ECE/CSC 506: Architecture of Parallel Computers

Team:

Ranganatha Muralidhar Kulkarni (rkulkar3)

Ajay Rao (abrao2)

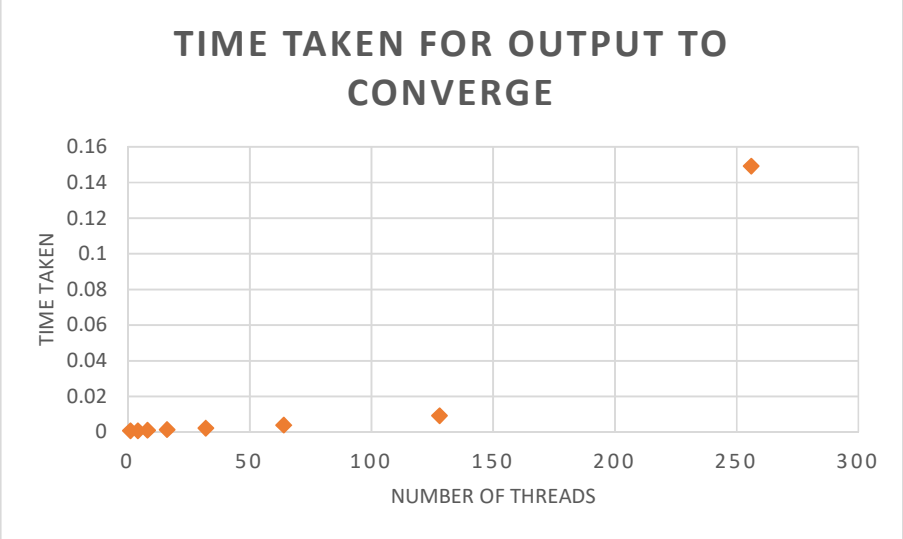
Program 2: Simulating Ocean Currents (Serial, OpenMP, and CUDA versions)

Ocean current simulation using red black ordering was implemented in Serial, OpenMP and CUDA versions. Following are the results and analysis of the simulation:

OpenMP Version:

For OpenMP there were two versions of code that was tried out. In the first one we had both the 'for' loops used for iterating over the red and black elements run in parallel. Thus making each element run on an individual thread. The instruction to add the individual diff of each element to a global variable was placed under the critical section. Thus only one thread at a time could update the diff variable. The results of this simulation for 16x16 input grid is as shown below:

No. of threads	Time taken for output to converge
1	0.00015
4	0.000656
8	0.001051
16	0.001853
32	0.002718
64	0.004233
128	0.007606
256	0.015583

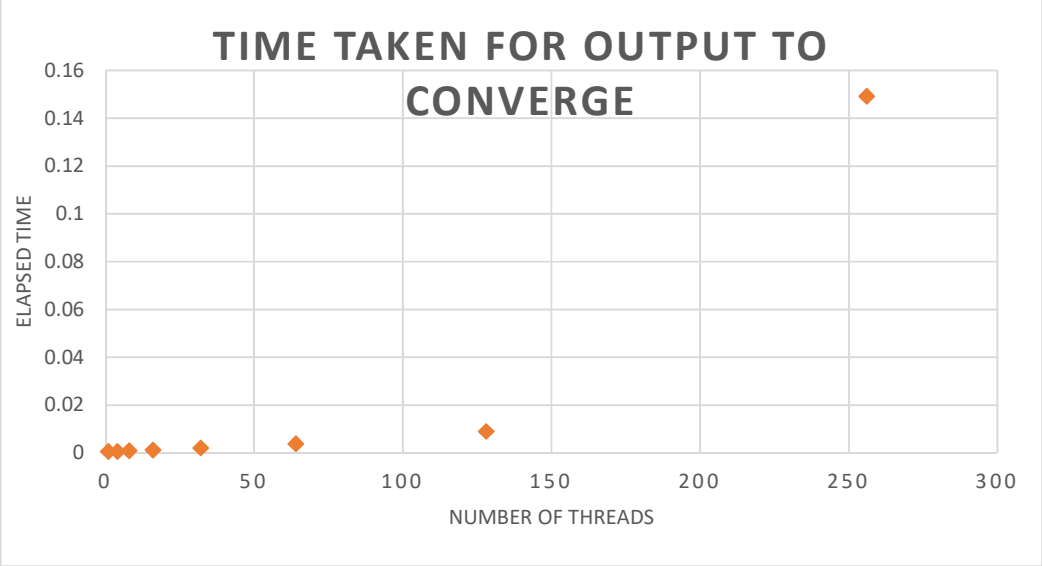


In the second version we just had the outer 'for' loop running in parallel. So, we had each row being a thread and each thread was computing the red or black elements in that entire row depending on whether it was the loop for executing red elements or black. The critical section remained the same.

Following results show the tables and graphs for variation in number of threads for each grid size starting from 16x16 to 1024x1024.

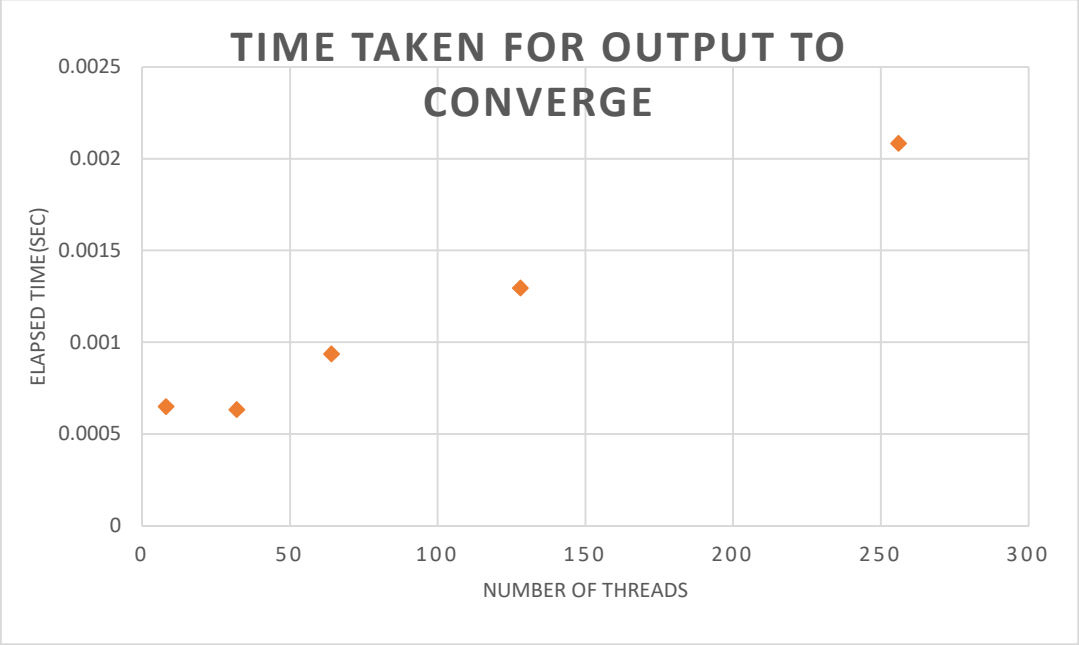
1. Grid Size: 16x16

No. of threads	Time taken for output to converge
1	0.00065
4	0.000633
8	0.000937
16	0.001295
32	0.002083
64	0.003826
128	0.009087
256	0.14916



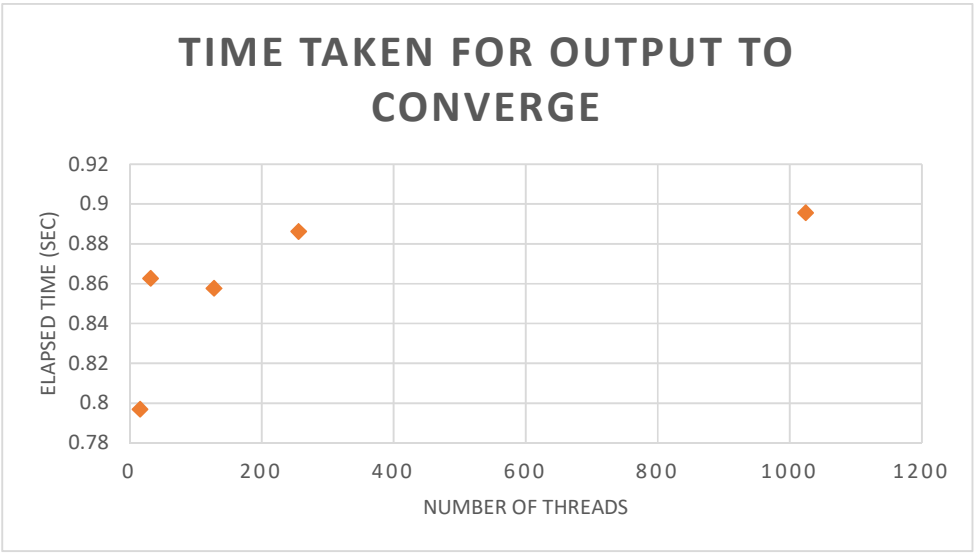
2. Grid Size: 128x128

No. of threads	Time taken for output to converge
8	0.00065
32	0.000633
64	0.000937
128	0.001295
256	0.002083



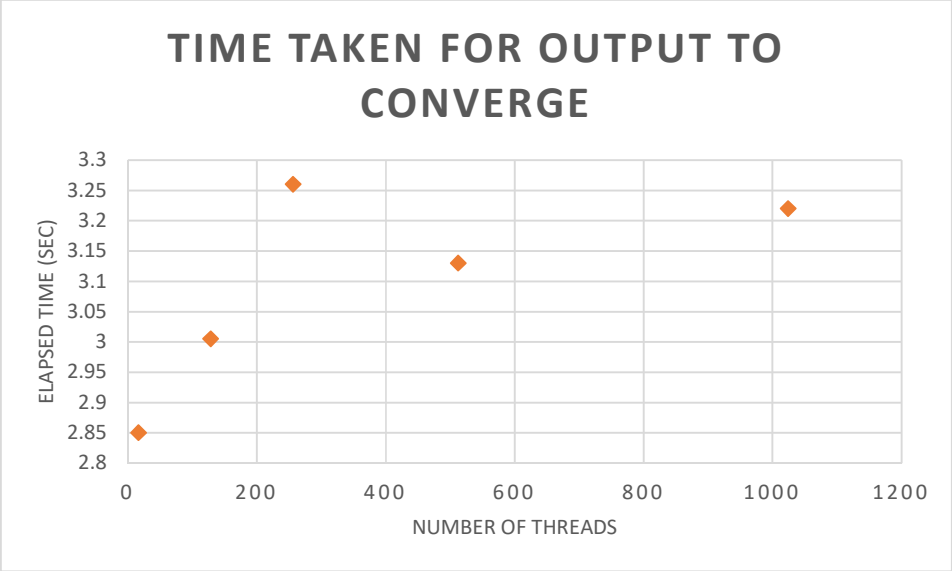
3. Grid Size: 256x256

No. of threads	Time taken for output to converge
16	0.796984
32	0.862754
128	0.857656
256	0.886344
1024	0.895671



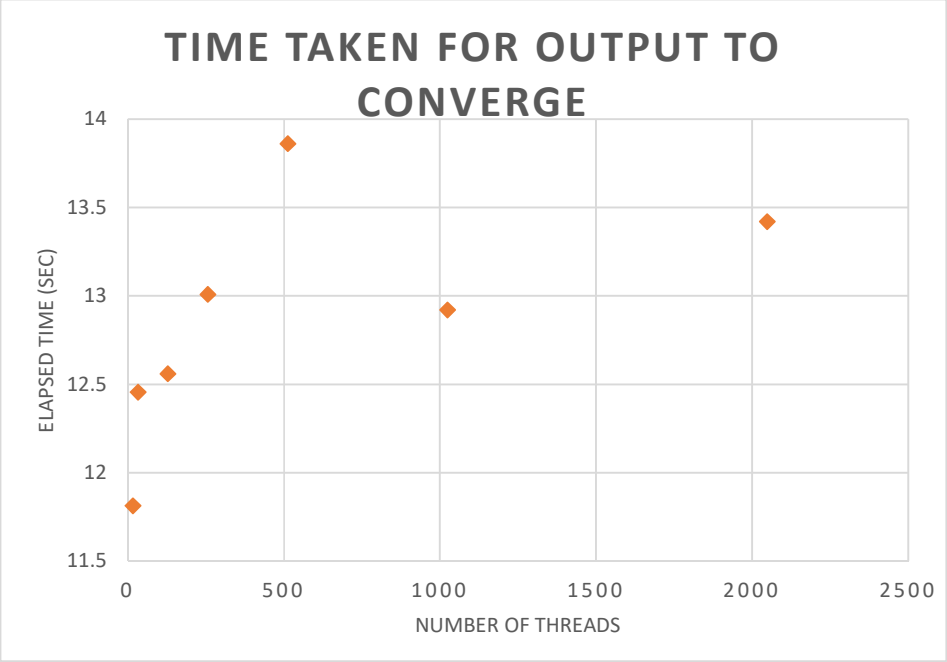
4. Grid Size: 512x512

No. of threads	Time taken for output to converge
16	2.85
128	3.005
256	3.26
512	3.13
1024	3.22



5. Grid Size: 1024x1024

No. of threads	Time taken for output to converge
16	11.812
32	12.456
128	12.56
256	13.009
512	13.86
1024	12.92
2048	13.42



Observations for OpenMP versions:

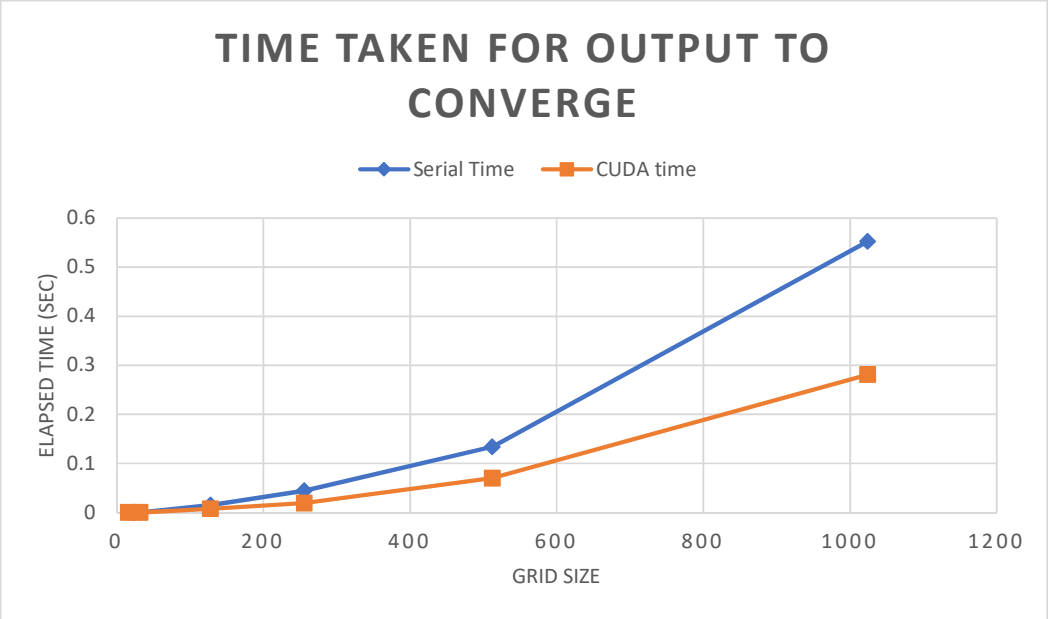
a. We observe that the time taken for convergence in the version of the code that had both 'for' loops in parallel is slightly more than that of the execution time when only the outer 'for' loop is parallelized.
b. The reason for such an observation is the overhead caused by critical section. When each thread had to go through the critical section all other threads had to wait for the thread that is being executed.
c. Since each thread calculated individual element of the matrix in this version, the overhead was very high.
d. However, when only the outer 'for' loop was parallelized the number of elements computed by each thread increased to the number of red/black elements in one row.
e. Even though the critical section remained the same, the number of elements that were computed before a thread could reach the critical section had increased. Thus, we could observe that the execution time for one 'for all' loop was lesser than two 'for all' loops.
f. This execution time could have been further reduced by increasing the number of rows computed by each thread i.e. a block partitioning of the matrix.
g. Another interesting observation is that the execution time not decreasing with increase in number of threads. The reason for this could be the hardware architecture of Henry cluster used in the NCSU HPC.
h. Each node in the cluster has up to 24 processors. Thus, when all the computation is performed on one node to ease the memory sharing, we can get a maximum of 24 threads at a point in time.
i. Thus, even though the number of threads spawned were increasing, the hardware could only support max of 24 threads simultaneously. This added OS overhead in terms of thread creation, thread scheduling and thread handling.
j. When the grid size was lesser than the number of threads spawned, the extra spawned threads were not turned off. Hence the execution time increased.
k. When there were grid elements to be computed for all threads spawned , the program was similar to running a group of 24 threads serially (i.e 0-23 threads run in parallel, after this completed 24-47 were run in parallel, after it completed 48-71 were run in parallel and so on).

Thus the overheads involving thread handling kept adding up as the number of threads increased with grid size, and the same was observed.

CUDA Version:

Following tables and graphs show the variation in elapsed time with respect to change in grid size for serial and CUDA code.

Grid Size	Serial Time	CUDA time
16x16	0.000026	0.000202
32x32	0.000111	0.000328
128x128	0.014814	0.007577
256x256	0.044413	0.019538
512x512	0.13414	0.07013
1024x1024	0.552349	0.28079



Explanation:

a. Sequence of commands for launching CUDA kernel as against serial code:
   - Allocate memory and initialize the array elements (NxN) in the host (CPU) memory – Common for both CUDA and Serial version.
   - Allocate and initialize array named diff1 to hold the difference of each element of the NxN matrix in the host (CPU) memory – Step performed only in CUDA version.

     [float *diff1 = NULL;
     diff1 = (float*) malloc (size);]

- Allocate memory in device (GPU) for the NxN input matrix and NxN difference matrix – Step performed only in CUDA version.

  [float *d_grid = NULL;
  cudaMalloc((void**)&d_grid, size);
  float *d_diff1 = NULL;
  cudaMalloc((void**))&d_diff, size);]

- Initialize the GPU kernel to have 256 threads per block and Ceil(NxN/256) number of blocks in the grid for parallel execution - Step performed only in CUDA version.

  [int threads_per_block = 256;
  int blocks_per_grid = (int)((row*col + threads_per_block - 1) / threads_per_block);]

- Copy the input array and difference array from the host (CPU) to the device (GPU) – step performed only in CUDA version.

  [cudaMemcpy(d_grid,array,size,cudaMemcpyHostToDevice);
  cudaMemcpy(diff1,d_diff1,size,cudaMemcpyHostToDevice);]

- Launch CUDA kernel with the set number of threads per block and number of blocks for computing RED elements first and then BLACK elements – Step done only in CUDA version. In the serial version the function red_black_ordering(…) is called twice to traverse the matrix for RED/BLACK elements serially and perform the computation.

  [red_black_ordering <<< blocks_per_grid, threads_per_block >>> (d_grid, row, RED, d_diff1);
  red_black_ordering <<< blocks_per_grid, threads_per_block >>> (d_grid, row, BLACK, d_diff1);]

- In CUDA version each element is computed by a thread, and hence there are as many threads as there are elements in the input matrix. Each thread updates the d_diff1[] array in the device (CUDA) memory for its corresponding element. Computation of each element of the input array is done in the device (GPU). Whereas in the serial version all the computations are done in the host(CPU).
- After all threads have finished their execution the d_diff1[] array is copied from the device (GPU) to host (CPU) memory – step is performed only in CUDA version. In serial version there is no need of an array for the diff calculation. The same memory location in the host is updated after computation of each element, as the computation is serial, one element at a time.

  [cudaMemcpy(diff1,d_diff1,size,cudaMemcpyDeviceToHost);]

- Convergence is calculated after all the elements are processed for one sweep in both CUDA and serial version. In both the cases this calculation is done in the host(CPU).
- CUDA kernel is launched until the convergence is achieved in CUDA version as compared to calling just the red_black_ordering(…) function in serial version.
- After convergence is achieved the device memories are freed – step performed only in CUDA version.
  [cudaFree(d_diff1);
  cudaFree(d_grid);]

b. It can be inferred from the data obtained that the CUDA version of the red black ordering simulation runs approximately twice as fast compared to serial version of the same algorithm on the same input grid size. The reason for this observation is that each element is computed in parallel in the CUDA version. The time taken for computation of each element serially in multiple sweeps is far greater than the time taken for the overhead in launching CUDA kernel and data exchange. Thus even though we use the reduction operation to get the final value of diff in the CUDA version, we see that this operation has minimum effect on the overall execution time. Speedup of approximately 2 is observed in this case.