Dan Gibson
ECE/CS 757 HW 1
Due 2/7/2005

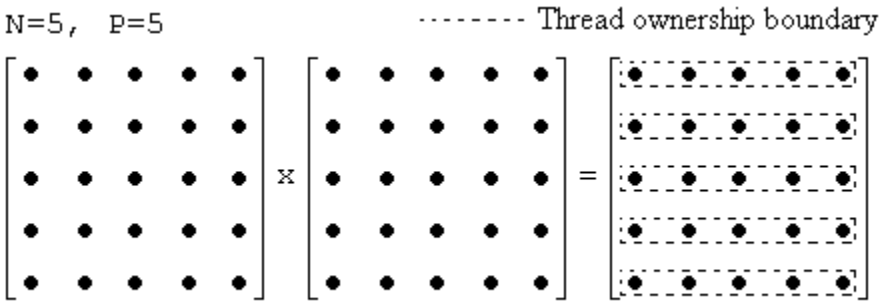
Program Descriptions

`randmatrix`:
A simple program that generates random square matrices of the specified dimension and integer range. Usage:
`randmatrix [N] [min_val] [max_val] > [output filename]`

`unimatrix`:
A single-threaded, very simple matrix multiplier. The output of this program is the file called "matrix." This file is automatically overwritten on execution. Usage:
`unimatrix [N] [infile1] [infile2] [infile3]`

`sharedmatrix`:
A multi-threaded matrix multiplier, written in the shared memory programming model. As with all the matrix multipliers, the output matrix is in the file "matrix." This file is automatically overwritten on execution. Usage:
`sharedmatrix [numprocs] [N] [infile1] [infile2] [infile3]`

This program is parallelized by assigning threads to calculate the values of rows of matrices—each thread is responsible for one or more rows of computation. The program requires no overt spin-locks, as in each epoch (separated by barrier statements) a given element is either read-only (in the case of the source matrices) or written by exactly one thread (in the case of the destination matrix). If there is no k for which N=k*P, then P is automatically reduces to the first P for which such a k exists.

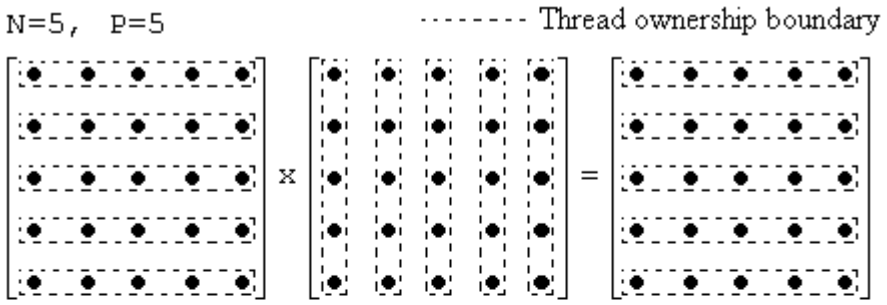


Note: Each thread may own more than one row.

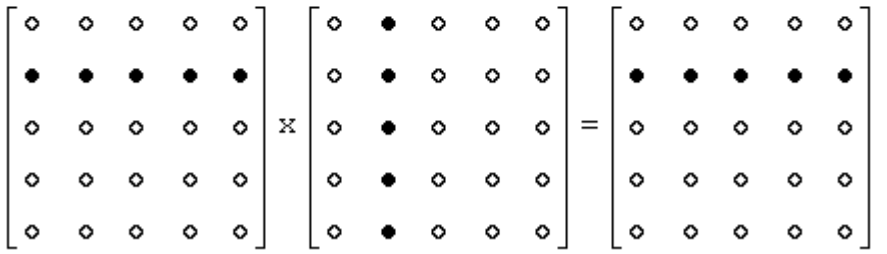
`passingmatrix`:
A multi-threaded matrix multiplier, written in the message-passing programming model. This matrix multiplier uses file "matrix" for output—this file is NOT overwritten on execution—it is appended. Between executions, one must execute `rm matrix` for proper output. Usage:
`mpirun.ch_shmem -np [numprocs] passingmatrix [N] [infile1] [infile2] [infile3]`

It should be noted that the `mpirun.ch_shmem` is necessary to execute `passingmatrix` correctly. It can be located in the same directory tree as `mpirun`.

This program is parallelized by giving each process ownership of separate portions of the matrices. The computational goal is A*B*C, which is implemented as R=A*B followed by R*C—each process "owns" k rows of A and R, and k columns of B and C, where k=N/P. The computations then proceed along column boundaries—the process that "owns" the current column provides the column via message passing to all other processes. Thereby, each process can compute a row of the result matrix per communication.



Note: Each thread may own more than one row/column.



Thread data visibility

Performance Analysis

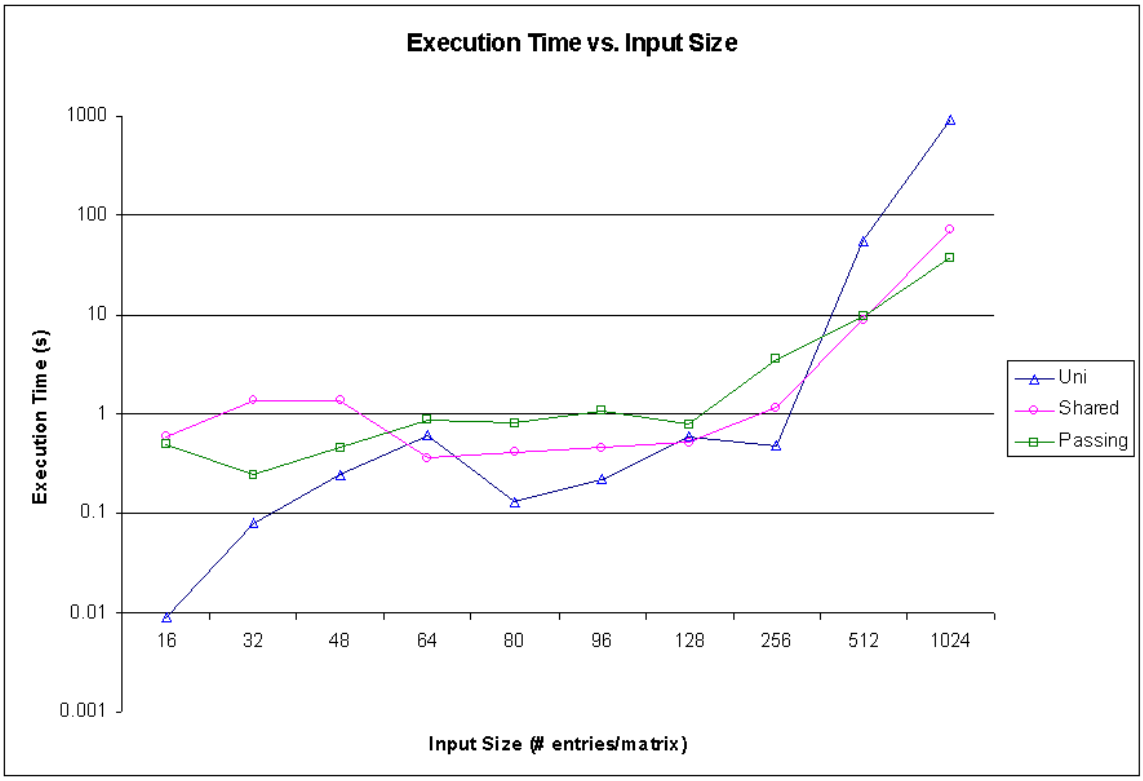
The performance of the matrix multipliers was tested for data sets ranging from N=16 to N=1024, where N denotes matrix dimensions of NxN. Each of the three input matrices were generated randomly. The data was gathered on the machine suggested for use in the homework description, `cabernet.cs.wisc.edu`. At the time that this data was gathered, there were some 10-20 users on the test machine, accounting for around 30 processes on average. As a result, average execution times of the matrix multipliers is somewhat unevenly skewed, but there is a general trend of speedup in the parallel cases.

In the case of the shared memory multiplier, a data size of N=1024 is required before the multiplier reliably requires more than sixty seconds of execution, an average of 84 seconds over three runs. Improvements to the synchronization structure of the shared matrix multiplier have reduces this runtime to 71 seconds on average, with quickest execution at approximately 61 seconds. These improvements reduced the number of barrier statements, and also improves the runtime of individual barriers.

Result of Empirical Testing

For all of the runtime tests below (except the uniprocessor case, naturally), 16 processors were used for the shared memory program, and 16 processes were allocated for the memory passing program. All execution times listed and plotted below are the result of two-way averaging. Times are listed in seconds.

N	Uni	Shared	Passing
16	0.009	0.581	0.497
32	0.079	1.366	0.248
48	0.245	1.378	0.465
64	0.606	0.36	0.86
80	0.13	0.42	0.822
96	0.218	0.467	1.09
128	0.597	0.504	0.777
256	0.485	1.15	3.592
512	55.614	8.884	9.426
1024	939.554	71.302	38.272



Appendix I
randmatrix

Appendix II
unimatrix

Appendix III
sharedmatrix

Appendix IV
passingmatrix