OpenMP Experiences and Comparisons

Author:

Terry Nelson

Organization:

NAS (NASA Advanced Supercomputing)

Abstract:

Are programs using OpenMP really completely portable? This paper discusses the differences of running OpenMP applications on the Cray SV1, C90, and SGI Origins. Experiences with the conversion of user codes to OpenMP will be presented. MPI and OpenMP coexistence will also be discussed.

Key Words:

OpenMP, MPI, C90, SV1, Origin

1. Site Background

The NAS facility has a long tradition with Cray vector platforms, but that technology may be nearing an end at NAS. The last C90, called vn, for (John) Von Neumann, was a 16 CPU, 1 Gigaword memory computer, and was decommissioned 1/31/02.
Users have been strongly encouraged to convert their vector codes to NAS' SGI Origin systems. Since a few users have thus far been unable to successfully convert their codes, an interim replacement was installed. It is an SV1ex, and is called bright, named for an Ames scientist, Loren Bright. It has 32 CPUs, and 4 Gigawords memory, and was put into service 2/01/02.

There is also at NAS a long tradition of distributed shared memory machines from SGI, including currently the largest single-image Origin in existence, the 1024 CPU Chapman.
A partial list of currently installed Origins would include:

CPUS	NAME
1024	(Chapman) - 600 MHZ CPUs
512	(Lomax)
512	(Lomax3)
256	(Delilah)
128	(Hilbert)
128	(Steger)
64	(Hopper)
32	(Turing)
16	(Evelyn)
DAO	- 10 more systems
...	

Surprisingly, I recently heard an SGI executive, in listing the largest systems, put GFDL, in Princeton, on top with over 1400 CPUs. Therefore, by his definition, total of all systems on site, NAS has an Origin with 3000 CPUs!

2. C90 --> SV1ex

Functionally, between the C90 and SV1, there are rather few differences, however, the following programs, written by Dr. Johnny Chang, at NAS, illustrate how the machines differ by more than simply maintenance costs.
To compare memory transfer speeds:

```
      program memintensive
! a memory access intensive program to
check
!  timings between vn and bright
      parameter (niter = 10000, nmax =
1000000)
      dimension a(nmax), b(nmax)
      t0 = second()
      call random_number(a)
      t1 = second()
      do i = 1,niter
         call sub(nmax,a,b)
      enddo
      t2 = second()
      print *,'Time for random_number
=',t1 - t0
      print *,'Time for memory copy
=',t2 - t1
      stop
      end
      subroutine sub(nmax,a,b)
      dimension a(nmax), b(nmax)
      b = a
      return
      end
```

To compare computation speed:

```
program compintensive
! a computationally intensive program
to check
!  timings between vn and bright
      parameter (niter = 10000000,
nmax = 64)
      dimension a(nmax)
      t0 = second()
      call random_number(a)
      t1 = second()
      do i = 1,niter
         call sub(nmax,a)
      enddo
      t2 = second()
      print *,'Time for random_number
=',t1 - t0
      print *,'Time for computation
=',t2 - t1
      stop
      end
      subroutine sub(nmax,a)
      dimension a(nmax)

a=a*(1.+a*(1.+a*(1.+a*(1.+a*(1.+a*(1.+
a*(1.+a*(1.+a*(1.+a)))))))))
a=0.1*a
      return
      end
```

These programs show quite clearly C90 and SV1 memory and CPU differences. The results were obtained for single cpu jobs on quiet systems, measure cputime, not wall time, and were easily reproducible.

For a memory-intensive code, the CPU time on the SV1 can be 4 times *slower* than on the C90.

For a computation-intensive code, the CPU time on the SV1 can be 2.5 times *faster* than on the C90.

However, when these results were broadcast, a representative of Cray pointed out that he felt they were unfair to the SV1, because they ignored the 32KW datacache which was available on the SV1, but not on the C90. And in fact, my tests showed that if the SV1 32KW datacache could be reused, a 57% CPU time improvement (603 -> 254 sec.) was achieved over the same SV1 test not taking advantage of the data cache. Of course, this is just one data point on this issue, and testing this is a little tricky, because the compiler will optimize around tests which are too simplistic.

Thinking ahead to code conversions from the Crays to the Origins, is this attention to the importance of 'multiple reuse of data' analogous to 'cache orientation' on the Origins?

3. SV1ex --> Origins

The SV1 and Origin systems are really quite disparate systems, with a number of distinct features.

The SV1 has a vector orientation, and a flat memory layout. Also, SV1 cpus are not in general dedicated, so OpenMP thread directives are really 'requests', not commands.

The Origins have a distributed shared memory layout. Origin cpus, and their accompanying memory, are in general dedicated to the life of the job, and are called cpusets. This is also why the environment variable, OMP_DYNAMIC, may be set to FALSE.

Whereas Cray options for parallelization, and memory distribution, tend to be 'inside' the compiler, using parameters, SGI has produced a long series of 'extensions', in especially two areas, parallel loop control, and data locality.

An example of loop control handled through a compiler parameter is the "loop nest optimization", -LNO.
The –pfa feature was originally from Kuck and Associates, and is now superceded by the "Automatic Parallelization Option", -apo, for which a separate license is required.

An early attempt to create a very extensive language to deal with the parallelizing and scheduling of loops and managing their data, produced a product called 'PCF'. PCF stands for 'parallel computing forum'. It was based in part on work from Sequent Corporation, and resulted in a proposed ANSI standard, ANSI-X3H5 91-0023-B. PCF contained directives like

```
  C$DOACROSS, C$MP_SCHEDTYPE, C$CHUNK,
C$COPYIN
```

As well as extensions like

```
C$PAR BARRIER, C$PAR CRITICAL SECTION,
C$PARALLEL DO
  C$PAR PDO, C$PAR SINGLE PROCESS, ...
```

and

multiprocessing utility routines
  mp_block, mp_unblock, mp_setup, mp_create, mp_destroy, mp_set_numthreads, mp_barrier, mp_slave_wait_for_work, etc.

PCF was very important in the creation of the OpenMP standard, and has a very OpenMP-like feel to it. However, as we will see later on, there are differences between PCF and OpenMP.

Because of the hierarchical layout of memory on the Origins, with various levels of cache, node memories, and virtual memory, and the criticality of cache hits and misses, false sharing, etc., having the data where you want it when you want it is a BIG deal. Not surprisingly there are many ways to affect data placement on Origins.

dplace - pre-execution NUMA memory placement tool

dlook – a tool for showing memory and process placement

dprof – a memory access profiling tool

numa_view – a tool for showing NUMA placement information

The need to pay close attention to the locality of the program's data, combined with the desire to parallelize loops whenever possible, resulted in a number of SGI extensions.

```
 Environment Variables
  _DSM_MIGRATION, _DSM_BARRIER,
_DSM_PLACEMENT, ...
   (see man pe_environ)
```

Indeed, some extensions are intended for use with OpenMP programs, e.g.

data distribution directives
  C$DISTRIBUTE, C$DISTRIBUTE_RESHAPE, C$DYNAMIC, etc.
and !$SGI_DISTRIBUTE_RESHAPE, !$SGI REDRITRIBUTE, etc.………

Thus, by way of comparison, the SV1 OpenMP was essentially a new implementation, whereas the OpenMP implementation on the Origins is generally based on PCF and the SGI extensions.

And besides all this, there is another very rich product, called shmem (shared memory). This was initially a Cray product, which SGI obtained during their merger. It could be viewed as data oriented (puts and gets) software with functionality somewhat like MPI, (or more precisely, I've been told, MPI-2 one-sided features).

There are some tricks and guidelines which may be helpful while testing OpenMP on the Origins.

Use loc() to distinguish which data items belong to which thread.
(But be careful if MPI is involved! Variables of the same thread number spun off by different MPI processes may appear to have the same location. They are distinct.)

Use sleep() to have time to view process activity.
(This is documented at 'man sleep 3C' on Cray, not in a 3F man page, as on the Origins, and as one would expect.)

The attempt to set the number of threads which the OpenMP job will use can lead to surprising or confusing results.

Job control systems (nqs, pbs, lsf) may set defaults or limits for the number of cpus the job will be permitted to use, (#PBS -l ncpus=4 in PBS, for example), or preset key environment variables like OMP_DYNAMIC.

On Cray, NCPUS overrules omp_num_threads.

On SGI, use setenv OMP_NUM_THREADS (csh environment variable)
or call omp_set_num_threads() (internal OpenMP call).

Note that earlier multiprocessing SGI extensions used mp_set_num_threads for the environment variable, which is confusingly close to the name for the routine for OpenMP to set/reset the number of threads, call omp_set_num_threads().

In general, users would be well advised to always set and test for themselves whatever CPU or thread values they require.

Process startup mechanisms vary between the Cray and SGI systems.

SV1ex - fork
tfork (for shared memory usage, the old Cray XMP task fork using ba - base address, and la - limit address)
Origins - fork
sproc (for shared memory)

The Origins are in general threadsafe, using either the older MPIO library with f77, or the newer SGI library craylibs. I would not assume the SV1 OpenMP code is thread safe.

SV1ex has f90 -a taskcommon, which could be viewed as an early version of THREADPRIVATE.

SV1 requires no special compiler options for OpenMP.
The Origins need –mp, (and assume –MP is correctly set to –MP:open_mp=ON, which is the default).

Origins produce a directory for '.rii' files called rii_files. The information in this directory is used to implement data distribution directives, as well as perform consistency checks of these directives across multiple source files. This feature can be turned off by compiling with –MP:dsm=off.

SV1 MPI has both nt (tasks) and np (processes) mpirun options.

On Origins MPI uses only mpirun -np.

4. MPI and OpenMP

In the area of parallelization, the two most prominent paradigms are probably MPI and OpenMP. Both are well defined, generally portable standards. Because they often operate in different domains, i.e. MPI on distributed memory systems, OpenMP in shared memory contexts, the question arises whether they can be combined in the same program to produce performance results with would be superior to those obtained from either model by itself.

The idea is to create multilevel parallelism. The motivation can be better work distribution, or load leveling. This idea is similar to a library called mlp (multilevel parallel), created by Jim Taft at NAS, which has produced very promising performance gains.

In terms of MPI and OpenMP together on a single application, this author is aware of considerable efforts on major codes, but not of significant success at NAS, although there may have been some success at other sites with this model. However, this still appears to be an important research area and definitely worth further effort.

In this regard, SGI recognized issues with data locality with programs using the "MPI over OpenMP" model and has worked on an improved data placement scheme.

There is an enhancement available in SGI release of MPT 1.6 (May '02). According to the SGI developer who was doing this enhancement:

The basic idea for this model is that the MPI processes are spread out to allow room for the OpenMP threads. The OpenMP threads for each MPI process are placed near the MPI parent. There is also an option to roundrobin the MPI process' data segment across the nodes that its threads are using. This has been found to help for higher thread/mpi process counts. Note that this feature will only be available for Origin 300 and 3000 series computers.

The model seems to benefit most applications where

1) the working data set does not reside in scache
2) more than 4 threads/MPI process

5. Conversion to OpenMP issues

Since OpenMP is purportedly portable, and the is standard, it is natural to want to convert codes to it, either from similar schemes like PCF, other platform's flavor of OpenMP, or for initial attempts at parallelization of a code. The latter is important because the move to OpenMP can be incremental, literally loop by loop, unlike similar attempts to convert codes to MPI.

So is OpenMP completely compatible to earlier SGI capabilities, i.e. PCF and its extensions?

Not entirely. For example, there was the NESTED feature.

There was a particular form of NEST supported in PCF, which allowed you to exploit parallelism across iterations of a perfectly nested loop-nest, e.g.
c$doacross nest(i,j)

This is not in OpenMP, and is therefore, in a sense, a degradation on the Origins.

Parallelization of nested DO loops is defined in the OpenMP standard. However, it is not often implemented. I have been 'assured' this will be available on the Origins in the next few months.

OpenMP is a standard, and is supposed to be portable. Does this imply that the SV1 and the Origins handle all programs in the same way?

Not necessarily. A particularly interesting example from an actual user code involved a very short reduction operation. The code fragment is as follows:

```
      nthreads = 0
!$OMP PARALLEL REDUCTION(+:nthreads)
      nthreads = 1
      print *, nthreads
      nthreads = nthreads + 1
!$OMP END PARALLEL
      write(*,*) nthreads
      end
```

On IRIX, both the f77 and f90 compilers object with the message
Error: Illegal reduction operator for reduction variable nthreads

On the other hand, the SV1 f90 compiler compiles the code without comment. So who is correct? It appears IRIX, since the compiler recognizes that the line

```
    nthreads = 1
```

is not a legal reduction type of operation. If that line is commented out, the code compiles (as well as on the SV1).
Now, this code seems to stand at the edge of the theory, since reductions almost invariably involve an operation over a DO loop, although neither the 1.1 or 2.0 OpenMP standard appear to rule out this type of usage. So what was the user's intent? He is using this construct to count the number of threads current at this time. It ran correctly on IBM and Linux systems, as well as the 'corrected' version on IRIX. But the SV1 does not perform a reduction in this case, and so will print '1' at the end of this fragment.

Is there another example of a Cray/Origin 'divergence' involving OpenMP?

For a line such as

```
!$OMP PARALLEL DO PRIVATE(iam)
!$OMP&        FIRSTPRIVATE(sum),
LASTPRIVATE( sum)
```

The Origin f77 and f90 compilers produce

```
"prog.f", line 8: Error: FIRSTPRIVATE
and LASTPRIVATE on same variable not
yet implemented for PARALLEL DO
```

These lines compile correctly on the Cray.

However, the form

```
!$OMP PARALLEL PRIVATE(iam)
!$OMP   DO   FIRSTPRIVATE(sum),
LASTPRIVATE( sum)
```

compiles on both platforms.

What kind of problems occur in converting real codes? In the world of OpenMP theory, almost all of the attention is on the parallelizing, variable handling, and scheduling of DO loops, and less frequently on parallel regions. This is all of value, but how does a real world code differ from this portrait? Very likely, in the middle of that very natural DO loop you want to parallelize, there will be calls, and not simply calls, but long call trees. These require a somewhat different focus than all of the theory referred to above.
The book "Parallel Programming in OpenMP", which I highly recommend at the end of this paper, devotes around 8 pages out of more than 200 to dealing with these types of situations. The usual nomenclature for these call trees is the dynamic extent of the loop.

So what cautions are appropriate? I would propose an almost biblical admonition:

He who saves his code shall lose it.

SAVE and f77/90 –static can cause problems in OpenMP programs if the variables concerned should really be scoped private. (The static parameter statically allocates all local variables. Such variables are initialized to zero and exist for the life of the program.) However, on the Origins, there is a static_threadprivate parameter, similar to –static, which could solve many of these problems.

If call trees are involved, TASKPRIVATE common blocks will likely be required.

In OpenMP 2.0 THREADPRIVATE may be applied to variables as well as COMMON blocks. However, there is no estimate as to when this version of the standard will be implemented on the Origins.

Converting codes to OpenMP, issues involving which variables need to be PRIVATE will probably need to be determined on a case by case basis,not deduced from cpu=1 cases.

Or is the code parallelized from a past life? As we saw, the system used to parallelize the code may not match OpenMP completely. Moreover one must watch out for issues of thread safety. And remember that, since the order in which threads run varies, actions such as REDUCTION can have different values on separate runs. One must know the allowable variation in precision.

6. Summary/Conclusions

- As system sizes continue to increase in terms of the number of nodes or cpus, understanding models such as MPI processes running OpenMP threads will only grow in importance.

- As distributed shared memory systems become ever more prominent, OpenMP grows in significance as the programming model of choice.

- Conversion of codes to OpenMP focuses most of all on the question of which variables must be scoped private in the parallel loop or region.

- Earlier SGI data distribution directives are still supported and can greatly enhance program performance, because of the memory layout of the Origins.

7. References

Parallel Programming in OpenMP (Chandra et al)

www.omp.org is the primary site for OpenMP information.

techpubs.sgi.com/library/ is the SGI site dor online documentation

www.nas.nasa.gov for more information on activities ongoing at NAS, and documentation on the systems there.

About the Author:

Terry Nelson
Scientific Computing Consultant
Terry Nelson, in a past life, spent 23 years with Control Data, and currently works for Computer Sciences Corp. at the NAS facility at the NASA Ames Research Center.

M/S 258-6
Moffett Field, Ca. 94035
(650) 604-4292
tnelson@nas.nasa.gov