OpenMP:An Autotasking Perspective

Neal Gaarder
Programming Environment Group
Silicon Graphics, Inc.
655-F Lone Oak Drive,
Eagan, MN 55121-1560, USA
nealg@cray.com
http://wwwsdiv.cray.com/~nealg

Monika ten Bruggencate
Programming Environment Group
Silicon Graphics, Inc.
1660 Old Pecos Trail, Suite F
Santa Fe, NM 87501, USA
monikatb@cray.com
http://wwwsdiv.cray.com/~monikatb

ABSTRACT:

This paper discusses the implementation of the new OpenMP standard for Fortran shared memory parallel processing in Programming Environment Release 3.1 for Cray PVP systems. Similarities and differences between OpenMP and Autotasking compiler directives, environment variables and other control mechanisms are discussed. It becomes clear that OpenMP provides equivalents for all of the essential features of current PVP Autotasking and Microtasking. The paper then discusses where users should be careful when converting Autotasked or Microtasked PVP codes to OpenMP and provides guidelines for the conversion.

KEYWORDS:

OpenMP, Autotasking, Microtasking, shared memory parallel processing, compilers.

What is OpenMP?

OpenMP is a new, standard application program interface (API) for Fortran shared memory parallel programming. Its goal is to provide a model for parallel programming that is portable across different shared memory architectures. The OpenMP Fortran API is documented at *http://www.openmp.org*. It contains descriptions of compiler directives, environment variables and library routines. The library routines and environment variables provide the functionality to control the run-time execution environment. The compiler directive sentinels are structured so that they are treated as comments in Fortran. Compilers that support the OpenMP Fortran API will provide a command line option that activates and allows interpretation of all OpenMP compiler directives. Note that, the OpenMP Fortran API specification describes only user-directed parallelization, that is, the user specifies the actions to be taken by the compiler and run-time system in order to execute the program in parallel.

OpenMP allows Fortran programmers to write compliant code which can be compiled and run on any system which complies with the standard. OpenMP is an open standard, supported by many vendors including SGI, IBM, and Intel. It is also supported by many Independent Software Vendors (ISVs) such as ANSYS.

Fortran 90 in Programming Environment Release 3.1 (3Q98) will support OpenMP.

OpenMP is SGI's direction for the future. While older PVP tasking, that is Microtasking, Macrotasking and Autotasking, is still supported, users should begin to migrate codes to OpenMP, as future products and platforms may not provide all of the old tasking directives. OpenMP provides equivalents for all of the essential features of current PVP tasking. Performance of OpenMP should be equivalent to performance of Autotasking, Microtasking or Macrotasking in most cases.

In the following sections we review a few of Cray's parallel programming paradigms available on PVPs which are closely related to OpenMP, present the platforms on which OpenMP is supported, and discuss how to convert tasked code to OpenMP.

Parallel Programming Paradigms

Cray's first parallel processing capability was Macrotasking. Users insert calls to library routines (such as TSKSTART) which creates and manages tasks for coarse grain tasking. Macrotasking is still supported, but we do not discuss it further here.

The second capability was Microtasking, where !MIC$ compiler directives were used to control finer grain tasking, generally at a loop level.

The third capability was Autotasking, where compilers automate most or all of the analysis and effectively Microtask a code. Users could control Autotasking by inserting Microtasking directives when necessary.

Now, OpenMP directives can be used to produce a standard multitasked code. OpenMP can also supplement Autotasking.

With the adoption of OpenMP, on future architectures Autotasking may be implemented on top of OpenMP, rather than Microtasking. Across the company, common technology will be used for products where it makes sense, perhaps using OpenMP mechanisms to implement Autotasking on future architectures. OpenMP and Autotasking are allowed to intermix. We will discuss benefits and restrictions of intermixing later.

OpenMP Implementation

Currently, OpenMP is supported on the following platforms:

- All IRIX platforms
- J90, T90, C90, and SV1 (requires Unicos 10.0.0.3 and PE 3.1)
- Not on YMP (no Unicos 10.0)
- Not on MPP (T3E threads do not span CPUs)

On an SV1 with Multi-streaming Processors (MSPs) OpenMP and Autotasking treat an MSP as a single CPU.

Converting Autotasked and Microtasked Codes to OpenMP

Converting code from one parallel programming paradigm to another can be a tedious and painful experience. However, we do not expect this to be the case when converting existing Autotasked or Microtasked code to OpenMP. In this case, the conversion can be done step-by-step in a straight-forward manner as we will explain subsequently. This section discusses what may change when an Autotasked or Microtasked code is converted to OpenMP. We address command line options, environment variables, loop scheduling, critical regions, PVM and MPI, number of threads, conditional compilation, library routines and compiler directives. For each of these topics we discuss what the user should be aware of when converting code in order to ease the conversion process as much as possible. See [1], [2] and [3] for more details on OpenMP and Autotasking compiler directives, environment variables, other control mechanisms and PVP optimization.

Fortran 90 Command Line

The F90 command line will generally not have to change. In addition to existing, unchanged command line options, there are new command line options to turn off the OpenMP conditional compilation sentinels (!$, c$, and *$) and to turn off OpenMP compiler directives (!$OMP). No new library options are required as all OpenMP library routines are in Libc.

The new command line options are:

-x conditional_omp : Turn off OpenMP conditional compilation sentinels

-x omp : Turn off OpenMP compiler directives

The other (unchanged) command line options to control tasking are:

-O task[n] : autotasking level

-O [no]taskinner : control autotasking of inner loops

-O [no]threshold : threshold for tasking

-O 1 and -O 2 : directive tasking, no automatic tasking

-O 3 : moderate automatic tasking

-a taskcommon : blocks are thread private as well as taskcommon

Environment Variables

OpenMP defines four environment variables to control parallelism. They are OMP_DYNAMIC, OMP_NESTED, OMP_SCHEDULE and OMP_NUM_THREADS. Next, we discuss how each one of them should be used.

Autotasking environment variables, such as NCPUS and MP_DEDICATED, also control OpenMP programs. The only interaction between OpenMP environment variables and Autotasking environment variables is between OMP_NUM_THREADS and NCPUS, which have identical functionality, limiting

the number of CPUs participating in a parallel region.

When converting tasked code to OpenMP, OMP_NUM_THREADS should be used instead of NCPUS. If it is necessary to define both environment variables, they must have identical values. Currently NCPUS overrides OMP_NUM_THREADS in the error case, ensuring that OpenMP will not unintentionally affect existing programs.

OMP_DYNAMIC specifies whether the number of CPUs can change during a parallel region. PVP systems ignore the environment variable if it is set to FALSE; that is, dynamic adjustment of the number of threads available in a parallel region cannot be disabled.

OMP_NESTED controls "nested serial parallelism"; that is, whether a nested parallel region can create new teams. PVP systems ignore the environment variable if it is set to TRUE; that is, nested parallelism cannot be enabled. This is discussed in more detail later.

OMP_SCHEDULE only applies to DO and PARALLEL DO directives that have the schedule tpye RUNTIME. This environment variable can be set to any of the recognized schedule types and specifies how loop iterations are divided among threads at run-time. If a schedule type other than RUNTIME is specified for the above directives, this environment variable is ignored. PVP systems support GUIDED and DYNAMIC (default) scheduling. If STATIC scheduling is specified, DYNAMIC scheduling is performed. See subsection "Loop Scheduling" below for more details.

Library Routines

OpenMP specifies a number of library functions. Only a few are new fuctionality. The remainder, such as locks, are alternate entry points to functions provided for Autotasking. The new functions get and set the new state variables. Examples are OMP_GET_NUM_THREADS() and OMP_SET_NUM_THREADS().

PVP systems ignore OMP_DYNAMIC. The only way to be assured of processor availability is to have the system dedicated. This is controlled by Autotasking’s MP_DEDICATED variable. See the discussion on threads below.

SGI systems ignore OMP_NESTED. Nested parallelism is not implemented. If SET_OMP_NESTED() is called, a warning is issued. See the notes for the PARALLEL and MASTER directives if you want to nest parallel regions.

On PVP systems OMP_SET_NUM_THREADS issues a warning on attempts to exceed the allowed number of tasks. See the following discussion on number of threads.

PVM and MPI

The rules for intermixing OpenMP, PVM and MPI are the same as for intermixing Autotasking, PVM and MPI. The network modes of PVM and MPI are compatible with OpenMP. The shared memory mode has restrictions which are not described here.

Number of Threads

PVP systems create a number of threads when tasking begins. This number cannot be exceeded for the duration of the application. The number of threads is OMP_NUM_THREADS, which defaults to 4 or the number of physical CPUS in the system, whichever is less. OMP_SET_NUM_THREADS issues a warning on attempts to exceed the limit.

NCPUS overrides OMP_NUM_THREADS. If both environment variables are set, they must be identical.

Loop Scheduling

On PVP systems when schedule type STATIC is specified, DYNAMIC is used instead, since STATIC scheduling is not supported. The default schedule type is system dependent. If no schedule type is specified, the schedule type for a loop is either DYNAMIC or GUIDED, depending on platform and loop type. Vectorization influences the number of iterations given to each thread during scheduling.

Critical Regions

OpenMP unnamed critical regions do not map onto unnumbered GUARDed regions, because of potential for unintentional interference with old code. We may map certain named critical regions in a future release if necessary.

Conditional Compilation

OpenMP requires that when OpenMP directives are active, lines beginning with sentinels "C$", "!$", and "*$" are conditionally compiled and the macro _OPENMP is defined. On PVP systems, OpenMP directives are active with F90 defaults, so conditional compilation is active.

Nontasking programs which have comment lines beginning with the sentinel must specify command line option "-x conditional_omp" to disable conditional compilation.

Directives

Autotasking, Microtasking, Macrotasking, and OpenMP directives can be intermixed, with few exceptions. For example, the following is a valid program:

```
REAL A(1000)
SUM=0

!MIC$ DO GLOBAL
DO J=1,1000
        A(J) = J**4

        !$OMP ATOMIC
        SUM=SUM+A(J)
END DO
END
```

The major restriction on intermixing is that in directive pairs, such as PARALLEL and ENDPARALLEL, both directives must either be OpenMP or Microtasking.

Intermixing is necessary, since large codes should be converted step-by-step. Intermixing also allows frequent testing as conversion progresses. Also, intermixing allows users who depend on third party libraries that use old directives and/or do not have OpenMP versions, to use OpenMP for at least part of their codes. In the future however, Microtasking directives may not be carried to new products or platforms.

Mapping OpenMP directives onto Autotasking directives

In order to facilitate the conversion of Autotasked code to OpenMP code we describe the mapping of all OpenMP compiler directives onto corresponding Autotasking directives. We also note any capability changes. This mapping allows users to easily determine which OpenMP directives to use when converting Autotasked code to OpenMP. Programs that use these OpenMP directives are portable and can be compiled by any compiler that supports the OpenMP standard.

Parallel region constructs

OpenMP directives !$OMP PARALLEL and !$OMP END PARALLEL mark, respectively, the beginning and end of a parallel region, where a parallel region is a block of code that is to be executed by multiple threads in parallel.
Similarly, Autotasking directives !MIC$ PARALLEL and !MIC$ END PARALLEL define a parallel region.
New capabilities: COPYIN is a new clause. A COPYIN clause on a parallel region specifies that the data of the master thread of the team can be copied to the thread private copies of the common block at the beginning of the parallel region. COPYIN can also be used on many of the other OpenMP directives.

Autotasking capabilities not in OpenMP: None.

Work-sharing constructs

A work-sharing construct divides the execution of an enclosed code region among the members of the team of threads that encounter the construct. A work-sharing construct must be enclosed within a parallel region in order for the directive to execute in parallel.

OpenMP directive !$OMP DO specifies that the iterations of the immediately following DO loop must be divided among the threads in the parallel region. If there is no enclosing parallel region, the DO loop is executed serially.
Similarly, Autotasking directive !MIC$ DOPARALLEL specifies that the immediately following DO loop may be executed in parallel by multiple processors.
New capabilities: Work-sharing parameters STATIC and RUNTIME are new. The NOWAIT clause is new.
Autotasking capabilities not in OpenMP: None.
Note that, the Autotasking directives have to be called from within a parallel region whereas the OpenMP directives don’t.

OpenMP directives !$OMP SECTIONS and !$OMP END SECTIONS specify that the enclosed sections of code are to be divided among threads in the team. Each section is executed once by a thread in the team. Threads that complete execution of their section wait at a barrier at the END SECTIONS directive

unless a NOWAIT is specified.
Similarly, Autotasking directives !MIC$ CASE and !MIC$ ENDCASE mark adjacent code blocks that can be executed concurrently, each one by a single processor.
New capabilities: The NOWAIT clause is new.
Autotasking capabilities not in OpenMP: None.

OpenMP directives !$OMP SINGLE and !$OMP END SINGLE specify that the enclosed code is to be executed by only one thread in the team. Threads in the team that are not executing the SINGLE directive wait at the END SINGLE directive unless NOWAIT is specified.
Similarly, Autotasking directives !MIC$ CASE and !MIC$ ENDCASE can enclose one code block only. They mark the beginning and end of a control structure and signal that the enclosed code will be executed on a single processor. All work within the control structure must complete before execution continues with the code below the ENDCASE.
New capabilities: None.
Autotasking capabilities not in OpenMP: None.

Combined parallel work-sharing constructs

Combined parallel work-sharing constructs are short cuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to the semantics of explicitly specifying a PARALLEL directive followed by a single work-sharing construct.

OpenMP directive !$OMP PARALLEL DO provides a shortcut form for specifying a parallel region that contains a single DO directive.
Similarly, Autotasking directive !MIC$ DOALL indicates that the DO loop beginning on the next line may be executed in parallel by multiple processors.
New capabilities: None.
Autotasking capabilities not in OpenMP: None.

OpenMP directives !$OMP PARALLEL SECTIONS and !$OMP END PARALLEL SECTIONS provide a shortcut form for specifying a parallel region that contains a single SECTIONS directive. The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a SECTIONS directive.
Similarly, Autotasking directives !MIC$ PARALLEL/ !MIC$ CASE and !MIC$ ENDCASE/ !MIC$ ENDPARALLEL have the same semantics.
New capabilities: This directive pairs provides a significant shortcut.
Autotasking capabilities not in OpenMP: None.

Synchronization constructs

OpenMP directives !OMP$ MASTER and !OMP$ END MASTER mark, respectively, the beginning and end of a master section, which is a structured block of code to be executed by the master thread of the team.
No corresponding Autotasking directives exists.
New capabilities: this is a new directive.
Autotasking capabilities not in OpenMP: Not Applicable.
Note, using the directives pair !OMP$ MASTER and !OMP$ END MASTER is equivalent to the following IF statement.

IF (this thread is the master of a team) THEN
...
ENDIF

All threads other than the master thread in a team skip the enclosed section of code and continue execution. The !OMP$ END MASTER directive does not end a parallel region and there is no implied barrier on entry to or exit from the master section.
In a nested serial parallel region all tasks become master of a team of one. Thus, every task executes the master section. The tasks may execute concurrently which may make it necessary to have critical regions and atomic updates (synchronization) inside master sections.

OpenMP directives !OMP$ CRITICAL and !OMP$ END CRITICAL mark, respectively, the beginning and end of a piece of code which can be accessed by only one thread at a time.
Similarly, Autotasking directives !MIC$ GUARD and !MIC$ ENDGUARD mark the beginning and end of a critical region, which is a piece of code to be executed be only one processor at a time.
New capabilities: None.
Autotasking capabilities not in OpenMP: None.
Note that, unnamed guarded regions are implemented using semaphores whereas critical regions are implemented using memory locks. In dedicated mode on a dedicated system an unnamed guarded region will perform better than a critical region. No performance predictions can be made for a loaded system since which of the two regions will perform better depends on the load of the system. Also note that, critical regions will not map onto any guards and thus guards will not interfere with critical regions.

OpenMP directive !OMP$ BARRIER synchronizes all the threads in a team.
Similarly, a single pair of Autotasking directives !MIC$ CASE and !MIC$ ENDCASE with no enclosed code can be used to synchronize all processors.
New capabilities: This is a new and short synchronization mechanism.
Autotasking capabilities not in OpenMP: None.
Note that, in Autotasked codes the library routine barsync() synchronizes all processors as well.

OpenMP directive !OMP$ ATOMIC ensures that a specific memory location is to be updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. This directive only applies to the immediately following statement.
Similarly, Autotasking directives !MIC$ GUARD and !MIC$ ENDGUARD with only a single line of code enclosed can achieve atomic memory updates.
New capabilities: This is a new and short mechanism for allowing atomic memory operations.
Autotasking capabilities not in OpenMP: None.

OpenMP directive !OMP$ FLUSH identifies synchronization points at which thread-visible variables are written back to memory.
Autotasking directive !MIC$ SUPPRESS achieves the same.
New capabilities: None.
Autotasking capabilities not in OpenMP: None.
Note that, neither directive invalidates the cache.

OpenMP directives !OMP$ ORDERED and !OMP$ END ORDERED mark the beginning and end of a piece of code which is executed in the order in which it would be executed in a sequential execution of the loop.

Similarly, Autotasking directives !MIC$ WAIT and !MIC$ SEND mark the beginning and end of a piece of code that must be executed sequentially.
New capabilities: None.
Autotasking capabilities not in OpenMP: None.

Data environment constructs

OpenMP directive !OMP$ THREADPRIVATE makes named common blocks private to a thread but global within the thread.
Similarly, Autotasking directive !DIR$ TASKCOMMON achieves the same.
New capabilities: None.
Autotasking capabilities not in OpenMP: None.
Note that, for slave threads, THREADPRIVATE common blocks are not guaranteed to persist beyond parallel regions because dynamic thread management cannot be disabled (see "Environment Variables"). Nevertheless, THREADPRIVATE is expected to work the same way as TASKCOMMON in Autotasking.

Conclusions

OpenMP is the direction of the future of multitasking at SGI. With F90 Programming Environment release 3.1, users should begin converting codes which contain tasking directives to OpenMP where equivalent functions exist, as future platforms and products may not support all of the obsolete Microtasking directives.

Conversion to OpenMP should not be difficult, but it should be done gradually, with frequent testing. The mapping of OpenMP and older tasking directives described in this paper should be useful in guiding the conversion and identifying any problems that may occur.

Finally, in most cases the performance of Autotasking and OpenMP should be equivalent.

References

[1] CF90 Commands and Directives Reference Manual. Silicon Graphics Inc., SR-3901, revision 3.1.

[2] Multitasking Programmer’s Manual. Silicon Graphics Inc., SR-0222.

[3] Optimizing Code on Cray PVP Systems. Silicon Graphics Inc., SG-2192, revision 3.0.

Author Biographies


Neal Gaarder received a MS in Operations Research from the University of Minnesota in 1977. He develops Math library and other PVP and MPP library software for Silicon Graphics, Inc. He worked on the implementation of OpenMP for PVP systems.

nealg@cray.com
Neal Gaarder


Monika ten Bruggencate received a Diploma in Computer Sciences from the Technical University Munich, Germany in 1990 and a Ph.D. in Electrical and Computer Engineering from the University of Wisconsin-Madison in 1997. She joined the CATS Programming Environment Group of Silicon Graphics, Inc. in 1997. Her responsibilities include development of performance tools and release coordination.

monikatb@cray.com
Monika ten Bruggencate