CS 4444 – Report for Homework 4

Jerry Sun (ys7va)

1 Problem Description

The goal of this assignment is to write a parallel C program using pthreads to optimize the runtime for a lightly twisted classic heated plate problem. The experiment is going to be tested on a shared memory machine(Hermes). The key optimization we need to observe is the effect of using NUMA library to bind threads and corresponding memories to avoid hotspot effect.

The halo problem I am dealing with here has the setting as below:

- The interior cells should all initialized to the same temperature (50 degrees)
- The border cells fixed at a specific temperature (0 degrees along the top and left sides and 100 degrees along the bottom and right sides)
- There is a single internal condition (col=6500 & row=4500) in which a single cell is held at 1000 degrees
- In each time step of the simulation, the temperature of each cell is computed by averaging the temperatures of the four neighboring cells in the previous time step

The original sequential program provided has a runtime of 2404 seconds for a 10000 iterations on $10000 * 10000$ plates using one single thread.

2 Result Summary

For 10000 iterations on $10000 * 10000$ plates with the hotspot, my final parallel program works, and takes 114 seconds to finish with 64 threads excluding the final image rendering. The overall speedup is 21 times faster than the original sequential program.

3 Metadata

3.1 Software

SLURM: version 14.11

GNU bash: version 4.1.2

gcc: version 4.4.7

POSIX Threads, Numa library

3.2 Hardware

64-core Hermes machine with four 16-core AMD Opteron 6276 server processors. Total RAM is 256G with three cache levels. A 6 MB L3 cache shared by 8 cores, a 2 MB L2 cache shared be 2 cores and a 16 KB L1 cache for each core.

4 Approach

For this assignment, it is required to parallelize the particular halo problem described above. The main design regarding the PThreads implementation is regarding the threads and memory allocation on different nodes using NUMA.

NOTE: According to NUMA library, the number of nodes on Hermes is 8 in total. Therefore, we only bind threads and memories on a node-level.

The detail performance comparison, and analysis can be found in optimization section.

4.1 PThreads

Since there are only 64 cores in Hermes machines, the maximum number of threads that can run in parallel using PThreads is 64. I then divide all the works in a row major order evenly. And pass the start and the end row into each thread, so that each thread can know which part of the heated plate does it need to compute. Also note that since there is a hotspot in the center of the plate, the program also needs to determine which thread that is going to handle the hotspot. The program then also pass a hotspot_id into each thread. Therefore, the corresponding thread will update the hotspot during each iteration.

Note that for each iteration, all the threads need to be synchronized. The program uses Pthread_barrier to implement this idea. The basic implementation logic behind barrier is that all the threads will stop at the barrier, unless a pre specified number of threads has hit the barrier. Here we just need to set that number equal to the number of total threads.

4.2 Naive Memory Allocation

For the unoptimized version, the memory of a given cell is allocated together at once in main function. Then it assigns the actual memory location of the first element of each row to its corresponding row pointer. However, this turns out to be inefficient, and the detail comparison can be found later. The detailed implementation $**$*allocate_cells_naive* can be found in Appendix *halo.c*

5 Performance

In this section, I will give a brief analysis for the data I retrieved for naive version using 1, 2, 4, 8, 16, 32, 64 threads. For time measurement, I use the *time()* function in ‘time.h‘ library. All binary executables are compiled by gcc using -O3 flag on. For timing, it counts the total runtime from the start of the program, to the point after all threads have been joined, excluding final image rendering.

The program run five times for each number of threads, and I will take the average of that result for further analysis.

5.1 Runtime data collection

Followed are all the timings recorded during testing on Hermes based on different number of threads using unoptimized memory allocation.

Table 1: Naive Memory Allocation Runtime

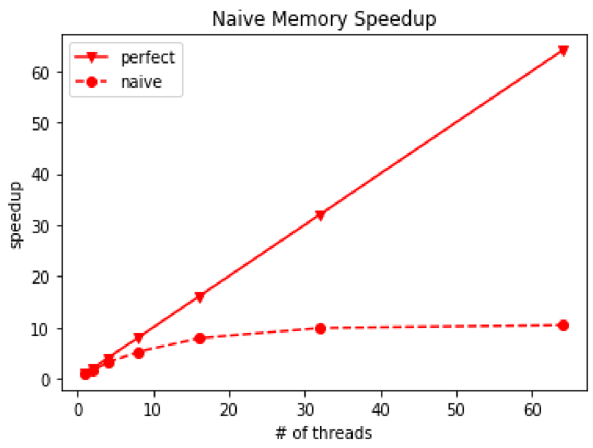
num of threads	1	2	4	8	16	32	64
	2410	1532	754	461	307	247	237
	2359	1533	756	460	305	243	228
	2357	1533	757	463	304	243	227
	2440	1535	760	464	304	249	227
	2453	1540	760	461	306	246	237
average	2404	1533	757	462	305	243	230

5.2 Analysis

The following charts are the speedup we get for different number of threads, given the average runtime of 1 thread as baseline.

Table 2: Naive Memory Allocation Speedup

num of threads	1	2	4	8	16	32	64
speedup	1.0	1.57	3.18	5.20	7.88	9.89	10.45



From this chart, we can find that while at first, doubling threads does give solid speedup. However, with an increasing number of threads, the performance improvement it gains when doubling the number of threads decreases really fast. This is due to the fact that the Hermes machine actually contains 4 sockets, and when allocating the memory using naive implementation, all the memory is allocated on one socket. This causes a hotspot and the intense memory accesses to that particular socket from remote cores greatly dragged the overall performance in this halo problem. The solution to solve this problem is by using NUMA library to control the memory allocation, so that it can reduce this problem and provide a better result in performance.

6 Optimization

6.1 Numa Memory Allocation

For the optimized version, we still assigns all the row pointer together. However, different from the original version, the program actually allocates the whole memory onto different nodes evenly using *num_alloc_onnode*. This function works the same as malloc, except that it can specified the node the memory is going to be allocated on. Therefore, the memory can then be evenly distributed. However, since we still allocate all the row pointer together as global variables in *main* function, the accessing pattern is the same for threads computation. Also when using NUMA Memory Allocation, the program also binds the corresponding thread onto the node, which the rows it needs to compute are allocated.

More specifically the whole process works as below:

- Divide the rows evenly into num_of_threads groups by storing an array containing all the rows that separate two groups.
- Allocate all the row pointer of two cells in the main memory.
- Unlike the unoptimized version, allocate each group mentioned above separately onto different numa_node(group i is allocated on node i % 8) since there are 8 numa_nodes in total on Hermes
- Launch thread function
- Inside thread function, run *numa_run_onnode* to bind the corresponding thread onto the specific node where its memory is allocated.
- Join all threads and print the output

The detailed implementation $**$*allocate_cells_numa* and $*$*thread_compute* can be found in Appendix *halo.c*.

6.2 Runtime data collection

Followed are all the timings recorded during testing on Hermes based on different number of threads using optimized memory allocation, with all other settings identical to the timing with unoptimized memory allocation.

Table 3: Optimized Memory Allocation

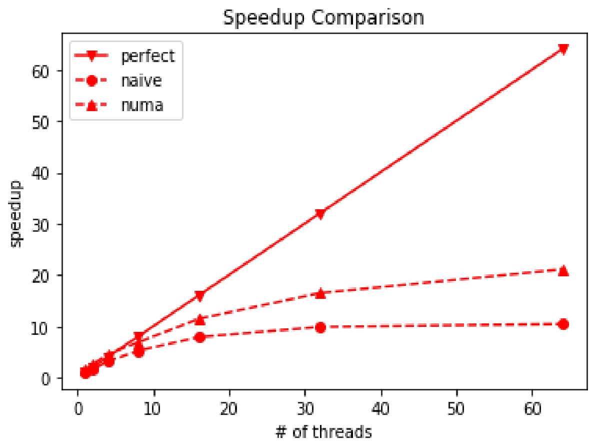
num of threads	1	2	4	8	16	32	64
	1652	944	520	348	205	143	114
	1651	933	522	355	209	147	111
	1644	935	529	353	211	148	111
	1665	937	522	353	211	142	107
	1655	935	533	350	212	139	113
average	1654	936	527	352	210	146	111

6.3 Analysis

The following charts are the speedup we get for both versions with different number of threads, given the average runtime of 1 thread as baseline.

Table 4: Naive Memory Allocation Speedup

num of threads	1	2	4	8	16	32	64
naive speedup	1.0	1.57	3.18	5.20	7.88	9.89	10.45
optimized speedup	1.45	2.57	4.54	6.87	11.45	16.47	21.09



From these two charts, we can find the while the optimized version still perform much worse than the ideal speedup, it actually has a better performance than the unoptimized version. However, we can still see a significant decrease in performance gain using increasing number of threads. The reason is that, while we do manage to reduce number of remote memory accesses, the intense memory access has not changed. With an increasing number of nodes, the parallelism will greatly decrease because of the memory access, as the parallel program will need to access memory at the same time, but the memory can only be accessed one at a time, thus hurting the performance.

7 conclusion

In this assignment, I have gain solid experience in designing and implementing parallel program using Pthread and NUMA library. The parallel program I implemented shows a substantial performance speedup than the original sequential program with a speedup rate over 20 times. The optimized version using NUMA is actually twice as fast as the unoptimized version. Also the observation made within two versions indicates the influence of memory hotspot created by parallel accesses is significant in large number of threads on a shared memory machine. While using NUMA can solve part of the problem, the overall speedup is still far from perfect.

8 Pledge

On my honor as a student, I have neither received nor given help on this assignment.

Signature: ______________

9 Appendix

9.1 halo.c

```
/* initial author: Prof. Andrew Grimshaw (ag8t)
  modified by Jerry Sun (ys7va)
  2017.05.04
  halo.c takes 4 inputs as num_thread, dim_x, dim_y and iterations and it will
  perform a iterations number of heat simulation on a plate of dim_x * dim_y size using ↩
      num_thread threads
  It will print the final runtime for this program.
  This program has two versions with different memory allocation strategies, detailed can ↩
      be found
  within **allocate_cells & **allocate_cells_numa
  */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <pthread.h>
#include <numa.h>
#include "pthread_bar.c"

/* predefined condition for HALO */
#define TOP_BOUNDARY 0
#define BOTTOM_BOUNDARY 100
#define LEFT_BOUNDARY 0
#define RIGHT_BOUNDARY 100
#define INITIAL_VALUE 50
#define HOTSPOT_ROW 4500
#define HOTSPOT_COL 6500
#define HOTSPOT_TEMP 1000

/*method header declaration, detail explanation can be found after the main method*/

/*exit method for possible errors*/
void die(const char *error);

/*helper to create ppm file for visualization*/
void create_snapshot(float **cells, int num_cols, int num_rows, int id);

/*set predefined conditions on a given cell*/
void set(float ***cells, int num_rows, int num_cols);

/*naive memory allocation based on the sequential version provided*/
float **allocate_cells_naive(int num_cols, int num_rows);

/*optimized memory allocation using numa_alloc_onnode*/
float **allocate_cells_numa(int num_cols, int num_rows, int *sep);

/*main computing funtion for each thread*/
static void *thread_compute(void *arg);

/*global variable declaration*/

int real_row, real_col;         /*The real size of the plate we are going to simulate*/
int total_row, total_col;       /*The actual size of the plate we are going to simulate, ↩
    including fixed border*/
int iterations;                 /*Total number of iterations*/
int num_threads;                /*Total number of threads*/
int hotspot_id = -1;           /*The thread_id of a thread to handle hotspot, -1 if not ↩
    applicable */
pthread_barrier_t barrier;     /*global barrier to synchronize the threads for each ↩
    iterations*/

/*the struct to pass into each thread*/
typedef struct{
  int start_row, end_row;      /*The rows this thread handles inclusive for start, ↩
      exclusive for end*/
  int thread_id;               /*The id of this thread*/
  float ***cell;               /*Two cells used for simulation*/
} thread_info;


int main(int argc, char **argv) {
  /*Read the comman line argument into global variable, if not satisfied, exit.*/
  if (argc != 5) die("argument incomplete");
  num_threads = atoi(argv[1]);
  real_row = atoi(argv[2]);
  real_col = atoi(argv[3]);
  iterations = atoi(argv[4]);

  int i, x, y;                  /*initialize some counters used later*/
  float **cell[2];              /*Two main cells to store simulation values*/
  int sep[num_threads + 1];     /*list to set the border (in row major order) of each ↩
      thread*/
  total_row = real_row + 2;     /*Set the actual plate row*/
  total_col = real_col + 2;     /*Set the actual plate col*/

  /*Compute the separation boundary in rows and stored in a list
    We want to evenly distribute the task into threads
    sep[i] specify the row to start for ith thread
    sep[num_threads] marks the end of the last thread */
  int start_row = 1;
  int increment = real_row/num_threads;
  for (i = 0; i < num_threads; i++) {
    sep[i] = start_row;
    start_row = start_row + increment;
  }
  sep[num_threads] = real_row + 1;

  pthread_t threads[num_threads]; /*initialize specifc number of threads asked*/
  thread_info data[num_threads];  /*initialize corresponding number of thread_info to pass↩
       into thread*/
  pthread_attr_t attrs;           /*misc for thread initialization*/
  pthread_attr_init(&attrs);      /*misc for thread initialization*/
  void *status; /*misc for thread join*/
  pthread_barrier_init(&barrier, NULL, num_threads); /*initialize the barrier with ↩
      corresponding number of threads*/

  /*Start the timer*/
  time_t start_time = time(NULL);

  /*Allocate two cell blocks
    In each cell block, the distribution of rows of memory is defined by list sep
    So that different memory partitions are actually on different nodes
    corresponding to later threads computation */
  cell[0] = allocate_cells_numa(total_col, total_row, sep);
  cell[1] = allocate_cells_numa(total_col, total_row, sep);

  /*set the precondition for both cells*/
  set(cell, real_row, real_col);

  /*loop through all the threads_id, initialize the thread info, and then initialize all ↩
      the corresponding threads*/
    for (i = 0; i < num_threads; i++) {
    /* initialize the info that is going to be passed into threads */
    data[i].cell = cell;
    data[i].thread_id= i;
    data[i].start_row = sep[i];
    data[i].end_row = sep[i + 1];

    /*determine the thread that is going to hand hotspot
      if not applicable then hotspot_id = -1, as default */
    if (HOTSPOT_ROW > data[i].start_row && HOTSPOT_ROW < data[i].end_row) hotspot_id = i;

    /*initialize thread within the corresponding info, if fail then quit*/
    int response = pthread_create(&threads[i], &attrs, thread_compute, (void*) &data[i]);
    if (response) die("thread initialization fail\n");
  }

  /*join all the threads initiated*/
  for (i = 0; i < num_threads; i++) {
    int response = pthread_join(threads[i], &status);
    if (response) die("error when joining a thread\n");
  }

  /*stop the timer*/
  time_t end_time = time(NULL);

  /*print the total runtime, and create the snapshot to check the final result*/
  printf("\nExecution time: %d seconds\n", (int) difftime(end_time, start_time));
  create_snapshot(cell[iterations % 2 == 0 ? 0 : 1], real_row, real_col, iterations);
}


/* main computing funtion for each thread */
static void *thread_compute(void *info) {
  int i, j, m;  /*general loop counter*/

  /*cast the passing info*/
  thread_info *data = (thread_info*) info;

  /*This line is only needed when using numa
  memory allocation */

  /*bind the thread onto a particular node
    hermes has 8 nodes, each has 8 cores,
    so we deploy 8 thread onto a single node
    by thread_id:
    0, 8, 16 ... goto node 0
    1, 9, 17 ... goto node 1 ... */
  int resp = numa_run_on_node((data->thread_id) % 8);
  /*Note: This line is only needed when using numa
  memory allocation */


  /*initital the index for two cell boxes
    for each iteration, the value will be computed
    based on the cur_cell, and saved into the next_cell
    and at the end of a single iteration, the cur and next index
    will flip, and continue */
  int cur_cells_index = 0;
  int next_cells_index = 1;
  float ***cell = data -> cell;

  /*main iterations*/
  for (m = 0; m < iterations; m++) {
    /*if it is the hotspot thread, it needs to set the HOTSPOT*/
    if (hotspot_id == data -> thread_id) {
      cell[cur_cells_index][HOTSPOT_ROW][HOTSPOT_COL] = HOTSPOT_TEMP;
    }

    /*each thread only need to work on the corresponding rows they are assigned*/
    for (i = data->start_row; i < data->end_row; i++) {
      for (j = 1; j < total_col - 1; j++) {
        cell[next_cells_index][i][j] = (cell[cur_cells_index][i][j-1]
          + cell[cur_cells_index][i][j+1]
          + cell[cur_cells_index][i-1][j]
          + cell[cur_cells_index][i+1][j]) * 0.25;
      }
    }

    /*indx flip*/
    cur_cells_index = next_cells_index;
    next_cells_index = !cur_cells_index;

    /*barrier for thread synchronization, for each iteration, */
    int s = pthread_barrier_wait(&barrier);
  }
  pthread_exit(NULL);
}

/* prints an error message and exits the program */
void die(const char *error) {
  printf("%s", error);
  exit(1);
}

/* Creates a snapshot of the current state of the cells in PPM format.
 The plate is scaled down so the image is at most 1,000 x 1,000 pixels.
 This function assumes the existence of a boundary layer, which is not
  included in the snapshot (i.e., it assumes that valid array indices
  are [1..num_rows][1..num_cols]).*/
void create_snapshot(float **cells, int num_cols, int num_rows, int id) {
  int scale_x, scale_y;
  scale_x = scale_y = 1;
  // Figure out if we need to scale down the snapshot (to 1,000 x 1,000)
  //  and, if so, how much to scale down
  if (num_cols > 1000) {
    if ((num_cols % 1000) == 0) scale_y = num_cols / 1000;
    else {
      die("Cannot create snapshot for x-dimensions >1,000 that are not multiples of ↩
          1,000!\n");
      return;
    }
  }
  if (num_rows > 1000) {
    if ((num_rows % 1000) == 0) scale_x = num_rows / 1000;
    else {
      printf("Cannot create snapshot for y-dimensions >1,000 that are not multiples of ↩
          1,000!\n");
      return;
    }
  }
    // Open/create the file
  char text[255];
  sprintf(text, "snapshot.%d.ppm", id);
  FILE *out = fopen(text, "w");
  // Make sure the file was created
  if (out == NULL) {
    printf("Error creating snapshot file!\n");
    return;
  }
  // Write header information to file
  // P3 = RGB values in decimal (P6 = RGB values in binary)
  fprintf(out, "P3 %d %d 100\n", num_cols / scale_x, num_rows / scale_y);
  // Precompute the value needed to scale down the cells
  float inverse_cells_per_pixel = 1.0 / ((float) scale_x * scale_y);
  // Write the values of the cells to the file
  int x, y, i, j;
  for (x = 0; x < num_rows; x += scale_x) {
    for (y = 0; y < num_cols; y += scale_y) {
      float sum = 0.0;
      for (i = x; i < x + scale_x; i++) {
        for (j = y; j < y + scale_y; j++) {
          sum += cells[i][j];
        }
      }
      // Write out the average value of the cells we just visited
      int average = (int) (sum * inverse_cells_per_pixel);
      fprintf(out, "%d 0 %d\t", average, 100 - average);
    }
    fwrite("\n", sizeof(char), 1, out);
  }
    // Close the file
  fclose(out);
}

/*set the initial value to both cells, except the hotspot
  hotspot is assigned at thread runtime*/
void set(float ***cells, int num_rows, int num_cols) {
  int x, y, i;
  /* Boundary value */
  for (x = 1; x <= num_cols; x++) cells[0][0][x] = cells[1][0][x] = TOP_BOUNDARY;
  for (x = 1; x <= num_cols; x++) cells[0][num_rows + 1][x] = cells[1][num_rows + 1][x] = ↩
      BOTTOM_BOUNDARY;
  for (y = 1; y <= num_rows; y++) cells[0][y][0] = cells[1][y][0] = LEFT_BOUNDARY;
  for (y = 1; y <= num_rows; y++) cells[0][y][num_cols + 1] = cells[1][y][num_cols + 1] = ↩
      RIGHT_BOUNDARY;

  /* Internal value */
  for (x = 1; x <= num_rows; x++)
        for (y = 1; y <= num_cols; y++)
            cells[0][x][y] = cells[1][x][y] = INITIAL_VALUE;
}

/* Allocates and returns a pointer to a 2D array of floats, original version*/
float **allocate_cells_naive(int num_cols, int num_rows) {
    float **array = (float **) malloc(num_rows * sizeof(float *));
    if (array == NULL) die("Error allocating array!\n");
    array[0] = (float *) malloc(num_rows * num_cols * sizeof(float));
    if (array[0] == NULL) die("Error allocating array!\n");
    int i;
    for (i = 1; i < num_rows; i++) {
        array[i] = array[0] + (i * num_cols);
    }
    return array;
}

/* Allocates and returns a pointer to a 2D array of floats using numa*/
float **allocate_cells_numa(int num_cols, int num_rows, int *sep) {
  /*some general counter*/
  int counter = 0;
  int i;

  /*number of floats need to be allocated*/
  int to_alloc = 0;

  /*node the memory is going to be allocated, all nodes are allocated on node i % 8 */
  int node = 0;
  /*First allocate all row pointers */
  float **array = (float **) malloc(num_rows * sizeof(float *));
  if (array == NULL) die("Error allocating array!\n");

  /*allocate the first row*/
  int to_alloc = (sep[1] - sep[0] + 1) * num_cols;
  array[0] = (float *) numa_alloc_onnode(to_alloc * sizeof(float), node);
  if (array[0] == NULL) die("Error allocating array!\n");
  for (counter = 1; counter < sep[1]; counter++) {
    array[counter] = array[0] + (counter * num_cols);
  }

  /*allocate the second to the second last row*/
  for (i = 1; i < num_threads - 1; i++) {
    int increment = sep[i + 1] - sep[i];
    to_alloc = increment * num_cols;
    node = i % 8;
    array[sep[i]] = (float *) numa_alloc_onnode(to_alloc * sizeof(float), node);
    if (array[sep[i]] == NULL) die("Error allocating array!\n");
    /*assign value pointers to row pointers*/
    for (counter = sep[i] + 1; counter < sep[i + 1]; counter++) {
      array[counter] = array[sep[i]] + (counter - sep[i]) * num_cols;
    }
  }

  /*allocate the last row*/
  to_alloc = (sep[num_threads] - sep[num_threads - 1] + 1) * num_cols;
  node = (num_threads - 1) % 8
  array[sep[num_threads - 1]] = (float *) numa_alloc_onnode(to_alloc * sizeof(float), node↩
      );
  /*assign value pointers to row pointers*/
  for (counter = sep[num_threads - 1] + 1; counter <= sep[num_threads]; counter++) {
    array[counter] = array[sep[num_threads - 1]] + (counter - sep[num_threads - 1]) * ↩
        num_cols;
  }

  return array;
}
```

9.2 halo.sh

```
#!/bin/bash
#SBATCH --nodelist=hermes1
#SBATCH --exclusive

# Wrote by Yijie Sun (ys7va) 2017.05.04
# the script is used to run halo simulation on Hermes
# the four inputs the program takes is num_thread, dim_x, dim_y and iterations
# details can be found in halo.

gcc -O3 -o halo halo.c -lpthread -lnuma
./halo 64 10000 10000 10000
```