

Compiler Support and Performance Tuning of OpenMP Programs on SunFire Servers

Myungho Lee, Larry Meadows, Darryl Gove, Dominic Paulraj, Sanjay Goil
Compiler Performance Engineering Group

Brian Whitney
Strategic Application Engineering Group

Nawal Copty, Yonghong Song
SPARC Compilation Technology Group

Sun Microsystems, Inc.

Abstract

OpenMP is fast becoming the standard paradigm for parallelizing applications for shared memory multiprocessor (SMP) systems. With a relatively small amount of coding effort, users can obtain scalable performance for their applications on an SMP system. In this paper, we present an overview of OpenMP support in the Sun ONE Studio 8 Compiler Collection(tm). We highlight the optimization capabilities of Sun's compilers on programs written with OpenMP directives. These optimizations, along with new features in the Solaris(tm) 9 Operating Environment, have resulted in high performance and good scalability for the SPEC OMPL benchmarks on the SunFire (tm) line of servers. The benchmarks scale well up to 71 processors on a 72-processor SunFire 15K system, and achieve an impressive SPEC OMPL peak ratio of 213,466.

1. Introduction

OpenMP [1, 2] is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in C, C++, and Fortran programs. The main motivations for using OpenMP are performance, scalability, throughput, and standardization. OpenMP has proven to be a powerful tool for parallelizing existing sequential programs with a relatively small amount of coding effort.

As application programs demand more computing power, achieving high performance using the OpenMP API becomes more important. In this paper, we review OpenMP support in Sun's compilers and tools, and describe performance tuning of OpenMP programs on Sun's SunFire(tm) servers.

The Sun ONE Studio 8 (S1S8) Compiler Collection(tm) [3] supports the C/C++ and Fortran OpenMP Specifications Version 2.0, and provides the programmer with tools for checking, debugging, and analyzing the performance of OpenMP applications. In addition to directive-based parallelization, S1S8 compilers can perform advanced compiler optimizations, such as loop optimizations and automatic parallelization, that improve the performance of both serial and parallel programs written using OpenMP.

SunFire(tm) 6800 and 15K systems are Sun's largest mid-range and high-end servers, respectively. These servers are based on Sun Ultra III Cu processors and can scale up to 24 processors (SunFire 6800) and 106 processors (SunFire 15K). Achieving high performance for OpenMP programs on SunFire servers involves analyzing program performance, identifying bottlenecks, and using a variety of compiler options and operating system tunable parameters to improve performance.

The rest of this paper is organized as follows. Section 2 presents compiler, runtime, and tools support for OpenMP in the S1S8 Compiler Collection(tm). Section 3 gives an overview of compiler and other optimizations that improve the performance of OpenMP programs. Section 4 describes the SPEC OMPL OpenMP benchmarks, and the SunFire(tm) benchmarking systems. Section 5 describes how OpenMP applications can be tuned to obtain high performance and scalability on SunFire(tm) systems, and presents Sun's current SPEC OMPL benchmark results on these systems. Finally, Section 6 concludes the paper with a brief description of Sun's future directions in supporting OpenMP.

2. OpenMP Support in Sun ONE Studio 8 Compiler Collection(tm)

The OpenMP model of parallel execution is the fork-join model. In this model, a *master* (initial) thread executes sequentially until a parallel region of code is encountered. At that point, the *master* thread forks a team of *worker* threads. All threads cooperatively execute the parallel region. At the end of the parallel execution, or the join point, the *master* thread alone continues sequential execution.

OpenMP support in Sun's compilers consists of two parts. First, the compiler processes OpenMP directives in a program and transforms the code so that it can be executed by multiple threads. Second, a runtime library provides support for thread management, synchronization, and scheduling [4].

2.1 Compiler Support

The compiler recognizes OpenMP directives, processes the information associated with them, and transforms the code so that it can be executed by multiple threads. When the compiler encounters a parallel construct (PARALLEL, PARALLEL DO, PARALLEL SECTIONS, or PARALLEL WORKSHARE directive) in the program, it does the following:

- First, the compiler analyzes the properties of variables in the body of the parallel construct, and determines whether a variable is SHARED, PRIVATE, FIRSTPRIVATE, LASTPRIVATE, REDUCTION, COPYPRIVATE, etc.
- Second, the compiler extracts the body of the parallel construct and places it in a separate procedure, called an *outlined procedure*. Variables that are SHARED are passed as arguments to the *outlined procedure*, so they can be accessed by multiple threads. Variables that are PRIVATE are declared to be local to the *outlined procedure*. Additional code is added to the *outlined procedure* to initialize FIRSTPRIVATE variables, update LASTPRIVATE variables, compute REDUCTIONS, etc.
- Third, the compiler replaces the original parallel construct by a call to the runtime library routine `__mt_MasterFunction_`. The address of the *outlined procedure* is passed as argument to `__mt_MasterFunction_`.

The *outlining* transformation described above has several advantages. First, an *outlined procedure* defines a context for parallel execution that can be easily executed by multiple threads. Second, *outlining* simplifies storage management, since variables that are local to the *outlined procedure* will automatically be allocated on different thread stacks. Third, placing the code in a separate procedure minimizes the impact of compiler optimizations on the code to be executed in parallel.

Worksharing constructs in the program (DO, SECTIONS, and WORKSHARE directives) are processed by the compiler in a similar fashion. The compiler, however, replaces a worksharing construct by a call to the runtime routine `__mt_WorkSharing_`. Figures 1(a) and (b) illustrate the compiler transformation applied to an OpenMP PARALLEL region.

2.2 Runtime Library Support

The OpenMP runtime support library, *libmtnsk*, provides support for thread management, synchronization, and scheduling of parallel work. As described above, the compiler replaces the code for a parallel construct by a call to `__mt_MasterFunction_`. The *master* (initial) thread of a program executes sequentially until it encounters a call to the `__mt_MasterFunction_`. The call to `__mt_MasterFunction_` causes the parallel region code (now in an *outlined procedure*) to be executed by the *master* thread and a

```

program test
integer n, id
...
!$omp parallel shared(n), private(id)
id = omp_get_thread_num()
call work (n, id)
...
!$omp end parallel
end

Figure 1(a): Example Program with a
Parallel Region

```

```

program test
integer n, id
task_info_t taskinfo
...
taskinfo = ...
call __mt_MasterFunction_ (taskinfo,
    _mf_parallel_1_, ...)
...
end

subroutine _mf_parallel_1_ (n)
integer n          ! shared
integer id         ! private

id = omp_get_thread_num()
call work (n, id)
end subroutine _mf_parallel_1_

Figure 1(b): Compiler-Generated Code
showing call to __mt_MasterFunction_
and Outlined Procedure

```

team of *worker* threads. The general logic of the `__mt_MasterFunction_()` is shown in Figure 2.

The team of *worker* threads is created when the *master* thread encounters an OpenMP region or an OpenMP runtime routine for the first time. Throughout its lifetime, a *worker* thread executes the runtime routine `__mt_SlaveFunction_()`, where it alternates between waiting for the next parallel task assigned to it by the *master* thread, and executing a task. While waiting for a task, a *worker* thread may be spinning or sleeping, depending on the setting of the environment variable `SUNW_MP_THR_IDLE`. When a thread finishes working on a task, it synchronizes with the *master* thread and other threads in the team via a call to a barrier routine `__mt_EndOfTask_Barrier_()`. The general logic of `__mt_SlaveFunction_()` is shown in Figure 3.

The runtime library supports multiple user threads. If a user program is threaded via explicit calls to the POSIX or Solaris(tm) thread libraries, then *libmtnsk* will treat each of the user program threads as a *master* thread, and provide it with its own team of *worker* threads.

2.3 Tools Support

The S1S8 Compiler Collection(tm) provides tools for checking, debugging, and analyzing the performance of OpenMP programs. We describe a few of these tools below.

Global Program Checking. Sun's Fortran 95 compiler can statically analyze a program and report inconsistencies and possible runtime problems in the code. The analysis is global (interprocedural) and is performed under the control of a compiler option. Problems reported include inconsistencies in the use of OpenMP directives, errors in alignment, disagreement in the number or type of procedure arguments, etc.

OpenMP Debugging. The *dbx* tool in the S1S8 Compiler Collection(tm) can be used to debug C and Fortran OpenMP programs. All of the *dbx* commands that operate on threads can be used for OpenMP debugging. *dbx* allows the user to single-step into a parallel region, set breakpoints in the body of an OpenMP construct, as well as print the values of `SHARED`, `PRIVATE`, `THREADPRIVATE`, etc., variables for a given thread.

```

__mt_MasterFunction_ (TaskInfo, ...)
{
...
if (libmtsk not initialized)
{
    __mt_init_()
    for (i=1; i < num_threads; i++)
    {
        thr_create (...,__mt_SlaveFunction_,...)
    }
}

Assign task to worker threads.
Work on task.
Wait for other threads to finish task (barrier).
}

```

*Figure 2: General Logic of the
__mt_MasterFunction() Runtime Routine*

```

__mt_SlaveFunction_ ()
{
...
while (1)
{
    Wait for next task to work on.
    Work on task.
    Wait for other threads to finish task (barrier).
}
}

```

*Figure 3: General Logic of the
__mt_SlaveFunction() Runtime Routine*

Collector and Performance Analyzer. The Collector and Performance Analyzer are a pair of tools that can be used to collect and analyze performance data for an application. Both tools can be used from the command line or from a graphical user interface. The Collector collects performance data using a statistical method called profiling and by tracing function calls. The data can include call-stacks, micro-state accounting information, thread-synchronization delay data, hardware-counter overflow data, memory allocation data, and summary information for the process, etc. The Performance Analyzer processes the data recorded by the Collector, and displays various metrics of performance at program, function, caller-callee, source-line, and assembly instruction levels. The Performance Analyzer can also display the raw data in a graphical format as a function of time. Information related to OpenMP programs, such as time spent at barriers, is presented in a way that corresponds to the original source code [4]. Compiler commentary in annotated source code listings informs the user about the various compiler optimizations and transformations that have been applied to the code.

3. Optimizing OpenMP Programs

Sun's compilers perform extensive analysis and apply OpenMP-specific and other optimizations in order to improve the performance of the code. Enhancements in S1S8 compilers include advanced data prefetching, inlining, loop unrolling, and skewed tiling, among many others. In what follows, we give a brief overview of some of the optimizations that are performed.

Scalar Optimizations. The compilers implement many scalar optimizations, such as arithmetic re-association, condition elimination, forward substitution, branch combination, loop invariant hoisting, induction variable-based optimizations, dead code elimination, common sub-expression elimination, etc.

Loop and Cache Optimizations. As modern processors have become increasingly faster than main memory, cache hierarchy has been introduced to reduce the average latency to main memory. To take advantage of the memory hierarchy and improve cache locality, the compilers implement many loop-and memory-oriented transformations, including loop distribution, loop fusion, loop interchange, loop tiling, array contraction, etc.

Prefetching. To alleviate memory latency, modern processors often have the ability to prefetch data into cache before its use. Sun's compilers analyze the code, determine which references are likely to incur

cache misses at runtime, and insert prefetch instructions in the code for these references. Prefetch candidates are determined based on reuse information. Most existing compilers can only handle affine array subscripts in terms of enclosing loop indices. Sun's compilers, however, can handle more complex subscripts based on data flow information.

Optimizations for Instruction-Level Parallelism. Modern processors have many functional units and are able to issue and execute several instructions in any given cycle. To utilize these capabilities, the compilers perform optimizations that improve instruction level parallelism, such as unroll-and-jam, scalar replacement, instruction scheduling, etc.

Interprocedural Analysis and Optimizations. The compilers perform interprocedural side-effect analysis, alias analysis, and whole program analysis. These analyses are used for various optimizations, including cache-conscious data layout, procedure inlining, and procedure cloning.

Automatic Parallelization. In addition to directive-based parallelization, the compilers can automatically parallelize loops in a program. These can be *DO* (Fortran) or *for* (C) loops in the source code, or loops generated by the compilers when transforming the code (for example, loops generated for Fortran 95 array statements). The compilers perform data dependence analysis and various loop transformations to enable automatic parallelization of loops. Specifically, the compilers perform loop distribution to create parallelizable loops, loop fusion to increase granularity of parallelizable loops, and loop interchange to parallelize outer loops. The compilers also recognize reduction operations for both scalar and array variables. The compilers then transform parallelizable loops to multi-threaded code in a fashion similar to that described in Section 2 for OpenMP. Directive-based parallelization and automatic parallelization can be freely mixed in a program.

OpenMP-specific Optimizations. OpenMP parallelization incurs an overhead cost that does not exist in sequential programs. This includes the cost of creating threads, synchronizing threads, accessing shared data, bookkeeping of information related to threads, and so on. Sun's compilers and *libmtnsk* employ a variety of techniques to reduce this overhead. For example, synchronization costs are minimized by using fast locking mechanisms, padding of lock variables to minimize false sharing, tree-based barriers, etc. The cost associated with accessing shared data are minimized by making a copy of read-only shared data on each thread stack. The cost of thread creation are alleviated by not destroying the threads at the end of a parallel region, but reusing the threads to execute subsequent parallel regions. To minimize the overhead of serialized regions, *libmtnsk* includes separate execution paths for single-thread execution and multiple-thread execution.

4. SPEC OMPL Benchmarks and Benchmarking Systems

In this Section, we describe our target application programs, SPEC OMPL. We then describe our benchmarking systems, SunFire(tm) servers, on which we tune the performance of SPEC OMPL, and the Solaris(tm) 9 operating system which provides high scalability and performance for these systems.

4.1 SPEC OMPL Benchmarks

The SPEC OMPL benchmark suite consists of nine application programs written in C and Fortran, and parallelized using the OpenMP API. These benchmarks are representative of HPC and scientific applications from the areas of chemistry, mechanical engineering, climate modeling, and physics. Each benchmark requires a memory size up to 6.4 GB when running on a single processor. Thus the benchmarks target large-scale systems with a 64-bit address space. *Table 1* lists the benchmarks and their application areas.

4.2 Benchmarking Systems

Our benchmarking systems are Sun's mid-range and high-end servers: 24-way SunFire 6800 (mid-range) and 72-way SunFire 15K (high-end). In the following, we describe the architecture and operating system of these servers.

Table 1: List of SPEC OMPL Benchmarks

Benchmark	Application Area	Language
311.wupwise_1	Quantum chromodynamics	Fortran
313.swim_1	Shallow water modeling	Fortran
315.mgrid_1	Multi-grid solver	Fortran
317.applu_1	Partial diff equations	Fortran
321.quake_1	Earthquake modeling	C
325.apsi_1	Air pollutants	Fortran
327.gafort_1	Genetic algorithm	Fortran
329.fma3d_1	Crash simulation	Fortran
331.art_1	Neural network simulation	C

Sun's UniBoard Architecture

The basic computational component of Sun's mid-range and high-end servers is the UniBoard. Each UniBoard consists of up to 4 Ultra III Cu processors with an integrated memory controller and associated memory. UniBoards can be interchanged between any of SunFire 3800, 4800, 6800, 12K and 15K servers.

SunFire 6800

The SunFire 6800 is Sun's largest mid-range server. It can contain up to 6 UniBoards, up to 24 Ultra III Cu processors, and up to 192 GB memory. The system bus clock rate is 150MHz. The SunFire 6800 uses bus-based, snoopy cache coherency; all UniBoards are in the same snooping domain. Peak bandwidth is limited by address snooping to 9.6 GB/sec. Memory latency (measured with `lat_mem_rd` from `lmbench`) is 220ns for transactions within a UniBoard, and 272ns for transactions between UniBoards. For more details on SunFire 6800, see [5].

SunFire 15K

The SunFire 15K is Sun's flagship high-end server. It can contain up to 18 UniBoards and up to 17 2-CPU MaxCPU boards, for a total of 106 Ultra III Cu processors. The maximum memory capacity is 576 GB. The system bus clock rate is 150MHz. The SunFire 15K uses bus-based, snoopy cache coherency within a UniBoard/MaxCPU board combination, and uses directory-based cache coherency between these units. Achieved hardware memory bandwidth on a 72-processor system has been measured at 80 GB/sec. Memory latency (measured with `lat_mem_rd` from `lmbench`) is 240ns for transactions within a UniBoard and 455ns for transactions between UniBoards. Memory latency within the same UniBoard on SF15K is slightly higher than that on SF6800 due to extra cycles generated by directory-based coherency. For more details on SunFire 15K, see [6].

Solaris 9

The Solaris(tm) 9 Operating Environment (Solaris 9) provides scalability and high performance for the SunFire(tm) systems [7]. New features in Solaris 9 which improve the performance of OpenMP programs without hardware upgrades are Memory Placement Optimizations (MPO) and Multiple Page Size Support (MPSS), among others.

MPO allows Solaris to allocate pages close to the processors that access those pages. As mentioned above, SunFire 6800 and SunFire 15K have different memory latencies within the same UniBoard versus between different UniBoards. The default MPO policy, called *first-touch*, allocates memory on the UniBoard containing the processor that first touches the memory. The *first-touch* policy can greatly improve the performance of applications where data accesses are made mostly to the memory local to each processor with *first-touch* placement. Compared to a *random* memory placement policy where the memory is evenly distributed throughout the system, the memory latencies for applications can be lowered and the bandwidth increased, leading to higher performance.

MPSS allows a program to use different page sizes for different regions of virtual memory. The default page size in Solaris is 8KB. With the 8KB page size, applications that use large memory can have a lot of TLB misses, since the number of TLB entries on Ultra III Cu allow accesses to only a few megabytes of memory. Ultra III Cu supports four different page sizes: 8 KB, 64 KB, 512 KB, and 4MB. With MPSS, user processes can request one of these four page sizes. Thus MPSS can significantly reduce the number of TLB misses and lead to improved performance for applications that use a large amount of memory.

5. Performance Tuning and Results

In this Section we describe our performance tuning of OpenMP programs and present Sun's latest SPEC OMPL results on SunFire(tm) servers.

5.1 Performance Tuning

Performance tuning of an OpenMP application involves first identifying bottlenecks in the program. These bottlenecks can be related to OpenMP directives used in the program or to user code. As mentioned in Section 3, OpenMP support in the compilers and the runtime library, *libmtnsk*, incur an overhead cost that does not exist in sequential programs. Moreover, the compilers may disable certain optimizations on code enclosed by OpenMP directives to maintain program correctness, while such optimizations may be applied in the sequential case.

Bottlenecks from user code not related to OpenMP directives can be removed by advanced compiler optimizations such as scalar optimizations, loop transformations, memory hierarchy optimizations, interprocedural optimizations, etc. Automatic parallelization by the compiler, beyond user-specified parallelization, can further improve the performance. These optimizations are activated by compiler options specified by the user. If there are parts of the code where the compiler can only do limited optimizations, the user can rewrite the code to enable more compiler optimizations. For example, in the SPEC OMPL suite, source changes can be proposed to improve scalability and performance of a benchmark program. The proposed source changes are reviewed by the SPEC HPG committee for approval.

Efficiently utilizing MPO in Solaris 9 can significantly improve the performance of programs with intensive data accesses to localized regions of memory. With MPO, memory accesses can be kept on the local board most of the time, whereas, without MPO, those accesses would be distributed over the boards (both local and remote) which can become very expensive. For programs which use a large amount of memory, using large pages (supported by MPSS in Solaris 9) can significantly reduce the number of TLB entries needed for the program and the number of TLB misses, thus significantly improving the performance.

OpenMP threads can be bound to processors using the environment variable `MT_BIND_PROCESSOR`. Processor binding, when used along with static scheduling, benefits applications that exhibit a certain data reuse pattern where data accessed by a thread in a parallel region will either be in the local cache from a previous invocation of a parallel region, or in local memory due to the OS *first-touch* memory allocation policy. Processor binding greatly improves the performance of such applications because of better cache locality and lack of remote memory accesses. Binding also leads to consistent timing results over multiple runs of the same program which is important in benchmarking.

5.2 SPEC OMPL Results

In April 2003, Sun published SPEC OMPL results on SunFire 6800 (P = 8, 16, 24) and SunFire 15K (P = 32, 48, 64, 71) servers based on 1200MHz Ultra III Cu processors. S1S8 compilers and Solaris 9 operating system were used for the published results. In order to easily compare speedups, P = 8, 16 results on SunFire 6800 and P = 32, 64 results on SunFire 15K are presented below in *Tables 2 and 3*.

Table 2: Performance Results on SunFire 6800

	P=8		P=16		Scalability	
	Base	Peak	Base	Peak	Base	Peak
311.wupwise_l	40680	39754	71325	71048	1.75	1.79
313.swim_l	26057	33233	35844	49865	1.38	1.5
315.mgrid_l	39022	39266	55784	56321	1.43	1.43
317.applu_l	20517	20384	57483	56636	2.8	2.78
321.quake_l	21218	24251	35942	38381	1.69	1.58
325.apsi_l	32568	32760	57886	58510	1.78	1.79
327.gafort_l	36602	36867	59249	59259	1.62	1.61
329.fma3d_l	26253	29761	48179	53675	1.84	1.8
331.art_l	40511	101419	72487	180882	1.79	1.78
Geometric Mean	30498	35670	53397	62523	1.75	1.75

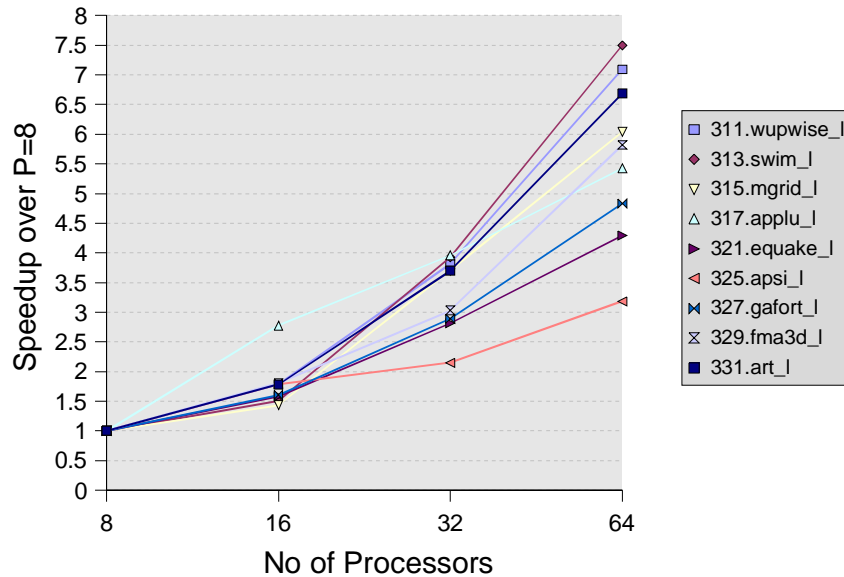
Table 3: Performance Results on SunFire 15K

	P=32		P=64		Scalability	
	Base	Peak	Base	Peak	Base	Peak
311.wupwise_l	142531	151614	247909	281935	1.74	1.86
313.swim_l	116186	130382	194392	249284	1.67	1.91
315.mgrid_l	140172	145518	224746	237173	1.6	1.63
317.applu_l	81221	80713	110801	110612	1.36	1.37
321.quake_l	50984	68301	65138	104128	1.28	1.52
325.apsi_l	70551	70551	104339	104339	1.48	1.48
327.gafort_l	106460	106574	179672	178228	1.69	1.67
329.fma3d_l	77285	90171	147689	173210	1.91	1.92
331.art_l	152586	374969	292223	678287	1.92	1.81
Geometric Mean	98228	116755	158531	195619	1.61	1.68

On SunFire 6800, we have obtained SPEC ratios 30498/35670 (base/peak) on P=8 and 53397/62523 on P=16. Comparing P=8 and P=16 results, we observe a speedup of 1.75 for both base and peak. Most benchmarks show good scaling. In particular, 317.applu_l shows a super-linear speedup (2.8/2.78). This is due to more efficient usage of cache as more processors are used. As the number of processors increases, the total amount of cache also increases linearly. Thus the data set begins to fit in the level-2, external cache when P=16. This reduces the number of data accesses to main memory and leads to super-linear speedup. 313.swim_l and 315.mgrid_l show low scaling. They are memory-intensive benchmarks and run into the memory bandwidth limits of SunFire 6800 when 16 processors out of 24 are used.

On SunFire 15K, we've obtained SPEC ratios 98228/116755 on P=32 and 158531/195619 on P=64. Comparing P=16 results on SunFire 6800 and P=32 results on SunFire 15K, we observe a speedup of 1.84 for base and 1.87 for peak. SunFire 6800 and SunFire 15K have different system architectures, thus directly comparing those results is not straightforward. SunFire 6800 has shorter remote memory latency compared with SunFire 15K which employs a directory-based coherence protocol. The memory bandwidth, however, is higher on SunFire15K. There are, therefore, performance tradeoffs on these machines. 313.swim_l performs and scales better on SunFire 15K, mainly because of the higher memory bandwidth. 315.mgrid_l also scales better on SunFire 15K.

Figure 4: Scalability for OMPL Peak Performance



Comparing P=32 performance with P=64 performance on SunFire 15K, we observe a speedup of 1.61/1.68. 317.applu_l scales poorly (1.36/1.37), whereas it scales super-linearly from P=8 to P=16 on SunFire 6800. As more processors are used, the effects of efficient cache usage diminishes and more cache coherency related overheads are introduced on a large number of processors. Thus the scalability is lower. 321.quake_l also shows low scalability (1.28) when the base performance on P=32 is compared with that on P=64. The scalability for the peak performance, however, is better (1.52). On P=64, the amount of data accessed by each processor is reduced compared with P=32. The 4MB page used for measuring the base performance is shared by different processors. This causes many remote memory accesses for those shared pages. A page size of 64KB is used for the peak performance and results in significantly better performance than base on P=64. The scalability is also better for peak.

Figure 4 shows that the benchmarks scale well as the number of processors is increased from P=8 to P=64. We extended the scaling further up to P=71, to exceed 200000 for peak (base/peak = 163554/213466).

6. Conclusion and Future Directions

In this paper, we gave an overview of OpenMP support in Sun ONE Studio 8 compilers and tools. We showed how performance tuning as well as compiler and other optimizations can be used to achieve improved performance. We presented SPEC OMPL performance results that exhibit good performance and scaling up to 71 processors on SunFire(tm) servers.

Further enhancements are planned in Sun's compilers, runtime libraries, tools, and Solaris operating system. Some future directions include auto-scoping of OpenMP programs [8], static error checking for C and C++ programs, runtime error checking, nested parallelism, and support for multi-threaded architectures

Acknowledgements

The authors extend their thanks to Hugh Caffey, Eric Duncan, Partha Tirumalai, and Joel Williamson for their helpful and insightful comments on this paper.

References

1. OpenMP Architecture Review Board, <http://www.openmp.org>
2. Rohit Chandra, et al, "Parallel Programming in OpenMP", Morgan Kaufmann Publishers, 2001.
3. Sun ONE Studio 8 Compiler Collection(tm)
<http://developers.sun.com/prodtech/cc/reference/docs/index.html>
4. Nawal Copty, Marty Itzkowitz, and Larry Meadows, "OpenMP Support in Sun's Compilers and Tools", SunTech 2001.
5. SunFire 6800 Server, <http://www.sun.com/servers/midrange/sunfire6800/index.html>
6. SunFire 15K server, <http://www.sun.com/servers/highend/sunfire15k/index.xml>
7. Solaris 9 Operating System, <http://www.sun.com/software/solaris>
8. Dieter an Mey, "A Compromise between Automatic and Manual Parallelization: Auto-Scoping".