Report

OVERVIEW

My submission includes a report, three source code files, and the makefile, i.e.
Report.pdf
gauss_mpi.C
gauss_mpiA.C
gauss_mpiB.C
makefile
where *gauss_mpi.C* refers my parallel Gaussian Elimination code with cyclic row-based partitioning. As a comparison, an alternative approach *gauss_mpiB.C,* refers the code with overall block-based partition. Besides, there is another *gauss_mpiA.C,* which is a quantity-optimized version of *gauss_mpi.C.* After "make", three corresponding executable files *gauss_mpi*, *gauss_mpiA* and *gauss_mpiA* are generated.

STRATEGY

In *gauss_mpi.C,* I use cyclic row-based partitioning approach. Specifically, there is an iteration from row 0 to row matrix_size-1 for gauss computing. The elimination task for corresponding row is conducted per term. In each term, firstly get the pivot row and broadcast it, then create an iteration. For each cycle, the program is going to send n rows to n processes, where n denotes the number of processes, and then gathering results from them respectively. This operation will terminate when reaching the bottom of the matrix.

Here is a more detailed breakdown of the procedure of each iteration:

. FP - Master process: finding pivot row, swap it and broad cast the pivot row

. SN - Master process: sending n-1 rows to the other n-1 processes

. GE - Slave process: receiving the row, do elimination and send it back; Master process: do elimination for one row

. RN - Mater process: receiving n-1 rows of data for the other n-1 processes and do back substitution

And the totol time is $\mathrm{N} * (T_{FP} + T_{SN} + T_{GE} + T_{RN})$

For *gauss_mpiB.C,* the difference is after broadcasting the pivot row, it's going to divide the whole data that needs to be computed into n blocks for the n processes. In that way the inner iteration is needless.

For the above two cases, the transferred data is the same but Gauss_mpiB requires less communication message.

In fact, as shown on the graph below, when we need to handle the data below i_{th} row, for each row we only need to consider row[j] where $\mathrm{j} \geq \mathrm{i}$. Thus, we can only transfer the data on the right side of i_{th} column. In general, after this optimization, the total amount of data transferred is $\frac{n!}{n^2}$, nearly half

of before. However, this strategy is only accessible for cyclic row-based partitioning, because a block of data is continuous over row scale.

$$A = \begin{bmatrix} 2 & -3 & 4 & 2 \\ 6 & -9 & 12 & 5 \\ 4 & -5 & 10 & 5 \\ 2 & 2 & 11 & 9 \end{bmatrix} \xrightarrow{E_{41}(-1)E_{31}(-2)E_{21}(-3)} \begin{bmatrix} 2 & -3 & 4 & 2 \\ \underline{0} & 0 & 0 & -1 \\ \underline{0} & 1 & 2 & 1 \\ \underline{0} & 5 & 7 & 7 \end{bmatrix}$$

$$\xrightarrow{P_{23}} \begin{bmatrix} 2 & -3 & 4 & 2 \\ \underline{0} & 1 & 2 & 1 \\ \underline{0} & 0 & 0 & -1 \\ \underline{0} & 5 & 7 & 7 \end{bmatrix} \xrightarrow{E_{42}(-5)E_{32}(0)} \begin{bmatrix} 2 & -3 & 4 & 2 \\ \underline{0} & 1 & 2 & 1 \\ \underline{0} & \underline{0} & 0 & -1 \\ \underline{0} & \underline{0} & -3 & 2 \end{bmatrix}$$

$$\xrightarrow{P_{34}} \begin{bmatrix} 2 & -3 & 4 & 2 \\ \underline{0} & 1 & 2 & 1 \\ \underline{0} & \underline{0} & -3 & 2 \\ \underline{0} & \underline{0} & 0 & -1 \end{bmatrix} \xrightarrow{E_{43}(0)} \begin{bmatrix} 2 & -3 & 4 & 2 \\ \underline{0} & 1 & 2 & 1 \\ \underline{0} & \underline{0} & -3 & 2 \\ \underline{0} & \underline{0} & \underline{0} & -1 \end{bmatrix} = U,$$

from http://ccjou.wordpress.com/2012/04/13/palu-/

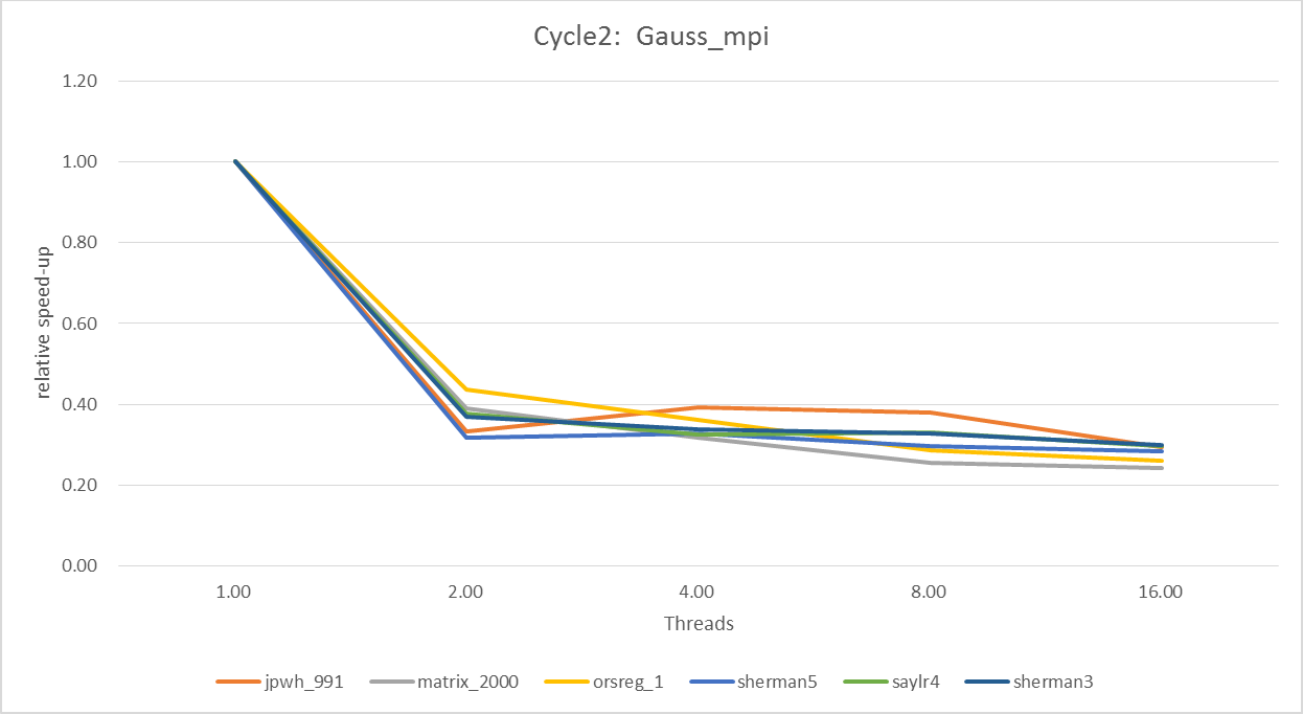
TESTING RESULT and EXPLANATION

Program testing is done on two platform: *cycle2* and *node2x12x1a*. Testing cases include *jpwh_991, matrix_2000, orsreg_1, sherman5, saylr4, Sherman3*.

(1) On cycle2

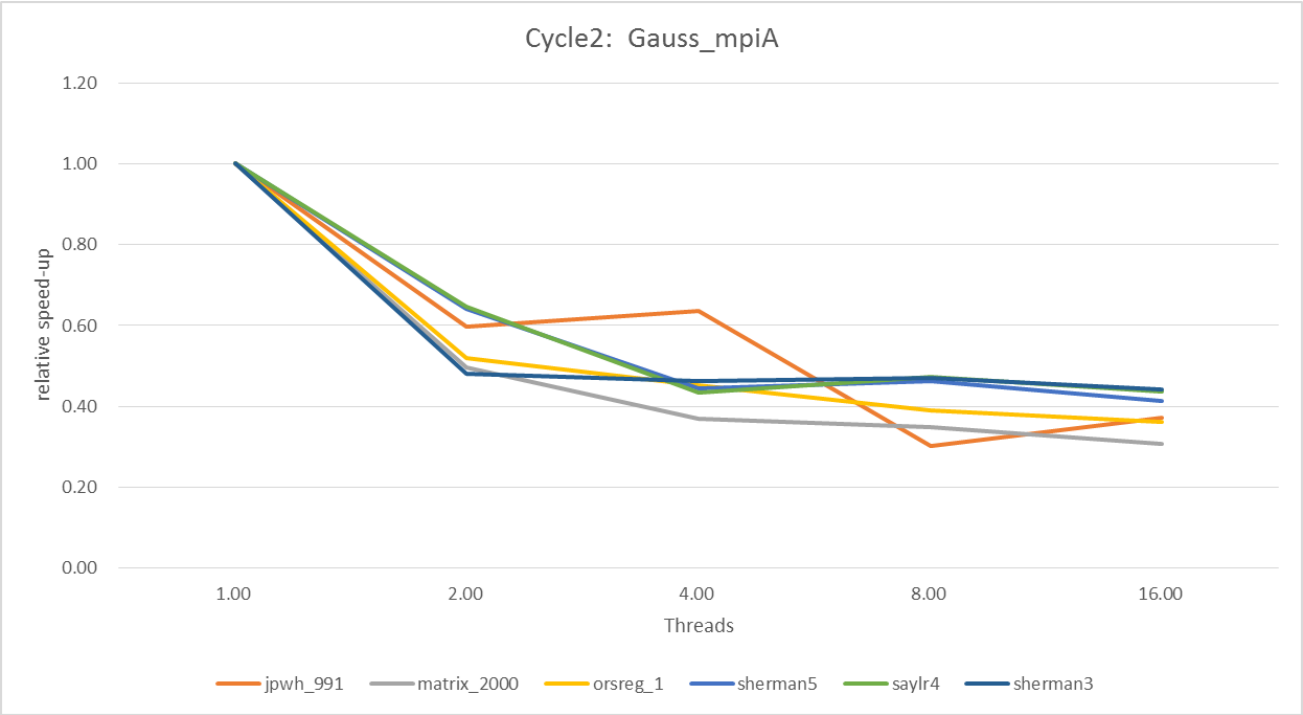
a) Gauss_mpi

processes	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.33	0.39	0.44	0.32	0.38	0.37
4	0.39	0.32	0.36	0.33	0.33	0.34
8	0.38	0.25	0.29	0.30	0.33	0.33
16	0.29	0.24	0.26	0.29	0.30	0.30



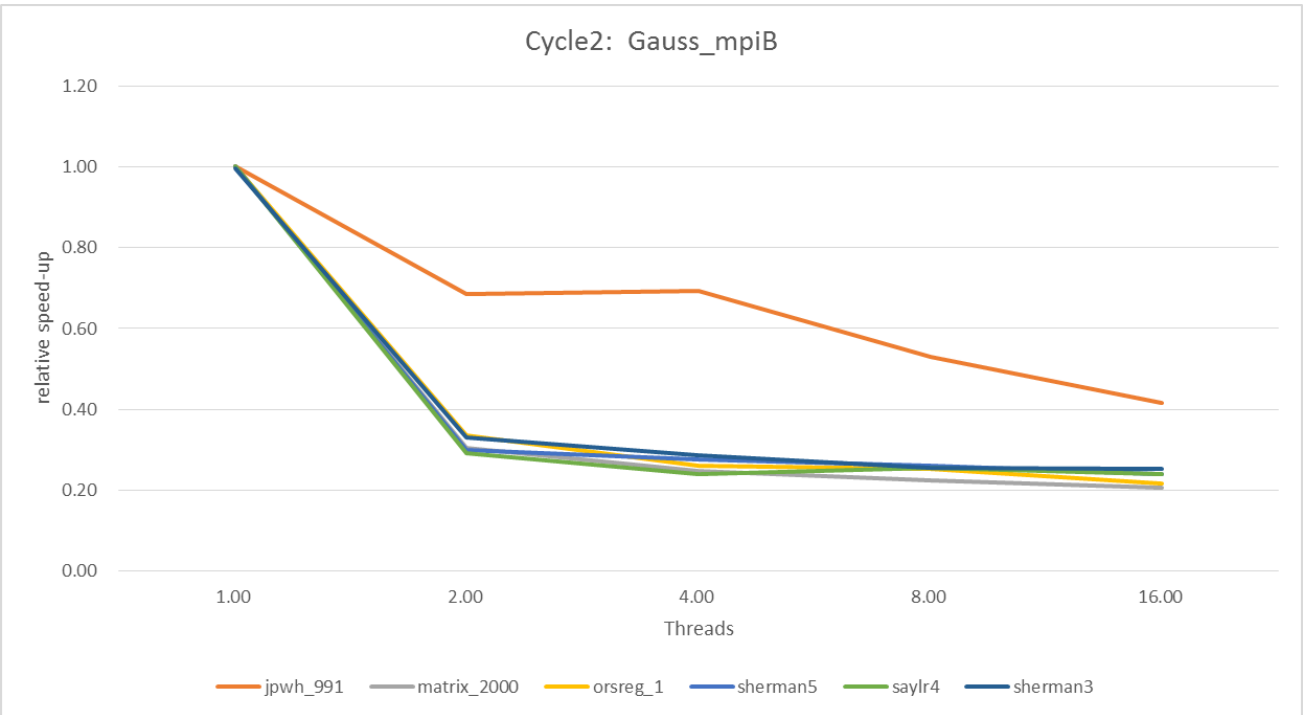
b) Gauss_mpiA

processes	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.60	0.50	0.52	0.64	0.65	0.48
4	0.64	0.37	0.45	0.45	0.43	0.46
8	0.30	0.35	0.39	0.46	0.47	0.47
16	0.37	0.31	0.36	0.41	0.44	0.44



c) Gauss_mpiB

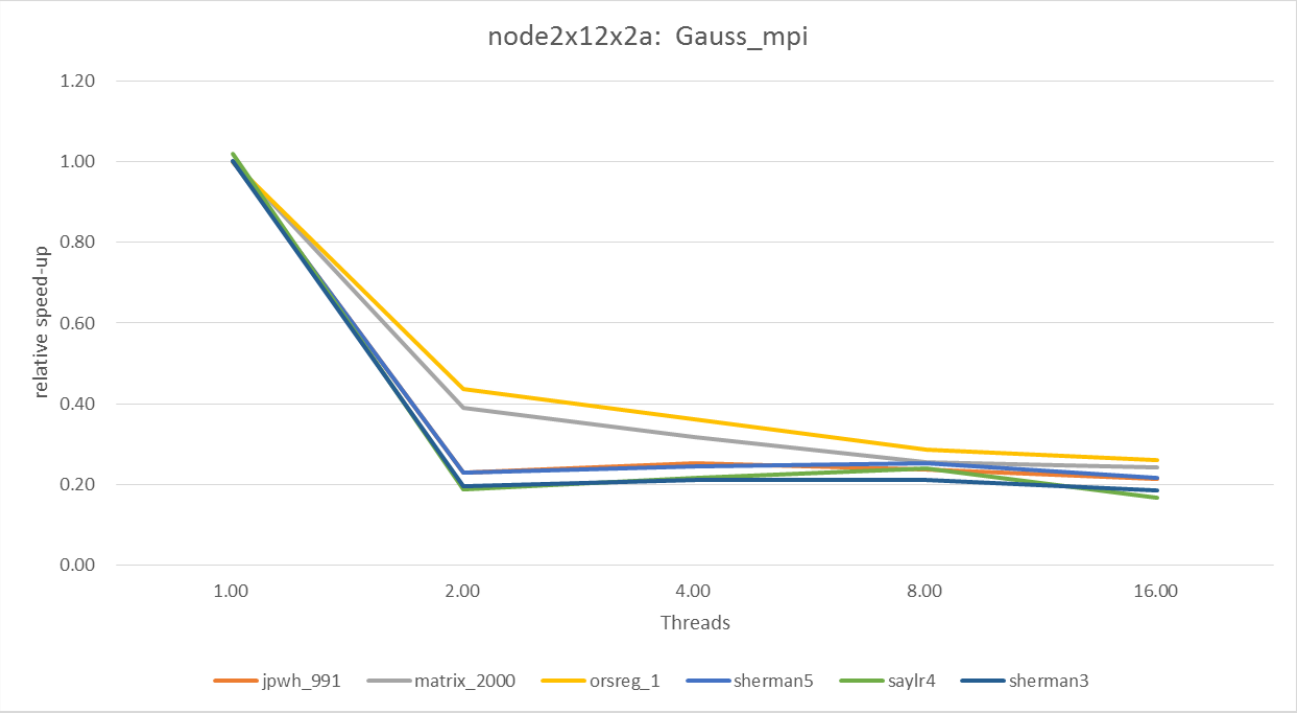
processes	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.68	0.31	0.33	0.30	0.29	0.33
4	0.69	0.25	0.26	0.28	0.24	0.29
8	0.53	0.23	0.25	0.26	0.26	0.26
16	0.42	0.21	0.22	0.24	0.24	0.25



(2) On node2x12x2a

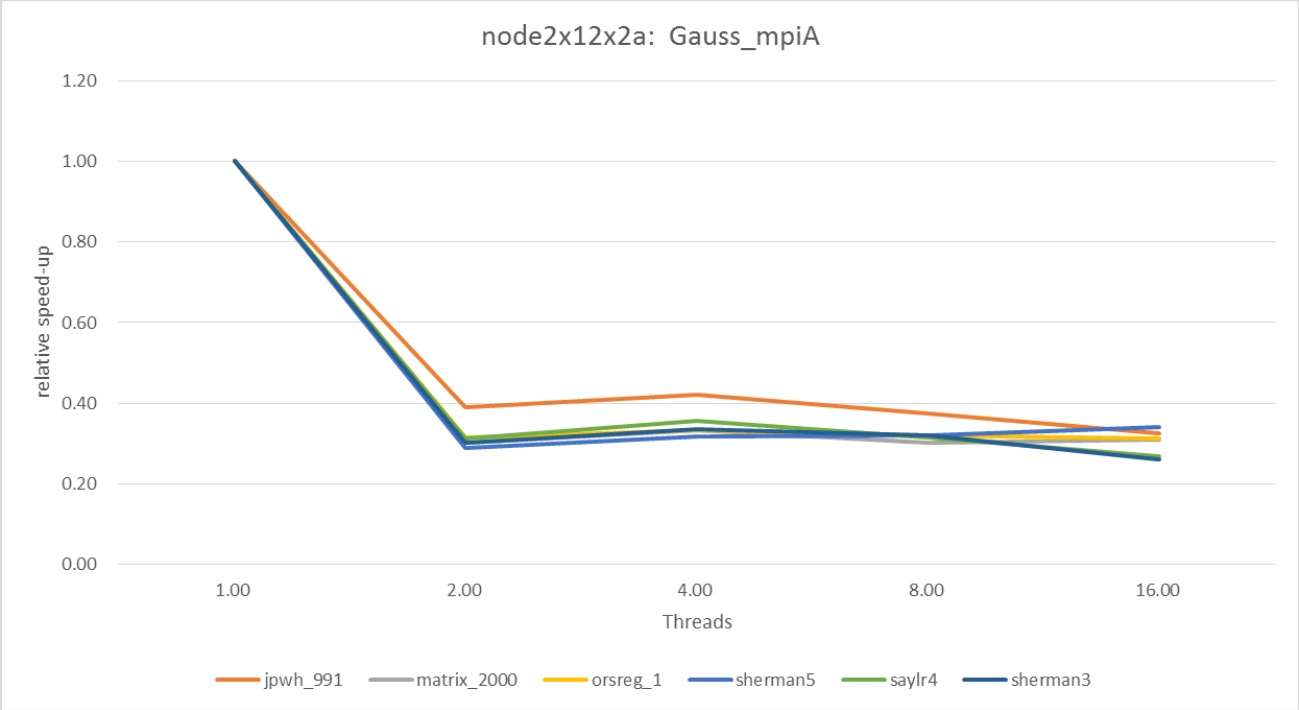
a) Gauss_mpi

processes	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	1.00	1.00	1.00	1.00	1.02	1.00
2	0.23	0.39	0.44	0.23	0.19	0.20
4	0.25	0.32	0.36	0.25	0.22	0.21
8	0.24	0.25	0.29	0.25	0.24	0.21
16	0.21	0.24	0.26	0.22	0.17	0.19



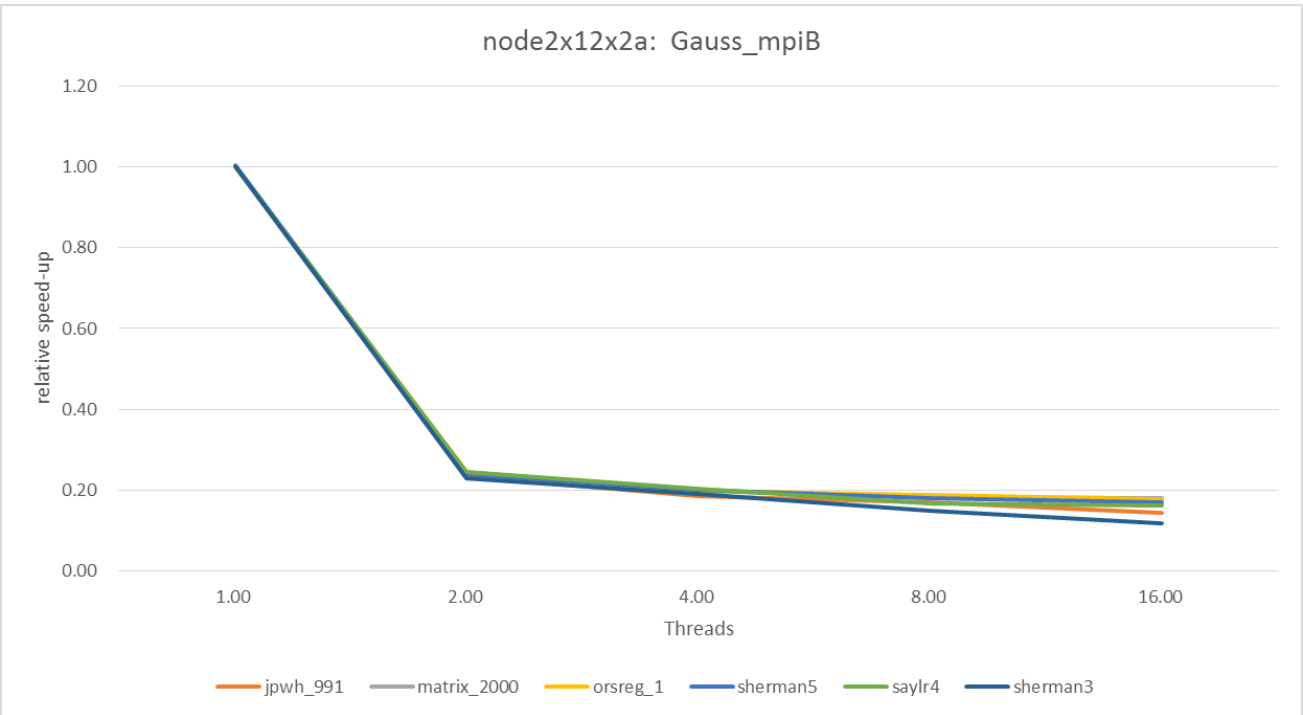
b) Gauss_mpiA

processes	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.39	0.30	0.31	0.29	0.31	0.30
4	0.42	0.33	0.33	0.32	0.36	0.33
8	0.37	0.30	0.32	0.32	0.32	0.32
16	0.32	0.31	0.31	0.34	0.27	0.26



c) Gauss_mpiB

processes	jpwh_991	matrix_2000	orsreg_1	sherman5	saylr4	Sherman3
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.23	0.24	0.25	0.24	0.25	0.23
4	0.19	0.20	0.20	0.20	0.20	0.19
8	0.17	0.18	0.19	0.18	0.17	0.15
16	0.14	0.18	0.18	0.17	0.16	0.12

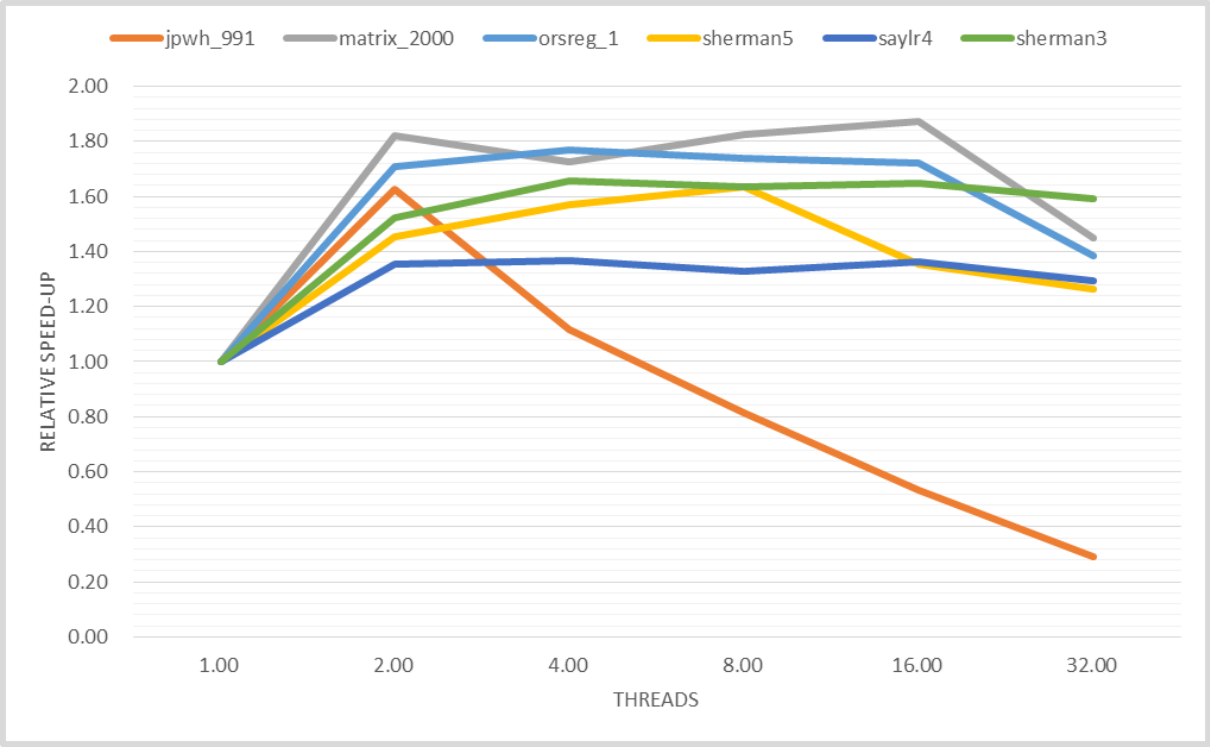


Very unfortunately, adding support of MPI is far from increasing the speed; on the contrary, the speed reduces a lot. That means, the communication costs is much more than the speed-up brought by task decomposed. And we can see that for our program, the most important factor that influence the speed-up is the communication speed rather than CPU performance. Because the speeds of single process running on node2x12x1a and cycle2 have no much difference, while the speed-up on cycle2 is much better that that on node2x12x1a, most likely the communication speed on node 2x12x1a is better.

Mathematically speaking, that means the $(\mathrm{N}-1)*T_{GE} < T_{FP}+T_{SN}+T_{RN}$, see my definition on page 1.

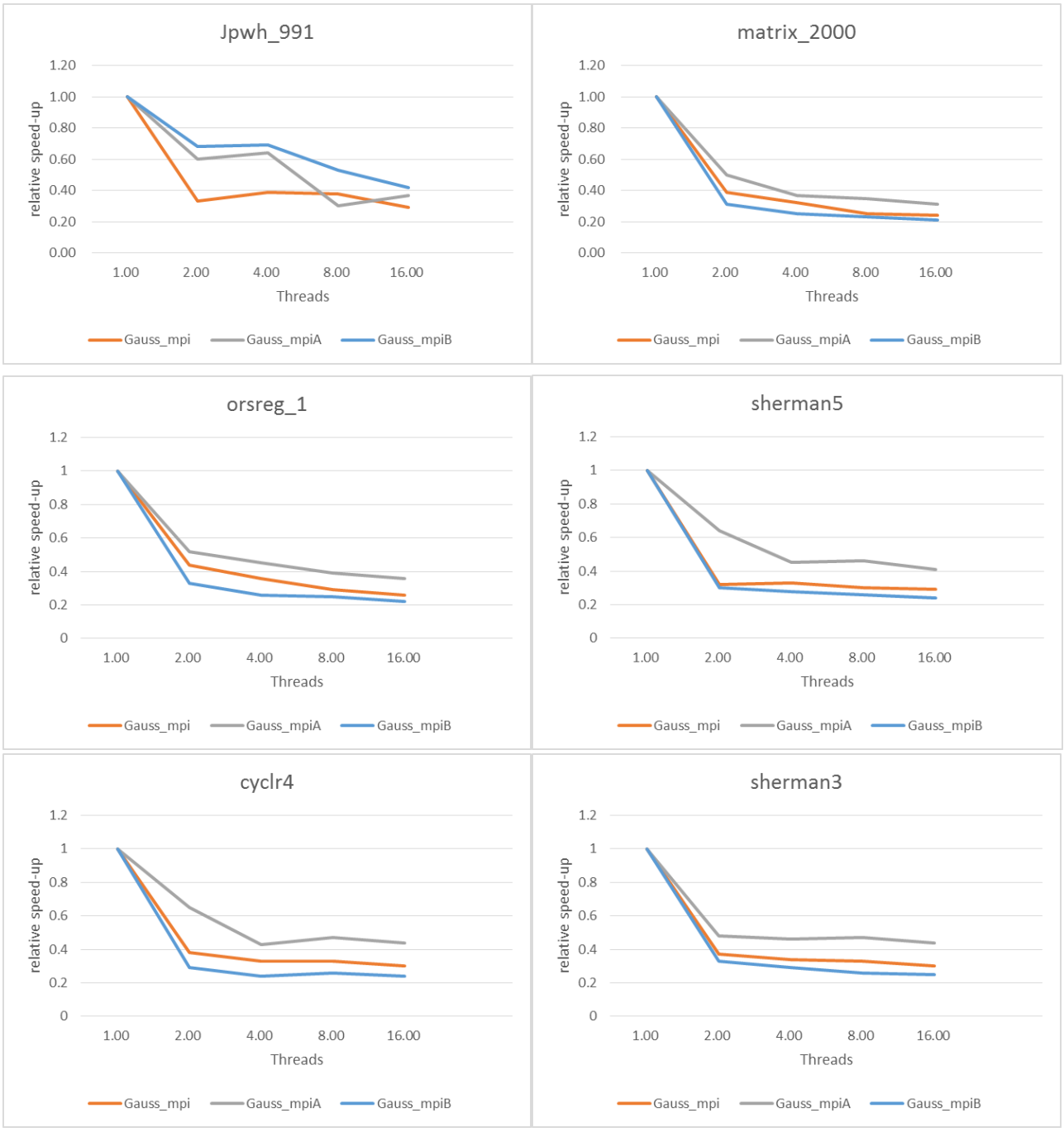
COMPARISON

(1) Comparison of MPI and pthread



A typical speed-up graph I got last time is shown above. Obviously, when running a single machine, pthreads will provide much better speed-up than MPI, as MPI even doesn't provide any speed-up. The reason is communication and data transmission are very high cost operation, which demands much more time than cache coherence. Take sherman3 as an example, when running on 4 MPI processes, each time the master process will send about 555KB and receive such much, which would naturally costs a lot. Especially for gauss elimination, it needs to transfer i*n data for n times, an amount of data much more than the matrix itself. In fact, I've written a MPI based matrix multiplication, for which the matrix only need to be transfer once for each process, and it have a comparable performance to pthread version.

(2) Block vs. cyclic partitioning
.on cyle2



To compare the performance of block-based partitioning and cyclic row-based partitioning, we can plot the speed-up trend for the three approaches.

Theoretically, block-based partitioning requires less communication message with the same communication quantity. But considering the cache of a processor, 128KB L1, which is about 1 rows even for the largest matrix sherman3 and 17 rows for jpwh_991. When use the cyclic row-based partitioning, for many cases each iteration can be done with total cache hit or high hit rate. But block-based approach won't benefit from it, which means cyclic row-based partitioning help reduce the

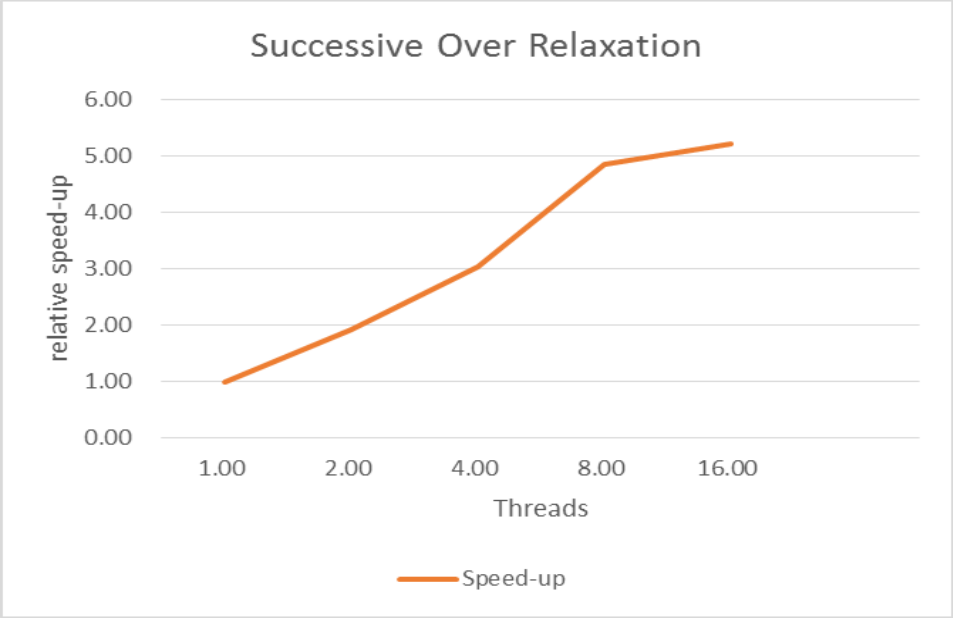
amount of idling because for many cases all processes can get and substitute tasks on cache.

What is more important is that, cyclic row-based partitioning give us an opportunity to reduce the communication quantity. As I have indicated, when we send a row to a process at iteration i, the first i-1 elements are in fact 0 and we don’t need them. So we can just deal with the data we really need, which is about half of the total. Thus, we can observe that optimized version Gauss_mpiA has the best performance over others.

Attachment: **Successive Over Relaxation**

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

Processes	Speed-up
1	1.00
2	1.92
4	3.04
8	4.87
16	5.21



It is shown that the speed of execution increases as the number of MPI processes increase.

(2) Program description and analysis
The program is going to at first divide the total task M into n partitions and each process dealing with such amount of data for 100 iterations. in each iteration, the cell is updated by averaging the north, south, east, west immediate neighbors. Then the process will send its result to the former and latter process, except for the head (0) and tail (n-1), as they don't have former or latter neighbor. And then they will get results from its former and latter neighbors. Obviously, the more the processors, each processor will get less partition of data. Thus we can see a nice speed-up. However, the communication message for each processors stays the same so the communication costs doesn't rigorously reduce as much as the reduction of data amount. Thus, according to Amdahl's law, the speed-up is less than the parallel level.