Cmpt 431

Assignment 1 Report – Rafiq Dandoo, 301263787

This program takes an MxN matrix and row reduces it using Gaussian elimination into Row Echelon format. It takes two arguments, the first is the matrix, and the second is the number of threads to use.

A brief overview of the functions that do the Gaussian elimination.

F1 – getPivotrow()

This function returns the row index with the largest absolute value

F2 – swapRows()

Swaps the current top row that doesn't currently have a leading pivot variable with the row with the index returned by the above function.

F3 – reduceRows()

Divides the current row by the leading variable so it turns into the row into 1 x2 x3 … xn

F4 – reduceOtherRows()

Zeros out the column under the current pivot variable by subtracting the pivot row from row I to row j, I and j are specified by the parameters.

Parallelization Strategy

The strategy I used for parallelizing the GE problem was to attempt to parallelize each function to gain maximum performance, but as I realized after running tests, parallelizing some of the functions actually didn't provide performance benefits, due to how minimal the time spent in them was, even on very large matrices.

The first thing I parallelized was the F4 function that reduces all the other rows to create 0s under the pivot variable. I did this with three methods, Static, Dynamic and Coarse Grain Work Distribution. The strategy of each changed how jobs were assigned to threads and synchronization details.

Static Work Distribution

My first method was Static Work Distribution between all the threads. This seemed like it was the most simple way, I took the number of lines that needed to be reduced, then divided it by n number of threads, and then did number of lines modulus the number of threads to find any remaining ones, then distributed them evenly across the threads. This way each thread should have 1/n portion of the work, plus at most one extra line to solve, which felt like an acceptable compromise. Because even if there were 1 remaining line, so one thread had to do work while the others idled, they would be idle for a very minimal amount of time.

Pros of this strategy:

Very easy to program and assign tasks
Locality of data should be very close since they're all in blocks, shouldn't be many cache misses

Cons:

The big one here is that the job blocks each thread is assigned could be vastly uneven in the amount of work that needs to be computed resulting in a lot of down time.

Dynamic Work Distribution

My second method was Dynamic Work Distribution. This strategy sends the threads off, then has a global variable with how many jobs need to be done, and a global counter that each thread can access that tells it which it's on, or if it should stop. Access to this global counter is guarded by a mutex to avoid any race conditions, each thread simply runs in an infinite while loop, gets the mutex, stores its value on its local stack, and increments it then unlocks the mutex. It then checks if the variable is greater or equal to the number of jobs done, and then exits if it is, otherwise it completes a line then loops again.

Pros:

The big pro of this strategy is that each job is much smaller size, so if there are more complex jobs other threads with simpler tasks can do more of them while a certain thread does a complex one, thus more evenly spreading out the actual work load. It also minimizes idle time.

Cons:

There aren't too many cons with this process, there may be higher overhead with the threads having to check a mutex each loop, but I don't think that takes much, if any real time at all.

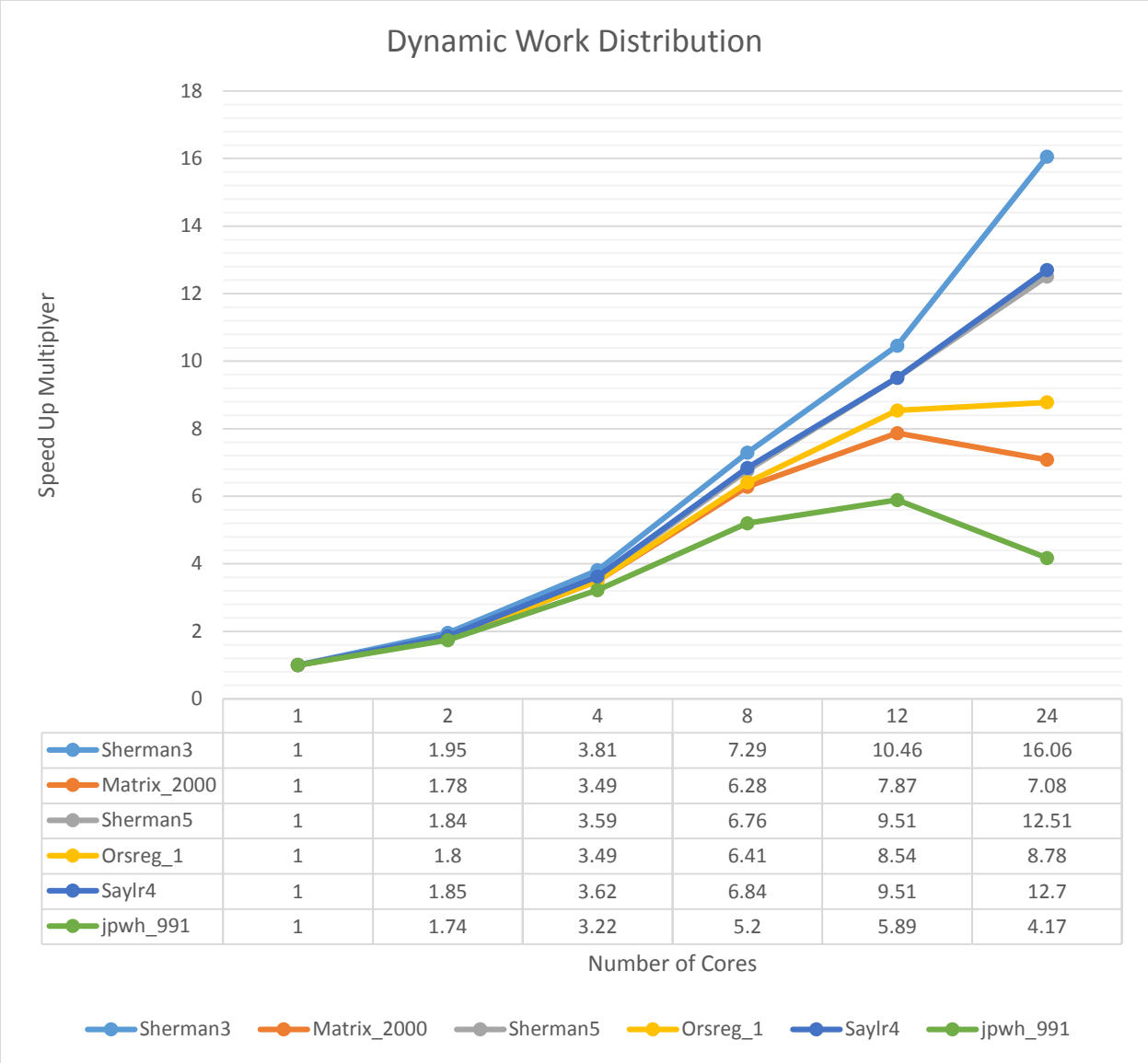
Data may also be less local since each iteration of the loop may have to jump something like 24 rows down the double array, but I think with modern day cache sizes even fairly large arrays could fit into a cache. But it still remains that there may be more cache misses with this route.

Coarse-Grain Dynamic Work Distribution

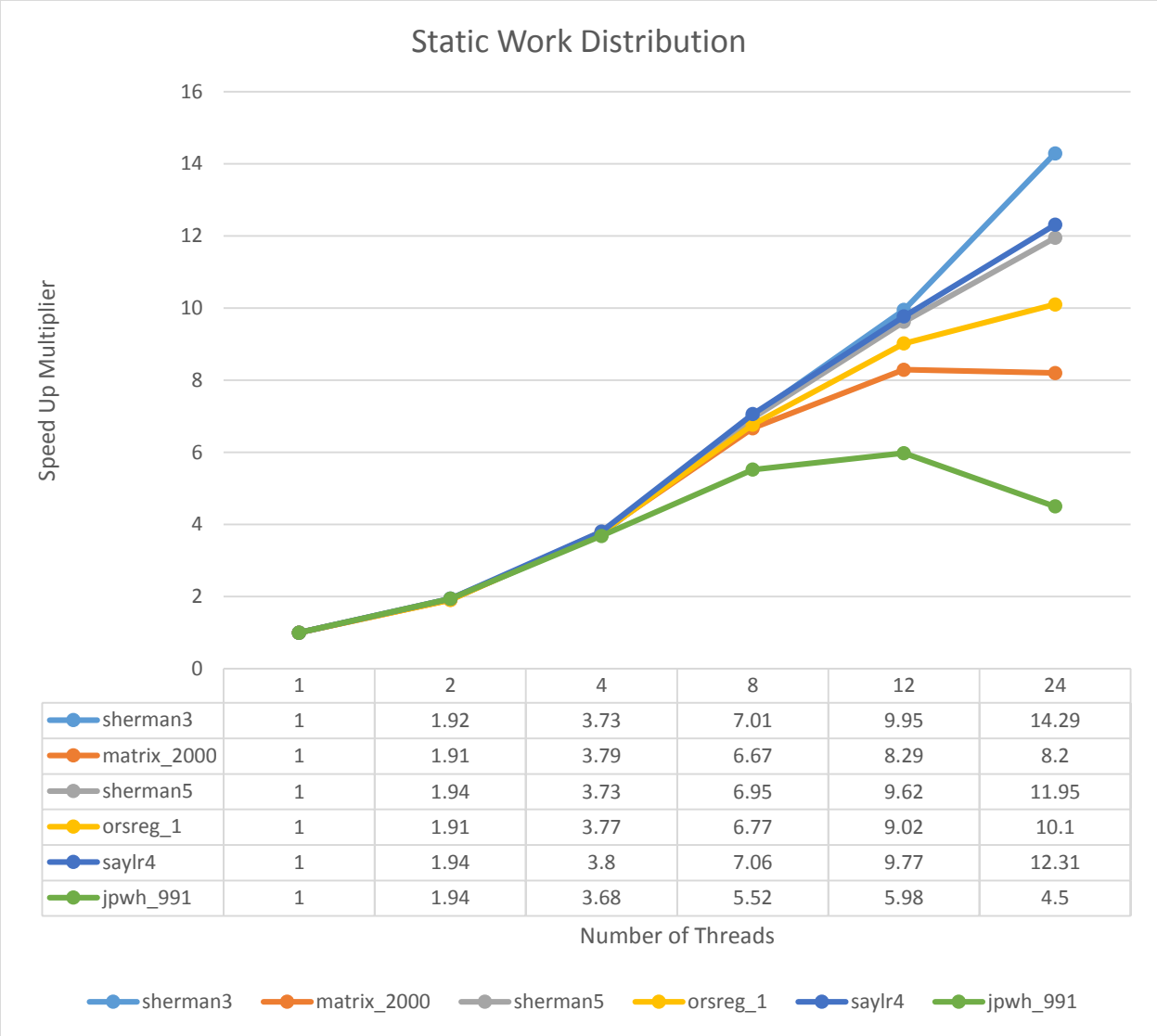
I attempted to do this method, but just couldn't figure out a good way to choose a block of jobs that's either not too big like Static distribution, or not to small like dynamic. This was mostly due to the fact the matrices size and number of threads can vary by a large amount, and these different sizes can drastically change what would be more optimal. This ended up making it really hard to either find an evenly distributable amount that's larger the regular dynamic allocation, and smaller than static's.

In the end I don't think coarse grain would improve much, if any over regular Dynamic Distribution because the main point of coarse grain seems to be to lower overhead cost and keep all cores working, but this version of dynamic distribution doesn't have much if any overhead cost.

The reason I didn't try dividing rows into multiple blocks to have multiple threads running on a single row is because it seemed to me the main reason to do that was to evenly divide jobs between processors where you have a very large amount of processors, like in the case of having a 10x10 matrix and having 24 processors, if you just give 1 processor to 1 line, you'll have 14 idle. Though the only case where you really have an advantage splitting it into blocks, so that you may have 2 to n processes working on a single line, is when you have too few more processors than rows. But, this would be a fairly small and quick to compute anyways, so it doesn't seem necessary to optimize for that situation. The only time it would be very advantageous is if you had extremely large/small numbers in a fairly small MxN matrix where M < Thread count, thus having idle threads while some are computing for a while, or if you had an MxN matrix where N is extremely large but M might be less than the thread count.



	1	2	4	8	12	24
Sherman3	1	1.95	3.81	7.29	10.46	16.06
Matrix_2000	1	1.78	3.49	6.28	7.87	7.08
Sherman5	1	1.84	3.59	6.76	9.51	12.51
Orsreg_1	1	1.8	3.49	6.41	8.54	8.78
Saylr4	1	1.85	3.62	6.84	9.51	12.7
jpwh_991	1	1.74	3.22	5.2	5.89	4.17



	1	2	4	8	12	24
sherman3	1	1.92	3.73	7.01	9.95	14.29
matrix_2000	1	1.91	3.79	6.67	8.29	8.2
sherman5	1	1.94	3.73	6.95	9.62	11.95
orsreg_1	1	1.91	3.77	6.77	9.02	10.1
saylr4	1	1.94	3.8	7.06	9.77	12.31
jpwh_991	1	1.94	3.68	5.52	5.98	4.5

This is a comparison of the two different methods I used and their speedups, they're fairly similar, but show some interesting differences.

What to take away from these charts:

1) Let's look at sherman3 since it got the most speedup out of all of the matrices. On Dynamic it got a 16x speedup verse a 14x on static work distribution. The reason is because the data in the matrix is extremely large/small values, but it isn't evenly spread out. This explains why Dynamic did a greater speed up because Static evenly split up the jobs, which will make some threads have idle time due to the complexity of their group of jobs being much lower than another thread's group of jobs. This effect increases more as more threads are added, with 2, 4, and even 8 threads it's not too noticeable because the static distribution's groups of jobs are so big

that the work gets “decently” spread out among the threads, but as you add in 12, and 24 cores dynamic gets quite a bit ahead because of the fact the work is much more split up for static distribution resulting in each thread’s group of jobs having much different complexities, making some take longer than others.
You can actually observe how this effect helps Static distribution beat dynamic in Matrix_2000’s run though. Matrix_2000 has a lot of data in it, but it’s all fairly small numbers, and it’s fully populated so the workload of each thread is much more evenly balanced, and the synchronization of Dynamic slows it down more than static in this case.

2) The next interesting thing to note is that on none of them does 2,4,8,12,24 cores = 2x,4x,8x,12x,24x performance, it’s close but not quite. The main reason is because the entire program isn’t parallel, and there are points where you will have processers idling for brief periods waiting for other threads to complete, or waiting on a mutex to unlock, this overhead cost will make it so you never actually get 2x the cores you add.
The cost of overhead from synchronization and creating threads appears more and more as you add in threads. While it does increase the overall run time drastically, it runs into more and more diminishing returns for adding more cores.
What I noticed during testing is that creating threads seems to have quite a bit of overhead cost, I would run tests on a simple 10x10 matrix and without creating threads it was done so fast the timer would show up as 0.00000. But adding even 2 threads made it ~.006, which showed me that creating a thread In that case took longer than actually solving such a small matrix.
The diminishing returns happen because as you add more threads, more synchronization is required between each one, resulting in some threads waiting a bit more and also creating more threads adds up to more overhead cost.

   One thing to note is you can witness this based on the data in the charts, for static distribution there’s no overhead on synchronization, only on creating threads, so at 2 threads it’s fairly close to 2x, and they’re all pretty even. On dynamic however you can notice a pretty big gap, from 1.74 to 1.95. The reason for this is because, jpwh_991 is fairly small and simple to compute, so that overhead cost cuts into more of the run time, where was sherman3 has high complexity and a massive amount of numbers to compute, so that overhead of creating threads takes less percentage of overall time.

3) The next interesting thing to note is what happens at 12 threads, this seems to be the point where adding more threads decreases performance, or does next to nothing for most of the matrices. The reason performance actually starts to fall for matrices like jpwh_991, or stall like matrix_2000, is because these ones are the fairly “small” or easy to compute matrices, and the overhead cost of creating more cores, and having more threads have to wait for synchronization outweighs the actual work they do because of their low compute times. That’s why this hurts performance on some matrices but not others, is due to the computing time complexity of each row, in some it might be because they’re so large, in others it might be because the numbers are massive.

I did test on my home computer, and found pretty equal speed up results up to 4 cores, but as my computer at home only has 4 cores, the results for 8, 12, 24 all dropped due to the fact there is going to be a lot of idling threads since there's only 4 cores and the OS has to schedule the threads on a core.

	1	2	4
saylr4	1	1.92	1.73
matrix_2000	1	1.85	1.69
orsreg_1	1	1.86	1.7
sherman3	1	1.94	1.73
sherman5	1	1.91	1.72

I'm not entirely sure why everything slowed down with 4 threads. I closed all background processes so there shouldn't be anything stealing cpu cycles, but my guess is something was stealing cpu cycles, possibly the OS and it's processes. That's or the intel might have some optimization going on for fewer cores, possibly using two hulk cores and two smaller ones. This would be supported by the absolute values of the speed results, my home machine completed the jobs almost 3.5x faster when comparing sequential runs.
Amobea sherman3 1 thread time : 723 seconds
I5-4690 sherman3 1 thread time : 202 seconds

These are the only two ideas I can think of why 4 cores declined in performance on my machine but improved a lot on the amoeba ones.