MPI CAPSTONE LAB

Emily Wasylenko

COMP 322 – Parallel Computing

12/11/18

**Purpose:** The goal of this lab is to find out how OMP affects programs in a Beowulf cluster. To determine this, runtime, speed-up, and efficiency of three different programs with three different layouts are compared.

**Set-Up:**

First, we set up a Beowulf cluster—four nodes, a “master” and three “slaves”, all connected through an Ethernet cable. Using a Linux environment, we built three CPU intensive programs (Trapezoidal, Jacobi, and Texas Holdem Poker) on the cluster.

Three serial trials were run for each program, which meant the master node did all the work. These runtimes served as control cases. After this, each program was modified to run over MPI (the message-passing interface). Finally, each program was modified to work with OMP and MPI *(****Fig. 1****)*.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | TRAPEZOIDAL | | | TEXAS HOLDEM | | | JACOBI | | |
| SERIAL | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 |
| MPI | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 |
| MPI WITH OMP | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 |

**Fig. 1: Format of trial runs**

**Trapezoidal**

The Trapezoidal program uses the rule for trapezoidal integration to approximately solve a given integral. In this lab, the given equation is , with the solution of -0.022740. This equation is evaluated with 64 to 67108864 subintervals, with the number increasing by powers of two each time.

To ensure accuracy across all schedules, the program prints the number of subintervals (we used 10 million in each case), the approximate answer, and the actual solution. The results of the approximations and actual solutions varied a bit, but not by a dramatic amount. These results are shown in Fig. 2.

The average runtimes are compared and displayed in Fig. 3, and speed-up and efficiency of all trials are shown in Fig. 4a-b. All trials were shown, rather than the average, because there were so few of them. Speed-up throughout this lab is (TimeParallel/TimeSerial), and efficiency is TimeParallel/#Threads.

Trapezoidal’s parallel section used the schedule (guided, 1), for four threads.

TRAP SERIAL

T1: 0.481 s sum = -0.022741 integral = -0.022740

T2: 0.489 s sum = -0.022741 integral = -0.022740

T3: 0.485 s sum = -0.022741 integral = -0.022740

Output for 10000000 (10 million) segments

TRAP MPI

T1: 0.227 s sum = -0.022742 integral = -0.022740

T2: 0.228 s sum = -0.022742 integral = -0.022740

T3: 0.229 s sum = -0.022742 integral = -0.022740

Output for 10000000 (10 million) segments

TRAP MPI OMP

T1: 0.335 s sum = -0.022742 integral = -0.022740

T2: 0.338 s sum = -0.022742 integral = -0.022740

T3: 0.338 s sum = -0.022742 integral = -0.022740

Output for 10000000 (10 million) segments

**Fig. 2: All results of Trapezoidal.**

MPI OMP

MPI

Serial

**Fig. 3: Comparison of all runtimes (in seconds) over all schedules**

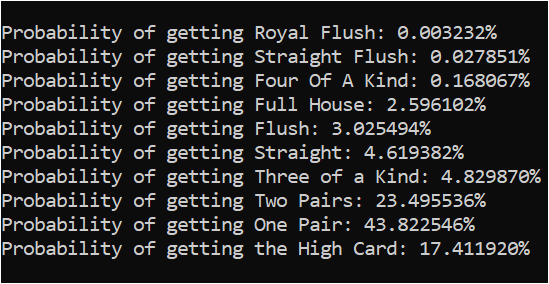
**Fig. 4a: Speed-up for Trapezoidal**

**Fig. 4b: Efficiency for Trapezoidal**

The MPI version of Trapezoid improves serial runtime by a factor of just over two! OMP-MPI, although faster than the runtime, only improves it by a factor of just under 1.5. We expected OMP-MPI to go faster than just plain MPI, since the work is being divided among more workers. However, Trapezoidal is a very simple program. It is more likely that all the extra threads will cause more overhead than is really necessary to perform this program that could be easily completed in just MPI.

**Texas Holdem Poker**

This program calculates the probability of ten different Texas Holdem Poker hands. According to the most credible of sources1, these probabilities are as follows:



**Fig. 5a: Printed probabilities across all runs.**

|  |  |
| --- | --- |
| TEXAS HOLDEM SERIAL | |
| TRIAL 1: 52.7 s |  |
| TRIAL 2: 53.047 s |  |
| TRIAL 3: 53.188 s |  |
| TEXAS HOLDEM MPI (4 PROCESSES) | |
| TRIAL 1: 36.6 s |  |
| TRIAL 2: 35.77 s |  |
| TRIAL 3: 35.8 s |  |
| TEXAS HOLDEM MPI OMP | |
| TRIAL 1: 29.65 s |  |
| TRIAL 2: 29.38 s |  |
| TRIAL 3: 29.35 s |  |

**Fig. 5b: Times per trial/program style.**

To ensure accuracy, the program prints out these probabilities at the end of the section run by each schedule (Fig. 5a). Incredibly, our program got accurate answers across nearly every run. There was an occasional, small deviation.

The parallelized section of this program is a seven-layer for loop that populates each poker hand with all 133,784,560 card combinations, and checks which kind of hand the player received. The average times of each schedule are shown and compared below (Fig. 6), and speed-up and efficiency are shown on the next page (Fig. 7a-b).

MPI OMP

MPI

Serial

**Fig. 6: Comparison of all runtimes (in seconds)**

**Fig. 7a: Speed-Up for Texas Holdem, comparison of schedules**

**Fig. 7b: Efficiency for Texas Holdem, comparison of schedules**

The big idea in Texas Holdem was load-balancing. Since work is very much non-uniform across the loop, we had to be careful about how the work was divided among the processes.

In the non-parallelized version of the program (MPI), we imitated the way the (dynamic, 1) schedule works. The master node waits for the slave nodes to request work. The master then sends out work until all hand combinations are calculated, and then the master tells everyone to finish up. Reduction is performed on the probability counters.

The parallelized version used the (dynamic, 1) scheduler inside a broad imitation of “dynamic.” As expected, this worked extremely well with Texas Holdem’s varied loop. MPI-OMP was the fastest, with a speed-up of almost two times serial. MPI-OMP’s efficiency was terrible, however, because the number of threads was 16; a fairly large number.

**Jacobi**

The Jacobi program uses Jacobi's Iteration to determine the final steady state temperature of a plate. The “plate,” in this case, is a 2D array of cells, with 1000 rows and 1500 columns. Initially, the north and east edges of the plate are set to a maximum temperature of 100 degrees, the south and west edges are set to a minimum temperature of 0 degrees, and all other cells are set to 50 degrees.

The parallelized part of the program is a double for-loop that averages each cell’s temperature based on its four ordinal neighbors. The loop ends once the largest temperature difference across all the cells is less than 0.001. Unlike in the OMP lab, every part of this program was timed, including the conversion to PPM. We are not just measuring OMP parallelization, but MPI parallelization across a whole cluster. The plates produced by the serial and the OMP-MPI (4 threads) are shown below (Fig. 8a-b).

To make the program more palatable for MPI, plates were passed around through the cluster row by row instead of all at once.

Although the plates look identical to the naked eye, it is not clear that they are perfectly identical. To test this, after each schedule finishes its Jacobi iteration, the program logs the temperature of three cells across all schedules. The cells are [10][10], [10][500] and [500,10], and their temperatures are 49.999916, 95.827171 and 4.172444 respectively. When tested, these temperatures remained constant across all schedules.



**Fig. 8a: PPM image, Serial Fig 8b: PPM image, OMP-MPI 4 threads**

Runtime of Jacobi is shown below, in Fig. 9. Speed-up and efficiency are shown in Fig. 7a-b.

MPI

Serial

MPI OMP

**Fig. 9: Comparison of all runtimes**

**Fig. 10a: Speed-Up for Jacobi**

**Fig. 10b: Efficiency for Jacobi**

An amazing thing to note is the difference between serial Jacobi and Jacobi run on the MPI cluster. The runtimes have a ratio of roughly 219/84 – MPI is more than two and a half times the original!

Although it created a marked improvement on serial, MPI-OMP ran more slowly than OMP, most likely because of hyperthreading. The introduction of extra threads into the mix, especially with the dynamic, 1 scheduler, caused a lot of overhead. The efficiency is awful.

**Conclusion**

With the combined work of OMP parallelization and MPI parallelization, you can do great and incredible things. We could never have achieved speeds like this on our measly little single thread programs.

**Sources**

1 Texas Holdem: https://en.wikipedia.org/wiki/Poker\_probability