Homework 5 – Mohammad Atif Faiz Afzal

HPC1

Date- 11/25/2014

Problem 1:

Part a

The serial code is written in C language and is attached in the Appendix 1. Intel (MPI ICC) compiler is used for all the compiling done in this assignment.

Part B

For understanding of performance the plots of execution time, parallel speed up and efficiency were plotted against the no. of cores used. All the runs were on a similar 12 core node (E5645) available on CCR.

Following are the plots for MPI

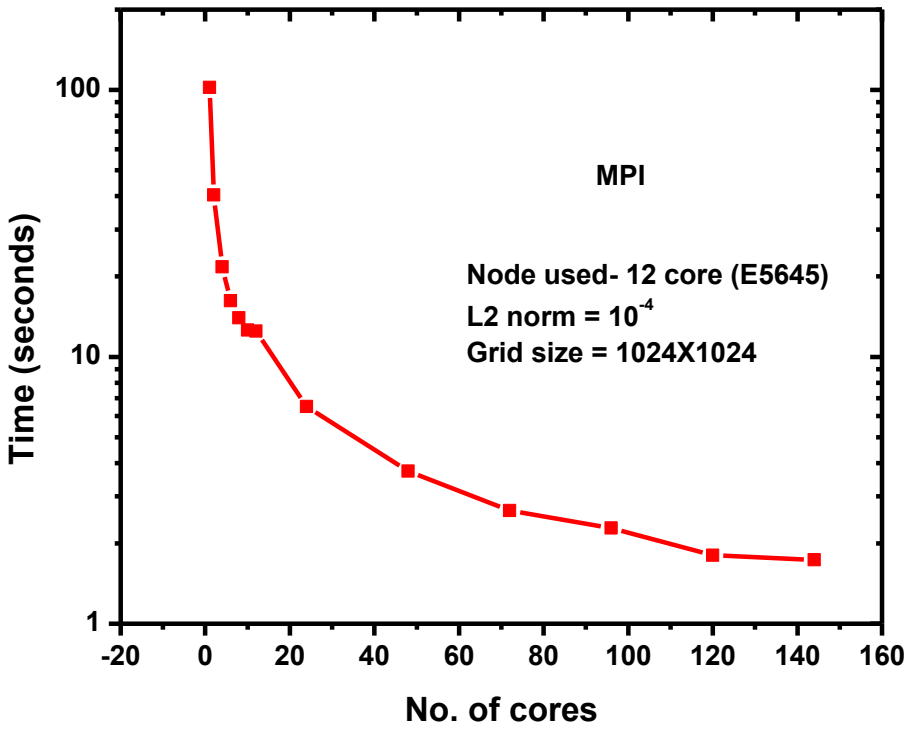

Figure 1: Execution time dependence on the no. of cores used for MPI (note that the time scale is a log scale)

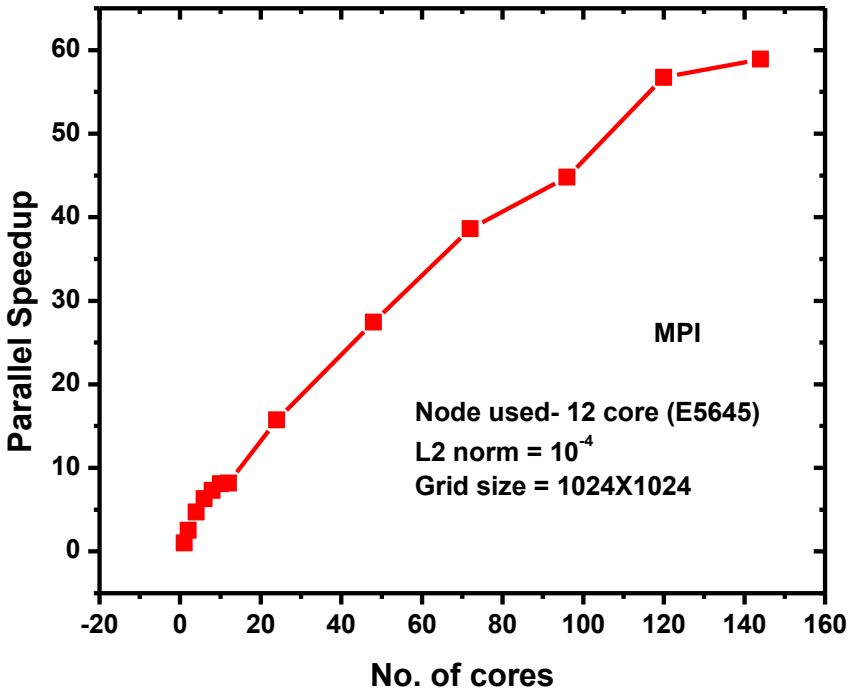

Figure 2: Parallel speedup with addition of cores

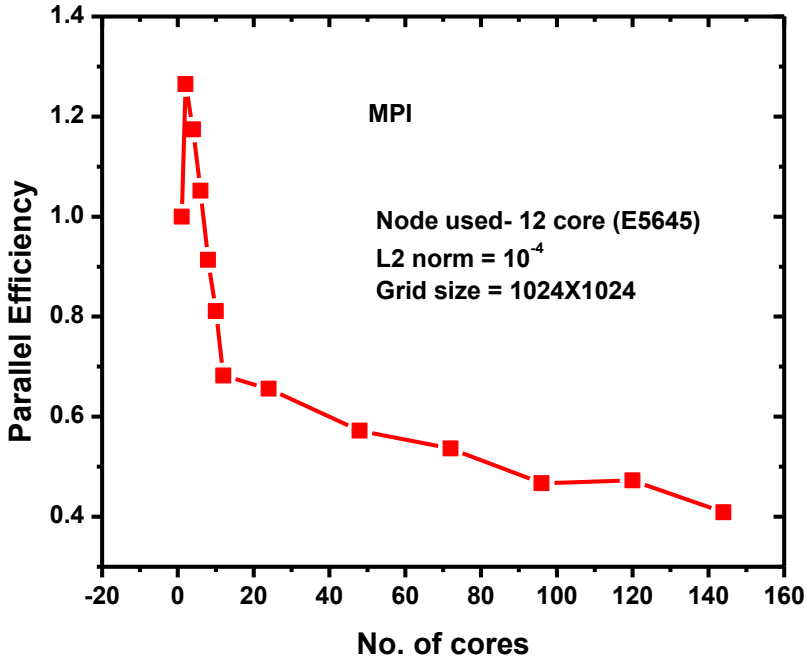

Figure 3: Parallel efficiency with addition of cores

Observations

1. It can be seen that the execution time decreases with increase in of cores.
2. From figure 1, it can be seen that the first 12 points in the plot (which correspond to a single node), it can be seen that the performance is not as good as it is expected. This is probably because as the cores in a single node is increasing, main memory is being used rather than L3 (more efficient) and therefore the performance is affected.
3. It can also be seen in figure 3 that the efficiency drops suddenly and becomes better as more number of nodes are being used.

To see what's happening only for 12 cores in a single node (figure 4)

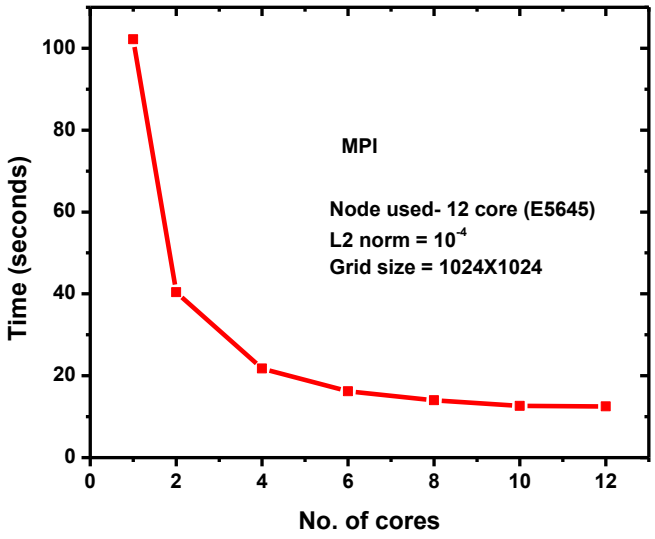

Figure 4: Execution time dependence on the no. of cores used for MPI

Parallel speedup is calculated by the formula

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

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

To check if the code is working properly, I used the slumjobvis command when the code was running on 4 nodes (E5645) and all the cores are being used. Figure 5 represents that all the cores were equally loaded

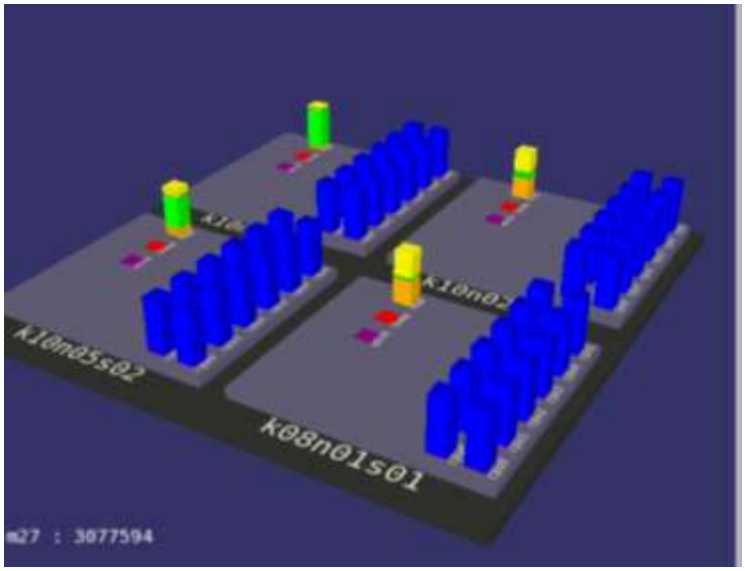

Figure 5: Slurmjobvis when my code was running on 4 E5645 nodes

Part C (for extra credit)

The code for hybrid MPI+OpenMP is written in C and is attached in Appendix 2. A slurm script to execute the code is also attached in Appendix 2.

Following are the plots for hybrid MPI + OpenMP code

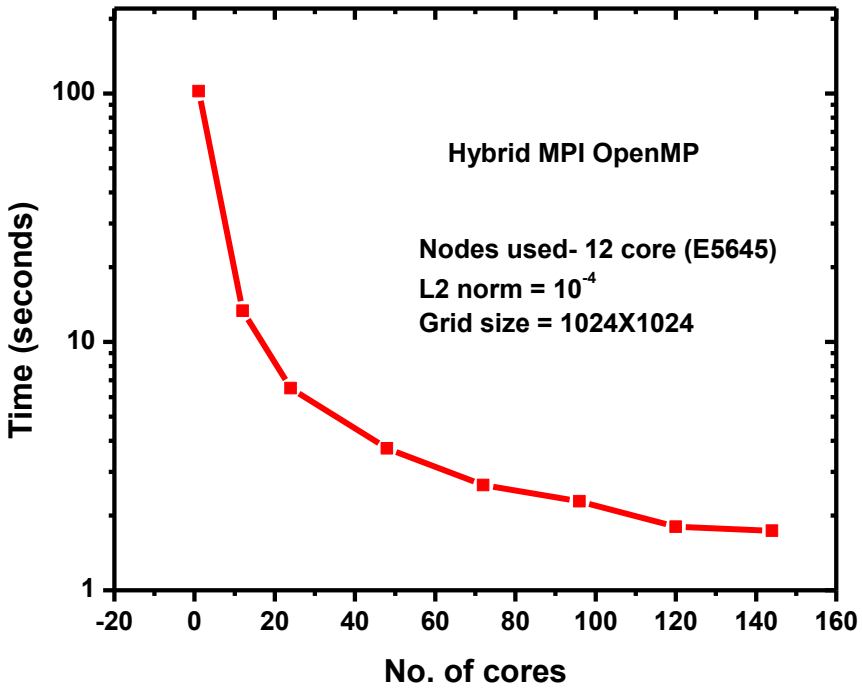

Figure 6: Execution time dependence on the no. of cores used for hybrid MPI+OpenMP (note that the time scale is a log scale)

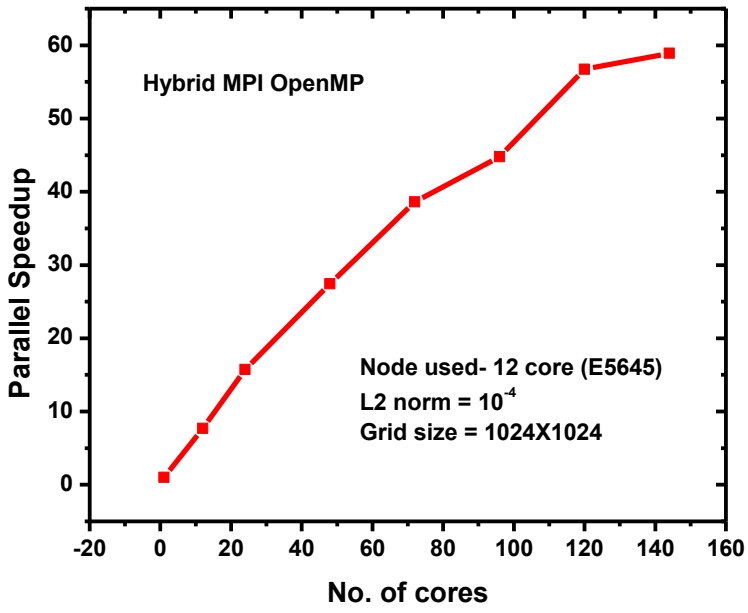

Figure 7: Parallel speedup with addition of cores

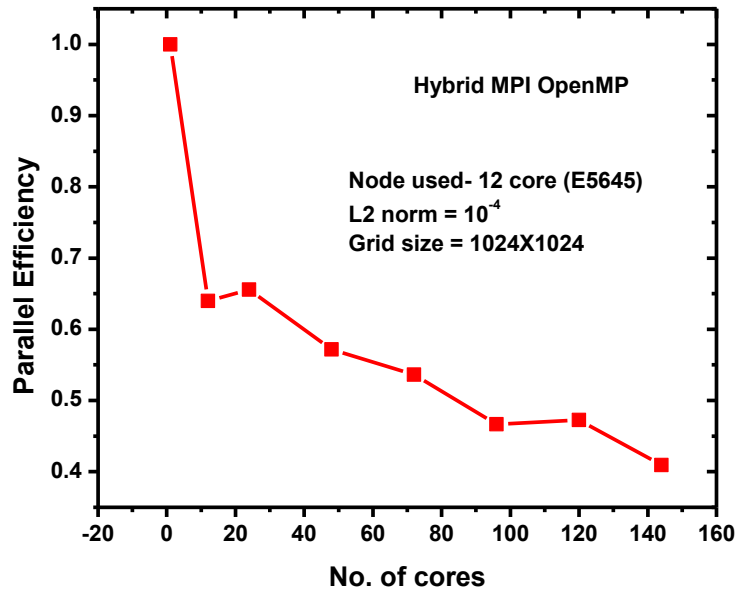

Figure 8: Parallel efficiency with addition of cores

Observation

There was not much difference in performance between the MPI and hybrid MPI OpenMP. I set the KMP_AFFINITY as compact

Appendix 1

C code for serial implementation of Jacobi iteration to solve the Laplace equation

```
#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 i,j,k,m,n,iter,pid,a,b,c,nproc;
 iter=20000;
 m=1024;n=1024;
 double sum,conv,r,w, double pi,start,end;
 w=.99;
 sum=0.0;
 pi= 3.14159265359;
 MPI_Status status;

 MPI_Init ( &argc, &argv );

 MPI_Comm_rank( MPI_COMM_WORLD, &pid );
 MPI_Comm_size( MPI_COMM_WORLD, &nproc );

 /* A different matrix (both intial and final matrix) is generated for each processor. The top and bottom rows of each
matrix is shared with adjacent top and bottom matrices respectively. */

 a = (pid)*(m-2)/nproc+1;
 b = (pid+1)*(m-2)/nproc;
 c = (b-a)+3; /* no.of rows of each matrix */

 double **fi_i = (double **) malloc(sizeof (double *) * c);
 double **fi_f = (double **) malloc(sizeof (double *) * c);

 for (i=0;i<c;i=i+1)
  {
    fi_i[i]=(double *) malloc(sizeof (double) * n);
    fi_f[i]=(double *) malloc(sizeof (double) * n);

  }

 for (i=1;i<c-1;i=i+1)
  {
    for (j=0;j<n;j=j+1)
                {
                  fi_i[i][j]=0.0;
                  fi_f[i][j]=0.0;
                }
  }
 /* assigning boundary conditions and interior points as 0.0 */
 if (pid==0){
  for (j=0;j<n;j=j+1)
                {
                    fi_i[0][j]=sin(pi*(j/((double)n-1)));
                    fi_f[0][j]=sin(pi*(j/((double)n-1)));
```

```
            }
}

if (pid==(nproc-1)){
 for (j=0;j<n;j=j+1)
            {
                fi_i[c-1][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
                fi_f[c-1][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
            }
}

start=MPI_Wtime();

for (k=1;k<=iter;k=k+1)
 {
  if (pid < nproc - 1)
            {
             MPI_Send( fi_i[c-2], n-1, MPI_DOUBLE, pid + 1, 0, MPI_COMM_WORLD );
             MPI_Recv( fi_i[c-1], n-1, MPI_DOUBLE, pid + 1, 1,MPI_COMM_WORLD, &status );
            }

  if (pid > 0)
            {
             MPI_Recv( fi_i[0], n-1, MPI_DOUBLE, pid - 1, 0,MPI_COMM_WORLD, &status );
             MPI_Send( fi_i[1], n-1, MPI_DOUBLE, pid - 1, 1, MPI_COMM_WORLD );
            }

  for (i=1;i<c-1;i=i+1)
            {
             for (j=1;j<n-1;j=j+1)
              {
               fi_f[i][j]=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_i[i+1][j-1]+fi_i[i+1][j+1])/4.0;

               /* r=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_f[i+1][j-1]+fi_f[i+1][j+1])/4.0; */
               /* fi_f[i][j]=r*w+(1-w)*fi_i[i][j]; */

               sum = sum + (fi_f[i][j]-fi_i[i][j])*(fi_f[i][j]-fi_i[i][j]);
              }
            }

  for (i=1;i<c-1;i=i+1)
            {
             for (j=1;j<n-1;j=j+1)
              {
               fi_i[i][j]=fi_f[i][j];
              }
            }

  MPI_Allreduce( &sum, &conv, 1, MPI_DOUBLE, MPI_SUM,MPI_COMM_WORLD );

  conv=sqrt(conv);

  if (conv < 0.0001)
            {
             break;
            }

  sum=0.0;
```

```
  }

end=MPI_Wtime();

double time=(end-start)*1000; /* calculate time in milli seconds */

if (pid==0)
  {
  printf("%d\t",nproc);
  printf(" %f\n",time);
  }

MPI_Finalize( );

return 0;
}
```

Appendix 2

C code for hybrid MPI+OpenMP

```
#include <omp.h>
#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 i,j,k,m,n,iter,pid,a,b,c,nproc;
 iter=20000;
 m=1024;n=1024;
 double sum,conv,r,w;
 w=.99;
 sum=0.0;
 long double pi,start,end;
 pi= 3.14159265359;
 MPI_Status status;

 MPI_Init ( &argc, &argv );

 MPI_Comm_rank( MPI_COMM_WORLD, &pid );
 MPI_Comm_size( MPI_COMM_WORLD, &nproc );

 a = (pid)*(m-2)/nproc+1;
 b = (pid+1)*(m-2)/nproc;
 c = (b-a)+3;

 double **fi_i = (double **) malloc(sizeof (double *) * c);
 double **fi_f = (double **) malloc(sizeof (double *) * c);

 for (i=0;i<c;i=i+1)
  {
   fi_i[i]=(double *) malloc(sizeof (double) * n);
   fi_f[i]=(double *) malloc(sizeof (double) * n);
  }

 for (i=1;i<c-1;i=i+1)
  {
   for (j=0;j<n;j=j+1)
              {
               fi_i[i][j]=0.0;
               fi_f[i][j]=0.0;
              }
  }

 if (pid==0){
  for (j=0;j<n;j=j+1)
              {
                  fi_i[0][j]=sin(pi*(j/((double)n-1)));
                  fi_f[0][j]=sin(pi*(j/((double)n-1)));
              }
 }
```

```
 if (pid==(nproc-1)){
  for (j=0;j<n;j=j+1)
                {
                    fi_i[c-1][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
                    fi_f[c-1][j]=sin(pi*(j/((double)n-1)))*exp(-pi);
                }
 }

 start=MPI_Wtime();

 for (k=1;k<=iter;k=k+1)
  {
    if (pid < nproc - 1)
                MPI_Send( fi_i[c-2], n-1, MPI_DOUBLE, pid + 1, 0, MPI_COMM_WORLD );

    if (pid > 0)
                MPI_Recv( fi_i[0], n-1, MPI_DOUBLE, pid - 1, 0,MPI_COMM_WORLD, &status );

    if (pid > 0)
                MPI_Send( fi_i[1], n-1, MPI_DOUBLE, pid - 1, 1, MPI_COMM_WORLD );

    if (pid < nproc-1)
                MPI_Recv( fi_i[c-1], n-1, MPI_DOUBLE, pid + 1, 1,MPI_COMM_WORLD, &status );

#pragma omp parallel default(shared) private(i,j,r)
    {

#pragma omp for schedule (static) reduction (+:sum)

    for (i=1;i<c-1;i=i+1)
                {
                  for (j=1;j<n-1;j=j+1)
                    {
                      fi_f[i][j]=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_i[i+1][j-1]+fi_i[i+1][j+1])/4.0;

                      /* r=(fi_i[i-1][j-1]+fi_i[i-1][j+1]+fi_f[i+1][j-1]+fi_f[i+1][j+1])/4.0; */
                      /* fi_f[i][j]=r*w+(1-w)*fi_i[i][j]; */

                      sum = sum + (fi_f[i][j]-fi_i[i][j])*(fi_f[i][j]-fi_i[i][j]);
                    }
                }
#pragma omp for schedule (static)

    for (i=1;i<c-1;i=i+1)
                {
                  for (j=1;j<n-1;j=j+1)
                    {
                      fi_i[i][j]=fi_f[i][j];
                    }
                }
    }

    MPI_Allreduce( &sum, &conv, 1, MPI_DOUBLE, MPI_SUM,MPI_COMM_WORLD );

    conv=sqrt(conv);

    if (conv < 0.0001)
                {
```

```
                    break;
                  }

     sum=0.0;

   }

  end=MPI_Wtime();

  double time=(end-start)*1000; /* calculate time in milli seconds */

  if (pid==0)
   {
   printf("%d\t",nproc);
   printf(" %f\n",time);
   }

  MPI_Finalize( );

  return 0;
}
```

SLURM script

```
#!/bin/sh
#SBATCH --nodes=10
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=12
#SBATCH --exclusive
#SBATCH --constraint=CPU-E5645
#SBATCH --partition=general-compute
#SBATCH --time=01:00:00
#SBATCH --mail-type=END
##SBATCH --mail-user=m27@buffalo.edu
#SBATCH --output=mpi_4_1.out
#SBATCH --job-name=mpi_jacobi_4

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

echo "Loading modules ..."

module load intel/15.0 intel-mpi
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`

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export KMP_AFFINITY=,compact
#export I_MPI_DEBUG=4
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so

mpiicc -openmp -o mpi_1.impi mpi_1.c

for i in $(seq 1 10); do
   echo $i
   srun -n $i ./mpi_1.impi
done


echo "All Done!"

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

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