Report

OVERVIEW

My submission includes a report, two source code files, and the makefile, i.e.
Report.pdf
gauss_pthreads.C
gauss_pthreads_dyn.C
makefile
where *gauss_pthreads.C* refers my parallel Gaussian Elimination code with static task assignment approach. As a comparison, an alternative approach *gauss_pthreads_dyn.C,* refers a different approach with dynamic data assignment. After "make", two corresponding executable files *gauss* and *gauss_dyn* is generated.

STRATEGY

In *gauss_pthreads.C,* I use static task assignment approach. In the thread function, there is an iteration from 0 to matrix_size-1. The elimination task for corresponding row is conducted per cycle. In each term, firstly lock all threads but leave one to do partial pivoting, and also normalization by multiplying a row by a nonzero scalar. After that, applying barrier function to do synchronization. The barrier function implements synchronization by counting incoming threads, broadcast and unlock them when counter = number of threads. Then, each thread will take charge of elimination of one of every n rows (n is number of threads) by adding to it a scalar multiple of another each term, until reaching the last row. Finally synchronize all the threads again and go to next iteration cycle.

For *gauss_pthreads_dyn.C,* the difference lies in the process of row elimination. Instead of statically allocating every n rows to n threads every term, I set a global counter to mark the "next" row to be handled. The global counter is protected by pthreads_mutex, thread that preform its task can grab the counter as its index and increase it by 1. In that way, it can guarantee good load balancing in a great extent. By the way, synchronization is also done by barrier function.

TESTING RESULT and EXPLANATION

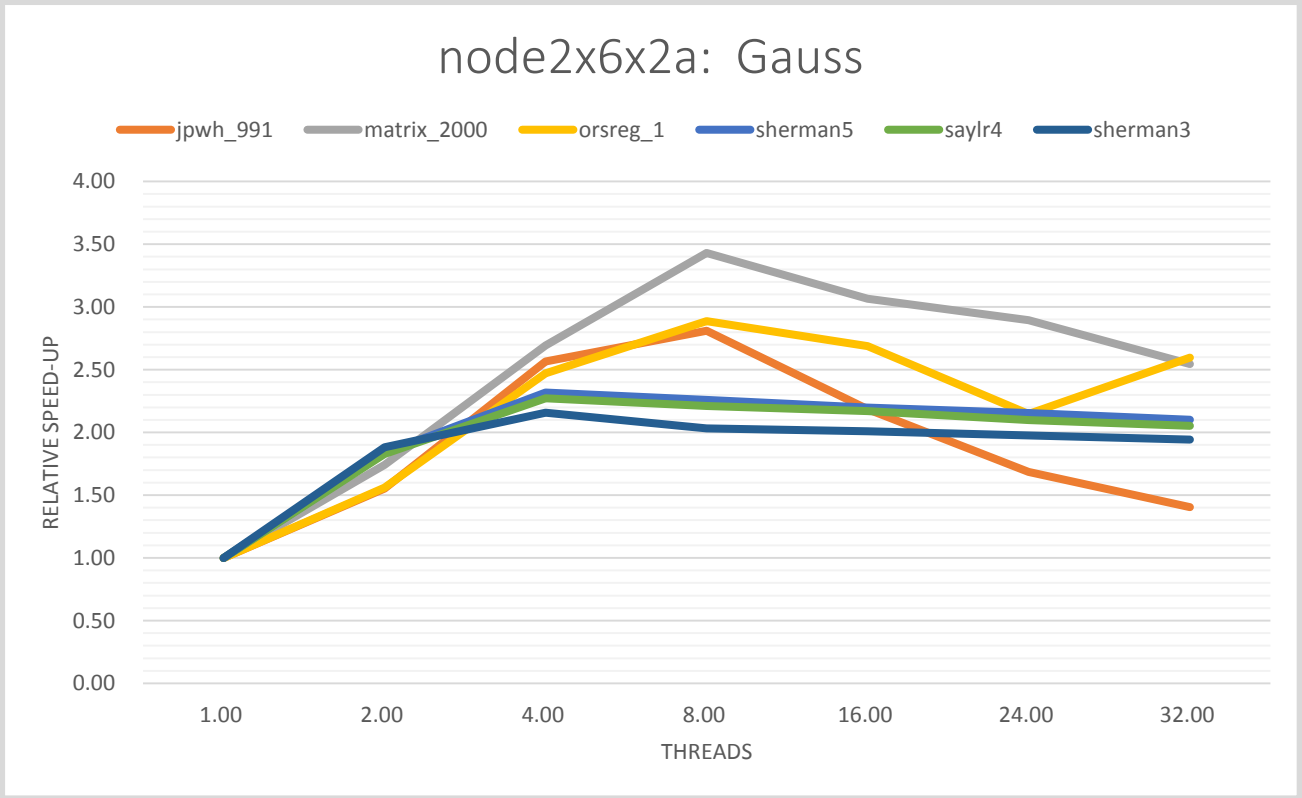
Program testing is done on two multiprocessor: *node2x6x2a* and *node2x4x2a*. Also, the testing of *gauss_dyn* is done on *node2x6x2a*. Testing cases include *jpwh_991, matrix_2000, orsreg_1, sherman5, saylr4, Sherman3,* with successively larger matrix size.

(1) Testing result of *gauss* on *node2x6x2a*

Threads	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	0.59	4.63	6.18	24.83	30.83	84.66
2	0.38	2.66	3.96	13.52	16.87	44.98
4	0.23	1.72	2.50	10.71	13.57	39.25
8	0.21	1.35	2.14	10.99	13.94	41.66
16	0.27	1.51	2.30	11.30	14.21	42.13
24	0.35	1.60	2.88	11.52	14.70	42.84
32	0.42	1.82	2.38	11.82	15.02	43.58

Normalized values

Threads	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.55	1.74	1.56	1.84	1.83	1.88
4	2.57	2.69	2.47	2.32	2.27	2.16
8	2.81	3.43	2.89	2.26	2.21	2.03
16	2.19	3.07	2.69	2.20	2.17	2.01
24	1.69	2.89	2.15	2.16	2.10	1.98
32	1.40	2.54	2.60	2.10	2.05	1.94



In fact, even for the biggest matrix, one row of data is just several kilobytes, much smaller than the L1 cache size. Each thread can all hold the chunk of normalized row data in the cache. Thus, the more important thing is synchronization. For each cycle of outermost iteration, it needs two barrier functions. In barrier function, every time a thread entering, we need to lock one thread, and broadcasting invalidation signal and a cache coherence for global value “arrived” should be done once. Thus, generally as the matrix size goes bigger, work weight of synchronization counts more. But bigger size

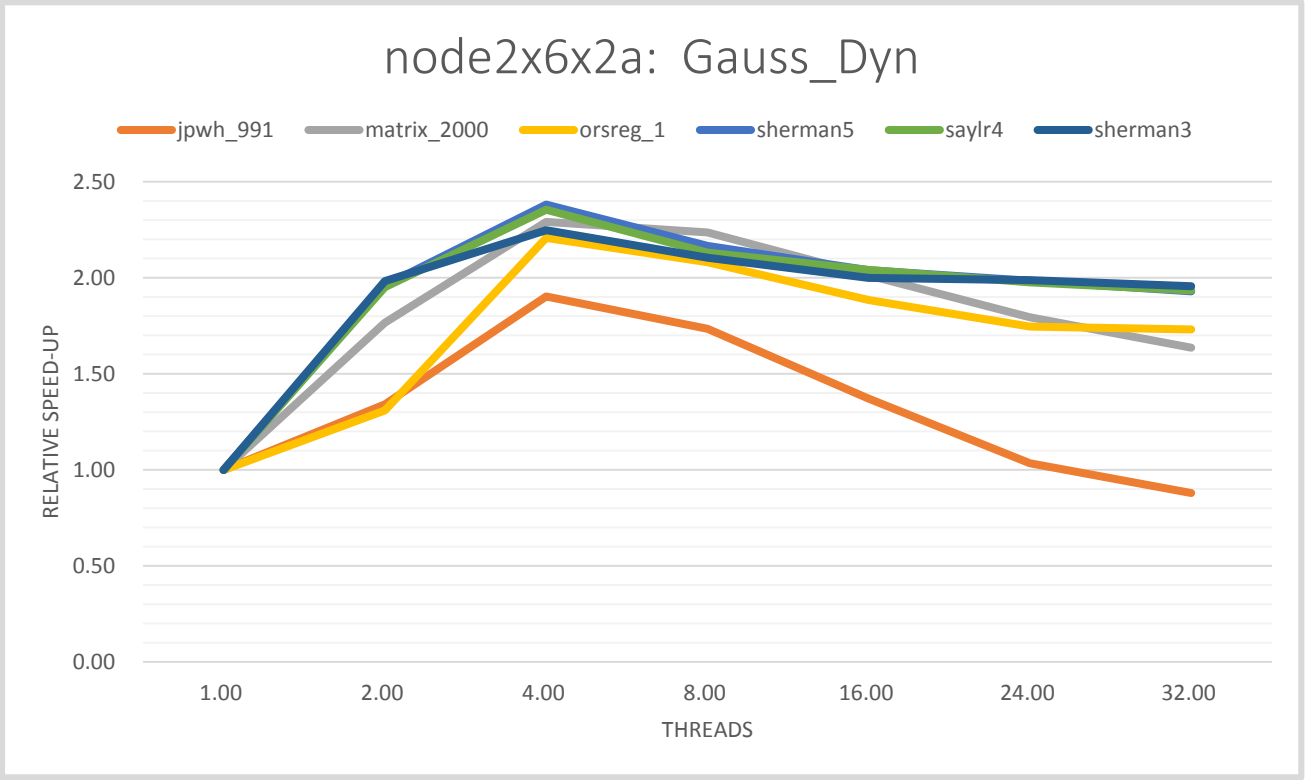
also indicates larger weight of parallel computation. The plot generally shows a matrix size of around 2000 seems a turning point for this tradeoff. Also, there exist another tradeoff of speed-up by parallelism and speed-down by synchronization with more threads. This is obviously not applicable by Amdhal's law, but also leads to unsatisfactory speed-up. Though we cannot quantified the actual amount of the two parameter, it is shown that as threads getting more than 8, speed-down by synchronization will overtake the speed-up by parallelism. Thus we can observe that more threads, even less than CPU threads (24), harms speed rather than helped it.

(2) Testing result of *gauss_dyn* on *node2x6x2a*

Threads	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	0.59	4.59	6.93	24.56	31.89	84.43
2	0.44	2.62	4.72	12.66	15.80	42.68
4	0.31	2.02	2.80	10.43	13.09	37.67
8	0.34	2.07	2.97	11.46	14.47	40.21
16	0.43	2.30	3.28	12.17	15.10	42.33
24	0.57	2.58	3.54	12.51	15.60	42.57
32	0.67	2.83	3.57	12.87	15.93	43.29

Normalized values

Threads	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.34	1.77	1.31	1.96	1.95	1.98
4	1.90	2.29	2.21	2.38	2.36	2.25
8	1.74	2.24	2.08	2.17	2.13	2.11
16	1.37	2.01	1.88	2.04	2.04	2.00
24	1.04	1.79	1.75	1.98	1.98	1.99
32	0.88	1.64	1.73	1.93	1.94	1.96



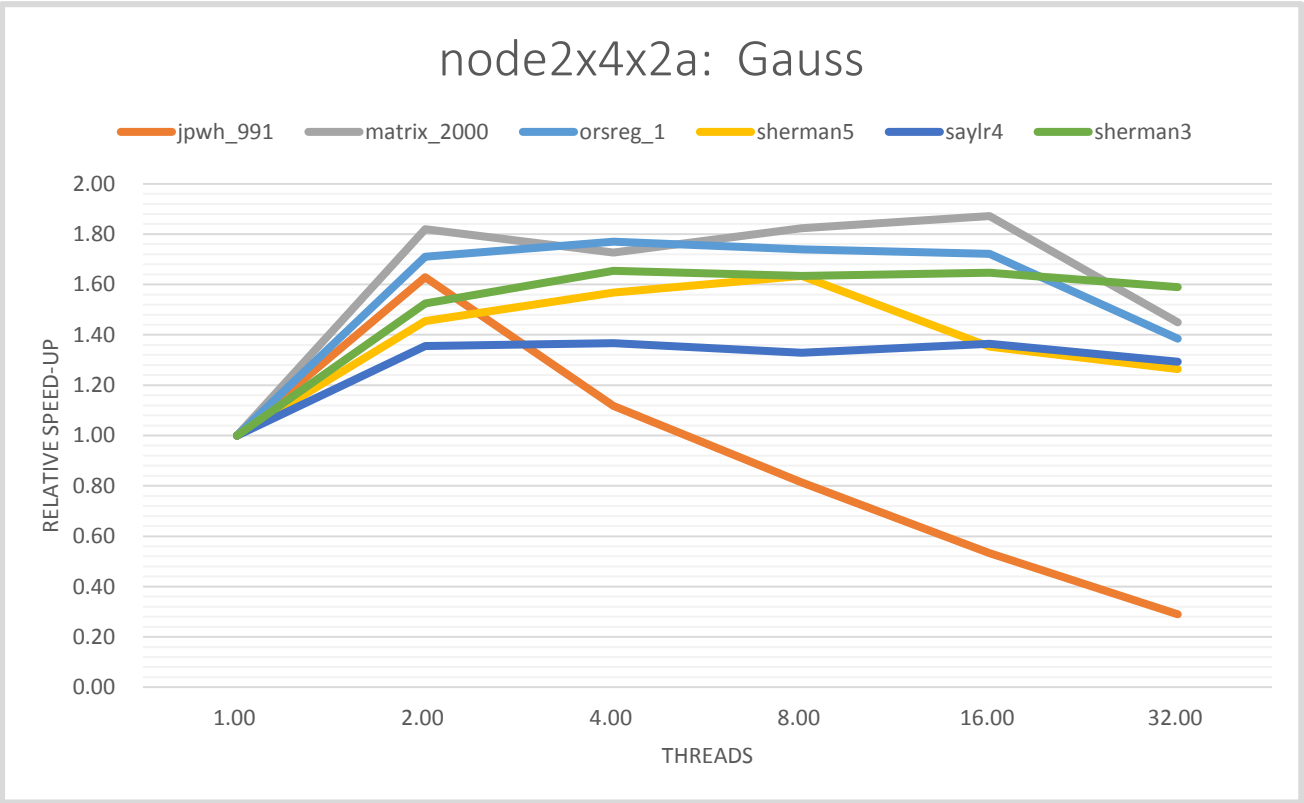
Comparing to the performance of the static version, the plot of the dynamic version has a very significant feature, i.e. trend of speed-up drops even earlier. The reason is very simple, besides synchronization of barrier function, every time a thread grabbing a row, we need to lock mutex, broadcasting invalidation signal and doing a cache coherence for global value “counter”. In that way, victim of more threads is more significant than that of static version. So we find that just after 4 threads, all the testcases already shows a speed-down trend.

(3) Testing result of *gauss* on *node2x4x2a*

Threads	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	0.57	8.48	11.17	38.63	50.16	140.35
2	0.35	4.66	6.53	26.56	37.00	92.04
4	0.51	4.91	6.31	24.63	36.68	84.81
8	0.70	4.65	6.42	23.64	37.75	85.92
16	1.07	4.53	6.49	28.55	36.76	85.22
32	1.97	5.85	8.07	30.56	38.80	88.27

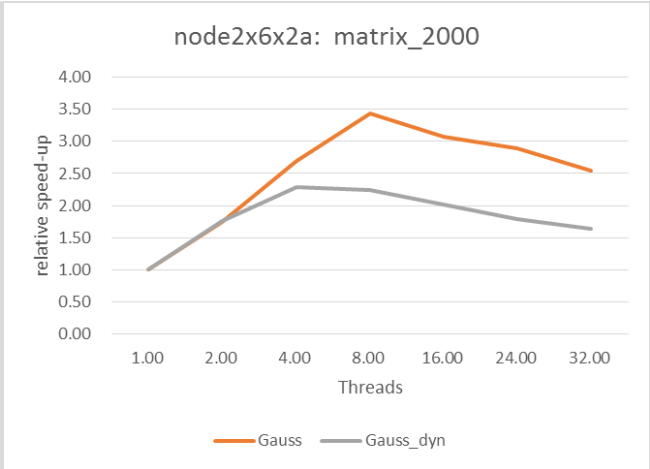
Normalized values

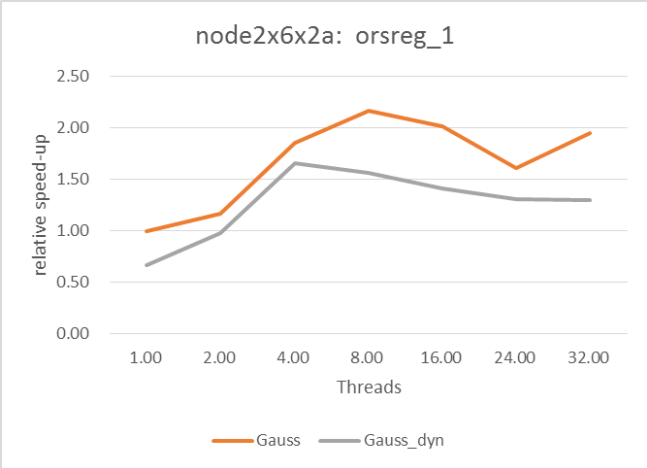
Threads	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.63	1.82	1.71	1.45	1.36	1.52
4	1.12	1.73	1.77	1.57	1.37	1.65
8	0.81	1.82	1.74	1.63	1.33	1.63
16	0.53	1.87	1.72	1.35	1.36	1.65
32	0.29	1.45	1.38	1.26	1.29	1.59

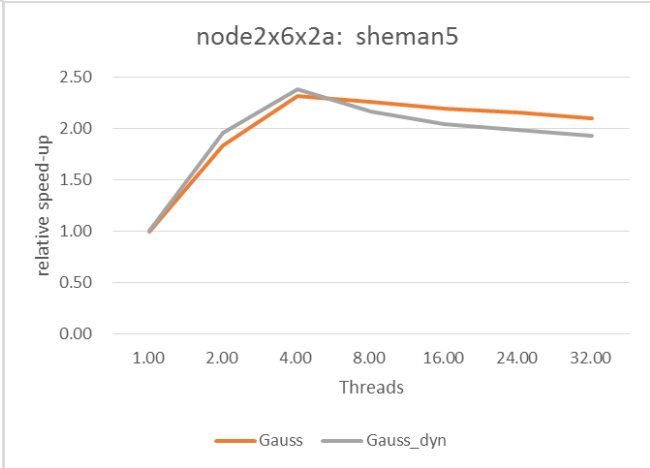


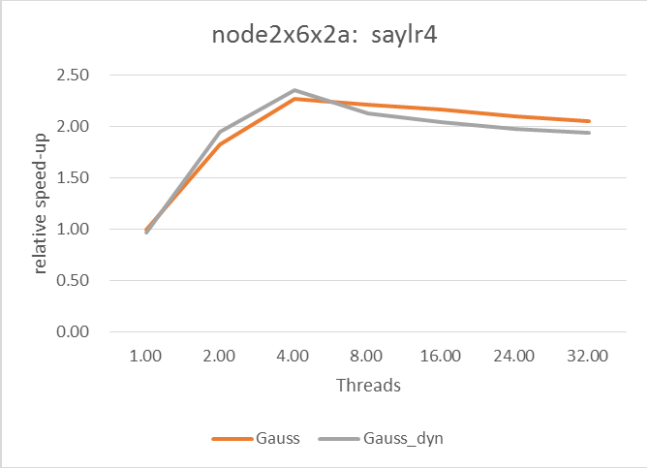
This server has only 16 CPU threads, and the computation ability is also weaker. So speed-down by synchronization overtakes the speed-up by parallelism after as early as 2 threads. An interesting thing happened on the smallest testcase jpwh_991. It shows a horrible speed-down after 2 threads. A reasonable speculation is that speed-up for small matrix on this slower CPU is less comparable to synchronization victim for large number of threads.

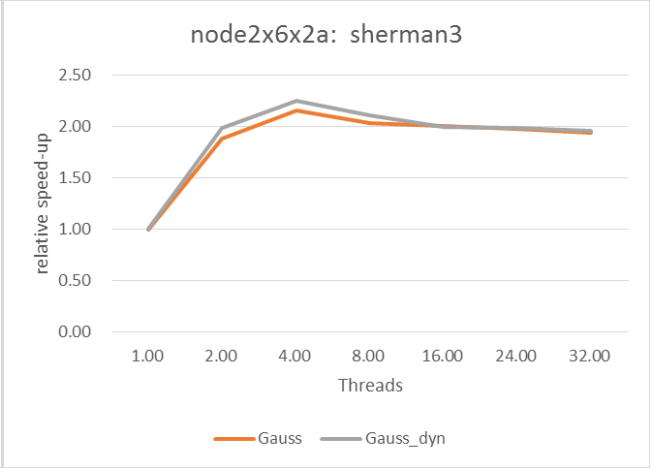
(4) Comparison of gauss and gauss_dyn on the same machine





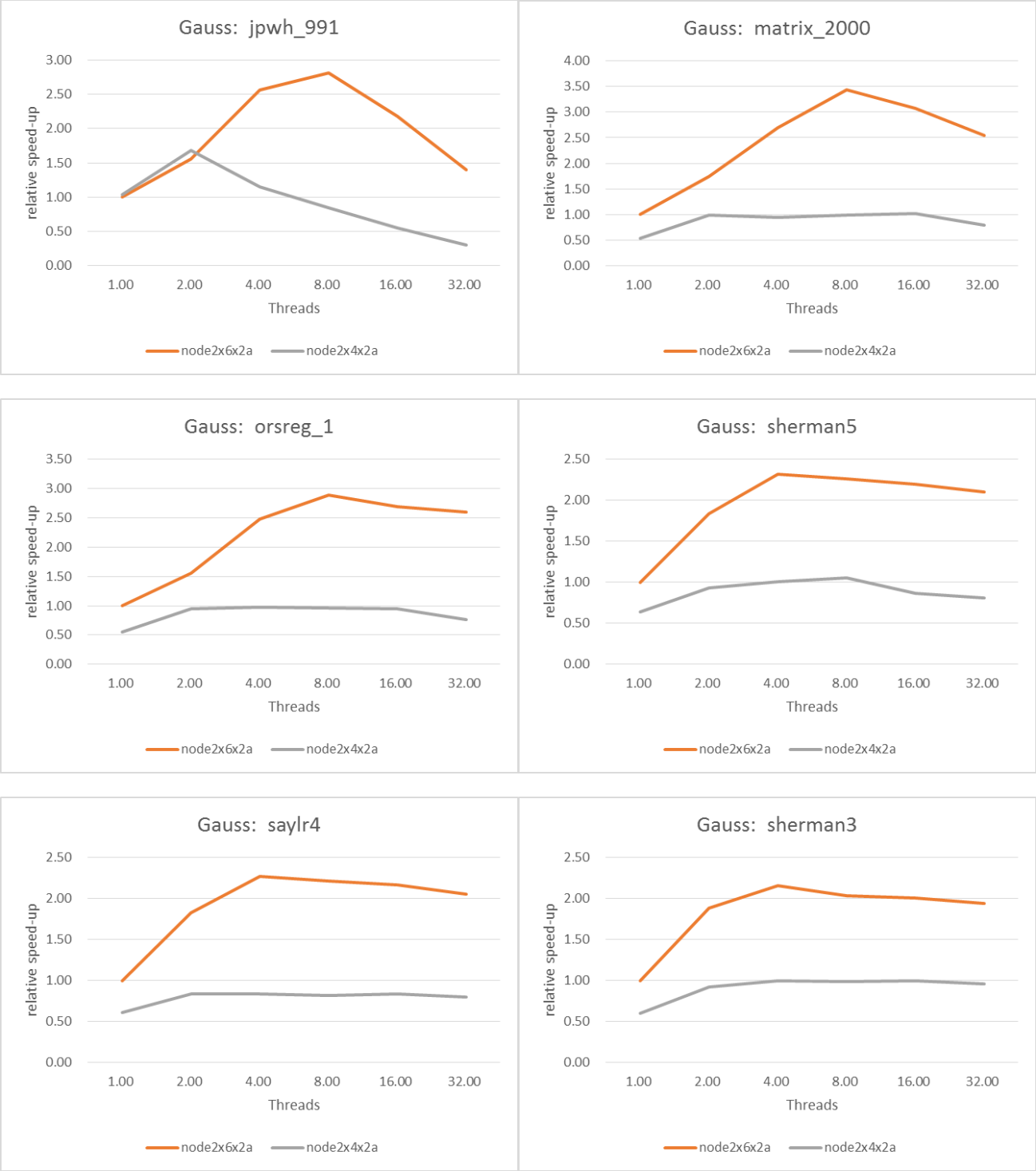






Although dynamic data assignment brings with better load balancing, this benefit is obviously concealed by more lock victim and cache coherence work, especially for small matrix. However, as matrix getting larger, static data assignment will naturally lead to huge workload gap between different threads. The benefit of dynamic assigning came out now. We can affirm that if the matrix getting even larger that *sherman3*, dynamic version will show better speed-up that static version.

(5) Comparison of gauss and gauss_dyn on different machine



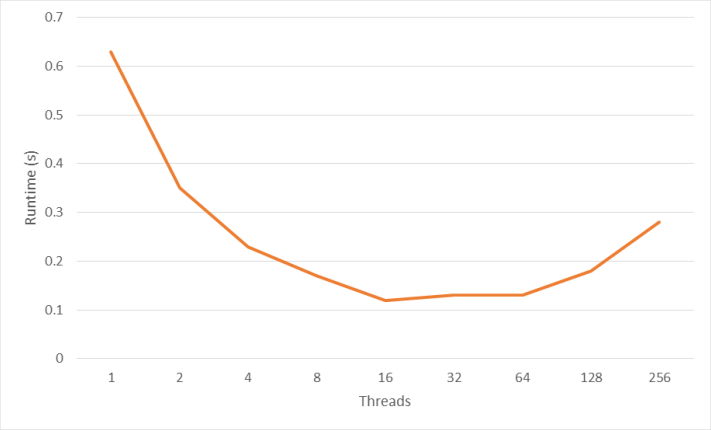
Viewing the data for single thread, it's shown that single-core performance of the second server is just half of the first. To some degree, this explains the lower speed-up of *node2x4x2a* to some degree. But in fact, as matrix size getting larger, though the first server still denotes fast speed, the speed-up of the two server are both nearly 2. It seems for huge data of this program, the strategy/ algorithm restraint the speed-up of the whole program with larger and larger amount of synchronization work. In this case, simply increase physical threads of CPU won't work.

Attachment: **Pre-Assignment**

(1) Machine: cycle2.cs.rochester.edu with Intel(R) Xeon(TM) CPU 3.00GHz
6 cores and 24 physical threads

It's

Threads	Runtime(s)
1	0.63
2	0.35
4	0.23
8	0.17
16	0.12
32	0.13
64	0.13
128	0.18
256	0.28



shown that the speed of execution increases as the number of threads increases until it reaches the number of CPU physical threads. After that, more threads, on the contrary, would lead to slower speed.

(2) Program description
How data is partitioned and assigned to multiple tasks in the parallel version of the program?
Take threads=4 as an example;
As the command "./sor –p4" indicates the situation of 4 threads;
Variable *task_num* will be signed as 4;

After that, the program use *pthread_create()* 4 times to create 4 threads; for each of them, use

*begin = (M * task_id) / task_num + 1;*

*end = (M * (task_id + 1)) / task_num;*

to define the start/end point of the data chunk that is allocated for each thread. For example, as calculated, the entire data (4000 elements) would be partitioned into four chunks and thread 0 would be allocated the data chunk from 1 to 1000, thread 1 gets 1001 to 2000, and so on. When all the works are done, the function *pthread_cond_broadcast()* is actived and the 4 threads begin to work simultaneously.