OPEN MP CAPSTONE LAB

Emily Wasylenko

COMP 322 – Parallel Computing

11/26/18

**Purpose:** The goal of this lab is to find out which thread schedules work best on which kinds of parallelized programs, and why. To determine this, runtime, speed-up, and efficiency of three different programs are compared.

**Set-Up:**

First, three CPU intensive programs (Trapezoidal, Jacobi, and Texas Holdem Poker) were parallelized, using the OpenMP API. Next, three serial trials were run for each program, in Release mode. These runtimes served as control cases. After this, each parallelized program was run in in Release mode, on four different thread schedules: default, static, dynamic, and guided. Each schedule was run with 2, 3, and 4 threads, and each thread team was given three trials *(****Fig. 1****)*.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | TRIAL 1 | | | TRIAL 2 | | | TRIAL 3 | | |
| SERIAL | / | | | / | | | / | | |
| DEFAULT | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads |
| STATIC | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads |
| DYNAMIC | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads |
| GUIDED | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads |

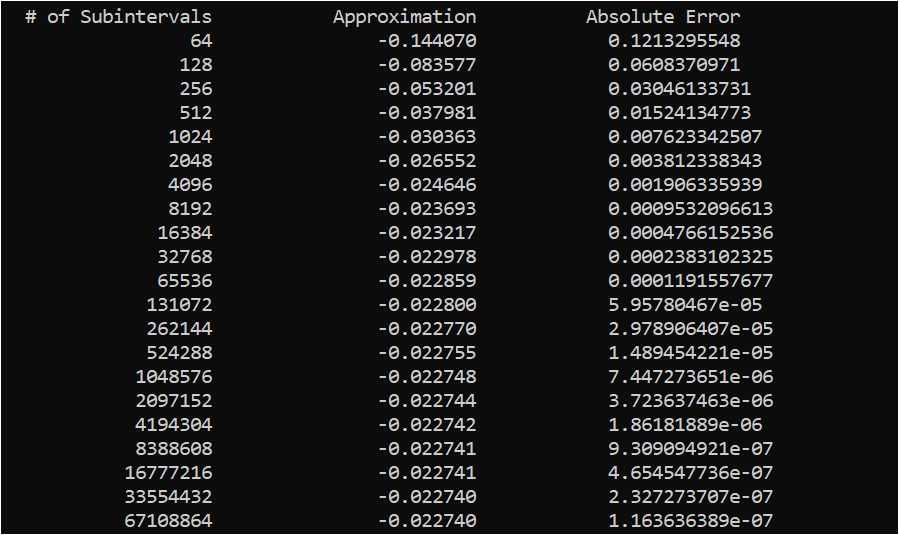
**Fig. 1: Format of trial runs**

**Trapezoidal**

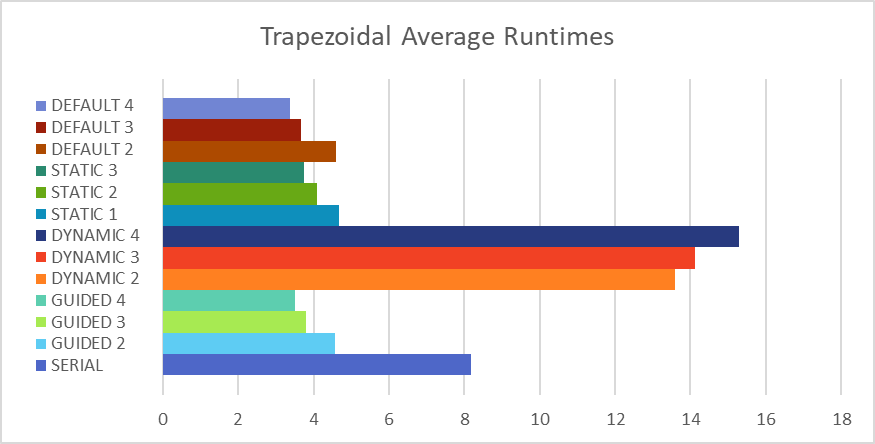
The Trapezoidal program uses the rule for trapezoidal integration to approximately solve a given integral. In this lab, the given equation is , with the solution of -0.022740. This equation is evaluated with 64 to 67108864 subintervals, with the number increasing by powers of two each time.

To ensure accuracy across all schedules, the program prints the number of subintervals, the approximate answer, and the absolute error from the actual solution after each schedule. Each schedule did produce the same results. These results are shown in Fig. 2. -0.073998

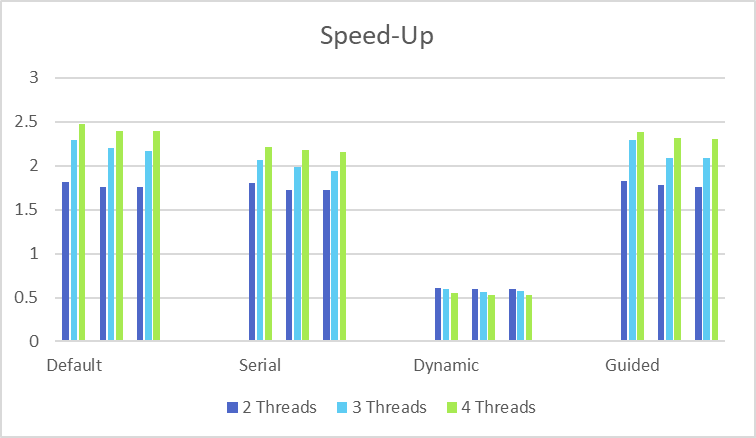
The average runtimes of different schedules are compared and displayed in Fig. 3, and speed-up and efficiency of all trials for all schedules are shown in Fig. 4a-b. Speed-up throughout this lab is (TimeParallel/TimeSerial), and efficiency is TimeParallel/#Threads.



**Fig. 2: Results of Trapezoidal. This example taken from dynamic with two threads.**

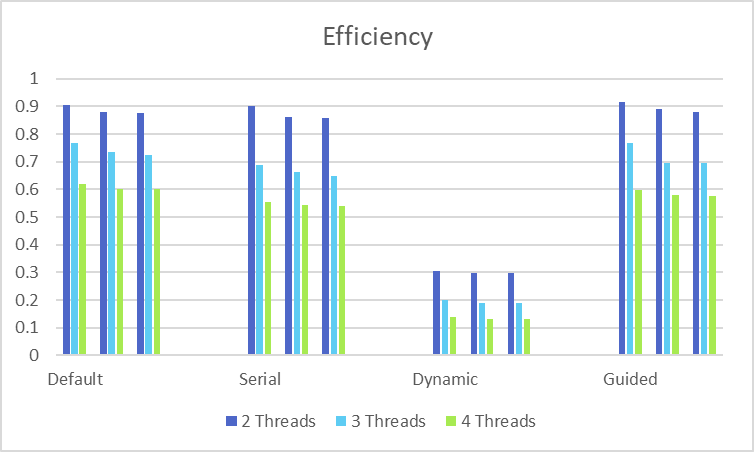


**Fig. 3: Comparison of all runtimes over all schedules**



Static

**Fig. 4a: Speed-up for Trapezoidal, comparison of schedules**



Static

**Fig. 4b: Efficiency for Trapezoidal, comparison of schedules**

From these results, we can see that all schedules work well with Trapezoidal, except Dynamic. All other schedules, on average, cut Serial time in half, producing around two times the speed-up. Dynamic’s time, on the other hand, nearly doubles Serial time. Also, for all other schedules, runtime and speed-up improves as thread number increases, which is what we expect. For Dynamic, however, runtime gets slower with increasing threads.

Default and Static are similar thread schedulers. In each, chunks of code are doled out in round-robin fashion to threads at compile time, so each thread has its work cut out for it in advance. Every thread is assigned about the same number of iterations as all other threads, thus, both schedules work best when workload is fairly uniform across the loop. For a program like Trapezoidal, where the parallelized loop repeatedly evaluates the same integral, this is ideal.

In Guided, chunk size starts out large and decreases as the program progresses. In Trapezoidal, the number of iterations increases as the loop nears its completion, which balances out the decreasing chunk size. It is likely that Guided performed slightly better than Default and Static because of this.

In Dynamic, each thread takes a piece of the program to work on at runtime and comes back to grab more once its task is done. Dynamic works best when workloads are vastly different across the loop. Since each thread can come and go depending on when it finishes, this allows for different threads to be held up for different amounts of time, but the necessary synchronization creates additional overhead. This overhead causes Dynamic to take much longer than the other schedules.

**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. The conversion to PPM was not included in the timing of this program, since parallelization does not affect this function. The plates produced by the serial and the default schedule (4 threads) are shown below.

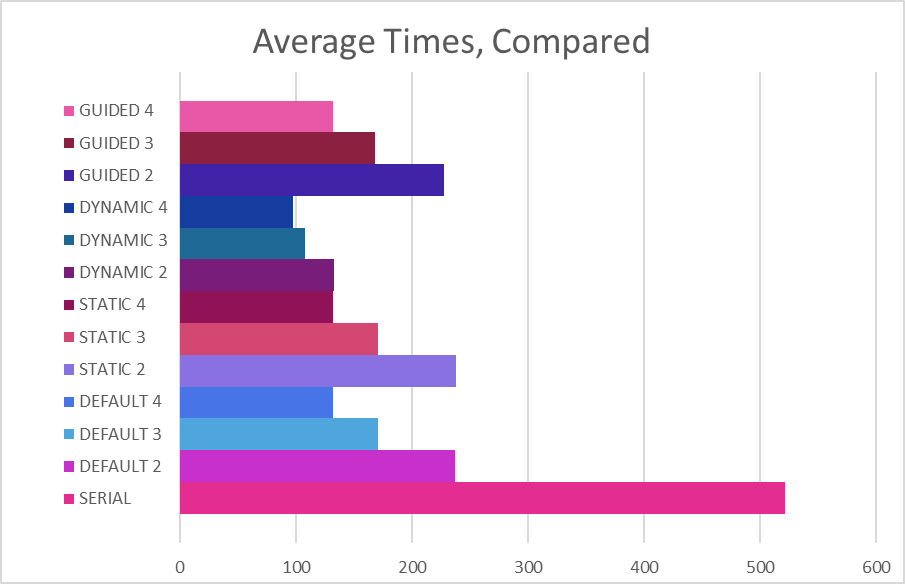
 

**Fig. 5a: PPM image, Serial**

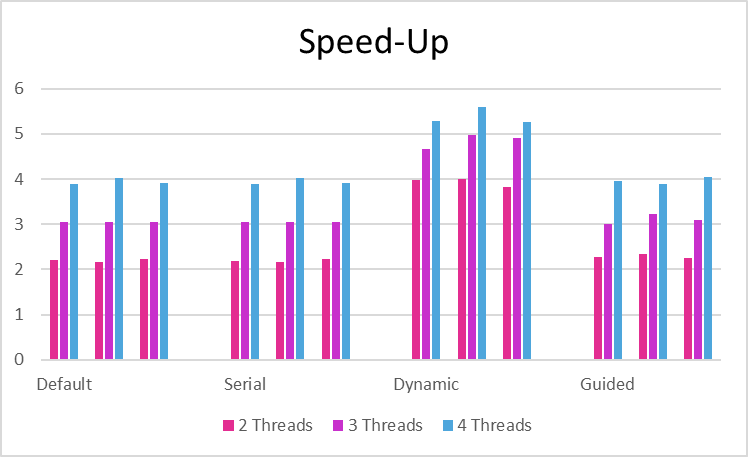
**Fig 5b: PPM image, default schedule, 4 threads**

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.

Runtime of Jacobi is shown below, in Fig. 6. Speed-up and efficiency are shown in Fig. 7a-b.

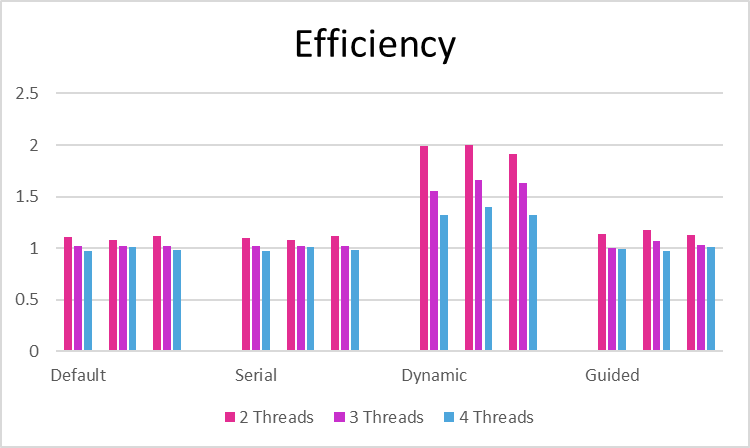
****

**Fig. 6: Comparison of all runtimes over all schedules**



Static

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



Static

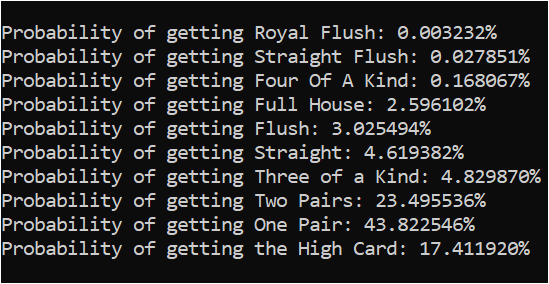
**Fig. 7b: Efficiency for Jacobi, comparison of schedules**

Like Trapezoidal’s results, Default, Static, and Guided have almost identical performance. Unlike Trapezoidal, Dynamic outperforms all other schedules by a factor of nearly two (although all four schedules do improve Serial’s runtime).

Inside Jacobi’s double for-loop, the program calculates and sets the next temperature of the cell based on the average surrounding temperatures, calculates the change in temperature for each cell, and tracks the biggest change in temperature throughout each iteration. These diverse tasks make the workload non-uniform across the loop, creating an environment well suited for Dynamic. Each thread can come and go as it finishes its task, rather than waiting for every other thread to finish up a huge block of temperature calculation before moving onto the next section.

**Texas Holdem Poker**

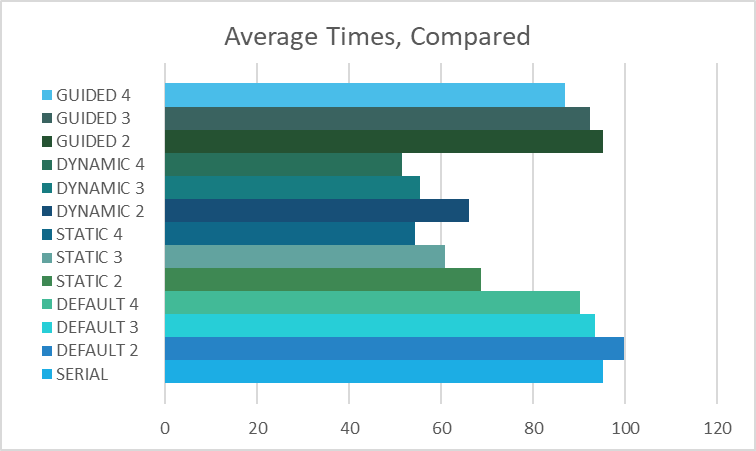
This program calculates the probability of ten different Texas Holdem Poker hands. According to the most credible of sources1, these probabilities are as follows:



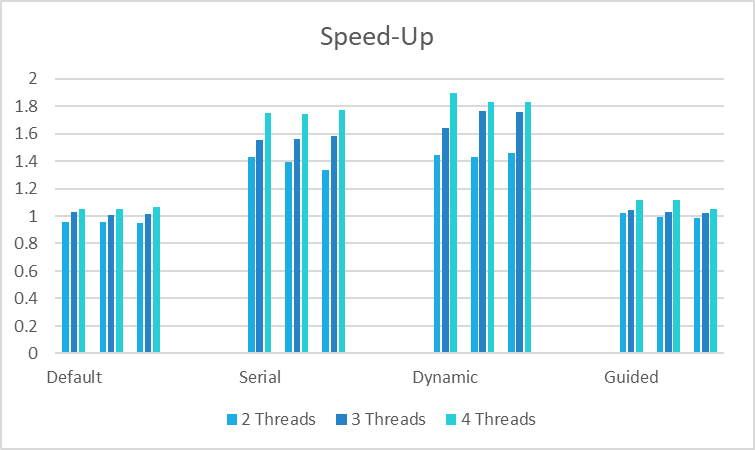
**Fig. 8: Printed probabilities. This example taken from static with three threads**

To ensure accuracy, the program prints out these probabilities at the end of the section run by each schedule (Fig. 8).

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. 9), and speed-up and efficiency are shown on the next page (Fig. 10a-b).

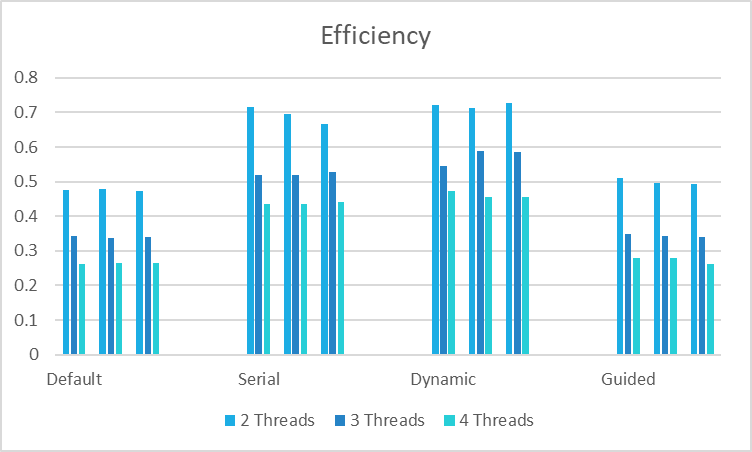
****

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



Static

**Fig. 10a: Speed-Up for Texas Holdem, comparison of schedules**



Static

**Fig. 10b: Efficiency for Texas Holdem, comparison of schedules**

Default and Guided perform quite poorly on this program. When run with two threads, their average performance is even worse than Serial. Static does well, and Dynamic does a little better than that.

It is no wonder Dynamic does so well—the parallelized loop is perfectly suited for it. Although initialization of the poker hand is fairly uniform, the ten tests are not, making the work load widely different across each iteration of the loop.

Default and Static are similar schedules, so it would make sense that they produce similar results. Surprisingly, Static outperforms Default. This is most likely due to the chunk size. Static’s chunk size = 1, which interleaves iterations. Default’s chunk size = number of iterations/number of threads. There are a very large number of iterations in Texas Holdem, causing the chunk size for each thread to also be very large. This slows down the program.

As for Guided, it makes no sense for the program to decrease chunk size per thread as it moves through the loop. There is no pattern in the workload that merits a gradual decrease in chunk size. It would be best to start out with a smaller, equal chunk size.

**Conclusion**

The results of this lab report show that Dynamic makes a great schedule, in most cases. However, if the parallel section’s main goal is to do the exact same simple task over and over again, Dynamic’s synchronization overhead will be costly. Static or Guided, overall, seems to be the safest choice.

**Sources**

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

Hyperthreading is bad!

Squirrely. Jacobi was the dark horse. Swapping cache and memory access. Shared memory is an awful way to attack this, even if we got a 50% improvement.

Go on cluster, record runtimes for Texas Holdem MPI.

One node in cluster, never do MP greater than four. The Gb Ethernet cable, comm channel is the bottleneck. One NIC (network interface controller) on each node. All four processes trying to get through pitifully slow network to get through the NIC. Processes of four on each node, one exception.

TRAPEZOID:

SPMD

JACOBI:

Master inits plate.

Then, master sends a slice of the plate to each process, including itself. Do a pass… For the middle slices to do their work, they need the row above and row below. MPI\_Scatter or MPI\_Scatterv, SECTION 3.4.6

Scatter only works on a vector (a 1D array). We have a 2D plate, but it could be built as a 1D hunk of memory.

Scatterv sends a different amount of data to each process.

We want to stay off the network as much as possible.

After each pass, how will you test whether to do another pass? MPI\_Reduce or MPI\_ReduceAll

Reduce all: every process winds up with the answer at the end of the operation.

Scatter has a thing called MPI\_Gather

Take all your cells, scatter them across all 4 processes, then Gather.

Texas Holdem (Load Balancing)

Dynamic, how will we use this? Only send to each process

Init master send “GO”

Once slave gets “go” message, request work. If there’s work to do, Master gives it some work (send card 0 to slave). else, send a STOP message (0 or 1). Slave receives the message from master. If it’s a card 0, it does the work. Else, if STOP, do reduction on your ten tallies.