

# MATLAB Parallel Programming

## Some Timing Results on a Intel Nehalem Cluster

J.V. Burkardt\* & E.M. Cliff† & M. Snow‡

May 2009

### 1 Summary

Wall-clock, or elapsed time experiments were run on five problems under MATLAB's `Parallel Computing Toolbox` and `Distributed Computing Engine` (R2008b) on a cluster using INTEL's *Xeon Nehalem* processors. Most of the examples employ a *Task Parallel* strategy wherein multiple *workers* (processor cores) execute the same instructions on different data. The results are about what one expects. In several cases using a small number of workers produces a modest speed-up (say four workers perform the job in one-half the time - a speed-up of two). The achieved speed-up was largely constrained by the structure of the algorithms. The results indicate that the hardware/software produced about the level of speed-up that the algorithm would theoretically allow. In the following we describe the problems and give detailed timing results.

### 2 Introduction

We investigate time-performance of MATLAB's `Parallel Computing Toolbox` and `Distributed Computing Engine` (R2008b) on a cluster using INTEL's *Xeon Nehalem* processors. The Nehalem cluster is organized as 4 nodes with 8 CPUs each. Hyperthreading is currently enabled, so each node appears

---

\*burkardt@vt.edu

†ecliff@vt.edu

‡misnow@vt.edu

to have 16 CPUs but we run only 8 workers on each node. Assuming the scheduler is acting intelligently, each worker should end up with 1 CPU plus some overhead for OS operations. Each node has 24GB of memory, or 3 GB per core. One node is running at 2.93 GHz while the other 3 are running at 2.67 GHz. The nodes are interconnected with Gbit Ethernet.

We ran four example codes and recorded elapsed wall-clock times with (five) specified values for the number-of-workers. In most cases we ran available codes with a *wrapper procedure* that invoked MATLAB's `clock` command to capture time information at the beginning and end of a run.

```
% Wrapper script to start matlabpool and run a code
n_labs = 31;

disp(['n_labs = ' int2str(n_labs)])

save_time = zeros( 3, 6);
save_time(1, :) = clock; % [yr month day hour minute seconds]
if n_labs > 0 ; matlabpool('open', n_labs); end
save_time(2, :) = clock; % [yr month day hour minute seconds]

program                                % the script/function to be executed

save_time(3, :) = clock; % [yr month day hour minute seconds]

open_time = (save_time( 2, 4:6) - save_time( 1, 4:6))*[3600; 60; 1];
exec_time = (save_time( 3, 4:6) - save_time( 2, 4:6))*[3600; 60; 1];

if n_labs > 0; matlabpool close; end

disp(['Time to open pool (s): ' num2str(open_time)])
disp(['Time to execute (s): ' num2str(exec_time)])
```

### 3 Data Parallel

Our only *data parallel* example comes from documentation on the MATHWORKS website.

### 3.1 *LU Solve*

The problem is to solve a large linear system using an LU factorization procedure. The code generates a  $10,000 \times 10,000$  matrix ( $A$ ) of random values, and a  $10,000 \times 1$  column vector  $b$  (row-sums of  $A$ ). It computes an LU factorization of  $A$  and solves the system  $Ax = b$ .

Here's the code:

```
% Script to generate timing for parallel LU solve

n_labs = 16;

n_size = 10000;

if n_labs > 0; matlabpool('open', n_labs); end
save_time = zeros(2, 6);

save_time(1, :) = clock;

spmd % single-process multiple data
% Create a distributed array;
    A = rand(n_size, n_size, codistributor());
    b = sum(A, 2); % rhs so solution is ones (n_size, 1)

    [L, U, p] = lu(A, 'vector'); % codistributor/lu
    x          = U\(L\b(p, :) ); % solve

end

save_time(2, :) = clock; % [yr mo day hr min sec];
execute = (save_time(2, 4:6) - save_time(1, 4:6))*[3600; 60; 1]

if n_labs > 0; matlabpool close; end

% exit
```

Results for running the code with  $n\_labs = 0, 4, 8, 16, 31$  are shown in Table 1 and in Figure 1. Note that with up to eight workers the speed-up is reasonable (with 8 workers the speed-up factor is a bit more than 6), but that for 16 and 31 workers the execution time actually goes up. It turns out that this example [1] is badly flawed. The backslash operator  $\backslash$  for the

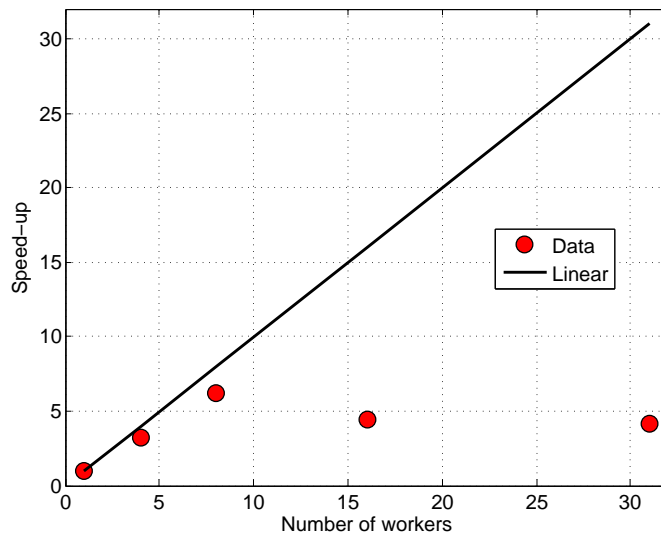


Figure 1: Speed-up for LU Problem

`codistributor` array-class is not nearly as sophisticated as the backslash operator for non-distributed arrays. In particular, it does not exploit the special ‘lower / upper’ structure engendered by the `lu` procedure [2].

Table 1: `lu` timing results

n_labs	time (s)
0	273.0
4	84.3
8	44.0
16	61.8
31	65.9

## 4 Task Parallel

In *task parallel* cases multiple *workers* (processor cores) concurrently execute the same instructions on different data. This construct imposes restrictions on the loop that mostly arise from the observation that unlike the single core case with multiple cores one can not know (in advance) what order the

loop index will follow. In some cases, the task is said to be *embarrassingly parallel*.

#### 4.1 *Molecular Dynamics*

This code simulates the motion of a collection of (NP) particles moving under pair-wise attractive forces. It starts with NP particles at random positions, and then integrates the equations of motion at fixed time steps. This requires computing, for each particle, the sum of the forces exerted on it by all other particles.

The main computational effort in the code arises from the calculation of forces on each particle. This task is performed in the function `compute.m`; the main effort is in the following loop:

```
parfor i = 1 : np

    Ri = pos - repmat ( pos( :, i ), 1, np );    % array of vectors to 'i'

    D = sqrt ( diag ( Ri' * Ri ) );            % array of distances

    Ri = Ri( :, ( D > 0.0 ) );

    D = D( D > 0.0 );                          % save only positive values

    D2 = D .* ( D <= pi2 ) + pi2 * ( D > pi2 ); % truncate the potential.

    pot = pot + 0.5 * sum ( sin ( D2 ).^2 );    % accumulate pot. energy

    f( :, i) = Ri * ( sin( 2*D2 ) ./ D );      % force on particle 'i'

end
```

In other parts of the code the applied forces, along with position/velocity data are used to update the position/velocity/acceleration of the particles. These calculations are performed on a single core. As a benchmark, we started with 1,000 particles and integrated for just ten time steps. We chose these values since this allowed the code to run in about five minutes on a single processor desktop machine.

The results for the timings are shown in Table 2 and in Figure 2. The speed-up displayed here is quite good. The *Linear* speed-up line is shown

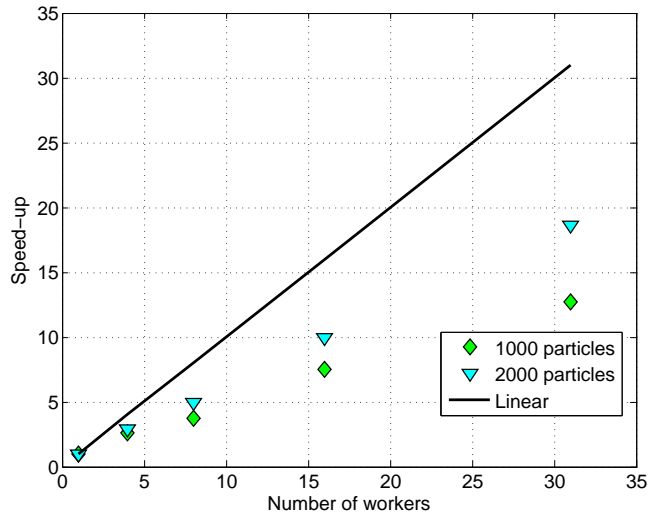


Figure 2: Speed-up for Molecular Dynamics Problem

for reference; since some calculations are performed on a single core, one cannot reasonably expect to achieve *Linear* speed-up.

Table 2: *Molecular Dynamics* timing results

n_labs	time (s) (1000 particles)	time (s) (2000 particles)
0	64.7	627.4
4	25.1	213.1
8	17.4	126.2
16	8.6	63.3
31	5.1	33.7

## 4.2 *Binary Match*

In this problem we generated patterns of 23-bits, representing the integers from 0 to  $2^{23} - 1 = 8,388,607$ . Each of these is to be checked against a logical template (pairwise combinations of bits either on or off). We seek to find the matching bit patterns and the associated integers. The check for match is done in a `parfor`-loop. With  $nw$  workers each worker checks

$\approx \frac{8388608}{nw}$  integers - an embarrassingly parallel construction.

The *wrapper* is the same form shown in § 2. The main loop in the executable program is shown here:

```
% solution_num = 0;

solutions = [];

parfor i = 0 : ihi - 1

    bvec = i4_to_bvec ( i, n );

    value = circuit_value ( n, bvec );

    if ( value == 1 )
%       solution_num = solution_num + 1;
        solutions = [solutions; i];
    end

end
```

Note that the original code wrote out the *matched values* of the index *i* and its binary representation in the loop. The MATLAB interpreter complained about the construct so it was replaced by accumulating the matching integers (the loop index) `match = [match; ii]`. After completing the check-loop the results in *match* were then printed. The `i4_to_bvec` converts the integer *i* to binary representation, while `circuit_value` performs the test against a logical template. For completeness we show the essential calculation of the logical value:

```
value = ...
    ( bvec(1) | bvec(2) ) & ( ~bvec(2) | ~bvec(4) ) ...
& ( bvec(3) | bvec(4) ) & ( ~bvec(4) | ~bvec(5) ) ...
& ( bvec(5) | ~bvec(6) ) & ( bvec(6) | ~bvec(7) ) ...
& ( bvec(6) | bvec(7) ) & ( bvec(7) | ~bvec(16) ) ...
& ( bvec(8) | ~bvec(9) ) & ( ~bvec(8) | ~bvec(14) ) ...
& ( bvec(9) | bvec(10) ) & ( bvec(9) | ~bvec(10) ) ...
& ( ~bvec(10) | ~bvec(11) ) & ( bvec(10) | bvec(12) ) ...
& ( bvec(11) | bvec(12) ) & ( bvec(13) | bvec(14) ) ...
& ( bvec(14) | ~bvec(15) ) & ( bvec(15) | bvec(16) ) ...
& ( bvec(15) | bvec(17) ) & ( bvec(18) | bvec(2) ) ...
```

```

& ( bvec(19) | ~bvec(1) ) & ( bvec(20) | bvec(2) ) ...
& ( bvec(20) | ~bvec(19) ) & ( ~bvec(20) | ~bvec(10) ) ...
& ( bvec(1) | bvec(18) ) & ( ~bvec(2) | bvec(21) ) ...
& ( ~bvec(22) | bvec(21) ) & ( ~bvec(23) | bvec(21) ) ...
& ( ~bvec(22) | ~bvec(21) ) & ( bvec(23) | ~bvec(21) );

```

After completing the check-loop the results in `match` were then printed. Timing results are shown in Table 3 and in Figure 3. The achieved parallelism is quite good; 31 workers achieved a speed-up of more than 28. The Virginia Tech system (available from head-nodes on `sysx.arc.vt.edu` has only 7 workers available. It achieved reasonable speed up with 4 workers, but performance was leveling off at 6 workers.

Table 3: *Binary Match* timing results

n_labs	time (s) <i>Nehalem</i>	time (s) <i>System X</i>
0	186.6	503.7
4	43.9	142.6
6	na	137.5
8	22.5	na
16	12.1	na
31	6.6	na

### 4.3 Constrained Minimization

MATLAB’s Optimization Toolbox provides several codes for finding *best-performance* - a common task in engineering design and in scientific analysis. In particular, `fmincon` offers several options for solving the constrained optimization problem

$$\text{Find } z \in \mathbb{R}^n \text{ to } \min f(z) \text{ subject to } c(z) = 0.$$

Here  $f$  and  $c$  are smooth nonlinear functions;  $f$  is real-valued, and  $c$  may be vector-valued. The algorithms are all sequential, and employ gradient information to produce an improved approximation of a solution. Thus, they rely on having (estimated) values for the partial derivatives of  $f$  and  $c$  with respect to the components in  $z$ . Commonly such derivatives are estimated from finite-difference quotients. The simplest forward difference

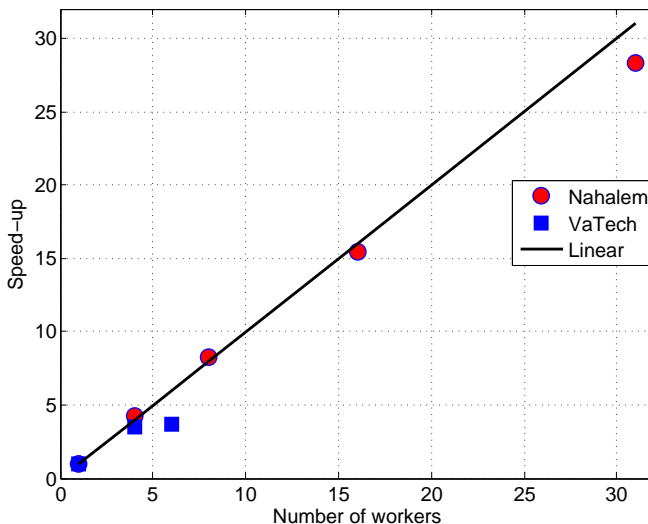


Figure 3: Speed-up for Binary-Match Problem

method requires an additional  $n$  evaluations of the functions  $f$  and  $c$ . The Optimization Toolbox (R2008b) allows for the parallel computation of these gradients (setting the option ‘UseParallel’, ‘always’ invokes the procedure `parfinitedifference` in the Optimization Toolbox). Thus, the user only needs to set this flag to invoke parallelism in this part of the code. Note that once the gradient information is assembled, the remainder of the calculations are done on a single processor. This includes additional function evaluations needed to determine an acceptable step from the current approximation  $z_k$  to the next one  $z_{k+1}$ .

In the present application, the unknowns  $z$  are certain parameters in a coupled system of five ordinary differential equations  $\dot{x}(t) = g(t, x(t), z)$ . More specifically,  $z$  parameterizes a piecewise-constant approximation to the control function on a uniform time-grid. A finer grid requires more components in  $z$  (larger  $n$ ). Evaluating  $f$  and  $c$  requires that one solve an associated initial-value problem on the interval  $[0, 1]$ .  $f$  and  $c$  are evaluated from the values of the  $x(1)$ . The related code is much too lengthy to include here.

Timing results for problems of various size run on several workers are shown in Table 4 and in Figure 4. We see that a modest number of processors provide some speed-up, but that the improvement falls off quickly with the number of processors. This may be in large part due to the algorithm. Specifically, we are using parallel processing only in the calculation of

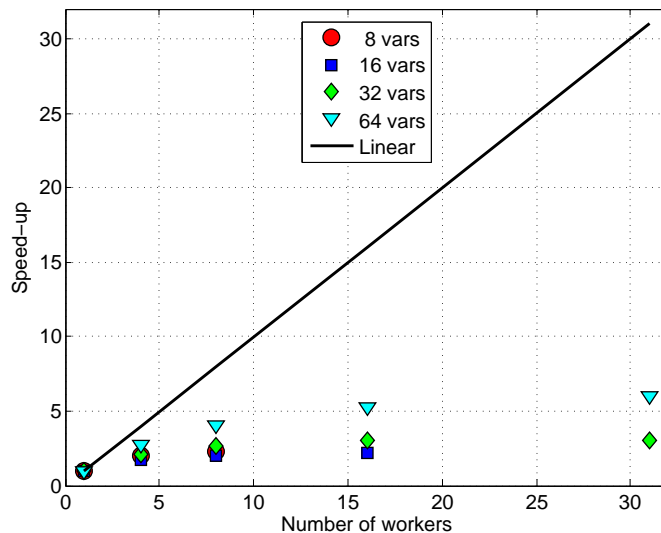


Figure 4: Speed-up for `fmincon`

finite-difference estimates for Jacobians. As noted above, the optimization algorithm requires additional function evaluations as part of its step-size selection strategy. As an example in the problem with 32 variables on 16 processors the code ran a total of 21 iterations using 846 function evaluations. 672 evaluations were for finite-difference estimates, and the remaining 174 were run in the step-size selection procedure (on a single processor). For each gradient calculation, each worker would ideally perform 2 function evaluations for a total of 42 function cycles. Add to this the 174 function cycles required in the step-size selection procedure for a total of 216 function cy-

Table 4: `fmincon` timing results

n_labs	time (s) 8 vars	time (s) 16 vars	time (s) 32 vars	time (s) 64 vars
0	60	123.6	246	695
4	29.7	70.9	118	254
8	26.7	61.4	92	173
16	na	57	82	131
31	na	na	82	115

cles. Thus, the ideal speed-up would be  $\frac{846}{216} \approx 3.9$ . The achieved speed up was  $\frac{266}{82} = 3$ . Finally, note that 31 workers would require the same number of function cycles. If 32 workers had been available we might have achieved  $\frac{846}{174+21} \approx 4.3$ .

## 5 Conclusions

The results indicate that the hardware/software produced about the level of speed-up that the algorithm would theoretically allow. The *Molecular Dynamics* and the *Binary Match* codes were instances of Task Parallel where a significant part of the computation could be done in parallel. The *fmincon* example resulted in more modest speed-up, mainly due to the structure of the algorithm. The only Data Parallel case *lu* was disappointing -at this point we have no insights to the observed behavior.

## References

- [1] anon, Implementing Data-Parallel Algorithms, at <http://www.mathworks.com/products/parallel-computing/description2.html>
- [2] N. Johanson, MathWorks, private communication.