PVM vs. MPI

MPI = Message-Passing Interface
- SPMD (Single program, multiple data)
- Each node runs the same program
- The program "just exists", it is not spawned explicitly
- PVM = Parallel Virtual Machine
- "MPMD" (Multiple program, multiple data)
- Processes are explicitly spawned
- Processes are assigned to nodes in separate layer, possibly multiple per node

PVM
- Can be used for heterogeneous network
- Arbitrary topology
- Messages can cross outside of host boundaries.
- Explicit packing and unpacking of messages required in code
- Fault tolerance features
- PVM came before MPI.
- Lower level, but more flexible

In PVM daemon processes must be resident on nodes prior to spawning PVM processes there.
- Upon command, the daemon launches the process.
- The PVM host file identifies participating nodes, or they can be added manually from the command line.
- Root process is started from pvm console command-line on one host.

Processes explicitly spawn child processes.
- Child can determine its parent.
- Processes have their own "task id".
- Point-to-point send/receive similar to MPI.
- Tags, wildcards similar to MPI.
int main(int argc, char* argv[]) {
    /* find out my task id number */
    mytid = pvm_mytid();

    /* find my parent's task id number */
    myparent = pvm_parent();

    if (myparent == PvmNoParent) {
        /* spawn the child tasks */
        info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0,
                        /* I'm still the parent */
        for (i = 0; i < ntask; i++) {
            /* recv a message from any child process */
            buf = pvm_recv(-1, JOINTAG);
            info = pvm_bufinfo(buf, &len, &tag, &tid);
            info = pvm_upkint(&mydata, 1, 1);
        }
        pvm_exit();
    }
    /* I'm a child */
    info = pvm_initsend(PvmDataDefault);
    info = pvm_pkint(&mytid, 1, 1);
    info = pvm_send(myparent, JOINTAG);
    pvm_exit();
}


PVM groups

- Processes explicitly join and leave groups, named symbolically.
- Multicast, gather, barriers, etc. are done relative to group.
- Multicast can be into group from outside.
- Reduce operator for +, *, max, min, or user-defined
- A process can be in multiple groups.

PVM

- Multicast to explicit receives, unlike MPI.

For further info, examples, and on-line manual, see:
- [http://www.cse.uncc.edu/~abw/parallel/orig_pvm/using_pvm.html](http://www.cse.uncc.edu/~abw/parallel/orig_pvm/using_pvm.html)
Timing Analysis for Parallel Applications

Time Decomposition

- Parallel execution time can be divided into:
  - Actual computation time +
  - Communication time

\[ t_{parallel} = t_{comp} + t_{comm} \]

If there are \( m \) non-parallel message steps overall, then

\[ t_{comm} = m \times t_{message} \]

Message Time Decomposition

- Message time can be divided into:
  - **Latency** (or start-up time) +
  - (number of data communicated)*(delay per datum)

\[ t_{message} = t_{startup} + n \times t_{datum} \]

\( 1 / t_{datum} \) is often called “bandwidth”, the number of data per unit time.

Some Comparative Times (Pacheco 1997)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Arithmetic Op</th>
<th>Latency</th>
<th>Delay per Double Ops</th>
<th>Omps/Latency</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3D</td>
<td>0.011 ms</td>
<td>21 ms</td>
<td>0.3 ms</td>
<td>1909</td>
<td></td>
</tr>
<tr>
<td>IBM SP-2</td>
<td>0.0042 ms</td>
<td>35 ms</td>
<td>0.23 ms</td>
<td>8333</td>
<td></td>
</tr>
<tr>
<td>Ethernet</td>
<td>N/A</td>
<td>500 ms</td>
<td>8.9 ms</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Latency Hiding

- In order to prevent \( t_{message} \) from destroying any speedup due to parallelism, we can try the following:
  - While a processing element is awaiting a message, perform some computation that doesn’t require the message.
  - Note that we are really trying to hide the entire communication cost, not just the “latency” component of it.

Latency Hiding (2)

- One technique for hiding latency is “multiprogramming”:
  - On a single processor, run more than one process.
  - While one process is awaiting a message, another could be doing useful computational work.
  - This requires that process-switching be relatively efficient (e.g. using threads rather than processes).
  - The ratio of processes to processors is sometimes called the “parallel slackness”.
Parallel Time Complexity

- We assume familiarity with $O$, $\Omega$, and $\Theta$ notation.
- Their use is to bound the time complexity as a function of the problem size “n”.

Complexity Example

- Matrix-vector multiplication:
  - $n \times n$ matrix
  - $n$ element vector
  - Assume $n$ processors
  - Every processor has a row of the matrix
  - Each row is multiplied by the vector simultaneously
  - It takes $O(n)$ to multiply one row, so $t_{\text{comp}}(n) \in O(n)$

Matrix-Vector Multiplication

- If the matrix first had to be distributed in order for the multiplication to take place, then the cost of distributing the rows from one processing element is $O(n^2)$, while the cost of collecting the result is $O(n)$.
- Therefore, the parallel cost is the same as the obvious sequential cost.

Cost Optimality

- A cost-optimal algorithm is defined to be one in which the effort, as a function of problem size, is bounded by a constant times the sequential effort.
- One-shot matrix-vector multiplication is not cost-optimal for distributed memory using the technique described, whereas multiplication repeated at least $n$ times is.

Using Graphs to Illustrate Algorithms

- The vector inner product can be shown thus (ignoring distribution cost):

```
  vector 1
   /     \
(1) --- (2) --- (3) --- (4)
  \     /          \     /    
   (5) --- (6) --- (7) --- (8)

  vector 2
```

- The resulting graph shows the connections between elements in the vectors.
Assume unit time for each operation.
The time is proportional to path length.
The longest path length for an n-element vector is $O(n)$, sim. to serial.

### Using Graphs to Illustrate Algorithms

- Restructuring the + nodes as a tree gives us faster performance on n processors.

### Algorithm Analysis

- The previous tree gives us $O(\log n)$ on n processors.
- Is it cost optimal?

### Scaling Down Processors

- As the size of the vector grows very large, we can divide the additions up among p processors, $p \ll n$, adding the elements within a processor sequentially and only using the tree at the end.
- The time is dominated by the sequential adds, which is $O(n)$ for a given $p$.
- Is this cost optimal?

### Generalization

- Whenever the number of operations (including communication as an operation) in the parallel case is proportional to the serial complexity, we can achieve cost optimality by scaling down.
- The general concept is captured by Brent’s Lemma.

### Brent’s Lemma

- If an algorithm A entails $m$ operations and can be done in parallel time $t$ with some number of processors, then $p$ processors can execute the algorithm in time $t + (m-t)/p$.
Brent's Lemma Summarized

- \( t \) = time on some number of processors
- \( m \) = number of operations (unit time each)
- time on \( p \) processors is \( \leq t + (m-t)/p \)

Application of Brent's Lemma

- To achieve cost optimality for vector inner product, use \( n/(\log n) \) processors.
- Observed that product can be done in \( \log n \) with arbitrarily-many processors.
- Brent's lemma says it can be done with \( p \) processors in
  \[
  \log n + \frac{(2n-1-\log n)}{(n/\log n)} \leq t + \frac{(m-t)}{p}
  \]

Proof of Brent's Lemma (1)

- Consider the graph of the algorithm done with some number of processors in time \( t \).
- Let \( s_i \) be the number of operations done at the \( i \)th level, i.e. at "time" \( i \).
- On \( p \) processors, we can reschedule the \( s_i \) operations in time ceiling(s/p).

Proof of Brent's Lemma (2)

- On \( p \) processors, we can reschedule the \( s_i \) operations in time ceiling(s/p).
- The total computation can therefore be done on \( p \) processors in time
  \[
  \sum(i = 1 \text{ to } t, \text{ceiling}(s/p))
  \]

Proof of Brent's Lemma (3)

- \[
  \sum(i = 1 \text{ to } t, \text{ceiling}(s/p))
  \]
  is bounded by
  \[
  \sum(i = 1 \text{ to } t, (s_i+p-1)/p)
  \]
  \[
  = \sum(i = 1 \text{ to } t, s/p) + \sum(i = 1 \text{ to } t, p/p) - \sum(i = 1 \text{ to } t, 1/p)
  \]
  \[
  = m/p + t - t/p
  \]
  \[
  = t + (m-t)/p, \text{ as advertised.}
  \]
Illustration of Brent

Illustration of Brent for 2 processors

<table>
<thead>
<tr>
<th>Brent's bound predicts: $1 + \frac{(n-t)}{p} = 6 + \frac{(12-6)}{2} = 9$</th>
<th>time on 2 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1 = 1$</td>
<td>1</td>
</tr>
<tr>
<td>$s_2 = 3$</td>
<td>2</td>
</tr>
<tr>
<td>$s_3 = 1$</td>
<td>1</td>
</tr>
<tr>
<td>$s_4 = 1$</td>
<td>1</td>
</tr>
<tr>
<td>$s_5 = 3$</td>
<td>2</td>
</tr>
<tr>
<td>$s_6 = 3$</td>
<td>2</td>
</tr>
<tr>
<td>total = 9</td>
<td></td>
</tr>
</tbody>
</table>

Graph Exercise

- By the prefix sum problem, we mean that of computing from an array $x_0, x_1, x_2, \ldots, x_{n-1}$
- the array $(x_0), (x_0+x_1), (x_0+x_1+x_2), \ldots, (x_0+x_1+\ldots+x_{n-1})$
- Can this problem be sped up using parallelism?
- Is there a cost optimal version?