	100	1000	10000
Sequential (OpenMP)	1	1	1
Sequential (MPI)	0.9322	0.9590	0.9386
MPI	1.9018 (4 processes)	8.6201 (16 processes)	13.2707 (16 processes)
OpenMP	2.1163 (4 threads)	13.4118 (16 threads)	14.8213 (32 threads)
MPI + OpenMP	1.1710 (2 processes, 2 threads)	10.6723 (4 processes, 4 threads)	18.5321 (8 processes, 32 threads)

Note: Cannot solve for problem size of 100000 due to the storage limit of 4GB on CIMS machine.

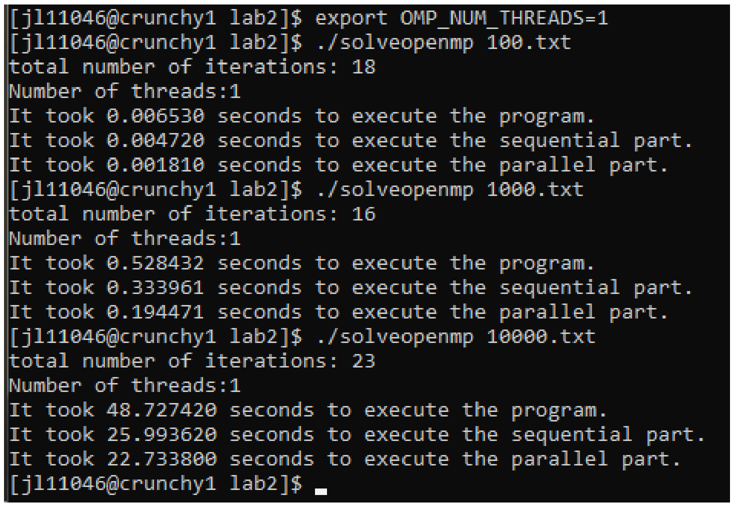
Note: Choices of number of threads are {2, 4, 8, 16, 32, 64}. Choices of number of processes are {2, 4, 8, 16, 32}.

Question 1: Are there differences between sequential OpenMP and sequential MPI? And why?

In terms of execution time, there are not much difference between sequential OpenMP and sequential MPI. Only that sequential MPI runs a little slower than sequential OpenMP, because the cost of creating processes and communications among processes are more expensive than those of threads.

Question 2: What is the best configuration for each problem size?

Screenshot showing the portion of sequential vs. parallel part in an OpenMP program:



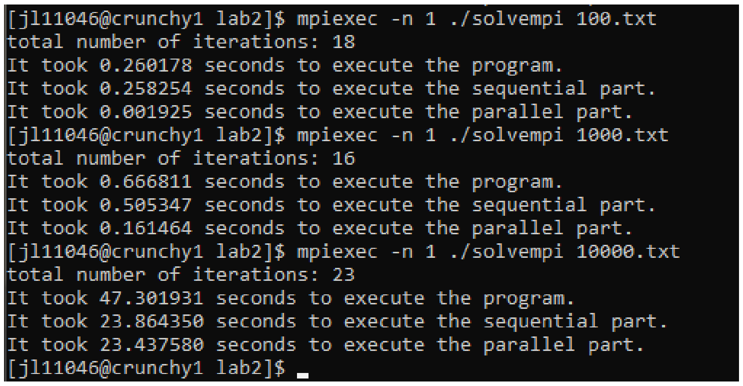
```
[jl11046@crunchy1 lab2]$ export OMP_NUM_THREADS=1
[jl11046@crunchy1 lab2]$ ./solveopenmp 100.txt
total number of iterations: 18
Number of threads:1
It took 0.006530 seconds to execute the program.
It took 0.004720 seconds to execute the sequential part.
It took 0.001810 seconds to execute the parallel part.
[jl11046@crunchy1 lab2]$ ./solveopenmp 1000.txt
total number of iterations: 16
Number of threads:1
It took 0.528432 seconds to execute the program.
It took 0.333961 seconds to execute the sequential part.
It took 0.194471 seconds to execute the parallel part.
[jl11046@crunchy1 lab2]$ ./solveopenmp 10000.txt
total number of iterations: 23
Number of threads:1
It took 48.727420 seconds to execute the program.
It took 25.993620 seconds to execute the sequential part.
It took 22.733800 seconds to execute the parallel part.
[jl11046@crunchy1 lab2]$
```

For problem size of 100, proportion of parallel part = 27.7%

For problem size of 1000, proportion of parallel part = 36.8%

For problem size of 10000, proportion of parallel part = 46.7%

Screenshot showing the proportion of sequential vs. parallel part in an MPI program:



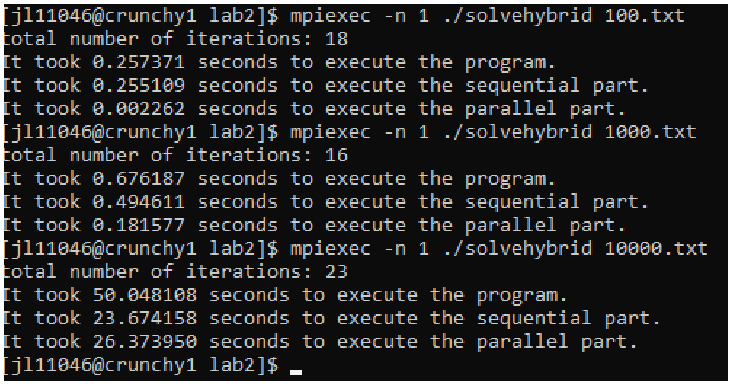
```
[jl11046@crunchy1 lab2]$ mpiexec -n 1 ./solvempi 100.txt
total number of iterations: 18
It took 0.260178 seconds to execute the program.
It took 0.258254 seconds to execute the sequential part.
It took 0.001925 seconds to execute the parallel part.
[jl11046@crunchy1 lab2]$ mpiexec -n 1 ./solvempi 1000.txt
total number of iterations: 16
It took 0.666811 seconds to execute the program.
It took 0.505347 seconds to execute the sequential part.
It took 0.161464 seconds to execute the parallel part.
[jl11046@crunchy1 lab2]$ mpiexec -n 1 ./solvempi 10000.txt
total number of iterations: 23
It took 47.301931 seconds to execute the program.
It took 23.864350 seconds to execute the sequential part.
It took 23.437580 seconds to execute the parallel part.
[jl11046@crunchy1 lab2]$ 
```

For problem size of 100, proportion of parallel part = 0.7%

For problem size of 1000, proportion of parallel part = 24.2%

For problem size of 10000, proportion of parallel part = 49.5%

Screenshot showing the proportion of sequential vs. parallel part in a hybrid program:



```
[jl11046@crunchy1 lab2]$ mpiexec -n 1 ./solvehybrid 100.txt
total number of iterations: 18
It took 0.257371 seconds to execute the program.
It took 0.255109 seconds to execute the sequential part.
It took 0.002262 seconds to execute the parallel part.
[jl11046@crunchy1 lab2]$ mpiexec -n 1 ./solvehybrid 1000.txt
total number of iterations: 16
It took 0.676187 seconds to execute the program.
It took 0.494611 seconds to execute the sequential part.
It took 0.181577 seconds to execute the parallel part.
[jl11046@crunchy1 lab2]$ mpiexec -n 1 ./solvehybrid 10000.txt
total number of iterations: 23
It took 50.048108 seconds to execute the program.
It took 23.674158 seconds to execute the sequential part.
It took 26.373950 seconds to execute the parallel part.
[jl11046@crunchy1 lab2]$ 
```

For problem size of 100, proportion of parallel part = 0.9%

For problem size of 1000, proportion of parallel part = 26.9%

For problem size of 10000, proportion of parallel part = 52.7%

The observation is that:

1. The proportion of the program that is parallelizable increases with the problem size. For a problem size as large as 10000, about half the program is parallelizable. This is because as problem size grows, our matrix calculations, which are parallelizable, grow faster than the sequential part of the program, such as handling input and output.
2. For a small problem size (such as 100), an OpenMP program will have more parallelizable portion than its MPI or MPI+OpenMP hybrid counterparts. This is because creating processes

and communications among processes in MPI are more expensive than creating threads in OpenMP.

The general suggestion is that for problem of a small size, use OpenMP with few threads, or simply use the sequential code. For problem of medium size, use OpenMP with a larger number of threads. For problem of large size, use MPI+OpenMP hybrid to achieve best performance, or choose either pure OpenMP or pure MPI to achieve better efficiency.

Specifically, for our problem:

For problem size of 100, the best configuration is to use OpenMP (4 threads). This configuration gives a speedup of around 2.1, with an efficiency of about 52.9%.

For problem size of 1000, the best configuration is still to use OpenMP (16 threads). This configuration gives a speedup of 13.4, with an efficiency of about 83.8%

For problem size of 10000, the best configuration is the hybrid scheme (8 processors each with 32 threads). This configuration gives a speedup of around 18.5, with a poor efficiency of 7.2%. To achieve best efficiency while keeping reasonable speedup, it is recommended to use MPI, which gives a comparable speedup of around 13.3, with an efficiency of 82.9%.

omp_get_wtime() is used in the OpenMP program for calculating the program execution time and the parallel portion execution time.

MPI_Wtime() is used in both the MPI program and the hybrid program for calculating the program execution time and the parallel portion execution time.

Data

	100	1000	10000
Sequential (OpenMP)	1	1	1
Sequential (MPI)	0.932194617	0.959044749	0.938638223
MPI	1.901795143	8.620146846	13.27070324
OpenMP	2.116333725	13.4117698	14.8212547
MPI + OpenMP	1.1710013	10.67233662	18.53207357

	100	1000	10000	
Sequential (OpenMP)	0.001801	0.154973	21.840512	
Sequential (MPI)	0.001932	0.161591	23.268296	
OpenMP(2)	0.001174	0.077284	12.133076	
OpenMP(4)	0.000851	0.040256	6.29723	
OpenMP(8)	0.001078	0.02099	3.082698	
OpenMP(16)	0.001858	0.011555	1.522371	
OpenMP(32)	0.004885	0.015527	1.473594	
OpenMP(64)	0.078116	0.075824	1.660622	
MPI(2)	0.001156	0.085157	11.860638	
MPI(4)	0.000947	0.045799	5.983577	
MPI(8)	0.000996	0.080642	3.086895	
MPI(16)	0.001977	0.017978	1.645769	
MPI(32)	0.031443	0.045495	1.723524	
MPI(64)	4.037704	6.210069		Too slow

100	OpenMP(2)	OpenMP(4)	OpenMP(8)	OpenMP(16)	OpenMP(32)	OpenMP(64)
MPI(2)	0.001538	0.002784	0.002817	0.008953	0.0937	0.175429
MPI(4)	0.002099	0.006787	0.003873	0.088333	0.183899	0.311301
MPI(8)	0.005419	0.005836	0.069444	0.182359	0.340023	0.556125
MPI(16)	0.003704	0.271435	0.191109	0.346524	0.571003	1.044417
MPI(32)	0.102936	0.345817	0.371114	0.58126	1.140773	2.23851

1000	OpenMP(2)	OpenMP(4)	OpenMP(8)	OpenMP(16)	OpenMP(32)	OpenMP(64)
MPI(2)	0.047677	0.026749	0.019992	0.036822	0.101517	0.169161
MPI(4)	0.026308	0.014521	0.036435	0.090641	0.171635	0.27671
MPI(8)	0.015253	0.028596	0.079489	0.171152	0.296684	0.51963
MPI(16)	0.019582	0.147103	0.189126	0.301135	0.519389	0.913902
MPI(32)	0.11963	0.514176	0.362071	0.558471	1.042664	1.891182

10000	OpenMP(2)	OpenMP(4)	OpenMP(8)	OpenMP(16)	OpenMP(32)	OpenMP(64)
MPI(2)	6.908109	3.412642	1.770674	1.458001	1.594402	1.263003
MPI(4)	3.512451	1.727247	1.364862	1.448915	1.441726	1.214892
MPI(8)	1.85069	1.410352	1.459047	1.238553	1.178525	1.355824
MPI(16)	1.347408	1.623692	1.227718	1.223589	1.411428	1.97154
MPI(32)	2.235022	1.813892	1.684438	1.675916	2.315064	3.648908