PARALLEL & DISTRIBUTED SYSTEMS

Project-1

Name: Deepak Subramani Velumani

Department: MS in CS

UR Student ID: 32132007

Experimenting with Threads: Understanding Locality, Load Balancing, and Synchronization Effects – Gaussian Elimination

Aim:

To parallelize the given sequential C program for Gaussian Elimination, using the pthreads library, optimise the code, and to create at least 2 different versions of the parallelized code, run then in 2 different systems and compare their performances for different parameters of input matrix sizes and the number of threads run by the pthread library.

Software Used:

GCC C compiler (version 11.2.1), pthread library, GProfiler.

Gaussian Elimination Algorithm:

In this method a matrix containing, the coefficient of several input variables with the constant term appended to the right side column (also known as augmented matrix) is taken as input. Matrix operations such as Swapping 2 rows, multiplying a row by a non-zero number, and adding/subtracting a multiple of one row to another row are recursively performed on the matrix until the matrix is reduced to an upper-triangular matrix with the leading diagonal as ones (which is called the echelon form).

In the code the getPivot() function does the role of swapping necessary rows [Time Complexity = O(n^2)] , and the remaining portion of the computeGauss() function takes care of the multiplication of a row, and subtraction of one row to another row operation [Time Complexity = O(n^3)].

Profiling of the Algorithm:

The GProfiler (gprof) tool was used to determine the time consumed by all the functions in the program along with their shares.

The following image shows the gprof analysis results for a matrix size of 2048:

```
[dsubrama@node2x14a seq]$ ./gauss_internal_input -s 2048
Application time: 10.158408 Secs
[dsubrama@node2x14a seq]$ gprof ./gauss_internal_input
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.93     10.12    10.12        1    10.12    10.12  computeGauss
  0.30     10.15     0.03        1     0.03     0.03  initMatrix
  0.00     10.15     0.00     2048     0.00     0.00  getPivot
  0.00     10.15     0.00        1     0.00     0.00  allocate_memory
```

The following image shows the gprof analysis results for a matrix size of 4096:

```
[dsubrama@node2x14a seq]$ ./gauss_internal_input -s 4096
Application time: 83.047938 Secs
[dsubrama@node2x14a seq]$ gprof ./gauss_internal_input
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.01     82.84    82.84        1    82.84    82.85  computeGauss
  0.16     82.97     0.13        1     0.13     0.13  initMatrix
  0.01     82.98     0.01     4096     0.00     0.00  getPivot
  0.00     82.98     0.00        1     0.00     0.00  allocate_memory
```

We can make the following observations from the profiler results:

- The $computeGauss()$ is the function which runs for almost the entire time of the program. In fact, it ran for about 99.93% (10.12 s) of the time for matrix size of 2048. This is in accordance with the fact that, the $computeGauss()$ which performs the multiplication & subtraction of one row from the other operation, has the highest time complexity of O(n^3), compared to the $initMatrix()$ and $getPivot()$ functions which only have a time complexity of O(n^2). Also, the effect of the O(n^3) complexity is clearly visible in the run with a matrix size of 4096, where the $computeGauss()$ takes almost the full 100% (82.84 s) share of the program runtime.
- The $initMatrix()$ function which initializes the initial values for the matrix ran for about 0.3% (0.03 s) of the time for a 2048 sized matrix and for 0.16% (0.13 s) of the time in a 4096 sized matrix.
- The $getPivot()$ function which just swaps rows, having a time complexity of O(n^2) ran for negligible time in 2048 sized matrix and ran for 0.01% (0.01 s) of the time in the 4096 sized matrix. The fact that this function only performs only read/write operations on continuous memory locations, but no starting/joining of threads can be reason for its fast execution than its rival O(n^2) function $initMatrix()$.
- The $allocate\_memory()$ has a time complexity of only O(n) and hence it's time taken is negligible and close to 0 seconds for both 2048 & 4096 matrix sizes.

Parallelization Strategies Used:

I have parallelized this program in 3 different ways, and call them as follows:

(i) Version-1: Factorization Parallelization
(ii) Version-2: Row wise Blocking
(iii) Version-3: Mutex Locking

Verification of Correctness of Versions:

As a note, this all the 3 versions have tested with the sizes of 128, 256, 512, 1024 & 2048 with varying number of threads from 1 to 72, on my local laptop, node2x18a & node2x14a server machines.

The "$gauss\_internal\_input.c$" file in all the 3 parallel optimized versions has been tested by setting the verify flag in the code to 1, and then the output vector values will be (0, -0.5) for the 1st row, (0.5, 0.5) for the last row & (0.5, 0) for the remaining rows, just as how the output is expected to be, which is exactly same as the sequential version's output. A sample of the output for all the versions for a small matrix size of 16 (for the sake of fitting in the PDF), is shown below:

```
Matrix Size: 16 ; Threads: 4
Application time: 0.003811 Secs
0.00000 -0.50000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.00000
0.50000 0.50000
```

The "$gauss\_external\_input.c$" file in all the 3 parallel optimized versions has been tested by ensuring that the 'Error' obtained for sequential versions with all the Matrices matches with all of the parallel versions when run with the corresponding matrix input files. A sample output of the error output of the program, with same error values for the sequential and parallel versions are shown below:

```
Matrix File: matrices_dense/jpwh_991.dat; Matrix Size: 0 ; Threads: 4
Time:  0.523470 seconds
Error: 4.814101e-15
```

(i) Version-1: Factorization Parallelization

In this method, I have parallelized the 2nd half of the outer loop of the $computeGauss()$ function, which contains the matrix factorization code (consisting of 2 nested for loops), that repeated multiplying of a row and decrementing another row, with the multiplied elements.

This is accomplished by repeatedly creating and joining threads within the outer for loop of the $computeGauss()$ function, for every iteration in the loop. Hence all the threads are created and joined for 'nsize' (matrix length) number of times.

The reason of parallelizing only the 'matrix factorization code' is, it is this portion of the code that has the maximum time complexity [O(n^3)] as discussed in the 'Profiling of the Algorithm' section.

This program exhibited a good amount of speedup when the number of threads were increased, but however when the number of threads were increased beyond 16 or 32 (depending on the matrix input size), it actually started to reduce the speed of the program rather than increasing. This effect is due to the fact that, the threads are initialized altogether during every single iteration of the outer loop of the $computeGauss()$, which repeats for 'nsize' number of times. The upper limit on the number of effective threads, actually increases when the size of the matrices is increased.

The results showing the run times of this version is shown in the results section.

(ii) Version-2: Row wise Blocking

This method is a modified version of Version-1 code, where the same 2nd half of the outer loop of the $computeGauss()$ function is parallelized, but in addition, each row is spitted into multiple blocks of fixed block sizes. After the split-up, each vertical set of blocks lying right below each other, is executed by each thread per iteration of the main outer loop of the $computeGauss()$.

This function was implemented to improve the speed of execution of the following code line:

$$matrix[j][k] -= pivotVal * matrix[i][k];$$

For matrices of large sizes, splitting the execution this way, can help ensure that the entire $matrix[i][k_{block\_start}\ to\ k_{block\_end}]$ is stored within the processor's local cache until the entire decrement process for the $matrix[j_{start}\ to\ j_{end}][k_{block\_start}\ to\ k_{block\_end}]$ is completed by the same thread on the same processor. Here since $matrix[i][k_{block\_start}\ to\ k_{block\_end}]$ is always stored in the local cache of the processor, this is expected to improve the efficiency of the program execution.

This version of the parallelization ran slightly slower than the Version-1. The expected reason is, since Version-1 already writes & reads from/to the array in continuous memory locations, they are already stored well, and updated on the cache, providing better accuracy.

However, the efficiency of this method will increase much more, for increasing sizes of the input matrix.

(iii) Version-3: Mutex Locking

In this method, the $computeGauss()$ function itself is parallelized, and pthread mutex locks are used to ensure that the critical section of the function $getPivot()$ has a barrier mutex lock right before and after its execution. This function performs the same role of Version-1, but complete avoid the need

of creating and joining threads in every iteration of the outer loop of the $computeGauss()$ function. So instead of the creation & joining processes, the mutex locks will take care of the order of execution of the (critical section of the) code. This lock is accomplished, by declaring a single mutex (called $threads\_lock$) and 2 pthread conditional signals (called $cond$ & $signal\_main$). Initially each thread will acquire 'threads_lock' and increment an integer variable called $threads\_completed$. On the meantime the thread-0, will check if the $threads\_completed$ has reached the value of $num\_threads$, and if not it will wait to receive the $signal\_main$ signal from the other threads. Within the same mutex lock of the other threads, $threads\_completed$ will be check if it ever becomes equal to $num\_threads$, and the last thread will send signal to the main thread via $signal\_main$ once all threads have finished running. This will cause the thread-0 to run the $getPivot()$ function alone, and then send a $pthread\_broadcast$ message to all other threads to start resuming the next iteration of the loop.

This function was tested to be the fastest version of all, since mutex locking runs faster than the overload of creating and joining new processes in every iteration.

Parallelization Strategies Used:

All the experiments were carried out in 2 different servers, namely the **node2x14a** and **node2x18a**. The performance of the versions in both these system nodes are discussed below:

A) Results on the node2x14a Server:

I) Environment Specification:

The CPU present in the node2x14a server is the "Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz" model. The specification of the processor is as follows:

CPU min MHz: 1200

CPU MHz: 1200

CPU max MHz: 3300

CPU Type: Cores: 56, Architecture: x86_64, Address sizes: 46 bits physical, 48 bits virtual

Cache Size: L1d: 896 KiB, L1i: 896 KiB, L2: 7 MiB, L3: 70 MiB

Memory Organization: RAM: 94.3 GB, Swap: 26.6 GB

OS: Linux Fedora 34

C Compiler: GCC 11.2.1

Optimization flags: -o, -pthread

II) Application Settings:

The following configurations for all the 3 versions of the codes were run on this Server and the performances were noted:

(i) **Number of Threads:** 1/2/4, 16, 32, 55

(ii) **Internal Input Matrix Sizes:** 128, 256, 512, 1024, 2048

(iii) **External I/P Matrices:** jpwh_991.dat, orsreg_1.dat, sherman5.dat, saylr4.dat, sherman3.dat

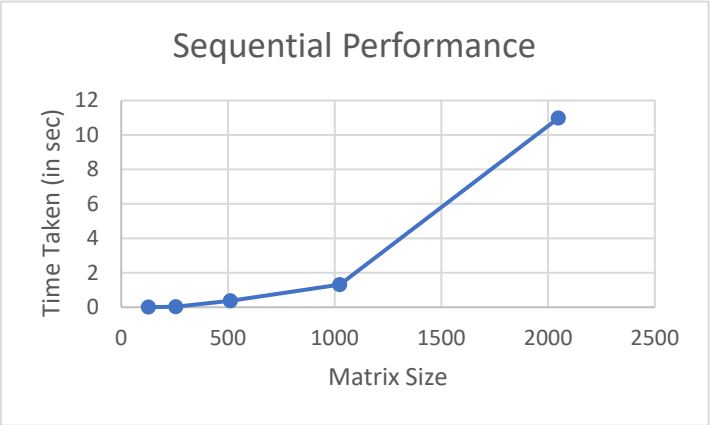
III) Performances:

Note: All time units are in seconds.

(i) Sequential Performances:

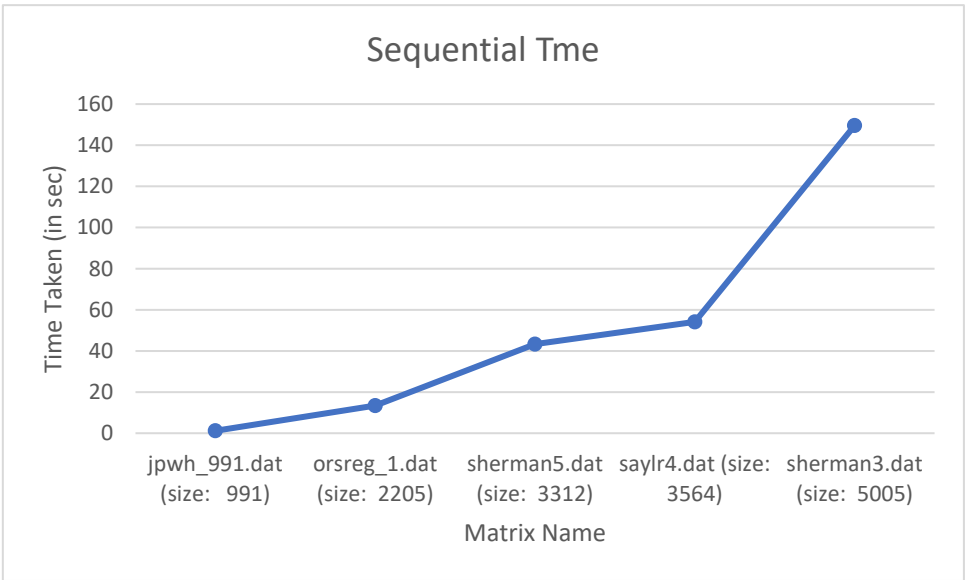
(a) Internal Input:

Matrix Size	Time Taken
128	0.00411
256	0.030433
512	0.374293
1024	1.310601
2048	10.977564



(b) External Input:

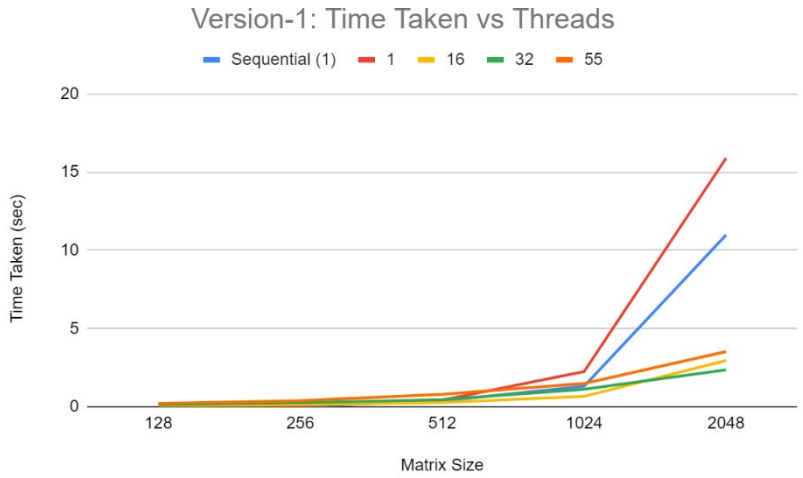
Matrix Name	Time Taken
jpwh_991.dat (size: 991)	1.183904
orsreg_1.dat (size: 2205)	13.413818
sherman5.dat (size: 3312)	43.277793
saylr4.dat (size: 3564)	54.102155
sherman3.dat (size: 5005)	149.583424



(ii) Version - 1 : Factorization Parallelization

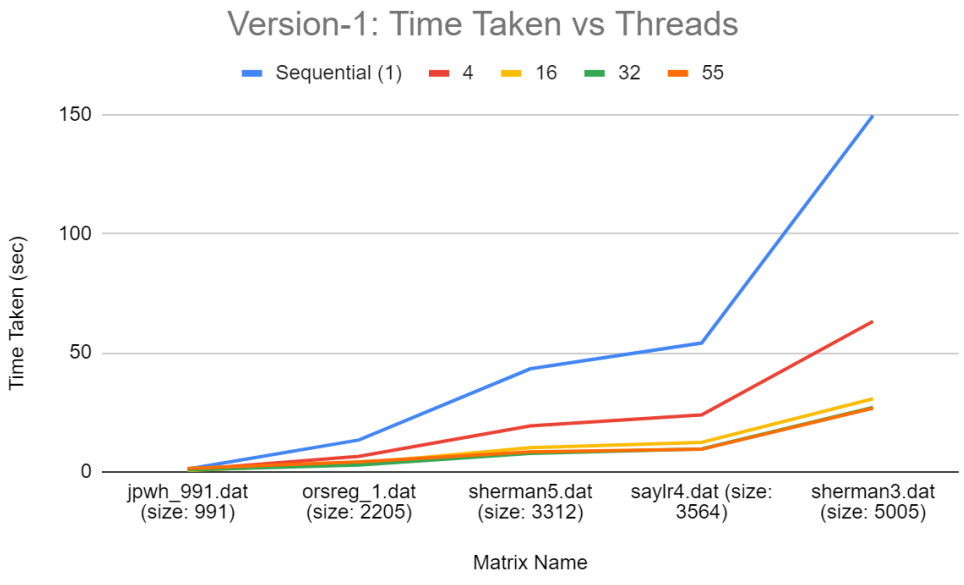
(a) Internal Input:

Mat Size / Num Threads	1	16	32	55
128	0.016296	0.070274	0.139839	0.210987
256	0.075757	0.131981	0.249234	0.392346
512	0.421204	0.261625	0.451907	0.790281
1024	2.244897	0.664143	1.115004	1.48489
2048	15.891465	2.95161	2.359622	3.525436



(b) External Input:

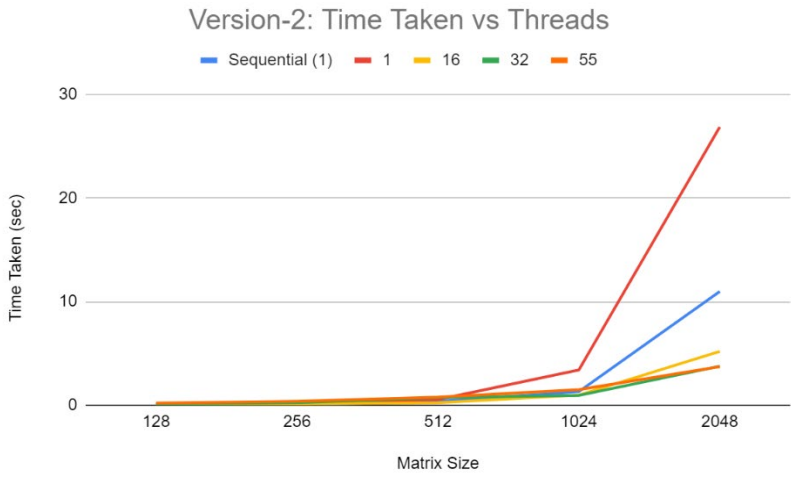
File Name / Num Threads	4	16	32	55
jpwh_991.dat (size: 991)	0.833888	0.62047	0.878244	1.430643
orsreg_1.dat (size: 2205)	6.503568	3.574812	2.911754	4.209121
sherman5.dat (size: 3312)	19.33827	10.16577	7.779744	8.381254
saylr4.dat (size: 3564)	23.94361	12.40871	9.65019	9.523194
sherman3.dat (size: 5005)	63.15016	30.67088	27.072	26.78871



(iii) Version - 2 : Row wise Blocking:

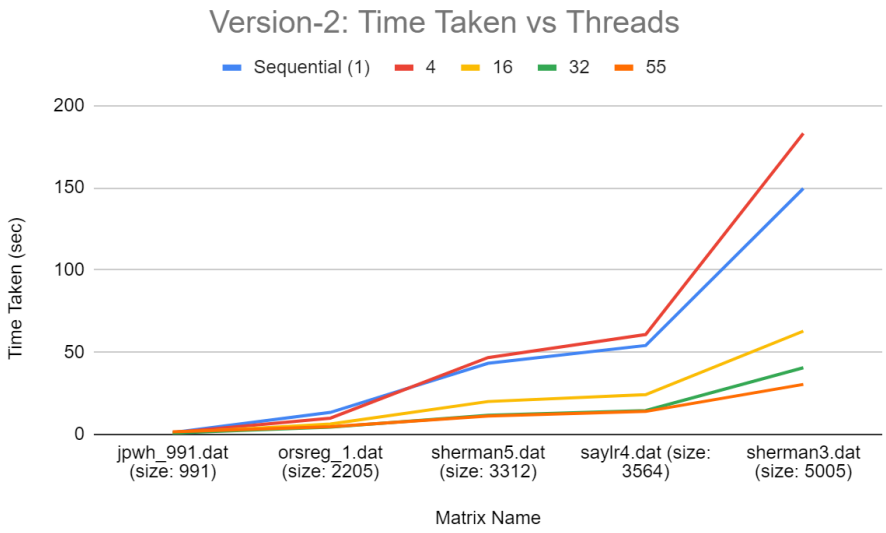
(a) Internal Input:

Mat Size / Num Threads	1	16	32	55
128	0.015734	0.069379	0.131335	0.214985
256	0.112859	0.132605	0.237233	0.38725
512	0.546195	0.251185	0.73912	0.784073
1024	3.409909	0.978819	0.949976	1.502408
2048	26.84552	5.196284	3.763661	3.725898



(b) External Input:

File Name / Num Threads	4	16	32	55
jpwh_991.dat (size: 991)	1.152353	0.931091	0.86875	1.559781
orsreg_1.dat (size: 2205)	9.923419	6.360693	4.405164	4.669612
sherman5.dat (size: 3312)	46.7155	20.02233	11.71491	11.2235
saylr4.dat (size: 3564)	60.79134	24.23166	14.52366	13.9928
sherman3.dat (size: 5005)	183.1465	62.82579	40.54938	30.43672

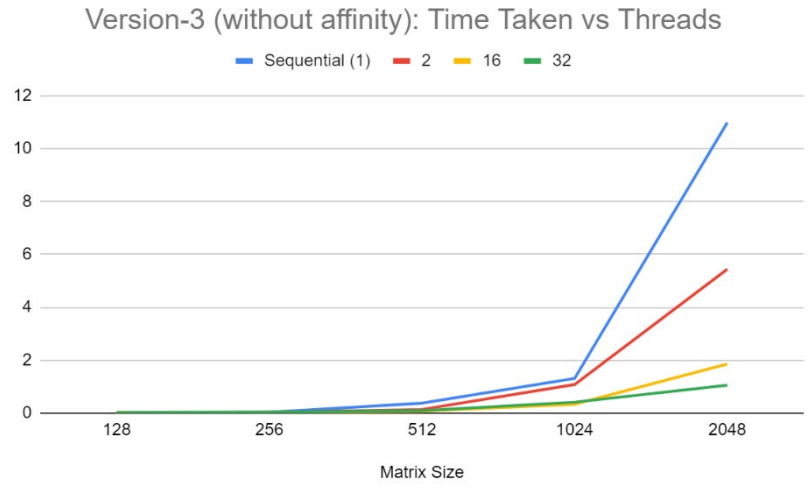


(iv) Version-3: Mutex Locking (without affinity):

Note: This is the performance of Version 3 code without manually setting affinity of all the threads to exclusive CPU cores.

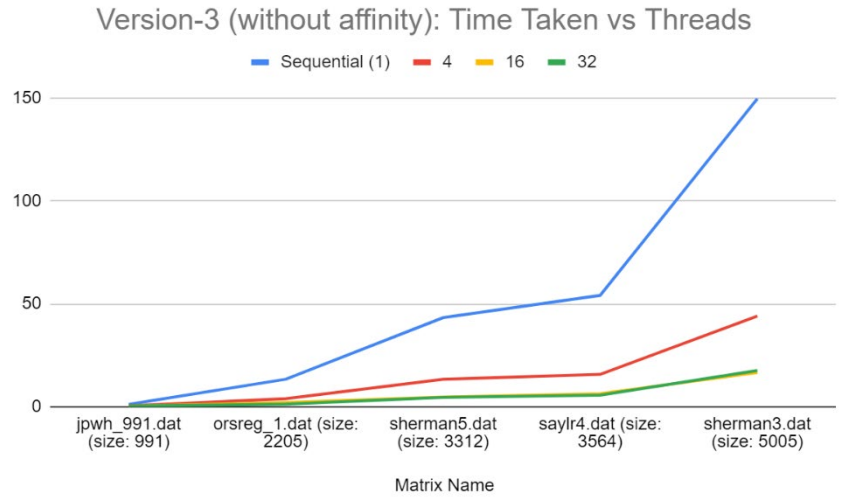
(a) **Internal Input:**

Mat Size / Num Threads	2	16	32
128	0.008066	0.011742	0.018353
256	0.034095	0.025214	0.037094
512	0.135064	0.073538	0.094268
1024	1.07875	0.334833	0.410688
2048	5.436107	1.853772	1.054302



(b) **External Input:**

File Name / Num Threads	4	16	32
jpwh_991.dat (size: 991)	0.491538	0.302549	0.393104
orsreg_1.dat (size: 2205)	3.949603	2.131055	1.302634
sherman5.dat (size: 3312)	13.36625	4.729388	4.622738
saylr4.dat (size: 3564)	15.75785	6.319857	5.679604
sherman3.dat (size: 5005)	44.0613	16.63661	17.6139

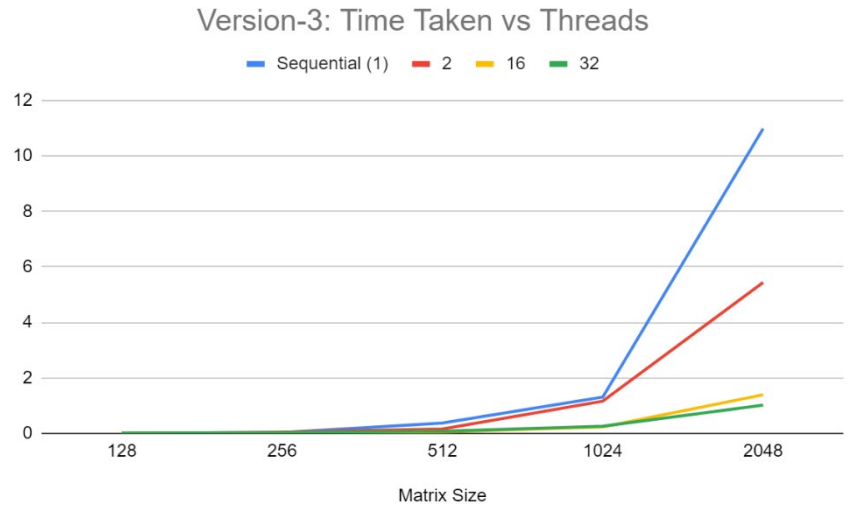


(v) Version-3: Mutex Locking

Note: This is the performance of Version 3 code after manually adding the set affinity attribute feature to all the threads, manually grant them exclusive CPU access.

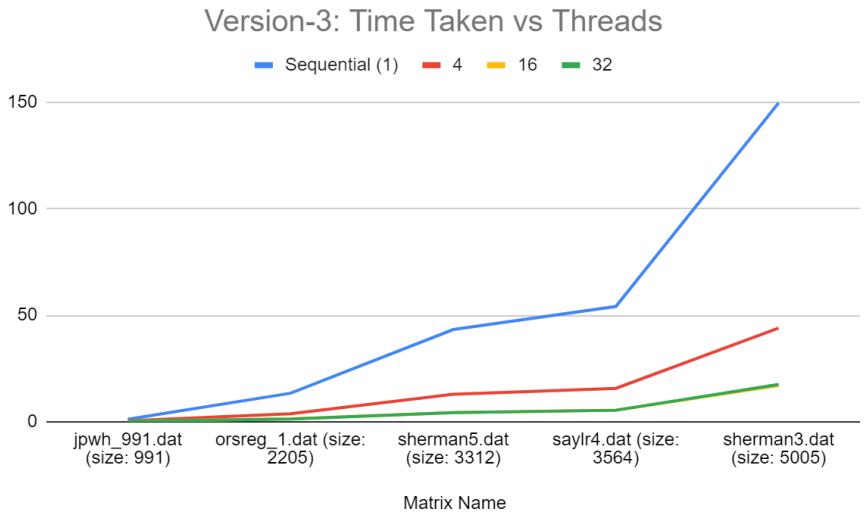
(a) Internal Input:

Mat Size / Num Threads	2	16	32
128	0.007276	0.009878	0.01809
256	0.043521	0.021522	0.034186
512	0.160191	0.057399	0.078341
1024	1.164065	0.234721	0.260833
2048	5.440565	1.392654	1.020164



(b) External Input:

File Name / Num Threads	4	16	32
jpwh_991.dat (size: 991)	0.471835	0.291244	0.282479
orsreg_1.dat (size: 2205)	3.824229	1.411881	1.294047
sherman5.dat (size: 3312)	12.91824	4.309983	4.275999
saylr4.dat (size: 3564)	15.69453	5.474061	5.424632
sherman3.dat (size: 5005)	43.93776	17.02679	17.52146



IV) Analysis of Results:

The times taken to run each function in each version of the code as measured by GProfiler are shown below:

Note: All time measurements were made by running the program on a 2048 sized matrix with 32 threads.

(i) Version-1: Factorization Parallelization

```
[dsubrama@node2x14a Version-1]$ gprof ./gauss_internal_input
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.98     14.89    14.89                             subtractElim
  0.20     14.92     0.03        1    30.10    30.10  initMatrix
  0.07     14.93     0.01     2048     0.00     0.00  getPivot
  0.07     14.94     0.01        1    10.03    20.06  computeGauss
  0.00     14.94     0.00        1     0.00     0.00  allocate_memory
```

(ii) Version-2: Row wise Blocking

```
[dsubrama@node2x14a Version-2]$ ./gauss_internal_input

Matrix Size: 2048 ; Threads: 32; Block Size: 16
Application time: 4.220221 Secs
[dsubrama@node2x14a Version-2]$ gprof ./gauss_internal_input
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.03     25.16    25.16                             subtractElim
  0.16     25.20     0.04     2048     0.02     0.02  getPivot
  0.12     25.23     0.03        1    30.09    30.09  initMatrix
  0.00     25.23     0.00        1     0.00     0.00  allocate_memory
  0.00     25.23     0.00        1     0.00    40.12  computeGauss
```

(iii) Version-3: Mutex Locking

```
[dsubrama@node2x14a Version-3]$ gcc -pg -no-pie -fno-builtin gauss_internal_input.c -o gauss_internal_input -pthread
[dsubrama@node2x14a Version-3]$ ./gauss_internal_input -s 2048 -t 32

Matrix Size: 2048 ; Threads: 32
Setting CPU Affinity : Yes
Application time: 1.241189 Secs
[dsubrama@node2x14a Version-3]$ gprof ./gauss_internal_input
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.02     11.37    11.37                             computeGauss
  0.26     11.40     0.03        1    30.08    30.08  initMatrix
  0.00     11.40     0.00     2048     0.00     0.00  getPivot
  0.00     11.40     0.00        1     0.00     0.00  allocate_memory
```

From the above results we can infer that, almost the entire time of the execution is taken by the 2nd half part of $computeGauss()$ function, and the parallel function $subtractElim()$ (in case of Version 1 & 2) called by the same, which contains the O(n^3) time complexity loop which runs after that getPivot() function in all of the 3 multithreaded versions. In addition the $initMatrix()$ function seems to have barely consumed around 0.15% to 0.26% of the runtime.

B) Results on the node2x18a server

I) Environment Specification:

The CPU present in the node2x14a server is the “Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz” model. The specification of the processor is as follows:

CPU min MHz: 1200

CPU MHz: 1200

CPU max MHz: 3300

CPU Type: Cores: 72, Architecture: x86_64, Address sizes: 46 bits physical, 48 bits virtual

Cache Size: L1d: 1.1 MiB, L1i: 1.1 MiB, L2: 9 MiB, L3: 90 MiB

Memory Organization: RAM: 94.3 GB, Swap: 26.6 GB

OS: Linux Fedora 34

C Compiler: GCC 11.2.1

Optimization flags: -o, -pthread

II) Application Settings:

The following configurations for all the 3 versions of the codes were run on this Server and the performances were noted:

(i) **Number of Threads:** 1/2/4, 16, 32, 72

(ii) **Internal Input Matrix Sizes:** 128, 256, 512, 1024, 2048

(iii) **External I/P Matrices:** jpwh_991.dat, orsreg_1.dat, sherman5.dat, saylr4.dat, sherman3.dat

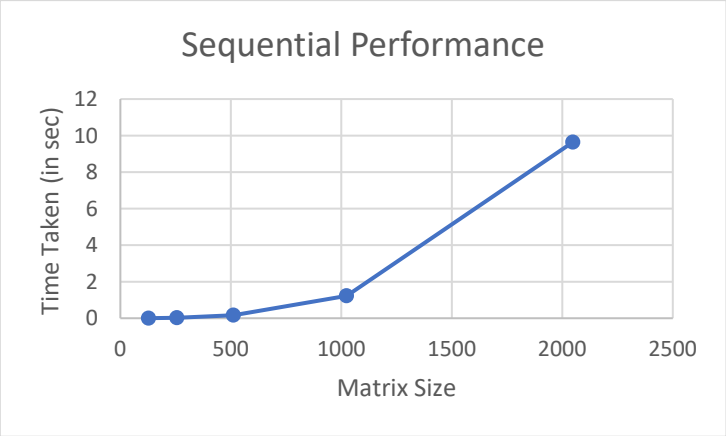
III) Performances:

Note: All time units are in seconds.

(i) Sequential Performances:

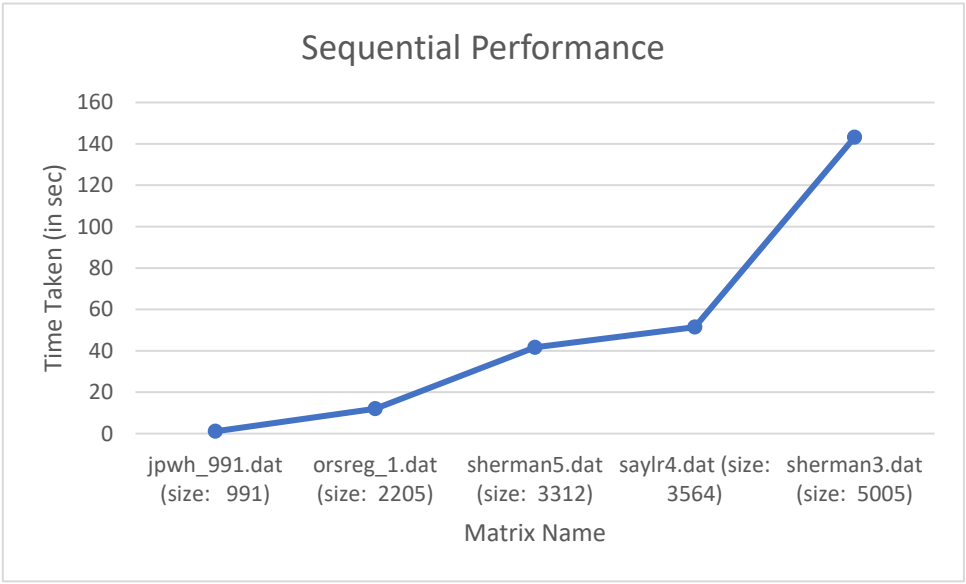
(a) Internal Input:

Matrix Size	Time Taken
128	0.003979
256	0.028694
512	0.168282
1024	1.228928
2048	9.644256



(b) External Input:

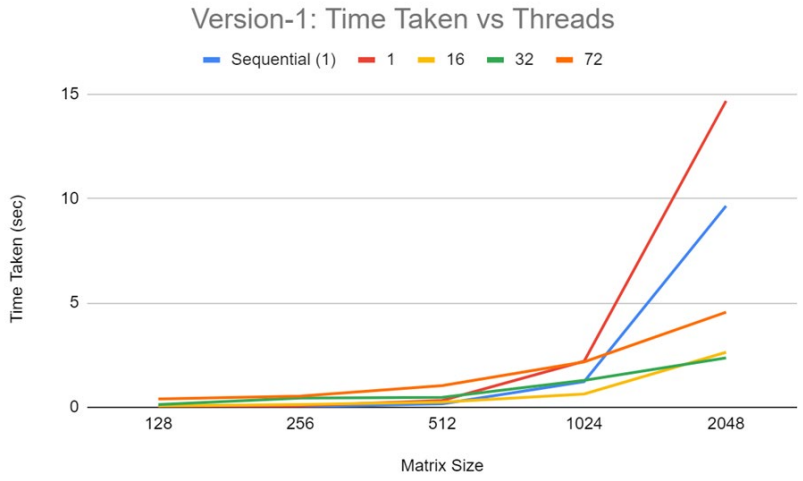
Matrix Name	Time Taken
jpwh_991.dat (size: 991)	1.102551
orsreg_1.dat (size: 2205)	12.009902
sherman5.dat (size: 3312)	41.651507
saylr4.dat (size: 3564)	51.446487
sherman3.dat (size: 5005)	143.196348



(ii) Version - 1 : Factorization Parallelization

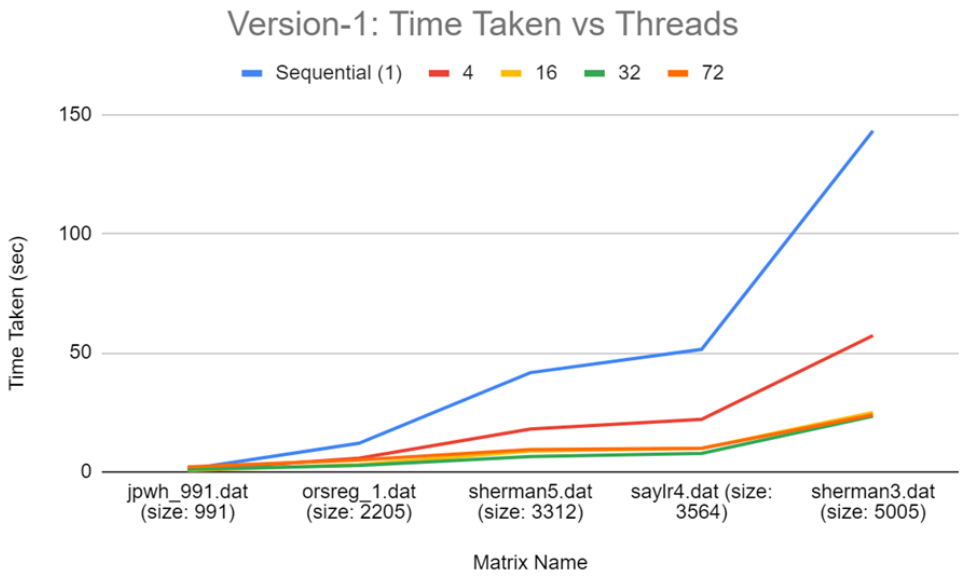
(a) Internal Input:

Mat Size / Num Threads	1	16	32	72
128	0.018207	0.070249	0.13375	0.397111
256	0.068644	0.132339	0.443028	0.53637
512	0.324174	0.246681	0.473269	1.036157
1024	2.206546	0.636434	1.289264	2.173144
2048	14.671464	2.63467	2.36825	4.551541



(b) External Input:

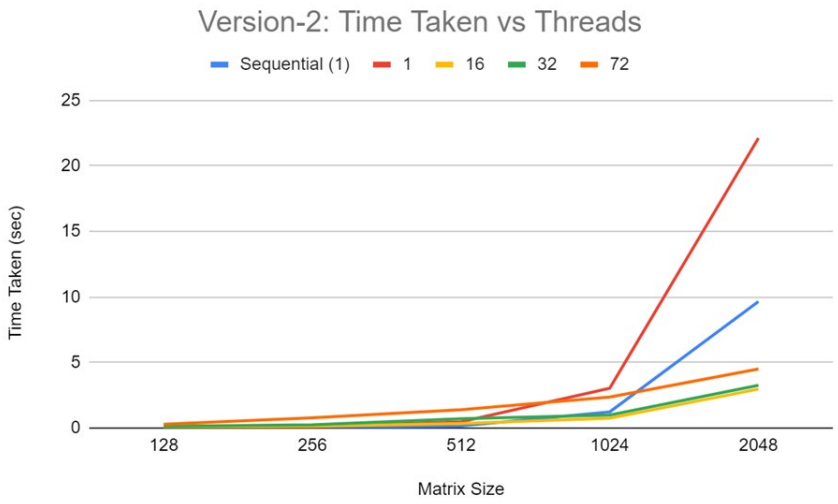
File Name / Num Threads	4	16	32	72
jpwh_991.dat (size: 991)	0.780633	0.56047	0.990569	2.105464
orsreg_1.dat (size: 2205)	5.736523	3.114312	2.677957	5.003929
sherman5.dat (size: 3312)	17.991189	8.794235	6.444104	9.420095
saylr4.dat (size: 3564)	22.081054	9.867663	7.807237	9.927974
sherman3.dat (size: 5005)	57.215988	24.9105	23.37104	23.9508



(iii) Version - 2 : Row wise Blocking:

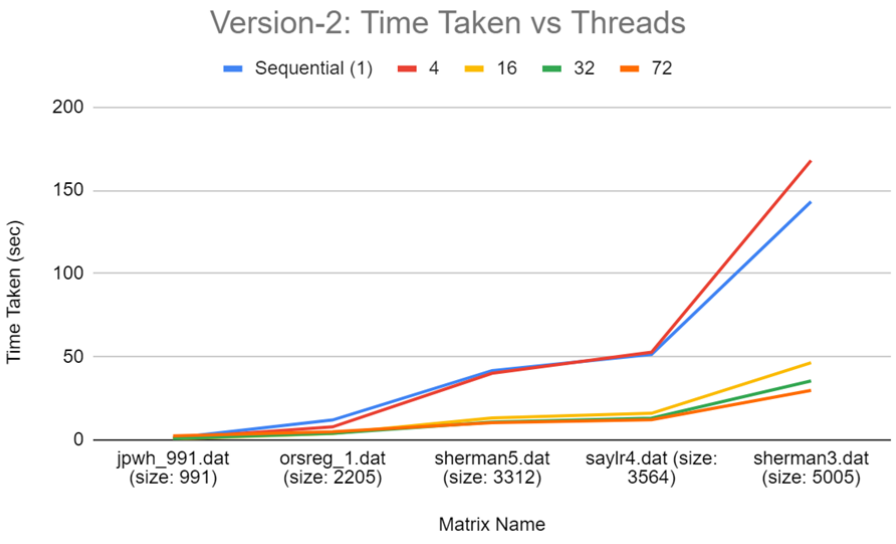
(a) Internal Input:

Mat Size / Num Threads	1	16	32	72
128	0.020921	0.069043	0.128933	0.282796
256	0.102638	0.130759	0.250281	0.772162
512	0.476818	0.348069	0.70851	1.395962
1024	3.039138	0.746356	0.994597	2.35804
2048	22.117214	2.970117	3.253115	4.506561



(b) External Input:

File Name / Num Threads	4	16	32	72
jpwh_991.dat (size: 991)	0.996957	0.670879	0.934643	2.494723
orsreg_1.dat (size: 2205)	7.864159	3.844888	4.011543	4.977277
sherman5.dat (size: 3312)	40.05914	13.20212	10.72553	10.38684
saylr4.dat (size: 3564)	52.730384	16.00873	13.0719	12.14691
sherman3.dat (size: 5005)	167.917715	46.42396	35.4999	29.8092

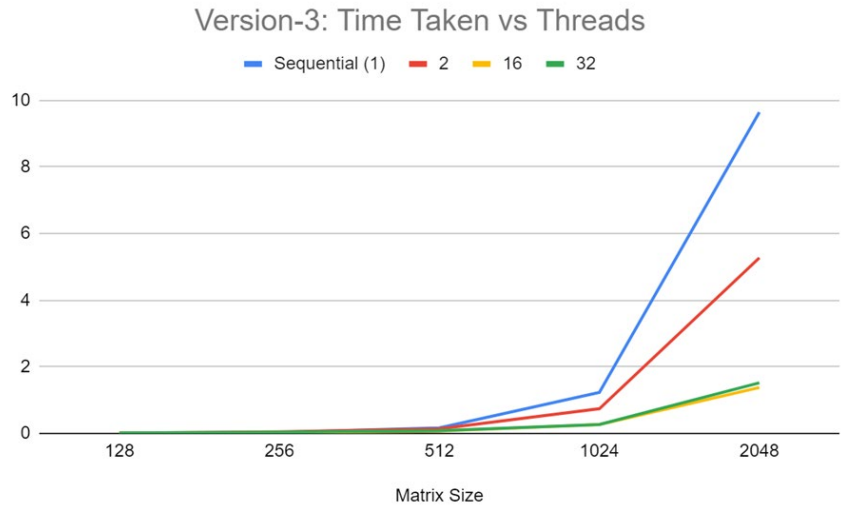


(iv) Version-4: Mutex Locking

Note: This is the performance of Version 3 code after manually adding the set affinity attribute feature to all the threads, manually grant them exclusive CPU access.

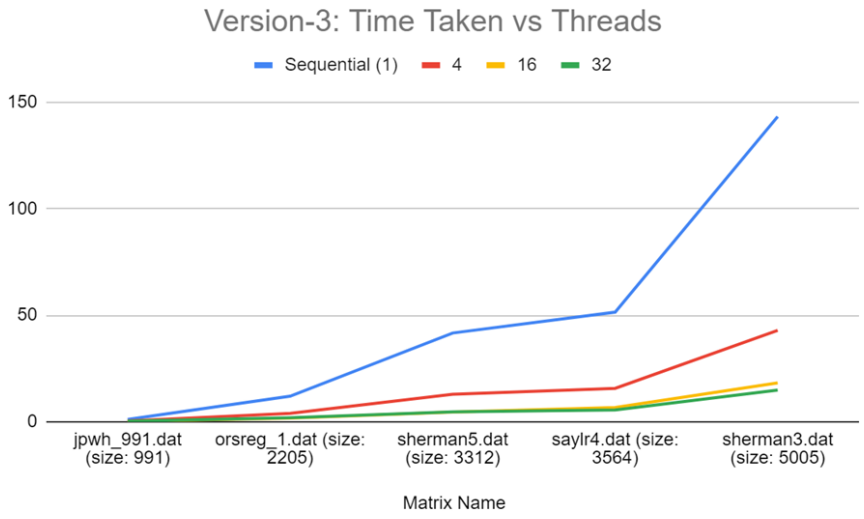
(a) Internal Input:

Mat Size / Num Threads	2	16	32
128	0.007941	0.010253	0.016532
256	0.045597	0.02406	0.034221
512	0.135924	0.07144	0.080886
1024	0.740627	0.266466	0.268919
2048	5.271772	1.377353	1.519789



(b) External Input:

File Name / Num Threads	4	16	32
jpwh_991.dat (size: 991)	0.424861	0.233257	0.293424
orsreg_1.dat (size: 2205)	3.998309	1.696003	1.929066
sherman5.dat (size: 3312)	12.910446	4.581856	4.640102
saylr4.dat (size: 3564)	15.6821	6.699445	5.584708
sherman3.dat (size: 5005)	42.873704	18.27971	14.84627



V) Analysis of Results:

The times taken to run each function in each version of the code as measured by GProfiler are shown below:

Note: All time measurements were made by running the program on a 2048 sized matrix with 32 threads.

(iv) Version-1: Factorization Parallelization

```
[dsubrama@node2x18a Version-1]$ gprof ./gauss_internal_input
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.04     14.08    14.08                             subtractElim
  0.21     14.12     0.03        1    30.10    30.10  initMatrix
  0.07     14.13     0.01        1    10.03    10.03  computeGauss
  0.00     14.13     0.00     2048     0.00     0.00  getPivot
  0.00     14.13     0.00        1     0.00     0.00  allocate_memory
```

(v) Version-2: Row wise Blocking

```
[dsubrama@node2x18a Version-2]$ gprof ./gauss_internal_input
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.98     19.66    19.66                             subtractElim
  0.15     19.69     0.03        1    30.09    30.09  initMatrix
  0.13     19.71     0.03     2048     0.01     0.01  getPivot
  0.05     19.72     0.01        1    10.03    35.11  computeGauss
  0.00     19.72     0.00        1     0.00     0.00  allocate_memory
```

(vi) Version-3: Mutex Locking

```
[dsubrama@node2x18a Version-3]$ gprof ./gauss_internal_input
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.08     19.48    19.48                             computeGauss
  0.15     19.51     0.03        1    30.08    30.08  initMatrix
  0.05     19.52     0.01     2048     0.00     0.00  getPivot
  0.00     19.52     0.00        1     0.00     0.00  allocate_memory
```

Similar to node2x14a, even in the node2x18a server machine, we can infer that the 2nd half of the computeGauss() Type equation here. function takes about 100% of the runtime, and functions such as $initMatrix()$ consumes about 0.2% of the entire runtime, which is almost negligible. Hence the entire approach has been at always optimising the inference loop which has O(n^3) time complecity in all the three applications.

Results:

Therefore, the given code for Gaussian Elimination is parallelized with 3 different approaches using pthreads library, tested their performance for different sizes & threads in the node2x14a & node2x18a servers. Overall, the observation is that the 3rd Version of the parallelization which created the threads just once and used mutex blocks to serve as barrier locks for the critical path in every iteration gave the best performance in terms of speed of execution, since we avoid the overhead of a new thread creation in every iteration. Also, the Version-1 (row wise parallelised) of the code was more efficient than the Version-2, which split the matrix into blocks and parallelized them. The reason for this is, the Version-1 which was already executing in row major form had all the elements it accesses using different threads in continuous memory locations for each thread, allowing the variables to be stored in the local cache of the processors and boosting the performance. Now since the cache hit was already low in the Version-1, Version-2 was unable to improve the performance further, due to some overheads involved in blocking. However, the Version-2 is expected to show a good speedup if executed with very large matrix sizes, where the actual blocking of overhead will become insignificant. So hence I developed the Version 3 which is an improvement of Version – 1 (since it was better among the 2 previous versions), where the creating & joining of new threads every iteration, were replaced with mutex locks for improving the synchronization of the parallel thread, and that (Version-3) gave the best performance across all the threads.