Assignment 1

Amina Nasrin[#1]

[#]*Department of Computer Science, Texas State University*
Comal Building Suit 211, San Marcos, Texas 78666
[1]hzm7@txstate.edu

Abstract— **This assignment is to present the assigned tasks of parallelization in computation while implementing OPENMP programs from the course of High Performance Computing.**

Keywords— **OpenMP, parallel processing, OPENMP, parallel processing,**

I. Introduction

OpenMP is an open standard API for shared memory parallelization. We can implement different types of execution policies by using various OpenMP constructs best suited for our purpose. We present certain execution in this report. We also impose different conditions on our parallelized environment and observe the performance of the computation node that we have got allocated.

II. MACHINE ARCHITECTURE

Our first task is to understand the machine architecture that we are working with. To do this, we are instructed to use the command of hwloc tool on Stampede2. We first login to stampede2 and then run hwloc-ls. The retrieved specifications about the compute node is presented in Table I. The allocated node was a KNL Compute node.

III. Parallelize the Program

Our second assignment is to parallelize the given C program of compute_pi.c by using the Open_MP construct. This program computes the value of pi (π) by executing a user-defined number of iterations.

TABLE I
ALLOCATED MACHINE ARCHITECTURE

Name of Components	Specifications of Components
Number of sockets	1
Number of cores per socket	68
Different levels of cache	• L1 • L2
Cache size	• 32KB L1 cache • 1 MB L2 cache
Total available memory	94GB (NUMA Node)

```
PS C:\Users\amina\OneDrive\Desktop> gcc compute_pi.c -o compute_pi
PS C:\Users\amina\OneDrive\Desktop> ./compute_pi 10
10
Pi = 3.142426
PS C:\Users\amina\OneDrive\Desktop>
```

(a)

```
login1.stampede2(1000)$ nano asgn1.c
login1.stampede2(1001)$ gcc asgn1.c -o asgn1
login1.stampede2(1002)$ export OMP_NUM_THREADS=4
login1.stampede2(1003)$ ./asgn1 10
10
Pi = 3.142426
login1.stampede2(1004)$
```

(b)

Fig. 1 The output of compute_pi.c in serial processing and parallel processing matches. (a) Program run of compute_pi.c in serial processing. (b) Program run of compute_pi.c in parallel processing

To deploy the Open_MP construct, it is important to include the omp.h library in the C file. We intend to parallelize the region where the program specifically does the computation to generate the value of pi instead of including the overall program into the parallel processing which includes declarations, initializations, iterations as well.

To ensure that the parallelization is correctly computing the value of pi, we compare the output of parallelized version of compute_pi.c with its serialized version output. The program run of both serial program and parallel program is given in Figure 1.

A. *Thread Placement*

Different thread placements on a node affects the execution of the same assigned program. We can vary these parameters to observe how these differently elevates or degrades performance of the same node. We keep the number of iterations unchanged for all the following scenarios as instructed.

Our execution policy will be:

- OMP_PLACES=core OMP_PROC_BIND=close
- OMP_PLACES=core OMP_PROC_BIND=spread
- OMP_PLACES=socket OMP_PROC_BIND=close
- OMP_PLACES=socket OMP_PROC_BIND=spread

OMP_PROC_BIND defines whether the threads can be migrated across the nodes.

We enlist the execution time of each policy in Table II. From table 2, we observe that the last execution policy is taking the minimum time to execute the program. The reason behind this can be we have got allocation of such a node where all the sockets are on one node, and we have chosen the to spread across the nodes. Hence the threads can process a computation less communication time, leading to lest execution time among four.

The maximum execution time is taken by the Execution Policy of OMP_PLACES=core and OMP_PROC_BIND=spread. If OMP_PLACES is set to cores, the compiler removes the unavailable resources from OMP_PLACES. It also specifies that each place corresponds to single core although the node host machine possesses more than one core. This is why the execution time for Execution Policies 1 and 2 with OMP_PALCES=core takes more time than OMP_PLACES=socket.

B. OMP Construct

The OpenMP construct we have used to parallelize the program is #pragma omp parallel.

TABLE III
EXECUTION TIME OF DIFFERENT EXECUTION POLICIES

Srl No.	Execution Policies	Execution Time
1	OMP_PLACES=core OMP_PROC_BIND=close	0.041057
2	OMP_PLACES=core OMP_PROC_BIND=spread	0.061496
3	OMP_PLACES=socket OMP_PROC_BIND=close	0.041694
4	OMP_PLACES=socket OMP_PROC_BIND=spread	0.039004

TABLE IIIII
EXECUTION TIME WITH DIFFERENT WORK SCHEDULE AND CHUNK SIZE

Srl No.	Work Schedule and Chunk Size	Execution Time
1	Static, 10	2.80709
2	Static, 100	2.910903
3	Static, 1000	2.768555
4	Dynamic, 10	2.868037
5	Dynamic, 100	2.896331
6	Dynamic, 1000	2.690565

This construct creates a parallel region with a group of threads. Each thread executes entire block of code that the parallel region is assigned to.

We have also deployed #pragma omp for. This construct instructs the compiler to parallelize the next for block.

OMP uses shared memory programming model. If not defined specifically, variables are not shared. But it liberates to define private variable if intended by using the key word shared or private.

We have kept the value of sum as shared. It is because the sum needs to be updated in each iteration. If we declare it private, the program will update sum in each iteration as a new variable losing all the values assigned to sum every time.

```
#!/bin/bash
#SBATCH -A Scalable-Supercomput
#SBATCH -J jobname
#SBATCH -o jobname.%j
#SBATCH -N 1
#SBATCH -n 20
#SBATCH -p normal
#SBATCH -t 00:00:05

date

export OMP_NUM_THREADS=16
export OMP_PROC_BIND=[core|socket]
export OMP_PROC_BIND=[close|spread]

ibrun ./asgn6 1000000 100

date
```

Fig. 2 Batch Script for parallel processing

```
login1.stampede2(1012)$ squeue -u hzm7
             JOBID   PARTITION     NAME     USER ST
          11038080      normal  jobname     hzm7 PD
login1.stampede2(1013)$ squeue -u hzm7
             JOBID   PARTITION     NAME     USER ST
          11038080      normal  jobname     hzm7  R
login1.stampede2(1014)$ squeue -u hzm7
             JOBID   PARTITION     NAME     USER ST
          11038080      normal  jobname     hzm7  R
login1.stampede2(1015)$ squeue -u hzm7
             JOBID   PARTITION     NAME     USER ST
          11038080      normal  jobname     hzm7  R
login1.stampede2(1016)$ squeue -u hzm7
             JOBID   PARTITION     NAME     USER ST
          11038080      normal  jobname     hzm7  R
login1.stampede2(1017)$ squeue -u hzm7
             JOBID   PARTITION     NAME     USER ST
          11038080      normal  jobname     hzm7 CG
```

Fig. 4 batch submission on a login-node

We have kept the variable x private because x never gets out of the loop. So, it does not need to be shared or store the previously assigned value which will lead to occupying additional memory space.

C. Varying Number of Threads

In this section, we observe the performance of the assigned node with different number of threads. We define the number threads as power of 2 starting from 1 to 4 and record the corresponding execution time as shown in Table III.

Our next task is to compare the performance of two different work scheduling of static and dynamic with number of nodes as 16. We are also instructed to assign three different chunk sizes of 10, 100, and 1000.

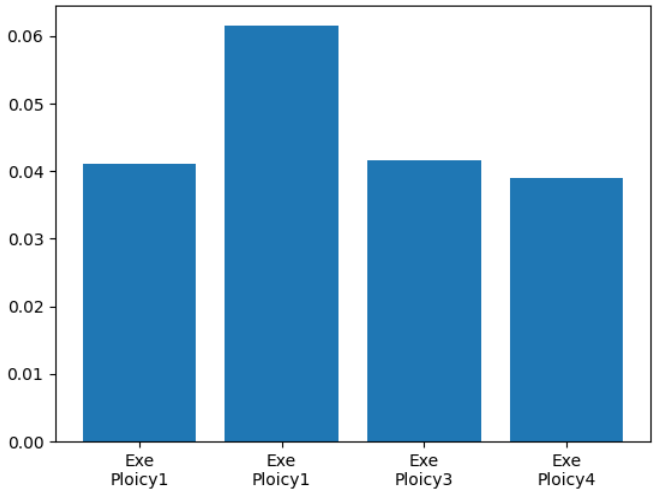

Fig. 4 Execution Policy vs Execution Time(sec)

D. Batch Submission

The executions we have done so far in parallel processing environment can be also done using batch submission. Multi-threaded jobs are expected to be submitted to the shared-memory queue using batch submission. The batch-script we have used to perform batch submission is given in Fig 2. We also observe the status of the submitted job by using the command of squeue. The job status is shown in Figure 3. Here, the value of ST set to PD indicates, the job is pending to run, R implies that the job is currently running, and CG refers to the job is completing.

E. Plotting Execution Time

We have acquired some insights based on the task executions described above. Now, we can plot compare the performance of different policies.

First, we plot the execution time of the four execution policies. The graph is plotted shown in Figure 4. The x-axis shows the name of the 4 execution policies and the y-axis shows the execution time in seconds.

We have acquired some insights based on the task executions described above. Now, we can plot these execution times to compare the performance of different policies.

Then we plot the execution times of the different work schedules with varying chunk sizes. The graph is presented in Figure 5. The x-axis shows the different work schedules. The enclosed number indicates the chunk size of that bar. The y-axis indicates the execution time in seconds.

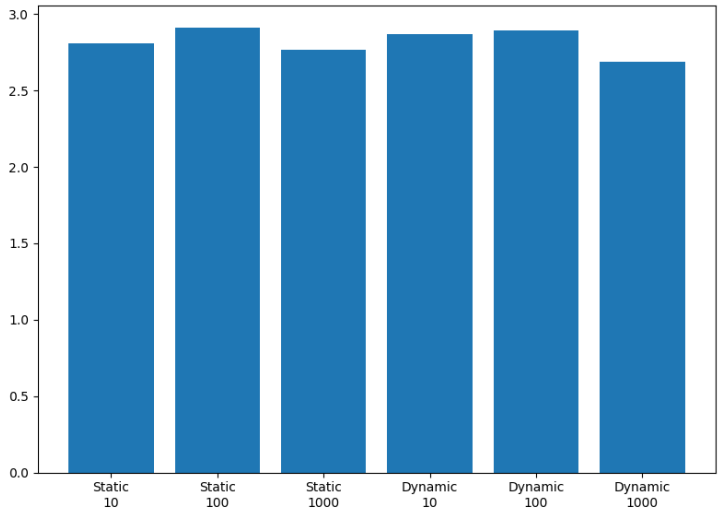

Fig. 5 Execution Time(sec) of Different Work Scheduling and Chunk Size

REFERENCES

[1] https://canvas.txstate.edu/courses/2052430/files/folder/Lec?preview=258405385

[2] https://www.ibm.com/docs/en/xl-fortran-linux/16.1.0?topic=openmp-omp-proc-bind

[3] https://www.bu.edu/tech/support/research/system-usage/running-jobs/parallel-batch/

[4] https://www.geeksforgeeks.org/graph-plotting-in-python-set-1/

[5] https://www.bu.edu/tech/support/research/system-usage/running-jobs/parallel-batch/

[6] https://www.rc.fas.harvard.edu/wpcontent/uploads/2016/04/Introduction-to-OpenMP.pdf