ORACLE

SOLARIS

An Oracle White Paper
May 2010

Parallel Programming with Oracle® Developer Tools

ORACLE

Introduction

This technical white paper targets developers interested in learning about parallel programming. After introducing major concepts needed to get started writing parallel applications, it delves into two commonly used parallel programming models: OpenMP and the Message Passing Interface (MPI). OpenMP provides for parallelization of applications running within a single multicore-based system. MPI provides for parallelization of applications running across many such systems, often referred to as a *compute cluster*.

Fully worked examples are provided for OpenMP and MPI, as well as examples of combining those two approaches, typically called the *Hybrid model*. The programming examples also include how to use the Oracle® Solaris Studio compilers and tools, plus the Oracle Message Passing Toolkit MPI software, providing the reader full information on both developing and deploying parallel applications on either the Oracle Solaris operating system or Oracle Enterprise Linux, as well as other Linux environments.

The growing interest in parallel processing is being driven by the multicore trend in today's microprocessor designs, which has now turned even laptops into parallel computers. From a software perspective, in order to achieve optimal application performance, developers will need to exploit multicore parallelization going forward. In order to do so, there are several important things to learn.

This technical white paper covers the essential topics needed to get started developing parallel applications. Basic parallel concepts are introduced in the first three sections, followed by discussions that delve into the OpenMP and MPI parallel programming models, including the Hybrid model. The remainder of the paper covers additional considerations for parallel applications, such as Amdahl's Law and parallel application speed up, as well as performance results for a parallelized application. Examples of OpenMP and MPI are provided throughout the paper, including their usage via the Oracle Solaris Studio and Oracle Message Passing

Toolkit products for development and deployment of both serial and parallel applications on Oracle's Sun SPARC® and x86/x64 based systems.

The Oracle Solaris Studio [1] [2] software provides state-of-the-art optimizing and parallelizing compilers for C, C++ and Fortran, an advanced debugger, and optimized mathematical and performance libraries. Also included are an extremely powerful performance analysis tool for profiling serial and parallel applications, a thread analysis tool to detect data races and deadlock in memory parallel programs, and an Integrated Development Environment (IDE).

The Oracle Message Passing Toolkit [3] [4] software provides the high-performance MPI libraries and associated run-time environment needed for message passing applications that can run on a single system or across multiple compute systems connected with high-performance networking, including Gigabit Ethernet, 10 Gigabit Ethernet, InfiniBand, and Myrinet.

Throughout this paper it is demonstrated how Oracle's software can be used to develop and deploy an application parallelized with OpenMP [5] and/or MPI [6]. In doing so, it touches upon several additional features to assist the development and deployment of these types of parallel applications.

Note that prior to Oracle's acquisition of Sun Microsystems in January 2010, the Oracle Solaris Studio and Oracle Message Passing Toolkit products were known by the name Sun Studio and Sun HPC ClusterTools respectively. These products support Oracle Solaris and Oracle Enterprise Linux, as well as other Linux environments.

Target Audience

This paper targets programmers and developers interested in utilizing parallel programming techniques to enhance application performance. No background in parallel computing is assumed. The topics covered are sufficient for a good understanding of the concepts presented, and to get started.

Multicore Processor Technology

These days, a computer program is almost always written in a high-level language such as Fortran, C, C++, or the Java™ programming language. Such a program consists of a symbolic description of the operations that need to be performed. These are human readable, but cannot be executed directly by the processor. For this, a translation to machine instructions needs to be performed. Such a translation can be performed up front, or *statically*, by a compiler, or *dynamically* at run time by an interpreter, as occurs with Java. The generated instruction stream is then executed by the processor.

In Figure 1, a short fragment of such an instruction stream is listed for demonstration purposes. These instructions were generated from the C program listed in Figure 12 on page 24, compiled with the Oracle Solaris Studio C compiler. Shown here are some instructions from source lines 5 and 6.

```
        ......
805136c:  addl    $-1,%ecx
805136f:  movl    0x24(%esp),%edx
8051373:  movl    %eax,%esi
8051375:  jns     .+4 [ 0x8051379 ]
8051377:  xorl    %esi,%esi
8051379:  cmpl    $8,%esi
805137c:  jl      .+0x41 [ 0x80513bd ]
805137e:  addl    $-8,%eax
8051381:  prefetcht0  0x100(%edx)
8051388:  addsd   (%edx),%xmm1
        ......
```

Figure 1. Fragment of an instruction stream

The address of the instruction is listed in the first column. The second column contains the instruction proper. The operands for the instruction are shown in the third column. The processor executes these instructions in sequence, but some instructions, like a branch, can move the execution flow up or down the stream.

Note that on a processor with a superscalar architecture, multiple instructions from the same stream can execute simultaneously. This is a very low level, but still important kind of parallel execution, often referred to as Instruction Level Parallelism (ILP) and supported by many current processors. The Oracle Solaris Studio compilers aggressively schedule instructions to exploit ILP and take advantage of this level of parallelism. Since it only affects the performance of the individual threads, ILP is not considered any further in this paper.

Until relatively recently, most processors executed one single instruction stream at any point in time. This is often referred to as *serial* or *sequential* execution.

In a multicore processor architecture there are multiple independent processing units available to execute an instruction stream. Such a unit is generally referred to as a *core*. A processor might consist of multiple cores, with each core capable of executing an instruction stream. Since each core can operate independently, different instruction streams can be executed simultaneously. Nowadays all major chip vendors offer various types of multicore processors. A block diagram of a generic multicore architecture is shown in Figure 2.

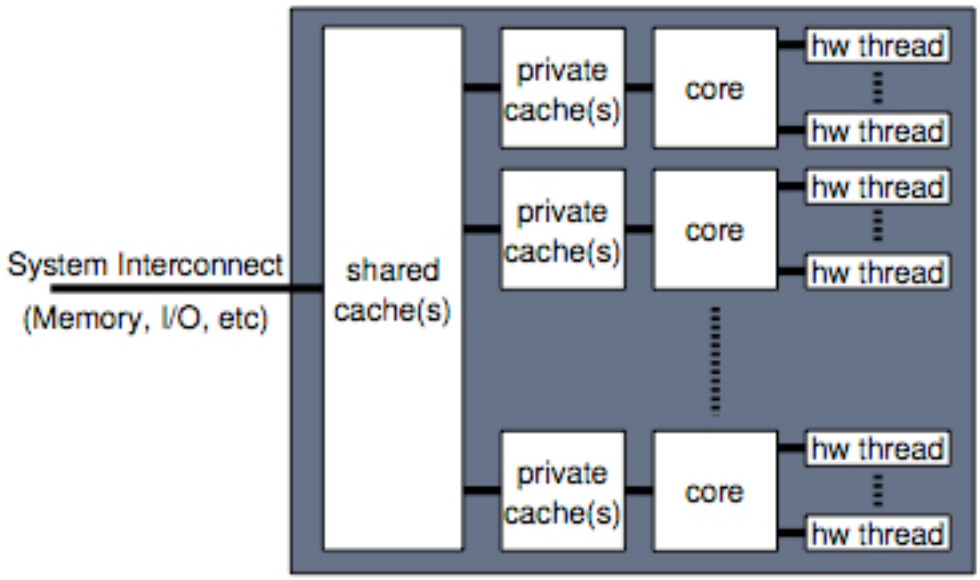

Figure 2. Block diagram of a generic multicore architecture

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

On the memory side, multiple levels of fast buffer memory can be found. These are generally referred to as *cache memory* or cache(s) for short. More information on caches can be found in [7], [8] and [9]. Today first level caches are typically local to the core. Higher-level caches can be local, but may also be shared across the cores. Typically at least the highest level of cache often is shared.

The instruction streams can be completely unrelated. For example, one might watch a video on a laptop, while having an email client open at the same time. This gives rise to (at least) two instruction streams. We say "at least" because each of these applications could be internally parallelized. If so, they might each execute more than one instruction stream. How to achieve this is the topic of the second part of this paper, starting with the Parallel Programming Models section.

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

In this paper, the focus is on a different use of the multiple cores available. The cores also can be used to improve the performance of a single application. By assigning different portions of work to the cores in the system, but still within the same application, performance can be improved. We refer to this as *parallel computing*.

It is important to realize that there are significant differences between the cores found in various architectures. These most likely have an effect on performance. For the purpose of parallel programming, the most significant feature is that the cores provide support for parallel execution in the hardware. This makes a single multicore processor a (small) parallel system.

In the remainder of this paper, a core is referred to as the hardware component providing the parallelism. For ease of reference, a processor that can only execute a single stream of instructions is often still referred to as a core.

Basic Concepts in Parallel Programming

In this section various aspects of parallel programming are covered. This is done from a practical perspective, informally explaining the terminology, as well as addressing important concepts one needs to be aware of before getting started writing a parallel application, or when parallelizing an existing program.

In the Additional Considerations Important In Parallel Applications section on page 55, additional considerations are presented and discussed. These are however not needed to get started and might come into the picture once the application has been parallelized.

What is a Thread?

Loosely speaking, a thread consists of a sequence of instructions. A thread is the software vehicle to implement parallelism in an application. A thread has its own state information and can execute independently of the other threads in an application. The creation, execution and scheduling of threads onto the cores is the responsibility of the operating system. This is illustrated in Figure 3.

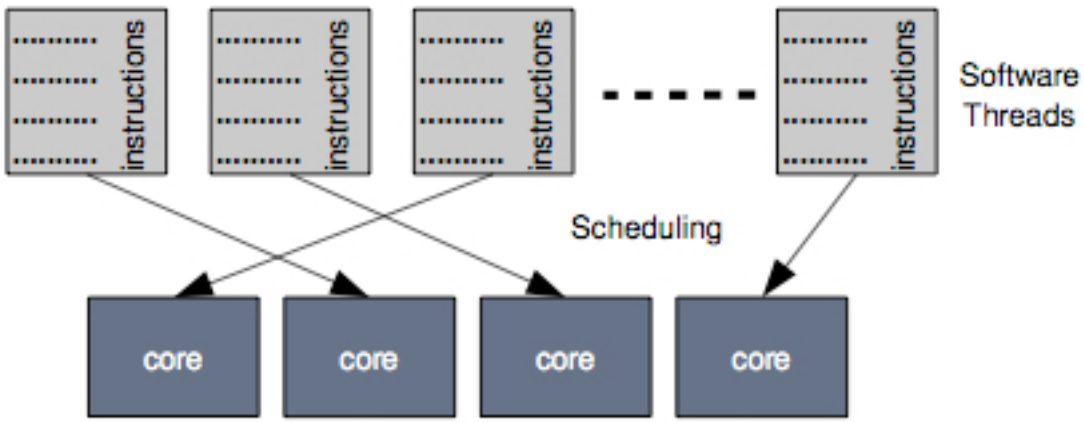

Figure 3. Software threads scheduled onto the cores

In general it is best for performance to make sure the hardware resources used are not overloaded and do not exceed their capacity. In case a resource is overloaded, the common phrase is to say that this resource is *oversubscribed*. For example, when executing more than one application on a single core, the operating system has to switch between these programs.

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

The exception is if a core has hardware support for multiple threads. In this case, some level of oversubscription of a core could be beneficial for performance. The number of software threads to use depends on the workload and the hardware implementation details. The Performance Results section on page 64 details some performance results on this kind of heavily threaded architecture.

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

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

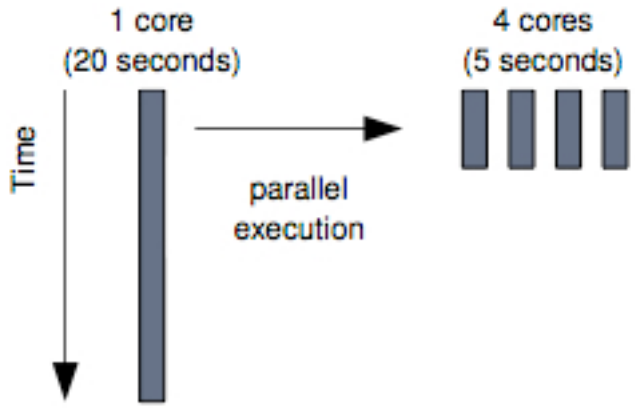

Figure 4. Parellelization reduces the execution time

Why Parallelization?

With serial optimization one tries to use the hardware resources available within one core more effectively. For example by utilizing the pipelines in the core in a more efficient way, and/or by improving the memory access pattern in the application, resulting in a better use of the cache subsystem. Using these techniques, the single core performance may be improved. In the worse case little or no gain can be realized this way. For an extensive coverage of this topic and many other aspects of application tuning, we refer to [9]

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

The execution time for a parallel program is also referred to as the *elapsed time*, or *wall clock time*, or in UNIX® terminology, the *real* time. It is essentially the time on one's watch from the moment the program starts until it finishes.

The significant difference here is that unlike tuning for serial performance, with parallelization more than one core is executing the program. So the goal must be to use these additional cores as efficiently as possible to reduce the execution time. But any performance improvement is not a given. Depending on the application characteristics, the parallel programming model selected, the implementation of the parallelism in the application, the system software and the hardware used, one might see little or no performance improvement when adding cores. And in some cases enabling more cores can degrade performance when compared to single core execution.

Parallelization is therefore not only about getting the right results -- performance should be a consideration early in the design phase as well. More information on this can be found in the Additional Considerations Important In Parallel Applications and Performance Results sections.

What is Parallelization?

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

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

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

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

The choice of the programming model has substantial consequences regarding the implementation, execution and maintenance of the application. Both approaches have their pros and cons. We strongly recommend to carefully consider these before making a choice.

Parallel Architectures

In this section an overview of various types of parallel systems is given. These are generic descriptions without any specific information on systems available today. For that we refer to the technical information on the particular system of interest. Before covering several architectures, a short introduction into cache coherence is given. This feature provides a single system *shared memory* view to the user, even if the memory is physically distributed.

Cache Coherence

A cache line is the unit of transfer in a general-purpose microprocessor. A typical line size is 32 or 64 bytes. Each time a core issues a load instruction to fetch a piece of data from memory, the cache line this data is part of, is copied into the cache hierarchy for this core.

In a parallel program it can easily happen that multiple cores simultaneously load data elements that are part of the same cache line. As a result, multiple copies of the same line exist in the system. As long as the lines are read only, all these copies are consistent, but what if one, or multiple cores, modifies the line they just loaded?

At this point, an inconsistency at the hardware level arises. There is now at least one cache line that has a different data element than the other copies. Care needs to be taken that any core issuing a load instruction for a data element in the same line receives the correct version of the line.

Cache coherence is a hardware mechanism that keeps track of such changes and ensures that modifications to cache lines are propagated throughout the system. To this end, a cache line is augmented with a status field, describing the state the line is in. For example, if the line is marked as "invalid", its contents can no longer be used.

Cache coherence provides a single system shared memory view to the application. The program can load and store data without the need to know where it is located. This greatly enhances ease of use when it comes to parallel programming.

For performance reasons, cache coherence is implemented in hardware. Different algorithms and implementations are used in the various systems available today. Since cache lines are modified very often, an efficient design has a noticeable and positive impact on the performance.

For completeness, one comment needs to be made here. Multiple and simultaneous writes to the same cache line is another important and different issue to be dealt with. The policy how to handle these is part of the memory consistency model implemented. This topic is however beyond the scope of this introductory paper. For an extensive coverage of this topic, as well as cache coherence, we refer to [7].

The Symmetric Multiprocessor (SMP) Architecture

The probably most well known parallel architecture is the Symmetric Multiprocessor (SMP). Such a system consists of a set of processors, possibly with a multicore architecture, I/O devices, one or more network interfaces and main memory. In Figure 5 a block diagram of this architecture is shown.

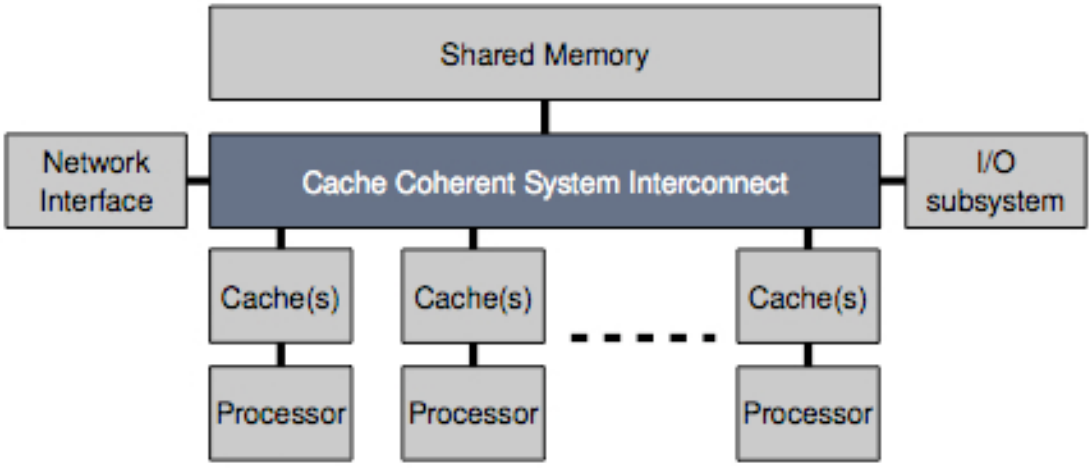

Figure 5. Block diagram of an SMP system

The SMP system is globally cache coherent, providing a single system view. The artery of the system is formed by the system interconnect.

There are significant architectural differences between various implementations. For example, the topology and architecture of the interconnect, latencies, bandwidths and cache coherence protocol all highly depend on the system specifics, and thus affect the performance of the system and applications.

The key characteristic all SMP systems have in common however is that each core has transparent access to any memory location in the system. Moreover, the main memory access time is the same for all processors/cores, regardless of where the data resides in memory. This is also referred to as a flat or uniform memory access (UMA) architecture. The two main advantages of the SMP are the flat memory and sharing of resources. Because of the single address space, even a serial application can make use of all the memory available in the system.

Note that at the architectural level, a multicore processor is very much like an SMP.

The Non-Uniform Memory Access (NUMA) Architecture

The opposite architecture of the SMP consists of a cluster of single core nodes, connected through a non-cache coherent network, like Ethernet or InfiniBand. A node consists of a complete system. It could for example be a PC, workstation, laptop or a more custom designed system. Figure 6 shows a block diagram of such a system.

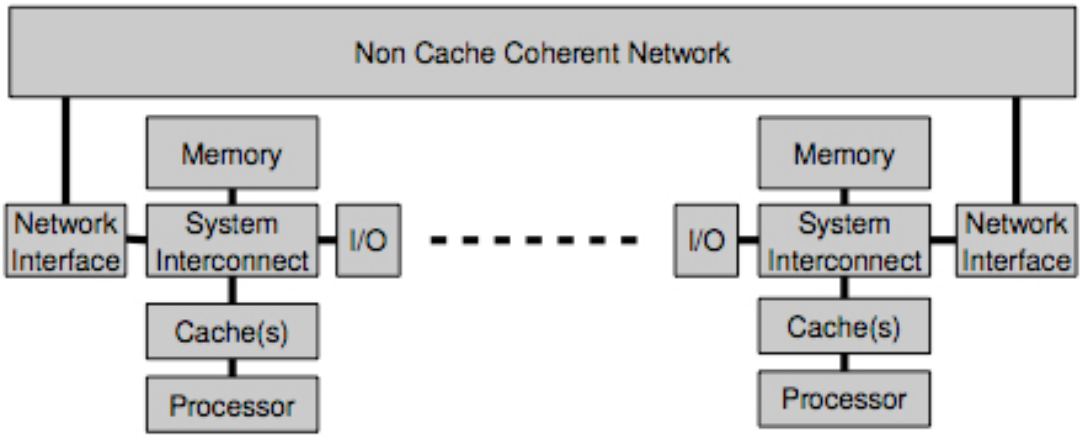

Figure 6. Block diagram of a NUMA system

In this kind of architecture, none of the hardware resources other than the network are shared. In particular, each node in the cluster has access to its own local memory only. Remote access to another memory is not transparent. It has to be explicitly programmed by the developer. For this reason, this type of system is also referred to as a *Distributed Memory* architecture.

In comparison with local access, accessing remote memory takes longer. How much longer depends on the interconnect characteristics, as well as the size of the packet transferred over the network, but in all cases, the delay is noticeable. This is often described as Non-Uniform Memory Access, or NUMA for short.

Because of the multicore trend, single core nodes become more and more rare over time. This is why this type of architecture will over time be replaced by the Hybrid architecture.

The Hybrid Architecture

A Hybrid system consists of a cluster of SMP or multicore nodes, connected through a non-cache coherent network. In Figure 7 a block diagram of such an architecture is shown.

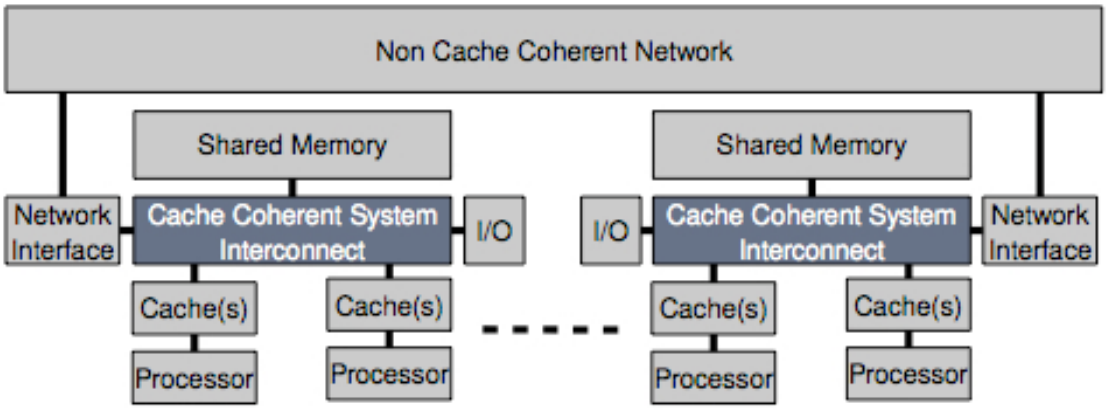

Figure 7. Block diagram of a Hybrid system

At the programming level, this kind of system can either be seen as a distributed memory architecture, ignoring the fact that the cores within one node share their memory, or as a two level parallel system with a distributed memory architecture at the top level and a shared memory system at the second level. With that, one has the best of both worlds.

This type of system is rapidly becoming the dominant architecture, because increasingly the individual nodes have one or more multicore processors. The interconnect of choice is either Ethernet or InfiniBand.

The Cache Coherent Non-Uniform Memory Access (cc-NUMA) Architecture

The last system in this overview is the cache coherent Non-Uniform Memory Access (cc-NUMA) architecture. It is very similar to the Hybrid system, but there is one major difference. The top level interconnect is cache coherent, providing the system with a single address space. In other words, an application can access any memory location in the system, regardless of where it is physically located.

Figure 8 shows a block diagram of a cc-NUMA system.

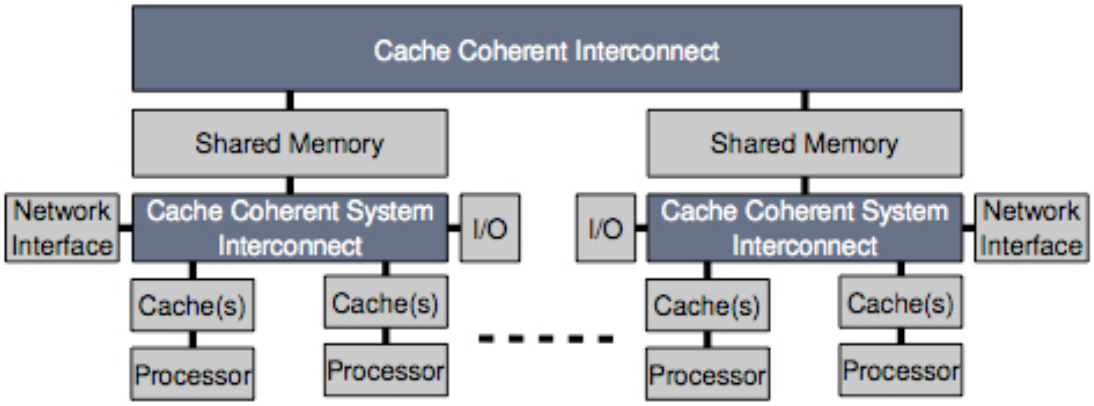

Figure 8. Block diagram of a cc-NUMA system

The transparent access of memory throughout the entire system greatly enhances ease of use, but there is still a performance penalty accessing remote data. How much of a penalty depends on the system characteristics. For good parallel performance it is important to take this into account and ensure that most, if not all, memory accesses are from a memory closest to the core that reads, and/or modifies the data.

Increasingly, microprocessor designs move away from a centralized memory architecture. Instead, each processor has its own memory controller, connecting to a part of the total memory available. The advantage is that memory bandwidth scales with the number of processors and each (multicore) processor has fast access to its local memory. Thanks to cache coherence, access to a remote memory is supported transparent to the application, but it takes longer to get to the data. In other words, such systems have a cc-NUMA architecture and these days they are more and more common. This trend also implies that the individual nodes in Hybrid architecture have a cc-NUMA architecture as well, further complicating the memory hierarchy in such systems.

Parallel Programming Models

There are many choices when it comes to selecting a programming model for a parallel system. In this section the focus is on the more commonly used models. After a global overview of some popular parallel programming models, we zoom in on two-shared memory models (Automatic Parallelization and OpenMP), followed by the Message Passing Interface (MPI) model for distributed memory and hybrid systems. We conclude with some remarks on the Hybrid programming model.

An Overview of Common Parallel Programming Models

Numerous choices are available to program a parallel computer. In this section some of the more commonly used programming models are briefly covered. In the remainder of this paper, the focus is on Automatic Parallelization, OpenMP, MPI, as well as the combination of MPI and OpenMP. The latter is an example of a Hybrid programming model, combining a shared memory and distributed memory model.

The Shared Memory Model

These are common ways to develop and deploy a parallel program for a system with a single address space, that is, a shared memory architecture as found in multicore, SMP, and cc-NUMA systems.

Automatic Parallelization

Here, the compiler performs the dependence analysis to determine if a specific part of the program can be executed in parallel. If it can prove this is safe to do, it generates the underlying infrastructure, typically a series of calls to a multitasking library. The user activates this feature through a compiler option. The source(s) compiled with this option are analyzed and where possible, parallel code is generated. The analysis is loop oriented. This could be a single level or nested loop. In case a loop is parallelized, the work associated with the loop iterations is split into disjoint chunks. These chunks of iterations are assigned to the threads.

Several factors affect success or failure of Automatic Parallelization:

- **The programming language.** A language like Fortran is easier to analyze at compile time than C++ for example. Due to the higher abstraction level in the latter, it can be harder for the compiler to identify independent portions of work.
- **The application area.** Certain algorithms are inherently parallel, whereas in other cases the parallelism is more indirect, or worse, non-existent.
- **Coding style.** To achieve the maximum benefit from the analysis, the compiler needs to extract as much information as possible from the application. Depending on the coding style, this can be a challenge, or not.
- **The compiler.** Identifying the parallelism at compile time can be quite complicated. This requires a serious investment in compiler technology.

The Oracle Solaris Studio compilers support Automatic Parallelization. It is simply enabled by adding the `-xautopar` option to the options already used for serial optimization. Additionally one can add the `-xreduction` option to allow the compiler to parallelize a specific class of operations. More information can be found in the Automatic Parallelization Using The Oracle Solaris Studio C Compiler section on page 25. There also an example is presented and discussed.

Explicit Parallelization Using the OpenMP Programming Model

This is a directive based model. By inserting compiler directives in the source program, the user defines what part(s) can be executed in parallel. The compiler then translates this into a parallel program.

OpenMP supports an extensive set of features to specify the parallelism, control the workload distribution, and synchronize the threads. Although not required, it turns out that current OpenMP implementations are built on top of a native threading model. The Oracle Solaris Studio compilers support a combination of Automatic Parallelization and OpenMP. This minimizes the effort, since the compiler can first be used to parallelize the application. Those parts that are too complicated for the compiler to handle can then subsequently be parallelized with OpenMP.

A more extensive coverage of OpenMP can be found in The OpenMP Parallel Programming Model section on page 16. A fully worked example is given in Parallelizing The Example Using OpenMP on page 28.

A Native Threading Model

Examples are POSIX Threads, Solaris Threads, Java Threads, or Windows Threads. Through an API, the developer has access to a set of specific functions that can be used to implement and control the parallelism. For example, function calls to create the threads and synchronize their activities.

Such a programming model provides a powerful, yet fairly low level of functionality and control. Higher level features need to be built using the lower level function calls. This type of programming model has been available for a relatively long time and successfully used to parallelize various types of applications. These days, higher-level alternatives like OpenMP, but also other programming models not discussed here, are available. If possible, such a model might be preferred, since it can reduce the development cycle and reduce maintenance cost.

The Distributed Memory Model

For a cluster of systems with a non-cache-coherent network, a distributed memory parallel programming model can be utilized to communicate between the nodes in the network. In this model, memory is not shared. Each process has only access to its own, local, memory. That is, a process can only read from or write data to this local memory. In case data has to be made accessible to another process, the owner of this data has to send it explicitly to the process(es) that need it. And, the process(es) that need the data have to issue an explicit receive request. Also in this case, several choices are available.

- **Network socket programming.** This is not really a programming model, but a network protocol to send and receive network packets across a cluster. It is very low level and quite far away from what the typical application developer requires to parallelize a program.
- **Parallel Virtual Machine (PVM).** This was the first successful effort to provide a de-facto standard for parallel programming targeting a cluster of systems. PVM not only provided an API to send and receive messages between processes running on the various nodes of the cluster, but also for example functionality to synchronize processes across the entire system. Implementations for many systems were available. Typically these were built on top of network sockets, but in some cases an implementation would take advantage of a more efficient communication between processes running on the same (shared memory) node.
- **The Message Passing Interface (MPI).** This is today's most popular programming model to program a cluster of nodes. Several shortcomings in PVM were addressed and especially with the introduction of MPI-2, more extensive functionality than PVM was provided. This is why MPI eventually superseded PVM and is today's standard to program a distributed memory system.

  Outside of the basic functions to create the processes, as well as send and receive messages, MPI provides a very rich API through an extensive set of additional functions, including various ways to handle I/O. We strongly recommend to check the specifications before implementing one's own more complicated case. Most likely, the MPI API already provides for what is needed. Examples are the `MPI_Bcast()` function to broadcast the same message to all processes, the `MPI_Reduce()` function to gather a message from all processes on one process and apply a basic arithmetic operator to it, and the `MPI_Isend()/MPI_Irecv()` functions to communicate messages asynchronously.

  In The Message Passing Interface (MPI) Parallel Programming Model section on page 18, a more extensive coverage of MPI can be found. Parallelizing The Example Using MPI on page 39 contains a fully worked example.
- **Partitioned Global Address Space (PGAS).** This type of programming model has been around for quite some time. Through language extensions and/or directives, parallel programming is simplified by avoiding the developer has to handle many of the low level details that typically come with other distributed memory programming models, like explicit send and receive operations. Relatively recent examples are Unified Parallel C (UPC) and Co-Array Fortran.

Automatic Parallelization

Automatic Parallelization is a feature supported by the Oracle Solaris Studio compilers. Through the `-xautopar` compiler option, the user requests the compiler to parallelize the source code for the shared memory model. In other words, such a parallel program can be run on a multicore, SMP, or cc-NUMA type of system.

Automatic Parallelization is based on parallelizing loops. In a nested loop, the compiler may first re-order the loops to achieve optimal serial performance. The resulting loop nest is then considered for parallelization. For efficiency reasons, the compiler first tries to parallelize the outermost loop. If this is not possible, it will consider the next loop, etc.

In order to guarantee correctness of the results, the compiler needs to determine whether the iterations of the loop under consideration can be executed independently. To this end, a dependence analysis is performed. If the analysis shows it is safe to parallelize the loop, the corresponding parallel infrastructure is generated. In case of doubt, or if there is a clear dependence, the loop will not be parallelized.

In the Automatic Parallelization Using The Oracle Solaris Studio C Compiler section on page 25, an example of automatic parallelization is given.

The OpenMP Parallel Programming Model

In this section we briefly touch upon this programming model. For the full specifications and more information, we refer to [5], [10], [11], and [12]. In the Parallelizing The Example Using OpenMP section on page 28, a complete OpenMP example program is presented and discussed extensively.

A Brief Overview Of OpenMP

OpenMP uses a directive based model. The developer adds directives to the source to specify and control parallel execution. For example, all that is needed to parallelize a for-loop in C is to insert a `#pragma omp parallel for` pragma line above the loop. In Fortran, the syntax to parallelize a do-loop is `!$omp parallel do`.

It is however important to note that OpenMP is much richer than simply parallelizing loops. Several flexible and powerful constructs are available to parallelize non-loop code in an application. For example, the tasking model allows arbitrary blocks of code to be executed in parallel. In the tasking model, the developer uses pragmas to define the chunks of work, called tasks. There is the implied assumption that all tasks can execute concurrently. Getting this right is the responsibility of the developer. The compiler and run time system handle the details to generate and execute the tasks.

In addition to the directives, a run time library is available. These routines can be used to query and change the settings as the program executes. A commonly used routine is `omp_get_num_threads()`. It returns the number of threads in the parallel region. The routine `omp_get_thread_num()` returns a unique thread identifier, starting with 0. It is also possible to change the number of threads while the program executes by calling the `omp_set_num_threads()` routine.

There are many more routines available. In the An Example Program Parallelized Using Various Models section and Appendix A it is shown how some of these routines can be used. For a complete overview and more details we refer to the OpenMP specifications [10]. Through a set of environment variables the user can specify certain settings prior to executing the program. An example is the `OMP_NUM_THREADS` variable to specify the number of threads to be used. In the example in the Parallelizing The Example Using OpenMP section on page 28, several of these features are shown and discussed in detail.

An OpenMP program can be seen as an extension of the serial version of the program, because the directives provide a portable mechanism to implement the parallelism. All current OpenMP compliant

compilers use a specific option to get the compiler to recognize the directives. If the source is compiled without this option, the directives are ignored, resulting in serial code to be generated.

The OpenMP run time routines require special care though. If such a routine is used in the source, compilation in serial mode causes the routine to not be recognized. Luckily there is an easy solution for this. Through the `#ifdef` construct in C/C++ and the conditional compilation OpenMP feature available in Fortran [10] [11] one can avoid that the OpenMP run time routines give rise to unresolved references at link time. This is illustrated in the example in the Parallelizing The Example Using OpenMP section on page 28.

The OpenMP Execution And Memory Model

An OpenMP program uses the fork-join execution model. It is shown in Figure 9.

Initially, the program starts in serial mode. Only the initial, master, thread is running. When a parallel region is encountered, the additional threads are activated and assigned work.

The parallel region forms the backbone of OpenMP. This is where activities take place concurrently. The details of what needs to be done depend on the constructs used within the parallel region and are specified by the developer. In C/C++ the `#pragma omp parallel` directive is used to define the parallel region. In Fortran the `!$omp parallel` directive needs to be used for this.

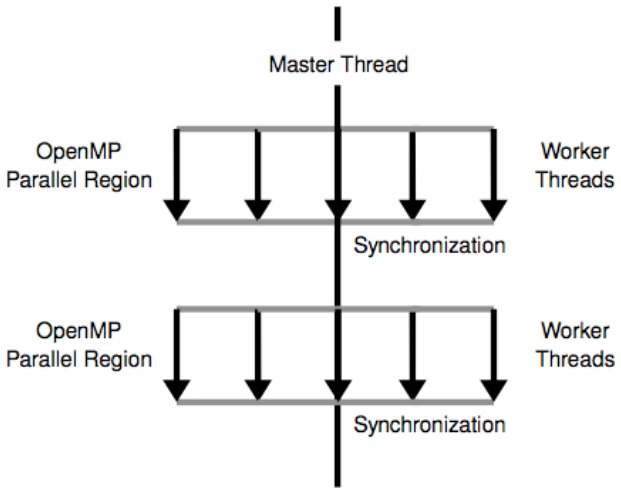

Figure 9. The fork-join execution model in OpenMP

Within the parallel region, the master thread behaves like one of the worker threads and participates in the activities to be performed. At the end of the parallel region all threads wait until the last one arrives. At this point, the master thread continues by itself until the next parallel region is encountered.

The memory model defines two types of memory: private and shared. Each thread has exclusive access to variables in its private memory. Any change made to such a variable is only seen by the thread that

made the modification. In OpenMP, private variables are undefined on entry to and exit of the parallel region, but this can be changed through the `firstprivate` and `lastprivate` clause respectively.

A shared variable behaves very differently. It is accessible by all of the threads. Essentially, there is only one copy of this variable and it is visible to all threads. The initial value is inherited from the master. Any thread can read and/or write a shared variable and it is up to the developer to make sure this happens in a correct way. An example of using private and shared variables can be found in the Parallelizing The Example Using OpenMP and Parallelizing The Example Using The Hybrid Model sections.

A subtle issue is determining at what point in time a change made to a shared variable is visible to all other threads. The OpenMP memory model allows threads to have a (potentially) different temporary view of a shared variable, but at certain execution points, a consistent view is guaranteed (e.g. after a barrier). By using the `flush` directive, the programmer can also enforce consistency. Although not required for OpenMP compliance, support for cache coherence makes this feature relatively easy to implement.

The Message Passing Interface (MPI) Parallel Programming Model

In this section this distributed memory-programming model is briefly covered. For the full specifications and more information, refer to [6], [13] and [14]. In the Parallelizing The Example Using MPI section on page 39, a complete MPI example program is presented and discussed extensively.

A Brief Overview Of MPI

An MPI program consists of a set of processes, running on one, or multiple, nodes of a cluster. Each process is a full operating system process. Communication between the processes has to be put in by the developer. If a process needs some data that is in the memory of another process, the process that owns this data needs to explicitly make it available to the process interested in it. The most common way to do this is for the owner to execute the `MPI_Send()` function, specifying the receiver as the destination. On the receiving side, the process that needs the data then has to execute the `MPI_Recv()` function to receive this data. In MPI terminology, such a data packet is called a *message*. Since multiple messages can be sent to a single receiver, the receiver needs to have a way to distinguish the various messages. This can be achieved by using a unique label, or message tag, in the send and receive call.

As stated in [13], only 6 functions are needed to write a message passing application, but a much richer environment is provided, including asynchronous and global operations.

For example, the commonly used `MPI_Send()` function returns when the message is considered to be sent and the user may do anything to the buffer, but it actually may have been queued up in a list to be sent out. There is no guarantee if or when the message has arrived. On the other hand, the `MPI_Ssend()` function only returns when the receiver has actually received the message. This can however cause the sender to be idle (block) while waiting for the message to arrive. In such cases, the asynchronous send and receive functions (`MPI_Isend()` and `MPI_Irecv()`) are very useful to allow communication and computation to overlap. This can be very beneficial for the performance. For example, the `MPI_Isend()` function is guaranteed not to block due to progress in the message

delivery not happening. Functions to test or wait for arrival of a message are available and needed when using this feature.

An important concept in MPI is the *communicator*. This refers to a set of MPI processes. Many MPI functions require the communicator to be specified. When the `MPI_Init()` function is called to initialize the MPI environment, the overall `MPI_COMM_WORLD` communicator is created (along with `MPI_COMM_SELF`, the communicator consisting of the process itself). This communicator is therefore always present and includes all processes in the MPI job. Each process is assigned a unique rank to the `MPI_COMM_WORLD` communicator, which is a consecutive integer number starting from zero.

This single communicator is sufficient for many MPI applications. Through the `MPI_Comm_create()` function it is however possible to create a subset of this global communicator and then restrict communication to this subset only by specifying the appropriate communicator in the MPI function call. The subset can for example be used to restrict a global operation, e.g. `MPI_Bcast()`, to all processes in the subset only.

Multiple communicators can be created and used. For example, two communicators can be used to define a compute set of processes, as well as a visualization set of processes. Communication can then be restricted to either one of these sets. If `MPI_COMM_WORLD` is used, a process can send to any of the other processes in the entire MPI job. Additional functionality to manage communicators is supported through various MPI functions.

Note that one can use the `MPI_Comm_dup()` function to make a copy of `MPI_COMM_WORLD` and essentially have two communicators with the same abilities of connections. This can for example be useful to separate control messages from data messages that need to go to any of the processes.

Another important type of communication is the global (or *collective* in MPI terminology) operations. These are very convenient and frequently used to perform the same type of communication, like a send or receive, but across all of the processes in the communicator. The convenience is that only a single function call needs to be used for this. They can also be optimized by MPI implementations for networks used, node sizes, communicator sizes and data sizes. Through the `MPI_Bcast()` function for example, one can send the same message to all processes.

The `MPI_Gather()` function performs the opposite operation. It collects a message from all processes. Through the `MPI_Reduce()` function one can perform a gather operation, while also applying a basic or user-defined arithmetic operation on the incoming data. Such a function is ideally suited to collect and sum up partial sums for example.

A very advanced feature is the one-sided communication. With this, a process can use the `MPI_Get()` and `MPI_Put()` functions to respectively independently read or write a memory section in another process. In other words, no matching send or receive operation is required. The developer needs to take care this happens in the correct way. With sufficient support by the implementation it provides an efficient, but low level, way to transfer a message. Note that when using this feature, more than these two MPI functions are needed.

MPI also comes with its own set of I/O functions. This part of the specification is usually referred to as MPI-IO and provides functionality to handle various I/O scenarios in an MPI application.

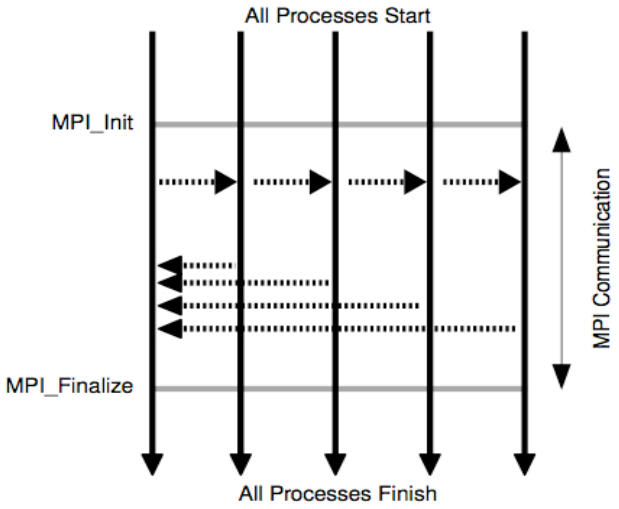

Figure 10. The MPI execution model

The MPI Execution and Memory Model

The MPI execution model is shown in Figure 10.

Through a command that is specific to the MPI implementation (e.g. `mpirun` for the Oracle Message Passing Toolkit), the user starts the program, specifying how many processes need to be started and which node(s) to use. Using options and/or configuration settings, one can for example also control what specific nodes of the cluster to use, how many processes should be run on each node, and if desired, on which core(s) of the node(s). Upon a successful launch, all processes start running independently. When an MPI function is executed, e.g. an `MPI_Send()` call to send a message to another process, the underlying MPI library and run time system perform the action requested.

The first and mandatory function to be executed prior to any other MPI function, is `MPI_Init()` to initialize the run time environment. After the completion of all MPI communications in the program, the `MPI_Finalize()` function should be called. After this, no MPI functions can be executed any longer. It is typically called just before the program finishes.

The horizontal arrows in Figure 10 indicate MPI communication. The process number starts with 0 and is numbered consecutively. Communication is between processes. On which node, or core, the process runs is determined by the configuration set up and options specified when executing the program with the `mpirun` command.

Unless one-sided communication is used, the memory model of MPI is very simple and explicit. Each process has access to its local memory only. Even when executed on a shared memory system, there is no sharing of data at the MPI level. If or when a process needs some data from another process, the owner of that data has to send it. The advantage of this model is that the behavior is consistent with a serial application and does not require additional attention. The downside is that any data that needs to

be shared, even if it is read only data, has to be copied and communicated over the network to all processes that need it.

Note that an MPI implementation can take advantage of shared memory under the hood though. Instead of using a network protocol for the communication, it can more efficiently communicate messages via shared memory. This is one of the features of the Oracle Message Passing Toolkit.

The Hybrid Parallel Programming Model

In a Hybrid parallel application, distributed and shared memory programming models are combined to parallelize an application at two levels. Typically, MPI is used for the distributed memory component, spreading the work over the nodes in a cluster. The process(es) running within one node are then further parallelized using a shared memory model, these days usually OpenMP. The Hybrid model is a very natural fit for a cluster consisting of multicore nodes. Clearly one can run an MPI application across all of the nodes and cores in the system, but what if is there is finer grained parallel work that would be hard to parallelize efficiently with MPI?

Another reason to consider the Hybrid model is that the memory within one node is used more economically by exploiting shared data through OpenMP and avoid the need to replicate data. In such cases, this combined model is very suitable. It is often relatively easy to use OpenMP to implement the second level parallelism.

Hybrid applications using MPI and OpenMP are best compiled and linked using the Oracle Message Passing Toolkit compiler drivers (e.g. `mpicc`) with the `-xopenmp` option added. Although not required, compiling with the `-xloopinfo` and -g options is highly recommended. The first option causes feedback on the parallelization to be printed on the screen. The second option is needed to get the compiler to annotate the object file(s) with very useful diagnostics on the serial optimizations and parallelization performed. This information can be extracted from the object file through the `er_src` command covered in the The Oracle Solaris Studio Compiler Commentary section on page 33.

Since this is a special case of an MPI application, the `mpirun` command should be used to execute the program. Prior to running the program, environment variable `OMP_NUM_THREADS` should be set to the desired number of OpenMP threads. Unless the application itself sets the number of OpenMP threads at run time. When running the application on a cluster, the `-x` option needs to be used to export environment variables like `OMP_NUM_THREADS` to the other nodes. Below is an example of the commands used to compile, link and execute a Hybrid application:

```
$ mpicc -fast -g -xopenmp -xloopinfo hybrid_program.c -o hybrid.exe
$ export OMP_NUM_THREADS=4
$ mpirun -np 8 -x OMP_NUM_THREADS ./hybrid.exe
```

When running this program, 8 MPI processes are created and executed on the system(s). Each MPI process uses four OpenMP threads. In the Parallelizing The Example Using The Hybrid Model section on page 50, an example of a Hybrid application written in C is given. Appendix C lists the Fortran equivalent of this example.

An Example Program Parallelized Using Various Models

In this section an example program is presented and discussed in great detail. The main function computes the average of a set of (double precision) numbers. We show how to design and implement the parallelism. Some performance results are given in the Performance Results section on page 64.

We begin by showing how to use the Oracle Solaris Studio C compiler to automatically parallelize the source code of the function implementing the computation of the average. Next, OpenMP is used to explicitly parallelize this computation. The same is then demonstrated using MPI. In the last section the Hybrid model is applied to create a two level parallel version. Although the example is written in C, there are only very small syntactical differences with Fortran regarding OpenMP and MPI. In Appendix C the Fortran source of the Hybrid version is listed. A Hybrid Fortran Implementation Of The Example Program on page 53 shows how to compile, link and run this program.

The Example Program

Before introducing the function that computes the average of a set of double precision numbers, the main driver program is presented and discussed. The source code is listed in Figure 11. This main program is fairly straightforward. Four additional non-system functions are used. They are highlighted in bold. In Appendix A the sources of these functions are listed and briefly explained. In Figure 11:

- Lines 1-14 contain the declarations and definitions found in a typical C program.
- At line 16 the user is prompted for the size of the array. This value is stored in variable `n`.
- At line 19 function `setup_data()` is called. This function allocates the memory for array `data`, initializes it and returns the exact result. The main program stores this value in variable `ref_result` and uses it later on to check for correctness of the computed result.
- At line 21 function `get_num_threads()` is called. It returns the number of threads used. This value is printed to verify the number of threads is according to what it is expected to be.
- At line 23 the timer is started by calling `get_wall_time()`. It returns the absolute time in seconds.
- At line 25 function `average()` is called. This function performs the computational work. The result is returned in variable `avg`.
- At line 27 the timer function is called again. By subtracting the new value from the old one, the elapsed time of the call to function `average()` is measured.
- At line 29 the computed result is compared against the reference result. As explained in Appendix A, the function `check_numerical_result()` that performs this task, takes differences caused by round off into account. If the test for correctness fails, diagnostic information regarding the numerical results is printed. Depending on the return value of this function, the main program either prints the final result and the elapsed time (line 32), or gives an error message (line 35). In other words, if the average is printed, it is implied the result meets the correctness criteria as checked for in this function.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 double average(int n, double *data);
 5 double setup_data(int n, double **pdata);
 6 double get_wall_time();
 7 int check_numerical_result(double avg, double ref_result);
 8 int get_num_threads();
 9
10 int main()
11 {
12   double avg, ref_result, *data;
13   double t_start, t_end;
14   int    n, ret_code = 0;
15
16   printf("Please give the number of data points: ");
17   scanf("%d",&n);
18
19   ref_result = setup_data(n, &data);
20
21   printf("Number of threads used: %d\n",get_num_threads());
22
23   t_start = get_wall_time();
24
25   avg = average(n,data);
26
27   t_end = get_wall_time() - t_start;
28
29   ret_code = check_numerical_result(avg, ref_result);
30
31   if ( ret_code ==  0 ) {
32     printf("n = %d average = %.2f (took %.3f seconds)\n",
33            n,avg,t_end);
34   else {
35     printf("ERROR: COMPUTED RESULT IS WRONG\n");
36     ret_code = -2;
37  }
38  free(data);
39  return(ret_code);
40 }
```

Figure 11. Main driver program to get the user input, set up the data, execute function "average", and check the result

The function listed in Figure 12 computes the average of a set of n double precision numbers, stored in array data.

```
1  double average(int n, double data[])
2  {
3    double sum = 0.0;
4
5    for (int i=0; i<n; i++)
6      sum += data[i];
7
8    return(sum/n);
9  }
```

Figure 12. Source of function "average" to compute the average of "n" double-precision numbers

The loop starting at line 5 accumulates the sum of the array elements into variable sum. The average is obtained by dividing this number by n, the total number of elements. This value is returned by the function (line 8).

The program is compiled and linked using the Oracle Solaris Studio C compiler:

```
$ cc -c -fast -g main.c
$ cc -c -fast -g check_numerical_result.c
$ cc -c -fast -g -xopenmp get_num_threads.c
$ cc -c -fast -g -xopenmp get_wall_time.c
$ cc -c -fast -g setup_data.c
$ cc -c -fast -g average.c
$ cc -o main.exe main.o check_numerical_result.o get_num_threads.o
  get_wall_time.o setup_data.o average.o -fast -g -xopenmp
```

The -fast option is used to have the compiler optimize the program. This option is a convenience macro that includes a set of performance related options. It is highly recommended to use, since it is an easy way to obtain good performance with a single option.

There is much more that can be said about the -fast option, as well as some additional options to further improve the performance. We refer to [9], [15] and the Oracle Solaris Studio documentation [16] for more details.

The -g option is not only used to generate the information for the debugger, it also causes compiler-generated information on the optimization and parallelization to be stored in the object file. This is a unique feature of the Oracle Solaris Studio compilers and is called "The Compiler Commentary". More information on this can be found in The Oracle Solaris Studio Compiler Commentary on page 33.

Since functions `get_num_threads()` and `get_wall_time()` use OpenMP to determine the number of threads and elapsed time respectively, the **`-xopenmp`** option is added when compiling this source. The Parallelizing The Example Using OpenMP section on page 28 contains more information on this option. To avoid unresolved references to the underlying OpenMP run time support library, the **`-xopenmp`** option is also used on the link command.

The binary can now be run for any value of `n` (assuming the number is not so big as to exceed the capacity of the int data type). Below the result for n = 100,000,000.

```
$ ./main.exe
Please give the number of data points: 100000000
Number of threads used: 1
n = 100000000 average = 50000000.50 (took 1.238 seconds)
$
```

It took about 1.24 seconds to perform the computation and the computed result is equal to the exact result. Note that this also illustrates the point made at the end of the Parallel Computing, Floating-Point Numbers And Numerical Results section on page 55. There it is shown that already for a much smaller value of `n`, round off errors affect the numerical result in case single precision numbers are used to perform this kind of computation. The difference is that in this example program double precision numbers are used instead. This greatly increases the relative precision.

Automatic Parallelization Using The Oracle Solaris Studio C Compiler

The computation of the average of a set of numbers is an example of a scalar reduction operation. This type of operation returns a single number as the result of an associative and commutative binary operator on a higher dimensional data structure [17]. Examples are the summation of the elements of an array (or as in this case, the related computation of the average) and determining the maximum or minimum value of a set of numbers.

A reduction operation can be parallelized. This is discussed in more detail in the Parallelization Strategy section on page 27. The Parallelizing The Example Using OpenMP, Parallelizing The Example Using MPI, and Parallelizing The Example Using The Hybrid Model sections show in what way this computation can be explicitly parallelized with OpenMP, MPI and the Hybrid model respectively. In this section it is demonstrated that the Oracle Solaris Studio C compiler can actually do all the work for us.

The source file listed in Figure 12 has been compiled using the Oracle Solaris Studio C compiler. The options added to the serial options are marked in bold.

```
$ cc -c -fast -g -xautopar -xreduction -xloopinfo average.c
"average.c", line 5: PARALLELIZED, reduction, and serial version generated
$
```

The `-xautopar` option invokes the automatic parallelization feature of the compiler. In this case this is not sufficient however. By default, reduction operations are not parallelized by the Oracle Solaris Studio compilers. This is why the `-xreduction` option has been included as well.

The `-xloopinfo` option, when used with the `-xautopar` option and/or the `-xopenmp` option, issues helpful diagnostic messages on the compiler's parallelization. It provides for an easy way to check which part(s) of the program the compiler has parallelized.

The Compiler Commentary provides more detailed diagnostics on the parallelization by the compiler. For more information on this feature we refer to The Oracle Solaris Studio Compiler Commentary section on page 33.

In this case the compiler has successfully identified the parallelism in the loop at line 5. This means that each thread gets assigned a subset of the total number of iterations to be executed. For example, if this loop has 1,000 iterations and two threads are used, each thread works on a subset of 500 iterations.

Note that the number of threads to be used at run time should be explicitly set with the `OMP_NUM_THREADS` environment variable. This needs to be done prior to running the program. By default, the number of threads is set to 1.

We also see the message "and serial version generated". This means that the compiler generated both a parallel, as well as a serial version of this loop. If, for example, there are only a few loop iterations, the additional parallel overhead may outweigh the performance benefit and the loop could actually run slower using multiple threads. A run-time check decides which version to execute in order to realize the best performance under all circumstances. For more information on compiler options specific to parallelization with Oracle Solaris Studio refer to [15].

The other source files are compiled using the same options as were used for the serial version. This is also true for the link command. This is how the automatically parallelized version has been compiled and linked.

```
$ cc -c -fast -g main.c
$ cc -c -fast -g check_numerical_result.c
$ cc -c -fast -g -xopenmp get_num_threads.c
$ cc -c -fast -g -xopenmp get_wall_time.c
$ cc -c -fast -g setup_data.c
$ cc -c -fast -g -xautopar -xreduction -xloopinfo average.c
"average.c", line 5: PARALLELIZED, reduction, and serial version generated
$ cc -o main_apar.exe main.o check_numerical_result.o get_num_threads.o
get_wall_time.o setup_data.o average.o -fast -g –xopenmp
$
```

There is no need to use the `-xautopar` option when linking, since the -xopenmp option is already used. Both cause the OpenMP run time support library to be linked in.

The parallel version of this program can be executed using any number of threads. Below is the output when running the program using 1, 2 and 4 threads:

```
$ export OMP_NUM_THREADS=1
$ ./main_apar.exe
Please give the number of data points:  100000000
Number of threads used: 1
n = 100000000 average = 50000000.50 (took 1.220 seconds)
$ export OMP_NUM_THREADS=2
$ ./main_apar.exe
Please give the number of data points:  100000000
Number of threads used: 2
n = 100000000 average = 50000000.50 (took 0.610 seconds)
$ export OMP_NUM_THREADS=4
$ ./main_apar.exe
Please give the number of data points:  100000000
Number of threads used: 4
n = 100000000 average = 50000000.50 (took 0.305 seconds)
$
```

The numerical result is identical for the three runs and equals the exact result. Clearly, automatic parallelization works really well for this computation. The parallel speed up is 1.220/0.610 = 2.00 using 2 threads, and on 4 threads the speed up is 1.220/0.305 = 4.00. In other words, in both cases a linear speed up is realized. For a definition and more information on the parallel speed up we refer to the Parallel Speed Up And Parallel Efficiency section on page 60.

The single thread time is even a little less than for the serial version but it is quite common to see some small variations in execution time and since the difference is only 1.4% we can ignore it. Additional performance results are given and discussed in the Performance Results section on page 64.

Parallelization Strategy

In the previous section it was shown that the Oracle Solaris Studio C compiler is able to detect the parallelism for us. In the remainder of this section it is explained how to do it ourselves using OpenMP, MPI and the Hybrid model. Before turning to the implementation, the parallelization strategy needs to be defined by formulating the steps to be taken to parallelize this computation.

The key observation to make is that summing up the elements of an array can be broken into pieces. Each thread can compute the sum of a subset of the array. These partial sums then need to be accumulated into the total sum and divided by the number of elements in the array.

The steps to be considered for a parallel version are outlined below. Note that although the word thread is used here, these steps are generic and not yet specific to a particular programming model.

1. Define for each thread a part of the data to work on
2. Make sure the thread has access to the part of the data it needs
3. Each thread computes the partial sum of its part of the data
4. This partial sum is accumulated into the total sum
5. One thread computes the final result by dividing the total sum by the number of elements

Depending on the programming model selected, some of these steps may not be needed, but it is good practice to design the algorithm before implementing it in a specific parallel programming model.

Parallelizing The Example Using OpenMP

In this section it is demonstrated how OpenMP can be used to parallelize the serial version of the algorithm. Using the steps given in the Parallelization Strategy section on page 27, the design of the algorithm is outlined. Most of these steps are easy to map onto an OpenMP version, but one in particular requires more attention. In the section following, the source code with the implementation is presented and discussed.

The Design Of The OpenMP Algorithm

Most of the steps outlined in the Parallelization Strategy section are very straightforward to implement using OpenMP. Access to data is very easily facilitated by making it shared. This allows every thread to read and, if needed, modify the data. Step 4 requires some more care though. When accumulating the final sum, each thread needs to add its contribution to it. This needs to be done in such a way that only one thread at a time can do so. Luckily, the OpenMP `critical section` construct provides for a very easy way to do this. It is explained in more detail in The Implementation Of The OpenMP Algorithm section on page 29.

This leads to the following OpenMP algorithm to parallelize the computations performed in the example program:

1. Use the default work allocation schedule of OpenMP to assign work to threads
2. Make array `data` shared, allowing all threads to access their part of the data
3. Each thread computes the partial sum of its part of the data and stores it in a private variable
4. Use the OpenMP `critical section` construct to add all partial sums to the total sum
5. The master thread divides the total sum by the number of elements to obtain the final result

It should be noted that this type of reduction operation can actually be very easily and conveniently handled through the `reduction` clause in OpenMP, which eliminates the need for an explicit critical section, and lets the compiler handle the details for us. We won't be using the `reduction` clause here because we want to demonstrate the use of a critical section in this example, a construct commonly used in OpenMP applications.

The Implementation Of The OpenMP Algorithm

The source code of the OpenMP version of function `average` is listed in Figure 13. The OpenMP specific parts are highlighted in bold. The first OpenMP specific part is at lines 4-8. OpenMP provides run time routines to query the parallel environment and change certain settings. For these routines to be known to the compiler, header file *omp.h* needs to be included.

One of the features of OpenMP is that, when care is taken, a program that uses OpenMP can still be compiled, all or in part, without the specific option on the compiler to recognize and translate the directives (like -xopenmp for the Oracle Solaris Studio compilers). This means the program runs sequentially for those parts compiled without the option.

The issue that remains is the run time system, because the OpenMP specific routines are then not known to the system either. Since an OpenMP compliant compiler is guaranteed to set the `_OPENMP` macro, this is tested for in the `#ifdef` construct (lines 4-8). In case the macro is set, file *omp.h* is included at line 5 to define the `omp_get_thread_num()` routine. Otherwise this function is explicitly defined at line 7 to return a value of zero. This is consistent with the value returned for an OpenMP program executed using one thread only. The OpenMP parallel region spans lines 14-27.

Two additional clauses are used on the directive (pragma) for the parallel region. The `default(none)` clause informs the compiler to not apply the default data-sharing attributes. It forces one to explicitly specify whether variables are private or shared. At first sight this may seem unnecessary, as one could rely on the default rules OpenMP defines for this. We strongly recommend not to do so though. Some of the rules are rather subtle and it is easy to make a mistake. It is also good practice to think about the variables and decide whether they should be stored in the private or shared memory.

Since this distinction is a new element compared to serial programming and part of the learning curve for OpenMP, the Oracle Solaris Studio compilers support the unique autoscoping feature. With this, the compiler assists with the assignment of the correct data-sharing attribute(s). An example of this feature can be found in the Autoscoping In OpenMP section on page 37.

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3
 4 #ifdef _OPENMP
 5   #include <omp.h>
 6 #else
 7   #define omp_get_thread_num() 1
 8 #endif
 9
10 double average(int n, double data[])
11 {
12   double sum = 0.0;
13
14   #pragma omp parallel default(none) \
15           shared(n,sum,data)
16   {
17     double Lsum = 0.0;
18     #pragma omp for
19     for (int i=0; i<n; i++)
20         Lsum += data[i];
21
22         printf("\tThread %d: has computed its local sum: %.2f\n",
23                omp_get_thread_num(), Lsum);
24
25     #pragma omp critical
26     {sum += Lsum;}
27   } // End of parallel region
28
29   return(sum/n);
30 }
```

Figure 13. OpenMP implementation of the computation of the average of a set of numbers

The list of shared variables is given at line 15. Although two private variables are used in the parallel region, they do not need to be specified here, since they are local to the code block and therefore automatically privatized. While this seems to violate the recommendation just given about explicit scoping, since the default scoping rules are straightforward and intuitively clear for this case, we can live with this one exception, and rely on the defaults.

When computing its part of the summation, each thread needs to be able read all (in case one thread is used), or a portion of array `data`. This is achieved by declaring the variable `data` to be shared. Since all

threads need to add their partial sum to the final result value `sum`, this variable needs to be accessible by all threads. It is also needed outside the parallel region. This is why it is also made shared.

For `n`, the number of elements in the array, there is a choice. Making it shared is easiest and most efficient in this case, but the `firstprivate` clause in OpenMP provides an alternative. With this attribute, each thread has a local copy of this variable, pre-initialized with the value prior to the parallel region.

All threads execute the declaration and assignment at line 17. From that point on they all have a private copy of variable `Lsum`, initialized to zero.

The `pragma omp for` directive at line 18 informs the compiler that the loop iterations of the for-loop at line 19 can be executed in parallel. When this loop is executed, the OpenMP run time system assigns a subset of the iterations to each thread. Each thread then computes its partial sum and stores it in its private copy of variable `Lsum` (lines 19-20).

For example, if there are 100 iterations to be executed and two threads are used, one thread might sum up the first 50 values, whereas the second thread would do this for the remaining 50 values. We say might, because the distribution of work is left to the compiler to decide upon. Through the `schedule` clause one has explicit control how this should be done, but in this case the default policy, as selected by the compiler, is used. After the for-loop has been executed, each thread has a private copy of `Lsum` containing the partial sum as computed by the thread. This completes step 3 from The Design Of The OpenMP Algorithm section on page 28.

For diagnostic purposes, each thread then prints its unique thread number (as returned by the `omp_get_thread_num()` routine) and the value of `Lsum` (lines 22-23).

The next thing to do is to perform step 4 from The Design Of The OpenMP Algorithm section, to accumulate the partial sums into the final result.

The statement at line 26 performs this computation, but care needs to be taken to do this in the correct way. Since this statement is inside the parallel region, all threads execute it. Without precaution, it might happen that multiple threads simultaneously try to add their contribution stored in `Lsum` to the same (shared) variable `sum`.

This is an example of a data race. A data race occurs in case multiple threads update the same (and therefore shared) address in memory at (potentially) the same time. The behavior is then undefined and leads to silent data corruption, in this case resulting in a wrong value for `sum`.

The Oracle Solaris Studio Thread Analyzer [1] [16] can be used to detect this kind of error, as well as deadlock. To avoid this data race problem, the update needs to be performed such that all threads perform the update at line 26, but only one thread at a time should be allowed to add its value of `Lsum` to variable `sum`.

This is conveniently achieved through the `#pragma omp critical` directive at line 25. The code block enclosed, in this case one statement only, is executed by all threads, but it is guaranteed by the run-time system that only one thread at a time can do so. This is called a *critical section* and is commonly used in shared memory parallel programming to correctly update a shared data structure.

The parallel region ends at line 27. Until the next parallel region is encountered, only the master thread is active. This means that it computes the return value, the average in this case, and exits the function at line 29. Note that a comment string is used at line 27 to mark the end of the parallel region at the curly brace. This is obviously optional, but for readability we highly recommend using a comment here to mark the end of the parallel execution.

Run Time Results For The OpenMP Implementation

The function has been compiled using the Oracle Solaris Studio C compiler:

```
$ cc -c -fast -g -xopenmp -xloopinfo average.c
"average.c", line 19: PARALLELIZED, user pragma used
$
```

The `-xopenmp` option is used to enable recognition of the OpenMP directives within the compiler. This has the effect that the compiler recognizes the specific pragma(s) (otherwise they are ignored) and generates the underlying infrastructure to parallelize the specific construct. In this case a parallel region including a parallel for-loop and a critical region. The diagnostic message from the compiler confirms it recognizes the OpenMP pragma above the for-loop at line 19 and parallelizes the loop.

The other sources are compiled using the same options as before. The link command is also identical:

```
$ cc -c -fast -g main.c
$ cc -c -fast -g check_numerical_result.c
$ cc -c -fast -g -xopenmp get_num_threads.c
$ cc -c -fast -g -xopenmp get_wall_time.c
$ cc -c -fast -g setup_data.c
$ cc -c -fast -g -xopenmp -xloopinfo average.c
"average.c", line 19: PARALLELIZED, user pragma used
$ cc -o main_omp.exe main.o check_numerical_result.o get_num_threads.o
get_wall_time.o setup_data.o average.o -fast -g -xopenmp
```

This program can now be executed in parallel. All one needs to do is to set the OpenMP environment variable `OMP_NUM_THREADS` to the desired number of threads. Below is the output when using one, two and four threads.

```
$ export OMP_NUM_THREADS=1
$ ./main_omp.exe
Please give the number of data points:  100000000
Number of threads used: 1
Thread 0: has computed its local sum: 5000000050000000.00
n = 100000000 average = 50000000.50 (took 1.220 seconds)
$ export OMP_NUM_THREADS=2
$ ./main_omp.exe
Please give the number of data points:  100000000
Number of threads used: 2
Thread 0: has computed its local sum: 1250000025000000.00
Thread 1: has computed its local sum: 3750000025000000.00
n = 100000000 average = 50000000.50 (took 0.610 seconds)
$ export OMP_NUM_THREADS=4
$ ./main_omp.exe
Please give the number of data points:  100000000
Number of threads used: 4
Thread 3: has computed its local sum: 2187500012500000.00
Thread 2: has computed its local sum: 1562500012500000.00
Thread 0: has computed its local sum: 312500012500000.00
Thread 1: has computed its local sum: 937500012500000.00
n = 100000000 average = 50000000.50 (took 0.305 seconds)
$
```

The numerical result is identical for the three runs and equals the exact result.

The local sums are increasing as a function of the thread number. This is because the default work allocation schedule on the Oracle Solaris Studio compilers is the `static` type as defined by the OpenMP specifications. With this schedule, the iterations are equally distributed over the threads, with the first contiguous set of iterations assigned to thread 0, the next set to thread 1, etc. Since the numbers in array `data` increase monotonically, the partial sums are increasing monotonically as well.

The performance is identical to what was measured for the automatically parallelized version. In both cases, a linear speed up is observed. Additional performance results are given in the Performance Results section on page 64.

The Oracle Solaris Studio Compiler Commentary

One of the features of Oracle Solaris Studio is the *Compiler Commentary*. By using the `–g` option (`-g0` for C++), the compiler adds optimization and parallelization information to the object file. This

information is very useful when tuning an application or if one wants to check what kind of optimization(s) the compiler has performed.

The Oracle Solaris Studio Performance Analyzer [18] shows this information by default, but there is also a convenient command line tool called `er_src` to extract the compiler information from the object file and display the annotated source.

Below, the commentary for routine `average`, compiled for automatic parallelization, is shown. The command and the messages specific to the parallelization are highlighted in bold:

```
$ er_src average_apar.o
        1. double average(int n, double data[])
        2. {
             <Function: average>
        3.    double sum = 0.0;
        4.
Source loop below has tag L1
L1 multi-versioned for parallelization. Specialized version is L2
L2 autoparallelized
L2 parallel loop-body code placed in function _$d1A5.average along with 0 inner
loops
L1 scheduled with steady-state cycle count = 5
L1 unrolled 4 times
L1 has 1 loads, 0 stores, 1 prefetches, 1 FPadds, 0 FPmuls, and 0 Fpdivs per
iteration
L1 has 0 int-loads, 0 int-stores, 3 alu-ops, 0 muls, 0 int-divs and 0 shifts per
iteration
L2 scheduled with steady-state cycle count = 5
L2 unrolled 4 times
L2 has 1 loads, 0 stores, 1 prefetches, 1 FPadds, 0 FPmuls, and 0 Fpdivs per
iteration
L2 has 0 int-loads, 0 int-stores, 3 alu-ops, 0 muls, 0 int-divs and 0 shifts per
iteration
        5.    for (int i=0; i<n; i++)
        6.        sum += data[i];
        7.
        8.    return(sum/n);
        9. }
```

The loops, one only in this case, are labeled or *tagged*. The first message indicates the for-loop at line 5 has been given label `L1`.

As explained in the Automatic Parallelization Using The Oracle Solaris Studio C Compiler section on page 25, the compiler generates a parallel, as well as a serial version. At run time it is decided which one to execute. This is an example of the compiler generating multiple versions of a loop and is referred to as *multi-versioned* in the second message. The parallel version of the loop has label `L2`.

The next message refers to a compiler generated function called `_$d1A5.average`. The compiler extracts the body of the parallelized loop, places it in a function, and replaces the loop with a call to a run-time library routine. This process is called *outlining*. This function is then executed in parallel. This is implemented in such a way that each thread works on a subset of the original set of iterations.

The remaining commentary refers to the serial, instruction level, optimizations the compiler has performed. This feature is supported for SPARC processors. Important characteristics regarding the number of load and store instructions, as well as several key floating-point and integer instructions are given. The most important information at this level is contained in the steady-state cycle count. In this case the value is 5. This means that each loop iteration takes five CPU cycles to complete, assuming there are no pipeline stalls. Given the operations performed here, and the architecture of the processor used, this is the optimal number of cycles.

The Compiler Commentary supports OpenMP as well. Through the `-scc` option, the `er_src` command supports filters to select certain types of messages. This feature has been used to select the parallelization messages only.

Below the output for the OpenMP source listed in Figure 13. The command as well as the OpenMP specific compiler information has been highlighted in bold.

```
$ er_src –scc parallel average_omp.o
  1. #include <stdlib.h>
  2. #include <stdio.h>
  3.
  4. #ifdef _OPENMP
  5.   #include <omp.h>
  6. #else
  7.   #define omp_get_thread_num() 1
  8. #endif
  9.
 10. double average(int n, double data[])
 11. {
     <Function: average>
 12.   double sum = 0.0;
 13.
```

```
Source OpenMP region below has tag R1
Private variables in R1: Lsum, i
Shared variables in R1: data, n, sum
 14.   #pragma omp parallel default(none) \
 15.           shared(n,sum,data)
 16.   {
 17.     double Lsum = 0.0;
 Source OpenMP region below has tag R2
 Private variables in R2: i
 18.     #pragma omp for
 Source loop below has tag L1
 L1 parallelized by explicit user directive
 L1 parallel loop-body code placed in function _$d1A18.average
 along with 0 inner loops
 19.     for (int i=0; i<n; i++)
 20.         Lsum += data[i];
 21.
 22.         printf("\tThread %d: has computed its local sum: %.2f\n",
 23.                omp_get_thread_num(), Lsum);
 24.
 25.     #pragma omp critical
 26.     {sum += Lsum;}
 27.   } // End of parallel region
 28.
 29.   return(sum/n);
 30. }
```

Just as with loops, the relevant OpenMP parallel constructs are labeled too. In the first commentary block it is seen that the compiler recognized the pragma defining the parallel region and created parallel region R1. The data-sharing attributes are listed as well.

Since the shared variables have been explicitly listed on the `shared` clause, this should be consistent with what was specified, but the messages also confirm that variables `i` and `Lsum` are automatically made private.

When using the unique autoscoping feature in Oracle Solaris Studio (see also the Autoscoping In OpenMP section), this kind of diagnostic information is also very helpful to check the compiler and understand the decisions it has made regarding the data-sharing attributes.

The next block of commentary shows that the `#pragma omp for` directive has been labeled `R2`. The messages above the for-loop at line 19 confirm the directive was recognized and parallel code for the

loop has been generated using the same outlining technique as used for the automatically parallelized version.

Autoscoping In OpenMP

An important aspect of learning OpenMP is to understand what data-sharing attributes (e.g. private or shared) to assign to the variables in the parallel region. This is something one did not have to think about in the serial version of the program.

Luckily, the Oracle Solaris Studio compilers support a unique extension to OpenMP called *autoscoping*. When using the `default(__auto)` clause, the compiler is requested to perform the dependence analysis and determine the data-sharing attributes of those variables not explicitly specified. The result of this analysis is listed as part of the Compiler Commentary.

The autoscoping feature is demonstrated using the OpenMP version of function `average`. The relevant source code fragment is shown in Figure 14. The only modification, highlighted in bold, is the use of the `default(__auto)` clause on the pragma that defines the parallel region. Note the absence of any data-sharing attributes.

```
#pragma omp parallel default(__auto)
  {
    double Lsum = 0.0;
    #pragma omp for
    for (int i=0; i<n; i++)
        Lsum += data[i];

    printf("\tThread %d: has computed its local sum: %.2f\n",
           omp_get_thread_num(), Lsum);

    #pragma omp critical
    {sum += Lsum;}
  } // End of parallel region
```

Figure 14. An example of the autoscoping feature in the Oracle Solaris Studio compilers

The corresponding part of the commentary is shown below. The information related to the data-sharing attributes is highlighted in bold.

```
Source OpenMP region below has tag R1
Variables autoscoped as SHARED in R1: n, data, sum
Variables autoscoped as PRIVATE in R1: Lsum, i
Private variables in R1: i, Lsum
Shared variables in R1: sum, n, data

14.   #pragma omp parallel default(__auto)
15.   {
16.     double Lsum = 0.0;

Source OpenMP region below has tag R2
Private variables in R2: i
17.     #pragma omp for

L1 parallelized by explicit user directive
L1 parallel loop-body code placed in function _$d1A17.average
   along with 0 inner loops
18.     for (int i=0; i<n; i++)
19.         Lsum += data[i];
20.
21.         printf("\tThread %d: has computed its local sum: %.2f\n",
22.                omp_get_thread_num(), Lsum);
23.
24.     #pragma omp critical
25.     {sum += Lsum;}
26.   } // End of parallel region
```

This output from the `er_src` command confirms the compiler was able to perform the analysis and successfully assign the correct data-sharing attribute to the variables.

The compiler is not always able to assign the data-sharing attributes automatically. It is not always possible to decide from the context what the correct assignment should be, or the analysis is not powerful enough to make a decision. In case this happens, a message is issued and serial code is generated. The message includes the name(s) of the variable(s) that need to be specified explicitly.

Parallelizing The Example Using MPI

In this section, the MPI version of the example is given. First, the steps outlined in the Parallelization Strategy are used to design the MPI version of this algorithm. Several of these steps require special care. After this, the implementation is presented and discussed.

The Design Of The MPI Algorithm

Although not required, it often works easiest to designate one rank to be in charge of the overall execution. If so, rank 0 should be chosen, since it is always guaranteed to be there, even if only one MPI process is launched.

In our case, this approach has been chosen. Rank 0 performs all the set up tasks and assigns work to the other ranks. After the computations have finished, it gathers the partial results and computes the average.

This leads to the following MPI algorithm to parallelize the computations performed in the example program:

1. Rank 0 performs all the set up tasks and defines what chunk of data the other ranks should work on
2. Rank 0 sends to each process the chunk of data to work on, as well the block of data needed
3. Each process receives this information and computes its partial sum
4. Rank 0 gathers all partial sums and adds them to the total sum
5. Rank 0 divides the total sum by the number of elements to obtain the final result

Just as with the OpenMP design in The Design Of The OpenMP Algorithm section, this is not necessarily the most efficient approach, but it allows us to demonstrate various major features very often used in an MPI application.

The Implementation Of The MPI Algorithm

In this section, the MPI version of the computation of the average of a set of numbers is presented and discussed. The steps outlined in the previous section are closely followed, but quite some important details need to be added, causing the source to be rather lengthy. This is why it is discussed in separate blocks, highlighting the most relevant parts in bold. The full source is given in Appendix B.

The first 18 lines are listed in Figure 15. They contain the typical definitions and declarations found in a C program. The inclusion of file *mpi.h* at line 4 is mandatory for any MPI program. For Fortran MPI programs this file is called *mpif.h*.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #include <mpi.h>
 5
 6 double setup_data(int n, double **pdata);
 7 int   check_numerical_result(double avg, double ref_result);
 8
 9 int main (int argc, char **argv)
10 {
11
12   double avg, sum, Lsum, ref_result, *data;
13   double t_start, t_end, t_comp_avg, t_mpi;
14   int    n, irem, nchunk, istart, iend, vlen, ret_code = 0;
15
16   int ier;
17   int me, nproc;
18   int master = 0, msg_tag1 = 1117, msg_tag2 = 2010;
```

Figure 15.Declarations and definitions of the MPI implementation

This program separately measures the time spent in the computational part and the MPI functions. The timings are stored in variables `t_comp_avg` and `t_mpi` respectively (line 13). Although not covered here, the Oracle Solaris Studio Performance Analyzer [18] is a highly recommended tool that provides very detailed timing information for any application. For this simple example it is feasible to implement the timing ourselves, but we strongly encourage using this tool for more comprehensive applications.

The variables highlighted at line 14 are used by rank 0 to determine the workload distribution for all ranks. The variables declared at lines 16-18 are used in the MPI communication functions. To differentiate rank 0 from the other ranks, it is referred to as the master (rank) in the source and in the remainder of this paper. Two variables, `msg_tag1` and `msg_tag2`, are used to contain message tags. The code block in Figure 16 lists the typical set of MPI initialization calls.

```
20   if ( (ier = MPI_Init(&argc, &argv)) != 0 ) {
21     printf("Error in MPI_Init: return code is %d\n",
22            ier); return(ier);
23   }
24
25   if ( (ier = MPI_Comm_size(MPI_COMM_WORLD,&nproc)) !=0 ) {
26     printf("Error in MPI_Comm_size: return code is %d\n",
```

```
27            ier); return(ier);
28    }
29
30    if ( (ier = MPI_Comm_rank(MPI_COMM_WORLD,&me)) !=0 ) {
31      printf("Error in MPI_Comm_rank: return code is %d\n",
32              ier); return(ier);
33    }
```

Figure 16. MPI initialization part

Line 20 shows the call to the `MPI_Init()` function. This is required for every MPI program. It is good practice to check for the return value of the MPI function that is executed. A non-zero value indicates an error has occurred.

At line 25 the `MPI_Comm_size()` function is used to determine how many processes there are.

At line 30 the `MPI_Comm_rank()` function is used to obtain a unique identifier within `MPI_COMM_WORLD` for the process executing this call. The value for this identifier is stored in variable `me`. This name is the same for all processes, but there is no risk of a conflict, as this is a distributed memory programming model. Each process has its own storage location in memory for variable `me`.

The number returned is called the *rank* in MPI. At run time, each process is assigned a unique rank to the `MPI_COMM_WORLD` communicator, which is a consecutive integer number starting from zero.

```
35    if ( me == master ) {
36      printf("Please give the number of data points: ");
37      fflush(stdout);
38      scanf("%d",&n);
39
40      printf("\nThere are %d MPI processes\n",nproc);
41      printf("Number of data points: %d\n",n);
42
43      ref_result = setup_data(n, &data);
44
45      t_start = MPI_Wtime();
46
47      irem   = n%nproc;
48      nchunk = (n-irem)/nproc;
49      for (int p=1; p<nproc; p++)
50      {
51        if (p < irem) {
52          istart = (nchunk+1)*p;
53          iend   = istart + nchunk;
```

```
54       } else {
55         istart = nchunk*p + irem;
56         iend   = istart + nchunk-1;
57       }
58       vlen = iend-istart+1;
59
60       if ( (ier = MPI_Send(&vlen,1, MPI_INT, p, msg_tag1,
61                            MPI_COMM_WORLD)) != 0 ) {
62         printf("Error in MPI_Send: return code is %d\n",
63                ier); return(ier);
64       }
65       if ( (ier = MPI_Send(&data[istart], vlen,
66                   MPI_DOUBLE_PRECISION, p, msg_tag2,
67                   MPI_COMM_WORLD)) != 0 ) {
68         printf("Error in MPI_Send: return code is %d\n",
69                ier); return(ier);
70       }
71     }
72
73     vlen = ( irem > 0 ) ? nchunk+1 : nchunk;
74
75   } else {
```

Figure 17.First part of the work performed by the master

At this point, all processes have performed the necessary set up steps and initial MPI calls, so the computations can start. The first part of this, as performed by the master, is shown in Figure 17.

At line 35 there is a check for this rank to be the master. If so, the user is prompted for input, reading in the number of elements `n`. At line 43 the master calls the function that allocates the memory for array `data` and initializes it. This function is listed and discussed in Appendix A. The exact result is returned by the function and stored in variable `ref_result`.

At line 45 the MPI wall clock timer function `MPI_Wtime()` is called to start the timer.

In the for-loop spanning lines 49-71, the master determines the chunk of iterations for each rank to work on. The logic at lines 51-57 is such that the load is balanced if `n` is not a multiple of the number of ranks.

The number of elements to work on is stored in variable `vlen`. At line 60 this value is sent first to the ranks, so that they can allocate the memory needed to store the block of data they receive next. This data is sent from the master through the second `MPI_Send()` call at line 65.

The `MPI_Send()` function used provides point-to-point communication with a receiving rank. It has 6 arguments.

- When sending a message, a buffer is sent. The address of this buffer, as well as the number of elements and the MPI specific data type need to be specified as the first three arguments respectively. In the call at line 60, the first argument is the address of variable `vlen`, the second argument is `1` because there is only one element to be sent. MPI comes with its own data types, but the user can define application specific data types as well. Since an integer is sent, the `MPI_INT` data type is used as the third parameter in this call. In the second call at line 65, a block of double precision number is sent. The starting address in array `data` is controlled through variable `istart` and depends on the rank the message is sent to. There are `vlen` number of elements to be sent and the data type is `MPI_DOUBLE_PRECISION`.
- The fourth parameter specifies the rank of the destination process. In this case this is rank `p`, ranging from 1 to nproc-1. In other words, the master sends the two messages to all ranks but itself.
- It is possible to send multiple messages to the same receiver. In order for the receiver to distinguish between the various messages, a tag (or label) is used as the fifth argument. In the call at line 60, tag `msg_tag1` is used. The second call at line 65 uses a different tag, stored in variable `msg_tag2`.

  Note that the two different tags are used for demonstration purposes only. In this case it is not really needed to use two different tags, since message ordering is guaranteed to be preserved. By using separate tags and wildcards on the tag, one message can over take another.

  Please note that in general, the key to get good performance and to avoid congesting the underlying message passing engine is to align the send(s) to occur very close to the receive(s).
- The last argument in this call is the default communicator, `MPI_COMM_WORLD`.

At line 73 the master determines its own part of the array to work on. Since it owns all of the data, there is no need to send it.

At line 75 the `else` part begins. This block of code is executed by all other ranks than the master and listed in Figure 18. Line 75 is repeated for clarity.

Each rank receives the two messages. At line 76 the number of iterations to process is received. This number is stored in variable `vlen`. Once this number is known, the rank allocates sufficient memory to store the portion of the data it needs to work on (line 83) and receives the data through the second `MPI_Recv()` call at line 87.

```
75   } else {
76     if ( (ier = MPI_Recv(&vlen, 1, MPI_INT, master,
77                          msg_tag1, MPI_COMM_WORLD,
78                          MPI_STATUS_IGNORE)) != 0 ) {
79       printf("Error in MPI_Recv: return code is %d\n",
80              ier); return(ier);
81     }
82
83     if ((data=(double *)malloc(vlen*sizeof(double)))==NULL){
```

```
84        perror("Fatal error in memory allocation"); exit(-1);
85      }
86
87      if ( (ier = MPI_Recv(data, vlen, MPI_DOUBLE_PRECISION,
88                           master, msg_tag2, MPI_COMM_WORLD,
89                           MPI_STATUS_IGNORE)) != 0 ) {
90        printf("Error in MPI_Recv: return code is %d\n",
91               ier); return(ier);
92      }
93    }
```

Figure 18. Code to receive the number of iterations to work on, allocate memory and receive the data. This part of the program is executed by all ranks, but the master

The `MPI_Recv()` function is the counterpart of the `MPI_Send()` function. It takes 7 arguments.

- The first argument is a pointer to a buffer that should be large enough to store the incoming message. In this case the buffer is `vlen` (line 76) and at line 87 it is the starting address of array data. Note that the names are the same as in the `MPI_Send()` call, but this is not required. Both buffers reside in a different memory than the sender, so there is no risk of a conflict, even if the name is the same. Similar to the `MPI_Send()` call, the second and third argument specify the number of elements and data type of the buffer.
- The fourth argument specifies the rank that has sent the message, in this case the master rank. One can however also use the pre-defined `MPI_ANY_SOURCE` variable. As suggested by the name, this means the receiving rank accepts a matching message tag from any sender. The `MPI_Status` data structure can then be queried to determine the sender.
- The fifth argument contains the message tag, but also in this case, the receive operation can be more flexible. If the pre-defined `MPI_ANY_TAG` variable is used, all messages matching the source specifications are received. The `MPI_Status` data structure can be used to query the system for the details on the tag(s) of the incoming message(s).

  Note that wildcards like `MPI_ANY_SOURCE` and `MPI_ANY_TAG` should be used with care since they can hinder performance as certain optimizations can't be performed then.
- The sixth argument specifies the MPI communicator, `MPI_COMM_WORLD` in this case.
- The last argument contains the MPI status data structure, or as shown in the example, the pre-defined variable `MPI_IGNORE_STATUS`. This variable is used here because there is no need to further query the message passing system. As explained above, in certain cases one may however want to use the status field(s) to obtain more information.

```
 95   if ( me == master ) {
 96     t_end   = MPI_Wtime();
 97     t_mpi   = t_end - t_start;
 98
 99     t_start = t_end;
100   }
101
102   Lsum = 0.0;
103   for (int i=0; i<vlen; i++)
104      Lsum += data[i];
105
106   if ( me == master ) {
107     t_comp_avg = MPI_Wtime() - t_start;
108   }
109
110   printf("MPI process %d has computed its local sum: %.2f\n",
111          me,Lsum);
112
113   if ( me == master ) t_start = MPI_Wtime();
```

Figure 19. Computation of the partial sum

After the second `MPI_Recv()` call at line 87, all ranks know how many loop iterations to work on and they have access to the relevant portion of the data. The code that implements the computation, as well as some calls to the MPI timer function, is listed in Figure 19.

The master is also in charge of the timing mechanism. At line 96, the second call to the `MPI_Wtime()` function is made. The difference between the two timing calls gives the time spent in the first part of the MPI communication. This timing is stored in variable `t_mpi`. Since the computational part is timed as well, this timer is initialized too (line 99).

Lines 102-104 implement the computation of the partial sum. These lines are virtual identical to the original algorithm, be it that at run time, `vlen` is less than n (in case at least 2 processes are used) and for any other rank than the master, array `data` is a subset of the original `data` array.

At lines 106-108 the master records the time again and determines the time its part of the computation took (line 107). The timer for the next portion of MPI calls starts at line 113. This is also done by the master.

This is a rather crude way to measure performance, because it could be that the other ranks need more time to perform their communication and computation. A better approach would be to have all ranks record their own timings and then decide on what the final two timings should be. This is however beyond the scope of this example.

Now that the partial sums have been computed, the things left to do are to gather these contributions, sum them all up, and compute the average by dividing the total sum by the number of elements.

The code that gathers the results and adds them up is listed in Figure 20.

```
115   if ( (ier = MPI_Reduce(&Lsum,&sum,1,MPI_DOUBLE_PRECISION,
116                          MPI_SUM,master,
117                          MPI_COMM_WORLD)) !=0 ) {
118     printf("Error in MPI_Reduce: return code is %d\n",
119            ier); return(ier);
120   }
```

Figure 20. The MPI_Reduce() function is used to gather all partial sums and accumulate them into the global sum. Note that all ranks execute this call.

The `MPI_Reduce()` function is called at line 115. All ranks execute this collective operation. It is a very convenient function to gather data from all ranks and apply a basic or user-defined arithmetic operation to it.

- The first argument is the address of the local variable, `Lsum` in this case, that needs to be sent to one specific rank. This rank is called the `root` and specified as the sixth argument. Here it is the master rank.
- The second argument specifies the address of the variable the final value should be stored into. With this specific function, this value is only available to the root. With the `MPI_Allreduce()` function the final value is distributed to all ranks, but that is not needed here. In our case the master has the final value in variable `sum` and that is all that is needed.
- The third and fourth arguments specify the number of elements sent, only one in this case, and the data type, which is `MPI_DOUBLE_PRECISION`.
- The fifth argument specifies the type of arithmetic operation that should be performed on the incoming data. Since the partial sums need to be summed up, the `MPI_SUM` operator is specified. Several standard basic operators, as well as user-defined functions, are supported.
- The last argument specifies this collective operation should be performed across the entire `MPI_COMM_WORLD` communicator.

At this point, the master has the value of the total sum and can compute the average. The code to do this, as well as print some statistics, is listed in Figure 21. The code executed by all ranks to free up the allocated memory and exit the MPI environment is also included.

The code block at lines 122-139 is executed by the master only. At line 123 the timing for the MPI part is updated. This is followed by the computation of the average at line 125. At line 127, function `check_numerical_result()` is used to check correctness of the result. This function is described in Appendix A. If the result is correct, it is printed together with the timing values. Otherwise an error message is printed.

The code at lines 141-148 is executed by all ranks. Each rank frees the allocated memory (line 141) and calls `MPI_Finalize()` at line 143. This is a mandatory call in an MPI program. After this call, it is illegal to execute any MPI function. This is why the call to `MPI_Finalize()` is often the very last function call in an MPI application.

In case this function call returns a non-zero value, an error message is printed. This value is returned by each rank that experiences this failure. If no error occurred in this function, but the computed result is incorrect, the master rank returns a negative value. The other ranks return a zero value. If no errors occurred and the result is correct, all ranks return a value of zero.

```
122   if ( me == master ) {
123     t_mpi += MPI_Wtime() - t_start;
124
125     avg = sum / n;
126
127     ret_code = check_numerical_result(avg, ref_result);
128
129     if ( ret_code ==  0 ) {
130       printf("n = %d average = %.2f\n",n,avg);
131       printf("Computation: %.3f (s) ",t_comp_avg);
132       printf("MPI communication: %.3f (s) ",t_mpi);
133       printf("Sum: %.3f (s)\n",t_comp_avg+t_mpi);
134     } else {
135       printf("ERROR: COMPUTED RESULT IS WRONG\n");
136       ret_code = -2;
137     }
138
139   }
140
141   free(data);
142
143   if ( (ier = MPI_Finalize()) != 0 ) {
144     printf("Error in MPI_Finalize: return code is %d\n",
145            ier); return(ier);
146   } else {
147     return(ret_code);
148   }
149
150 }
```

Figure 21.Final part of the computation by the master and the exit phase for all ranks

As mentioned earlier, this implementation is not necessarily the most optimal from a performance point of view. By using the asynchronous versions to send (`MPI_Isend()`) and receive (`MPI_Irecv()`) the performance may be improved. Another alternative is to use one-sided communication and send the information directly from the master to the ranks. This saves a receive operation. Both approaches are however beyond the scope of this paper.

Run Time Results For The MPI Implementation

The MPI implementation is integrated into the main program. The source file is called *average_mpi.c*. The program has been compiled and linked using the following command and options:

```
$ mpicc -c -fast -g average_mpi.c
$ mpicc -c -fast -g check_numerical_result.c
$ mpicc -c -fast -g setup_data.c
$ mpicc -o main_mpi.exe average_mpi.o check_numerical_result.o setup_data.o -fast
-g
```

The `mpicc` compiler driver that comes with the Oracle Message Passing Toolkit product has been used. Similar drivers for Fortran (`mpif90`) and C++ (`mpiCC`) are available as well.

We strongly recommend using these drivers because they greatly simplify building an MPI program. Essentially one can use the same Oracle Solaris Studio compile and link options as used in the serial version. The driver handles using the appropriate version of the MPI include file, as well as linking the right MPI libraries.

Similar to the serial and OpenMP version, the `-fast` option on the Oracle Solaris Studio C compiler is used. This invokes a set of powerful optimizations within the compiler.

The `mpirun` command has to be used to execute the MPI program. This command has many options, including ways to specify which node(s) to use, the number of processes per node, and memory/processor affinity. The latter can have a substantial performance benefit on cc-NUMA architectures and is worth considering, but it is outside the scope of this introductory paper.

In this case, only the `-np` option to specify the number of processes has been used. The result for 1, 2 and 4 processes is shown below:

```
$ mpirun –np 1 ./main_mpi.exe
Please give the number of data points: 100000000
There are 1 MPI processes
Number of data points: 100000000
MPI process 0 has computed its local sum: 5000000050000000.00
n = 100000000 average = 50000000.50
Computation: 1.237 (s) MPI communication: 0.000 (s) Sum: 1.237 (s)
```

```
$ mpirun –np 2 ./main_mpi.exe
Please give the number of data points: 100000000
There are 2 MPI processes
Number of data points: 100000000
MPI process 1 has computed its local sum: 3750000025000000.00
MPI process 0 has computed its local sum: 1250000025000000.00
n = 100000000 average = 50000000.50
Computation: 0.619 (s) MPI communication: 0.974 (s) Sum: 1.592 (s)
$ mpirun –np 4 ./main_mpi.exe
Please give the number of data points: 100000000
There are 4 MPI processes
Number of data points: 100000000
MPI process 1 has computed its local sum: 937500012500000.00
MPI process 2 has computed its local sum: 1562500012500000.00
MPI process 0 has computed its local sum: 312500012500000.00
MPI process 3 has computed its local sum: 2187500012500000.00
n = 100000000 average = 50000000.50
Computation: 0.309 (s) MPI communication: 1.314 (s) Sum: 1.624 (s)
$
```

In all three cases the numerical result is in perfect agreement with the exact result. It is no coincidence that the partial sums are the same as the results for the OpenMP version. This is because the workload distribution implemented in the MPI version (see also Figure 17) is identical to what the Oracle Solaris Studio C compiler selects by default on a `for`-loop parallelized with OpenMP. Note that this default could change in the future though.

As is clear from the timings, the computational part scales very well. The performance for a single process is the same as the performance of the serial version and a linear speed up is realized using two and four processes.

The overall performance is however dominated by the cost of executing the MPI functions. This component also increases when adding processes.

The reason this part is dominant is because the computational work is very small relative to the message passing related activities. If more computations would be performed on the data, the cost of MPI would be much less dominant.

```
108   Lsum = 0.0;
109   #pragma omp parallel default(none) \
110               shared(me,vlen,data,Lsum)
111   {
112      #pragma omp single
113      {printf("MPI process %d uses %d OpenMP threads\n",
114              me,omp_get_num_threads());}
115
116       double ThreadSum = 0.0;
117       #pragma omp for
118       for (int i=0; i<vlen; i++)
119           ThreadSum += data[i];
120
121       #pragma omp critical
122       {Lsum += ThreadSum;
123        printf("MPI process %d executes OpenMP thread %d: ",
124                me,omp_get_thread_num());
125        printf("ThreadSum = %.2f\n",ThreadSum);
126       }
127
128   } // End of parallel region
```

Figure 22.Relevant code fragment of the Hybrid implementation using MPI and OpenMP

Parallelizing The Example Using The Hybrid Model

As mentioned in The Implementation Of The MPI Algorithm section, the computational work performed in the MPI implementation has an identical structure as the original, serial, algorithm we started with. But that means it can easily be parallelized further either by using Automatic Parallelization or with OpenMP.

By far the easiest approach is to use Automatic Parallelization by adding the `-xautopar`, `-xreduction` (and `-xloopinfo`) options when compiling source file *average_mpi.c*.

In this section we however want to demonstrate how to use OpenMP to do this explicitly. The relevant source code fragment is listed in Figure 22. This replaces lines 102-104 in Figure 19. The number of source lines added is actually much higher than necessary. This is because diagnostic print statements have been added. If these were left out, there are only nine additional lines needed.

The structure of the new code is identical to the code shown in Figure 13.

The parallel region spans lines 109 to 128. Variable `Lsum` now has the role of variable `sum` in Figure 13 and is therefore a shared variable.

The `pragma omp single` directive at line 112 has not been covered earlier. It defines a *single processor* region and is also used in support function `get_num_threads()` described in Appendix A. The code block enclosed in this region is executed by one thread only. It is typically used for I/O and initializations.

In this case it is used to print the process number of the MPI process executing the parallel region, and the number of OpenMP threads executing this region.

Lines 116-119 are very similar to lines 17-20 in Figure 13. The length of the `for`-loop is now `vlen` and the local variable is called `ThreadSum`. This is the equivalent of variable `Lsum` in Figure 13.

As before, each thread computes its partial sum, now stored in `ThreadSum`. These partial sums then need to be accumulated into the final sum, `Lsum`. This is done at line 122 in the critical section, starting at line 121 and ending at line 126. To demonstrate how this works at run time, a diagnostic message is printed (lines 123-125).

Run Time Results For The Hybrid Implementation

The Hybrid program is compiled and linked as shown below. As can be seen, it is very easy to combine MPI and OpenMP. All that needs to be done is to add the `-xopenmp` option to the compile and link line:

```
$ cc -c -fast -g check_numerical_result.c
$ cc -c -fast -g setup_data.c
$ mpicc -c -fast -g -xopenmp -xloopinfo average_hybrid.c
"average_hybrid.c", line 55: not parallelized, loop has multiple exits
"average_hybrid.c", line 118: PARALLELIZED, user pragma used
$ mpicc -o main_hybrid.exe check_numerical_result.o setup_data.o average_hybrid.o
-fast –g -xopenmp
$
```

Executing the Hybrid program is also very straightforward. Below it is shown how to run the program using four MPI processes and two OpenMP threads per process. Note the use of the `-x` option to export environment variable `OMP_NUM_THREADS` to the other nodes.

```
$ export OMP_NUM_THREADS=2
$ mpirun -np 4 -x OMP_NUM_THREADS ./main_hybrid.exe
Please give the number of data points: 100000000
There are 4 MPI processes
Number of data points: 100000000
MPI process 1 uses 2 OpenMP threads
MPI process 1 executes OpenMP thread 0: ThreadSum = 390625006250000.00
MPI process 1 executes OpenMP thread 1: ThreadSum = 546875006250000.00
```

```
MPI process 1 has computed its local sum: 937500012500000.00
MPI process 2 uses 2 OpenMP threads
MPI process 2 executes OpenMP thread 0: ThreadSum = 703125006250000.00
MPI process 2 executes OpenMP thread 1: ThreadSum = 859375006250000.00
MPI process 2 has computed its local sum: 1562500012500000.00
MPI process 0 uses 2 OpenMP threads
MPI process 3 uses 2 OpenMP threads
MPI process 0 executes OpenMP thread 0: ThreadSum = 78125006250000.00
MPI process 0 executes OpenMP thread 1: ThreadSum = 234375006250000.00
MPI process 0 has computed its local sum: 312500012500000.00
MPI process 3 executes OpenMP thread 0: ThreadSum = 1015625006250000.00
MPI process 3 executes OpenMP thread 1: ThreadSum = 1171875006250000.00
MPI process 3 has computed its local sum: 2187500012500000.00
n = 100000000 average = 50000000.50
Computation: 0.157 (s) MPI communication: 1.323 (s) Sum: 1.480 (s)
$
```

Even though the timing of the computational part includes the overhead of the diagnostic print statements and the single processor region, the performance is still very good when using a total of 4x2 = 8 threads. Compared to the timings for the OpenMP version using four threads, as well as the MPI version using four processes, the time for the computational part is almost cut in half. In other words, near linear scaling is achieved using four MPI processes and two OpenMP threads per process.

The hierarchy in the summation algorithm is nicely demonstrated. For each MPI process, the two OpenMP threads each compute their partial sum. These two sums are then combined to form the partial sum of the MPI process. The master process then collects these four partial sums and computes the final result.

The output of the various MPI processes is not sorted by rank number, but printed in an interleaved way. This is because the order in which the ranks write their output is not determined. If the same job is run again, the output lines might come out in a different order.

This is why the `-output-filename` <*filename*> option on the `mpirun` command is very useful. With this option, all output sent to `stdout`, `stderr`, and `stdiag` is redirected to a rank-unique version of the specified filename. The output file for a specific rank is *filename.*<*ranknumber*>.

For example, the `mpirun` command below produces two output files called *avg.0* and *avg.1* for a hybrid run using two MPI processes and three OpenMP threads per process. Note that standard input is redirected to read from file *INPUT*.

With this option it is much easier to verify and compare the output from each individual rank.

```
$ export OMP_NUM_THREADS=3
$ mpirun -np 2 -x OMP_NUM_THREADS -output-filename avg ./main_hybrid.exe <
INPUT
$ cat avg.0
Please give the number of data points:
There are 2 MPI processes
Number of data points: 100000000
MPI process 0 uses 3 OpenMP threads
MPI process 0 executes OpenMP thread 1: ThreadSum = 416666691666667.00
MPI process 0 executes OpenMP thread 2: ThreadSum = 694444430555555.00
MPI process 0 executes OpenMP thread 0: ThreadSum = 138888902777778.00
MPI process 0 has computed its local sum: 1250000025000000.00
n = 100000000 average = 50000000.50
Computation: 0.210 (s) MPI communication: 0.988 (s) Sum: 1.198 (s)
$ cat avg.1
MPI process 1 uses 3 OpenMP threads
MPI process 1 executes OpenMP thread 0: ThreadSum = 972222252777778.00
MPI process 1 executes OpenMP thread 2: ThreadSum = 1527777730555555.00
MPI process 1 executes OpenMP thread 1: ThreadSum = 1250000041666667.00
MPI process 1 has computed its local sum: 3750000025000000.00
$
```

A Hybrid Fortran Implementation Of The Example Program

In this section it is shown how to use the Oracle Solaris Studio compilers and the Oracle Message Passing Toolkit software to compile, link and run a Hybrid Fortran program. The name of the source file is *avg_hybrid_ftn.f90*. Other than syntactical and some small semantic differences, this version is very similar to the C version. The full source is listed in Appendix C.

This Hybrid Fortran program has been compiled and linked using the `mpif90` compiler driver:

```
$ mpif90 -c -fast -g -xopenmp -xloopinfo avg_hybrid_ftn.f90
"avg_hybrid_ftn.f90", line 61: not parallelized, call may be unsafe
"avg_hybrid_ftn.f90", line 132: PARALLELIZED, user pragma used
"avg_hybrid_ftn.f90", line 212: not parallelized, unsafe dependence
$ mpif90 -o main_hybrid_ftn.exe avg_hybrid_ftn.o -fast -g –xopenmp
$
```

Note that the compile and link options are identical to what is used for the C version.

The same job as for the C version in the Run Time Results For The Hybrid Implementation section has been run, using two MPI processes and three OpenMP threads per process. The `-output-filename` option on the `mpirun` command has been used to obtain individual output files for each rank. The results are shown below.

```
$ export OMP_NUM_THREADS=3
$ mpirun -np 2 -x OMP_NUM_THREADS -output-filename avg.ftn \
./main_hybrid_ftn.exe < INPUT
$ cat avg.ftn.0
Please give the number of data points:
There are   2 MPI processes
Number of data points:  100000000
MPI process   0 uses   3 OpenMP threads
MPI process   0 executes OpenMP thread   1
ThreadSum =  416666691666667.00
MPI process   0 executes OpenMP thread   2
ThreadSum =  694444430555555.00
MPI process   0 executes OpenMP thread   0
ThreadSum =  138888902777778.00
MPI process   0 has computed its local sum:  1250000025000000.00
n =  100000000 average =   50000000.50
Computation: 0.211 (s) MPI communication: 0.982 (s) Sum: 1.192 (s)
$ cat avg.ftn.1
MPI process   1 uses   3 OpenMP threads
MPI process   1 executes OpenMP thread   0
ThreadSum =  972222252777778.00
MPI process   1 executes OpenMP thread   2
ThreadSum =  1527777730555555.0
MPI process   1 executes OpenMP thread   1
ThreadSum =  1250000041666667.0
MPI process   1 has computed its local sum:  3750000025000000.00
$
```

As one would expect, the numerical results are identical to same job using the C version. The timings are about the same as well.

Additional Considerations Important In Parallel Applications

In the Basic Concepts in Parallel Programming section starting at page 6, several key concepts one needs to know before getting started with parallel programming were covered. In this section several additional aspects are presented and discussed. These are mostly of relevance once the parallel application has been designed and implemented.

Parallel Computing, Floating-Point Numbers And Numerical Results

When floating-point numbers are used in add, subtract, multiply or divide operation, round off errors may occur. This is because not all numbers can be represented precisely within the processor and because a finite number of bits are used to contain the number.

A consequence of this is that the order in which the computations on floating-point data are performed affects the numerical result. Manual source code changes, optimizing compilers, as well as parallelization, may all cause such a change in the way the computations are carried out. Even though algebraically equivalent, a different order of the operations could give rise to a difference in the numerical result. The reason for discussing this topic is that certain parallel versions of commonly used computations can be sensitive to round off behavior. This is because the parallel version of the algorithm changes the order in which the computations are performed. Examples are the summation of the elements of an array, or the related computation of the dot product of two vectors.

It is important to realize this is not uniquely related to parallel programming though. It is a consequence of using floating-point arithmetic in a computer and a well-known phenomenon already observed in serial applications. The serial program listed in Figure 23 actually demonstrates how a difference in the order of operations affects the round off behavior. Two different methods are used to sum up the elements of a single precision array called `data`. The numerical results are compared against the exact result.

It should be noted upfront that for two reasons this example presents a worst-case scenario. The relative difference between the smallest and largest number is a monotonically increasing function of `n` and single precision arithmetic is used. This only gives six to seven digits of accuracy at best. If we were to use double precision, the accuracy is 14 to 15 digits. This is why double precision is preferred for computations on large data sets and that is also the data type used in the example in the An Example Program Parallelized Using Various Models section.

We also would like to underline that this example is not about one method being better than another. In general input data is itself subject to error and is usually not sorted in a worst-case order. The only reason we know one is better than another is that we know the correct answer in this example. In a more complicated computation one does not know this, but one can compute bounds on errors using numerical methods. Simple, inexpensive, error bounds generally would show these two methods to be about equally good.

At lines 10-11 the user is prompted to specify the number of array elements. This value is stored in variable `n`. It is assumed `n` is even. This is not an essential restriction, but is imposed to ease coding.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <math.h>
 4
 5 int main()
 6 {
 7   float sum1,sum2,ref_result,rel_err1,rel_err2,*data;
 8   int   n;
 9
10   printf("Please give the value of n (must be even): ");
11   scanf("%d",&n);
12
13   if ( n%2 != 0 ) {
14     printf("Error: n=%d (must be even)\n",n);return(-1);
15   } else {
16     if ( (data=(float *)malloc(n*sizeof(float)))==NULL) {
17       perror("Memory allocation"); return(-1);
18     } else {
19       for (int i=0; i<n; i++) data[i] = i+1;
20       ref_result = 0.5*n*(n+1);
21     }
22   }
23
24   sum1 = 0.0;
25   for (int i=0; i<n; i++)
26      sum1 += data[i];
27   rel_err1 = 100.0*fabs((ref_result-sum1)/ref_result);
28
29   sum2 = 0.0;
30   for (int i=0; i<n/2; i++)
31      sum2 += data[i] + data[i+n/2];
32   rel_err2 = 100.0*fabs((ref_result-sum2)/ref_result);
33
34   printf("exact result = %.1f\n",ref_result);
35   printf("sum1         = %.1f (%.1e%%)\n",sum1,rel_err1);
36   printf("sum2         = %.1f (%.1e%%)\n",sum2,rel_err2);
37   free(data); return(0);
38 }
```

Figure 23. C program to demonstrate the difference in round off behavior

Upon a successful allocation of the memory for array `data`, it is initialized at line 19. Although this is a floating-point array, it contains integer values only. These can be represented precisely in the processor, thereby avoiding a round off error in the initial values.

At line 20 the exact reference result is computed. Given the sequence of numbers, this is easy to calculate since it is the well-known formula for the sum of `n` consecutive positive integer numbers starting with 1. This sum equals n*(n+1)/2.

At lines 24-26 the sum is computed by accessing the array in sequence. With each pass through the loop at line 26, the final result `sum1` is updated with an element of the array. This is the most common and straightforward way to compute the sum of the elements of an array. At line 27 the relative difference with the exact result is computed.

At lines 29-31 a different method is used. With each pass through the loop, half in length compared to the previous loop, a pair of numbers is added to the final result stored in variable `sum2`. The numbers are n/2 elements apart. Also in this case the relative difference with the exact result is computed (line 32).

At lines 34-36 the exact result, both sums, as well as the relative differences are printed.

With both methods one should expect to see differences caused by round off for `n` sufficiently large. Although algebraically equivalent to the first method, the round off behavior is different for the second method, since now a relatively small number is added to a potentially much bigger number. The relative difference between the two numbers depends on `n`. The higher this value is, the bigger the difference. For a high enough value, differences caused by round off will be noticeable.

This is illustrated below. The program has been compiled using the Oracle Solaris Studio C compiler. To avoid compiler transformations affect the numerical results, no optimization options have been specified. The program is executed for n = 100, 1000 and 10000. The numerical results are highlighted in bold.

```
$ cc -c summation.c
$ cc -o summation.exe summation.o -lm
Please give the value of n (must be even): 100
exact result = 5050.0
sum1         = 5050.0 (0.0e+00%)
sum2         = 5050.0 (0.0e+00%)
Please give the value of n (must be even): 1000
exact result = 500500.0
sum1         = 500500.0 (0.0e+00%)
sum2         = 500500.0 (0.0e+00%)
Please give the value of n (must be even): 10000
exact result = 50005000.0
sum1         = 50002896.0 (4.2e-03%)
sum2         = 50003808.0 (2.4e-03%)
```

The results show that both methods produce the exact result for n = 100 and n = 1000, but for n = 10000 this is no longer the case. Both results are different from the exact result and also not consistent.

Clearly, the order in which the elements are summed up affects the numerical result. As demonstrated in the An Example Program Parallelized Using Various Models section, it is fairly easy to parallelize this operation. In order to do this however, the order of summation changes, thereby possible affecting the numerical result.

This is a big topic in itself. For an in-depth coverage of floating-point behavior we highly recommend the extensive *Numerical Computation Guide* [19]. Included in this document is an extremely useful article covering the basics and definitely worth reading [20].

Elapsed Time And CPU Time

Before discussing common types of parallel overhead in more detail, it is needed to elaborate on the difference between the elapsed time and CPU time. The reason is that, compared to the serial version, the CPU time of the parallel version might increase, while the elapsed time decreases.

The elapsed time is the total time the program takes from start to finish. One could measure this with a stopwatch, but luckily it can more easily be obtained from the operating system. The /bin/time command on UNIX based systems can for example be used to obtain this value for any command, or application, executed. It is given as the *real time* in the output. This command also returns the system time and the time spent in the user part of the operating system kernel. This time is given as *user time*, but is often also referred to as the *CPU time*.

This CPU time is a different metric. It is based on the number of processor cycles that are executed. In a serial program, the CPU time is equal to or less than the elapsed time. If for example, the application performs I/O, the processor could be inactive while waiting for the I/O operation to finish. As a result, the cycle counter is not incremented until the processor resumes execution again. The duration of the I/O operation does however contribute to the total execution time and therefore the elapsed time exceeds the CPU time.

In a parallel program, the situation is more complicated. As explained below, various types of overheads caused by the parallel execution may consume additional CPU cycles, negatively impacting the performance. Despite this increase, the elapsed time for the parallel program might still be reduced.

This is easiest illustrated with an example. Assume the serial version of the program takes 100 CPU seconds and that it equals the elapsed time. In the ideal case, using 4 cores, the parallel version of the program takes 100/4 = 25 CPU seconds. In the absence of additional overheads, the elapsed time would therefore be 25 seconds, but what if each core needs 1 CPU second longer? This could for example be to execute library calls that have been introduced as part of the parallelization. The elapsed time will then be 26 seconds. This is still significantly faster than the serial version, but the total CPU time is now 100+4*1 = 104 seconds, which is higher compared to the serial version.

Parallel Overheads

While running an application over multiple processors and threads should theoretically improve performance compared to a single processor or thread, how things really work out in practice will depend on several important factors that are worth considering.

To start with, be aware that the underlying framework of a parallel program is going to be different than the original serial version. This is because there are additional activities to support the parallel execution, and therefore more work has to be performed. A major consideration while developing an efficient parallel program is to minimize this extra work. For example, there has to be a mechanism that creates, executes and synchronizes the threads, and there is a cost associated with this thread management. But there are additional factors to be considered that can negatively impact the performance.

In most parallel applications, threads will need to exchange information, and the cost associated with these exchanges is usually referred to as the *communication overhead*.

Performance can suffer if the workload over the threads is not balanced appropriately. A barrier is a synchronization point in the program where all threads wait until the last one arrives. If not all threads take the same amount of time to perform their work, there will be an additional delay while threads wait in the barrier. This is called a *load imbalance* and it might have a noticeable negative impact on the performance.

We say "might" here because the impact depends on various factors, such as the severity of the delay, the details of the implementation of the barrier, the number of threads/processes, the system interconnect characteristics, the operating system and how idle threads are handled, and the cleverness of the compiler.

Instead of actively waiting (also called *spinning*), idle threads may be put to sleep and woken up when needed again. Although this strategy reduces the number of processor cycles executed while waiting, it is not without a cost either. Waking up the idle thread(s) again takes cycles too. The selection of the best strategy often requires some experimentation with the various choices that might be available.

All of the above factors cost extra time compared to the serial version, and collectively they are called the *parallel overheads*.

There are some applications where the overhead costs are minimal, relative to the total execution time. Such programs are called *embarrassingly parallel* because their performance tends to benefit well running on a system with a very large number of cores and/or configured with a low cost interconnect. This is covered in more detail in the Parallel Speed Up And Parallel Efficiency section on page 60.

However, such extremely parallel applications are more the exception than the rule. For most applications, the parallel overheads need to be taken into account and minimized as much as possible. How to do this not only depends on the parallel programming model, but also on the system software and hardware. Regarding the latter, not only the processor used, but also the characteristics of the internal interconnect that combines the processors, memory and I/O subsystem(s) into a parallel system, as well as the network connecting the systems in case of a distributed memory architecture, might play a significant role.

Although not covered here, the Oracle Solaris Studio Performance Analyzer [18] that is part of Oracle Solaris Studio is an excellent tool to identify these kinds of overheads in an application.

Parallel Speed Up And Parallel Efficiency

How can we determine how well a parallel program will perform and what performance to expect when increasing the number of threads? The parallel speed up S_p of an application is a reasonable and easy value to compute for any parallel program. It is defined as

$$S_p := T_1 / T_p$$

Here, T_p denotes the elapsed time measured using "p" threads. T_1 is the elapsed time measured using one thread only to execute the parallel program. Defined this way, the parallel speed up measures how the parallel overhead affects the performance when increasing the number of threads.

In the ideal case, $T_p = T_1 / p$ and $S_p = p$. This means a perfect speed up was realized. If p is very large, the application is said to scale very well. For embarrassingly parallel applications, the speed up will stay very close to p, even for a high number of threads.

More typically, a perfect speed up may be hard to achieve. Even if the initial speed up is (close to) perfect with a small number of threads, the parallel overhead could start to dominate as the number of threads increases. While the elapsed time might continue to decrease, the speed up will be less and less.

In some cases we might even see a *super-linear* speed up however. This would be a nice bonus, because in this case the parallel performance exceeds what could be expected based on the number of threads used. For example, the elapsed time using 4 threads could be 1/5, not 1/4, of the single threaded time.

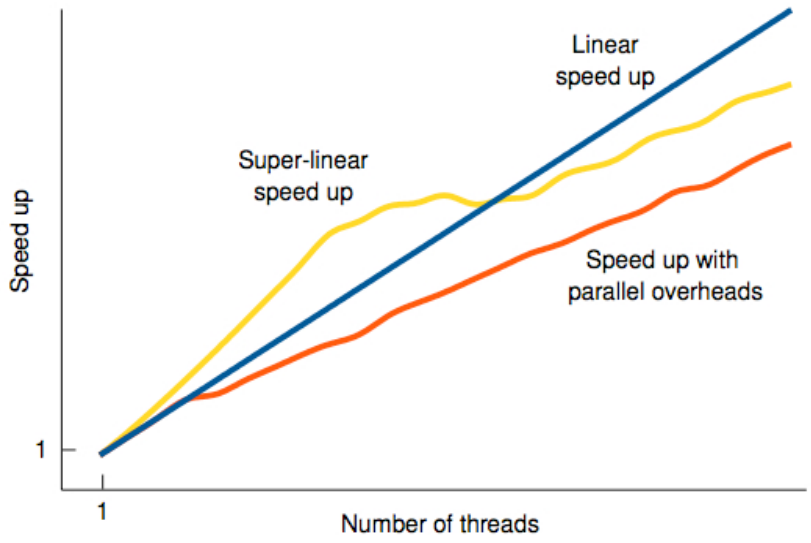

Figure 24.Perfect, non-perfect and super-linear speed up

The explanation is that adding more cores to participate in the computation means typically more cache memory can be used, and then more data might be loaded from the faster cache instead of main memory. So, adding more hardware might improve performance not just with better CPU utilization, but better memory utilization as well. However, often this behavior can not be sustained, because eventually there are too many threads, the parallel overhead starts to dominate, and the super-linear speed up is no longer realized. All three cases are illustrated in Figure 24.

Related to S_p, the parallel efficiency is defined as $E_p := S_p / p$. This metric is directly coupled to the speed up. In the ideal case, $E_p := S_p / p = p/p = 1 = 100\%$. In case the speed up is not perfect, E_p will be less than 100%. In case of a super-linear speed up it exceeds 100%.

There is one major caveat with both S_p and E_p. Because they are relative metrics, they do not provide any information on the absolute execution time. In other words, a parallel algorithm with a very good, or even linear, speed up could still be slower compared to an alternative algorithm with a lesser speed up.

There can be several reasons for this. In case of an iterative solver for example, the algorithm with the lower speed up may require fewer iterations to converge to the same solution. Or the serial performance could be higher to start with.

For example, if a certain algorithm has two different implementations, say A and B, the following might occur. Let's assume $T_1 = 100$ seconds for implementation A and $T_1 = 80$ seconds for implementation B, because it uses a more efficient approach. If $T_2 = 55$ seconds for A and $T_2 = 47$ seconds for B, $S_2(A) = 100/55 = 1.82$ and $S_2(B) = 80/47 = 1.70$. Even though the speed up for implementation B is less than for A, the performance is still 17% higher.

In some cases, the elapsed time of the *serial* version is used for T_1 in the definition of the speed up. In the absence of parallel overhead this time is typically less than the elapsed time of the parallel version using one thread only. As a result, the ratio T_1 / T_p is lower in case the serial time is used as a reference.

The motivation to use this value for T_1 is that it reflects how much performance has been gained compared to the serial version. It is meaningful to also measure this, since reducing the serial time is after all the goal of parallel computing.

Amdahl's Law

Although derived from a very simple performance model, this law is surprisingly useful. It gives a first order approximation of the parallel performance that can be expected when adding threads.

Amdahl's law is very simple, and makes the observation that in a running program, some of the program's code will run serially and some in parallel. To reflect this, the single core execution time T_1 is decomposed into 2 components: T_1 = T(sequential) + T(parallel). By definition, the sequential (or non-parallel) time is the part of the application that does not benefit from parallel execution (yet). The second component represents the execution time that could be parallelized.

A fraction f is introduced to describe both components in terms of T_1. The non-parallel part is given by $(1-f)*T_1$, whereas the parallel part is given by $f*T_1$. Clearly, "f" can be anything between 0 and 1, in other words: $f \in [0,1]$. Both end points of this interval are literally corner cases. A value of f = 0 means that none of the execution time benefits from parallelization. If f = 1, the program is embarrassingly parallel.

Using "p" threads, the parallel version has an execution time of

$$T_p = T(\text{serial}) + T(\text{parallel})/p = (1-f)*T_1 + f*T_1/p$$

Note that this is a simplification, because in reality a component describing the parallel overhead should be added to T_p. With Amdahl's law this is not counted, although it should be.

As a result, and with the exception of super-linear scaling, any projections made using Amdahl's law are best case results. As we will see shortly, the implications of this law are already profound enough, even without including the parallel overhead. The parallel speed up is then given by

$$S_p(f) = T_1/T_p = 1/(1-f + f/p) \qquad \text{(Amdahl's Law)}$$

This formula also illustrates the comment made in the previous section: the speed up does not depend on T_1 and can therefore not be used to assert and compare the absolute performance.

For the two corner cases, f = 0 and f = 1, the formula for $S_p(f)$ is rather simple: $S_p(0) = 1$ and $S_p(1) = p$. This obviously corresponds to what one would expect. If f = 0, the program does not benefit from parallel execution and the speed up can never be higher than 1. In case f = 1, the program is embarrassingly parallel and a perfect, linear, speed up is always realized. In most of the cases, f will be somewhere between 0 and 1. Unfortunately, the resulting curve is rather non-linear, as demonstrated in Figure 25. In this chart it is assumed the system has 16 cores.

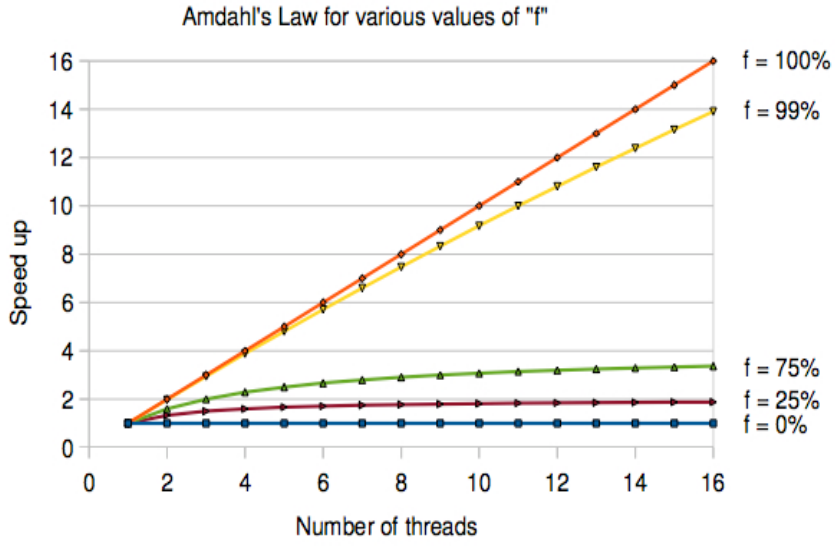

Figure 25. Amdahl's law on a 16 core system for various values of "f". Only if a significant portion of the execution time has been parallelized, near linear scaling is realized

The consequences are rather dramatic. Even if f = 99%, the parallel speed up using 16 threads is not 16, but 14. This is not a bad result, but it is not a perfect speed up — despite the fact that 99% of the original execution time benefits from parallel execution. For lower values of f, the situation is worse. If f = 25%, the parallel speed up will never exceed 4, regardless how many threads are used. Remember that this is the optimistic estimate, because the assumption is that there is no parallel overhead.

There is however also something encouraging. Even for a fairly low value of "f", there is a speed up when using a few threads only. This is also observed in practice: it is often not so hard to get a parallel speed up using just a few threads. Amdahl's law tells us what to do in order to further improve the speed up.

It is possible to use the formulas given above to estimate "f" for a parallel application. By rewriting the formula for S_p we can express "f" in terms of S_p and "p":

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

Using this estimate for "f", it is then possible to predict the speed up using any number of threads. All one has to do is conduct two performance experiments. For example, assume T_1 = 100 seconds and the elapsed time T_4 using 4 threads is 37 seconds. The parallel speed up is then $S_4 = 100/37 = 2.70$. Substituting this in the formula for "f" results in f = (1-1/2.70)/(1-1/4) = 84%.

Using this value for "f", the following estimate for the parallel speed up using 8 threads is obtained: S_8 = 1/(1-0.84+0.84/8) = 3.77. It is also easy to estimate the expected execution time: T_8 = (0.84/8+1-0.84)*100 = 26.5 seconds.

There are two ways to view these results. A speed up of 3.77 using 8 threads may be disappointing, but the program is still ~28% faster compared to using 4 threads. As the cost of cores decreases, inefficient use of some cores becomes less of a problem. The time to solution is what matters and the fact that not all 8 cores are utilized very efficiently when using 8 threads may therefore be less of a concern.

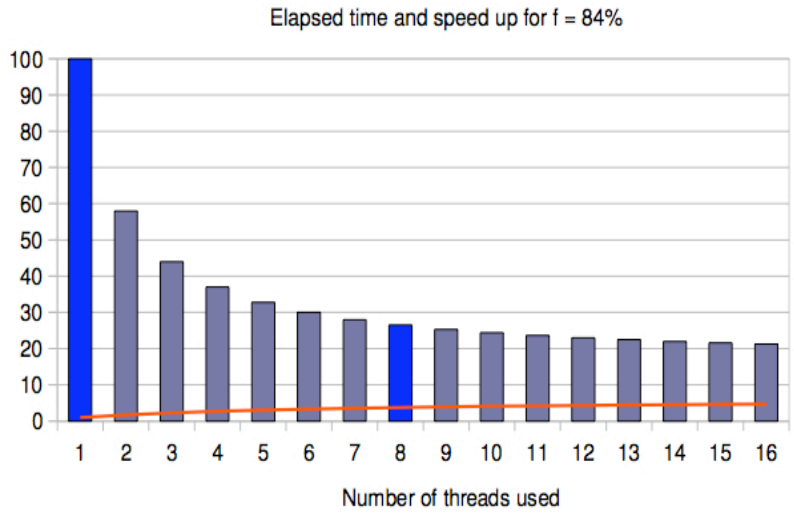

Figure 26.An example of Amdahl's Law. The bar chart shows the estimated elapsed time as a function of the number of threads used. The solid line is the parallel speed up. The measured timings for 1 and 8 threads are highlighted

This is illustrated in Figure 25. Assuming f = 84%, the estimated elapsed time and speed up are shown for up to 16 threads. Clearly there is a diminishing benefit when adding threads. On the other hand, the performance is improved by a factor of almost 5 over the serial performance.

Performance Results

Several performance results were given in the An Example Program Parallelized Using Various Models section. The timings were very encouraging. In this section more performance data is presented and discussed. The results were obtained on a Sun SPARC Enterprise® T5120 server from Oracle. The system had a single UltraSPARC® T2 processor with 8 cores and 8 hardware threads per core.

In Figure 27 the elapsed times in seconds for the Automatically Parallelized and OpenMP implementations are plotted as a function of the number of threads used. Since the single thread time is the same for both versions, it is easy to compare these timings against perfect, linear, scaling. This is the line labeled *Linear Scaling* in the chart. Note that a log scale is used on the vertical axis.

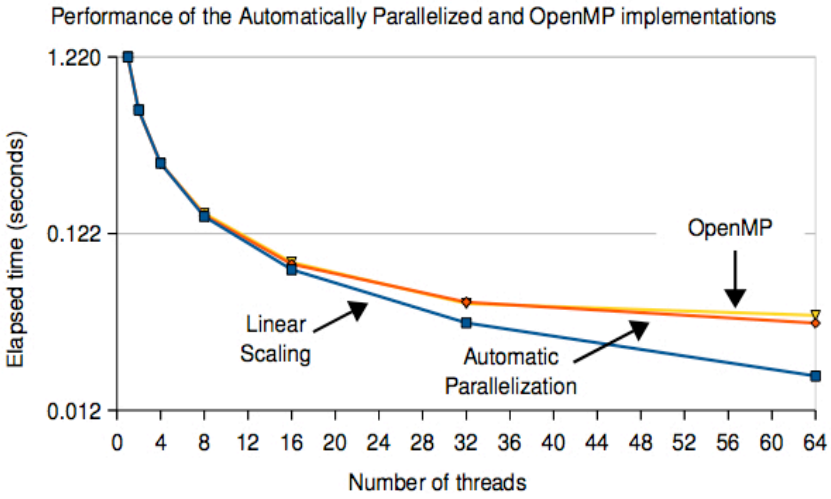

Figure 27. Performance of the Automatically Parallelized and OpenMP implementations. A curve with perfect, linear, scaling has been added for comparison purposes. A log scale is used on the vertical axis

For up to 32 threads, both versions perform equal. For 64 threads the Automatically Parallelized version performs about 9% faster than the OpenMP version. This is not really a surprise since in The Design Of The OpenMP Algorithm section it was already noted that the latter is sub-optimal because the reduction clause is not used here. If this were the case, the performance difference might be less, but in any case the difference is not that big.

Both versions scale very well for up to 16 threads. When using 32 threads, the performance deviation compared to linear scaling is about 30% for both. For 64 threads, the elapsed time is about twice as high. This difference is caused by the parallel overheads increasing as more threads are used. If more computational work was performed, this overhead would not be as dominant.

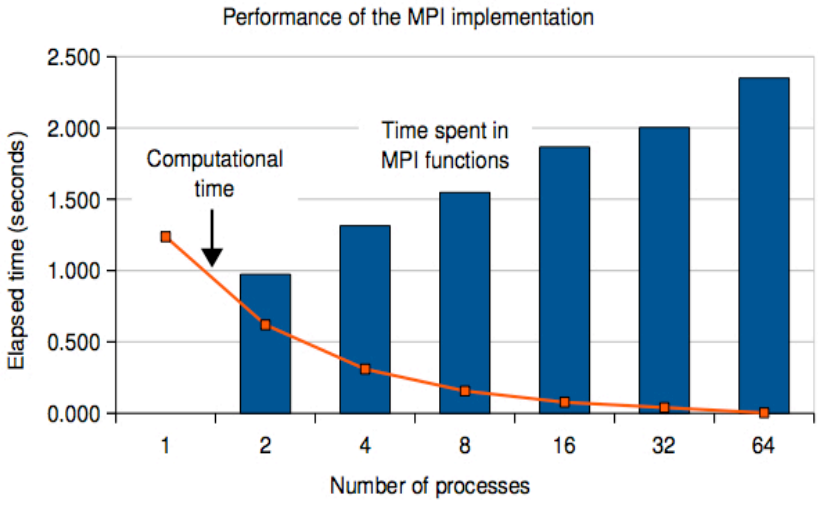

Figure 28. Performance of the MPI implementation. Both the timings for the computational part (solid line) as well as the time spent in the MPI functions (bar chart) are shown

In Figure 28 the performance of the MPI implementation is plotted as a function of the number of processes. Both the computational time (solid line) as well as the total time spent in the MPI functions (bar chart) are shown. The total execution time for the algorithm is the sum of these two numbers.

The computational part scales linearly up to 64 threads. The elapsed time using a single thread is 1.237 seconds. On 64 threads this is reduced to 0.019 seconds, giving rise to even a slightly super linear speed up of 1.237/0.019 = 65.1. The time spent in the MPI functions is obviously zero if only one process is used. When using two processes it is just below one second, going up to 2.4 seconds on 64 threads.

The computational work is very small. As a result, the cost of message passing is relatively dominant and no overall performance gain is achieved when running in parallel. If more work were performed, this would be different. On the other hand, the two shared memory versions scale very well for up to 16 threads, while the performance using 32 threads is quite acceptable. This opens up opportunities for the Hybrid version to deliver the best of both worlds. In Table 1 this is illustrated using 2 MPI processes with 16 and 32 OpenMP threads. These results were obtained with a version of the Hybrid implementation that does not have the diagnostic print statements in the OpenMP part.

TABLE 1. SOME PERFORMANCE NUMBERS FOR THE HYBRID IMPLEMENTATION

MPI PROCESSES	OPENMP THREADS	COMPUTATIONAL TIME (SECONDS)	MPI TIME (SECONDS)	SUM (SECONDS)
2	16	0.307	0.998	1.295
2	32	0.155	0.984	1.139

Although the total time is still greater than for the shared memory versions, the results for the Hybrid implementation are encouraging. The computational time scales well and since the number of MPI processes is fixed, the time spent in the MPI part does not increase.

Conclusion

The goal of parallel computing is to reduce the elapsed time of an application. To this end, multiple processors, or cores, are used to execute the application.

The program needs to be adapted to execute in parallel and take advantage of the availability of multiple processors. To achieve this, a single, serial, application needs to be split into independent pieces of work that can be executed simultaneously. Threads execute the resulting independent execution streams. A single thread executes those parts of the program that cannot be broken up for parallel execution.

Amdahl's Law is something every developer needs to be aware of. This law can be used to predict the parallel performance of an application. The expected speed up depends on the number of threads used, but also on the fraction of the execution time that can be parallelized. Only if this fraction is very high, scalable performance to a very high number of cores can be expected.

A wide range of parallel architectures is available these days. Thanks to the multicore technology, even a modern laptop is a small parallel system. On the server side, various small to very large shared memory systems are on the market today. For those applications that need more, systems can be clustered through a high-speed network to form an even bigger parallel computer.

Since the software needs to be adapted to take advantage of these parallel architectures, a suitable parallel programming model needs to be selected to implement the parallelism. Numerous choices are available, each with their pros and cons.

Automatic Parallelization is a feature available on the Oracle Solaris Studio compilers. Through an option, the user activates this loop-based mechanism in the compiler to identify those loops that can be executed in parallel. In case the dependence analysis proves it is safe to do so, the compiler generates the parallel code. All the user needs to do is to set an environment variable to specify the number of threads prior to running the application.

OpenMP is a de-facto standard to explicitly implement parallelism. Like Automatic Parallelization, it is suitable for multicore and bigger types of shared memory systems. It is a directive based model, augmented with run time functions and environment variables. The Oracle Solaris Studio compilers fully support OpenMP, as well as additional features to assist with the development of applications using this programming model.

In case the target architecture is a cluster of systems, the Message Passing Interface, or MPI for short, is a very suitable model to implement this type of parallelism in an application. MPI consists of an extensive set of functions, covering a wide range of functionality. The Oracle Message Passing Toolkit product offers a very mature and efficient MPI implementation.

Contemporary cluster architectures provide for two levels of parallelism as each node in the cluster consists of a multicore system. This makes the Hybrid model a very natural choice, since it combines MPI and a shared memory model, nowadays most often OpenMP. MPI is used to distribute the work over the nodes, as well as handle the communication between the nodes. More fine-grained portions of

work are then further parallelized using Automatic Parallelization and/or OpenMP. Together with the Oracle Message Passing Toolkit, the Oracle Solaris Studio compilers can be used to develop and deploy these kinds of applications.

After having selected a parallel programming model, the developer is still faced with several additional challenges when considering parallelizing an application. The parallelism not only needs to be identified in the application, but in view of parallel overheads and Amdahl's law, one has to carefully avoid performance pitfalls. The specifics depend on the programming model chosen, and to a certain extent also on the computer system(s) used

References

[1] Oracle Solaris Studio Home Page, http://developers.sun.com/sunstudio

[2] Oracle Solaris Studio Downloads, http://developers.sun.com/sunstudio/downloads

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

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

[5] OpenMP Home Page, http://www.openmp.org

[6] MPI Home Page, http://www.mcs.anl.gov/research/projects/mpi

[7] John L. Hennessy, David A. Patterson, "Computer Architecture: A Quantitative Approach, 4th Edition", Morgan Kaufmann Publishers, ISBN 10: 0-12-370490-1 ISBN 13: 978-0-12-370490-0, 2007.

[8] Ruud van der Pas, "Memory Hierarchy in Cache-Based Systems", http://www.sun.com/blueprints/1102/817-0742.pdf, 2002.

[9] Darryl Gove, "Solaris Application Programming", Prentice Hall, ISBN 0138134553, 2008.

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

[11] OpenMP Forum, http://openmp.org/forum

[12] Barbara Chapman, Gabriele Jost, Ruud van der Pas, "Using OpenMP", The MIT Press, ISBN-10: 0-262- 53302-2, ISBN-13: 978-0-262-53302-7, 2008.

[13] William Gropp, Ewing Lusk, Anthony Skjellum, "Using MPI, 2nd Edition", MIT Press, ISBN-10:0-262-57132- 3 ISBN-13:978-0-262-57132-6, 1999.

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

[15] Darryl Gove, "Using Sun Studio to Develop Parallel Applications", Oracle Solaris Studio Whitepaper, 2008.

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

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

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

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

[20] David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", Association for Computing Machinery, Inc, 1991.

Appendix A - Additional Source Codes

The focus of the example program in the An Example Program Parallelized Using Various Models section is on the computation of the average, but in order to compile and run the test cases, several support functions were used. In this appendix the sources of these functions are given and discussed briefly.

The check_numerical result() function

In the Parallel Computing, Floating-Point Numbers And Numerical Results section it was shown that the order in which the numbers are summed may impact the final result. This is true for the serial version, but also for the various parallel versions discussed in the An Example Program Parallelized Using Various Models section, since each thread or MPI process only sums up the elements of a subset of array data. From a numerical point of view this is different that the original serial order.

This is why the check for correctness has a tolerance factor to take into account some loss of precision may occur due to differences in round off behavior. The source code of the function that performs this check is shown below.

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <math.h>
 4 #include <float.h>
 5
 6 int check_numerical_result(double avg, double ref_result)
 7 {
 8
 9  double tolerance = 100.0 * DBL_EPSILON;
10  double rel_error;
11  int    ret_code = 0;
12
13  if ( fabs(ref_result) > DBL_MIN ) {
14    rel_error = fabs( (avg - ref_result)/ref_result );
15  } else {
16    rel_error = fabs( avg - ref_result );
17  }
18
19  if ( rel_error > tolerance ) {
20    printf("\nFATAL ERROR: computed  result = %f\n",avg);
21    printf("     reference result = %f\n",ref_result);
22    printf("     relative error   = %e%%\n",100.0*rel_error);
```

```
23    ret_code = -1;
24  }
25
26  return(ret_code);
27 }
```

At line 9 the tolerance is set using the system provided relative precision value `DBL_EPSILON` for double precision numbers. By multiplying this number by 100, small relative differences are still accepted.

At lines 13-17 the relative difference is computed. If the reference value is too small, as compared to the smallest positive double precision number, the absolute error is computed. Otherwise the relative error is used. In case this error exceeds the tolerance value a diagnostic message is printed and a non-zero value is returned to the calling program. In case the computed results meets the criteria, a value of zero is returned.

The get_num_threads() Function

In the serial, automatically parallelized and OpenMP versions of the example a function called `get_num_threads()` is used. It returns the number of threads currently set. The source code is shown below.

This function is written such that it compiles without OpenMP as well. This is achieved by the `#ifdef` construct at lines 1-5. In case OpenMP is enabled in the compiler, the `_OPENMP` macro is set and file *omp.h* needs to be included, because one of the OpenMP run time routines is used. If the macro is not set the `omp_get_num_threads()` routine is defined to return 1.

The advantage of setting it up this way is that the source can now also be compiled without enabling OpenMP in the compiler. Or in case a compiler is used that does not support OpenMP at all, but that is very rare these days.

The OpenMP parallel region spans lines 11-15. The `#pragma omp single` directive has the effect that only one thread executes the enclosed block of code. In this case it is one statement only, using the `omp_get_num_threads()` routine that returns the number of threads active in the parallel region. The value is stored in variable `nthreads`, which is then returned by the function.

```
 1 #ifdef _OPENMP
 2   #include <omp.h>
 3 #else
 4   #define omp_get_num_threads()  1
 5 #endif
 6
 7 int get_num_threads()
 8 {
 9   int nthreads;
10
11   #pragma omp parallel
12   {
13       #pragma omp single nowait
14       {nthreads = omp_get_num_threads();}
15   } // End of parallel region
16   return(nthreads);
17 }
```

The get_wall_time() Function

In the serial, automatically parallelized and OpenMP versions of the example a function called `get_wall_time()` is used. It returns the absolute elapsed time in seconds. The source code is shown below.

This function uses the same `#ifdef` construct as function `get_num_threads()` to ensure the source code also compiles in case OpenMP is not used. In this case a value of zero is returned, but that is a placeholder only. Most likely one would like to substitute an alternate timer like `gettimeofday` in such a case. At line 10 the OpenMP routine `omp_get_wtime()` is used to return the elapsed time since some moment in the past.

```
 1 #ifdef _OPENMP
 2   #include <omp.h>
 3 #else
 4   #define omp_get_wtime()  0
 5 #endif
 6
 7 double get_wall_time()
 8 {
 9   return(omp_get_wtime());
10 }
```

The setup_data() Function

This function allocates the memory for array `data`, initializes this array and computes the exact result. The source code is listed below.

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3
 4 double setup_data(int n, double **pdata)
 5 {
 6   double *data, ref_result;
 7
 8   if ( (data=(double *) malloc(n*sizeof(double)))==NULL) {
 9     perror("Fatal error in memory allocation"); exit(-1);
10   } else {
11     for (int i=0; i<n; i++)
12        data[i] = i+1;
13     ref_result = 0.5*(n+1);
14   }
15
16   *pdata = data;
17
18   return(ref_result);
19 }
```

At line 8 a pointer to a block of memory is returned. Upon a successful allocation, the array is initialized (lines 11-12) and the exact result is computed at line 13. This is also the return value of this function. At line 16 the pointer is stored in `pdata` so that the calling function can get the address of the memory block.

Appendix B - Full C Source Code Of The MPI Implementation

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #include <mpi.h>
 5
 6 double setup_data(int n, double **pdata);
 7 int    check_numerical_result(double avg, double ref_result);
 8
 9 int main (int argc, char **argv)
10 {
11
12   double avg, sum, Lsum, ref_result, *data;
13   double t_start, t_end, t_comp_avg, t_mpi;
14   int    n, irem, nchunk, istart, iend, vlen, ret_code = 0;
15
16   int ier;
17   int me, nproc;
18   int master = 0, msg_tag1 = 1117, msg_tag2 = 2009;
19
20   if ( (ier = MPI_Init(&argc, &argv)) != 0 ) {
21     printf("Error in MPI_Init: return code is %d\n",
22            ier); return(ier);
23   }
24
25   if ( (ier = MPI_Comm_size(MPI_COMM_WORLD,&nproc)) !=0 ) {
26     printf("Error in MPI_Comm_size: return code is %d\n",
27            ier); return(ier);
28   }
29
30   if ( (ier = MPI_Comm_rank(MPI_COMM_WORLD,&me)) !=0 ) {
31     printf("Error in MPI_Comm_rank: return code is %d\n",
32             ier); return(ier);
33   }
34
35   if ( me == master ) {
```

```
36     printf("Please give the number of data points: ");
37     fflush(stdout);
38     scanf("%d",&n);
39
40     printf("\nThere are %d MPI processes\n",nproc);
41     printf("Number of data points: %d\n",n);
42
43     ref_result = setup_data(n, &data);
44
45     t_start = MPI_Wtime();
46
47     irem   = n%nproc;
48     nchunk = (n-irem)/nproc;
49     for (int p=1; p<nproc; p++)
50     {
51       if (p < irem) {
52         istart = (nchunk+1)*p;
53         iend   = istart + nchunk;
54       } else {
55         istart = nchunk*p + irem;
56         iend   = istart + nchunk-1;
57       }
58       vlen = iend-istart+1;
59
60       if ( (ier = MPI_Send(&vlen,1, MPI_INT, p, msg_tag1,
61                            MPI_COMM_WORLD)) != 0 ) {
62         printf("Error in MPI_Send: return code is %d\n",
63                ier); return(ier);
64       }
65       if ( (ier = MPI_Send(&data[istart], vlen,
66                   MPI_DOUBLE_PRECISION, p, msg_tag2,
67                   MPI_COMM_WORLD)) != 0 ) {
68         printf("Error in MPI_Send: return code is %d\n",
69                ier); return(ier);
70       }
71     }
72
73     vlen = ( irem > 0 ) ? nchunk+1 : nchunk;
74
```

```
75   } else {
76     if ( (ier = MPI_Recv(&vlen, 1, MPI_INT, master,
77                          msg_tag1, MPI_COMM_WORLD,
78                          MPI_STATUS_IGNORE)) != 0 ) {
79       printf("Error in MPI_Recv: return code is %d\n",
80              ier); return(ier);
81     }
82
83     if ((data=(double *)malloc(vlen*sizeof(double)))==NULL){
84       perror("Fatal error in memory allocation"); exit(-1);
85     }
86
87     if ( (ier = MPI_Recv(data, vlen, MPI_DOUBLE_PRECISION,
88                          master, msg_tag2, MPI_COMM_WORLD,
89                          MPI_STATUS_IGNORE)) != 0 ) {
90       printf("Error in MPI_Recv: return code is %d\n",
91              ier); return(ier);
92     }
93   }
94
95   if ( me == master ) {
96     t_end   = MPI_Wtime();
97     t_mpi   = t_end - t_start;
98
99     t_start = t_end;
100  }
101
102  Lsum = 0.0;
103  for (int i=0; i<vlen; i++)
104     Lsum += data[i];
105
106  if ( me == master ) {
107    t_comp_avg = MPI_Wtime() - t_start;
108  }
109
110     printf("MPI process %d has computed its local sum: %.2f\n",
111            me,Lsum);
112
113  if ( me == master ) t_start = MPI_Wtime();
```

```
114
115   if ( (ier = MPI_Reduce(&Lsum,&sum,1,MPI_DOUBLE_PRECISION,
116                          MPI_SUM,master,
117                          MPI_COMM_WORLD)) !=0 ) {
118     printf("Error in MPI_Reduce: return code is %d\n",
119            ier); return(ier);
120   }
121
122   if ( me == master ) {
123     t_mpi += MPI_Wtime() - t_start;
124
125     avg = sum / n;
126
127     ret_code = check_numerical_result(avg, ref_result);
128
129     if ( ret_code ==  0 ) {
130       printf("n = %d average = %.2f\n",n,avg);
131       printf("Computation: %.3f (s) ",t_comp_avg);
132       printf("MPI communication: %.3f (s) ",t_mpi);
133       printf("Sum: %.3f (s)\n",t_comp_avg+t_mpi);
134     } else {
135       printf("ERROR: COMPUTED RESULT IS WRONG\n");
136       ret_code = -2;
137     }
138
139   }
140
141   free(data);
142
143   if ( (ier = MPI_Finalize()) != 0 ) {
144     printf("Error in MPI_Finalize: return code is %d\n",
145            ier); return(ier);
146   } else {
147     return(ret_code);
148   }
149
150 }
```

Appendix C - Full Fortran Source Code Of The Hybrid Implementation

Below, the full Fortran source of the Hybrid implementation of the example program is listed. Other than syntactical and some small semantic differences, this version is very similar to the C version described in The Implementation Of The MPI Algorithm and Parallelizing The Example Using The Hybrid Model sections. The MPI calls and OpenMP directives, as well as the run time functions, are highlighted in bold.

The A Hybrid Fortran Implementation Of The Example Program section starting on page 53 covers how to compile, link and run this program using the Oracle Solaris Studio compilers and Oracle Message Passing Toolkit software.

```
  program main

!$ USE OMP_LIB

  implicit none

  include 'mpif.h'

  interface
    real(kind=8) function setup_data(n, data)
       integer,                   intent(in)   :: n
       real(kind=8), allocatable, intent(inout):: data(:)
    end function setup_data
    integer function check_numerical_result(avg, ref_result)
       real(kind=8), intent(in):: avg, ref_result
    end function check_numerical_result
  end interface

  real(kind=8), allocatable:: data(:)
  real(kind=8) :: avg, sum, Lsum, ref_result
  real(kind=8) :: ThreadSum
  real(kind=8) :: t_start, t_end, t_comp_avg, t_mpi
  integer      :: n, irem, nchunk, istart, iend, vlen
  integer      :: ier, me, nproc, ret_code = 0
  integer      :: master = 0, msg_tag1 = 1117, msg_tag2 = 2010
  integer      :: i, p, memstat
```

```
call MPI_Init(ier)
if ( ier /= 0 ) then
  write(*,"('Error in MPI_Init: return code is ',I6)") &
        ier; stop
end if

call MPI_Comm_size(MPI_COMM_WORLD,nproc,ier)
if ( ier /= 0 ) then
  write(*,"('Error in MPI_Comm_size: return code is ',I6)") &
        ier; stop
end if

call MPI_Comm_rank(MPI_COMM_WORLD,me,ier)
if ( ier /= 0 ) then
  write(*,"('Error in MPI_Comm_rank: return code is ',I6)") &
        ier; stop
end if

if ( me == master ) then

  write(*,"('Please give the number of data points:')")
  read(*,*) n

  write (*,"(/,'There are ',I3,' MPI processes')") nproc
  write (*,"(  'Number of data points: ',I10)") n

  ref_result = setup_data(n, data)

  t_start = MPI_Wtime()

  irem   = mod(n,nproc)
  nchunk = (n-irem)/nproc

  do p = 1, nproc-1
    if ( p < irem ) then
       istart = (nchunk+1)*p + 1
       iend   = istart + nchunk
    else
```

```
        istart = nchunk*p + irem + 1
        iend   = istart + nchunk - 1
      end if
      vlen = iend-istart+1

      call MPI_Send(vlen, 1, MPI_INTEGER, p, msg_tag1, &
                    MPI_COMM_WORLD, ier)
      if ( ier /= 0 ) then
         write(*,"('Error in MPI_Send: return code is ',I6)") &
         ier; stop
      end if
      call MPI_Send(data(istart), vlen, MPI_DOUBLE_PRECISION, &
                    p, msg_tag2, MPI_COMM_WORLD, ier)
      if ( ier /= 0 ) then
        write(*,"('Error in MPI_Send: return code is ',I6)") &
              ier; stop
      end if
    end do

    if ( irem > 0 ) then
      vlen = nchunk + 1
    else
      vlen = nchunk
    end if

  else

    call MPI_Recv(vlen, 1, MPI_INTEGER, master, msg_tag1, &
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ier)
    if ( ier /= 0 ) then
       write(*,"('Error in MPI_Recv: return code is ',I6)") &
             ier; stop
    end if

    allocate(data(1:vlen),STAT=memstat)

    if ( memstat /= 0 ) stop 'Fatal error in memory allocation'

    call MPI_Recv(data, vlen, MPI_DOUBLE_PRECISION, master,    &
```

```
                msg_tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE, &
                ier)
    if ( ier /= 0 ) then
      write(*,"('Error in MPI_Recv: return code is ',I6)") &
            ier; stop
    end if

  end if

  if ( me == master ) then
    t_end   = MPI_Wtime()
    t_mpi   = t_end - t_start

    t_start = t_end
  end if

  Lsum = 0.0
!$OMP PARALLEL DEFAULT(NONE) SHARED(me,vlen,data,Lsum) &
!$OMP          PRIVATE(i,ThreadSum)

   !$OMP SINGLE
     write(*,"('MPI process ',I3,' uses ', I3, &
  &        ' OpenMP threads')") me, OMP_GET_NUM_THREADS()
   !$OMP END SINGLE

   ThreadSum = 0.0
   !$OMP DO
     do i = 1, vlen
       ThreadSum = ThreadSum + data(i)
     end do
   !$OMP END DO

   !$OMP CRITICAL
     Lsum = Lsum + ThreadSum
     write(*,"('MPI process ',I3,' excutes OpenMP thread ', &
  &        I3)") me, OMP_GET_THREAD_NUM()
     write(*,"('ThreadSum = ',G23.17)") ThreadSum
   !$OMP END CRITICAL
```

```
!$OMP END PARALLEL

  if ( me == master ) then
    t_comp_avg = MPI_Wtime() - t_start
  end if

  write(*,"('MPI process ',I3,' has computed its local sum: ', &
 &      G24.18)") me, Lsum

  if ( me == master ) t_start = MPI_Wtime()

  call MPI_Reduce(Lsum,sum,1,MPI_DOUBLE_PRECISION,MPI_SUM, &
                  master,MPI_COMM_WORLD, ier)
  if ( ier /= 0 ) then
    write(*,"('Error in MPI_Reduce: return code is ',I6)") &
          ier; stop
  end if

  if ( me == master ) then
    t_mpi = t_mpi + MPI_Wtime() - t_start

    avg = sum / n

    ret_code = check_numerical_result(avg, ref_result)

    if ( ret_code ==  0 ) then
      write(*,"('n = ',I10,' average = ',G17.10)") n, avg
      write(*,"('Computation: ',       F8.3,' (s)')", &
            advance='no') t_comp_avg
      write(*,"(' MPI communication: ',F8.3,' (s)')", &
            advance='no') t_mpi
      write(*,"(' Sum: ',              F8.3,' (s)')") &
            t_comp_avg+t_mpi
    else
       write(*,"('ERROR: COMPUTED RESULT IS WRONG')")
    end if

  end if
```

```
if (allocated(data)) deallocate(data)

call MPI_Finalize(ier)

if ( ier /= 0 ) then
  write(*,"('Error in MPI_Finalize: return code is ',I6)") &
        ier; stop
end if

stop
end program main

! ------------------------------------------------------------
! ------------------------------------------------------------
real(kind=8) function setup_data(n, data)

  implicit none

  integer                  :: n
  real(kind=8), allocatable:: data(:)

  real(kind=8):: ref_result
  integer     :: memstat, i

  allocate(data(1:n),STAT=memstat)

  if ( memstat /= 0 ) then
    stop 'Fatal error in memory allocation'
  else
    do i = 1, n
      data(i) = i
    end do
    ref_result = 0.5*dble((n+1))
  end if

  setup_data = ref_result

return
end function setup_data
```

```
! ------------------------------------------------------------
! ------------------------------------------------------------
integer function check_numerical_result(avg, ref_result)

  implicit none

  real(kind=8):: avg, ref_result

  real(kind=8):: tolerance
  real(kind=8):: rel_error
  integer     :: ret_code = 0
  real(kind=8):: DBL_EPSILON, DBL_MIN

  DBL_EPSILON = epsilon(ref_result)
  DBL_MIN     = tiny(ref_result)
  tolerance   = 100.0 * DBL_EPSILON

  if ( abs(ref_result) > DBL_MIN ) then
    rel_error = abs( (avg - ref_result)/ref_result )
  else
    rel_error = abs( avg - ref_result )
  end if

  if ( rel_error > tolerance ) then
    print *,'FATAL ERROR: computed  result = ',avg
    print *,'             reference result = ',ref_result
    print *,'             relative error   = ', &
          100.0*rel_error,' %'
    ret_code = -1;
  end if

  check_numerical_result = ret_code

return
end function check_numerical_result
```

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

Parallel Programming with Oracle Developer Tools
May 2010
Author: Ruud van der Pas
Contributing Author: Bhawna Mittal

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

SOFTWARE. HARDWARE. COMPLETE.