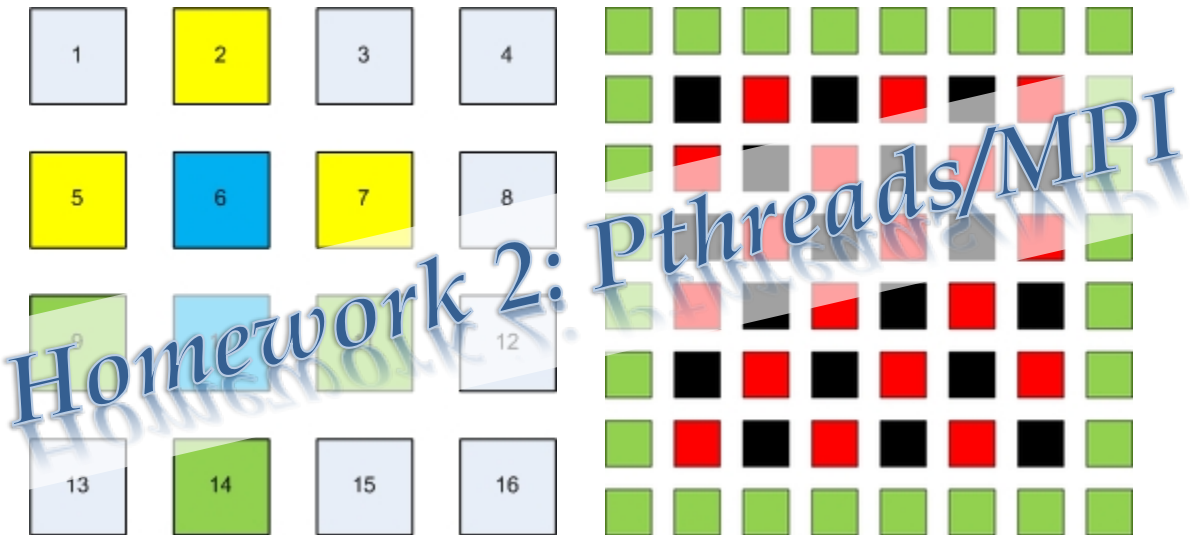
CS 757 - Parallel Computer Architecture

Spring 2014



THE UNIVERSITY of WISCONSIN MADISON

Aman Rakesh Chadha
(906 835 4597)

Ranjini Mysore Nagaraju
(906 924 4441)

1 Overview

This assignment consisted of writing explicit parallel programs using two different programming styles - shared-memory and message passing. The Pthreads API was used for shared-memory programming and MPI API was used for message passing.

2 Problem 1: Pthreads Implementation

a Source code

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>
#include <sys/errno.h>
#include "hwtimer.h"

int num_threads;

pthread_mutex_t   SyncLock; /* mutex */
pthread_cond_t    SyncCV; /* condition variable */
int               SyncCount; /* number of processors at the barrier so far */

int xdim, ydim, timesteps;
int** grid;

void printGrid(int** grid, int xdim, int ydim){
  int i, j;
  for(i=0;i<ydim;i++){
    for(j=0;j<xdim;j++)
      printf("%-6d  ",grid[i][j]);
    printf("\n");
  }
}

void Barrier()
{
  int ret;

  pthread_mutex_lock(&SyncLock); /* Get the thread lock */
  SyncCount++;
  if(SyncCount == num_threads) {
    ret = pthread_cond_broadcast(&SyncCV);
    SyncCount = 0;
    assert(ret == 0);
  } else {
    ret = pthread_cond_wait(&SyncCV, &SyncLock);
    assert(ret == 0);
  }
  pthread_mutex_unlock(&SyncLock);
}

/* The function which is called once the thread is allocated */
void* ocean(void* tmp)
{
  /* each thread has a private version of local variables */
  int tid = *((int*) tmp);

  int ret;

  /* ********************* Execute Job ******************************** */
  for (int t = 0; t < timesteps; t++) {
    for(int i=(((xdim-2)/num_threads)*tid)+1;
              i<=(((xdim-2)/num_threads)*(tid+1)); i++) {
      int offset = (i+t)%2;
      for(int j=1+offset; j<ydim-1; j+=2) {
       grid[i][j] = (grid[i][j] + grid[i-1][j] + grid[i+1][j]
                     + grid[i][j-1] + grid[i][j+1])/5;
```

```
      }
    }
    Barrier();
  }
}

int main(int argc, char** argv)
{
  pthread_t*     threads;
  pthread_attr_t attr;
  int            ret;
  int            dx;

  int i, j, t;

  if (argc! = 5) {
    printf("The Arguments you entered are wrong.\n");
    printf("./serial_ocean <x-dim> <y-dim> <timesteps> <num_threads>\n");
    return EXIT_FAILURE;
  } else {
    xdim = atoi(argv[1]);
    ydim = atoi(argv[2]);
    timesteps = atoi(argv[3]);
    num_threads = atoi(argv[4]);
  }

  /* Initialize array of thread structures */
  threads = (pthread_t *) malloc(sizeof(pthread_t) * num_threads);
  assert(threads != NULL);

  /* Initialize thread attribute */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); // sys manages contention

  /* Initialize mutexes */
  ret = pthread_mutex_init(&SyncLock, NULL);
  assert(ret == 0);

  /* Initialize condition variable */
  ret = pthread_cond_init(&SyncCV, NULL);
  assert(ret == 0);
  SyncCount = 0;

  /* declare and initialize grid*/
    grid = (int**) malloc(ydim*sizeof(int*));
    int *temp = (int*) malloc(xdim*ydim*sizeof(int));
    for (i=0; i<ydim; i++) {
      grid[i] = &temp[i*xdim];
    }
    for (i=0; i<ydim; i++) {
        for (j=0; j<xdim; j++) {
         grid[i][j] = rand();
        }
    }

  hwtimer_t timer;
  initTimer(&timer);
  startTimer(&timer); //start the time measurement before the algorithm starts
  int* id = (int*) malloc(sizeof(int)*num_threads);

  for(dx = 0; dx < num_threads; dx++) {
    /* ************************************************************
     * pthread_create takes 4 parameters
     *  p1: threads(output)
     *  p2: thread attribute
     *  p3: start routine, where new thread begins
     *  p4: arguments to the thread
     * ************************************************************ */
    id[dx]=dx;
    ret = pthread_create(&threads[dx], &attr, ocean, (void*) &id[dx]);
```

```
    assert(ret == 0);
  }

  /* Wait for each of the threads to terminate */
  for(dx=0; dx < num_threads; dx++) {
    ret = pthread_join(threads[dx], NULL);
    assert(ret == 0);
  }

  stopTimer(&timer); //End the time measurement here since the algorithm ended

  printGrid(grid,xdim,ydim);
  printf("Total Execution time: %lld ns\n", getTimerNs(&timer));

  pthread_mutex_destroy(&SyncLock);
  pthread_cond_destroy(&SyncCV);
  pthread_attr_destroy(&attr);

  free(temp);
  free(grid);

  return 0;
}
```

b Comments on implementation

We started off by initializing an array of thread structures and thread attributes using `pthread_attr_init()` and `pthread_attr_setscope()`.

We also initialized mutexes and condition variables using `pthread_cond_init()`.

We then declared and initialized the grid to random values using `rand()`.

Once the grid was initialized, we initialized and started the timer.

At this stage, we spawned threads using `pthread_create()`.

We then programmed threads to work on their own chunk of grid which they calculate using their own thread ID.

We inserted Barrier after each timestep to avoid data races to grid values.

We then wait for the threads to finish execution using `pthread_join()`.

Post this, we stopped the timer and reported the timing value on the screen.

Finally, we freed memory allocated for the mutex, condition variable, thread attributes and the grid.

c Arguments on the validity of implementation

We obtain similar observations as in the case of our parallelized implementation using OpenMP. Our implementation seeks to average values of each grid location over a series of timesteps. If the operation is done over a sufficiently large number of such timesteps, we expect to obtain a grid with its values distributed over a lesser range (due to convergence) as compared to the original grid. This is similar to a Gaussian Blur operation in Image Processing.

Indeed, when we initialized the grid to a 130x130 size, and performed the averaging operation over a large number of timesteps (100, in our case), we obtained the expected results. Here, X Axis represents the X co-ordinate of the grid location and Y Axis represents the Y co-ordinate of the grid location. Figure 1 shows the input grid to our ocean implementation.

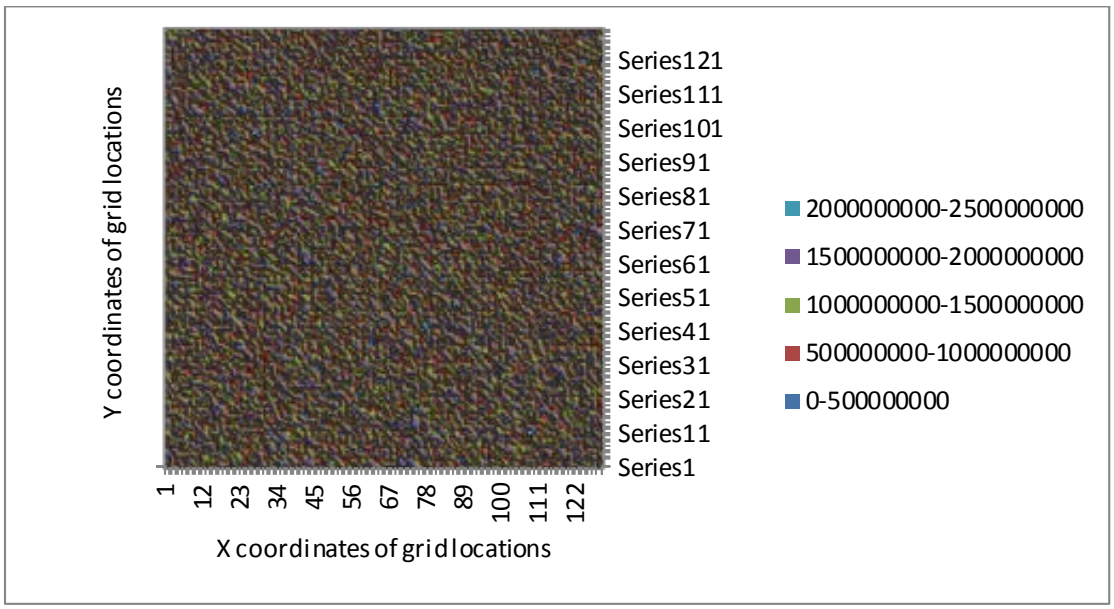


Figure 1: Input grid

After performing the averaging operation over 100 timesteps, we see the values at the grid locations (apart from those at the edges) tend to converge to a single value. This is a direct product of the averaging operation. Figure 2 shows the output grid from our implementation.

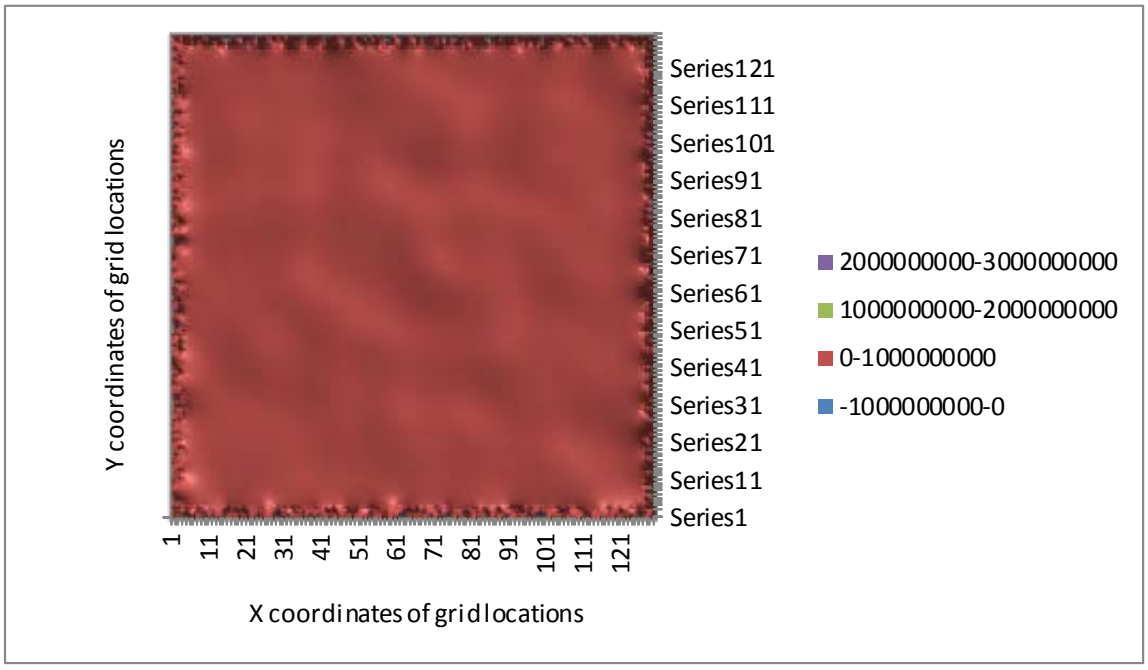


Figure 2: Output grid after 100 time steps

Further, upon performing the averaging operation over 10000 timesteps, the values get even closer to converging to a single value as shown in Figure 3.

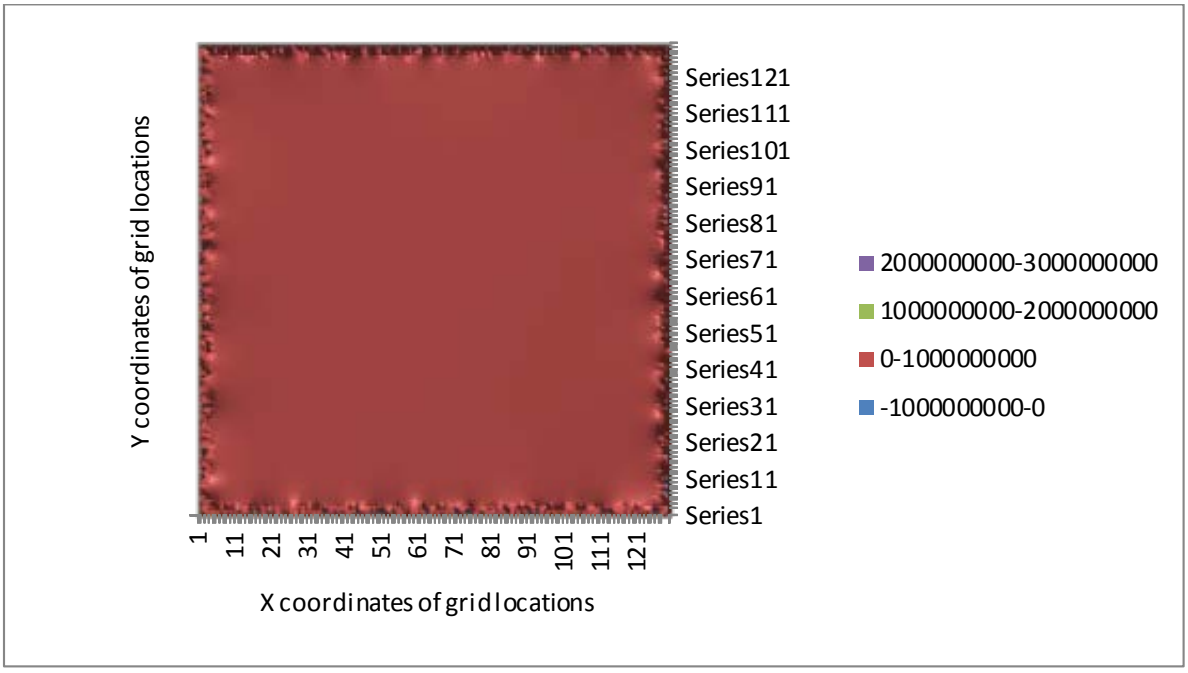


Figure 3: Output grid after 10000 time steps

The results of the parallel Pthreads implementation were matched against those of our serial implementation. As expected, the resultant grid values with and without the parallelization were equal.

3 Problem 2: MPI Implementation

a Source code

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BUFLEN 8194*8
#include "hwtimer.h"

void printGrid(int** grid, int xdim, int ydim){
  int i, j;
  for(i=0; i<ydim; i++){
    for(j=0; j<xdim; j++)
      printf("%-6d  ", grid[i][j]);
    printf("\n");
  }
}

int main(int argc, char* argv[])
{
    int i, myid, numprocs, next, rc, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int recv_buffer1[BUFLEN],recv_buffer2[BUFLEN];
    double startwtime, endwtime;
    MPI_Status status;
    MPI_Request request;


    int xdim,ydim,timesteps;
    int** grid;


    if (argc!=4) {
      printf("The Arguments you entered are wrong.\n");
      printf("./serial_ocean <x-dim> <y-dim> <timesteps> <num_procs>\n");
      return 0;
```

```
} else {
  xdim = atoi(argv[1]);
  ydim = atoi(argv[2]);
  timesteps = atoi(argv[3]);
}

/* Initialize the MPI execution environment */
MPI_Init(&argc,&argv);

/*
 * Determines the size of the group associated
 * with a communicator
 *
 *     int MPI_Comm_size ( comm, size )
 *     MPI_Comm comm; //  communicator (handle)
 *     int *size; // number of processes in the group of comm (integer)
 */
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

/*
 * Determines the rank of the calling process in the
 * communicator
 *
 *      int MPI_Comm_rank ( comm, rank )
 *      MPI_Comm  comm; //  communicator (handle)
 *      int      *rank; //  rank of the calling process in group of
 *                      //  comm (integer)
 */
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

/*
 * Gets the name of the processor
 *
 *      int MPI_Get_processor_name( name, resultlen )
 *      char *name;  //  A unique specifier for the actual
 *                   //  (as opposed to virtual) node.
 *      int *resultlen; //  Length (in characters) of the name
 */
MPI_Get_processor_name(processor_name,&namelen);

/* Initializing the grid. Could have initialized in process 0 and did a broadcast.
Just chose to initialize in each process since they would be the same anyway*/
grid = (int**) malloc(ydim*sizeof(int*));
int *temp = (int*) malloc(xdim*ydim*sizeof(int));
for (int i=0; i<ydim; i++) {
  grid[i] = &temp[i*xdim];
}
for (int i=0; i<ydim; i++) {
  for (int j=0; j<xdim; j++) {
grid[i][j] = rand();
  }
}
hwtimer_t timer;
/*Start Timers*/
if(myid==0){
initTimer(&timer);
startwtime = MPI_Wtime();
startTimer(&timer);
}
for(int t=0;t<timesteps;t++){

  if(numprocs==1){
/*If there is only one process, still send and receive messages to itself
 to model MPI message passing overhead*/
if (myid == 0){
  MPI_Send(grid[(((xdim-2)/numprocs)*myid)+1], xdim*sizeof(int), MPI_INT, myid,
99,MPI_COMM_WORLD);
  MPI_Recv(recv_buffer2,xdim*sizeof(int),MPI_INT,myid,99,MPI_COMM_WORLD,&status);
  MPI_Send(grid[(((xdim-2)/numprocs)*(myid+1))], xdim*sizeof(int), MPI_INT, myid,
99,MPI_COMM_WORLD);
```

```
    MPI_Recv(recv_buffer1,xdim*sizeof(int),MPI_INT,myid,99,MPI_COMM_WORLD,&status);

    for(int i=(((xdim-2)/numprocs)*myid)+1; i<=(((xdim-2)/numprocs)*(myid+1)); i++){
      int offset = (i+t)%2;
      for(int j=1+offset; j<ydim-1; j+=2) {
        grid[i][j] = (grid[i][j] + grid[i-1][j] + grid[i+1][j]
                + grid[i][j-1] + grid[i][j+1])/5;
      }
    }
  }
}
    else{
  /*There is more than one process*/
  if (myid == 0 || myid == numprocs-1){
    /*Identify first and last process*/
    if(myid==0){
      /*First process sends and receives messages only to its next process to share
  rows in their respective borders*/
      MPI_Send(grid[(((xdim-2)/numprocs)*(myid+1))], (xdim)*sizeof(int), MPI_INT,
  myid+1, 99, MPI_COMM_WORLD);
      MPI_Recv(recv_buffer2,(xdim)*sizeof(int),MPI_INT,myid+1,99,MPI_COMM_WORLD,&status);
      for(int i=(((xdim-2)/numprocs)*myid)+1; i<=(((xdim-2)/numprocs)*(myid+1));
  i++){
        int offset = (i+t)%2;
        for(int j=1+offset; j<ydim-1; j+=2) {
           if(i==(((xdim-2)/numprocs)*(myid+1))){
           grid[i][j] = (grid[i][j] + grid[i-1][j] + recv_buffer2[j]
                       + grid[i][j-1] + grid[i][j+1])/5;
           }
           else   {
                 grid[i][j] = (grid[i][j] + grid[i-1][j] + grid[i+1][j]
                             + grid[i][j-1] + grid[i][j+1])/5;
           }
        }
      }
    }
    else{
      /*Last process sends and receives messages only to its previous process*/
     MPI_Recv(recv_buffer1,(xdim)*sizeof(int),MPI_INT,myid-1,99, MPI_COMM_WORLD,
     &status);
     MPI_Send(grid[(((xdim-2)/numprocs)*myid)+1], (xdim)*sizeof(int),MPI_INT,myid-1,
     99, MPI_COMM_WORLD);
      for(int i=(((xdim-2)/numprocs)*myid)+1; i<=(((xdim-2)/numprocs)*(myid+1)); i++){
        int offset = (i+t)%2;
        for(int j=1+offset; j<ydim-1; j+=2) {
           if(i==(((xdim-2)/numprocs)*myid)+1){
           grid[i][j] = (grid[i][j] + recv_buffer1[j] + grid[i+1][j]
                       + grid[i][j-1] + grid[i][j+1])/5;
           }
           else   {
                 grid[i][j] = (grid[i][j] + grid[i-1][j] + grid[i+1][j]
                             + grid[i][j-1] + grid[i][j+1])/5;
           }
        }
      }
    }
  }
  else{
    /*Rest of the processes send and receives messages to both their previous and
  next processes*/
    MPI_Recv(recv_buffer2,(xdim)*sizeof(int),MPI_INT,myid-1,99, MPI_COMM_WORLD,
  &status);
    MPI_Send(grid[(((xdim-2)/numprocs)*myid)+1],(xdim)*sizeof(int),MPI_INT,myid-
  1,99,MPI_COMM_WORLD);
    MPI_Send(grid[(((xdim-2)/numprocs)*(myid+1))], (xdim)*sizeof(int), MPI_INT,
  myid+1, 99, MPI_COMM_WORLD);
    MPI_Recv(recv_buffer1,(xdim)*sizeof(int),MPI_INT,myid+1,99,MPI_COMM_WORLD,&status);
    for(int i=(((xdim-2)/numprocs)*myid)+1; i<=(((xdim-2)/numprocs)*(myid+1)); i++){
      int offset = (i+t)%2;
      for(int j=1+offset; j<ydim-1; j+=2) {
```

```
          if(i==(((xdim-2)/numprocs)*(myid+1))){
       grid[i][j] = (grid[i][j] + grid[i-1][j] + recv_buffer2[j]
                   + grid[i][j-1] + grid[i][j+1])/5;
          }
          else if(i==(((xdim-2)/numprocs)*myid)+1){
       grid[i][j] = (grid[i][j] + recv_buffer1[j] + grid[i+1][j]
                   + grid[i][j-1] + grid[i][j+1])/5;

          }
          else{
       grid[i][j] = (grid[i][j] + grid[i-1][j] + grid[i+1][j]
                   + grid[i][j-1] + grid[i][j+1])/5;
          }
        }
      }
    }
  }
}

    /*
     * Blocks until all process have reached this routine.
     *
     *   int MPI_Barrier ( comm )
     *   MPI_Comm comm;  //  communicator (handle)
     */

    MPI_Barrier(MPI_COMM_WORLD);
    if(myid!=0){
             //All processes send their chunk of grid back to process 0
             MPI_Send(grid[(((xdim-2)/numprocs)*myid)+1],xdim*((ydim-
             2)/numprocs)*sizeof(int),MPI_INT,0,99,MPI_COMM_WORLD);
    }

    if(myid==0){
      for(int i =1; i<=numprocs-1; i++){
       //process 0 receives parts of grid from all processes and assembles the final
       grid
             MPI_Recv(&(grid[(((xdim-2)/numprocs)*i)+1][0]),(xdim)*((ydim-
             2)/numprocs)*sizeof(int),MPI_INT,i,99,MPI_COMM_WORLD,&status);
        }
      }

    MPI_Barrier(MPI_COMM_WORLD);
    if (myid == 0)
      {
    //Stop and print time values and grid from process 0
    stopTimer(&timer);
    endwtime = MPI_Wtime();
    printf("Total Execution time: %lld ns\n", getTimerNs(&timer));
    printf("wall clock time = %f\n",
           endwtime-startwtime);
    //printGrid(grid,xdim,ydim);
      }

    /*Free allocated heap memory*/
    free(temp);
    free(grid);

    /*
     * Terminates MPI execution environment
     *
     *   int MPI_Finalize()
     */
    MPI_Finalize();
}
```

b Comments on implementation

We started off by initializing the MPI execution environment using `MPI_Init()`.

We determined the size of the group associated with a communicator using `MPI_Comm_size()`.

We then determined the rank of the calling process in the communicator using `MPI_Comm_rank()`.

We obtain the name of the processor using `MPI_Get_processor_name()`.

We then declared and initialized the grid to random values using `rand()`.

Once the grid was initialized, we initialized and started the timer using `MPI_Wtime()`.

Now, we begin with the parallelization part using `MPI_Send()` and `MPI_Recv()`.

> If there is only one process, we still send and receive messages to itself to model MPI message passing overhead.
>
> If there are more than one processes, first process sends and receives messages only to its next process to share rows in their respective borders. Similarly, last process sends and receives messages only to its previous process. Rest of the processes send and receives messages to both their previous and next processes.

A `MPI_Barrier()` is inserted at this stage which blocks until all process have reached till this breakpoint. Post this, all processes send their chunk of grid back to process 0, which in turn, assembles the final grid.

After another `MPI_Barrier()`, process 0 prints the grid and execution time value.

We then freed memory allocated for the grid.

Finally, we terminated the MPI execution environment using `MPI_Finalize()`.

c Arguments on the validity of implementation

A similar procedure was followed to ensure validity of implementation as followed in the case of the Pthreads implementation. The results of the parallel MPI implementation were matched against that of the serial implementation. As expected, the resultant grid values with and without the parallelization were equal.

4 Problem 3: Analysis of Ocean

a Comparison of Pthreads, MPI and OpenMP implementations for a 4098x4098 grid

For 100 time steps, we plot the normalized (versus the sequential version of Ocean) speedups of our implementations with N=[1,2,4,8,16] threads for a 4098x4098 ocean.

We observed that the Pthread version of our ocean implementation scales well with increasing number of threads. For Case with one thread, the graph shows slight loss in performance due to overhead of creating and launching threads. This loss in performance can be considered as overhead is Pthread programming model. We observed a slight decrease in performance at 8 threads which we are at loss to reason with.

We did not observe any speedup using MPI. Though MPI version with four processes performs better than with two processes, the overhead of message passing outweighs the benefit of parallel execution. Hence both of them perform worse than simple serial execution.

OpenMP however provides scaling in between that of MPI and Pthreads. However as thread count increases beyond 8 we observed performance degradation. This is due to the fact that ale machine has 8 cores. While using 16 threads, they can be context switched out of a core. In the event that a thread context switched out from one core gets scheduled on another core, its working set, TLB entries etc need to migrate to the new core. This is not very unlikely to happen. By setting environment variable to set thread affinity that binds threads to processor, we observed better results. But the results are not plotted in this manner.

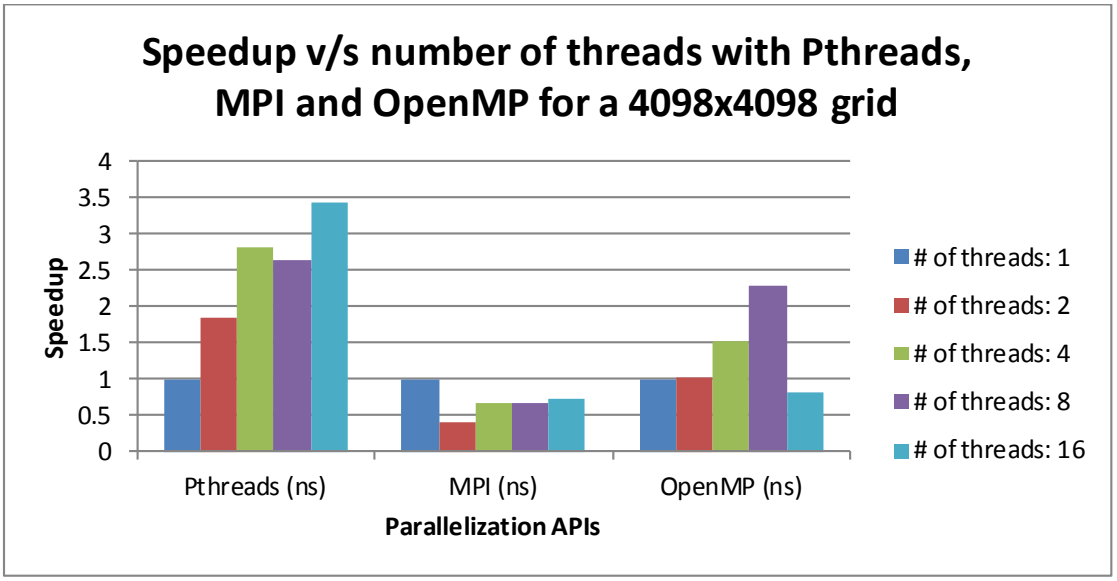


Figure 4: Speedup v/s number of threads for a 4098x4098 grid

b Comparison of Pthreads, MPI and OpenMP implementations for a 8194x8194 grid

For 100 time steps, we plot the normalized (versus the sequential version of Ocean) speedups of our implementations with N=[1,2,4,8,16] threads for a 8194x8194 ocean.

Here we observe similar behavior from Pthreads as explained before. However, performance gap between 4 and 8 threads has decreased which we did not understand why.

MPI still has high overheads which are more than the benefits provided by parallelization. However with 8 processes we see slight speedup. Beyond this, performance decreases due to increase number of messages exchanged between processes.

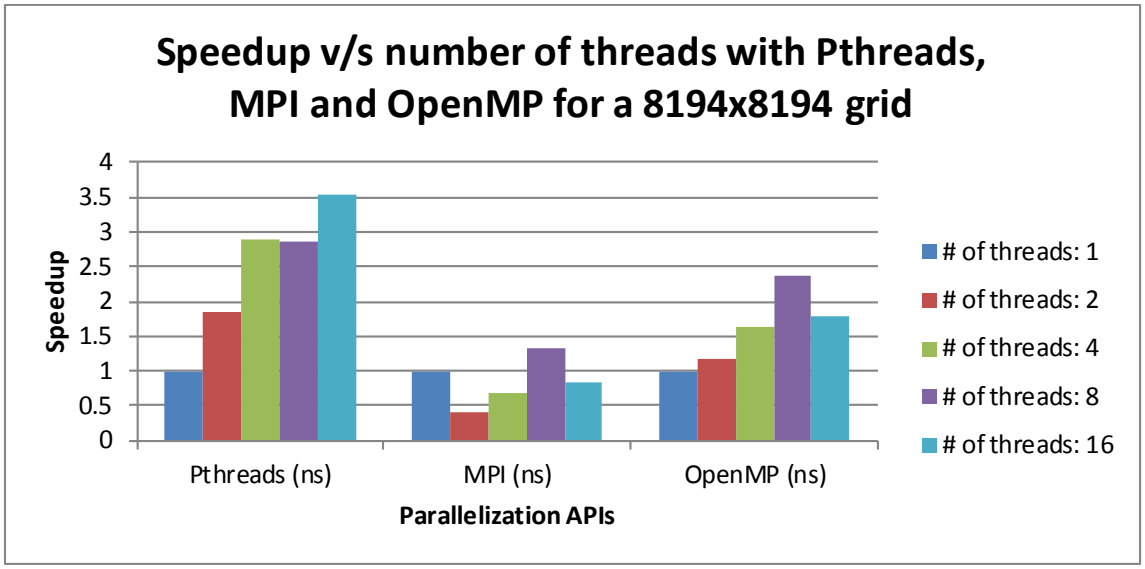


Figure 5: Speedup v/s number of threads for a 8194x8194 grid

Note that the N=1 case represents the sequential version of Ocean, not the parallel version using only 1 thread.

6 Comparison of Parallelization Approaches

a OpenMP

Parallelization with OpenMP was simple and straightforward. It was even incremental so that loops could be parallelized one after the other removing bugs in each step separately. This allowed more scope for optimization.

b Pthreads

Parallelization with Pthreads was slightly more complicated since each thread had to be explicitly created, alongwith the grid explicitly divided and assigned to each thread. We had to deal with calculating expressions for each thread with its thread ID. We also had to work on synchronizing all threads after each step. A lot of programmer involvement is required

c MPI

Parallelization with MPI is much more complicated. It involves all complexities described with Pthreads as well as Message sending and receiving overheads. Further it requires the grid to be dismantled and sent to each process in the beginning as well as reassembling the grid at the end. All these responsibilities fall on the programmer which makes using this model difficult.

d Easiest approach in terms of programming effort

Since processes in message-passing programs do not share a single address space, one has to explicitly split data structures between various processes. This requires programmering effort. On the other hand, a shared memory system is relatively easy to program since all processors share a single view of data and they only need to be synchronized. Communication between processors is implicit.

Thus, OpenMP saved us a lot of programming effort compared to Pthreads and MPI. While the Pthreads programming model is slightly complicated, it is still easier to use than MPI.

e Comparison of programming effort, performance and scalability for each implementation

Programming effort: In addition to the reasons mentioned in part (d), OpenMP was found to be the simplest to implement due to its 'incremental parallelization' features. A simple pragma when added to the Ocean code would enable us in carrying out the parallelization of the code. Further OpenMP provides numerous options to optimize the implementation easily. On the other hand, Pthreads required multiple functions to carry out thread creation, joining etc. Also, Pthreads requires explicit synchronization and is inflexible to do runtime optimizations.

Performance: Given a grid size and upon spawning the same number of threads, we observed that Pthreads offered more speedup compared to OpenMP. MPI in general performed poorly.

Scalability: All the models offered scalability in the order of Pthreads, OpenMP, MPI (though MPI did not offer performance improvement, it showed scalability).

7 Appendix: Data obtained from Experiments

Problem 3				
Comparison of Pthreads, MPI and OpenMP implementations for a 4098x4098 grid				
4098 4098 100				
Threads	**Pthreads (ns)**	**MPI (ns)**	**OpenMP (ns)**	**Serial (ns)**
1	2742182686	2742182686	2742182686	2742182686
2	1492298834	6741213494	2665349460	
4	970451685	4021420724	1801705255	
8	1040974805	3997433723	1200339821	
16	773212820	3723556164	3290786859	
Comparison of Pthreads, MPI and OpenMP implementations for a 8194x8194 grid				
8194 8194 100				
Threads	**Pthreads (ns)**	**MPI (ns)**	**OpenMP (ns)**	**Serial (ns)**
1	10929037468	10929037468	10929037468	10929037468
2	5951435821	27535393911	9256707774	
4	3770595520	16160021297	6724573342	
8	3902788594	8174024234	4628920034	
16	3076837348	13270621820	6105573914	
NORMALIZED DATA W.R.T. SERIAL IMPLEMENTATION				
4098 4098 100				
Threads	**Pthreads**	**MPI**	**OpenMP**	**Serial**
1	1	1	1	1
2	1.83755600656055	0.40677879263736	1.02882669876992	
4	2.82567667034346	0.681894005676786	1.52199294440089	
8	2.63424501037756	0.685985778881668	2.28450530260297	
16	3.43895487296844	0.736441875783131	0.833290882543906	
8194 8194 100				
Threads	**Pthreads**	**MPI**	**OpenMP**	**Serial**
1	1	1	1	1
2	1.83636987723807	0.396908702425862	1.18066139007836	
4	2.89849107655016	0.676300932228901	1.62523879392317	
8	2.85965497956808	1.33704490653948	2.36103397503626	
16	3.55203614357583	0.823551271088818	1.79000985360932	