Parallel Programming with MATLAB

John Burkardt and Gene Cliff

October 14, 2013

Contents

Chapter 1

THE COMBINATION LOCK

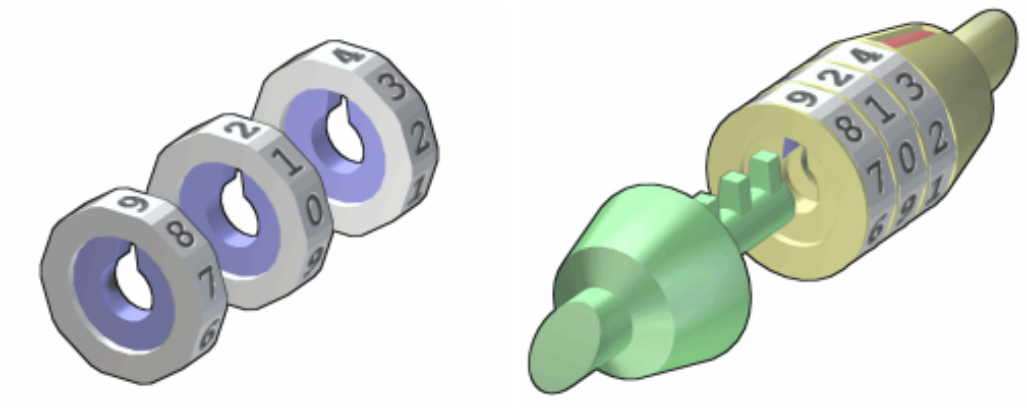

Figure 1.1: The locking rings (left) and the combination lock (right)

1.1 Introduction

A determined, methodical, and careful person can open a combination lock, given enough time, simply by trying every possible combination in order. If someone offers to help, and there's a copy of the lock with the same unknown combination, then we can expect the task to be accomplished in half the time, and if more helpers are available, the better.

We can formulate the combination lock problem on the computer, and simulate the procedure of a person trying to determine the combination.

What may be surprising is that, if more than one processor is available to the computer, it is again possible to accomplish the task more quickly, by sharing the work among the available "helpers".

1.2 The Combination Lock Problem

A common class of combination locks involves a set of identical numbered dials, each of which can be turned independently to display one of the numbers. The lock is manufactured in such a way that it will only open if each dial has been set to the correct numeric setting, called the "combination". A typical lock used for a bicycle or suitcase might have three dials, each labeled with the digits 0 through 9. Such a lock obviously has exactly $10 \times 10 \times 10 = 1,000$ possible combinations.

Since we will be interested in variations of this problem, let's reserve m to represent the number of dials, and n be the number of digits or symbols or settings to choose from on each dial. Using this notation, we may conveniently refer to a combination lock by specifying the two values (m, n), in that order. The bicycle lock is thus a $(3, 10)$ lock.

The number of combinations in an (m, n) lock is n^m, which means that, simply by adding a few more dials or symbols, we can quickly be looking at millions or billions of combinations. The combination lock thus has a feature common to many computational problems that can be described by a few parameters: for small parameter values, the problem is easily solved, but with a modest increase, the problem quickly becomes extremely difficult.

If we wish to pose an example of the combination lock problem, we must include a choice for the combination. We can designate this by c. The quantity c must be a vector of length m, containing the correct settings for the dials. Once we have selected the lock parameters (m, n) and the combination c, we have described a problem.

A locksmith could build this lock, and allow contestants to try to open it. We, instead, will "build" a simulation of the lock on the computer, and follow that up by simulating the procedure by which someone might try to determine the combination.

1.3 Opening the Bicycle Lock Sequentially

We'll start by determining the combination of a (3,10) bicycle lock. Since there are just 10 symbols on the dials, we can think of represent them as the digits 0 through 9. By concatenating the three symbols of the combination, we can think of it as simply a number between 0 and 999. This point of view makes it easy to come up with a solution procedure, (just start counting!). It also makes it possible to pause at any time, and restart later, as long as we know the last combination we tried. We will use that fact shortly.

A simple MATLAB procedure for the bicycle lock problem might be:

```
%% combination_lock.m
%
%  Set the combination C.
%
  rng ( 'shuffle' );
  c = randi ( [ 0, 999 ], 1, 1 );
%
%  Generate every number A between 0 and 999,
%  and then see if it is the combination C.
%
  for a = 0 : 999

    if ( a == c )
      fprintf ( 1, '\n' );
      fprintf ( 1, '  The combination is %d!\n', a );
      break
    end

  end
```

Given how simple the solution is, it might hardly seem that this problem was worth coding up. But before we go on, let us note that the reason we were able to solve the problem was that we came up with a representation for the combinations, and a procedure for generating each combination exactly once.

1.4 Opening the Bicycle Lock in Parallel

Because we were able to represent the possible combinations as the integers between 0 and 999, it should be clear how the solution procedure could be

speeded up if we had one or more workers to help us: simply divide up the range [0,999] among the available workers so that each combination is checked by somebody. Perhaps the worker who finds the answer should also announce this fact so that the others can stop looking.

Since this problem will run very quickly on a single processor, it might not seem worth using parallel computing. Nonetheless, it's so easy to request parallel processing that we might as well introduce it for this example. We have to replace the keyword **for** by **parfor**, if a MATLAB loop is appropriate for parallel processing. The **parfor** statement indicates that the work of the loop, (the individual iterations) can be divided up among the set of available processors.

Our sequential code includes a **break** statement. The rules for using the **parfor** statement require that the loop not involve any **break** statements. Fortunately, we can easily modify our program to omit the **break**; as long as we print the combination when we find it, we will get the information we need.

Thus, the text of a parallel version of our algorithm might be:

```
1  %% combination_lock_parfor.m
2  %
3  %  Set the combination C.
4  %
5    rng ( 'shuffle' );
6    c = randi ( [ 0, 999 ], 1, 1 );
7  %
8  %  Generate every number A between 0 and 999,
9  %  and then see if it is the combination C.
10 %
11   parfor a = 0 : 999
12
13     if ( a == c )
14       fprintf ( 1, '\n' );
15       fprintf ( 1, '  The combination is %d!\n', a );
16     end
17
18   end
```

The revised program can still be executed by MATLAB, in the usual way; however, the "usual way" involves running sequentially. So once we've created a parallel program, we have to issue some commands to MATLAB to request that the program actually be run using multiple processors. Assuming our desktop machine has the Parallel Computing Toolbox installed,

let's run the algorithm with 4 processors:

```
%% Run the parfor script on 4 local workers

  matlabpool open local 4

  combination_lock_parallel

  matlabpool close
```

The **matlabpool** command directs MATLAB to set up a pool of 4 workers; the script begins executing with the client copy of MATLAB, but when the **parfor** statement is reached, these workers are called, and each is assigned a segment of the loop iteration range. One of the workers finds the combination and prints it; but each worker continues until it has checked every value in its range. Once all the loop iterations are completed, the workers "retire", and the client resumes control of the calculation.

1.5 Discussion

Our simple example asks us to find the combination, and we simply pick a number at random as our goal, which might make the whole search seem somewhat contrived. But in fact, many real problems share this structure, with the exception that now the answer isn't known in advance. For example, instead of searching for a 3 digit integer a that equals c, we might be searching for a number a such that the function $f(x) = x^5 - 123x^4 + 2x^3 - 246x^2 + 3x - 369$ is equal to zero. There is just one value a between 0 and 999 for which this is true, and we can search for it in the same way, but now the answer is no longer obvious.

In this example, it's easy to believe that we can replace a sequential calculation by one in which multiple workers divide up the job and complete it correctly, because the task was so simple. There are no complicated arrays to be read or written, no worker has to consider what any other worker has done. But surely most problems are much more complicated than this one; it's natural to worry about a complicated numerical calculation can similarly be divided up among cooperating workers. We will address such issues shortly.

Another issue is that while we have been successful in running the program in parallel, our real goal is to get answers faster. For this tiny example,

it's not really clear that we have achieved any such benefit. In fact, if you actually run the parallel code, you will notice that the **matlabpool** command takes a perceptible time to complete, perhaps longer than the calculation itself. Our main purpose in this example was simply to get *something* running in parallel, the simpler the better - however, it is important not to imagine that a parallel program is better, or guaranteed to be faster, than a sequential version. In fact, we will see that it is always important, when using parallel programming, to check that parallelism is helpful for the number of processors and problem sizes that we are likely to consider.

Chapter 2

QUADRATURE WITH PARFOR

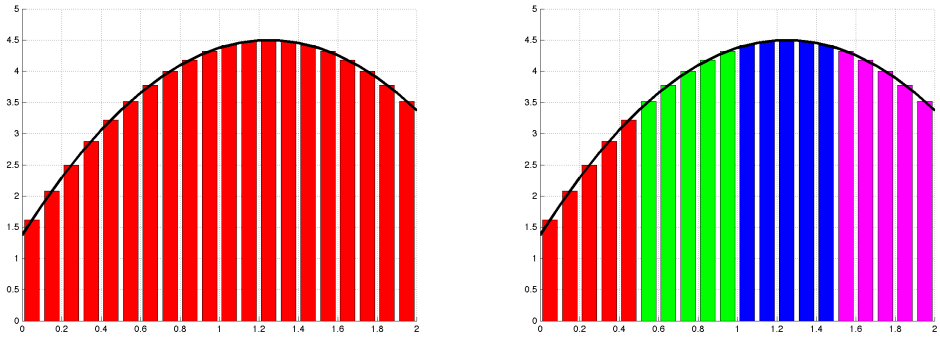

Figure 2.1: Sequential and Parallel Integral Estimates by Quadrature

2.1 Introduction

The integral of a function $f(x)$ over the interval $[a, b]$ can be interpreted as the area between the x-axis and the curve $y = f(x)$. If we can't see a way to determine the exact value of the integral through calculus, we can think of approximating the integral by considering the area. This approach is called *quadrature*. It makes an estimate of the integral by evaluating the function at many points.

Quadrature is an ideal application for parallel programming, because it involves a large number of simple, independent tasks whose properties can be defined in advance. Notice however, that the quadrature problem raises a new issue that did not arise before, namely *communication*. In the combination lock problem, there was never any interaction between the workers, and the client simply assigned tasks to the workers and never asked for a result; instead, a worker that found a combination simply printed it directly to the user. For the quadrature problem, the workers still won't need to communicate with each other, but once each worker has computed its partial estimate, it must communicate this information to the client, so that it can be gathered into a single final estimate.

This is a logical feature of the computation, and we shall want to know that MATLAB is able to handle it properly.

2.2 The Sequential Quadrature Problem

We suppose that we are given a function $f(x)$, and an interval $[a, b]$. The integral is symbolized by $I(f, [a, b]) = \int_a^b f(x)dx$. We suppose that it is not possible to determine the exact value of $I(f, [a, b])$, and so instead we plan to approximate the value by a quadrature formula, which we will write as $Q(f, [a, b])$.

One of the simplest quadrature formulas is the *midpoint rule*:

$$Q_M(f, [a, b]) = (b - a) \cdot f(\frac{a + b}{2})$$

While the accuracy of this rule is low, improved results can be obtained by using the *composite midpoint rule*, which involves dividing the interval $[a, b]$ into n equal subintervals, applying the midpoint rule to each, and then summing the result:

$$\begin{aligned} Q_{CM}(f, [a, b], n) &= \sum_{i=1}^{n} (b_i - a_i) \cdot f(\frac{a_i + b_i}{2}) \\ &= \frac{b - a}{n} \cdot \sum_{i=0}^{n} f(\frac{(n - i)a + ib}{n}) \end{aligned}$$

A simple MATLAB procedure for the quadrature problem might be:

```
function q = quad_sequential ( a, b, n, f )
%% quad_sequential.m
  q = 0.0;
  for i = 1 : n
    ai = ( ( n - i + 1 ) * a + ( i - 1 ) * b ) / n;
    bi = ( ( n - i     ) * a +   i       * b ) / n;
    xi = ( ai + bi ) / 2.0;
    q = q + f ( xi );
  end
  q = q * ( b - a ) / n;

  return
end
```

Our computation is written as a MATLAB M-file; we invoke it by setting the values of the input. Here, our function to integrate will be $f(x) = (4x+1)\cdot(11-4x)/8$, and for convenience, we will use MATLAB's *anonymous function* facility to define the function f on a single command line:

```
%% quad_sequential_call.m
  a = 0.0;
  b = 2.0;
  n = 21;
  f = @(x) (4*x+1)*(11-4*x)/8;
  q = quad_sequential ( a, b, n, f );
  fprintf ( 1, '  The integral estimate is %f\n', q );
```

2.3 Quadrature Using PARFOR

We can make a parallel version of the quadrature computation simply by replacing **for** with **parfor**.

```
function q = quad_parfor ( a, b, n, f )
%% quad_parfor.m
  q = 0.0;
  parfor i = 1 : n
    ai = ( ( n - i + 1 ) * a + ( i - 1 ) * b ) / n;
    bi = ( ( n - i     ) * a +   i       * b ) / n;
    xi = ( ai + bi ) / 2.0;
    q = q + f ( xi );
  end
  q = q * ( b - a ) / n;

```

```
12    return
13  end
```

Before we go on, though, we need to try to make a mental model of how the variables in the parallel program are accessed by the workers while executing the loop. It's easy to imagine that the variables **a**, **b**, and **n**, as well as the expression for **f** are "owned" by the client copy of MATLAB, and that the workers either get a copy of these values, or can refer to them at any time (and this is correct).

However, we can't really think about the variable **q** in the same way. The client initializes **q** to zero, but doesn't execute the statements inside the loop which update its value. So it's the workers that have to update the value of **q** with the new information. If we suppose there's only one copy of **q** available, and we carefully insist that only one worker at a time can update it, then we will surely have a considerable amount of undesirable delay while workers wait for their turn to apply their update. On the other hand, if we simply allow any worker at any time to read or write the single copy of **q**, then we are liable to get erroneous results. Two workers could try to update at the same time, and only one of the updates would be correctly stored.

Luckily, MATLAB is automatically able to recognize what is going on with the variable **q**, and essentially, as the loop starts, it creates for each worker a zeroed-out private variable called **q**, which that worker alone is free to modify. On the completion of the loop, MATLAB quietly gathers the multiple copies of **q**, and adds them to the value of **q** that was stored in the client before the parallel loop began. The variable **q** is an example of what's called a *reduction variable*.

Notice that the variable **x** has a simpler behavior. It is simply a temporary variable, which is only used inside the loop. We might imagine that each worker makes a separate private copy of **x** to be used during its iterations. The variable **x** is just a temporary convenience, and once we exit the loop, its value is of no importance. Thus, even though the sequential code uses a single copy of **x**, but the parallel code uses multiple versions of this variable, we are right in assuming that this is not going to cause any problems.

To invoke the code, once again we have to use the **matlabpool** command to request the help of the workers:

```
1  %% quad_parfor_call.m
2    a = 0.0;
3    b = 2.0;
```

```
4   n = 21;
5   workers = 4;
6   f = @(x) (4*x+1)*(11-4*x)/8;
7   matlabpool ( 'open', 'local', workers );
8   q = quad_parfor ( a, b, n, f );
9   matlabpool close
10  fprintf ( 1, '  The integral estimate is %f\n', q );
```

2.4 Comparing Sequential and Parallel Rates

When we try the parallel program out on our sample problem, for some small problem sizes, the results don't look very good! We run the sequential code ("0 workers") versus the parallel code with 1, 2 or 4 workers, and for n ranging from 1 to 2^{10}. The parallel timings are consistently worse than the sequential code and even seem to get worse as we add more workers!

However, it's important to be cautious here. Note that even the sequential timings are not showing a pattern of doubling as the problem size doubles. This suggests that the computational work in the problem is insignificant compared to the other things that MATLAB is doing for us. We have to remember that the benefits of parallelism come for problems with a "significant" amount of work. Certainly our quadrature function f is trivial to evaluate, and perhaps a few hundred evaluation points of an easy function don't constitute enough work.

N/W	0	1	2	4
1	0.2923	0.4164	0.2306	0.2436
2	0.0073	0.0294	0.0270	0.2131
4	0.0008	0.0234	0.0223	0.2150
8	0.0008	0.0205	0.0235	0.0293
16	0.0008	0.0263	0.0263	0.0391
32	0.0008	0.0306	0.0395	0.0435
64	0.0008	0.0207	0.0326	0.0489
128	0.0008	0.0201	0.0303	0.0410
256	0.0009	0.0201	0.0332	0.0362
512	0.0010	0.0213	0.0258	0.0320
1024	0.0012	0.0204	0.0250	0.0342

We still assume that there are some versions of this problem for which parallelism provides a benefit. How do we look for them? We could look

at cases where the function is more expensive to evaluate, but instead, let's stick with our current function, gradually increase the value of n, and see if there's at least some range over which the parallel code runs faster.

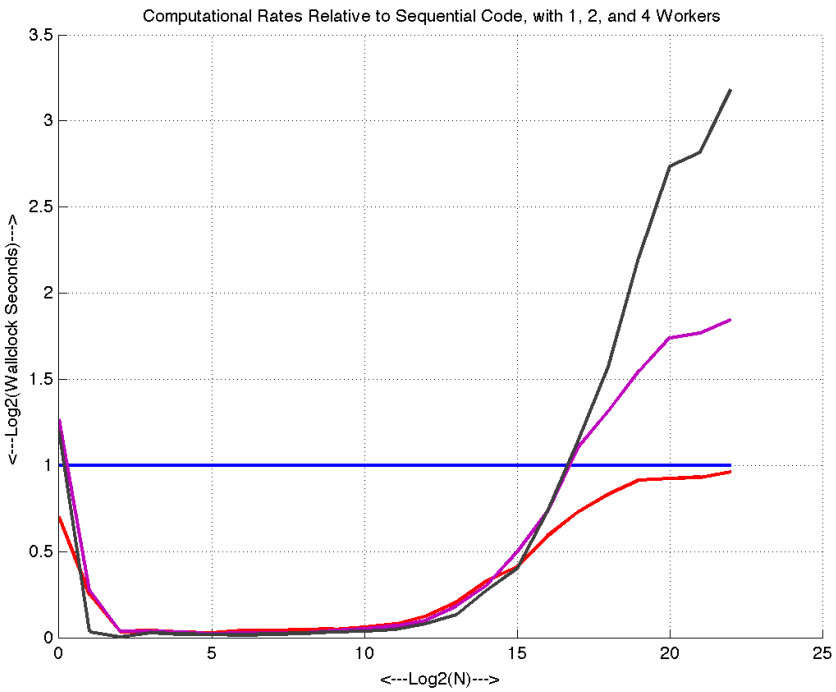

Figure 2.2: Rates for 1, 2, 4 workers, relative to sequential code

Our comparisons will use n = 1, 2, 4, ..., 2^{23} points, and we will try the sequential code ("0 workers"), as well as parallel runs with 1, 2, and 4 workers. We suspect that the 1 worker code will *never* run faster than the sequential code, but we hope that there will be a range where the 2 and 4 worker codes are about twice and four times as fast as the sequential code.

To make this trend easy to spot, we'll plot the computational rates relative to the sequential code. That means that, for every problem size, the rate of the sequential code will be plotted as 1, showing up as a horizontal line. If a parallel code is faster for that problem size, it will appear above the line.

From the plot, we get some reassurance and understanding. The sequential code is represented by the blue line. The 1 worker code is red, and we see that it is never better than the sequential code, and at the highest values

of n seems to be leveling off in rate. The 2 and 4 worker codes first beat the sequential code at $n = 2^16$, and thereafter, their rates seem to be rising towards their theoretical limits of 2 and 4.

We can see that if we had only studied values of n up to 2^15, we would never have seen a benefit from parallelism. The fact that we have to wait so long to see a benefit is also because the integrand is so simple to evaluate. This suggests that the small, simple problems we like to use for benchmarking will often lead us to incorrect conclusions. To understand whether parallelism will benefit your problem, it's really important to examine a realistic range of problem sizes, parameters, and difficulties.

Chapter 3

KNOWING WHEN TO STOP

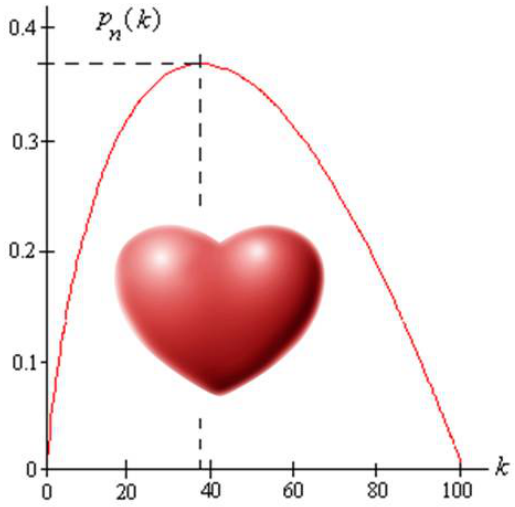

Figure 3.1: To find Ms Right in 100 candidates, try 37 dates, then get serious!

3.1 Introduction

A mysterious stranger has arrived in town with a deck of cards, and a wagonload of gold. He announces that he has decided to give away his fortune, by letting each person in the town play a game. He has a deck of ten blank cards. On the face of each card, he writes a number representing an amount

in gold coins, and then turns the cards face down. Now he turns over the first card. "Is this the card you want?" he asks, and the player can either accept it, winning the corresponding amount, or reject it. If the player rejects that card, the next one is turned over, and so on, until the player accepts a card, or reaches the last card which is then, by default, the player's winnings.

A greedy player will naturally wish to be lucky enough to choose the very highest card. If the player decides in advance to choose a card at random, the seventh card, perhaps, then this strategy would have a 1/10 chance of correctly selecting the highest card. But somehow, that doesn't seem the best strategy. After all, that's the same as always picking the first card. But it's hard to resist the feeling that you really ought to turn over some cards before making a choice. Except that the price of seeing the next card is giving up the one you have.

The game would be easy if it was known in advance what numbers appeared on the cards, so we will suppose the range is not known. Even if we don't know the range, particular values such as "1" or "2" would almost certainly not be the maximum; so to make sure the puzzle is hard, we will suppose that the card values are chosen in such a way that every card could be the maximum. Here is a sequence of numbers chosen in such a way.

```
512   256   896   320   928    80   680   116   118   571
```

Given 10 cards, we start with 2^9, and then randomly add or subtract 2^8 to get the next card, then randomly add or subtract 2^8 and 2^7 to get the next card, and so on. This guarantees that every card is equally likely to be the maximum, and that the location of the maximum can never be determined from the observed values of the preceding cards.

This is an example of what is called an *optimal stopping problem.* Martin Gardner described this problem as "The Game of Googol". A variation, called "The secretary problem", involves an employer seeking a secretary, who has scheduled interviews with 50 applicants, and will either say "Next!" or "You're hired, send the rest home!" at the end of each interview. The problem has also been posed as a person wanting to get married; a dating service has provided the names of 100 candidates, and the person, while wishing to marry the best of the candidates, must date each in turn and either accept that person, or permanently reject them and request the next candidate. Here, the assumption is made, of course, that the candidate is willing to accept the marriage proposal, and is not also simultaneously dating a number of people looking for the best!

3.2 An Optimal Strategy

Surprisingly, a strategy can be devised which, on average, will select the highest card, the best secretary, or the ideal mate about 37% of the time. Moreover, this is true pretty much independent of n, the number of choices available. The strategy is in two parts:

- examine, but reject, the first $k-1$ items;
- examine items k through n, accepting the very first one whose value is greater than the first $k-1$.

For a given value of n, we can work out the probability that a given k will product the optimal value, and in fact, we can write a MATLAB function to do this for us:

```
1 function p = high_card_exact ( n, k )
2 %% high_card_exact.m
3   p = ( 1 + ( k - 1 ) * sum ( 1 ./ ( k:n-1) ) ) / n;
```

For our mysterious stranger's game using a 10 card deck, the results are:

k	1	2	3	4	5	6	7	8	9	10
p(k)	0.100	0.283	0.366	0.399	0.398	0.373	0.327	0.265	0.189	0.100

The optimal value of k for this problem is clearly 4. But, except for very small values of n, the optimal value can be well approximated by $k^* \approx \frac{n}{e}$, which here would give us $k^* = 3.6788$, and in general, the chances that this strategy will succeed are $p(k^*) = 1/3 = 0.3679$.

3.3 A Sequential Simulation

Let us assume that we know about the high card game, and wish to analyze it computationally. We choose a deck size n, and then want to see how the value of the skipping strategy for various skip values k.

When a human is playing the game, it's important to try to disguise the range of values. However, we'll assume the computer makes no conclusions from the actual values displayed, so our deck of n cards can simply be modeled as a random permutation of the integers 1 through n.

So our procedure will be, for each skip value k from 1 to n, to simulate t realizations of the game using the given skip strategy. If h is the number of

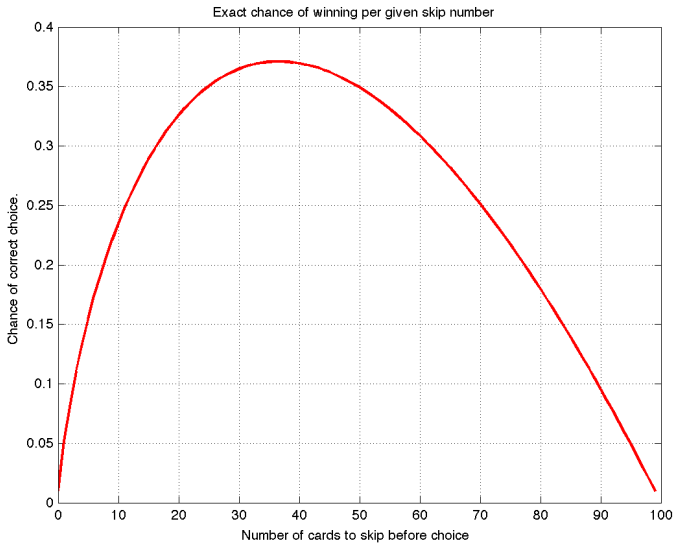

Figure 3.2: The exact probability of winning high card with a deck of 100

times we picked the high card (which here will simply be the one labeled n), then we estimate the probability of winning with skip k as

$$p(k) \approx \frac{h}{t}$$

Given values of n and k, a function to estimate p could be:

```
function p = high_card_estimated ( n, k, t )
%% high_card_estimated.m
  h = 0;
  for j = 1 : t
    cards = randperm ( n );
    margin = max ( cards(1:k-1) );
    for i = k : n
      if ( margin < cards(i) )
        if ( cards(i) == n )
          h = h + 1;
        end
        break;
      end
    end
  end
  p = h / t;
```

Of course, while the function is the key to the calculation, it only computes the estimated probability for a particular value of k. To judge what the optimal value of k is, or to make a plot of the estimated probabilities, we would need to call *prob_estimat(n,k,t)* for each value of k from 1 to n.

3.4 A Parallel Simulation Using parfor

If we wish to compute a table $p(1:n)$ of the probability estimates for each skipping value, then we only need to add a simple loop to generate each k and call the function. Computing this data in parallel is therefore also quite easy, since the details are hidden in the function; moreover, the fact that the function depends only on the fixed numbers n and t and the loop index k means it is almost certain that the computation is suitable for parallel treatment.

Since the loop is so short, we'll include the calls that summon and dismiss the workers, and suggest how the result vector can be displayed:

```
1  %% high_card_estimated_call.m
2    n = 100;
3    t = 1000;
4    matlabpool open local 4
5
6    parfor k = 1 : n
7      p(k) = high_card_estimated ( n, k, t );
8    end
9
10   matlabpool close
11
12   plot ( 1:n, p, 'b-' );
13   title ( 'Estimated winning probabilities for 100 card deck.' )
```

An plot of the results for the 100 card deck, estimated by 1,000 trials for each skip value k, captures the general shape of the curve computed from the exact formula, although it remains surprisingly irregular in detail.

As far as parallelism goes, however, there are several points to consider.

A new feature of this example is that the *parfor* loop stores data into an array whose results will be needed after the loop is completed. This array is accessed by the loop parameter in the simplest way. Loops controlled by parfor impose some strict requirements on the ways in which arrays can be used on the left or right hand sides of the statements they control. Luckily,

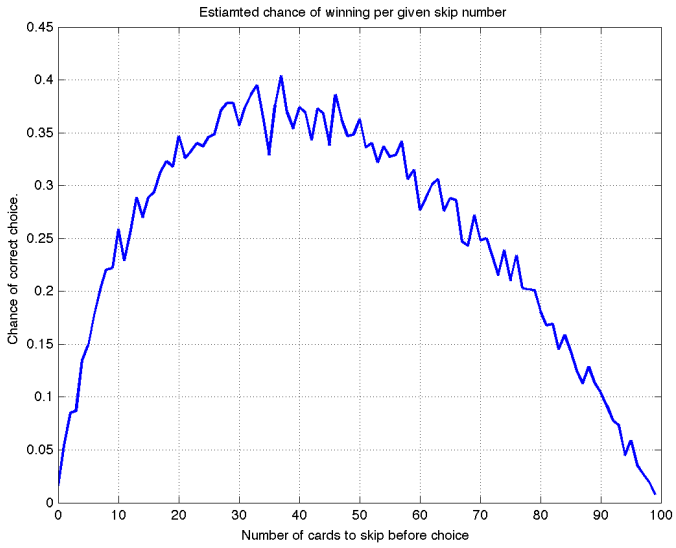

Figure 3.3: The estimated probabilities, with a deck of 100, and 1,000 trials.

using the loop index to access a vector is always legal!

A second thing to consider is that we parallelized on the skipping number k. In fact, there really are two loops surrounding the calculation; the other loop involves the number of trials. The fact that we buried the calculation inside a function obscures this detail, which might have led us to consider reordering the loops, especially if n was small relative to the number of processors, or if we actually only wanted to check a few values of k, not the entire range possible.

Finally, note that the function *prob_estimat()* invokes the random number generator. In a sequential calculation, we should be comfortable with the idea that there is a single generator with an internal state, and that each call to this generator both produces a random value and advances the state. What happens in a parallel calculation? Does each worker invoke a separate random number generator? Does each random number generator begin in the same state? For some applications, this would cause unacceptable correlations instead of the desired random behavior. At the moment, we will only raise these issues, to encourage you to see that moving from a sequential to a parallel system means that sometimes you are probably only assuming that things work the way they should!

Chapter 4

THE PRIME SIEVE

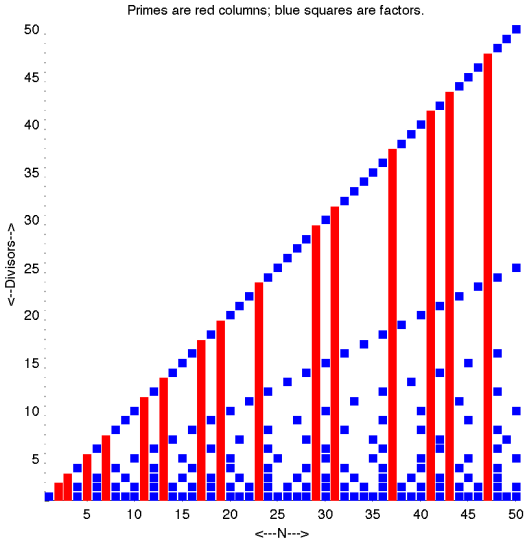

Figure 4.1: Primes (red columns) are scattered among the integers

4.1 Introduction

The Greeks were fascinated by the geometric properties of numbers; 1, 3, 6, 10 and so on were *triangular numbers*, just as 1, 4, 9 and 16 are the more familiar *squares*. They knew a number could be factored whenever

that many unit squares could be arranged into a rectangle. Some numbers wouldn't cooperate, though, and a number p was termed *prime* if the only rectangle that could be formed from p unit squares was the rectangle that was 1 unit by p units; which is the same as saying that p has no factors except 1 and p.

Prime numbers are a fundamental object in mathematics. One of the early achievements in mathematics was Euclid's proof that the number of primes is infinite. Gauss was just one of the mathematicians who tried to find patterns in the irregular jumps in the sequence of prime numbers. And even today, prime numbers arise repeatedly in mathematics (Gödel's incompleteness proof) and computer science (trapdoor functions for cryptoggraphy).

A very old algorithm exists for constructing a list of prime numbers. This method works quickly at first, but soon bogs down as the list increases in length. We will use this algorithm to consider the problem of *load balancing*. In parallel computing, an unbalanced problem consists of a number of tasks, some of which are much more time consuming than others. If an unbalanced problem is divided up naively among the available workers, it is possible that one unlucky worker gets all the work, while the others finish quickly and then are idle.

Unfortunately, the simplicity of the **parfor** command also implies that when things go wrong, it may not be easy for the user to think of an alternative approach. In our example, we will be able to predict the problem in advance (big numbers take longer to check), and we can even think of a way to "fool" **parfor** into a more balanced work division (but that's only because we can guess how it likes to divide the work.) For a more complicated problem that is unbalanced, the **parfor** command might not be the right tool.

So one purpose of this chapter is to present a simple example of bad load balancing, show how it might be detected, and then to raise the possibility that the proper treatment for a balancing problem may require moving to some of the other, more flexible, parallel tools that MATLAB makes available.

4.2 The Sieve of Eratosthenes

If we wish to determine a *list* of all the primes from 2 to n, the simplest approach is known as the "sieve of Eratosthenes". We create n boxes, each

representing a number. Then we cross out 1, because 1's technically not a prime. Since 2's not crossed out, that's our first prime, but we now cross out the numbers divisible by 2, namely 4, 6, 8, and so on. We move on to 3, which is not crossed out, so it's prime, but we now cross out 6, 9, 12, and so on. When we get to 4, it's crossed out, so we move on to 5, and so on. At the end, the boxes not crossed out represent primes. Computationally, we could create a logical array **prime(1:n)**, and then use the following loop (which is actually an implicit double loop):

```
1  function primes = prime_sieve ( n )
2  %% prime_sieve.m
3  %
4    prime = ones ( n, 1 );
5    prime(1) = 0;
6    for i = 2 : n
7      if ( prime(i) )
8        prime(2*i:i:n) = 0;
9      end
10   end
11 %
12 %  The primes index the nonzero entries of prime().
13 %
14   primes = find ( prime );
```

Another way to compute the **prime()** array is to consider each number i and check all its divisors:

```
1  function primes = prime_hunt ( n )
2  %% prime_hunt.m
3  %
4    prime = ones ( n, 1 );
5    prime(1) = 0;
6    for i = 2 : n
7      for j = 2 : i - 1
8        if ( mod ( i, j ) == 0 )
9          prime(i) = 0;
10         break;
11       end
12     end
13   end
14   primes = find ( prime );
```

Although both algorithms produce the same result, notice that in the second procedure, the value of **prime(i)** is determined once and for all on iteration i. In parallel programming, rapid and simple access to data can be

as important as computational aspects. Imagine several workers executing the loop for distinct values of i simultaneously. In the first procedure, it's possible that two workers will want to update the same entry of *prime*() at the same time, while in the second procedure this can never happen.

If we try to make a parallel version of the *prime_sieve()* function with a **parfor** statement, MATLAB complains immediately: *"Error: File: prime_sieve.m The variable prime in a parfor cannot be classified."* This message arises because one of the prerequisites for using the parfor command is that the output variable if it is an array, must be "sliced". That is, essentially, that MATLAB must be able to divide the *prime()* array into pieces uniquely associated with loop iterations. In this case, we would need for *prime(i)* to be modified by loop iteration i and no other. But that's precisely what the sieve algorithm cannot do, and so, at least in this form, it is not suitable for parallelization with parfor.

On the other hand, the *prime_hunt* algorithm carries out the same procedures, but in such a way that *prime(i)* is modified exactly by loop iteration i and no other. This matches the way parfor wants to work, and so for the remainder of this discussion, we will be working with that version of the algorithm. It should be emphasized, though, that when working with arrays and other indexed data structures, the parfor statement has some strict requirements on the way data is accessed. Even though the user may feel that a computation is perfectly parallelizable, these additional constraints must be satisfied or MATLAB will refuse to execute the code.

4.3 The Work Grows With N

Since we're going to be looking at large values of n, we really don't want to actually see a huge list of prime numbers. To make sure our program is actually working though, our modified function, called *prime_total_fun()*, will print the total number of primes it finds.

Another modification you will notice is that the loop going from 2 to $i-1$ now goes from 2 to $\sqrt{(n)}$. This is purely a practical matter; the old program's work increases like n^2 and as we start looking at numbers as large as 1,000,000 the program can become painfully slow. Lowering the inner loop limit means getting to 1,000,000 is about 1,000 times faster than before!

```
1 function total = prime_hunt2 ( n )
2 %% prime_hunt2.m
```

```
3  %
4  %  Modified version of prime_hunt.
5  %
6    total = 0;
7    parfor i = 2 : n
8      prime = 1;
9      for j = 2 : i - 1
10       if ( mod ( i, j ) == 0 )
11         prime = 0;
12         break
13       end
14     end
15     total = total + prime;
16   end
```

Our plan is to let n increase, compute the elapsed time for each computation, and try to understand how the work is growing with n. We can do this by calling the sequential version of the *prime_hunt_total* code for an increasing sequence of upper limits n. When we plot the timing data, we see a rise that suggests a quadratic growth in time (and presumably work) with n. A little thought makes this fact plausible. The work of loop iteration i involves checking all $i-2$ potential factors. So testing the numbers 2 through n means checking $0+1+2+...+(n-3)+(n-2)$ factors. The total number of factors checked is the sum of the integers from 0 to $n-2$, whose value is $\frac{1}{2}(n-2)(n-1)$ which means the work is growing roughly as n^2.

The quadratic growth in work with increasing n will have several implications.

4.4 Counting Primes in Parallel

The first implication of the quadratic growth in work is that over a relatively tight range in n we can generate problems that have a small or a heavy workload. We expect that the bigger problems will show parallelism to the most advantage. So now we will try to set up a parallel code, run it over a range of input parameters n and workers w, and see if we can exhibit a performance improvement of the kind we expect.

Let's state some of our automatic assumptions:

- We'll guess that, for small n, the sequential code is faster. (although we don't exactly know yet what "small" will mean!);

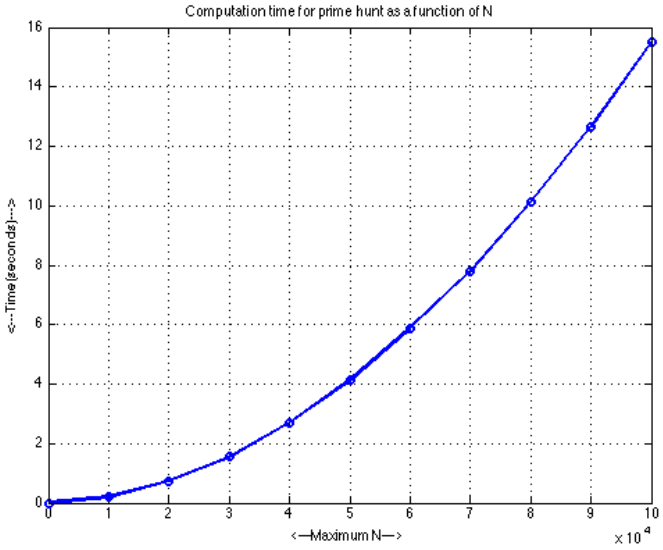

Figure 4.2: Computing primes from 1 to N gets quadratically harder

- We'll guess that using 1 worker in parallel is almost the same as using no workers, sequentially.
- We'll guess that, as the number of workers increases, the time goes down in a simple way. Two workers, twice as fast, ten workers, ten times as fast.

```
1  function total = prime_total_fun ( n )
2  %% prime_total_fun.m
3  %
4  %  Parallel version of prime_hunt2 with reduced inner loop limit
       .
5  %
6    total = 0;
7    parfor i = 2 : n
8      prime = 1;
9      parfor j = 2 : sqrt ( i )
10       if ( mod ( i, j ) == 0 )
11         prime = 0;
12         break
13       end
14     end
15     total = total + prime;
```

```
16    end
```

Chapter 5

HAILSTONE WITH PARFOR

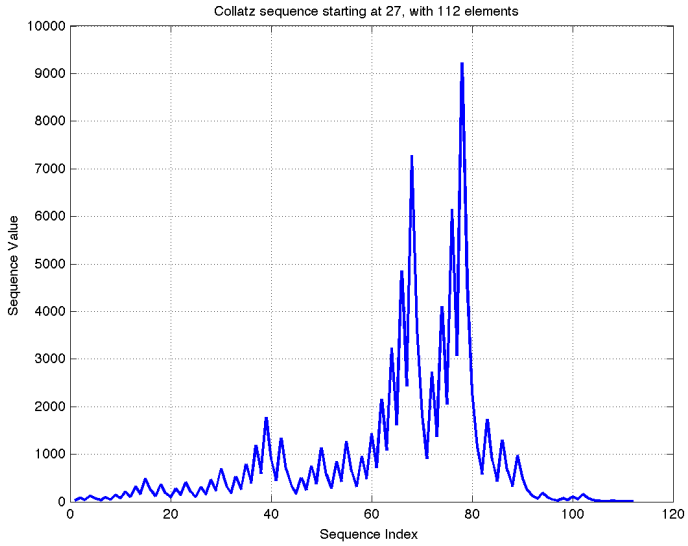

Figure 5.1: Collatz sequence for 27

5.1 Introduction

Pick a number. If it's even, divide it by two, but if it's odd, triple it and add 1. If you repeat the process, you get a sequence of numbers which seem to pop up and plummet down unpredictably, the way a hailstone forms by repeatedly

rising and falling within a thundercloud. For this reason, the numbers are sometimes called the "hailstone sequence", as well as the "3n+1" or "Collatz sequence", and they have been a subject of interest to mathematicians and computational scientists for years.

If we start with 7, for instance, we will compute the sequence

$$7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, \ldots$$

which begins to cycle when it reaches 1. Although this process seems to reach 1 no matter what starting point is used, this is still an open question known as the Collatz conjecture. We'll assume the sequence always terminates at 1. For a given starting value n, then, two natural quantities of interest are $l(n)$, the length of the sequence, and $h(n)$, the "height" or maximum value, occurring in the sequence.

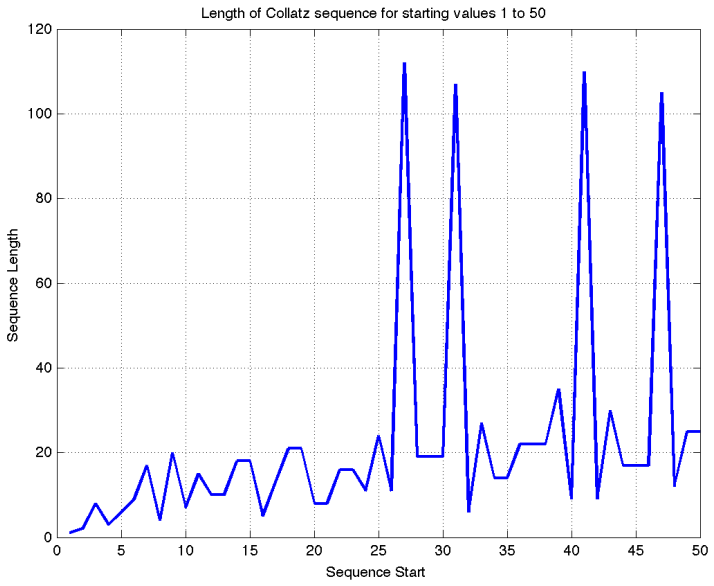

Figure 5.2: Collatz sequence lengths for 1 to 50

5.2 Computing the Length and Height

It's remarkably hard to tell in advance whether the Collatz sequence of a number will reach 1 quickly or not. Thus, if we start with 27, we end up

taking 111 steps before reaching 1, whereas starting at 26 and 28 take 11 and 19 steps respectively. It's easy to write a program that can compute $l(n)$ for any value of n we choose:

```
function l = collatz_length ( n )
%% collatz_length.m
%
  l = 1;
  while ( n ~= 1 )
    if ( n % 2 == 0 )
      n = n / 2;
    else
      n = 3 * n + 1;
    end
    l = l + 1;
  end
```

```
function h = collatz_height ( n )
%% collatz_height.m
%
  h = n;
  while ( n ~= 1 )
    if ( n % 2 == 0 )
      n = n / 2;
    else
      n = 3 * n + 1;
    end
    h = max ( h, n );
  end
```

Chapter 6

THE MOLECULAR DYNAMICS PROBLEM

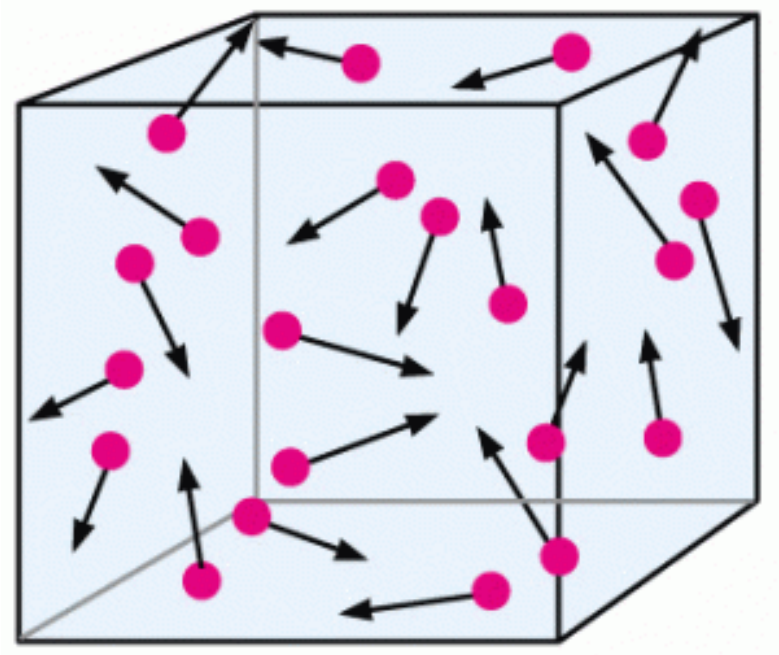

Figure 6.1: Particles with Momentum and Weak Attraction

6.1 Introduction

A great deal is known about the structure and behavior of atoms and molecules, as long as we consider them one at a time. Some properties of a material can be explained directly by corresponding features that can be noticed in a single atom. But there are other phenomena, such as phase transitions, the formation of surfaces, the effect of catalysts, and various mixing effects, whose understanding can only arise from considering large groups of molecules having a typical range of energies, and interacting by collisions and forces of attraction or repulsion.

The field of molecular dynamics provides a simulated laboratory for such studies. Instead of atoms or molecules, we consider a collection of abstract *particles*, which don't react chemically. We think of them as point masses which move though space, driven by their momentum. A pair of particles exert a mutual repulsive force when close, and a weaker attractive force when further apart.

A computation might involve hundreds of thousands of particles. The strategy requires creating a series of "snapshots" of the particle positions over time. However, to achieve accurate results over a time interval of interest, a huge number of snapshots must be taken. To create the next snapshot requires selecting each particle, measuring its distance to all other particles, summing up the resultant forces, and then using Newton's law to move the particle a tiny distance. The same process must be carried out for each of the particles before the snapshot can be completed. If we have n particles to study, then we have roughly n^2 pairs of molecular interactions to sum up on a snapshot. Thus, a molecular dynamics code can provide a problem with quadratic growth; as n increases, our problem quickly grows from difficult to intractable. (For this very reason, sophisticated molecular dynamics codes cut down the work by concentrating on the local molecular interactions, while approximating the weaker effect of far away molecules.)

Packages for doing heavy duty molecular dynamics simulation include AMBER, CHARMM, DL_POLY, Gromacs, and NAMD.

6.2 The Calculation

Suppose that we have n particles, and that at some starting time t_0 we are given, for each particle i, its mass m_i, and the initial position, velocity,

and acceleration vectors $p_i(t), v_i(t), a_i(t)$. Our goal is to model the behavior of the particles up to some final time $t*$ by measuring the forces on the particles, and allowing them to respond over a short time step. If our steps are small enough and our approximation scheme accurate, then we can get reliable estimates of the behavior of the particle system over the time range of interest.

In order to determine what is going to happen at time step $t + dt$, we need to compute the force vector f. In the molecular dynamics model, the force on particle i is the sum of the individual forces exerted by all the other particles in the system. The force $f_{i,j}$ exerted on particle i by particle j depends only on the distance $d_{i,j} = ||p_i - p_j||$ between the two particles, and its value is $f_{i,j} = V'(d_{i,j})$, that is, the derivative of the potential function. This step is carried out in the function *compute_new()*.

The potential function used in the *md()* program is the sine-squared potential $V(d) = \sin^2(\min(d, \frac{\pi}{2}))$, which is completely zero beyond a distance of $\frac{\pi}{2}$, and attractive at closer range.

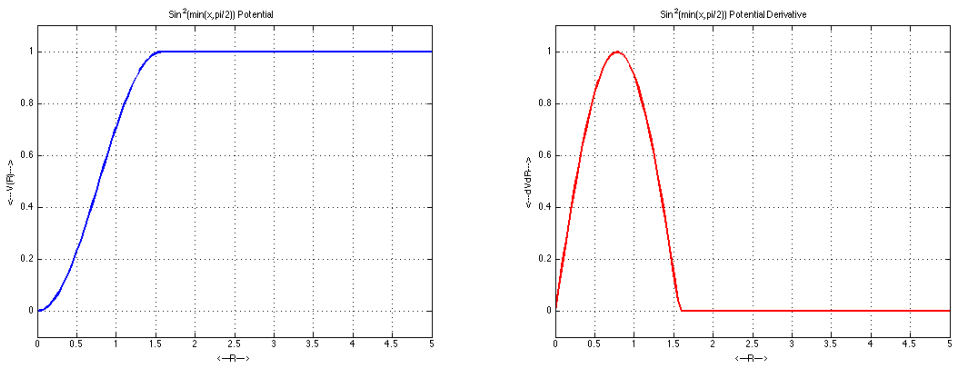

Figure 6.2: The $\sin^2$ Potential V(d) and Derivative V'(d)

(A more realistic potential function is known as the Lennard-Jones potential. Particles repel each other when close because of the Pauli principle, and attract at a distance because of the van der Waals force.)

Once the force vector f has been computed, it is possible to determine the position, velocity and acceleration vectors for snapshot at time $t + dt$ using the values known at time t. One technique for doing this is known as

the *Velocity Verlet* algorithm:

$$\begin{aligned} p(t+dt) =& p(t) + v(t)dt + \frac{1}{2}dt^2 \\ a(t+dt) =& f(t)/m \\ v(t+dt) =& v(t) + \frac{1}{2}(a(t) + a(t+dt)) \end{aligned}$$

This step is carried out in the function *update()*.

Once the new position, velocity and acceleration have been computed, a new snapshot is complete. The program can gather statistics from this new data, such as the pressure, temperature, kinetic and potential energy, before beginning the computation of the next step.

6.3 Estimating the Work

A typical run of our particular program might involve 1,000 particles for 500 time steps. This seems to mean that a million computations of some sort will be involved, which suggests our program is not going to run in the blink of an eye! In fact, if we are a little more careful, we can make a reasonable estimate of how the work and time will grow with the values of np, the number of particles, and $step\_num$ the number of steps.

Assume we are using np particles, and taking $step_num$ time steps. The basic calculation in the code computes the total force on one particle p_i at the k-th time. This requires summing up the individual force from each particle p_j. If it costs c units of work to calculate an individual force, then it will cost $(np-1)*c$ units to get the total force on particle p_i, and thus $np*(np-1)*c \approx np^2*c$ units to compute the total force on all the particles, at the k-th time. To do this for each time step will cost us a total work of about $step\_num * np^2$ units. This means that the problem is linear in time steps, but *quadratic* in the number of particles. You should be warned that problem work depends quadratically on a parameter, that the work can very quickly explode beyond your computer's capacity.

If we really believe that the work is linear in time, then we can estimate the work involved in a 1,000 particle and 500 timestep run by using **tic** and **toc** to time 1 step of a 1,000 particle run, and multiplying by 500. While we're at it, we can do this experiment for a range of particle sizes as well.

NP	1 step time	500 step estimate	1,000 step estimate
1,000	1s	8m	17m
2,000	4s	33m	1h 6m
4,000	14s	1h 57m	3h 53m
8,000	58s	8h 3m	16h 6m
16,000	3m 51s	1d 8h	2d 16h

Our estimates are a warning that this program can eat up a lot of time, especially if we try to increase the particle size. This suggests that it would be worth our trouble to try to rewrite the code in a way that speeds it up. It's hard to be certain about what to try; before changing our code, the next thing we should do is find out what lines of code are being used the most. This is where our work is being done, and this is where we should concentrate our efforts.

Typical calculations with a production molecular dynamics code might involve 20,000 to 3,000,000 particles and take 10,000 to 50,000 time steps! The problem of controlling the computational cost becomes so serious that special algorithms have been developed to cut down on the quadratic cost of the particle count parameter.

6.4 Profiling the Sequential Code

The molecular dynamics calculation that we are using as an example is unlike most of the other example programs we have encountered so far. It's longer, made up of multiple functions, and involves loops that look complicated. That's why it is not easy to point to any particular line of the program and assume that speeding up that line will speed up the entire program. We often aren't able to form a realistic assessment of the importance of the individual portions of our code.

In practice, however, it is often the case that, when a large problem is being solve, most of the computational work occurs in just a few program locations; If we can locate those locations, and try to improve their operation, we can achieve a huge improvement in performance. And, conversely, it is often the case that programmers waste time optimizing a portion of their program that contributes almost nothing to the overall cost. Since we only have a certain amount of time to spend modifying and testing the code, it is wise to identify those sections containing most of the work, and then concentrate on them alone.

Thus, we propose to run the program sequentially, using a realistic problem size, and try to spot the busiest loops, where optimization might help. Fortunately, MATLAB comes with a very useful tool for this purpose, the *program profiler*. To use the program profiler for the **md** program, we simply type the following three commands:

```
1 profile on
2 md
3 profile viewer
```

When the program is complete, the command **profile viewer** opens up a report on the program which lists the functions that were called (both user and internal MATLAB functions), the number of times called, the total time spent in the function or its subfunctions, and the "self time" spent precisely in that function itself.

Now it's normal for the program to take somewhat longer to run if the profiler is on, because When the profiler is turned on, MATLAB is doing more work that just executing your program; it also keeps track of which statement is begin carried out, and how long it took. Typically, this adds a noticeable amount of time to the original running time. However, there's something about the **md** program that makes the profiled version much, much slower. Running with 1,000 particles and 100 steps, the program took about 90 seconds to run, but with the profiler turned on, the program used more than 5,000 seconds. This is surprising, but it's actually an indirect indication that there's something wrong with the program. We won't try to explain the time discrepancy further, and will concentrate on the report itself.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
md	1	5158.553 s	0.020 s	
md>compute	101	5158.451 s	5158.451 s	
md>timestamp	2	0.030 s	-0.000 s	
md>update	100	0.030 s	0.030 s	
datestr	2	0.020 s	0.010 s	
rng	1	0.020 s	0.000 s	

Figure 6.3: A portion of the profile report for **md**

Figure 6.3 displays the initial part of the profile report for the **md** program, and indeed **md** shows up on the first line, called once (when we invoked it from the command line), and with a total time equal to the the program running time. However, the self time is negligible - that is, as soon as we call **md**, **md** is calling other functions, and that is where the work is really going on.

Moving to the second line of the report, we see the item **md>compute**, meaning that we are reporting on time spent in the **compute** function, called by **md**. This is called 101 times (once for start up, and once for each of 100 steps). The total time is again very large, but now notice that the self time is about the same. In other words, we've found where the work is actually being carried out. Although **md** includes functions **initialize()** and **update()**, the report indicates that their contribution to the total cost is negligible.

Clearly, our concern should focus on bfcompute(). Fortunately, the profiler can give us more detailed information of what is going on in that function. We simply click on the **md>compute** line of the report, and we are transferred to a document that analyzes the **compute()** function in greater detail (Figure 6.4).

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
269	`f(k,i) = f(k,i) - rij(k) * sin...`	302697000	585.483 s	11.3%	
249	`rij(k) = pos(k,i) - pos(k,j);`	302697000	563.304 s	10.9%	
250	`end`	302697000	544.555 s	10.6%	
254	`d = d + rij(k)^2;`	302697000	543.763 s	10.5%	
270	`end`	302697000	542.667 s	10.5%	
All other lines			2378.679 s	46.1%	
Totals			5158.451 s	100%	

Figure 6.4: A portion of the profile report for **md>compute**

Now we see individual lines of the code labeled with the number of times they were called, and their contributions to the overall time. From this report, it is clear that 50% of the total program time is spent executing 5 lines of code (really just 3 lines, since the other two lines are **end** statements).

```
1   f = zeros ( nd, np );
2   pot = 0.0;
```

```
  for i = 1 : np
    for j = 1 : np
      if ( i ~= j )
        for k = 1 : nd
          rij(k) = pos(k,i) - pos(k,j);
        end
        d = 0.0;
        for k = 1 : nd
          d = d + rij(k)^2;
        end
        d = sqrt ( d );
        d2 = min ( d, pi / 2.0 );
        pot = pot + 0.5 * sin ( d2 ) * sin ( d2 );
        for k = 1 : nd
          f(k,i) = f(k,i) - rij(k) * sin ( 2.0 * d2 ) / d;
        end
      end
    end
  end
  kin = 0.0;
  for k = 1 : nd
    for j = 1 : np
      kin = kin + vel(k,j)^2;
    end
  end
  kin = 0.5 * mass * kin;
```

At this point, we need to get advice from someone familiar with MATLAB optimization about whether we can improve the function. However, you should be aware that MATLAB is not efficient when asked to carry out **for** loops that don't actually contain a lot of work, and are called many times. If possible, such loops should be replaced by equivalent vector operations, which MATLAB can execute much more quickly.

On the advice of an experienced MATLAB code, the **compute()** function was revised. The new function was called **compute_new()**, and it was put inside a new program called **md_new()**. The core of the new function is dramatically changed:

```
  f = zeros ( nd, np );
  pot = 0.0;
  pi2 = pi / 2.0;
  for i = 1 : np
    Ri = pos - repmat ( pos( :, i ), 1, np );
    D = sqrt ( sum ( Ri.^2 ) );
```

```
7       Ri = Ri( :, ( D > 0.0 ) );
8       D = D( D > 0.0 );
9       D2 = D .* ( D <= pi2 ) + pi2 * ( D > pi2 );
10      pot = pot + 0.5 * sum ( sin ( D2 ).^2 );
11      f( :, i) = Ri * ( sin ( 2*D2 ) ./ D )';
12    end
13    kin = 0.5 * mass * sum ( diag ( vel' * vel ) );
```

Now let us redo the simple timing estimate with the revised code to see if the changes had any effect:

NP	1 step time	500 step estimate	1,000 step estimate
1,000	1/5s	1m40	3m 20s
2,000	2/3s	5m33	11m 6s
4,000	2s	16m 40s	33m 20s
8,000	8s	1h 6m	2h 13m
16,000	38s	5h 16m	10h 33m

After seriously reworking the code, we've managed to get it to run about 4 times faster on the larger problems. In one sense, this is wonderful. In another sense, it's unfortunate because after working so hard on the **compute** function, it's really hard to see how we can squeeze out any more speed, short of rethinking the algorithm.

6.5 Performance of the Parallel Code

If we want to run bigger problems, or get our results faster, then the natural thing to consider is to carry out the calculations in parallel. If we run the profiler on the revised **md_new**, we will still observe that the **compute_new** function accounts for almost all the work. Luckily, the work is contained inside a **for** loop that carries out the force calculation for each particle. These force calculations are independent. This means we can set up a parallel version of the program quite simply.

```
1     f = zeros ( nd, np );
2     pot = 0.0;
3     pi2 = pi / 2.0;
4     parfor i = 1 : np
5       Ri = pos - repmat ( pos( :, i ), 1, np );
6       D = sqrt ( sum ( Ri.^2 ) );
7       Ri = Ri( :, ( D > 0.0 ) );
8       D = D( D > 0.0 );
9       D2 = D .* ( D <= pi2 ) + pi2 * ( D > pi2 );
```

```
10      pot = pot + 0.5 * sum ( sin ( D2 ).^2 );
11      f( :, i) = Ri * ( sin ( 2*D2 ) ./ D )';
12    end
13    kin = 0.5 * mass * sum ( diag ( vel' * vel ) );
```

If this code will execute in parallel, then if we decide to run a bigger, longer job, we

Chapter 7

TIME, WORK, RATE

Figure 7.1: Distance / Time = Rate

7.1 Introduction

We use a computer because we want an answer. We use parallel programming because we want that answer *faster*. So we have to assume we already know how to compute the answer, and we are looking for ways to speed up the computation. There are many things we can try, but we need to have a reliable method of measuring whether a revised program really does run faster. This means we need to be understand how to measure computational work, time, and rate.

7.2 Absolute Computational Work

When Charles Babbage was designing his Analytical Engine in 1837, he used the word "mill" to describe the part of the machine that carried out operations on numbers. Even today, it is sometimes useful to think of a computer as a kind of machine that takes a bucket of raw numbers as input, carries out some kind of work on them, and produces a bucket of output numbers. We can actually take this simplified model and make some rough approximations of the amount of work involved.

As long as we're ruthlessly simplifying, we'll assume that we're only dealing with a floating point numbers, and that the task we have to carry out can always be arranged so that it involves nothing more than the four basic arithmetic operations of addition/subtraction, and multiplication/division. The amount of work, called the *operation count* can then be determined by looking at the code and counting the floating point operations.

To get an example of such an operation, consider the process of Gauss elimination when applied to a matrix. On step k, we determine the pivot row, move it into the k-th row of the matrix, and then add a multiple of row k to each row i from $k+1$ to n in such a way that we zero out the entry $a(i,k)$. This is an example of a "saxpy" operation, symbolized by $x \leftarrow x + s * y$, where we multiply a vector y by a scalar s and add it to a vector x.

MATLAB allows us to write the code for such an operation in the compact form

```
1  x = x + s * y;
```

but for clarity, let's write it out:

```
1  for i = 1 : n
2    x(i) = x(i) + s * y(i);
3  end
```

We see that the operation count for the saxpy operation is $2n$.

Similarly, it is not difficult to see that the operation count for a dot product of two vectors is also $2n$, while multiplying an $m \times n$ matrix A times an n-vector x is $2mn$. Counting the work in Gaussian elimination is more difficult because at each step, the length and number of the vectors involved decreases; the operation count is on the order of $\frac{2n^3}{3} + \frac{3n^2}{2} - \frac{7n}{6}$ and the value $\frac{2n^3}{3}$ is generally an acceptable estimate.

7.3 Wallclock and Processor Times

We use parallel programming because it has the potential to get our results faster. If we're running a program interactively, then we might be able to guess when a program runs faster; for a better measurement, we could look at a clock. However, since computers can do billions of operations in one second, we'll really need a more accurate and automatic procedure for measuring time if we want to judge performance rates.

We know that a computer has an internal clock that controls the pace of operations. We'll make the assumption that each "tick" of the computer's clock measures a time period during which one operation can be carried out. To measure the amount of (human) time that elapse during a computation, the computer can simply count the number of ticks between the start and finish, and multiply by a conversion value that records the length of a single tick in seconds. This measurement is called the *wallclock time*, because it represents a very accurate value for time elapsed from the start to finish of our program, and it's the quantity that we hope to reduce by converting to parallel programming.

You may have heard of another time measurement that can be used in computing, which was once called *CPU time*, or now, *processor time*. To explain this, we must first mention that a computer is typically running many programs, not just your MATLAB job. If the computer has a single processor, then it is responsible for executing all these programs. It does this by a sort of time sharing, in which each program is brought into the processor, executed for a fixed slice of time, then temporarily frozen so that the next program can be brought in. Under such a system, it would be unwise to use the wallclock time to measure performance, since the program we are interested in would actually only have been executing for a portion of that time. Hence, methods were developed that allowed a program to request a separate report that only counted the time during which the program was actually being run by the processor.

Even in the old days, if you were running a large computation, the difference between wallclock and CPU time was usually not outrageous; moreover, as long as you were running on a desktop computer that you controlled, you could make sure to turn off as many other unnecessary programs as you could, so that the processor could focus on the program of interest.

Now, when we are trying to compare sequential and parallel programs, the CPU time is of much less interest. In fact, we'd expect that the parallel

program would always run up at least as much processor time as the sequential program, since it's doing the same work. We're far more interested in whether or not that work was completed in a shorter time interval, because it was properly divided up among several processors.

7.4 Measuring Time

MATLAB includes the **tic** and **toc** functions which set a starting time and then return the time elapsed since that point:

```
1  tic;
2
3  combination_lock
4
5  elapsed_time = toc;
```

The same command can be used in parallel. The **matlabpool** command itself can take some time. Typically, we want to ignore that, and focus on the parallel computation, so the timing calls should go *inside* the calls to **matlabpool**:

```
1  matlabpool open local 4
2
3  tic;
4
5  combination_lock_parallel
6
7  elapsed_time = toc;
8
9  matlabpool close
```

The value returned by **toc** is measured in seconds, and has an accuracy that can be as good as one-millionth of a second; this varies from one computer system to another, and you can find out your computer's resolution using the following simple program:

```
1  tic;
2  elapsed_time = toc;
3  fprintf ( 1, 'The tic/toc resolution on this computer is %g\n'
       , elapsed_time );
4  fprintf ( 1, 'The number of tic/toc ticks is about %d\n', 1 /
       elapsed_time );
```

7.5 A Test Drive on Your Computer

If I look up the specifications for my computer, I find out that I have a 2.8GHz Quad-Core processor. This means I have four processors (that's good, I can do parallel processing), and that each processor runs with a clock speed of $R = 2.8$ GigaHertz, that is, it "ticks" 2.8 billion times a second.

When I run the `tick_tock_resolution.m` code on this computer, I get a tic/toc resolution of about 1.4 millionths of a second, and about 680,000 tic/toc ticks per second; in other words, tic/toc is pretty good, but the shortest event it can measure lasts 4 ticks on my computer's clock. Since we will usually be interested in measure large computations with thousands or millions of operations, tic and toc will be accurate enough for us.

Now let's assume that I am running a sequential MATLAB program, and that the amount of work involved is w floating point operations. If the processor could really do exactly one operation per clock tick, and it had nothing to do but my explicit instructions, then it will be interesting to compute the computational rate $r = \frac{w \text{ (floating point operations)}}{s \text{ (seconds)}}$ and compare it to R.

```
%% dot_product_timing
  x = rand ( n, 1 );
  y = rand ( n, 1 );

  tic;
  z = 0.0;
  for i = 1 : n
    z = z + x(i) * y(i);
  end
  t = toc;

  r = ( 2 * n ) / t;
  R = 2800000000;
  fprintf ( 1, '%d  %g  %g\n', n, r, R );
```

Our script leaves the value n as a variable. We will run the script for values of n that are powers of 2, from 2^0 to 2^{25}, and keep track of the results. Once the problem size n gets big enough, the plot for the computational time seems to follow a reasonable linear pattern, which suggests that the time measurements made by **tic** and **toc** are properly related to the work being carried out.

When we plot the computational rate, we see a "ramping up" effect; that

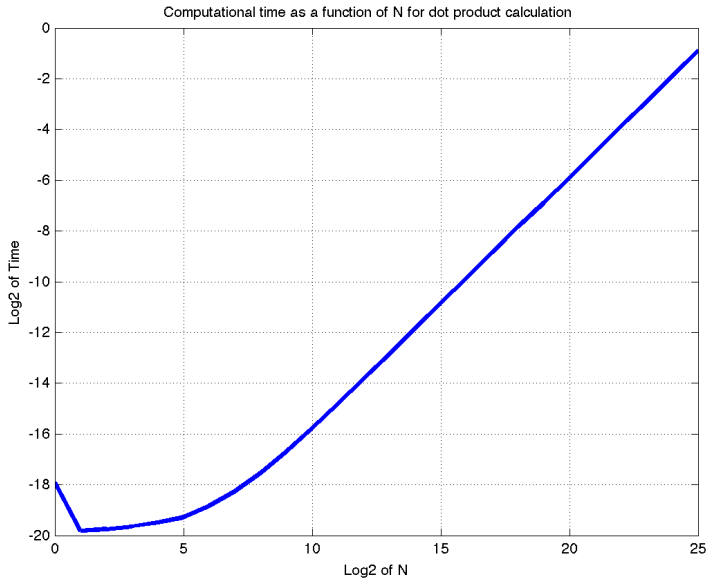

Figure 7.2: Computational Time for the Dot Product

is, over a significant initial range of values of n, the rate rises somewhat steeply, and then levels off to a plateau, at which we might assume the computation is proceeding at its maximum possible rate.

7.6 Relative Computational Work

Unlike the dot product, most computations are not simple enough that we can essentially estimate the number of floating point operations. This could be because the computation is complicated, involves many conditional statements, or invokes special functions whose evaluation requires an iteration of an unspecified number of steps. Estimating the number of floating point operations allows us to measure whether we are using the computer efficiently; but even if we can't do that, we can still using the idea of timing our code to be able to judge whether one algorithm is faster than another, or how rapidly the work must be growing as the problem size increases. In this case, we are using the idea of relative comparison. We don't know how long it takes to evaluate the sine function once, but we can expect that it takes ten times longer to evaluate 10 sine functions. And we might compare the speed

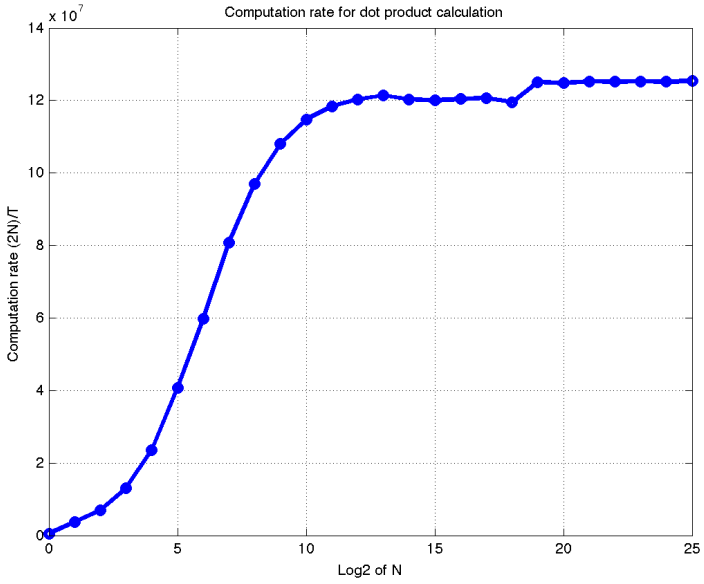

Figure 7.3: Computational Rate for the Dot Product

of evaluation of the sine function using MATLAB's built-in *sin()* function to the speed where we used a Taylor series.

Suppose we need to fill an array *a(,)* with products of sines and exponentials, according to the following scheme:

```
function a = sin_exp1 ( m, n )
%% sin_exp1.m
  a = zeros ( m, n );
  for i = 1 : m
    for j = 1 : n
      a(i,j) = sin ( i * pi / m ) * exp ( j * pi / n );
    end
  end
```

We don't know how long this process will take, and it will certainly depend on the values m and n. However, we may realize that the evaluation of *sin()* and *exp()* are expensive operations, and we notice that we're calling these expensive functions a lot, mn times in fact. Since we really only need to know the values of m sines and n exponentials, we could precompute them in vectors u and v, and then hope to save time, at least if the problem size is large.

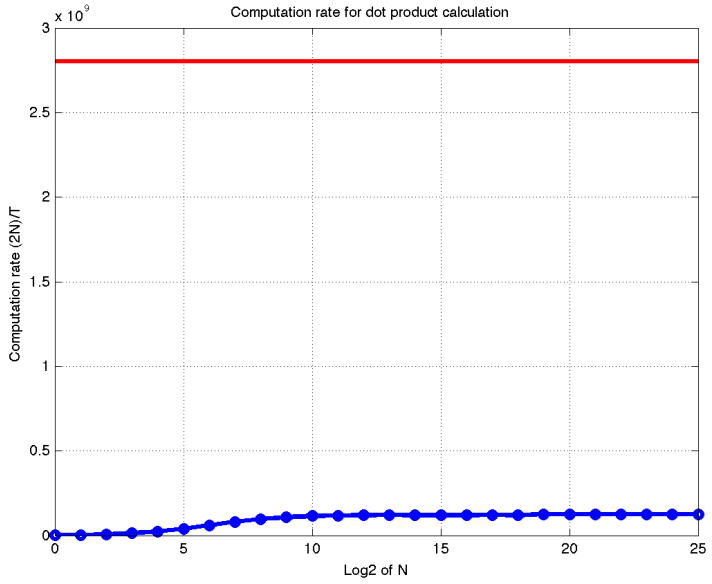

Figure 7.4: Comparison of Dot Product Rate to Maximum

```
function a = sin_exp2 ( m, n )
%% sin_exp2.m
  u = zeros ( m, 1 );
  for i = 1 : m
    u(i) = sin ( i * pi / m );
  end
  v = zeros ( n, 1 );
  for j = 1 : n
    v(j) = exp ( j * pi / n );
  end
  for i = 1 : m
    for j = 1 : n
      a(i,j) = u(i) * v(j);
    end
  end
```

Looking more carefully at the calculation of the result *a(,)*, we might notice that it has the form of a vector "outer product", that is, the product of two vectors that creates an array as the result. MATLAB knows about outer products just like it knows about inner products, and can evaluate

them rapidly. So we might try to speed up our calculation by writing it in the language of vectors:

```
function a = sin_exp3 ( m, n )
%% sin_exp3.m
  u = sin ( ( 1 : m ) * pi / m );
  v = exp ( ( 1 : n ) * pi / n );
  a = u' * v;
```

We know how to time these calculations:

```
  nv = zeros ( 12, 1 );
  t1 = zeros ( 12, 1 );
  t2 = zeros ( 12, 1 );
  t3 = zeros ( 12, 1 );
  m = 1000;
  n = 1;
  for j = 1 : 12
    nv(j) = n;
    tic
    a = sin_exp1 ( m, n );
    t1(j) = toc;
    tic
    b = sin_exp2 ( m, n );
    t2(j) = toc;
    tic
    c = sin_exp3 ( m, n );
    t3(j) = toc;
    n = n * 2;
  end
```

For this study, the value of n doubles at each successive measurement point. That suggests that if we plot the data, we should use a logarithmic scale for n; otherwise, much of the data will be bunched up on the left hand side and less visible.

Now the actual times are not of interest to us. It means much more to be able to say that algorithm 3 took twice or a tenth as much time as algorithm 1. Assuming that algorithm 1 is our base for comparison, we can make a plot of the ratio between the time a given algorithm took and the time algorithm 1 required, which is a relative time comparison.

Although the relative time plot is easier to understand, it's hard to avoid the psychological reaction that the graph rising the highest represents the best result. But we can avoid that misinterpretation by plotting the relative *rate* or *speed* rather than time; that is, we plot the time algorithm 1 required

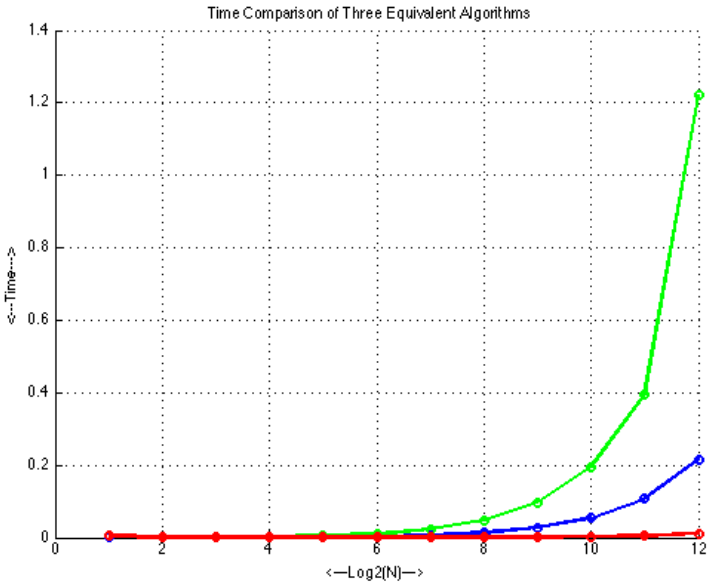

Figure 7.5: Absolute Time Comparison for Three Equivalent Algorithms

divided by the time the given algorithm required. Notice that, at least for this example, our decision to display relative speeds means that the results for algorithm 3 (the winner in this particular competition) are shown with greater detail, whereas the slower algorithm is much less prominent.

Recall that the computation rate was defined by $r = w/t$, where w was the work, and t the time. We ran algorithms 1, 2, and 3 and got timings $t1$, $t2$ and $t3$, but we weren't able to estimate w, so we can't compute a true (absolute) rate. However, the relative speed is the *ratio* of the rates, and in that case, w is a common factor, and drops out, so it doesn't matter if we don't actually know its value:

$$\frac{r3}{r1} = \frac{w/t3}{w/t1} = \frac{t1}{t3}$$

which justifies our natural assumption that, when judging speeds, even if we can't say whether either of two programs is actually fast, we should always prefer the one that is faster than the other!

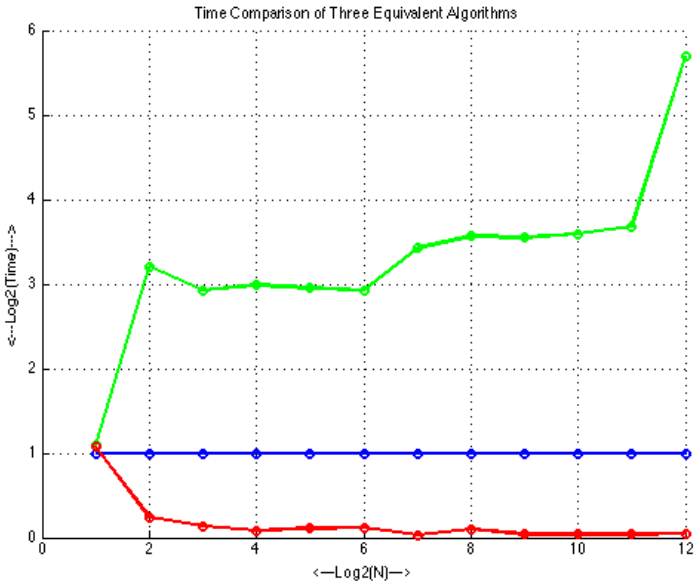

Figure 7.6: Relative Time Comparison for Three Equivalent Algorithms

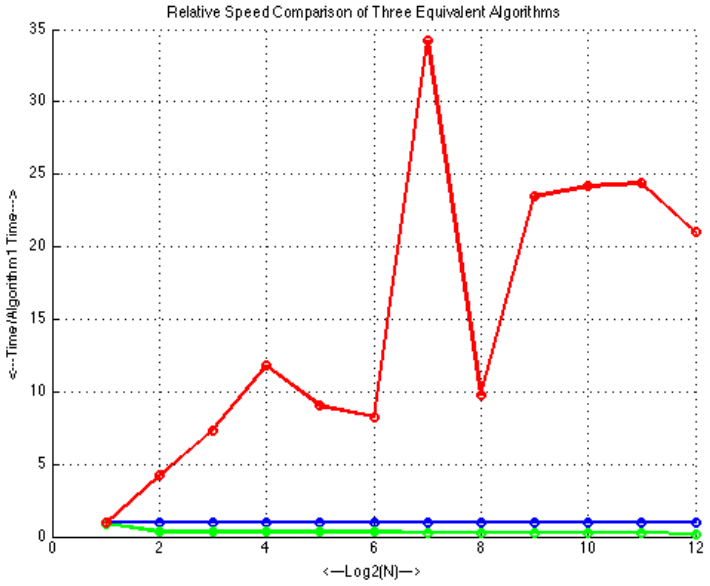

Figure 7.7: Relative Speed Comparison for Three Equivalent Algorithms

Chapter 8

RULES OF THE (PARFOR) ROAD

Figure 8.1: Some Do's and Don'ts

8.1 Introduction

In Chapter 1 we saw a code that repeated a calculation sequence many times; splitting the work over several processors was straight forward. In the present Chapter we discuss the rules under which such work-sharing is permissible in MATLAB.

Many computer languages provide a **for** (or **do**) loop to designate a block of code (the loop-body) for repeated execution. In MATLAB a **for** loop

No.	Limitation
1	the range of a **parfor** must be increasing, consecutive integers
2	the results must not depend on the order of execution
3	restrictions on allowable statements in the loop-body

Table 8.1: Restrictions on **parfor** loops

requires an explicit *counter* or *loop-index*. The range of the loop can be a 2D array, in which case the loop index successively takes the values of the columns of the array. If the range is a 1D array, the loop values are scalars.

The *Parallel Computing Toolbox* provides a **parfor** loop (*parallel for loop*) that evaluates the loop-body statements in parallel on separate workers. These workers could be on your multi-core machine or on a cluster. The workers are anonymous in the sense that the base code can not explicitly communicate with the workers, nor can the workers communicate with each other. Most importantly, the order in which the loop-body is executed is not known.

8.2 The basics

Clearly, not all **for** loops can be converted to **parfor**. The implementation provided by the *Parallel Computing Toolbox* imposes limitations on loops eligible for **parfor** (see Table 8.1).

Whereas the limitation on the range of the loop-index is easily understood, the consequences of the (unknowable) execution order may be more subtle. Since there is no way to know the order in which the loop-index values occur, there's no way to know its final value. Thus, after exiting a **parfor** loop, the value of the loop-index is not defined. Additionally, in a sequential **for** loop the loop-body can control the actual execution through logical tests with **break** and/or **continue** statements. The statements **break** and **continue** can not appear in the loop-body of a **parfor** loop. To explain the 3^{rd} limitation, it's necessary to classify the variables in a **parfor** loop-body (see Table 8.2).

Classification	Description
Loop-index	Scalar integer
Sliced	An array whose segments are defined and/or appear in expression in the loop-body
Broadcast	Defined before the loop and used in expressions in the loop-body
Reduction	Accumulates a value over iterations of the loop-body, independent of iteration order
Temporary	Created in the loop-body, but not defined after exiting the loop; the variable is cleared after every loop iteration

Table 8.2: Classification of **parfor** loop-variables

As an introduction to this classification, consider the code displayed in Listing 8.1.

```
1  %%Parfor_01
2  % based on example in PCT User's Guide
3
4      a = 0;
5      c = pi;            % broadcast
6      z = 0;            % reduction
7      r = rand(1, 10);  % sliced input
8
9      parfor ii = 1:10 % ii is the loop-index,
10                      %  1:10 is the range
11
12         a = ii;       % temporary, not the previously defined 'a'
13
14         if ii < c
15             d = 2*a;  % 'd' is  temporary
16         else
17             d = 0;
18         end
19
20         z = z +d*ii;    % reduction
21
22         b(ii) = r(ii);% sliced output
23
24
25     end
```

Listing 8.1: Classification of variables

The variable a outside the loop is not related to the a inside the loop. The later is a *temporary* variable; it is assigned inside the loop and its scope is limited to the loop-body. It's bad practice to re-use the symbol and rely on the scoping rules. For one thing, changing **parfor** to **for** changes the behavior. The variable z is a reduction, based on the fact that the addition operation is both commutative and associative. r is a sliced input array; only the necessary elements are sent from the client to each worker. b is a sliced output variable; the results are sent from the workers and accumulated on the client.

8.3 More on the classification

Computer cycles associated with communication between the client and the workers can impose a significant computational burden on the use of **parfor**. The notion of *sliced* variables offers a mechanism for efficiency; only a portion of each *sliced* variable has to be sent between the client and a worker. To ease the work of deciding what information must be communicated the MATLAB developers have imposed limitations on *sliced* variables. These limitations can lead to great frustration for users (including the authors !).

8.3.1 *sliced variables*

Recall that in MATLAB an object can be indexed in three ways:

1. by *name* in a **structure** , *e.g.* $A.var\_1$,
2. by *braces* in a **cell**, *e.g.* $A\{\,1\,\}$,
3. by *parentheses* in an array, *e.g.* $A(\,1\,)$.

A variable in the loop-body is *sliced* if and only if it has all of the following characteristics:

- The first-level of indexing is either parentheses, ( ), or braces, { } .
- Within the first-level index, the list of indices is the same for all occurrences of the variable.
- In the list of indices for a *sliced* variable, precisely one index involves the loop-index, and this index must be one for the following forms: $\{ii, ii+k, ii-k, k+ii, k-ii\}$ where ii is the loop-index, and k is either a scalar integer, or a scalar *broadcast* variable, or a *colon* or **end**.

- The right-hand side assignment does not change the shape of the variable (, e.g. no [ ], or ‘ ’ which delete elements, or (**end**+1) which inserts an element.

8.3.2 broadcast variables

A *broadcast* variable is any variable other than the *loop-index* or a *sliced* variable that is not the object of an assignment in the loop-body. Before the loop is initiated all *broadcast* variables are sent to the workers. Clearly, one should be judicious in the use of such variables.

8.3.3 reduction variables

Loop independence as defined in item-2 of Table 8.1 requires that the *results* of a **parfor**-loop be independent of the loop execution order. One could impose a stronger notion; namely, that each pass through the loop is independent of the others. Such a strong limitation would rule out many useful cases. For example, the variable z on line 20 of Listing 8.1 appears both in an assignment, and in an expression (on the right side of an assignment). In a **for** loop the initial value (on line 6, $z = 0$) is set and the resulting sum is accumulated. In a **parfor** loop the value $z = 0$ outside the loop is not sent to any worker. Each worker accumulates its partial sum; these are sent to the client where the final sum is accumulated. However, since real addition is associative, the result from executing the loop is independent of the order. Assignments of expressions that are associative is the defining characteristic of a *Reduction* variable. The astute reader will note that *floating point* addition is not associative, so that results from **for** and **parfor** loops need not be identical. Note that since the *reduction* variable only assumes a **for**-loop consistent value at loop-exit, it can be used reliably only in the *reduction* assignment statement.

To discuss *reduction* variables in more detail, it's convenient to abstract the replacement operation as a function evaluation, that is:

$$X = f(X, \text{expr}) \quad \text{or} \quad X = f(\text{expr}, X) \ .$$

On line 20 of Listing 8.1 we have $f \sim +$ and $\text{expr} \sim d * ii$, and in this case f is commutative, so that each worker can have an arbitrary collection of values for the loop-index, and the client can accumulate the results from

the workers in any order. The MATLAB developers have included several useful exceptions to the requirement that f be commutative. In a reduction assignment the following built-in functions are permitted:

1. matrix multiplication $X * \text{expr}$ or $\text{expr} * X$
2. column concatenation in an array$[X, \text{expr}]$ or $[\text{expr}, X]$
3. row concatenation in an array$[X; \text{expr}]$ or $[\text{expr}; X]$
4. column concatenation in a **cell** $\{X, \text{expr}\}$ or $\{\text{expr}, X\}$
5. row concatenation in a **cell** $\{X; \text{expr}\}$ or $\{\text{expr}; X\}$.

Note that the result of **cell** concatenation will always be either a 1×2 or a 2×1 **cell** array. Any user supplied function in a reduction operation is assumed to be commutative.

8.3.4 temporary variables

A *temporary* variable is a variable that is the target of a non-indexed assignment and is not a *reduction* variable. In Listing 8.1 the variables a (line 12) and d (lines 15 & 17) are *temporary* variables.

8.4 Efficiency