OpenMP threading on Mio and AuN.

Timothy H. Kaiser, Ph.D.
Feb 23, 2015

Abstract

The nodes on Mio have between 8 and 24 cores each. AuN nodes have 16 cores. Mc2 nodes also have 16 cores each. Many people use OpenMP to parallelize their codes across the cores of each node. In recent comparisons between OpenMP codes running on Mio and codes running on GPUs I have not seen the scaling of the OpenMP codes that was expected, that is, the OpenMP codes were scaling poorly.

There are a number of runtime environmental variables that modify the mapping of OpenMP threads to cores. Some of the environmental variables are part of the OpenMP specification. Others are specific to compilers ,operating systems or job schedulers.

Several tests were running while using the Intel, PGI, and gcc compilers. Tests were run using a simple finite difference program. It was discovered that environmental variable settings can improve the scaling of this simple code. The finite difference program was run using different setting of KMP_AFFINITY, MP_BLIST, OMP_PROC_BIND, and the job scheduler specific variable —cpus-per-task.

KMP_AFFINITY is an environmental variable that is not part of the OpenMP specification. However, when using the Intel complier setting KMP_AFFINITY can be important. Two of many possible setting for this variable are:

export KMP_AFFINITY=verbose,compact
export KMP_AFFINITY=verbose,scatter

For the Portland Group (PGI) compiler setting MP_BLIST and OMP_PROC_BIND=true was useful.

For the gcc compiler, setting OMP_PROC_BIND=true was also useful.

This document does not address running using pthreads or multiple MPI tasks per node.

(1) Motivation:

OpenACC is a directives based parallelization API similar to OpenMP but for GPUs. I have been comparing similar codes, one using OpenACC to parallelize across GPUs and the same code using OpenMP directives to parallelize across cores of the CPUs. The OpenMP code was not showing the scaling I expected, that is, it was running slower using multiple threads.

I put that code aside and started working with the Stommel code that is used to teach MPI and OpenMP. I wanted to be sure that I was seeing the expected behavior with that code.

Also, there is a setting specific to our batch scheduler —cpus-per-task. I wanted to also see the effects of changing this parameter on the Stommel code. So one of the settings I changed in my initial test was —cpus-per-task.

(2) Test Code

The test code was stc_00.c from http://hpc.mines.edu/examples/ modified with the OpenMP directives shown in Figure 1.

```
[tkaiser@mio001 acc]$ grep -B3 -A3 pragma stc_00.c
    FLT idiff;
    *diff=0.0;
    idiff=0.0;
#pragma omp parallel
{
#pragma omp  for schedule(static) reduction(+: idiff) private(j) firstprivate
(a1,a2,a3,a4,a5)
    for( i=i1;i<=i2;i++) {
        for(j=j1;j<=j2;j++){
            new_psi[i][j]=a1*psi[i+1][j] + a2*psi[i-1][j] +
--
            idiff=idiff+fabs(new_psi[i][j]-psi[i][j]);
         }
    }
#pragma omp  for schedule(static) private(j)
    for( i=i1;i<=i2;i++)
        for(j=j1;j<=j2;j++)
           psi[i][j]=new_psi[i][j];
[tkaiser@mio001 acc]$
```

Figure 1. Code modifications for OpenMP.

The write_grid routine was not called and timing routine was modified to return wall time.

```
FLT walltime() {
        double t;
        double six=1.0e-6;
        struct timeval tb;
        struct timezone tz;
        gettimeofday(&tb,&tz);
        t=(double)tb.tv_sec+((double)tb.tv_usec)*six;
        return(t);
}
```

Figure 2. Modified timing routine.

For these initial tests I used the Intel compiler. Results using the Portland Group compiler and gcc are given below. The Intel compile line and data file are:

```
[tkaiser@mio001 acc]$ icpc -O3 -openmp stc_00.c

[tkaiser@mio001 acc]$ cat st.in
200 200
2000000 2000000
1.0e-9 2.25e-11 3.0e-6
75000
```

Figure 3. Compile line and input file.

(3) Effect of setting —cpus-per-task

I thought some of the scaling issues might be related to the setting of —cpus-per-task.

We note that the description of this option is different for sbatch and srun.

For srun the man page gives us the description in figure 4. The description for this option for sbatch is given in figure 5.

```
-c, --cpus-per-task=<ncpus>
Request that ncpus be allocated per process. This may be useful if the job
is multithreaded and requires more than one CPU per task for optimal
performance. The default is one CPU per process. If -c is specified without
-n, as many tasks will be allocated per node as possible while satisfying
the -c restriction. For instance on a cluster with 8 CPUs per node, a job
request for 4 nodes and 3 CPUs per task may be allocated 3 or 6 CPUs per node
(1 or 2 tasks per node) depending upon resource consumption by other jobs.
Such a job may be unable to execute more than a total of 4 tasks. This
option may also be useful to spawn tasks without allocating resources to the
job step from the job's allocation when running multiple job steps with the
--exclusive option.
```

```
WARNING:  There  are configurations and options interpreted differently by
job and job step requests which can result in inconsistencies for this
option.  For example srun -c2 --threads-per-core=1 prog may allocate two
cores for the job, but if each of those cores contains two threads, the job
alloca- tion will include four CPUs. The job step allocation will then launch
two  threads  per  CPU  for  a total of two tasks.

WARNING:  When  srun  is executed from within salloc or sbatch, there are
configurations and options which can result in inconsistent allocations when
-c has a  value  greater  than  -c  on  salloc  or sbatch.
```

Figure 4. Man page for srun entry—cpus-per-task option

```
-c, --cpus-per-task=<ncpus>
Advise the SLURM controller that ensuing job steps will require ncpus number
of processors per task. Without this option, the controller will just try to
allocate one processor per task.

For instance, consider an application that has 4 tasks, each requiring 3
processors.  If our cluster is comprised of quad-processors nodes and we
simply ask for 12 processors, the controller might give us only 3 nodes.
However, by using the --cpus-per-task=3 options, the controller  knows  that
each task requires 3 processors on the same node, and the controller will
grant an allocation of 4 nodes, one for each of the 4 tasks.
```

Figure 5. sbatch man page option —cpus-per-task

(3.1) Using —cpus-per-task as part of a sbatch command

I was running a threads only application, not using MPI and only a single task per node. This option did not appear to have any effect. Running on 8 core nodes with KMP_AFFINITY set to verbose it is reported that all 8 cores are available for thread scheduling. That is, the bound set is {0,1,2,3,4,5,6,7} and:

SBATCH_CPU_BIND_TYPE=mask_cpu:
SBATCH_CPU_BIND=quiet,mask_cpu:0xFF
SLURM_CPUS_PER_TASK=x

where x is equal to the number of —cpus-per-task

(3.2) Using —cpus-per-task as part of srun

I also ran an interactive session first using the srun command:

srun --exclusive --tasks-per-node=1 --cpus-per-task=1 --pty bash

with different values for —cpus-per-task.

The results below are for an 8 core node compute008.

This has an effect on thread binding and the reported number of cores available on a node for running applications, SLURM_JOB_CPUS_PER_NODE.

Effect on various Environmental Variables from changing –cpus-per-task					
–cpus-per-task	SLURM_CPUS_PER_TASK	SLURM_CPUS_ON_NODE	SLURM_JOB_CPUS_PER_NODE	SLURM_CPU_BIND	KMP_AFFINITY thread x bound to OS proc set {
1	1	1	8	quiet,mask_cpu:0x01	{0}
2	2	2	8	quiet,mask_cpu:0x11	{0,4}
3	3	3	6	quiet,mask_cpu:0x13	{0,1,4}
4	4	4	8	quiet,mask_cpu:0x33	{0,1,4,5}
5	5	5	5	quiet,mask_cpu:0x37	{0,1,2,4,5}
6	6	6	6	quiet,mask_cpu:0x77	{0,1,2,4,5,6}
7	7	7	7	quiet,mask_cpu:0x7F	{0,1,2,3,4,5,6}
8	8	8	8	quiet,mask_cpu:0xFF	{0,1,2,3,4,5,6,7}

Table 1. Effects of changing —cpus-per-task on various environmental variables.

Note that the variable SLURM_JOB_CPUS_PER_NODE does not match —cpus-per-task. Instead

SLURM_JOB_CPUS_PER_NODE= N*(cpus-per-task)
with N chosen so that SLURM_JOB_CPUS_PER_NODE ≤ 8

Also, the number of cores listed in the OP proc set is equal to SLURM_JOB_CPUS_PER_NODE.

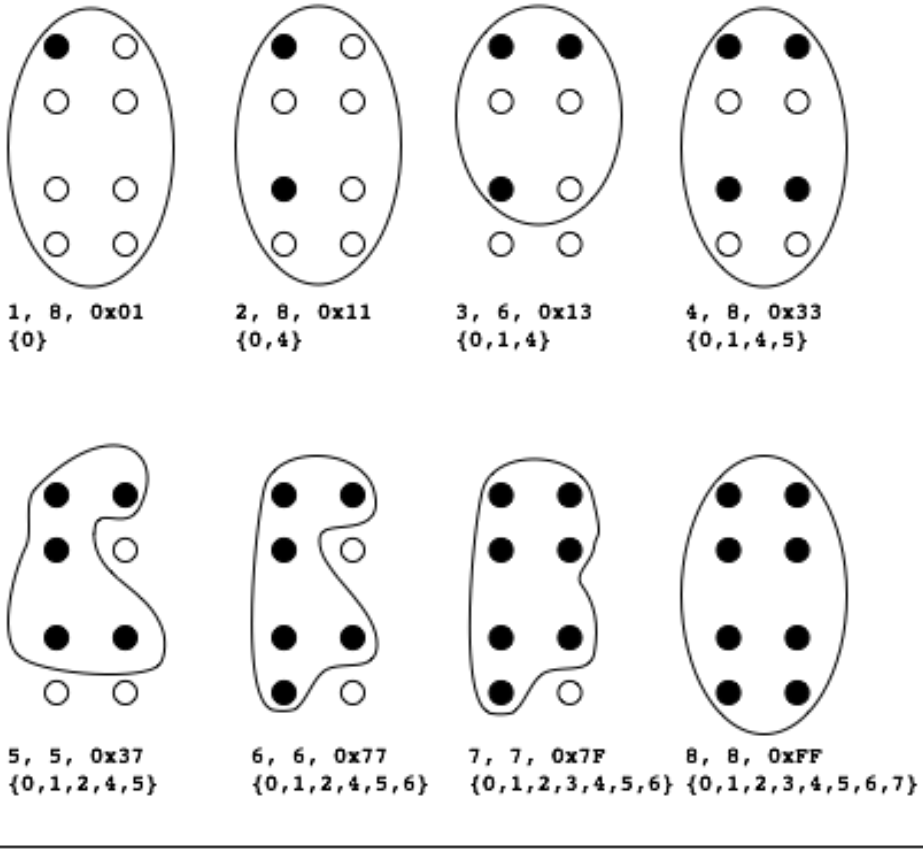

Figure 6. Graphical representation of the effects of changing —cpus-per-task on various environmental variables on an 8 core node. The captions give: —cpus-per-task, SLURM_JOB_CPUS_PER_NODE, SLURM_CPU_BIND mask, and the KMP_AFFINITY bindings.

Note that —cpus-per-task = SLURM_CPUS_PER_TASK = SLURM_CPUS_ON_NODE.

Each inner circle represents a core with the dark circles representing cores available to the application with the number equal to SLURM_JOB_CPUS_PER_NODE and the particular selection of cores given by the SLURM_CPU_BIND mask, and the KMP_AFFINITY bindings.

The outer grouping represents the setting of SLURM_JOB_CPUS_PER_NODE.

Figure 6 is a representation of the information in Table 1. It shows that we get different sets of cores available as —cpus-per-task is changed. The number of cores available is equal to —cpus-per-task and the set of cores is given by KMP_AFFINITY OS proc set and the the SLURM_CPU_BIND mask.

If we try to run using more threads than the reported OS proc set then we see a slowdown.

For example I started an interactive session with the command

[tkaiser@mio001 acc]$ srun --exclusive --tasks-per-node=1 --cpus-per-task=4 --pty bash

I then ran the program at each setting of OMP_NUM_THREADS and took the average over 5 runs and got the results shown in table 2. We saw improvement in runtime up to 4 threads. Past 4 threads the program ran slower.

OMP_NUM_THREADS	Time (sec)
1	12.44
2	6.39
3	4.30
4	5.99
5	8.13
6	7.92
7	8.46
8	8.42
Not Set	5.81

Table 2. Effects of changing —cpus-per-task on various environmental variables.

When OMP_NUM_THREADS was not set the program used a value of 4 as reported by omp_get_max_threads(). That is, the default number of threads is equal to SLURM_JOB_CPUS_PER_NODE.

(4) KMP_AFFINITY Tests

For OpenMP runs where the number of threads was approaching the number of cores I was not seeing the expected scaling. This looked like an affinity issue.

To test the effect of setting KMP_AFFINITY I ran on Mio node compute124 which is a 16 core node and in the debug queue on AuN, nodes 1,2,3. The times reported are the average of 10 runs. That is, the runscript ran through the set of values for OMP_NUM_THREADS. Finally OMP_NUM_THREADS was unset and then started over with OMP_NUM_THREADS=1. This was repeated 10 times.

The settings for KMP_AFFINITY were:

export KMP_AFFINITY=verbose
export KMP_AFFINITY=verbose,compact
export KMP_AFFINITY=verbose,scatter

Stommel Code runs on Mio node compute124			
OMP_NUM_THREADS	**Time (sec) - verbose**	**Time (sec) - compact**	**Time (sec) - scatter**
1	11.51	11.56	11.48
2	5.93	5.83	6.09
3	4.12	3.87	4.15
4	3.18	2.93	3.27
5	2.79	2.38	2.82
6	2.41	2.15	2.58
7	2.26	1.79	2.38
8	2.11	1.65	2.28
10	2.02	1.67	2.13
12	1.98	1.59	2.18
14	1.91	1.45	2.10
16	1.95	1.42	1.94
Not Set	1.88	1.42	1.95

Table 3. Changing the setting of KMP_AFFINITY effects the scaling of OpenMP codes.

Stommel Code runs on AuN nodes 1,2,3			
OMP_NUM_THREADS	**Time (sec) - verbose**	**Time (sec) - compact**	**Time (sec) - scatter**
1	9.82	9.73	9.60
2	5.12	4.89	5.09
3	3.56	3.29	3.56
4	2.75	2.48	2.70
5	2.33	2.08	2.35
6	2.10	1.75	2.10
7	1.93	1.59	1.93
8	1.69	1.43	1.86
10	1.61	1.36	1.55
12	1.49	1.28	1.55
14	3.93	1.24	1.47
16	5.74	1.18	1.49
Not Set	5.63	1.20	1.47

Table 4. Changing the setting of KMP_AFFINITY affects the scaling of OpenMP codes.

(5) Conclusions Running using Intel OpenMP :

(5.1) —cpus-per-task option

We see that if the —cpus-per-task option is used with an srun command it changes the values of the environmental variables:

SLURM_CPUS_PER_TASK
SLURM_CPUS_ON_NODE
SLURM_JOB_CPUS_PER_NODE
SLURM_CPU_BIND
and the KMP_AFFINITY bind set

It effectively sets the number of cores that are available for a process. If you run more threads than available cores you will have multiple threads on a core and you will see a slowdown. The default number of threads is equal to SLURM_JOB_CPUS_PER_NODE.

SLURM_JOB_CPUS_PER_NODE does not always equal —cpus-per-task, but instead

SLURM_JOB_CPUS_PER_NODE= N*(cpus-per-task) with N chosen so that SLURM_JOB_CPUS_PER_NODE ≤ NREAL where NREAL is the actual number of cores on a node.

(5.2) KMP_AFFINITY

Setting KMP_AFFINITY to verbose does not change the assignment of threads to cores. It only causes the potential assignments to be reported. There is a tendency for threads to not be assigned well when you do not set KMP_AFFINITY to do specific mapping. For this particular application we got the best runtime and scaling when KMP_AFFINITY=verbose,compact

(6) Portland Group Compiler runtime variables

(6.1) Motivation

The Portland Group compiler set may become more important in the future. It is important now because it supports running on GPU nodes directly using OpenACC.

The effect of setting KMP_INFINITY on Portland Group OpenMP programs was not know.

(6.2) Tests of setting KMP_AFFINITY

The program was built using the compile line:

pgcc -O4 -mp stc_00.c

The program was then run as before using sbatch on AuN nodes 1-3 with the results shown in Table 5. We see that setting KMP_AFFINITY to compact or scatter did not provide good scaling. It does not appear to change the times. We also note with OMP_NUM_THREADS set to 14 and 16 the program showed significant slowdown.

Stommel Code runs using the Portland Group Compiler & Setting KMP_AFFINITY					
KMP_AFFINITY= verbose		**KMP_AFFINITY= verbose,compact**		**KMP_AFFINITY= verbose,scatter**	
Threads	**Time (sec)**	**Threads**	**Time (sec)**	**Threads**	**Time (sec)**
1	18.60	1	16.11	1	16.21
2	9.59	2	8.21	2	8.26
3	6.46	3	5.58	3	5.60
4	4.87	4	4.27	4	4.23
5	4.09	5	3.59	5	3.57
6	3.50	6	3.05	6	3.04
7	3.14	7	2.75	7	2.74
8	2.90	8	2.58	8	2.60
10	2.61	10	2.34	10	2.35
12	2.39	12	2.12	12	2.12
14	175.89	14	216.18	14	424.35
16	155.02	16	332.15	16	680.14

Table 5. Changing the setting of KMP_AFFINITY does not improve the runtime of Portland Group compiled OpenMP codes.

(6.3) Tests of setting MP_BLIST

The Portland Group documentation discusses setting the variable MP_BLIST. MP_BLIST is similar to KMP_AFFILITY in that it tells the cores that are available for threads to use. MP_BLIST was tested by setting it to MP_BLIST=0,1,2,3,4,5,6,7 and MP_BLIST=0,8,1,9,2,10,3,11, 4,12,5,13,6,14,7,15 and running on an 8 and 16 core node on Mio.

As seen in table 6 just setting MP_BLIST does not improve the scaling of Portland Group Openmp programs.

Stommel Code runs using the Portland Group Compiler & Setting MP_BLIST			
MP_BLIST= 0,1,2,3,4,5,6,7		MP_BLIST= 0,8,1,9,2,10,3,11, 4,12,5,13,6,14,7,15	
Threads	Time (sec)	Threads	Time (sec)
1	34.62	1	18.44
2	17.76	2	9.56
3	11.92	3	6.44
4	9.03	4	4.87
5	7.37	5	4.04
6	6.32	6	3.48
7	5.52	7	3.11
8	66.01	8	2.98
		10	2.63
		12	2.38
		14	193.92
		16	283.92

Table 6. Just setting MP_BLIST does not improve the runtime of Portland Group compiled OpenMP codes.

(6.4) Tests of setting OMP_PROC_BIND

Further reading of the Portland Group documentation indicated that setting OMP_PROC_BIND=true would force the threads to the cores set in MP_BLIST. The results of setting this variable are shown in table 7. These runs were done on Mio 8 and 16 core nodes. There is relatively good scaling out to the number of cores on the node.

MP_BLIST= 0,8,1,9,2,10,3,11, 4,12,5,13,6,14,7,1 5 OMP_PROC_BIN D= true		MP_BLIST= 0,1,2,3,4,5,6,7 OMP_PROC_BIND=t rue		MP_BLIST= 0,1,2,3,4,5,6,7, 8,9,10,11,12,13,14,15 OMP_PROC_BIND= true		OMP_PROC_BIN D= true		OMP_PROC_BIND= true	
Threads	**Time (sec)**	**Threads**	**Time (sec)**	**Threads**	**Time (sec)**	**Threads**	**Time (sec)**	**Threads**	**Time (sec)**
1	16.12	1	34.68	1	16.07	1	16.16	1	34.60
2	8.30	2	17.58	2	8.13	2	8.15	2	17.56
3	5.60	3	11.77	3	5.44	3	5.44	3	11.78
4	4.26	4	8.82	4	4.08	4	4.06	4	8.84
5	3.57	5	7.23	5	3.25	5	3.28	5	7.26
6	3.08	6	6.13	6	2.80	6	2.81	6	6.14
7	2.71	7	5.34	7	2.43	7	2.45	7	5.33
8	2.52	8	4.85	8	2.21	8	2.22	8	4.87
10	2.24			10	2.14	10	2.125		
12	2.05			12	1.89	12	1.904		
14	2.00			14	1.92	14	1.934		
16	2.06			16	1.82	16	1.811		

Table 7. Setting OMP_PROC_BIND helps the runtime of Portland Group compiled OpenMP codes.

(6.5) Portland Group compiler conclusions

If the environmental variable OMP_NUM_THREADS is not set OpenMP programs will default to running a single thread.

MP_BLIST can be used to specify the cores on which threads can run, however just setting this variable does not ensure that all cores will be used efficiently.

Setting OMP_PROC_BIND=true maps the threads to cores. Specifying this along with listing the cores in MP_BLIST provides good scaling.

7.0 The gcc compiler

The gcc compiler is popular and now supports OpenMP. The source was compiled using the command:

gcc -O3 -fopenmp -lm stc_00.c

The resulting program was run on 16 core Mio nodes, compute124 and compute125. In this case MP_BLIST and KMP_AFFINITY were not set. For the first test OMP_PROC_BIND was set to true in the second case it was not set.

For this compiler, if OMP_NUM_TUREADS is not set it defaults to 16, the number of cores on the node.

Table 8 shows the results. With OMP_PROC_BIND set to true we saw good scaling. When it was not set the program showed dramatic variability between runs and slow down for larger thread counts.

As a side note, we have the output of a “ps” command run while the program was running on compute124 using 16 threads. PSR is the core number. We see that the 16 threads were mapped to each of the cores.

```
[tkaiser@mio001 acc]$ ssh compute124 ps –utkaiser –m –o
user,pid,pcpu,s,stime,pmem,comm,psr,pset
USER       PID %CPU S STIME %MEM COMMAND         PSR PSET
tkaiser   5478 1022 – 14:04  0.0 a.out             –    –
tkaiser      – 62.5 R 14:04    – –                 0    –
tkaiser      – 64.0 R 14:04    – –                 1    –
tkaiser      – 63.5 R 14:04    – –                 2    –
tkaiser      – 63.5 R 14:04    – –                 3    –
tkaiser      – 63.5 R 14:04    – –                 4    –
tkaiser      – 63.5 R 14:04    – –                 5    –
tkaiser      – 63.5 R 14:04    – –                 6    –
tkaiser      – 64.0 R 14:04    – –                 7    –
tkaiser      – 63.0 R 14:04    – –                 8    –
tkaiser      – 63.5 R 14:04    – –                 9    –
tkaiser      – 64.0 R 14:04    – –                10    –
tkaiser      – 64.0 R 14:04    – –                11    –
tkaiser      – 64.0 R 14:04    – –                12    –
tkaiser      – 63.5 R 14:04    – –                13    –
tkaiser      – 64.0 R 14:04    – –                14    –
tkaiser      – 64.0 R 14:04    – –                15    –
```

Threads	Time (sec) OMP_PROC_BIND=true	Time (sec) OMP_PROC_BIND not set
1	10.79	10.62
2	5.46	5.64
3	3.65	3.90
4	2.87	3.14
5	2.28	2.59
6	1.92	2.24
7	1.81	1.97
8	1.57	1.80
10	1.60	1.65
12	1.48	1.70
14	1.50	58.57
16	1.39	205.03

Table 8. Setting OMP_PROC_BIND helps the runtime of gcc compiled OpenMP codes.

7.1 gcc conclusions.

Setting OMP_PROC_BIND=true maps the threads to cores. If this is not set the program did not scale well.

8.0 Future work

Additional tests need to be done using pthreads and multiple MPI tasks per node. Tests should also be done on Mc2.

https://computing.llnl.gov/tutorials/openMP/ProcessThreadAffinity.pdf