1 Why do we need parallel computing?

“The number of parallel programs I have embarked on is less than the number of deadlocks I have created.” (Ken Kennedy)

- One year \(= 3 \cdot 10^7 \approx 2^{25} \) seconds
- At 1 GHz, approximately \(3 \cdot 10^{16} \) ops per year
- The true test of someone who does high-performance computing is whether they know how many seconds there are in a year.
- Ok: one op equals one clock tick isn’t really quite right
- All these are ballpark estimates; being off by a couple of orders of magnitude just shows us how close we are to infeasible
2 Measuring speed

“After floating point, no one every thought again. ” (Dan Slotnick)

- We usually measure scientific computing in the number of floating point operations per second
- Usually we count the number of multiplications, because addition and subtraction are very cheap by comparison and division so horrible we do everything we can to avoid it.

“I took a vow 27 years ago never to do another divide. ” (Dan Slotnick)

- Megaflops = MFLOPS = $10^6$ FLOPS, Gigaflops = GFLOPS = $10^9$ FLOPS, Teraflops = TFLOPS = $10^{12}$ FLOPS

- A 1 GHz Intel processor will do $10^9$ 32-bit floating point multiplies in about 4.5 seconds, assuming all entries are in registers. This is $1/4.5$ GFLOP = 222 MFLOPS (32-bit)

- All such numbers are inherently suspect, and there is a large community that is devoted to benchmarking high end machines

- The rating from the vendor will be a number that you can be sure you will never be exceed

- There is great linguistic abuse of the word FLOPS.

- Make sure you know whether you are being quoted 32-bit or 64-bit numbers
Measuring speed (continued)

• Cray 2, 1986

• four processors, 250MHz clock

• Each processor could do one FP multiply and one FP add in one tick, and we call this 1.5 FP ops per tick

• \((4) \cdot (250 \cdot 10^6) \cdot (1.5) = 1500\) MFLOPS

• Yes, you could get this performance over an interval about 64 ticks (= 256 nanoseconds)

• No, you could not sustain that performance

• Typical high-end machines get sustained performance down in the range of 10% of peak or less

• Pipelining inside the processor, multiple functional units, memory access patterns, memory bank conflicts, branch predictions

• Japanese Earth Simulator is 40 TFLOPS peak and allegedly sustains half that on some computations

• Check out the “top 500” list www.top500.org
3 Matrix multiplies

• The naive matrix multiply algorithm is $n^3$ multiplies in operation count.

\[
\begin{align*}
1000 \times 1000 & \quad 1 \text{ GFLOPS} \\
2000 \times 2000 & \quad 8 \text{ GFLOPS} \\
3000 \times 3000 & \quad 27 \text{ GFLOPS} \\
4000 \times 4000 & \quad 64 \text{ GFLOPS} \\
10000 \times 10000 & \quad 10^3 \text{ GFLOPS} \\
100000 \times 100000 & \quad 10^6 \text{ GFLOPS}
\end{align*}
\]

• At 222 MFLOPS, one Intel processor does one matmult on a $10^4$ size matrix in 4500 seconds = 75 minutes

• One matmult on a $10^5$ size matrix in 4500000 seconds = 75000 minutes = 1250 hours = 52 days

• Even if we’re high by a factor of 100, we’re not going to be able to do a lot of these in a reasonable time...
Computers are FAST: Why Do We Care About Speed?
4 Weather modelling

• U.S. is about 3.6 million square miles
• If we sample 10 per mile, horizontally, and 10 per mile for 10 miles up, we have about $3.6 \cdot 10^{10}$ grid points.
• Double this to include necessary area outside the U.S.
• If one “update” includes temp, pressure, and $x$, $y$, $z$ changes, that’s at least 6 flops per update
• Total of $6 \cdot 7.2 \cdot 10^{10} \approx 432 \cdot 10^9$ flops per update
• Hence about 2000 seconds (33 minutes) per update.
• Actually, it’s more like about 200 flops per update (factor of 30 worse).
• Maybe we sample only one per mile (factor of 1000 better).
• This still comes out as minutes per update.
5 Digital Encryption Standard (DES)

• 56 bit keys
• Exhaustion is $2^{56} \approx 7 \cdot 10^{16}$ keys
• At 1 GHz, if 1 key per op for a test, 3 years to exhaust

Of course, this is using the BFI (Brute Force and Ignorance) method
6 Pairwise Particle-Particle Interactions

• This is usually $N^2$ or $\frac{N^2}{2}$ interactions

• Since we’re talking about atomic particles, we would like $N$ to be very very large
7 Definitions

• We are not really going to define the term parallel computer.

• We often call something a multicomputer if it is an aggregation of multiple computers together with an interconnection network. (i.e., each of the subunits of the multicomputer sort of looks like an independent computer)

• In contrast, we call something a multiprocessor if there are multiple computing units but it would be hard to view any single unit as a computer all by itself.

• We usually use the term symmetric multiprocessor to refer to a machine with multiple processing units (CPUs), a common memory, connected in such a way that all processors are equal and access by any processor to any memory location is at least logically the same as access of any other location by any other processor.

• A supercomputer is any of the class of whatever happens to be the biggest, most powerful machine available.

• Or, a supercomputer is any computer that costs at least $10 million dollars.
• A machine has a **shared memory** architecture if all the CPUs can “see” all the memory addresses.

• A machine has a **distributed memory** architecture if a CPU sees only the addresses local to itself.

• A memory architecture has **NUMA (non-uniform memory access)** or **ccNUMA (cache-coherent non-uniform memory access)** if the cost of a processor’s access to memory is not constant; rather, the cost of memory access by a processor depends on which address is being referenced.

• The **communication paths** (e.g., bus, grid/mesh, switched) are the logical paths by which processors access memory or send data among themselves.

• The **interconnection network** is the logical structure by which the processors, memories, and other units/devices are connected to each other.

• A **node** in a parallel computer is an entity on the interconnection network. (This could be a processor, or a memory block, or ...)

• Michael Flynn provided a basic taxonomy of parallel computers as **MIMD, SIMD, SPMD**.

• Some computations are **data parallel** in that the computations can be executed in parallel because the data can be accessed in parallel (e.g., array \( C = array \ A + array \ B \) element-wise)

• We refer to a computation as **embarrassingly parallel** if it can be made parallel in a trivially obvious way.
A computation is **inherently sequential** if there is no way to make it parallel.

The **granularity** of a computation is the size of the basic computational module. Fine-grain parallelism is the sort of thing that takes place inside the CPU. Coarse-grain parallelism might be at the level of a subroutine that could be spawned off as many independent processes running on as many independent computers as one had available.

Most computations exhibit **locality** either in space or time or both: if a given memory location is accessed in a given interval in time, then it is likely that it will be accessed more than once in that interval; and if a given memory location is accessed in a given interval in time, then it is likely that locations nearby will also be accessed in that interval.

A parallel algorithm/computation/implementation/program is called **scalable** if one can increase the number of processors executing in parallel and observe a proportional decrease in the execution time.

A computation is a **needle-in-a-haystack computations** if one is searching for one (or a very small number) optimal answer(s) among a vast number of possible answers.

In contrast to **shared memory** computations, in which processors communicate by reading and writing values in their shared memory space, communication in a **message passing** computation takes place by means of explicit data blocks, or messages, sent from one processor to another.
### 8 Historical Companies and Machines (page 7 of Quinn)

**ILLIAC IV**

<table>
<thead>
<tr>
<th><strong>Vector machines</strong></th>
<th><strong>SIMD machines</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray Research 1, 2, XMP, YMP, C90, T90</td>
<td>Goodyear Aerospace (LORAL) MPP</td>
</tr>
<tr>
<td>CDC 6400, 6600, Cyber 205</td>
<td>Thinking Machines CM-2</td>
</tr>
<tr>
<td>ETA 10</td>
<td>Denelcor</td>
</tr>
<tr>
<td>NEC</td>
<td>MASPAR</td>
</tr>
<tr>
<td>Fujitsu</td>
<td></td>
</tr>
<tr>
<td>Hitachi</td>
<td></td>
</tr>
<tr>
<td>Convex C1, C2</td>
<td></td>
</tr>
</tbody>
</table>
### Historical Companies and Machines (continued)

<table>
<thead>
<tr>
<th>Machines With Structure</th>
<th>Symmetric Multiprocessors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cosmic Cube</td>
<td>Ardent</td>
</tr>
<tr>
<td>Intel Hypercube</td>
<td>Stardent</td>
</tr>
<tr>
<td>BBN</td>
<td>Sequent</td>
</tr>
<tr>
<td>Thinking Machines CM-5</td>
<td>Alliant</td>
</tr>
<tr>
<td>nCUBE</td>
<td>Encore</td>
</tr>
<tr>
<td>Parsytec</td>
<td>Hewlett-Packard (Convex)</td>
</tr>
<tr>
<td>Myrias</td>
<td>Sun Microsystems</td>
</tr>
<tr>
<td>Silicon Graphics (Challenge, Origin)</td>
<td>Digital Equipment Corporation</td>
</tr>
<tr>
<td>Kendall Square Research</td>
<td></td>
</tr>
<tr>
<td>Cray Research T3D, T3E</td>
<td></td>
</tr>
<tr>
<td>IBM SP-1, SP-2</td>
<td></td>
</tr>
<tr>
<td>Beowulf</td>
<td></td>
</tr>
</tbody>
</table>
### Historical Companies and Machines (continued)

<table>
<thead>
<tr>
<th>Symmetric Multiprocessors plus</th>
<th>Other?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSC</td>
<td>Meiko</td>
</tr>
<tr>
<td>Cray Computer Corporation</td>
<td>Floating Point Systems</td>
</tr>
<tr>
<td>SRC Computers, Inc.</td>
<td>Ametek</td>
</tr>
<tr>
<td></td>
<td>C-DAC</td>
</tr>
</tbody>
</table>
9 Rules for Parallel Computing

Rule One: There are no rules.

If a computation consists of two pieces that could in fact be done at the same time, chances are that some computer built at some time will have been able to do this.

Let us try to think of the kinds of parallelism we can exploit in a computation and of the communication patterns that must be efficient in order to get high performance.
9.1 Matrix Multiplication

Consider $A_{m\times n} \cdot B_{n\times p} = C_{m\times p}$, and worry about floating point multiplies.

We have data parallelism here quite suitable for vectorization.

    for(i = 1; i <= m; i++) /* loop on rows i */
    {
        for(j = 1; j <= p; j++) /* loop on columns j */
        {
            c[i,j] = 0.0;
            for(k = 1; k <= n; k++) /* inner loop */
            {
                c[i,j] += a[i,k] * b[k,j];
            } /* end for k */
        } /* end for j */
    } /* end for i */
for(i = 1; i <= m; i++) /* loop on rows i */
{
    for(j = 1; j <= p; j++) /* loop on columns j */
    {
        c[i,j] = 0.0;

        // vectorize the following loop
        for(k = 1; k <= n; k++) /* inner loop */
        {
            c[i,j] += a[i,k] * b[k,j];
        } /* end for k */
    } /* end for j */
} /* end for i */

In general, we only worry about making the innermost loop efficient. It is the innermost loop that is executed the most number of times.

**General rule: high performance computing is applied bottleneckology.**

Now, what about the next levels of parallelism?
If we had a PRAM, we would want to use \( n \) processors to compute all the products \( a[i,k] \times b[k,j] \) in parallel in one tick and store them in an array of products \( \text{prod}[.] \).

Then we would use \( n/2 \) processors in parallel to add pairs of elements of \( \text{prod}[.] \), and then add the pairs of those sums, etc., in a binary tree in \( O(\log n) \) steps.

But of course this is very unrealistic. For a large matrix we can’t hope to have as many processors as we have data elements.

On the other hand, this vector loop has almost perfect \textbf{scalability}:

\begin{itemize}
  \item \( N \) elements and a vector register of length 1, \( 4N \) ticks \ (load, load, multiply, store for each subscript)\).
  \item \( N \) elements and a vector register of length 2, \( 2N+1 = 4(N/2)+1 \) ticks \ (load(2 elts), load(2 elts), multiply(2 elts), store(2 elts) for each pair of subscripts), plus one for adding the two partial sums together.
  \item \( N \) elements and a vector register of length 4, \( N+3 = 4(N/4)+3 \) ticks \ (load(4 elts), load(4 elts), multiply(4 elts), store(4 elts) for each pair of subscripts), plus 3 for adding the four partial sums together using a binary tree.
\end{itemize}

Of course, this is parallelism at a very low level, inside a CPU that has hardware to handle vector arithmetic.

This is \textbf{fine-grained} parallelism.
9.2 Numerical integration

Now consider something like

\[ I = \int_{a}^{b} f(x) \, dx \]

and let’s assume we are going to do a numerical integration (trapezoidal rule or something similarly simple). This can clearly be done as a **coarse-grained** computation. With \( N \) processors that are “real” computers, let \( \delta = (b - a)/N \), and then

\[
I = \int_{a}^{b} f(x) \, dx = \sum_{i=1}^{N} I_i = \sum_{i=1}^{N} \int_{a+(i-1)\delta}^{a+i\delta} f(x) \, dx
\]

Each of these sums is an independent numerical integration and the problem is clearly **embarrassingly parallel**. If a processor knows its own subscript number \( i \) (in the list of processors), then it can compute its own interval \( a + (i - 1)\delta \) to \( a + i\delta \) and thus know how to compute its piece of the computation.

- **BROADCAST** \( a, b, \) and \( f(x) \) from a supervisor node to all other nodes
- Each node computes its own integral \( I_i \) in parallel
- **REDUCE** the \( I_i \) to get \( I \)

This turns \( N \times local\_computation\_cost \) into

**BROADCAST\_cost + 1 \times local\_computation\_cost + REDUCE\_cost**
9.3 Communication is everything

- Clearly, if this approach is to make things go fast, we need the BROADCAST and the REDUCE to go fast.

  That is, we need communication costs to be small compared to the cost of the computational part.

- Note that both BROADCAST and REDUCE should actually involve only a small amount of data transferred.

- The computation

  ```c
  result = INITIAL_VALUE;
  for(i = 1; i <= N; i++) /* loop on rows i */
  {
      result = result OP local_value
  } /* end for i */
  ```

  where OP is any reasonable binary operation (add, multiply, AND, ...) is called a reduction operation.

  It happens in any parallel computation where we farm out the work to multiple processors, and the reduction should always be done using a binary tree.

- In MPI, there is a REDUCE function that automatically implements the most efficient communication pattern for the architecture of the machine.
9.4 A coarse grained embarrassingly parallel example

- Compute $\pi$ by Monte Carlo method
- Compute random points $(x, y)$ in the first quadrant
- Compute $y' = \sqrt{x^2 + y^2}$
- If $y' > 1.0$ then $(x, y)$ is outside the half-unit circle, else inside
- The fraction of points randomly falling inside the circle is $\pi/4$
- Again, this is embarrassingly parallel. We can compute $N$ random pairs $(x, y)$ on one processor, or $N/2$ pairs on two processors, or whatever, and get almost perfect speedup.
- This is a somewhat trivial example, but it illustrates one very important point. If we are to get the benefit of the parallelism, then we need to have a random number generator that will generate independent pseudo-random number pairs in parallel. This is an additional requirement of the RN generator in order for us to get results that we can believe.
9.5 Optimization and Tree Searches

- Suppose we’re minimizing a multivariable function $f(\mathbf{x})$
- We usually would discretize the space because we can’t do continuous math on a computer
- Now we have a choice of breadth-first or depth-first search
- The \textit{parallel} strategy is usually a combination of both
  1. Breadth first across processors, for as many processors as one has
  2. Depth first within processors
  3. Synchronize and share the minimum values found
  4. Repeat

- The problem now becomes one of \textbf{load-balancing}. If we synchronize too often, we spend a lot of time in communication of intermediate results and less time actually computing.
- But if we don’t synchronize often enough, then we have a lot of processors doing useless work because they don’t know what current minimum they have to beat in order to have a new minimum.
9.6 Parallel computing is part theory and part practice

- Yes, you want to look at the theory to get ideas for how to parallelize.
- Much of the time the theory is not much more than a motherhood argument; the real work will be in doing the load-balancing based on the practical costs of memory access, cache behavior, communication among processors, average network behavior, and average case behavior of the “progress” of each processor in the computation.
- A theoretical analysis of this is usually too difficult because there are too many variables.
10 Amdahl’s Law

**Theorem 10.1.** Let $f$ be the fraction of operations in a parallel computation that must be done sequentially, with $0 \leq f \leq 1$. The maximum speedup $\phi$ that can be achieved by running this computation on a machine with $p$ processors is

$$\phi \leq \frac{1}{f + (1 - f)/p}$$

*Proof.* The best we can hope for is that the fraction $1 - f$ of parallelizable operations gets speeded up by a factor of $p$. So the previous running time of

$$time_{\text{sequential}} = (f + (1 - f)) \times time/\text{operation} \times \text{operations}$$

becomes a new running time of

$$time_{\text{parallel}} = f + (1 - f)/p \times time/\text{operation} \times \text{operations}$$

and the speedup, which is the quotient, is as above. \qed
10.1 The bad news from Amdahl’s Law

Assume that we have infinitely many processors. Then

<table>
<thead>
<tr>
<th>Sequential fraction $f$</th>
<th>Max possible speedup</th>
<th>Sequential fraction $f$</th>
<th>Max possible speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.90</td>
<td>1.11</td>
<td>0.30</td>
<td>3.33</td>
</tr>
<tr>
<td>0.80</td>
<td>1.25</td>
<td>0.20</td>
<td>5.00</td>
</tr>
<tr>
<td>0.70</td>
<td>1.43</td>
<td>0.10</td>
<td>10.00</td>
</tr>
<tr>
<td>0.60</td>
<td>1.67</td>
<td>0.05</td>
<td>20.00</td>
</tr>
<tr>
<td>0.50</td>
<td>2.00</td>
<td>0.01</td>
<td>100.00</td>
</tr>
<tr>
<td>0.40</td>
<td>2.50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Even if only one percent of the code cannot be parallelized, you can’t get a speedup of more than 100.

On the other hand, Amdahl’s law is partly bogus, since it really only addresses the speedup on a particular computer program. If there were a radically different algorithm, then better speedup could be achieved.
11 Parallel Programming Asymptotics

• **Big Oh:** (Landau, 1890s) We write

\[ f(x) = O(g(x)) \]

if \( \exists \) constants \( c \) and \( C \) such that \( x > c \implies |f(x)| \leq C \cdot g(x) \).

• **Little Oh:** (Landau, 1890s) We write

\[ f(x) = o(g(x)) \]

if \( |f(x)/g(x)| \to 0 \) as \( \to \infty \).
•**Big Omega:** (Hardy, about 1915) We write

\[ f(x) = \Omega(g(x)) \]

if \( f(x) \) is *not* \( o(g(x)) \), that is: there exists a sequence \( x_1, x_2, ..., x_n, ... \), tending to \( \infty \), such that for any fixed constant \( C \), there exists a constant \( c \) such that \( x_i > c \implies |f(x_i)| \geq C \cdot g(x_i) \).

•**Knuth’s Big Omega:** (Knuth’s corrupted notation, about 1975) We write

\[ f(x) = \Omega_K(g(x)) \]

if for any fixed constant \( C \) there exists a constant \( c \) such that

\[ x > c \implies |f(x)| \geq C \cdot g(x). \]

•**Theta:** (??) We write

\[ f(x) = \Theta(g(x)) \]

if there exist constants \( c, C_1, \) and \( C_2 \) such that

\[ x > c \implies C_1 \cdot g(x) \leq |f(x)| \leq C_2 \cdot g(x). \]
11.1 Why Not Asymptotics?

- Parallel programming is a lot of work.
- The people who are willing to do parallel programming are usually forced to do it in order to solve real problems.
- Proving theorems about asymptotics is not their goal. Solving problems is their goal.
- Parallel efficiency is a multivariable problem involving execution time costs, communication costs, memory access costs, etc.
- Relative costs vary from one machine to another.
- Relative costs can depend on the workload.
- The constants can dominate the analysis.