# 2: Computer Performance

John Burkardt
Information Technology Department
Virginia Tech

..........

FDI Summer Track V:
Parallel Programming

..........

https://people.sc.fsu.edu/∼jburkardt/presentations/
performance_2008_vt.pdf

10-12 June 2008

# Computer Performance - Measurements

## Computer Performance

We need “yardsticks” to say

- this problem is harder;
- this computer is faster;
- this algorithm is better;
- this compiler produces more efficient programs;
- the change I made was helpful

## Computer Performance - Work in FLOP's

Our first yardstick will measure a computation as a sequence of *floating point operations* or **FLOPs**.

Our basic unit of work looks like this:

```
    X = X + Y * Z
```

For technical reasons, a multiply and an addition counts as 1 **FLOP**.

All numerical computations will be reduced to this scale.

## Computer Performance - Work in FLOP's

Cramer's rule to solve a 2x2 linear system

```
a * x + b * y = c
d * x + e * y = f
```

yields:

```
x = ( c * e - b * f ) / ( a * e - b * d )
y = ( a * f - c * d ) / ( a * e - b * d )
```

How many **FLOPs** will this cost?

## Computer Performance - Work in FLOP's

Evaluate polynomial at 500 equally spaced points.

$p(x) = x^4 + 10 * x^3 - 7 * x^2 + 55 * x - 109$

How many **FLOP**s?

Rewrite as:

$p(x) = (((x + 10) * x - 7) * x + 550) * x - 109$

Now how many **FLOP**s?

## Computer Performance - Work in FLOP's

Matrix-vector Multiplication:

$y = A * x$

$A$ is MxN matrix, $x$ is an N vector, $y$ is an M vector.

How many **FLOP**s to compute all of $y$?

## Computer Performance - Work in FLOP's

Solving an $NxN$ linear system for $x$, given $A$ and $b$:

$A * x = b$

- roughly $\frac{2}{3}N^3$ **FLOP**s for Gauss elimination;
- roughly $\frac{1}{2}N^2$ **FLOP**s for back substitution;

## Computer Performance - Time in Seconds

Our second measurement attempts to account for the **time** it takes to do the work.

There are two common measures:

- **Wallclock** or **Real** time, the time you waited
- **CPU time**, the time the CPU was working on your job.

If you are the only user, these two measures might be close.

## Computer Performance - Time in Seconds

On UNIX systems, you can get the time of a command:

**time** *command*

This command may return the elapsed time broken down into

- **real time**, the time that passed.
- **user time**, time spent directly on your command.
- **system time**, overhead associated with your command.

It is usually the case that system time $<<$ user time
and
system time $+$ user time $<=$ real time

## Computer Performance - Time in Seconds

If you are analyzing a program, it's usually not good enough to time the entire run. You probably need to know information at a finer level. Most computer languages can access wallclock and CPU time:

```
t1 = cputime ( );
x = very_big_calculation ( y );
t2 = cputime ( );
cpu_elapsed = t2 - t1;
```

## Computer Performance - Rate in MegaFLOPS

Our third measurement considers the **rate** at which our work is done. For some time, this rate was measured in **MegaFLOPS**, that is, "millions of FLOPs per Second".

As you will see, this particular rate is becoming obsolete.

## Computer Performance - Rate in MegaFLOPS

If we can estimate the work in **FLOP**s, we can get the time, and compute the MegaFLOPS rate for a calculation. Here, we multiply two NxN matrices:

```
flop = n * n * n;
t1 = cputime ( );
a = b * c;
t2 = cputime ( );
cpu_elapsed = t2 - t1;
megaflops = flop / ( 1000000 * cpu_elapsed );
```

## Computer Performance - The LINPACK Benchmark

For a system of $N$=1,000 equations, we need almost one billion operations. This is a respectable amount of work. Any computer (with enough memory) can solve this system.

But now we can ask an interesting question:

*How* **fast** *can a given computer solve a system of 1,000 equations?*

The answer is known as the **LINPACK Benchmark**.

## Computer Performance - The LINPACK Benchmark

The LINPACK benchmark program

- sets up a system of 1,000 equations with random coefficients,
- factors and solves the system using the LINPACK routines SGEFA/SGESL or DGEFA/DGESL (double precision),
- computes **FLOP**s, the amount of work,
- measures **S**, the elapsed CPU time,

The LINPACK BENCHMARK rating $R$, in **MegaFLOPS** is

$R = \frac{FLOP}{1000000 * S}$

## Computer Performance - The LINPACK Benchmark

The LINPACK benchmark is available in various languages.

The benchmark gives one way of comparing many things:

- computers
- languages
- compilers
- variations on linear algebra algorithms

We will also use it to measure the power of **parallelism**.

## Computer Performance - The LINPACK Benchmark

Table: Sample LINPACK Ratings

| Rating | Computer | Language | Comment |
|---|---|---|---|
| 108 | Apple G5 | C | used "rolled" loops |
| 184 | Apple G5 | C | used "unrolled" loops |
| 221 | Apple G5 | F77 | gfortran compiler |
| 227 | Apple G5 | Java | |
| 20 | Apple G5 | MATLAB | using "verbatim" LINPACK |
| 1857 | Apple G5 | MATLAB | using MATLAB "backslash" |

## Computer Performance - The LINPACK Benchmark

Table: Sample LINPACK Ratings

| Rating | Computer | Site | Processors |
|---|---|---|---|
| 12,250,000 | 2.3 GHz Apple | Virginia Tech, System X | 2,200 |
| 13,380,000 | Xeon 53xx | "A bank in Germany" | 2,640 |
| 42,390,000 | PowerEdge | Maui | 5,200 |
| 102,200,000 | Cray XT3 | "Red Storm", Sandia | 26,569 |
| 126,900,000 | SGI Altix | New Mexico | 14336 |
| 167,300,000 | BlueGeneP | FZ Juelich | 65,536 |
| 478,200,000 | BlueGeneL | DOE/NNSA/LLNL | 212,992 |

$$\begin{aligned} 1000\,\text{MegaFLOPS} &= 1\,\text{GigaFLOPS} \\ 1000\,\text{GigaFLOPS} &= 1\,\text{TeraFLOPS} \\ 1000\,\text{TeraFLOPS} &= 1\,\text{PetaFLOPS}\text{(not quite there yet!)} \end{aligned}$$

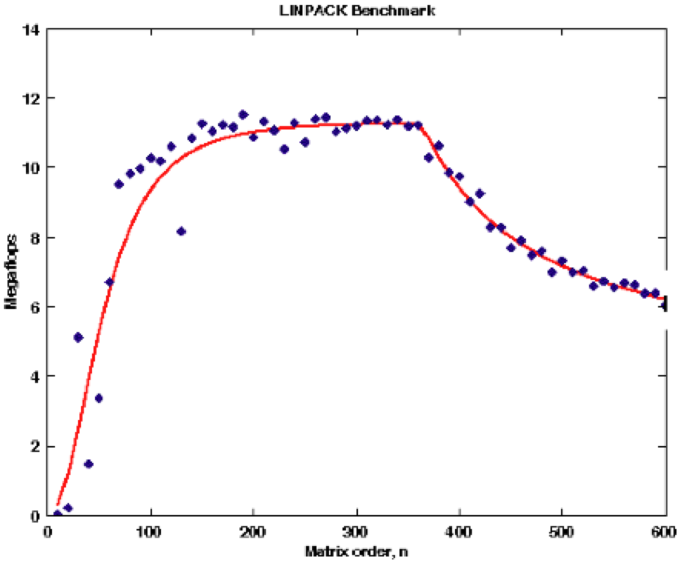
## Computer Performance - The LINPACK Benchmark

Even if you only have your own computer to experiment with, the LINPACK benchmark can sometimes give you some insight into what goes on inside.

Run the program for a series of values of **N**. We often see three phases of behavior:

- a **rising** zone: cache memory and processor are not challenged.
- a **flat** zone, the processor is performing at top efficiency.
- a **decaying** zone, the matrix is so large the cache can't hold enough data to keep the processor running at top speed.

## Computer Performance - The LINPACK Benchmark



## Computer Performance - Measurements

Ruler **FLOP** + Clock **S** = Speedometer **MFLOPS**.

## Computer Performance - Measurements

In practice, we don't count **FLOP**s, just CPU time.

This tells us whether one algorithm is faster than another.

It does not tell us whether we are fully exploiting the computer's potential.

## Computer Performance - Measurements

```
t1 = cputime ( );
C = A * B;
t2 = cputime ( );
```

MATLAB executes this in **0.79** CPU seconds on an Apple G5. (Matrices are 1000x1000).

## Computer Performance - Measurements

```
t3 = cputime ( );
for i = 1 : n
  for k = 1 : n
    C(i,k) = 0;
    for j = 1 : n
      C(i,k) = C(i,k) + A(i,j) * B(j,k);
    end
  end
end
t4 = cputime ( );
```

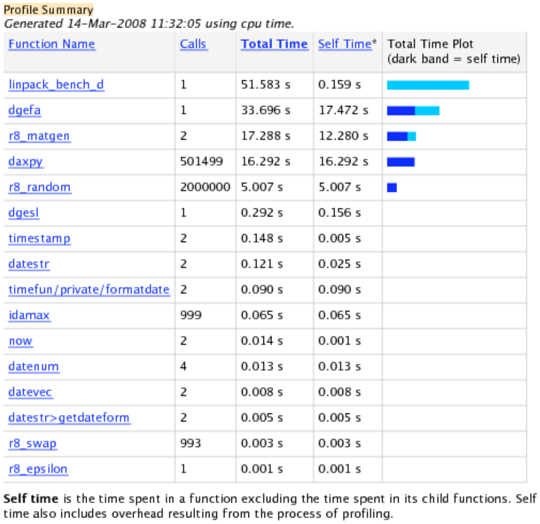
MATLAB executes this in **242.65** CPU seconds on an Apple G5.

## Computer Performance - PROFILE

PROFILE is a MATLAB program profiler.

```
profile on
linpack_bench_d
profile viewer
```

## Computer Performance - PROFILE



Profile Summary
*Generated 14-Mar-2008 11:32:05 using cpu time.*

| Function Name | Calls | Total Time | Self Time* | Total Time Plot (dark band = self time) |
|---|---|---|---|---|
| linpack_bench_d | 1 | 51.583 s | 0.159 s | |
| dgefa | 1 | 33.696 s | 17.472 s | |
| r8_matgen | 2 | 17.288 s | 12.280 s | |
| daxpy | 501499 | 16.292 s | 16.292 s | |
| r8_random | 2000000 | 5.007 s | 5.007 s | |
| dgesl | 1 | 0.292 s | 0.156 s | |
| timestamp | 2 | 0.148 s | 0.005 s | |
| datestr | 2 | 0.121 s | 0.025 s | |
| timefun/private/formatdate | 2 | 0.090 s | 0.090 s | |
| idamax | 999 | 0.065 s | 0.065 s | |
| now | 2 | 0.014 s | 0.001 s | |
| datenum | 4 | 0.013 s | 0.013 s | |
| datevec | 2 | 0.008 s | 0.008 s | |
| datestr>getdateform | 2 | 0.005 s | 0.005 s | |
| r8_swap | 993 | 0.003 s | 0.003 s | |
| r8_epsilon | 1 | 0.001 s | 0.001 s | |

**Self time** is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

## GPROF

GPROF is a program profiler available for Unix systems.

First **compile** your program with the appropriate option, GPROF inserts checks into each routine in your program.

```
gcc -pg myprog.c
g++ -pg myprog.C
gfortran -pg myprog.f
gfortran -pg myprog.f90
```

## GPROF - Run Your Program

Then **run** your program.

```
myprog
```

As the program runs, GPROF regularly checks what it is doing, (what function it is executing), and stores this information in a file.

## GPROF - Create the Report

Then **create the report**:

Once your program **myprog** has executed, the GPROF raw data file must be processed to make a report. You do this by running the GPLOT command and giving it the name of the program you were monitoring:

```
gprof myprog
```

GPROF can report:

- How many times a routine is called.
- Which routines called routine A, and how often.
- Which routines were called by A, and how often.
- Total and average time spent in each routine or function;
- Whether some routine is never called.

## GPROF - one part of the report

```
Each sample counts as 0.000976562 seconds.
  %   cumulative   self              self     total
 time    secs    secs     calls  s/call  s/call  name
 92.07    6.24    6.24   501499    0.00    0.00  daxpy_
  4.48    6.54    0.30        1    0.30    6.77  main_
  1.85    6.67    0.12  2000000    0.00    0.00  d_random__
  1.15    6.74    0.08        1    0.08    6.29  dgefa_
  0.39    6.77    0.03        2    0.01    0.08  d_matgen__
  0.06    6.77    0.00      999    0.00    0.00  idamax_
  0.00    6.77    0.00      993    0.00    0.00  d_swap__
  0.00    6.77    0.00        2    0.00    0.00  timestamp_
  0.00    6.77    0.00        1    0.00    0.02  dgesl_
```

## Measuring Parallel Performance

What we have said so far should help you gather performance data for a sequential version of your program.

Sequential performance data is a vital tool for judging parallel execution data.

Many (good) parallel programs will run **slower** than the sequential version for small problems or a small number of processors! So you want to know when it makes sense to go parallel.

With a sequential code for comparison, you can also detect certain cases when a bad job of parallelizing the code has been done. (Some programs can run slower as more processors are added!)

## Measuring Parallel Performance - Wallclock Time

The most important measure of improvement for parallel programs is an estimate of the speedup.

Sequential programs are usually rated based on CPU time; we are used to the idea that we are trying to run fast, but that the computer is multitasking and doing other things as well.

But parallel programs are rated based on wallclock time. This means that we expect the computer to be more or less dedicated to our job for its duration!

## Measuring Parallel Performance - Wallclock Time

The reasons for this change in measurement include the fact that parallel programs include delays caused by communication between processors and interference between “cooperating” processes. A good parallel program must control these non-computational costs.

Another reason is that parallel programming aims to reduce the **real time** of computing. If I run my program using 8 computers, I don’t want to use 1/8 the CPU time on each...I want to get my results 8 times faster. A parallel program will, in fact, use more resources than a sequential program. Thus, it is less efficient in terms of resources. But our resources (processors) have become cheap, and now time is the main issue!

## Measuring Parallel Performance - Wallclock Time

If we focus on the idea of speedup, then there is a natural way to rate the performance of a parallel program.

We choose a “natural” problem size **N**, and for a range of the number of processors **P**, we measure the wallclock time **T** it takes the program to solve the same problem.

For a “perfect speedup”, we would hope that

$$T = f(N)/P$$

That is, for a fixed **N**, doubling the processors halves the time.

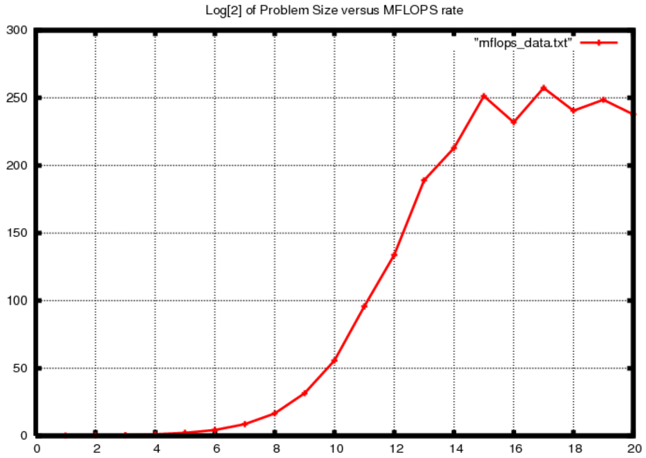
## Measuring Parallel Performance - Problem Size

While communication and synchronization issues have been added to our concerns, it's still important to see know what effect of the problem size **N** has on our program's performance.

We choose a "natural" number of processors **P**, and for a range of problem sizes **N**, we measure the wallclock time **T** it takes the program to solve the same problem. If we know our problem well, we can also determine the MegaFLOPS rate. Since we are allowing **P** processors to cooperate, the MegaFLOPS rate might be as high as **P** times the single processor rate.

If we examine a wide range of problem sizes, we often see that the MFLOPS rate reaches a plateau, and then begins to fall.

## MFLOPS Plot for FFT + OpenMP on 2 Processors



## Measuring Parallel Performance - Problem Size

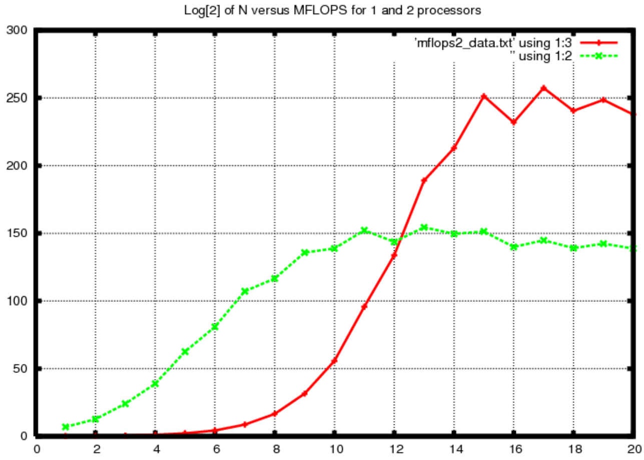
The story is incomplete though! Let's go back (like I said you should do!) and gather the same data for 1 processor.

It's useful to ask ourselves, before we see the plot, what we expect or hope to see. Certainly, we'd like evidence that the parallel code was faster.

But what about the range of values of **N** for which this is true?

Here we will see an example of a "crossover" point. For some values of **N**, running on one processor gives us a better computational rate (**and** a faster execution time!)

## MFLOPS Plot for FFT + OpenMP on 1 versus 2 Processors



## Measuring Parallel Performance - Problem Size

Now, roughly speaking, we can describe the behavior of this FFT algorithm, (at least on 2 processors) as having three ranges, and I will include another which we do not see yet:

- **startup phase**, of very slow performance;
- **surge phase**, in which performance rate increases linearly;
- **plateau phase**, in which performance levels off;
- **saturation phase**, in which performance deteriorates;

## Measuring Parallel Performance - Problem Size

It should become clear that a parallel program may be better than a sequential program...*eventually* or *over a range of problem sizes* or *when the number of processors is large enough*.

Just from this examination of problem size as an influence on performance, we can see evidence that parallel programming comes with a *startup cost*. This cost may be excessive for small problem size or number of processors.

## Measuring Parallel Performance - Number of Processors

For a given problem size **N**, we may have a choice of how many processors **P** we use to solve the problem.

Our measurement of performance is **T**, elapsed wallclock time.

If we increase **P**, we certainly expect **T** to drop.

It is natural to hope that doubling **P** causes **T** to halve.

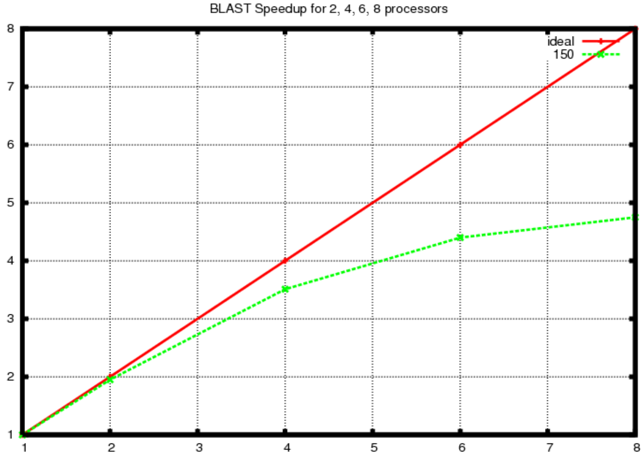
At some point, increasing **P** can't help as much, and eventually it won't help at all. (Too many helpers with too little to do, too much communication)

## Measuring Parallel Performance - Number of Processors

For graphical display, it's better to plot the "speed" or computational rate, that is, **1/T** rather than **T**.

In this way, the ideal behavior is plotted as a diagonal line, and the actual behavior is likely to be a curve that strives for the diagonal line at first, and then "gets tired" and flattens out.

## Performance Data for a version of BLAST



## Measuring Parallel Performance - Number of Processors

When you plot data this way, there is a nice interpretation of the scale. In particular, note that when we use 6 processors, our speedup is about 4.5. This means that we when use 6 processors, the program speeds up as though we had 4.5 processors (assuming a perfect speedup).
You can also see the curve flattening out as the number of processors increases.

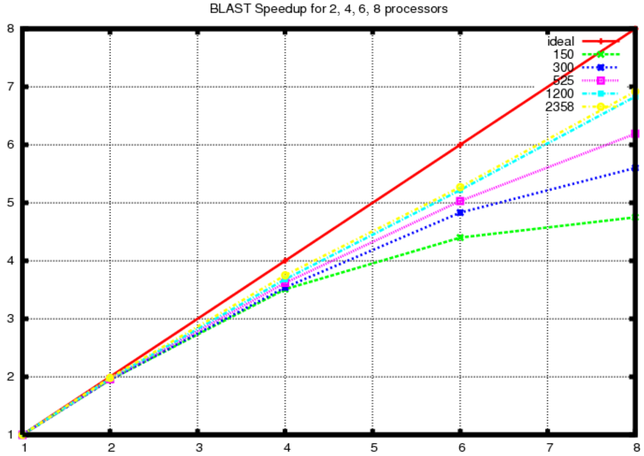
## Measuring Parallel Performance - Number of Processors

We know that performance can depend on the problem size as well as the number of processors. To get a feeling for the interplay of both parameters, it is useful to plot timings in which you solve problems of increasing size over a range of processors.

For a “healthy” parallel program on a “decent” size problem, you may expect that the runs of bigger problems stay closer to the ideal speedup curve.

The problem size for the BLAST program is measured in the length of the protein sequence being analyzed. Let’s go back and do timings for sequences of 5 different sizes.

## Performance Data for a version of BLAST



## Practical Implications - Applying for Computer Time

To get time on a parallel clusters, you must justify your request with performance data.

You should prepare a speedup plot for typical problem size **N** and processors from 1, 2, 4, 8, ... to **P**.

If you still get about 30% speedup using **P** processors, your request is reasonable.

In other words, if a 100 processor machine would let you run 30 times faster, you're got a good case.

## Practical Implications - Applying for Computer Time

The user will be requesting access for a given number of units, which are probably measured in **processor hours**.

Suppose you know you could do these computations sequentially in roughly T hours.

From your speedup plot, you may be able to estimate the number of processors **P*** after which more processors don't help much. (That's when the speedup plot gets pretty flat.)

Call S* the speedup that you get using P* processors.

Take sequential time T and divide by S* to get parallel time T*.

You want to request P* processors for T* hours or a total of (P*)x(T*) processor hours.

## Practical Implications - Applying for Computer Time

A sequential run of my corn genome code takes 50 hours.

I need to do 300 runs, for a total of 15,000 processor hours of work.

My speedup plot for my parallel code flattens out at about P* $=$ 20 processors, at which point I'm getting a speedup of about S* $=$ 10.

So I ask for P* $=$ 20 processors, running S* $=$ 10 times faster.

I *wait less* time (1,500 hours), but I *use more* computer time (30,000 processor hours).

## System X Allocation Request

Please describe the research that will be carried out with the proposed allocation. Large allocations (greater than **100,000 cpu-hours**) are expected to submit descriptions of at least two pages. Medium requests (up to **100,000 cpu-hours**) require at least one page. Small requests (less than **10,000 cpu-hours**) require no more than a page.

- Describe the scientific merit and impact of your research.
- Summarize the research questions of interest.
- Indicate recent publications and funded research.
- Indicate the codes that will be used.
- Mention any previous experience with these codes on distributed-memory parallel systems.
- **Describe how you estimated the total cpu-hours you are requesting.**

## Conclusion

The most powerful and flexible method of measuring computer performance really boils down to simple comparisons.

If you learn how to make and display such comparisons, you can make intelligent judgments about your algorithm, your program, your computer, and your parallel system.

And you may be able to talk your way into some free computer time on our big systems.

Now, as you can see, our time is up!