REPORT

1. Sol
   Attachment: *tso.C* *tso_test.C* *pso.C* *pso_test.C*

The files *tso.C* and *pso.C* are respectively to collect abnormal cases that indicates W-R reordering and W-W reordering upon 1 million times of testing. And there are also two test programs to count the number of abnormal cases. Specifically, *tso.C* is going to run

P1	P2
A = 1;	B = 1;
print B;	print A;

1 million times. And we finally collect 38 abnormal cases;
pso.C is going to run

P1	P2
flag = 0;	
A = 1;	while (flag == 0);
flag = 1;	print A

1 million times. But there is no abnormal cases found.
Generally, the machine's underlying consistency model is likely to be TSO.

2. Sol
   Attachment: *MatMul.C* *MatMul2.C*

MatMul.C is the original code which refers to distributing the workload by blocks of lines, and *MatMul2.C* is the new code which refers to distributing the workload by elements. The testing work is done on *node1x12x1a*.

Threads	Dist by Lines		Dist by elements	
1	1001.31s	1.00x	1006.21s	1.00x
2	479.48s	2.09x	558.18s	1.80x
4	242.65s	4.13x	357.83s	2.81x
8	129.68s	7.72x	193.19s	5.21x
16	95.94s	10.44x	130.81s	7.69x
24	87.09s	11.50x	127.32s	7.90x

It's obvious that the second approach is much slower than the first one, especially when threads gets more and more. Because distributing by elements indicates that several processes operate on independent data in the same memory address region storable in a single line. The cache coherency mechanisms in the system may force the whole line across the bus or interconnect with every data write, forcing memory stalls in addition to wasting system bandwidth. Thus, more threads working on one line will lead to more wasting on coherence work. And the program is going to be slower.

3. Sol
   Attachment: *MatMul_file.C MatMul_mpi.C hosts*

Because of the poor performance of stdin redirection in MPI compiling, Matrix reading is done by directly reading file instead of stdin redirection. To excluding this changing's effects, I also change the original file *MatMul.C* into *MatMul_file.C,* which reads matrix by reading file; thus, to run the program, you need to type

time mpirun -np n -machinefile hosts ./a.out mat2000 >out

I do the test on node18 and node24. But in fact they are very old machines with only two physical cores and very poor performance, and I fail to run MPI on node33 which is more powerful. So I have no choice but do the test upon 2000*2000 matrix instead of 4000*4000.

Threads/ processes	Pthread		MPI_single		MPI_dual	
1	194.54s	1.00x	186.22s	1.00x	184.36s	1.00x
2	110.25s	1.76x	126.16s	1.48x	101.52s	1.82x
4	164.96s	1.18x	256.15s	0.73x	244.42s	0.75x
8	-	-	-	-	233.94s	0.79x

When running pthread code, the poor machine loses speed-up as early as when it comes to 4 threads. We can get that when running on a single machine, MPI usually don't have advantage over pthread. The reason is for MPI, it needs to do send/receive work for every process. And the time wasting is generally much more than the time for pthread program to waste on threading work. But when running on two machines, MPI shows some advantage over pthread – it can get rid of the restriction of single machine's limited computing resources. Due to the limited mechanism of node18 and node24, I fail to further the performance by applying more processes. But we can

predict that if the matrix being bigger, which means bigger portion of computing work, and more powerful machine, it's possible to maximal the performance by applying more processes on several machines.

4. Sol
   Attachment: *MatMul_mpi_pth.C*

The program is run by typing

time mpirun -np n -machinefile hosts ./a.out m mat2000 >out

where m denotes the number of threads for each processes.

Very unfortunately, adding support of pthread is far from increasing the speed; on the contrary, the speed reduces a lot. I have read some papers and try to come up with two possible explanation.

(1) For my code, the case is that by applying pthread, there is no help for reducing the communication workload for MPI, as it is only related to the number of MPI processes. But at the same time, communication of MPI processes takes up memory bandwidth and makes it much more expensive for pthread's work like memory consistency. A persuasive evidence is that when I use "time" command to measure the testing result, find that the utilization ratio of CPU for a 2-processes 2-threads case is only 54.7%, while a pure 2-threads case can be more than 94%.

(2) Because the poor performance and limited physical core (only 2) of the CPU, the promotion brought by multithread is much less than the cost. So if we have a machine with more cores, we can look forward that the speed-up brought by multithread will be comparable to its cost, as the communication workload for MPI won't increase with more pthreads.

Threads	Processes=2		Processes=4	
1	104.15s	1.00x	172.01s	1.00x
2	172.62s	0.60x	203.91s	1.76x
4	176.51s	0.59x	191.23s	1.18x