MATLAB Parallel Computing Toolbox Benchmark for an Embarrassingly Parallel Application

By Nils Oberg, Benjamin Ruddell, Marcelo H. García, and Praveen Kumar

Department of Civil and Environmental Engineering
University of Illinois
Urbana, IL

June 2008

Introduction

MATLAB is a scientific matrix manipulation program that provides an environment for rapid application development (RAD), scientific computations, plotting, and a wide variety of additional tasks. This environment is useful for such tasks as Particle Image Velocimetry (PIV) image manipulation, Geographical Information Systems (GIS) mapping, and statistics computations. MATLAB is extended by toolboxes, and includes two useful toolboxes, the Parallel Computing Toolbox and the Parallel Computing Server. These toolboxes provide a parallel computing environment for new and existing MATLAB code, ranging from a full-blown Message-Passing Interface (MPI) to a simple parallel for loop (`parfor`).

This technical note provides results of benchmarking an embarrassingly parallel[1] code written by Benjamin Ruddell. Three environments were used to test the code: 1) a desktop computer with Intel Hyperthreading, 2) a desktop computer utilizing a quad-core CPU, and 3) a 40 CPU-core cluster. The procedure used in benchmarking is provided in addition to the benchmark analysis.

Code Analysis

As a part of his PhD thesis, Benjamin Ruddell wrote code to compute Shannon entropies and the transfer entropy (information flow) between any number of observed timeseries variables, taken in pairs. This code is inherently embarrassingly parallel. The core of the code was rewritten in C++ and compiled into a MATLAB MEX-file using the MATLAB `mex` command. This alone resulted in a significant (4x) increase in performance, even over vectorized MATLAB code.

Procedure

The goal of benchmarking is to determine the performance of a code under controlled conditions. Therefore the first step is to ensure that no programs are utilizing the CPU on the benchmarked computer. Then the following pseudo-code may be used:

```
Set N = the set of node counts to benchmark with
Set L = the set of iteration counts to benchmark with

for every element of L -> L_i, do
      for I = 1 to L_i, do
            compute entropy
      end for
```

[1] An embarrassingly parallel problem is one that requires little or no effort to segment the problem into a large number of parallel tasks, and none of the segmented tasks are dependent on each other.

```
        for every element of N → $N_j$, do
              configure matlab worker pool for $N_j$ nodes

              $t_1$ = get time
              parfor I = 1 to $L_i$, do
                    compute entropy
              end parfor
              $t_2$ = get time

              deallocate matlab worker pool

              display ($N_j$, $L_i$, $t_2$-$t_1$)
        end for
  end for
```

In order to achieve accurate results, several iterations of the above pseudo-code must be performed, and the results averaged.

Results

Three separate experiments were performed to determine the performance of the MATLAB Parallel Computing Toolbox and Server under varying conditions. All three experiments used MATLAB version R2007b; however, the first and second experiments additionally tested the performance of version R2008a. These two experiments were used to compare the performance improvement in R2008a over the previous version.

Experiment 1

This experiment tested the performance of the Ruddell code on an Intel Pentium 4 3.0 Ghz processor with Hyperthreading enabled. RAM size was not relevant due to the size of the problem. The results are provided in Tables 1 and 2.

	Number of workers R2008a			Number of workers R2007b		
Iterations	1 (for)	1	2 (parfor)	1 (for)	1 (parfor)	2 (parfor)
10,000	2.00	2.33	2.18	2.02	8.65	9.03
50,000	9.85	9.91	8.80	10.3	9.68	8.71
100,000	19.6	19.5	16.8	20.0	19.2	16.9
500,000	98.9	97.5	83.7	101	95.6	83.5
1,000,000	199	191	164	206	194	170

Table 1: Hyperthreading time duration for workers vs. iterations

	Number of workers R2008a			Number of workers R2007b		
Iterations	1 (for)	1 (parfor)	2 (parfor)	1 (for)	1 (parfor)	2 (parfor)
10,000	1.00	0.86	0.92	1.00	0.23	0.22
50,000	1.00	0.99	1.12	1.00	1.06	1.18
100,000	1.00	1.01	1.17	1.00	1.04	1.18
500,000	1.00	1.01	1.18	1.00	1.05	1.21
1,000,000	1.00	1.04	1.21	1.00	1.06	1.22

Table 2: Hyperthreading speedup for T_{for} / T_{parfor}

Experiment 2

Experiment 2 tested the performance of the Ruddell code on an Intel Core 2 Quad 2.66 Ghz processor. RAM size was not relevant due to the size of the problem. The results are provided in Tables 3 and 4.

	Number of workers R2008a				Number of workers R2007b			
Iterations	1 (for)	1	2	4	1 (for)	1	2	4
10,000	1.22	1.53	0.947	0.664	1.22	7.72	7.21	7.21
50,000	6.13	6.23	3.38	1.93	6.15	12.6	9.74	8.60
100,000	12.3	12.6	6.27	3.39	12.4	18.2	12.6	9.84
500,000	61.8	61.4	30.4	15.7	62.0	65.7	36.4	22.3
1,000,000	123	121	61.5	30.9	123	125	66.7	37.4

Table 3: Quad-core time duration for workers vs. iterations

	Number of workers R2008a				Number of workers R2007b			
Iterations	1 (for)	1	2	4	1 (for)	1	2	4
10,000	1.00	0.79	1.28	1.83	1.00	0.16	0.17	0.17
50,000	1.00	0.98	1.82	3.18	1.00	0.49	0.63	0.72
100,000	1.00	0.98	1.96	3.62	1.00	0.68	0.98	1.26
500,000	1.00	1.01	2.04	3.95	1.00	0.94	1.70	2.78
1,000,000	1.00	1.02	2.00	3.99	1.00	0.98	1.85	3.29

Table 4: Quad-core speedup for T_{for} / T_{parfor}

Experiment 3

This experiment tested the performance of the Ruddell code, using MATLAB R2007b, on a 40 CPU-core cluster. MATLAB was configured to use up to 32 workers. One worker corresponded to one core. The cluster was comprised of 5 dual-CPU, quad-core systems for a total of 8 cores per system; connectivity was via gigabit Ethernet. Results are presented in Figure 1 and Tables 5 and 6.

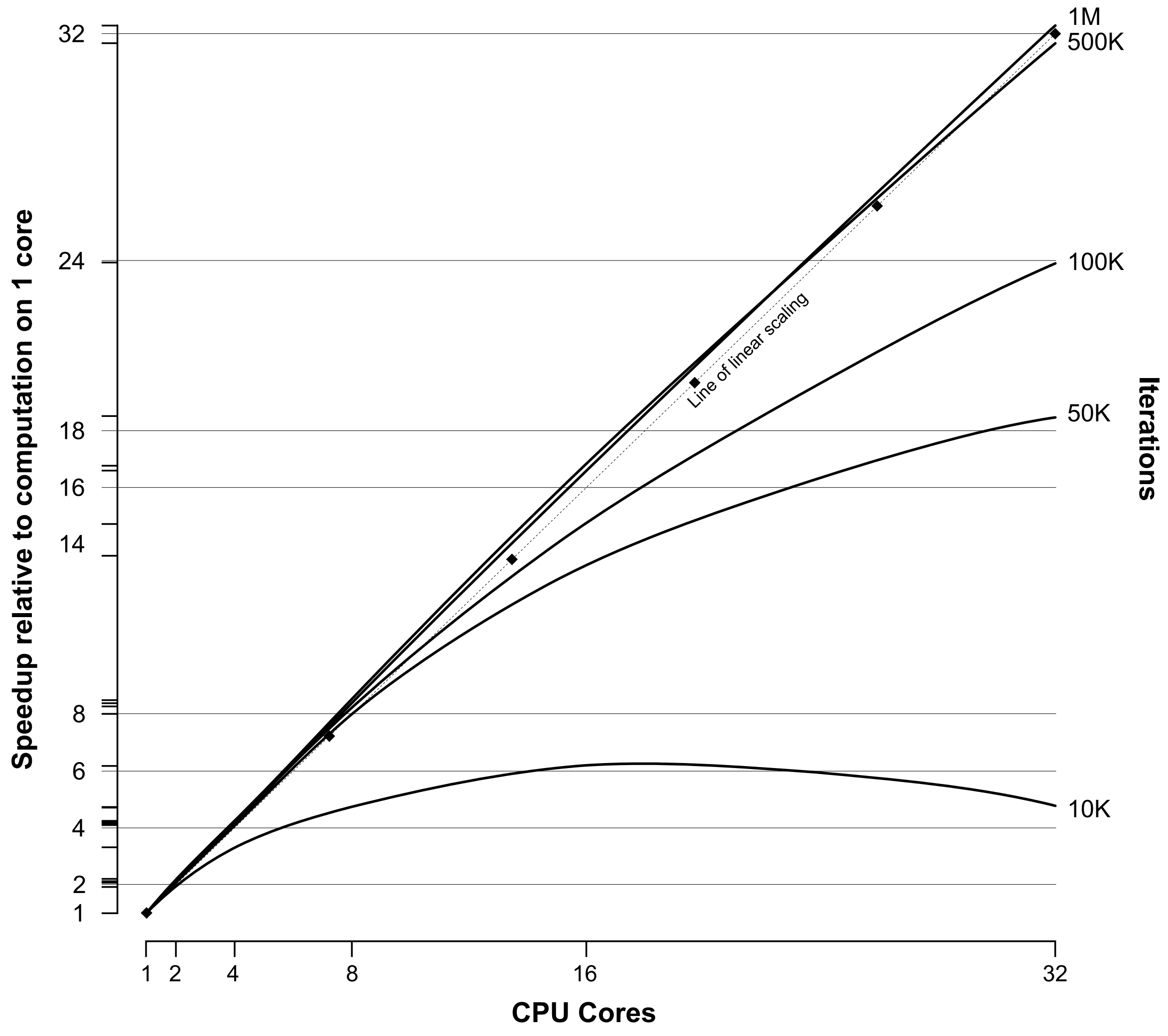

Figure 1: Speedup curves for cores vs. iterations

	Number of cores					
Iterations	1	2	4	8	16	32
10,000	1.99	1.02	0.603	0.419	0.320	0.416
50,000	9.24	4.46	2.27	1.15	0.696	0.500
100,000	18.3	8.72	4.34	2.22	1.24	0.765
500,000	90.4	41.4	21.3	10.6	5.37	2.86
1,000,000	176	82.8	42.1	21.0	10.6	5.46

Table 5: Cluster time duration for cores vs. iterations

	Number of cores					
Iterations	1	2	4	8	16	32
10,000	1.00	1.94	3.30	4.74	6.21	4.78
50,000	1.00	2.07	4.08	8.01	13.3	18.5
100,000	1.00	2.10	4.21	8.24	14.8	23.9
500,000	1.00	2.18	4.25	8.54	16.8	31.7
1,000,000	1.00	2.13	4.18	8.40	16.6	32.3

Table 6: Cluster speedup for T_1 / T_n

Observations and Conclusions

From the results above, it is clear that the parfor loop in MATLAB gives performance improvements over a for loop for some cases in all three experiments. In order to achieve a performance boost, a large number of iterations must be performed, the worker-to-core ratio must not exceed 1, and the number of workers must be 2 or more.

This is evidenced in Experiment 1, where for low numbers of iterations the overhead that parfor incurs decreases the parallel code performance below that of the serial code (for loop). Experiment 1 also shows that modest performance gains can be achieved on a Pentium 4 with Hyperthreading using 2 workers. In contrast, Experiment 2 shows that using multiple cores result in near-linear performance improvements as the core count increases for large numbers of iterations.

In Experiments 1 and 2, MATLAB version R2007b was compared to R2008a on identical hardware. The results indicate that R2008a provides significant improvements over R2007b for this particular problem.

In Experiment 3, nearly all iteration counts start out with super-linear increases in performance. That is, as the core count increases, a larger than linear increase in performance is achieved. For low iteration counts, however, performance decreases after a certain point due to the parallelization overhead incurred. The best performing case was for 1,000,000 iterations: it resulted in super-linear scaling up to 32 workers. It is estimated that for even large numbers of iterations the performance gain will taper off and eventually decrease as the worker count increases.

In conclusion, three principles govern the parallelization of this particular problem:

1. Use multiple workers
2. As the worker-count increases, the number of iterations must increase
3. Avoid small problems (low iteration counts)

It is important to note that these results are only reflective of an embarrassingly parallel problem, and these performance gains may not be realized in other types of problems.

Code

```
function testSuite

% Number of iterations to test for
iters = [10000, 50000, 100000, 500000, 1000000];

% Hyperthreading test
nodes = [1, 2];

% Quad-core test
%nodes = [1, 2, 4];

% Cluster test
%nodes = [1, 2, 4, 8, 16, 32];


% Run multiple times to get an average
for (t = 1:5)

    for (i = 1: length(iters))

        % Test the for loop first
        fprintf(1, '    Testing for, %i iterations ... ', iters(i));
        t = testParallelRuddell(iters(i), 0);
        fprintf(1, ' %.4f seconds\n', t);

        % Now test parfor on each node count
        for (n = 1: length(nodes))
            % Open the worker pool for a local hyperthreading/
            % quad-core test.
            matlabpool('open', nodes(n));

            % Open the worker pool for the parallel configuration
            % named 'cluster'.
            %matlabpool('open', 'cluster', nodes(n));

            fprintf(1, 'Testing on %d nodes\n', nodes(n));
            fprintf(1, 'Testing parfor, %i iterations ... ', iters(i));
            [t,results] = testParallelRuddell(iters(i), 1);
            fprintf(1, ' %.4f seconds\n', t);

            matlabpool close;
        end
    end
end


end % end function
```

```
function [tm, results] = testParallelRuddell(nIterations, useParFor)

% Set up parameters and input values here
params = [];


% Start the timer
tic;

% useParFor allows us to use the same function for testing both parfor
and for
if (useParFor)
    parfor (i=1:nIterations)
        [temp_results] = ShannonEntropy(params);
    end
else
    for (i=1:nIterations)
        [temp_results] = ShannonEntropy(params);
    end
end

% End the timer
tm = toc;

results = [];

end % end function
```