Parallel algorithms and parallel architectures

In order to solve large problems, a good match between algorithm and architecture is necessary.

A large program may require $10^{12}$ to $10^{15}$ floating-point operations.

But there are only $3.15 \times 10^{13}$ microseconds per year. So a conventional architecture would take years to finish the program.

Thus, it may be worthwhile to

- tailor a parallel machine toward a specific problem, or
- invest a lot of effort in optimizing a program for an architecture.

The architect needs to know the characteristics of parallel algorithms in order to design an architecture that is not too “special purpose.”

Large-scale numerical problems have these common features:

- A heavy dependence on floating-point arithmetic because of a potentially large range of values.
- The calculation uses discrete points in space and time to model what might be a continuous function of space and time.
- The algorithm designer has some flexibility in determining the size of a problem.
- The precision of the result can be traded off against the time to execute the program.

Example: A 64×64 processor array is obviously more efficient for some sizes of problems than others. Which?

However the algorithm is tailored to an architecture, it may have to be changed when the problem is moved to another machine.
• Current optimizing compilers can’t do that job very well.
• Expert programmers usually beat optimizing compilers.
• So it may be worthwhile to program at a lower level.

**Continuum models**
Physical computational models fall mostly into two categories.

• *Continuum models*, in which
  ° time and space are considered to vary continuously, and
  ° typical parameters are charge density, temperature, and pressure—physical measures averaged over regions.

• *Particle models*, in which
  ° the universe is modeled as discrete particles, and
  ° typical parameters are physical variables such as velocity, force and momentum—which measure the current state of individual particles.

**Example:** Views of plasma.

• Continuous entity with internal temperature that varies continuously throughout according to physical laws.

• Discrete: each particle is a distinct entity; temperature is a function of the velocity and density of the particles.

Obviously, continuous models have to be converted to discrete form in order to be run on a digital computer.

A continuum model has this major advantage:

• It obeys partial differential equations.
• When cast in discrete form, it produces equations where all changes to variables are functions of nearby variables.
A particle model permits particles to be affected by distant particles. Of course, distant particles have limited effect.

What is the impact on parallel algorithms?

- In the continuum model, as the effects of several particles are propagated through near neighbors, they tend to combine, and propagate as a combined effect.
- Also, continuum computations grow as the product of
  - the number of points in the continuum,
  - the number of neighbors, and
  - the length of the longest propagation path.

However, interactions between particles are reflected more quickly by a discrete model.

Also, the number of points in the continuum model is not directly related to the number of particles in the discrete model.

**Poisson’s equation on the continuum model**

A typical problem is Poisson’s equation for the potential in a region as a function of charge density in that region.

The equation in two dimensions can be written as—

\[
\frac{\partial^2 V(x, y)}{\partial x^2} + \frac{\partial^2 V(x, y)}{\partial y^2} = -C(x, y)
\]

where—

\(V(x, y)\) is the voltage potential at the point \((x, y)\), and \(C(x, y)\) is the charge at the point \((x, y)\).
A solution depends on boundary conditions, which can be expressed in various ways.

For example, suppose—

- we are to solve the equations in a region \(0 \leq x \leq 1, 0 \leq y \leq 1\),
- and that we are given the values of \(V(x, y)\) on the boundaries of this region.

To match a (discrete) digital computer, we must transform this equation.

We can represent a continuous region by a mesh of discrete points:

```
  0, 0  0, 1  0, 2  0, 3  ...
  1, 0  1, 1  1, 2  1, 3  ...
  2, 0  2, 1  2, 2  2, 3  ...
   ...   ...   ...   ...
```

At the intersection \((i, j)\) in this mesh, we store the values \(V(i, j)\) and \(C(i, j)\).

If we have an \(N \times N\) processor array, the indices \(i\) and \(j\) run from

so the corresponding values of \(x\) and \(y\) are given by

\[
x = \quad \text{and} \quad y =
\]

Increasing the size of the mesh increases

- the fidelity of the representation, and
- the solution time.
Now we can transform the continuous equations into their discrete analog.

The equation involving second derivatives can be approximated in a similar fashion to equations of first derivatives, e.g.,

$$\frac{\partial V(x)}{\partial x} = -C(x)$$

Should we use the discrete analogue of the classical definition of a derivative?

$$\frac{\partial V(x)}{\partial x} = \frac{V_{i+1} - V_i}{\frac{1}{N}}$$

The denominator is $1/N$, the mesh spacing. This equation expresses the derivative at what point?

The derivative at point $i$ is more closely approximated by—

$$\frac{\partial V(x)}{\partial x} = \frac{V_{i+1/2} - V_{i-1/2}}{\frac{1}{N}}$$

where the grid points $i+1/2$ and $i-1/2$ are fictitious.

If we needed to use real points, we could

- use __________ , and
- adjust

However, the fictitious points cancel out when we take the second derivative.

Let us calculate the discrete approximation to the second derivative in one dimension:
\[ \frac{d^2 V(x)}{dx^2} = \frac{d}{dx} \left( \frac{d V(x)}{dx} \right) = \frac{(V_{i+1,j} - V_{i,j}) - (V_{i,j} - V_{i-1,j})}{(1/N)^2} = \frac{V_{i+1,j} + V_{i-1,j} - 2V_{i,j}}{(1/N)^2} \]

Using this expression, our original equation becomes—

\[ \frac{V_{i+1,j} + V_{i-1,j} - 2V_{i,j}}{(1/N)^2} + \left( \frac{1}{N} \right)^2 = V_{i,j} - \frac{C_{i,j}}{4N^2} \]

for \( 0 < i < N, 0 < j < N \).

In other words, the potential at each point in the mesh is—

- the average of the values at its four neighbors,
- plus a term that reflects the charge located at that point.

This is a system of linear equations involving the \((N-2)^2\) unknown values of \(V(x, y)\) on the interior of the mesh.

How can we solve this system? Standard algorithms will work, but we would like methods that take advantage of the structure of the system of equations.

One popular way is **four-point iteration**:

\[ V_{i,j}^{(t+1)} = \frac{V_{i+1,j}^{(t)} + V_{i-1,j}^{(t)} + V_{i,j+1}^{(t)} + V_{i,j-1}^{(t)}}{4} + C_{i,j} \]

(The superscript indicates the iteration number.)

The algorithm has converged when no point changes value as a result of performing the iteration.

Clearly, this sort of calculation is well suited to a rectangular mesh.

Here is a program for performing this computation.
An instruction has the form OPCODE DEST, SOURCE

- If the instruction has two operands, DEST holds the second operand.
- So MOVE REG[1], MEM[V] copies

To transfer data between processors, it must be loaded into the routing register:

```
MOVE ROUTE, MEM[V]  or  MOVE ROUTE, REG[1].
```

The ROUTE instruction (e.g., “ROUTE NORTH”) specifies the direction of transfer. (All cells route in the same direction.)

No subscripts are required, because the mesh point \((i, j)\) is held in processor \((i, j)\).

**LOOP:**

```
LOAD   REG[1], MEM[V]  \quad \text{REG[1] = Local potential value}
MOVE   ROUTE, REG[1]  \quad \text{Prepare to route value}
ROUTE  NORTH           \quad \text{Send it north}
MOVE   REG[2], ROUTE   \quad \text{Save value from south}
MOVE   ROUTE, REG[1]   \quad \text{Send it east}
ADD    REG[2], ROUTE   \quad \text{Add in value from the west}
MOVE   ROUTE, REG[1]   \quad \text{Send it south}
ROUTE  SOUTH           \quad \text{Add in value from the north}
ADD    REG[2], ROUTE   \quad \text{Send it west}
MOVE   ROUTE, REG[1]   \quad \text{Add in value from the east}
DIV    REG[2]/4         \quad \text{Find average of neighbors}
ADD    REG[2], MEM[C]  \quad \text{Add in local charge}
MOVE   MEM[V], REG[2]  \quad \text{REG[2] has new value of V}
...     ...               \quad \text{REG[1] has old value of V}
MOVE   MEM[V], REG[2]  \quad \text{Add code here to detect convergence}
```

How is convergence tested for? The \(i\)th processor can set or reset the \(i\)th bit of the operand returned to the control unit.
• Each processor compares the magnitude of the change in the potential function to a small constant $\epsilon$.

• The control unit requests a response to the comparison.

• The $i$th processor places a 1 in the $i$th bit of the response vector if the change exceeds $\epsilon$; otherwise it places a 0 in that position.

• The response vector is transmitted over the bus to the control unit.

• The control unit examines the response. If any bits are one, it initiates a new iteration.

If the number of mesh points exceeds the number of processors, then the mesh can be broken into smaller regions.

Then the program may have to use indexing—

$$\text{MEM}[V] \text{ becomes } \text{MEM}[V[K]]$$

This algorithm is a good match for a rectangular mesh.

• The problem partitions easily by assigning a processor to each node of the mesh.

• The memories are independent, and each can be accessed each cycle.

• All communication is done between neighboring processors. There is never any contention for communication paths.

• The processors are synchronized on the instruction level, where there is almost no synchronization overhead.

However, the architecture has these limitations:

• Varying the mesh size might be inefficient.

• There is no way to perform different calculations on different parts of the mesh. Some algorithms require this.
MIMD processors overcome some of the inflexibility of SIMD machines, but at the cost of—

- contention for resources, and
- greater synchronization overhead.

**Matrix multiplication on the Connection Machine**

The Connection Machine CM-1 and CM-2 were MODS machines of the late ’80s. Because they had thousands of processors (up to 65K processors), they could run interesting data-parallel algorithms.

We will proceed to a discussion of matrix multiplication on the CM. First, however, it is necessary to tell how \( n \) items can be summed in \( \log n \) time.

For simplicity, we assume that the number of elements to be summed is a power of 2.

“\( \text{for\_all } k \)” causes all processors to execute the same statement in parallel, each with their own value of \( k \). Specifically, \( k \) is the index of the processor within the array.

\[
\text{for } j := 1 \text{ to } \log_2 n \text{ do }
\text{for\_all } k \text{ do }
\text{if } ((k+1) \mod 2^j) = 0 \text{ then }
\quad x[k] := x[k-2^j] + x[k];
\]

Which processors are active on the first iteration?

Which processors are active on the 2nd iteration?

Which processor is active on the last iteration?

On the first iteration, the sum of two numbers is produced. On the second iteration, the sum of four numbers is produced.

In general, on the \( j \)th iteration, the sum of \( 2^j \) numbers is produced.
This is sometimes called *parallel sum reduction*, or a *fanin* operation.

Many parallel matrix-multiplication algorithms are possible with varying efficiency for different array sizes. We will consider four algorithms due to Tichy.

The problem is to multiply two matrices to produce a third. The superscripts below represent the sizes of the matrices.

\[ C^{l,n} = A^{l,m} \times B^{m,n} \]

1. *An \( O(n^2 \log n) \) algorithm.* Each of the processors contains a row of each of the matrices. Thus we need _______ processors.

   The algorithm computes ______ inner products in sequence, one for each element of \( C \).

   An inner product requires \( m \) multiplications. These can be done in parallel in constant time.

   After the products are computed, a ________________ is done to produce the sum in \( O(\log n) \) time.

   Here is a diagram that shows how it is done.

\[
\begin{array}{c}
A \\
\text{ROW } i \\
\hline
\text{\(c_{i,j}\)}
\end{array}
\quad \Rightarrow \quad
\begin{array}{c}
\text{\(B\)}
\end{array}
\]

Since arrays are stored one row per processor, a single column of \( B \) is spread over ___ processors.

The idea is that we sequentially send each row of \( A \) to each of the processors that will compute an element of \( C \) using that row.
Since a processor can send only one element at a time, the sending of a row takes $O(m)$ communication steps.

2. *The slow $O(n \log n)$ algorithm.* This algorithm is like algorithm 1, except …

Instead of using $m$ processors to compute a single inner product in parallel, we use $l \times m$ processors to compute an entire column of $C$ at once.

Each of the $O(n^2)$ processors contains one element of $A$, $B$, and $C$.

Assume that the matrix elements are assigned to the processors in row-major order.

To compute a column of $C$ at once, we—

- broadcast the relevant column of $B$ to all rows of $A$,
- execute $l \times m$ multiplications in parallel, then
- perform $l$ sum reductions in parallel.

How do we broadcast the columns of $B$?

- We could use the host to broadcast to all processors at once. However, it can only broadcast one element at a time, so the process would take _____ for broadcast alone.

- We could use the router: Each element of $A$ could simply retrieve the required coefficient from $B$ directly.

Unfortunately, this would again result in quadratic runtime, since
- each element of $B$ would be requested by $l$ rows of $A$ simultaneously, and
- each processor can only honor one request at a time.

Instead, we program a *fanout tree* for each column element. A fanout tree has the same structure as a fanin tree, except that data flows from the root to the leaves instead of vice versa.
Since we need a tree for each column element, we actually need to construct a fanout forest. The forest completes the broadcast in logarithmic time, as the diagram at the right shows.

The fanout is done like this.

- We “seed” the column of $B$ into the first row of $A$, in parallel for all the elements.
- The first row of $A$ duplicates the seeded coefficients one row down.
- The first two rows of $A$ duplicate their elements two rows down, then the first four rows duplicate elements four rows down, etc.
- The process continues until the (processors for the) entire matrix $A$ is filled with copies of this column of $B$.

The process takes $O(\log l)$ communication steps.

The $l$ sum-reductions are performed along the rows of $A$, orthogonal to the broadcast.

3. *The fast $O(n \log n)$ algorithm.* Algorithm 2 uses processors and communication bandwidth poorly.

- In the first step of the broadcast, only $m$ of the $l \times m$ processors operate.
- Even in the last step, only half the processors operate.

Again, the matrices are laid out one element per processor, in row-major order.
• We transpose $B$, and overlay it on matrix $A$. The router performs this operation in constant time.

Now row $i$ of $A$ is lined up with column $i$ of $B$.

• Then we perform $l$ parallel inner-product steps, producing the main diagonal of $C$ in $O(\log m)$ time.

• Then we rotate the matrix $B$ up one row, with the topmost row re-entering at the bottom.

Now row $i$ of $A$ is lined up with column ______ of $B$, and we compute the upper main diagonal of $C$, along with element $C_{l,0}$.

• After $n$ steps of inner-product computation and rotation, $C$ is complete.

Tichy also gives two $O(n)$ algorithms, but they are somewhat more complicated to describe, so we will skip them. We proceed to the $O(\log n)$ algorithm.

4. The $O(\log n)$ algorithm. This algorithm uses $n^3$ processors to compute all $n^3$ products simultaneously. Then it performs ______ sum reductions in parallel.

Before the multiplications can be done, the data must be distributed to the $n^3$ processors.

• Each row of $A$ must be duplicated $n$ times.
• Each column of $B$ must be duplicated $l$ times.

Two fanout forests accomplish this in logarithmic time.
What is a major limitation of this algorithm?

The $O(\log n)$ algorithm was the fastest until it exhausted the number of available processors. After that, it used CM's virtual processors, which are multiplexed onto the physical processors at a low level of microcode. But that made it slower than the fast $O(n)$ algorithm.

For $100 \times 100$ arrays, the $O(\log n)$ algorithm exhausts the CM memory with its replicated data and virtual processor stack.