, where `P` denotes the number of threads used, while `f` is the fraction of time spent in those part(s) of the application that can be parallelized. Obviously, `f` is in the interval `[0,1]` and both endpoints are corner cases. The case `f=0` denotes a purely sequential program, and no speedup due to parallelization is possible. A value of `f=1` means the entire application can be parallelized from start to end, and is, therefore, linearly scalable based on the number of available threads. This type of application is often referred to as "embarrassingly parallel."

In a more realistic scenario, the value of `f` is somewhere between zero and one, and hopefully closer to the latter!

There is an important thing we did not mention yet: Amdahl's Law does not include any parallel overhead. This is not a realistic assumption though and does imply that this law actually presents a best-case scenario.

As simple as this formula seems, its implications should not be ignored.

In Figure 5 we show the expected speedup as a function of various values for the fraction `f`. It is assumed up to 16 threads are used. Note that in this chart, `f` is expressed as a percentage.

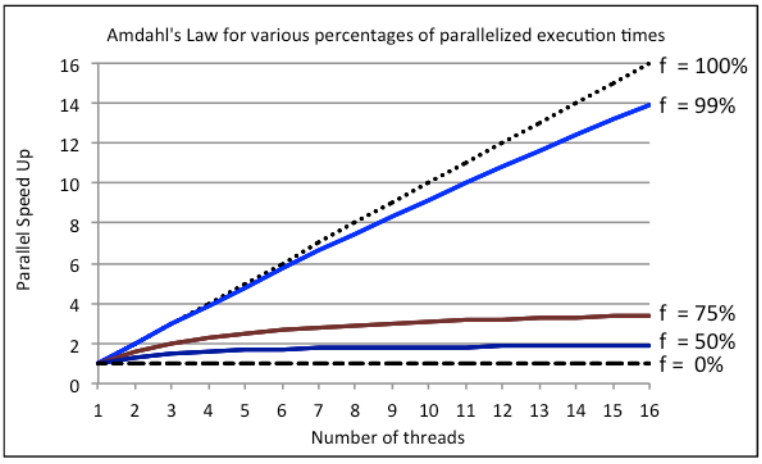

Figure 5. The expected parallel speedup as a function of various values for the fraction `f`.

The two dashed lines are the parallel speedup values for the corner cases `f = 0%` and `f = 100%`. As to be expected, the curve for `f = 0%` shows no speedup at all. The opposite situation, `f = 100%`, gives a perfectly linear speedup. As just mentioned it is, however, more realistic to assume f is neither of these, but somewhere in between.

The ruthlessness of Amdahl's Law is demonstrated through the 3 solid curves. As one can see, if 99% of the execution time in the serial application can be parallelized, the speedup using 16 threads is 14, not 16. Now this is not bad at all, but adding more threads quickly gives a diminishing return.

The other two curves are for `f = 75%` and `f = 50%`, respectively. As one can see, the speedup does not exceed 4 and 2 respectively, no matter how many threads are used.

This is a general property of Amdahl's Law. As we can derive from the formula, the parallel speedup is always less than `1/(1-f)`.

We conclude this section with a practical tip. By just using two timings, one can estimate the fraction `f` for any application.

Amdahl's Law can be rewritten to express `f` in terms of things we either know or can measure:

$f = (1-1/S_p)/(1-1/P)$

In this formula, we obviously know `P`, the number of threads used, and we can measure the parallel speedup.

For example, assume the single thread time of a job is 100 seconds. Also assume that when using two threads we measure 57 seconds of elapsed time. Applying the formula above gives:

```
f = (1-1/(100/57))/(1-1/2) = 0.86 = 86%.
```

We can now use this value for `f` and apply Amdahl's Law. According to this, the estimated speedup on 4 threads is $S_4(0.86) = 1/(1-0.86 + 0.86/4) = 2.82$.

In a similar way, we estimate the speedup on 8 threads to be 4.04. Although still faster than the run using 4 threads, there is a diminishing return.

Amdahl's Law should not be used as a reason to not parallelize an application at all. A growing group of developers achieves great results going down the path of parallel computing. What Amdahl's Law says, is that when aiming for a very scalable application, one needs to be aware of any serial part in the application and find ways to parallelize it. Very often this is possible and very rewarding.

In that respect a classical "gotcha" are the synchronization primitives, like locks and barriers. Although part of the parallel infrastructure, these are serial operations at a certain point deep down inside the system. To make matters worse, the cost of these primitives grows as more threads are added. Therefore, they must be used with great care, as described next.

Synchronization

As just discussed, in order for an application to scale well it needs to spend little time in the serial, single-threaded part of the code.

Synchronization typically implies serial execution. For example, suppose all the threads complete some work, then need to update a common state that is protected by a mutex lock, before all of them can resume their work.

Only one thread at a time executes the code protected through the mutex lock. All other threads that have not executed this code yet need to wait, but that means execution is serialized: one thread executes after another, not simultaneously.

The more threads that require synchronization, the longer the synchronization takes, and the larger proportion of the runtime is spent synchronizing.

To produce codes that scale, the threads should spend as little time as possible synchronizing, and as much time as possible doing independent work.

There are multiple degrees of intrusiveness of synchronization, and they can be classified as follows:

- **No synchronization.** Every thread completes its own work without interactions with other threads. This is the ideal approach because the resulting application should be able to scale to large numbers of threads with no synchronization costs.
- **Local copies of shared data.** In some instances there is some shared state to maintain, but each thread can update their own copy of this data, and the final value can be computed only when it is necessary. For example, to calculate the total value of an array of numbers, each thread can have a local variable that holds the partial sum resulting from adding up a segment of the array. Once all threads have completed their work, these partial sums can be combined into a single value. Although the act of combining the partial sums is quite expensive, it is less expensive than attempting to hold the resulting sum in a single variable and protect each update through a lock.
- **Lightweight synchronization.** Some synchronization primitives, like atomic operations, are lower cost, but obviously still more expensive than *not* synchronizing.
- **Pthreads synchronization primitives.** The synchronization primitives provided by the Pthreads API are portable, but heavier weight than atomic operations. They often require calls into the operating system kernel, and sometimes cause a thread to sleep rather than loop while waiting for an event. Waking up the thread again is relatively expensive.
- **"Stop the world synchronization."** The worst kind of synchronization is the type where all the threads stop working until they all have completed a lock-protected block of code. This is very inefficient because not only is execution serialized (Amdahl's Law), the cost of the synchronization increases as threads are added.

One advantage of using higher-level parallelization approaches like OpenMP is that the support libraries provide efficient algorithms for solving common data sharing problems.

For example, in OpenMP the above-mentioned summation of the elements of an array can be simply handled through the *reduction* clause. The compiler then handles this is in the most efficient way, taking advantage of local storage to avoid excessive synchronization.

Optimizing for Data Locality

All contemporary parallel computer systems have a cc-NUMA architecture. The reason is simple. To provide scalable memory bandwidth, all modern general-purpose microprocessors have the memory controller on the chip. As we've seen earlier, through cache coherence all of the memory is visible throughout the entire system.

This works very well for applications. Even a serial application can use all of the memory in the system.

However, there is one issue: accessing another processor's memory takes longer. Access to data is totally transparent to the application, since it is simply a load (or store) instruction with a specific address. The hardware resolves where the data is and moves it to the thread that needs it, but this takes longer if the data has to come from the memory on another socket.

On SPARC T4, T5, and M5 based systems the difference between local and remote access is not as large as on older cc-NUMA systems, but not to be ignored if one wants to achieve a high degree of scalable performance, that is, scaling to a large number of threads.

To achieve the fastest memory access for serial applications, Oracle Solaris allocates memory using the "First Touch" policy. This means that the process or thread touching the data for the first time owns the data. It has the data in the memory connected to the processor where the process runs at the time the data is touched and with "touch" being defined as creating the TLB entries for the first time.

This default choice works very well because in a serial application there is only one thread executing and therefore it has all the data in its local memory. In case the data does not fit, it overflows to nearby memory. For many years, Oracle Solaris has increasingly been made cc-NUMA aware and the system is smart enough to handle such overflow in the most efficient way.

For a parallel application, things are slightly different because it is no longer obvious that one thread should own all the data. On the contrary, this is probably not desirable, as threads executing on a different processor may need to access remote memory to get to their data.

Luckily, in many cases the First Touch policy can also come to the rescue here, but it may require source code changes in the data initialization part.

Given how First Touch works, you must find where data is initialized for the first time and see how this part can be parallelized. The best strategy is to determine the relationship between the threads and data and make sure a thread initializes its portion of the data.

To make this a little more specific we show a very simple example using OpenMP. Assume that the threads use a subset of a vector called `a`. If we were to have `a` initialized by a single thread, all others threads need to access the memory where `a` resides. Instead, we use the OpenMP `#pragma omp parallel for` construct to perform the data initialization in parallel like this:

```
#pragma omp parallel for schedule(static)
    for (int i=0; i<n; i++)
       a[i] = 0;
```

The OpenMP runtime system distributes the loop iterations over the threads. By virtue of the `schedule(static)` clause, the first thread initializes the first section of `a`, the second thread the next chunk, etc. As a result, each thread now "owns" a section of this array and has it in its local memory, wherever that may be on the system.

This approach provides a portable and easy way to optimize data placement, but it is not applicable to all situations and for every application. It may be, for example, that the data access pattern changes during the execution. For cases like this and for the utmost flexibility, Oracle Solaris provides the `madvise` (3C) system call.

This function takes a starting address, a length, and a keyword as arguments. The keyword is the advice what to do with the data. For example, you might advise the system to migrate data to the next thread that uses the data, not the first thread that initialized it.

Last, but not least, it should be mentioned that data placement is controlled at the page level.

False Sharing

False sharing is one of the least understood factors that negatively impact scalability. This may be because it is not very common, luckily, but if it happens, the performance degradation is noticeable.

To explain false sharing we should take a closer look at what is happening at the level of the data cache(s). Data at this level is organized in "cache lines." The size of a cache line is architecture dependent and need not even be the same across all cache levels. Typical sizes are 32 or 64 bytes.

When the processor needs a data element, not only this element is fetched, but also the entire line containing the data needed is brought into the (level 1) cache. The reason to do it this way is to amortize the cost of the transfer. For good performance all elements in the same cache line should be used before a new line is brought in.

In a parallel application, it can easily happen that multiple copies of the same line exists. One reason this may occur is that multiple threads read different variables that are all part of the same line.

So far there is nothing to worry about, but what to do if one of these threads changes a variable in such a shared line? At that point there is an inconsistency between the newly modified line and the other copies. The way the system handles this is by "invalidating" these other copies. In this way, a thread knows it cannot use this line anymore and needs to get the most recent copy from elsewhere in the system.

Note that this could in fact be overkill because perhaps the portion of the cache line a thread is interested in has not been modified, but since this status is maintained on a line basis, not at the byte level, there is no way for the thread to know this and it will fetch a fresh copy.

The result of this false sharing is increased coherence traffic and cache lines travelling through the system. This can be costly, even though great efforts are made to handle this situation as efficiently as possible.

False sharing is always taking place and is not really noticeable. It gets to be a problem if multiple writes to the same cache line happen very frequently and in a relatively short time span, such as in the core part of an application.[5]

This is a situation that should be avoided and very often this can be done. Often the easiest solution is to "localize" updates and only write the final result to a shared variable.

You can achieve this in OpenMP very easily by using variables that are private to a thread.

Tips and Tricks

We conclude this section by giving additional advice about obtaining good performance.

- **Good performance starts by using the right compiler**. This is true for both serial and parallel performance. The Oracle Solaris Studio compilers are continuously tuned to generate optimal code. The teams work closely together with the SPARC engineers to ensure the best possible performance.
- **Use the Oracle Solaris Studio Performance Analyzer** that is bundled with these compilers. This tool easily identifies the performance hot spots in any serial or parallel application written in C, C++, Java or Fortran. The tool not only helps the user prioritize optimization opportunities, it also quickly shows communication bottlenecks, time spent in synchronization, etc.
- **Experiment with compiler options**. In addition to identifying "hot" sections in the code, also indicates optimizations the compiler has carried out. This may point at the need to explore some other options tailored to the situation.
- **Write clear code**. Try to think what you would do if you were the compiler. Looking at the source only, could you really figure out what the goal of this part of the code is? If it is clear to the compiler what the intention of a certain code block is it can generate more efficient code. Obscure coding can make this much harder. In that respect you may want to find another purpose for that older book with tuning tips. Many of these are no longer needed and can actually mask the intentions to the extent that the compiler will not generate the best possible code.
- **Verify your data structures**. Compilers can do a lot for you, but cannot fix every data structure issue. Make sure to set up your data structures such that cache lines are used in the optimal way. This is achieved by ensuring consecutive data accesses are also consecutive in memory.

[5] Note that false sharing does not occur on read only data.

- **Look at the bigger picture**. It is best to leave low level details to the compiler. It knows the characteristics of the processor in detail and can very well decide what kind of code to generate. Your responsibility is to make sure the compiler targets the right processor. By default it will optimize for the processor performing the compilation. If this is a different processor than the one used for deployment, use the `-xtarget` option on the Oracle Solaris Studio compiler to be specific what to optimize for.
- **Never forget Amdahl's Law**. This has been as true as ever and as ruthless too. Most scalability issues go back to this law. Therefore, avoid serial parts in the program as much as possible.
- **Use private data as much as possible.** Sharing of data is very convenient and at times necessary in a parallel application, but only share data if it is really needed. Using private data has several performance advantages. There is no risk multiple threads access the same memory spot at the same time and false sharing is avoided. It is also easier to place private data in the memory of the thread that needs it. This kind of local memory access is highly recommended for cc-NUMA systems.
- **Avoid excessive use of locks and barriers.** We see too many applications that make excessive use of locks. They are definitely needed at times and convenient to use, but few realize how expensive they are. The cost of locking also increases as more threads are used and can soon become a scalability bottleneck.
- **Think Parallel.** Last, but certainly not least, think in a parallel way when developing new code. Is it really necessary that only one thread performs the work, or can it be distributed? Why not have each thread do similar work as opposed to have one thread do all of it while the others are waiting in a potentially expensive synchronization construct such as a lock or barrier [20].

Many Cores Make Virtualization Work

One reason to use systems with many threads is to deploy parallel programs, but perhaps the scalability of the application is limited and not all threads can be used efficiently, or the workload consists of a mix of serial and parallel jobs.

This is where virtualization, resource partitioning and workload isolation comes into the picture.

The idea is to consolidate multiple applications that each use a single or limited number of threads on the *same* server. If the applications don't all have peak loads at the same time or were over-provisioned, the aggregate compute resources they share can be much lower than the sum of the separate machines. This can provide deep savings in financial cost, server count, floor space, software licensing, electricity, cooling, and maintenance effort.

A traditional way to do this is to combine multiple applications on the same multiprogramming operating system instance. This can be problematic if the different applications require incompatible operating system versions, different maintenance windows, or as is very common, application owners require independence and freedom from performance and security interference. The answer to this is the increasingly popular option of workload isolation provided by server virtualization. While server virtualization has been available on some platforms for many years, it is an increasingly popular deployment method used for workload isolation on modern UNIX servers.

Multiple models of workload isolation exist. Some methods, e.g., physical partitioning, provide complete electrical isolation. All of the components in one partition can fail in one instant, or degrade gradually, and the other partitions will be unaffected. Another method, virtual machines (VMs), uses a hypervisor to create a fiction of separate computers, when in reality those virtual machines share components such as HWTs, DIMMs, a memory bus, system interconnect, and an I/O subsystem. A third model, called "operating system virtualization" or "OS virtualization" provides the fiction of multiple operating system instances, even though multiple sets of distinct processes share the same OS kernel.

Virtualization

Oracle's server virtualization products support x86 and SPARC architectures and a variety of operating systems, such as Linux, Windows, and Oracle Solaris. In addition to solutions that are hypervisor-based, Oracle also offers virtualization built into hardware and Oracle operating systems to deliver the most complete and optimized solutions for your entire computing environment.

Oracle's SPARC T5 and M5 servers cover the entire range of virtualization methods, including Dynamic Hardware Domains, Oracle VM Server for SPARC (hypervisor-based), and Oracle Solaris Zones (OS-based). As a result of the SPARC T5 and M5's extreme level of multithreading, these features provide unprecedented flexibility and efficient scalability for server consolidation and cloud computing infrastructure.

Oracle Solaris Zones

Oracle Solaris Zones is the most widely recognized form of OS virtualization technology in the industry. First released in Oracle Solaris 10 in 2005, Oracle Solaris Zones began with a security boundary—isolation—that prevents a workload in one zone from interacting with a process in a different zone. Since then, features have been added on a regular basis, and integration with other innovations in Oracle Solaris have made this form of virtualization very popular. Oracle estimates that 40% of SPARC-based Oracle Solaris systems use Oracle Solaris Zones.

A zone is a virtualized operating system environment created within a single instance of the Oracle Solaris system; it is *not* a full OS kernel. When you create a zone, you produce an application execution environment in which processes are isolated from the rest of the system. This isolation prevents processes that are running in one zone from monitoring or affecting processes that are running in other zones. Even a process running with superuser credentials cannot view or affect activity in other zones.

A zone also provides an abstract layer that separates applications from the physical attributes of the machine on which they are deployed. Examples of these attributes include physical device paths.

Zones can be used on any machine that is running at least the Oracle Solaris 10 release. The upper limit for the number of zones on a system is 8192. The number of zones that can be effectively hosted on a single system is determined by the total resource requirements of the application software running in all of the zones.

Integrated and related features provide Oracle Solaris Zones with the following benefits [22]:

- Configurable security boundary, enabling you to fine-tune functionality, increasing or decreasing a zone's abilities and security
- Management control of a zone's resource utilization, including HWT use, quantity of HWTs available to a zone, quantity of RAM and virtual or shared memory available to a zone, network bandwidth consumed, and many others
- A wide variety of methods to access storage, including many different file systems and access to raw storage
- Ability to create "branded zones" that run binaries and use configurations from earlier versions of Oracle Solaris

OS virtualization uses a model of server virtualization that has, at its core, the assumption that multiple, independent sets of processes share a common kernel. The benefits of a common kernel include better efficiency and observability and simpler management, when compared with virtual machines.

Performance Advantages of Zones

OS virtualization does not need a hypervisor or other layer of software to implement isolation. Such software hypervisors can cause performance overhead of various types. These include the time the hypervisor runs on HWTs that would otherwise be running real workloads, inefficiencies caused by cache-flushing, and any operations that the hypervisor must perform on behalf of a VM, such as I/O using shared I/O resources. Instead, Oracle Solaris Zones do not suffer from these performance-draining effects because the individual processes do not follow a different code path just because they are running in a zone.

Security Advantages of Zones

Instead of using a hypervisor, isolation between zones is managed by the OS kernel. The kernel understands the isolation boundaries of zones, and prevents actions that would cross a boundary. Those security checks are performed whether a process is running in a non-global zone or in the global zone—the environment and set of processes that run when Oracle Solaris first boots. In this way, Oracle Solaris Zones can achieve the highest security rating achieved by general-purpose operating systems, and it does so without incurring a virtualization performance penalty.

Troubleshooting Advantages of Zones

Because all (non-global) zones are managed from the global zone of the system, a global zone user with sufficient privileges must be able to control each zone and its contents. Because the processes in a zone are, in one sense, one set of processes managed by one kernel, the privileged global zone user can see all of the processes in all of the zones. This is a great advantage of zones: when it is time to troubleshoot a system problem, the usual Oracle Solaris tools can be used to determine the process—in any zone—that is causing the problem. There is no need to first determine the guest that is causing the problem, and then login the guest to determine the process that is causing the problem. The latter method must be used when troubleshooting a hypervisor-based virtual machine environment.

Zones are typically, but not always, used to consolidate many workloads into one Solaris instance. That instance might be an entire computer (SPARC or x86), a hardware domain on an enterprise-class SPARC server, or a logical domain on any current SPARC server. In all of those cases, the HWT usage of workloads in each zone is managed by the Oracle Solaris scheduler and other parts of the kernel. The Oracle Solaris scheduler is one of those components that has received modifications regularly, enhancing its ability to scale to large number of HWTs.

By default, all of the processes of the zones of one Oracle Solaris instance can, theoretically, run on any of the cores available to the Oracle Solaris instance. With that configuration, the scheduler does an excellent job of maintaining HWT and memory locality. In other words, it tries to maximize the benefit of a core's cache by running the same process on the same core, each time the process is assigned a time slice. It also tries to minimize memory latency by assigning physical memory blocks that have the lowest latency when accessed by the HWT on which that process typically runs.

With Oracle Solaris' history of linear performance gains from an increasing quantity of cores, and the zero-overhead efficiency of zones, it is no surprise that zones scale as linearly as Oracle Solaris does. By default the Oracle Solaris scheduler treats all processes in different zones equally. In fact, it doesn't consider the association between a process and a zone—by default—when making scheduling decisions. On the other hand, there are several resource controls from which to choose, and the use of those tools modifies the scheduling outcomes in predictable ways.

Any compute environment should provide predictable performance. If a workload performs 100 transactions per second this week, its users will expect it to provide the same performance next week. In a single-workload environment, a stable system will deliver that predictable performance. However, in multitenant environments, the consumption of resources such as HWT capacity and RAM must be managed. Resource controls are needed to provide predictable performance of multiple workload environments.

Although Oracle Solaris tries to be fair when allotting resources to running workloads, an uncontrolled workload can consume enough resources that other workloads will not perform well. This is not a new problem, and resource controls have been added to Oracle Solaris gradually during the two decades it has been in the market.

Oracle Solaris has many resource controls, including items in the following list. All of these can be applied to Oracle Solaris Zones.

- HWT capacity: processor sets, HWT caps, multiple scheduling algorithms and priorities
- RAM usage: a RAM cap limits the amount of RAM that a zone's processes can use
- Virtual memory: a VM cap limits the amount of VM (RAM plus swap space) that a zone's processes can use

Oracle Solaris considers each hardware thread in a processor to be a "CPU" when it counts CPUs and when it schedules a process onto a HWT. SPARC systems in the current generation are equipped with a very large number of hardware threads. This gives the scheduler more flexibility when making scheduling decisions, and gives you more flexibility when deciding on a quantity of HWTs to assign to a workload—if you decide that method is appropriate for a set of workloads sharing a computer.

Consider the following example. A SPARC T5-4 has 4 processors, 64 cores, and 512 hardware threads.[6] One can quite easily create and run hundreds of Oracle Solaris Zones on that SPARC T5-4 system, and it is likely that they will run very well without the use of any resource controls. However, a set of 30 workloads, each in a zone, might include a small number that have specific performance needs. The Oracle Solaris resource controls can then be used to assign a set of 8 cores to one production database zone, and four sets of 8 cores to four production application servers, but leave the rest of the zones to share the remaining 24 cores.

Oracle Corporation charges license fees for its software based on the quantity of cores in the target computer. The licensing policy allows customers to limit the quantity of cores needed for a license if "hard partitioning" is used. At the time this paper was written, Oracle Solaris Zones that use a processor set (i.e. a resource pool) qualify as "hard partitions" and licensing fees can be reduced by using these "capped Containers." However, be aware that capped Containers as mentioned in the public licensing document uses processor sets, and do not use the `capped-cpu` feature of Oracle Solaris Zones [19].

Oracle VM Server for SPARC

Oracle VM Server for SPARC (previously called Sun Logical Domains) is a virtualization technology supported on Oracle's SPARC servers. Oracle VM Server for SPARC allows server processor resources to be partitioned and allocated to separate virtual machines, also called "domains."

Oracle VM Server for SPARC has a novel architecture that reduces overhead and makes it ideal for systems with many hardware threads. Traditional hypervisors virtualize multiple "virtual CPUs" onto a smaller number of physical HWTs. This causes overhead from software-based time-slicing and the need to emulate the guest operating system's use of "privileged instructions." This is necessary to prevent guests from changing the physical state of shared HWTs (like memory mapping and interrupt status). Every time the guest OS wants to perform certain privileged instructions, it has to trap and context switch into the hypervisor or perform a binary rewrite of the instruction. This causes substantial overhead, but has been the traditional model of virtualization for over 40 years.

In contrast, Oracle VM Server for SPARC is designed for and ideally suited for systems with many HWTs, for example, the SPARC T5-8 with 1,024 HWTs. Unlike a traditional hypervisor, hardware threads and memory are allocated *directly* to the logical domains. The guest OS (Oracle Solaris) running in that logical domain runs on "bare metal" and can perform the aforementioned privileged operations directly, without the need to context switch into the hypervisor. This key differentiator illustrates why the logical domain model does not carry the "virtualization tax" of HWT and memory overhead as the traditional model does. Guest domains run on their own dedicated hardware threads with full bare-metal performance.

[6] As a rough comparison, the compute capacity of such a system compares favorably with 20 UltraSPARC Sun Fire E10000's from 1999.

Another difference is that traditional VM systems depend on a monolithic hypervisor that combines management control point, resource manager, user interface, and device drivers along with the task of providing virtual machines. The architecture of Oracle VM Server for SPARC, on the other hand, is modular and consists of these main components:

1. A firmware based hypervisor that allocates resources (HWT, memory and optionally I/O devices) directly to logical domains.
2. A "control domain" that serves as a management control point, either via a command line interface (CLI) or via optional management suites such as Oracle Enterprise Manager Ops Center 12*c* or Oracle VM Manager 3.2 or later.
3. Software based I/O virtualization, which runs in a "service domain" and provides virtual I/O services to the guest domains. The control domain is typically used for this purpose. For redundancy there can be additional domains providing virtual I/O. This is an optional component, since I/O devices and PCI busses can be provisioned directly to domains.

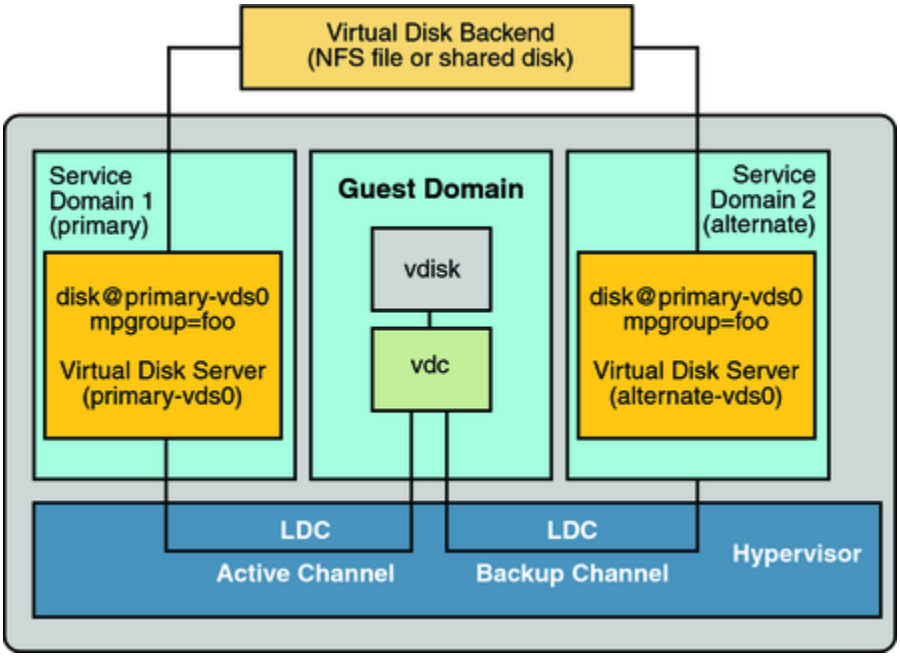

Figure 6. A guest logical domain with redundant service domains.

Oracle VM Server for SPARC delivers the following:

- **Leading Price/Performance.** The low-overhead architecture provides scalable performance under increasing workloads without additional license cost.
- **Secure Live Migration.** Lets you migrate an active domain from one physical SPARC server to another while the guest continues to operate and provide application services to users. Memory contents are always compressed and encrypted while transmitted from source to destination server, thereby guaranteeing security and privacy. On-chip cryptographic accelerators in the S3 core deliver secure, wire-speed encryption for live migration without any additional hardware investment.
- **Single-Root I/O Virtualization (SR-IOV).** Delivers superior I/O throughput to guest domains with qualified PCIe devices. Such devices can appear as if they were separate I/O devices, each of which can be directly assigned to a domain. SR-IOV enables efficient sharing of PCIe network devices so applications can achieve native I/O performance.
- **Dynamic Reconfiguration (DR).** Allows computing resources to be dynamically added to or removed from an active domain. You can make resource assignment changes to processors, virtual I/O, and memory on an active domain. These capabilities help organizations to better align IT and business priorities.
- **Advanced RAS.** Each logical domain is an entirely independent virtual machine with its own OS, as is expected in a VM environment. The logical domain supports virtual disk multipathing and failover, as well as network failover with IP multipathing (IPMP) support. The logical domain can also handle path failure between an I/O domain and storage. When virtual I/O is used, multiple service domains can be used to provision virtual I/O to client guests. This provides device and path redundancy, and also permits rolling upgrades without service interruption: when one service domain is taken down for maintenance, virtual I/O is routed through the active service domain. Domains are fully integrated with the Oracle Solaris FMA (Fault Management Architecture), which enables Predictive Self Healing.
- **Whole Core Allocation and Core and Memory Affinity.** Permits higher performance by avoiding false cache sharing between guest domains, and reducing the non-uniform memory access (NUMA) effects on latency between HWTs and RAM. This enables organizations to deliver higher and more predictable performance for all workloads.
- **HWT Dynamic Resource Management (DRM).** Enables a resource management policy and domain workload to trigger automatic addition and removal of HWTs based on domain resource requirements and priorities. A policy could be set to add HWTs to a domain, up to a maximum preset number, as long as it has HWT utilization above a specified threshold, or conversely, remove HWTs from domains that are mostly idle. Priorities can be specified to establish which domains have preferential access to HWTs. This helps you to better align your IT and business priorities.
- **Physical-to-virtual (P2V) Conversion.** Quickly convert an existing SPARC server that runs the Oracle Solaris 8, Oracle Solaris 9, or Oracle Solaris 10 OS into a virtualized Oracle Solaris 10 image. Domains can also host branded zones, in which an Oracle Solaris 11 OS hosts both Oracle Solaris 11 and Oracle Solaris 10 zones, or an Oracle Solaris 10 OS hosts Oracle Solaris 10 zones along with Oracle Solaris 9 and Oracle Solaris 8 branded zones.

- **Server Power Management.** Implements power saving by disabling each core that has all of its hardware threads idle or adjust HWT clock speed and memory power consumption.

For further guidance about virtualizing your SPARC Solaris environment, see [22] and the Oracle white paper *Best Practices for Building a Virtualized SPARC Computing Environment* [23].

Oracle Enterprise Manager Ops Center

Oracle Enterprise Manager provides an integrated and cost-effective solution for complete physical and virtual server lifecycle management. It delivers comprehensive provisioning, patching, monitoring, administration, and configuration management capabilities via a web-based user interface, and significantly reduces the complexity and cost associated with managing Oracle VM, Linux, UNIX, and Windows operating system environments. It helps accelerate the adoption of virtualization and cloud computing to optimize IT resources, improve hardware utilization, streamline IT processes, and reduce costs.

Oracle Enterprise Manager Ops Center 12*c* provides a comprehensive solution for operating system, firmware and BIOS configuration, bare-metal and virtual machine provisioning, hardware fault analysis, automatic My Oracle Support service request generation, and performance management.

The Oracle Enterprise Manager Ops Center Virtualization Management Pack provides full lifecycle management of virtual guests, including resource management and mobility orchestration. It helps customers streamline operations and reduce downtime. Together, the Virtualization Management Pack and the Oracle Enterprise Manager Ops Center Provisioning and Patch Automation Pack provide an end-to-end management solution for physical and virtual systems through a single web-based console. This solution automates the lifecycle management of virtual systems and is the most effective systems management solution for Oracle's Sun infrastructure.

Oracle Solaris 11 Scalability

None of the advanced features of the SPARC multicore processors would be of much use if the operating system were unable to take advantage of them. And as the number of processor cores and hardware threads increases dramatically, OS algorithms for managing huge physical and virtual memory and page sizes, for scheduling thousands of processes and software threads, and for handling large numbers of physical and virtual devices, all have to be made scalable.

Oracle Solaris engineers also use a powerful kernel observability tool—DTrace [6]—to examine and improve every aspect of Oracle Solaris to maximize OS performance and scalability as the underlying hardware evolves. For example, as the number of interacting software threads increases, lock contention can become a major problem; the Oracle Solaris 11 lock management algorithm now scales linearly with the number of hardware threads. Additionally, the Oracle Solaris virtual memory system has been enhanced to better handle very large pages and to predict and optimize page replacement, as well as allocating both kernel and user memory close to the processor cores where the related processes are running, significantly reducing memory latency.

Kernel tunables in `/etc/system`, in particular `maxusers`, `max_nprocs`, and `pidmax`, now automatically scale with the number of available hardware threads and memory on the system; Oracle Solaris 11.1 now supports more than one million software threads.

Conclusion

In this paper, we have covered many of the key concepts and techniques of parallel programming. We have emphasized the great value and some of the limitations of multiprogramming and multithreading for applications, and have focused specifically on the advantages of Oracle's new massively multithreaded SPARC T5 and M5 processors. We also highlighted important developer tools such as OpenMP, Oracle Solaris Studio compilers, and the Oracle Solaris Studio Performance Analyzer that assist developers in finding and exploiting parallelism in their applications.

In addition to explaining how the SPARC processors enhance multithreaded applications, we also discussed how Oracle's virtualization technologies—Oracle VM Server for SPARC and Oracle Solaris Zones—allow system managers and IT architects to efficiently consolidate multiple application environments onto a single server.

In summary, Oracle's new SPARC T5 and M5 processors, and the servers built with them, provide unprecedented scalability and opportunities to exploit both application and system parallelism. The proof of this is not only in the many computing world records these systems have achieved [21], but also in the wide and successful deployment of these systems in a wide variety of scenarios.

Glossary

API

API stands for "Application Programming Interface." It specifies how software components interact. An API typically provides a library with (optional) specific data structures and object classes. The functions in the library provide the toolkit one can draw upon. Examples in parallel programming are POSIX Threads, Java Threads, OpenMP, and MPI (Message Passing Interface).

Atomic Operations

These are low-level operations used in parallel programming. They are called "atomic" because several related actions act as an indivisible unit.

Unfortunately, atomic operations are not part of the Pthreads API, however, there are many (non-portable) APIs for atomic operations available. OpenMP also provides support for atomic operations.

Barrier

This is a point in a parallel application where no thread can continue until the last one has arrived. Barriers are definitely very useful and needed, but use them with care. The cost grows as more threads are added and this can quickly dominate the performance.

Cache

This is a memory buffer with a shorter access time than main memory. It is used for data and instructions. The goal is to keep frequently referenced data (or instructions) in cache, greatly speeding up accesses. In contemporary microprocessors one can find multiple levels of cache, typically denoted with L1, L2, etc. The notation is such that access to an L1 cache is faster than an L2 cache. It is similar for the L2 cache versus L3 cache. A unified cache stores both data and instructions.

Cache Coherence

This is a crucial part of the design of a shared memory parallel system. Since it can easily happen that multiple copies of the same cache line are present in the various caches in the system, maintaining consistency of the same cache line across the entire system needs to be handled. This is what cache coherence does. To this end, each cache line has state bits and they are used to describe the status of the line. The cache coherence protocol dictates how cache lines change state as a result of a certain action. For example when an element in a line is modified not only the state bits of the modified line need to be changed, but any other copy of the same line needs to be invalidated. This is achieved by changing the state bits of those lines accordingly.

Container

See Zone

CPU

This stands for "central processing unit," an archaic term referring to the computational component (ALU) of a processor. Modern processors consist of many cores, each including multiple "hardware threads" (equivalent to a traditional CPU).

HWT

A hardware thread, equivalent to the traditional concept of a CPU. Modern processors consist of multiple core units, each including multiple HWTs. This is also referred to as a "strand" or "virtual processor" in some Oracle Solaris tools (e.g., the `psrinfo` command).

Mutliprogramming

The technique of running multiple processes on a single HWT (or CPU); the operating system schedules each of the processes a limited amount of time to execute, then switches to a different process.

Multithreaded

This is another word for a parallel application. Such a program is said to be multithreaded. This means that the code has a software infrastructure to create and execute multiple threads.

Parallelization

The process of identifying those parts in an application that are independent and can be executed in any order without affecting correctness of the results. If this is the case, those parts can also be executed at the same time, or "in parallel." Once such program sections have been identified one needs to select a suitable parallel programming model to implement this type of concurrent execution. How to express the parallelism strongly depends on the model chosen.

Parallel Programming Model

This provides the software toolbox to implement the parallelism in an application. It also defines what kind of functionality is available. There are many models available, both for single systems as well as clusters. Typically, a parallel programming model provides a way to create threads, execute work in parallel, exchange data, synchronize threads and terminate parallel execution.

Pragma

This provides for a portable way to pass on additional information to a C/C++ compiler. A pragma starts with the `#pragma` keyword, followed by one or more qualifiers. Such qualifiers can either be compiler-specific or portable across platforms, in the case of OpenMP, for example.

A pragma does not affect portability because a compiler ignores a pragma it does not support. In this way, one can give specific information to a compiler without violating portability.

OpenMP is built upon this model. All language constructs start with `#pragma omp`. If a compiler is not instructed to recognize these, the source code can still be built with a compiler that does not support OpenMP. The pragmas are ignored and the runtime functions can be handled through an appropriate `#ifdef`.

Processor Pipeline

Each instruction goes through various stages. For example, in the fetch stage, the instruction is fetched from the instruction cache. It is then decoded in the decode stage. Collectively, these stages form a pipeline. The hardware automatically advances an instruction from stage to stage. It flows through the pipeline, but the cost of certain stages may vary. Like a data look up stage, depending on where the data is, this can be very quick or take a much longer time (more clock cycles). These delays in the pipeline are also called "stalls" or "bubbles."

RAS

This stands for reliability, availability, and serviceability.

Shared and Private Data

In a shared memory parallel program there are two fundamental types of data: shared and private.

Shared data is accessible by all threads. Each thread can read and/or write such data at any point during the execution. It is the responsibility of the developer to ensure this occurs in the right way. Depending on the parallel programming model chosen there are specific ways to do so.

Private data is only accessible to the thread that owns it. Even if the variable names are the same, threads cannot read or write each other's private data. If this is needed, one has to copy such data to a shared buffer.

In OpenMP the "shared" and "private" clauses on the relevant pragmas can be used to label variables as such.

Thread

In the context of this paper we often refer to a software thread simply as "a thread." With this we mean a sequence of instructions with an independent Program Counter (PC) to keep track of the point of execution. Multiple threads can execute simultaneously and are scheduled by the operating system onto the hardware. In the case of the SPARC T4, T5 and M5 system, Oracle Solaris first selects a core to run on and then a hardware thread ("virtual processor") within this core.

TLB

This stands for "Translation Look-Aside Buffer." It is a special type of cache the stores mapping information between the virtual and physical memory addresses. It is possible to have separate TLBs for instructions (I-TLB) and data (D-TLB).

Zone

The Oracle Solaris Zones virtualization technology is used to virtualize operating system services and to provide an isolated and secure environment for running applications.

References

1. Moore's Law, Wikipedia.org

2. "SPARC T5 Deep Dive: An interview with Oracle's Rick Hetherington," Oracle.com

3. "SPARC T5: 16-core CMT Processor with Glueless 1-Hop Scaling to 8-Sockets," a presentation of SPARC T5 at HotChips 24

4. Oracle Engineered Systems, Oracle.com

5. Oracle Optimized Solutions, Oracle.com

6. DTrace, Oracle documentation site, Oracle.com

7. Gregg, Brendan and Jim Mauro, *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, Prentice Hall, 2011.

8. Oracle Solaris Studio, Oracle.com

9. Combs, Gary, "Oracle's SPARC T5-2, SPARC T5-4, SPARC T5-8, and SPARC T5-1B Server Architecture," an Oracle white paper, March 2013.

10. Combs, Gary, "SPARC M5-32 Server Architecture," an Oracle white paper, March 2013.

11. "Oracle's SPARC T4-1, SPARC T4-2, SPARC T4-4, and SPARC T4-1B Server Architecture," an Oracle white paper, October 2011.

12. van der Pas, Ruud and Jared Smolens, "How the SPARC T4 Processor Optimizes Throughput Capacity: A Case Study," an Oracle technical white paper, April 2012.

13. POSIX Threads, Wikipedia.org

14. OpenMP.org

15. Chapman, Barbara, Gabriele Jost and Ruud van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, October 2007.

16. van der Pas, Ruud, "Parallel Programming with Oracle Developer Tools," an Oracle white paper, May 2010.

17. *Oracle Solaris Studio OpenMP User's Guide*, Oracle.com

18. Amdahl, Gene M., "Validity of the single processor approach to achieving large scale computing capabilities," presented at the AFIPS Spring Joint Computer Conference, 1967.

19. Oracle Partitioning Policy, http://www.oracle.com/us/corporate/pricing/partitioning-070609.pdf

20. Gove, Darryl, *Multicore Application Programming: for Windows, Linux, and Oracle Solaris*, Addison-Wesley Professional, Nov. 9, 2010.

21. Oracle Unveils SPARC Servers with the World's Fastest Microprocessor, an Oracle press release, March 26, 2013.

22. Victor, Jeff, et al., *Oracle Solaris 10 System Virtualization Essentials*, (Boston: Pearson, 2010), p. 169–226.

23. "Best Practices for Building a Virtualized SPARC Computing Environment," an Oracle white paper, September 2012.

READ_ME_FIRST: What Do I Do with All of Those SPARC Threads?

August 2013
Authors: Ruud van der Pas, Jeff Savit, Jeff Victor, Darryl Gove, and Harry Foxwell

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0613

Hardware and Software, Engineered to Work Together