HW3: 50/50

Homework 3 – Mohammad Atif Faiz Afzal

HPC1

Date- 10/23/2014

Problem 1:

Below figure shows the execution time as a function of the number of terms used for calculating the value of pi using OpenMP. Number of cores were changed from 1 to 8 on 8 core node (L5520).

Idea on using OpenMP – The summation can be divided into the number of processors and all of the partial sum calculations will be independent. To get value of partial sums from each processor the 'reduction' function of OpemMP was used. The code is written in C language (in appendix1) and the code is self-explanatory. A SLURM script (in appendix 1) was written to make sure same node is used in all cases for comparison.

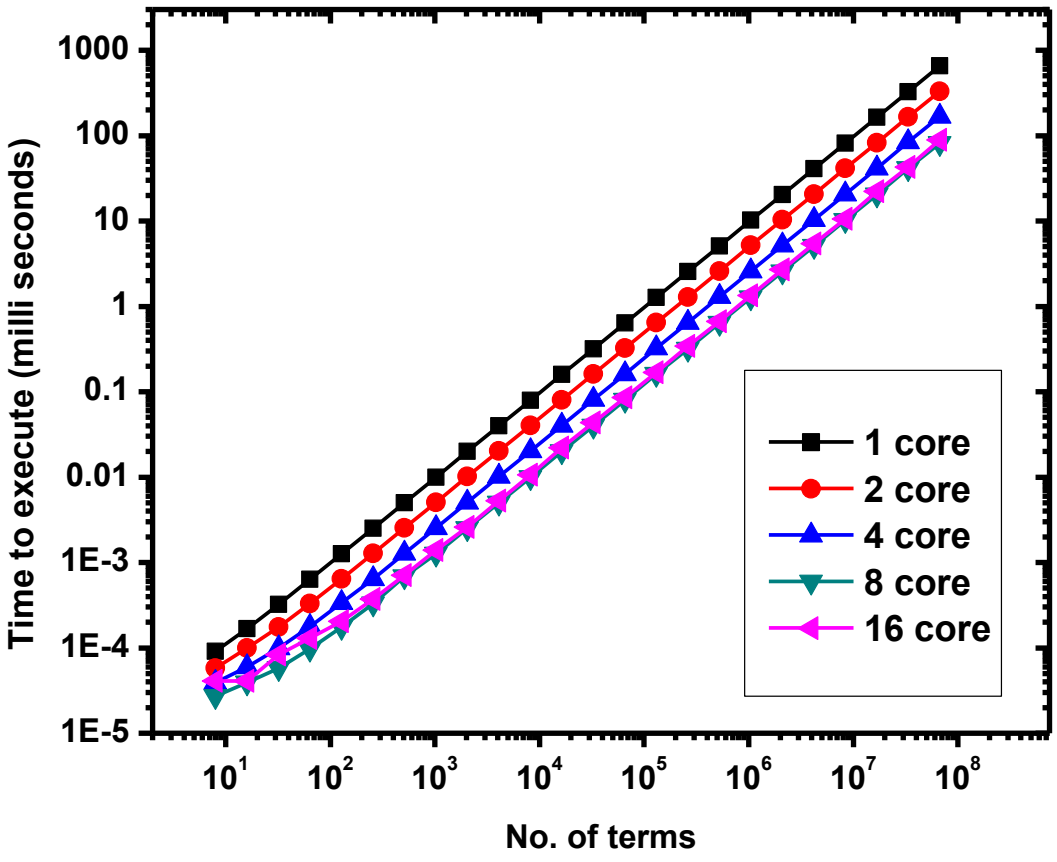


Figure 1 : Pi calculation using OpenMP on a 8 core (L5520) node available on CCR

Observations

- ➔ The execution time increases with the number terms being used.
- ➔ The execution time is less when higher number of cores is used.
- ➔ OpenMP works only on one machine

Yes, that is pretty much by design, OpenMP is for shared memory only

Just for curiosity I also ran the code by calling 16 cores on 2 nodes (the pink line in the figure 1). The performance in this case was equivalent to 8 cores. This is because OpenMP works only on shared memory cores and therefore works only on a single machine. For example, the below figure shows that in 4 nodes (total 32 cores) only one node (8 cores) is being used. When run on a single node, all the cores can be used for OpenMP.

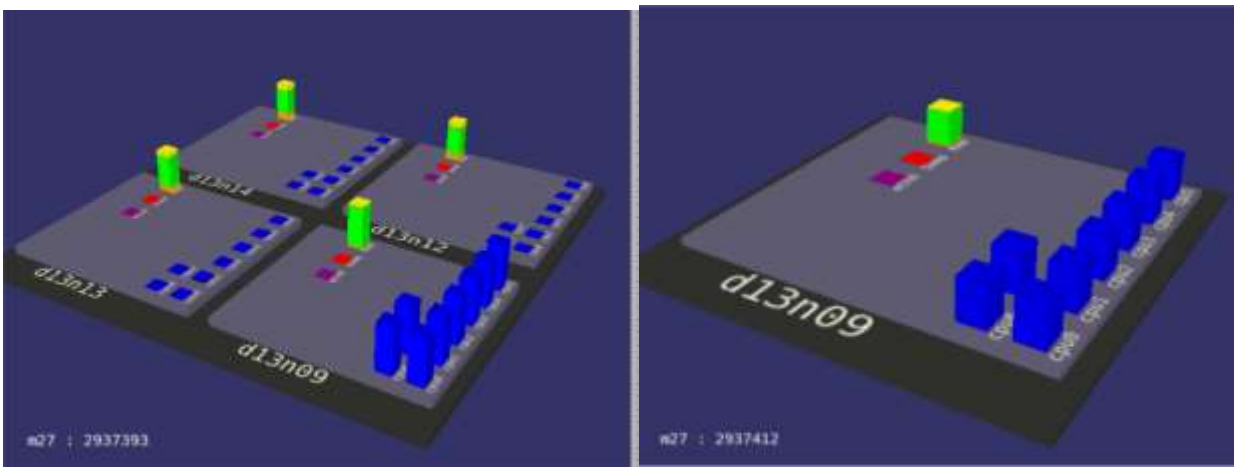


Figure 2: Snapshot of nodes showing which the cores that are being used. (a) When OpenMp run on 4 nodes. (b) When OpenMP run on single node

For comparison I also ran the code on 12 core nodes (E5645) that is available on CCR. Following the time execution as a function of number of terms

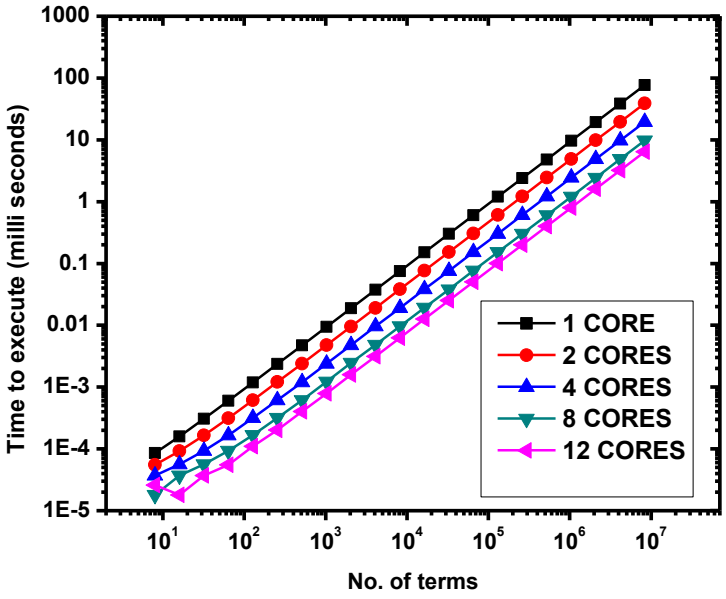


Figure 3 : Pi calculation using OpenMP on a 12 core (E5645) node available on CCR

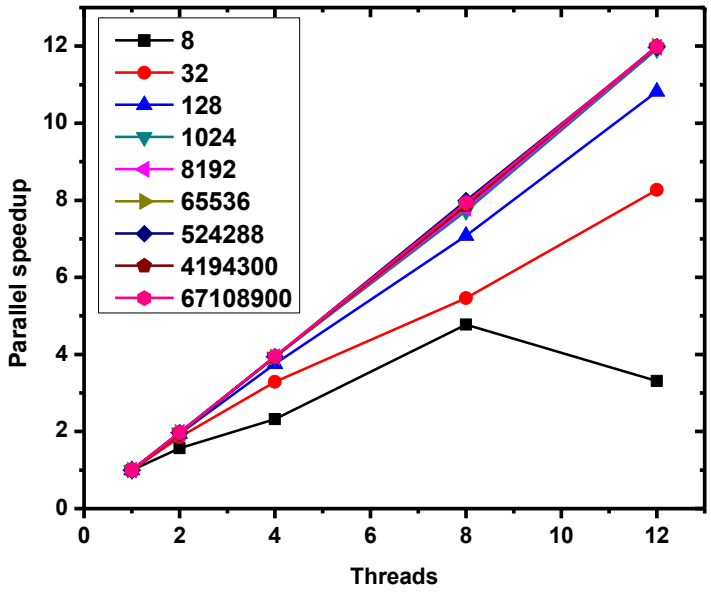
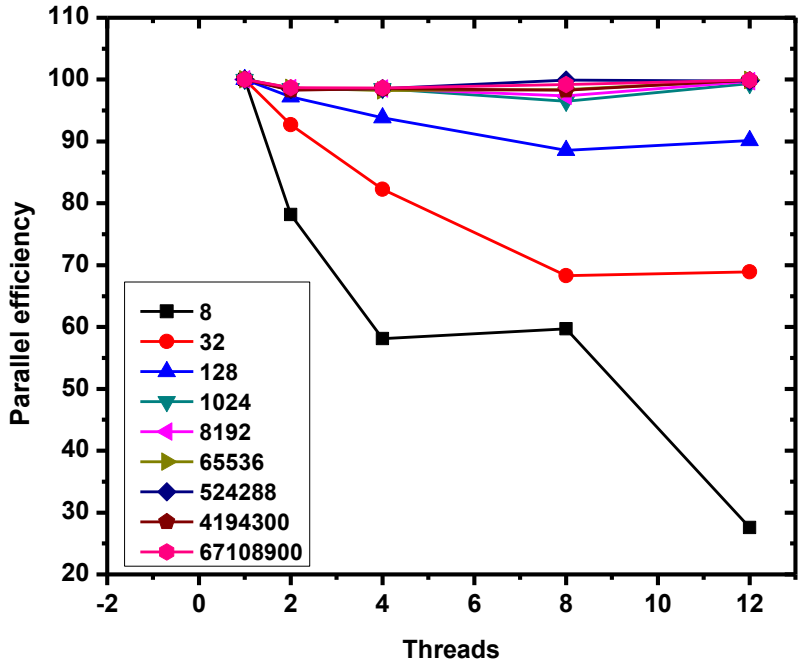


Figure 4 : OpenMP parallel speedup as a function of no. of threads and No. of terms (inset) on a 12 core (E5645) node available on CCR



Very nice plots

Figure 5 : OpenMP parallel efficiency as a function of no. of threads and No. of terms (inset) on a 12 core (E5645) node available on CCR

Notes-

Parallel speedup is calculated by the formula

you should actually turn OpenMP off for the sequential version

$$Parallel\ Speedup = \frac{Time\ for\ sequential\ running}{Time\ for\ running\ on\ N\ Threads}$$

For Sequential time, the run time of single thread OpenMp calculation is considered. If actual serial code run time is used it is possible that the parallel speedup can go below 1 for small no. of terms.

Parallel efficiency is calculated by dividing the parallel speedup with the no. of threads used.

Observations

- ➔ From figure 4, it can be seen that parallel speedup increases with the no. of threads
- ➔ For larger no. of terms the speedup is linear with the number of threads used. This is probably because the communication time is negligible for very large no. of terms
- ➔ It can be verified in parallel efficiency (figure 5) that the efficiency for small no. of terms is poor. As the no. of terms increases the efficiency reaches 100%

Problem 2:

Below figure shows the execution time as a function of the number of terms used for calculating the value of pi using MPI. Number of cores were changed from 1 to 12 on 12 core node (E5645).

Idea on using MPI – The idea is similar in MPI too. The summation is divided into the number of processors and all of the partial sum calculations will be independent. Here, I am calculating partial sums from each thread and sending the partial sums to a master thread where all the partial sums are received added to give the final sum. The code is written in C language (attached in appendix2) and the code is self-explanatory. A SLURM script (attached in appendix 2) was written to make sure same node is used in all cases for comparison.

If different machines are used, the processes are selected randomly. For example, figure 8 shows process distribution on two 12 core nodes when MPI is used. It can be seen that when 12 processes are running, the processes can run on any machine and use random cores from the available 24 cores.

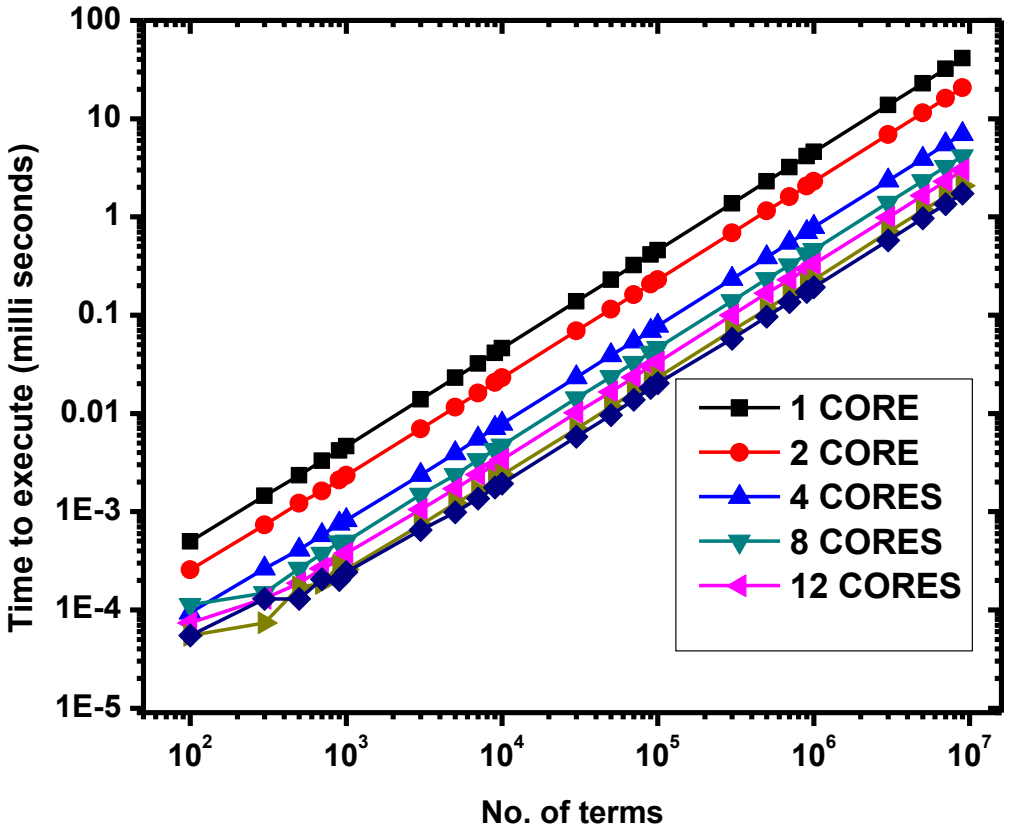


Figure 6 : Pi calculation using MPI on a 12 core (E5645) node available on CCR

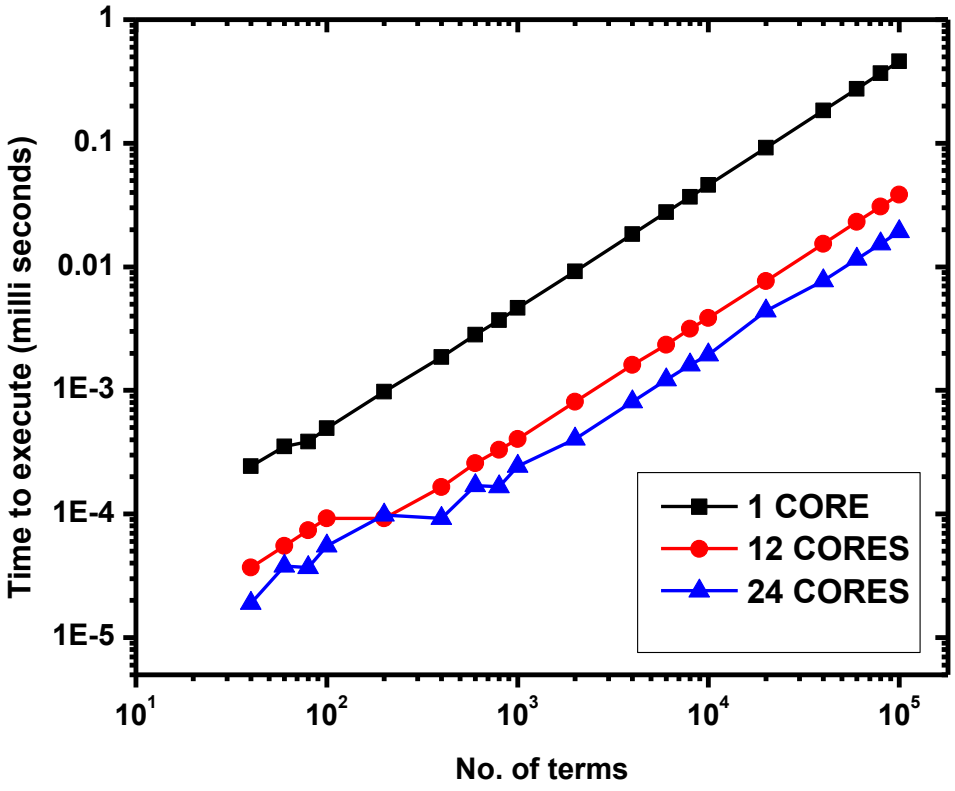


Figure 7 : Pi calculation using OpenMP on two 12 core nodes (E5645) node available on CCR. Comparison of 1 core, 12 core and 24 core

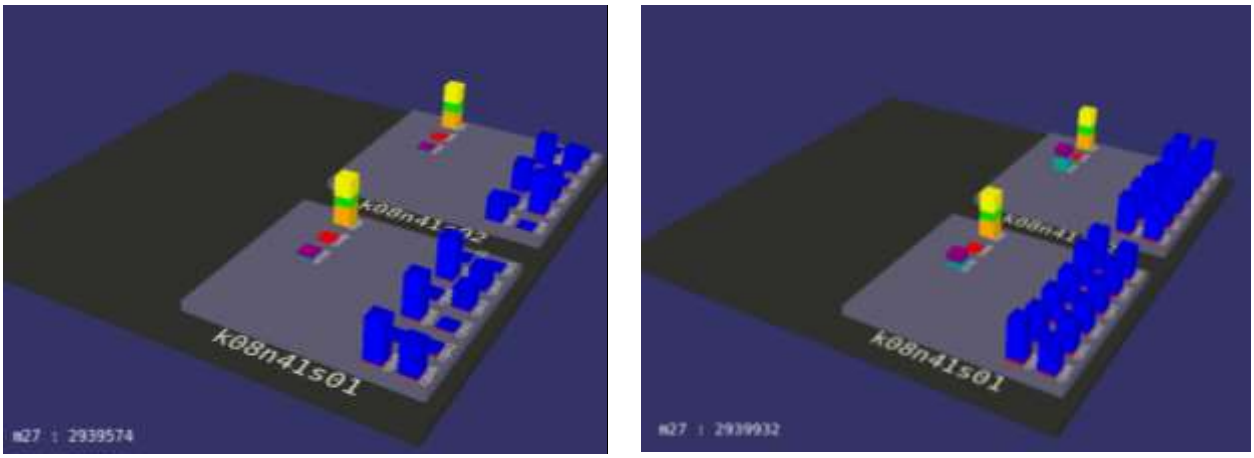


Figure 8 : Process distribution on two 12 core nodes when MPI is used. (a) 12 process distribution, (b) 24 process distribution

Observations

- ➔ The execution time increases with the number of terms being used. In figure 6, it can be seen that the variation of execution time with the terms is linear in log scales
- ➔ The execution time is less when higher number of cores is used.
- ➔ MPI can work on multiple machines as is shown in figure 8

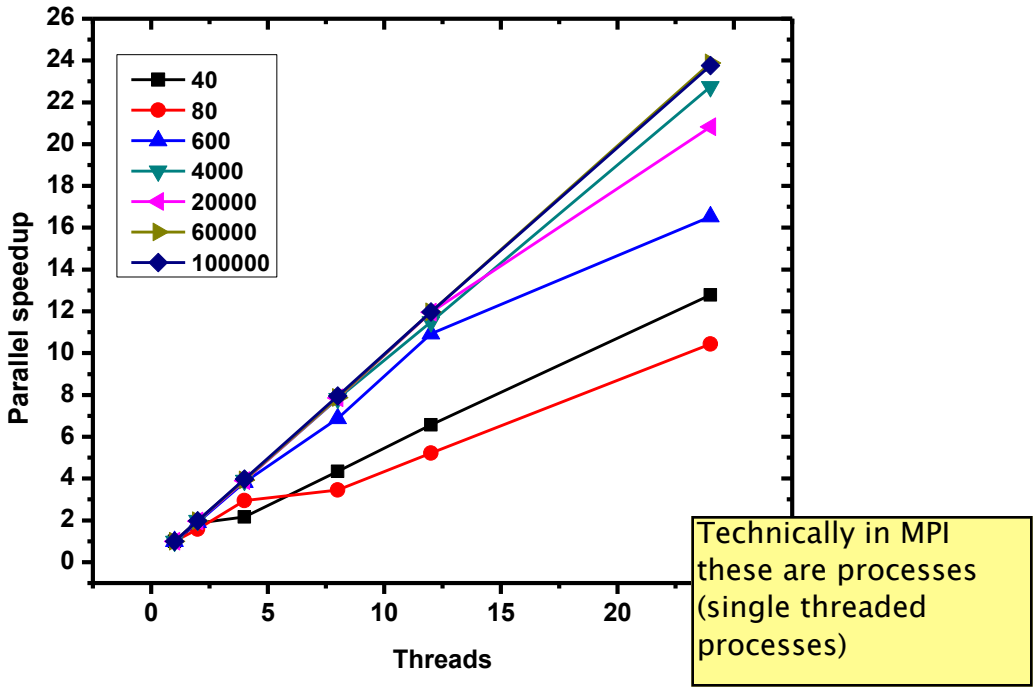


Figure 9 : MPI parallel speedup as a function of no. of threads and No. of terms (inset) on a 12 core (E5645) node available on CCR

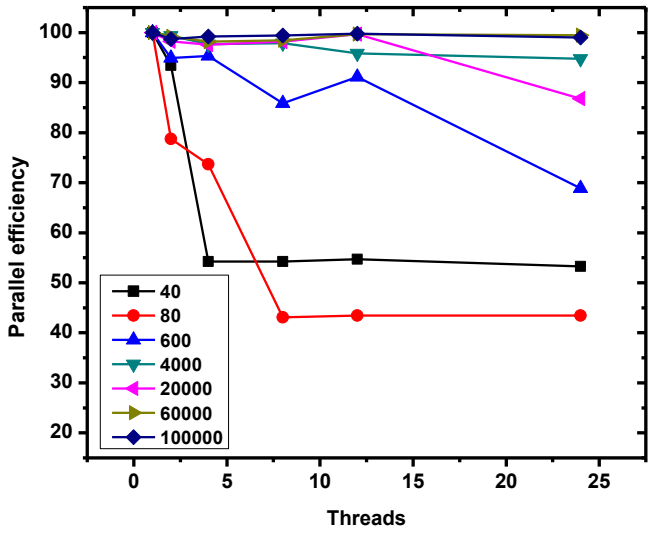


Figure 10 : MPI parallel efficiency as a function of no. of threads and No. of terms (inset) on a 12 core (E5645) node available on CCR

Please note that the values obtained for 1 thread is done by MPI with input thread number as 1 and not in a serial code.

Observations (similar observations are made in MPI too)

- ➔ From figure 9, it can be seen that parallel speedup increases with the no. of threads
- ➔ For larger no. of terms the speedup is linear with the number of threads used. This is probably because the communication time is negligible for very large no. of terms
- ➔ It can be verified in parallel efficiency (figure 10) that the efficiency for small no. of terms is poor. As the no. of terms increases the efficiency reaches 100%
- ➔ As the efficiency is very poor for smaller no. terms, it is better to use a serial code for small terms. However, the value of pi will not be accurate for small no. of terms.

For large no. of terms it is clear that both OpenMP and MPI give a superior performance.

Appendix 1

SLURM script for problem 1

```
#!/bin/sh
#SBATCH --partition=general-compute
#SBATCH --time=00:15:00
#SBATCH --job-name="prob1_nvt_core8"
#SBATCH --output=prob1_nvt_core8.out
#SBATCH --mail-user=m27@buffalo.edu
#SBATCH --mail-type=ALL
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --constraint=CPU-L5520

tic=`date +%s`
echo "Start Time = "`date`
echo "Loading modules ..."

ulimit -s unlimited
module list

echo "SLURM job ID        = "$SLURM_JOB_ID
echo "Working Dir         = "`pwd`
echo "Compute Nodes       = "`nodeset -e $SLURM_NODELIST`
echo "Number of Processors = "$SLURM_NPROCS
echo "Number of Nodes     = "$SLURM_NNODES

gcc -fopenmp -o prob1_8core prob1_8core.c

./prob1_8core

echo "All Done!"

echo "End Time = "`date`
toc=`date +%s`

elapsedTime=`expr $toc - $tic`
echo "Elapsed Time = $elapsedTime seconds"
```

Problem 1 Code in C language

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>

int main (int argc, char *argv[])
{
 int Nterms, Nreps, Npert;
 int Nthreads, myid, i, j, k, l;
 long double mysum,pi,ctime1,ctime2;

 for (l=8;l<=1000000;l=l*2)
  {
   Nterms=l;
   Nreps=1000000000/Nterms;

   for (i=8;i<=8;i=i+2)
        {
          Nthreads=i;

          omp_set_num_threads(Nthreads);
          mysum=0.0;
          /* mysum is global parameter for all processes so include in reduction */

#pragma omp parallel shared(Nthreads,Nterms, Nreps, Npert,Nthreadst,i,l) private(myid,k,j) reduction
(+:mysum)

          {
           ctime1 = omp_get_wtime(); /* using OpemMP time */

           myid = omp_get_thread_num();
           Npert=(Nterms/Nthreads); /* Note that Npert is an integer */

           if ((Npert%2)==0)
            {
                Npert++; /* Making sure that the Nterms/Nthreads is even so the Pi calculation is
accurate */
            }
           for (j=1;j<=Nreps;j=j++)
            {
                for (k = myid*(Npert)+1;k<=(myid+1)*(Npert);k=k+2)
                 {
```

```
                    mysum = mysum + (1.0/(2*k-1)) - (1.0/(2*k+1));
                  }

              }

            ctime2 = omp_get_wtime();
          }
          mysum = mysum*4/Nreps;
          printf("%d\t",Nterms);

          double time=(ctime2-ctime1)*1000/Nreps; /* calculate time in milli seconds */
          printf(" %f\n",time);


        }
    }
}
```

Appendix 2

SLURM script for problem 2

```
#!/bin/sh
#SBATCH --partition=debug
#SBATCH --time=00:15:00
#SBATCH --job-name="prob2_2"
#SBATCH --output=prob2_2.out
#SBATCH --mail-user=m27@buffalo.edu
#SBATCH --mail-type=ALL

#SBATCH --nodes=2
#SBATCH --cpus-per-task=12
#SBATCH --constraint=CPU-E5645

tic=`date +%s`
echo "Start Time = "`date`
echo "Loading modules ..."

module load intel-mpi intel
ulimit -s unlimited
module list

echo "SLURM job ID        = "$SLURM_JOB_ID
echo "Working Dir         = "`pwd`

echo "Compute Nodes       = "`nodeset -e $SLURM_NODELIST`
echo "Number of Processors = "$SLURM_NPROCS
echo "Number of Nodes     = "$SLURM_NNODES
echo "mpirun command      = "`which mpirun`

for i in $(seq 1 24); do

   for j in $(seq 40 20 100); do

         mpiicc -o prob2try1.impi prob2try1.c
         mpirun -np $i ./prob2try1.impi $j

   done

   for j in $(seq 200 200 1000); do

         mpiicc -o prob2try1.impi prob2try1.c
         mpirun -np $i ./prob2try1.impi $j
```

When in slurm you should use srun – the cpus-per-task setting likely allowed you to get the binding right, but srun is a safer bet

```
    done

    for j in $(seq 2000 2000 10000); do

            mpiicc -o prob2try1.impi prob2try1.c
            mpirun -np $i ./prob2try1.impi $j

    done

    for j in $(seq 20000 20000 100000); do

            mpiicc -o prob2try1.impi prob2try1.c
            mpirun -np $i ./prob2try1.impi $j
    done

done

echo "All Done!"

echo "End Time = "`date`
toc=`date +%s`

elapsedTime=`expr $toc - $tic`
echo "Elapsed Time = $elapsedTime seconds"
```

Problem 2 Code in C language

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>

# include "mpi.h"


int main (int argc, char *argv[])
{
  int Nthreads, myid, Nterms, Npert, Nreps,i,j,k,l, Nthreadst;
  double mysum,pi,ctime1,ctime2, start,end, MPItime, sum_all;

  MPI_Status status;

  Nterms = atoi(argv[1]);
  Nreps=100000/Nterms;

  MPI_Init ( &argc, &argv );
```

```
MPI_Comm_size ( MPI_COMM_WORLD, &Nthreads );
MPI_Comm_rank ( MPI_COMM_WORLD, &myid );

start=MPI_Wtime();

Npert=Nterms/Nthreads;
if ((Npert%2)==0)
 {
  Npert++; /* Making sure that the value of Nterms/Nthreads is odd */
 }

mysum = 0.0;

for (j=1;j<=Nreps;j=j++)
 {
  for (k = myid*(Npert)+1;k<=(myid+1)*(Npert);k=k+2)
        {
         mysum = mysum + (1.0/(2*k-1)) - (1.0/(2*k+1));
         /* here k is always an even number as I made sure that
            Npert is a always an odd and hence gives always gives
            right answer no matter what the Nterm value is  */
        }

 }

if ( myid != 0 )
 {
  MPI_Send ( &mysum, 1, MPI_DOUBLE, 0 , 1, MPI_COMM_WORLD );
  /* sending the the value of mysum calculated in each thread to the master thread */
 }
else
{
 sum_all = mysum;
 for ( i = 1; i < Nthreads; i++ )
  {
        MPI_Recv ( &mysum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status );
        /* recieveing the sum from all other threads to the master thread */
        sum_all = sum_all + mysum;
  }
}

if ( myid == 0 )
 {

  pi=4*sum_all/Nreps;

 }
```

```
  end=MPI_Wtime();

  if ( myid == 0 )
   {
     MPItime=(end-start)/Nreps*1000;
     printf ( "%d\t", Nterms);
     printf ( "Nthreads %d\t", Nthreads);
     printf ( "%f\n", MPItime);
   }

  MPI_Finalize ( );

  return 0;
}
```