CS 4444 Parallel Computing Spring 2017

Homework 4: Shared Memory Heated Plate

Eamon Collins

When run, my code successfully performs a simulation of a heated plate to the specifications laid out in submit.slurm. The speedup achieved by using 64 pthreads as compared to just 1 was a factor of 5.27. When compared to a run using non-locally allocated memory, both on 64 threads, the locally allocated version achieved a relative speedup of 2.02.

Approach:

This problem was much easier using pthreads than using MPI, as the shared memory aspect of it allows effortless communication between threads by simply accessing the same variables.

To start, I set up all the pthreads. In my version, I had the thread performing the main method create as many other threads as was needed minus 1, and then it called the *halo* method itself as well. The purpose of this was an attempt to have exactly as many threads as cores, so that when using 64 threads for the problem, one of the cores didn't have a worker thread as well as the main thread switching on for some timeslices, waiting for it's call to pthread_join to be satisfied. I later learned that pthread_join suspends execution until the thread it's waiting on has terminated, but I had already implemented this feature and kind of like the tidiness that every thread involved does a portion of the work. To standardize the threads, I set tids[0] = pthread_self(), and separately set the args for thread 0.

While setting up the threads, I set the affinity for each thread to a certain core. I needed each thread to be "pinned" to a certain core so that they didn't successfully locally allocate their memory and then move cores, making their memory accesses suddenly non-local. All threads except thread 0 are set while setting them up, with thread 0 I had problems pinning it to a core until I used sched_affinity instead of the pthread function, and used it from within the *halo* function itself. After pinning thread 0, and since I set the other ones while they are actually running rather than in their attributes before they are created, I have a while loop that waits and checks that all threads are on the core number corresponding to their pthread id. A barrier in this loop makes sure that each and every thread is on the core they will stay on for the rest of the execution before any memory is allocated.

My approach to memory allocation was to create the global cell arrays outside of the *halo* function, each consisting of an array of float pointers that will serve as the pointers to each row, but I left these pointers uninitialized. Inside the *halo* function, each thread initializes the pointers to the rows it is responsible for by using calls to the numa function numa_alloc_local(). This ensures that the memory is allocated on the memory bank closest to the core that thread is operating on. I figured that even though every time an array access is made, the core must retrieve two pointers before getting to the locally allocated memory, (the pointer to the correct timestep of the array and then to the actual row,) and neither of those are guaranteed to be local, the cache size guarantees that these will be easily accessible at all times. The L1 data cache for each node is 64K, and the array of row pointers for each timestep will be 80K, for a total of 160K, but since each node only needs to access ¼ of that

to find all the rows for all the cores on that node, that will be 40K per node, and then we can actually cut that down to 20K because they will all only be reading one of the timesteps at a time. This is able to fit in the cache with space left over for the actual row data.

To handle the row accesses easily, I created a row offset array with the start and end indices of the responsibilities of each thread according to id. The problem was chunked similarly to the MPI implementation, 1 dimensionally by groups of rows. To deal with the fact that 10,000 rows is not divisible by many of the thread counts we had to test, a number of threads equal to 10,000 % thread_count were responsible for one extra row. Later, to iterate over the rows one process was responsible for, all they had to do was loop rowStart <= y < rowEnd. With this, the pattern of access was extremely easy to implement as no explicit communication was needed, the update loops were barely changed from the sequential implementation.

Setting the hotspot was also easier than the MPI implementation, as there were no ghost cells to confuse the process. I simply set the cell to its temp with the processor that had that row in its responsibility.

Again, like the MPI implementation, barriers were needed before the iterations started to make sure everything was properly allocated and initialized, and then another barrier was needed at the end of each iteration to make sure every thread was working on the same iteration at all times, in order to maintain deterministic behavior.

For the non-local test, I produced another code file called pthread_nonlocal, in which everything is the same except I don't pin the threads to cores and all memory allocation for the cells matrix is done outside the *halo* function with the c function malloc() instead of numa_alloc_local(). Technically malloc()'s default behavior will tend to allocate the memory on the local node, so this ensures that for most threads at least, each row is likely on a non-local memory module.

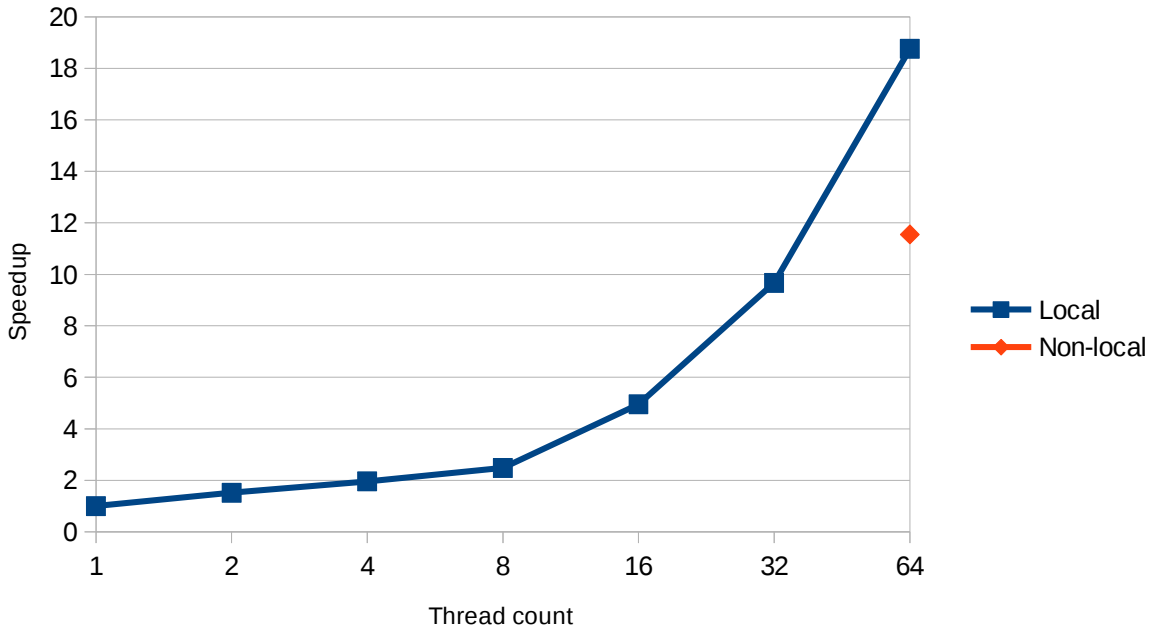
Results:

All results reported are on a 10,000x10,000 matrix for 10,000 iterations with the -O3 flag enabled.

Speedup is relative to a run with 1 thread with local memory access enabled.

Thread Count	**Execution Time (sec)**	**Speedup**
1	2195	1
2	1442	1.52
4	1119	1.96
8	885	2.48
16	443	4.95
32	227	9.67
64	117	18.76
64 **(non-local)**	236	9.3

Below is a graph of speedup vs. thread count



Analysis:

My results show a much greater improvement per thread at the higher thread counts, while at the lower thread counts there is improvement but much less than what one might expect. In addition, it seems that the strategy of local allocation produced a much faster implementation than did the unpinned, non-locally allocated version.

I believe the struggling of the lower thread counts to improve the execution time is a result of cache overloading. Essentially, to make things easy, I pinned all threads to the core number corresponding to their thread id. This means that at thread counts of 16 and below, all threads are running on the same numa-node of the overall Hermes chip. I would expect each thread to process at the same rate as at any other time and therefore issue memory requests at the same intervals, but now each row pointer array is too big to fit into the L1 cache by itself, meaning many of the threads must go to other memory nodes to find the pointers to their rows. In addition the actual row data will eject other threads row pointers from the array when accessed, all in all forming a very poor cache utilization pattern. A solution to this would have been to make sure the threads are equally spread over the 4 numa-nodes at all thread-counts, so that the power of having 4 separate cache systems could be fully realized.

The cache effects make it significantly worse

The non-local allocation run proves how valuable properly managing NUMA nodes can be: when no attention is paid to where the data is allocated and the threads are allowed to switch around on the cpus according to normal scheduling, the speed up achieved was only 9.3 compared to the fully optimized 18.76. That's over twice as fast, just from pinning cores and allocating locally.

Conclusion:

Because memory bandwidth is often the limiting factor of high performance computing, paying close attention to how memory is managed is crucial to attaining performance, especially on shared memory machines where it is possible to naively allocate memory to non-local nodes without realizing anything is wrong. Also, whenever memory is an issue caches must also be explicitly considered, as they are key in making sure you aren't running back to the RAM every time you need something, which will be slow even if that RAM is on a module local to the current node.

PLEDGE

On my honor as a student, I have neither given nor received unauthorized aid on this assignment.
-Eamon Collins

RELEVANT FILES:

pthread.c – main code file, simulates the heated plate with as many pthreads as specified in submit.slurm. Uses local memory allocation.
pthread_nonlocal.c – alternative to pthread.c, the same except does not pin threads to cores and allocates all the memory naively in the main method. (I don't include this below because it is mostly identical to pthread.c)
Submit.slurm – submission script, also where you specify iteration count, problem size, and thread count.
Makefile – my makefile, will make the pthread binary from pthread.c
snapshot.* - the tc specifies what threadcount this was produced by, the number after that is the iteration count. Both tc1 and tc64 are the same as expected.
Output – directory containing the outputs from my benchmarking runs.

HOW TO RUN MY CODE:

log onto the cs compute nodes
make
set parameters you want in submit.slurm
sbatch submit.slurm
check output file created.

PTHREAD.C

```
// This program simulates the flow of heat through a two-dimensional plate.
// The number of grid cells used to model the plate as well as the number of
//  iterations to simulate can be specified on the command-line as follows:
//  ./heated_plate_sequential <columns> <rows> <iterations>
// For example, to execute with a 500 x 500 grid for 250 iterations, use:
//  ./heated_plate_sequential 500 500 250

#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <numa.h>
#include <utmpx.h>
#include <sched.h>

// Define the immutable boundary conditions and the inital cell value
#define TOP_BOUNDARY_VALUE 0.0
#define BOTTOM_BOUNDARY_VALUE 100.0
#define LEFT_BOUNDARY_VALUE 0.0
#define RIGHT_BOUNDARY_VALUE 100.0
#define INITIAL_CELL_VALUE 50.0
#define hotSpotRow 4500
#define hotSptCol 6500
#define hotSpotTemp 1000
#define ncols 10000
#define nrows 10000


// Function prototypes
void print_cells(float **cells, int n_x, int n_y);
void initialize_cells(float **cells, int n_x, int n_y);
void create_snapshot(float **cells, int n_x, int n_y, int id, int tc);
float **allocate_cells(int n_x, int n_y);
void die(const char *error);
void *halo(void* a);

typedef struct arg {
        int tnum;
        float ***cells;
        int iterations;
        int thread_count;
        int iters_per_snap;
} t_arg;

pthread_barrier_t barr;

int main(int argc, char **argv) {
```

```
	// Record the start time of the program
	time_t start_time = time(NULL);

	// Extract the input parameters from the command line arguments
	// Number of columns in the grid (default = 1,000)
	// int num_cols = (argc > 1) ? atoi(argv[1]) : 1000;
	// // Number of rows in the grid (default = 1,000)
	// int num_rows = (argc > 2) ? atoi(argv[2]) : 1000;
	// // Number of iterations to simulate (default = 100)
	// int iterations = (argc > 3) ? atoi(argv[3]) : 100;
	//number of chunks in x dimension
	int thread_count = (argc > 1) ? atoi(argv[1]) : 1;

	int iters_per_snap = (argc > 2) ? atoi(argv[2]) : 5000;
	// Number of iterations to simulate (default = 100)
	int iterations = (argc > 3) ? atoi(argv[3]) : 1000;

	float **cells[2];
	cells[0] = (float **) malloc((nrows+2)*sizeof(float*));
	cells[1] = (float **) malloc((nrows+2)*sizeof(float*));

	//set up threads, attrs and the barrier we will need
	pthread_t tids[thread_count];
	t_arg args[thread_count];
	pthread_attr_t attr;
	pthread_attr_init(&attr);
	pthread_barrier_init(&barr, NULL, thread_count);
	int i;
	cpu_set_t cpuset;
	//makes the first pthread in tids the main process so that it can be included in the
overall count
	//I made the thread executing the main method also call halo because I thought
pthread_join was implemented
	//by basically a loop that continually tested for completion of the other threads, and
therefore thought that when creating
	//64 threads on 64 cores, the creating thread would be a 65th thread that would take up
time-slices on one of the processors
	//delaying the thread computing on that core. I now know it suspends execution, but
live and learn. This way works too.
	tids[0] = pthread_self();
	for(i=1; i < thread_count; i++){
		args[i].tnum = i;
		args[i].cells = cells;
		args[i].iterations = iterations;
		args[i].thread_count = thread_count;
		args[i].iters_per_snap = iters_per_snap;
		//create the threads.
		pthread_create(&tids[i], &attr, halo, &args[i]);
```

```
                //set the affinity of the thread to the core numbered the same as the thread.
                CPU_ZERO(&cpuset);
                CPU_SET(i, &cpuset);
                pthread_setaffinity_np(tids[i], sizeof(cpu_set_t), &cpuset);
        }
        //setting up args for thread 0 as well
        args[0].tnum = 0;
        args[0].cells = cells;
        args[0].iterations = iterations;
        args[0].thread_count = thread_count;
        args[0].iters_per_snap = iters_per_snap;

        printf("threads: %d\n",thread_count);fflush(stdout);
        //the mmain funciton calls halo with the correct args.
        halo(&args[0]);
        //start at one because we know proc 0 has returned from halo if it has reached this
point. Join the other threads.
        for(i=1; i < thread_count; i++){
                pthread_join(tids[i], NULL);
        }

        // Compute and output the execution time
        time_t end_time = time(NULL);
        printf("\nExecution time: %d seconds\n", (int) difftime(end_time, start_time));
        fflush(stdout);

        free(cells[0]); free(cells[1]);

        //exit pthreads
        pthread_exit(NULL);

}

void *halo(void *a){
        t_arg *arg = (t_arg*) a;
        int id = arg->tnum;
        int iterations = arg->iterations;
        float ***cells = arg->cells;
        int thread_count = arg->thread_count;
        int i, x, y;
        int num_cols = ncols;
        int num_rows = nrows;
        //set the affinity for the main process (thread 0)
        if(id == 0){
                cpu_set_t cpuset;
                CPU_ZERO(&cpuset);
                CPU_SET(0, &cpuset);
                sched_setaffinity(0, sizeof(cpu_set_t), &cpuset);
        }
```

```
	//wait until all threads are on the core they will be pinned to to proceed with allocation of data
	//need to do this because the affinity is set after the thread starts, have to wait for them to switch to the right core
	while(1){
		if(sched_getcpu() == id){
			pthread_barrier_wait(&barr);
			break;
		}
	}

	if (id == 0){
		// Output the simulation parameters
		printf("Grid: %dx%d, Iterations: %d\n", num_cols, num_rows, iterations);
		fflush(stdout);
	}
	int rows_each = num_rows / thread_count;
	int offsets[thread_count];
	offsets[0] = 1; //1 to account for boundary along the top
	for(i = 1; i <= thread_count; i++){
		//creates a list of the row offsets from 0 of the start point of each of the threads responsibility.
		//last element is the boundary row at the bottom, so all access loops can go offsets[id] <= i < offsets[id+1]
		//ternary operator tests whether to add one to account for a matrix size not perfectly divisible by the thread count (threads at the beginning have one more row than threads at the end)
		offsets[i] = offsets[i-1] + rows_each + (((i-1) < (num_rows % thread_count)) ? 1 : 0);
	}
	//store these in variables so I don't spend time in array lookups in the loop bodies later
	int rowStart = offsets[id];
	int rowEnd = offsets[id+1];

	//allocate each row the processor is responsible for locally to it
	//since this takes place after we make sure every process is on the core it will be pinned
	//to for the whole process, we know that memory allocated locally here will stay local to that process.
	if(id == 0) {
		cells[0][0] = (float *)numa_alloc_local((num_cols+2) * sizeof(float));
		cells[1][0] = (float *)numa_alloc_local((num_cols+2) * sizeof(float));
	}
	if (id == thread_count - 1){
		cells[0][num_rows + 1] = (float *)numa_alloc_local((num_cols+2) * sizeof(float));
		cells[1][num_rows + 1] = (float *)numa_alloc_local((num_cols+2) * sizeof(float));
	}
	for(i = rowStart; i < rowEnd; i++){
		cells[0][i] = (float *)numa_alloc_local((num_cols+2) * sizeof(float));
```

```
                cells[1][i] = (float *)numa_alloc_local((num_cols+2)* sizeof(float));
        }
        //initialize the non-boundary cells
        for(y = rowStart; y < rowEnd; y++){
                for(x=1; x < num_cols + 1; x++){
                        cells[0][y][x] = INITIAL_CELL_VALUE;
                }
        }

        //initialize boundary conditions
        for(y = rowStart; y < rowEnd; y++){
                cells[0][y][0] = cells[1][y][0] = LEFT_BOUNDARY_VALUE;
                cells[0][y][num_cols+1] = cells[1][y][num_cols+1] =
RIGHT_BOUNDARY_VALUE;
        }
        if(id==0){
                for(x=0; x < num_cols + 2; x++) cells[0][0][x] = cells[1][0][x] =
TOP_BOUNDARY_VALUE;
        }
        if(id == thread_count -1){
                for(x=0; x < num_cols + 2; x++) cells[0][num_rows + 1][x] = cells[1][num_rows +
1][x] = BOTTOM_BOUNDARY_VALUE;
        }

        //barrier so that no threads start until the whole array is allocated
        pthread_barrier_wait(&barr);

        int cur_cells_index = 0, next_cells_index = 1;
        for (i = 0; i < iterations; i++){
                for(y = rowStart; y < rowEnd; y++){
                        for (x = 1; x < num_cols + 1; x++){
                                // The new value of this cell is the average of the old values of this
cell's four neighbors
                                cells[next_cells_index][y][x] = (cells[cur_cells_index][y][x - 1]  +
                                                                 cells[cur_cells_index][y][x
+ 1]  +
                                                                 cells[cur_cells_index][y - 1]
[x]  +
                                                                 cells[cur_cells_index][y +
1][x]) * 0.25;
                        }
                }

                // Print the current progress every time you iterate
                //took this out to speed things up but mostly cause it's annoying when I'm not
testing
//              if(id==0){
//                      printf("Iteration: %d / %d\n", i + 1, iterations);
//                      fflush(stdout);
```

```
//                }
                  //switch the arrays
                  cur_cells_index = next_cells_index;
                  next_cells_index = !cur_cells_index;

                  //sets the hotspot temp with the thread that is responsible for it, and therefore
local to it
                  if(rowStart <= hotSpotRow && rowEnd > hotSpotRow && num_cols >=
hotSptCol) {
                        cells[cur_cells_index][hotSpotRow][hotSptCol] = hotSpotTemp;
                  }

                  //only thread 0 calls create snapshot, then that thread accesses the whole array,
including non-local portions, and compiles the snap
                  //not that costly cause in benchmarking runs I only do it once in the whole
program
                  if ((i+1) % arg->iters_per_snap == 0 && id == 0){
                        int final_cells = (iterations % 2 == 0) ? 0 : 1;
                        create_snapshot(cells[final_cells], num_cols, num_rows, i+1,
thread_count);
                  }
                  //wait until all threads have finished the iteration to proceed to the next one
                  pthread_barrier_wait(&barr);
            }


            //free each row by the processor that is responsible/local for it
            if(id == 0) {
                  numa_free(cells[0][0],(num_cols+2) * sizeof(float));
                  numa_free(cells[1][0],(num_cols+2) * sizeof(float));
            }
            if (id == thread_count - 1){
                  numa_free(cells[0][num_rows + 1],(num_cols+2) * sizeof(float));
                  numa_free(cells[1][num_rows + 1],(num_cols+2) * sizeof(float));
            }
            for(i = rowStart; i < rowEnd; i++){
                  numa_free(cells[0][i],(num_cols+2) * sizeof(float));
                  numa_free(cells[1][i],(num_cols+2) * sizeof(float));
            }
}


// Allocates and returns a pointer to a 2D array of floats
float **allocate_cells(int cols, int rows) {
            float **array = (float **) malloc(rows * sizeof(float *));
            if (array == NULL) die("Error allocating array!\n");

            array[0] = (float *) malloc(rows * cols * sizeof(float));
            if (array[0] == NULL) die("Error allocating array!\n");
```

```
	int i;
	for (i = 1; i < rows; i++) {
		array[i] = array[0] + (i * cols);
	}

	return array;
}


// Sets all of the specified cells to their initial value.
// Assumes the existence of a one-cell thick boundary layer.
void initialize_cells(float **cells, int num_cols, int num_rows) {
	int x, y;
	for (y = 1; y <= num_rows; y++) {
		for (x = 1; x <= num_cols; x++) {
			cells[y][x] = INITIAL_CELL_VALUE;
		}
	}
}


// 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 cols, int rows, int id, int thread_count) {
	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 (cols > 1000) {
		if ((cols % 1000) == 0) scale_x = cols / 1000;
		else {
			die("Cannot create snapshot for x-dimensions >1,000 that are not
multiples of 1,000!\n");
			return;
		}
	}
	if (rows > 1000) {
		if ((rows % 1000) == 0) scale_y = 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.tc%d.%d.ppm",thread_count, 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", cols / scale_x, 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 (y = 1; y <= rows; y += scale_y) {
		for (x = 1; x <= cols; x += scale_x) {
			float sum = 0.0;
			for (j = y; j < y + scale_y; j++) {
				for (i = x; i < x + scale_x; i++) {
					sum += cells[j][i];
				}
			}
			// Write out the average value of the cells we just visited
			int average = (int) (sum * inverse_cells_per_pixel);
			if (average > 100)
				average = 100;

			fprintf(out, "%d 0 %d\t", average, 100 - average);
		}
		fwrite("\n", sizeof(char), 1, out);
	}

	// Close the file
	fclose(out);
}


// Prints the specified error message and then exits
void die(const char *error) {
	printf("%s", error);
	exit(1);
}
```

SUBMIT.SLURM

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=64
#SBATCH --output="output64"
#SBATCH --nodelist=hermes1
#SBATCH --mem_bind=local
#SBATCH --exclusive

#SBATCH --time=02:00:00
#SBATCH --account=parallelcomputing

srun pthread 64 10000 10000
```

MAKEFILE

```
CC=gcc
CFLAGS=-O3
OPTIONS=-lnuma -lpthread

default: pthread

pthread: pthread.c
	$(CC) $(CFLAGS) -o pthread pthread.c $(OPTIONS)
```