Parallel Programming Model Update

Greg Fischer *and* **Charlie Carroll**, *Cray Inc.*

ABSTRACT: *This paper will present status for the various compiler-based parallel programming models on the Cray X1: Co-array Fortran, UPC and OpenMP. Brief descriptions and motivations for the models will be given. We describe functionality of the current Cray X1 implementations and our plans for future work. We discuss how the various parallel models may be combined to achieve the best utilization of the machine. We describe the gang-scheduling capability on the Cray X1 and its implications for OpenMP applications. We give some recommendations on when to use SSP mode and MSP mode with OpenMP applications. Finally, some preliminary OpenMP performance numbers on the Cray X1 are presented.*

1. Introduction

While the Cray X1 offers industry-leading single processor performance, its full power is realized through parallel execution of its processors through various parallel programming models. While MPI continues to be the most important parallel model at Cray due to its wide popularity among existing applications and across many different platforms, several other compiler-based models are offered by Cray that can give the programmer additional performance, capability and productivity gains. This paper will give an update on the status of three of these models on the Cray X1: Co-array Fortran, UPC and OpenMP.

2. Co-array Fortran and UPC

Co-array Fortran and UPC are distributed memory programming models implemented via extensions to the Fortran and C programming languages, respectively. Like MPI, the user program is replicated as multiple processes, and these processes perform local computations and communicate with each other. But whereas MPI processes communicate by sending messages, Co-array Fortran and UPC processes communicate by directly reading and writing arrays that are shared and distributed across the processes. While not official standards, these languages each have a community of implementers and users in both industry and academia that maintain their specifications. [1,2]

Co-array Fortran and UPC offer improved productivity over message passing models by halving the amount of communication code, and giving better readability and reliability by integration with the high-level language. They can also give improved performance on platforms that support one-way communication between processes (such as the Cray X1) by eliminating library routine overhead.

Co-array Fortran is available in the latest release of the Cray X1 Programming Environment. The implementation is equivalent in functionality to that offered on the Cray T3E, and no additional functionality is planned for the 5.0 or 5.1 release of the Programming Environment.

UPC is also available in the latest release of the Cray X1 Programming Environment. While the current implementation is a subset of the language, it offers those features of the language which are of most use to Cray users and which can be implemented most efficiently by the Cray compiler.

Nevertheless, for those customers that require the full UPC language, an implementation is in development and will be available in Programming Environment 5.1, planned for Fall 2003. Among the language features which are planned: support for all shared array types; support for pointers to all shared array types; support for the upc_forall statement; support for the upc_global_alloc and upc_free memory allocation routines; support for intrinsics added in version 1.1 of the UPC Specification.

Because the full UPC language is so expressive, it is possible for a programmer to create data types and control constructs that are inherently difficult for the compiler to implement efficiently. In order to help the user be aware of these, the compiler will print caution messages when the user uses a construct that the compiler was not able to implement in an “efficient” manner.

3. OpenMP

OpenMP is an industry standard parallel programming model. At its core is a set of directives that can be applied to a single-image application to create additional threads of execution and distribute the work of the application across those threads. It is a "shared-memory” programming model, designed for platforms with uniform memory access time from each processor to any location in memory.

OpenMP will be made available in the 5.1 release of the Cray X1 Programming Environment, currently planned for Fall of 2003. The Programming Environment will offer a complete implementation of OpenMP version 2.0 [3] in the Fortran, C and C++ programming languages. This will be the first time that Cray has offered OpenMP in C and C++ on any of its platforms. In addition, it will be the first time that Cray has supported the complete OpenMP 2.0 specification, most notably the WORKSHARE directive in Fortran.

4. Node and Hybrid Programming

Since OpenMP is intended for uniform memory access (UMA) platforms, it is not appropriate for programming a whole Cray X1, which is a non-uniform access (NUMA) platform. Nevertheless, OpenMP can play an important role in better utilizing the capability of the Cray X1.

The Cray X1 implementation of OpenMP is targeted to the largest component that has uniform memory access: the node. Thus X1 programmers will be able to program up to 4 MSPs (in MSP mode) or 16 SSPs (in SSP mode) using OpenMP, applying the power of an entire Cray X1 node to a single-image application.

Not only can OpenMP be used by itself, but it can also be used in combination with the distributed memory models (MPI, SHMEM, Co-arrays, UPC) to add additional capability and flexibility to those models. This technique is known by different names, including Multi-level Parallelism and Hybrid Parallel Programming.

On the Cray X1, an OpenMP program is implemented on top of POSIX threads (also known as Pthreads), contained in a single UNICOS process. Thus OpenMP can be used to bring multiple threads of execution (and thus multiple Cray X1 processors) to bear on each of the multiple processes of a distributed memory application. This technique can be effective for programs with parallelism which cannot be exploited by compiler vectorization or multi-streaming. This technique is also particularly effective for "irregular grid" applications, whose individual processes have varying amounts of parallel computation to perform. OpenMP can be used to apply more processors to those processes with more work to do, and fewer processors to those processes with less work to do. This results in better load balancing across an application, and thus better machine utilization and efficiency.

5. Gang Scheduling

The new gang scheduling feature for processor scheduling on the Cray X1 developed for distributed memory users also offers OpenMP users a capability previously not available on Cray parallel-vector platforms. By invoking the *aprun* command with the “-d *n*” option, the user assures that each OpenMP process has *n* processors available to its threads whenever it is scheduled, even in interactive sessions. This guarantees that the threads of the OpenMP application will truly run in parallel, reducing its wall-clock time. On previous Cray parallel-vector platforms, the threads of an OpenMP application were scheduled like any other, as processors became available, based on priority. This meant that parallel execution of threads within an OpenMP application was only possible, but was not guaranteed.

Guaranteed reduced wall-clock time, however, comes with a price. Even the most parallel of OpenMP applications have periods of serial execution, where only one thread is executing. During these periods, only one processor is utilized, and the others simply idle, waiting for the next parallel region to be started. Users must evaluate the additional cost of gang scheduling and weigh it against the advantage of improved wall-clock time. Users can get processor scheduling similar to that on the previous Cray parallel-vector platforms by not specifying the –d option to *aprun.*

6. MSP vs. SSP

With the Cray X1 Programming Environment release 5.0 (available in Summer 2003), users will first be able to program using an SSP as a processor, rather than an MSP. This capability is known as "SSP mode". Thus users must decide whether to use SSP mode or MSP mode to execute their OpenMP program.

One significant element to consider is E-cache performance. The E-cache is the cache shared between the four SSPs of an MSP. Since the E-cache is only a two-way cache, care must be taken to minimize cache conflict between the four SSPs.

One of the most effective ways to minimize cache conflict between SSPs is to have them executing different iterations of the same loop in the same function. This usually means the SSPs are referencing different areas of the same arrays, reducing the chance of conflict in the cache. This cooperation between SSPs is typically achieved with the multi-streaming capability of the compiler. It can, however,
also be achieved with OpenMP using the "DO" or "for" loop distribution constructs, in Fortran and C, respectively.

Another factor to consider is the style of parallel programming used by the OpenMP application. If the program is parallelized at the loop-level, using the OpenMP "DO" or "for" constructs (sometimes referred to as Fine-grain Parallelism), cache cooperation between processors is already achieved. In this case, the user may consider executing the program in SSP mode.

However, if the OpenMP program is parallelized across functions that work on "threadprivate" data (sometimes referred to as Coarse-grain, or Functional Parallelism), no cache cooperation is guaranteed between processors. In this case, the user may wish to consider executing in MSP mode in order to use multi-streaming to achieve cache cooperation between SSPs.

7. OpenMP Performance

At the time of this writing, OpenMP is available inside Cray for testing purposes. We have performed some preliminary timing tests. The data presented here should be viewed in this context—preliminary performance figures that can be expected to change, generally for the better but not always, as we finalize OpenMP. All results were obtained in SSP mode.

7.1 Overhead

J.M. Bull of the University of Edinburgh has developed a methodology to measure OpenMP overhead. We have used it to measure our implementation of OpenMP on the Cray X1. A full explanation of Bull's methodology can be found at [4]. A brief explanation is given here.

Consider a kernel of work that executes in time **Ts**. If that work is spread across **p** threads, the theoretical minimum time is **Ts/p**. The actual time will be something greater; let's call that **Tp**. The difference between these two, **Tp** and **Ts/p**, measures the overhead introduced by OpenMP.

Overhead = Tp –Ts/p

Here is the same idea illustrated with code examples. The reference code looks like this.

```
do j=1,numiter
     call delay(delaylength)
end do
```

Here is the same work done on each of p threads. The difference between the times to execute these codes is the OpenMP overhead.

```
!$OMP PARALLEL
do j=1,numiter
!$OMP DO
     do k=1,omp_get_num_threads()
             call delay(delaylength)
     end do
!$OMP END DO
end do
!$OMP END PARALLEL
```

The table below shows the overhead measurements for Parallel Do with varying numbers of threads.

Number of SSPs	Overhead
1	5.87 microseconds
2	5.93 microseconds
4	9.28 microseconds
8	15.9 microseconds

These numbers demonstrate reasonable overhead for the Cray X1, and reasonable scaling as the thread count increases.

The table below shows the overhead measurements for different chunk sizes with a Dynamic scheduling algorithm. In all cases, loops are for 1,024 iterations and with eight OpenMP threads (in SSP mode on the Cray X1). Results are shown for both the Cray X1 and Cray SV1 systems.

Chunk Size	X1 Overhead	SV1 Overhead
1	161 us	690 us
2	88 us	344 us
4	60 us	252 us
8	43 us	174 us
16	32 us	128 us
32	25 us	107 us
64	25 us	99 us
128	22 us	95 us

Overhead decreases as the chunk size increases, with the curve getting fairly flat at a chunk size of 32 or greater. Cray X1 OpenMP overhead is about 25% of the Cray SV1 OpenMP overhead.

7.2 NAS Parallel Benchmarks

We also measured OpenMP's ability to scale by applying it to codes that are known to scale well. We selected two kernels from the NAS Parallel Benchmarks.

The first chart shows results for the CG kernel. The speedup increases nearly in synch with the thread count.

SSP Count	Speedup
1	1.0
2	2.0x
4	3.9x
8	7.9x
16	15.5x

MSP performance on CG, through compiler multi-streaming, was about 40% better than the four-thread OpenMP performance. To put it another way, MSP performance is equivalent to about 5.6 OpenMP threads. This might be expected due to the better synchronization available to an MSP, plus the fact that an MSP utilizes a static distribution mechanism, which is faster than the dynamic mechanism used by OpenMP for this code. In addition, since the multi-streaming is an automatic form of parallelism, it can make some additional optimizations regarding placement of parallelism that are not available to the fixed source OpenMP version.

Here is a similar chart for the MG kernel:

SSP Count	Speedup
1	1.0
2	1.9 x
4	3.1x
8	7.0x
16	12.3x

This kernel does scale quite as spectacularly as the CG kernel, but it's still quite respectable.

8. Conclusion

Cray X1 users will continue to see additional support for compiler-based parallel programming models over the coming year. Full UPC support will be available in the Fall 2003 for those who wish to take advantage of its expressiveness. OpenMP will also be available in this time frame, giving programmers the ability to apply a full Cray X1 node to a single-image application, as well as enabling Hybrid Parallel programming with the distributed memory models. Initial OpenMP performance results show reasonable overhead and promising scalability. The new gang scheduling capability on the X1 can improve wall-clock time for OpenMP applications at some additional cost. Finally, the new SSP mode gives users more flexibility in choosing the most efficient method for running OpenMP applications.

References

[1]http://www.co-array.org
[2]http://upc.gwu.edu
[3]http://www.openmp.org
[4]http://www.epcc.ed.ac.uk/computing/research_activities/openmpbench/

Acknowledgments

The authors wish to thank David Nault, Vince Graziano and Don Ferry of Cray Inc. for their assistance in gathering performance data for this paper.

About the Authors

Greg Fischer is a member of the technical staff in the Compiler Optimization and Code Generation section of the Programming Environment group at Cray Inc. He can be reached at Cray Inc., 1340 Mendota Heights Rd., Mendota Heights, MN 55120, E-Mail: gsf@cray.com.

Charlie Carroll is the section leader of the Compiler Optimization and Code Generation section of the Programming Environment group at Cray Inc. He can be reached at Cray Inc., 1340 Mendota Heights Rd., Mendota Heights, MN 55120, E-Mail: charliec@cray.com.