Large Scale Multiprocessors and Scientific Applications

By
Pushkar Ratnalikar
Namrata Lele
• Introduction
• Interprocessor Communication
• Characteristics of Scientific Applications
• Synchronization: Scaling up
• Performance of Scientific Applications on Shared Memory Multiprocessors
• Performance of Scientific Applications on Distributed Memory Multiprocessors
• Implementing Cache Coherence
• Custom Cluster Approach: Blue Gene/L
• Primary application of large-scale multiprocessors is for true parallel programming
• Characteristics of parallel programs:
  - Amount of parallelism
  - Size of parallel tasks
  - Frequency and nature of intertask communication
  - Frequency and nature of synchronization
Performance metrics:

1. Communication bandwidth
2. Communication latency = Sender overhead + Time of Flight + Transmission time + Receiver overhead
3. Communication latency hiding
Advantages of Shared Memory Communication Mechanism

• Ease of programming for complex and dynamic communication patterns among processors

• Lower overhead for communication and better use of communication bandwidth when communicating small items

• Ability to use hardware controlled caching to reduce the frequency of remote communication by supporting automatic caching of all data.
Advantages of Message Passing Communication Mechanism

- Hardware can be simpler compared to shared memory
- Communication is explicit which means it's simple to understand
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization
Characteristics of Scientific Applications

- FFT kernel: Used in fields ranging from signal processing to fluid flow to climate modeling
- LU kernel: LU factorization of a dense matrix
- Barnes Application: Solves a problem in galaxy evolution
- Oceans Application: Simulates the influence of eddy and boundary currents on large scale flow in the ocean
Sequential time for n data points $O(n \log n)$
Sequential time for $n \times n$ matrix is $o(n^3)$
Barnes Application

Replacing Clusters by their Centers of Mass Recursively

Applying the idea of force calculation recursively
Each processor is allocated a subtree
Size of subtree allocated to a processor is based on some measure of work it has to do (how many other cells it needs to visit) rather than just on the number of nodes in the subtree

Sequential time for n data points $O(n \log n)$
• Red-black Gauss-Seidel colors points in grid to consistently update points based on previous values of adjacent neighbors.

• Each grid in hierarchy has fewer points than the grid below and is an approximation to the lower grid.

• The entire ocean basin is partitioned into square subgrids that are allocated to the portion of the address space corresponding to the local memory of individual processors.

• Communication occurs when boundary points of a subgrid are accessed by adjacent subgrid.

• Sequential time for nxn grid: $O(n^2)$
• If the ratio of computation to communication is high, it means the application has lots of computation for each datum communicated.

• Knowing how the ratio changes as we increase the processor count sheds light on how well the application can be sped up.
<table>
<thead>
<tr>
<th>Application</th>
<th>Scaling of computation</th>
<th>Scaling of communication</th>
<th>Scaling of computation-to-communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>$\frac{n\log n}{p}$</td>
<td>$\frac{n}{p}$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>LU</td>
<td>$\frac{n}{p}$</td>
<td>$\frac{\sqrt{n}}{\sqrt{p}}$</td>
<td>$\frac{\sqrt{n}}{\sqrt{p}}$</td>
</tr>
<tr>
<td>Barnes</td>
<td>$\frac{n\log n}{p}$</td>
<td>approximately $\frac{\sqrt{n} \log n}{\sqrt{p}}$</td>
<td>approximately $\frac{\sqrt{n}}{\sqrt{p}}$</td>
</tr>
<tr>
<td>Ocean</td>
<td>$\frac{n}{p}$</td>
<td>$\frac{\sqrt{n}}{\sqrt{p}}$</td>
<td>$\frac{\sqrt{n}}{\sqrt{p}}$</td>
</tr>
</tbody>
</table>
### Computation/Communication Example: Ocean Application

<table>
<thead>
<tr>
<th>Processors (p)</th>
<th>Size (n)</th>
<th>Time (min)</th>
<th>Computation</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>58</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>40</td>
<td>1/2</td>
<td>sqrt(1)/sqrt(2)</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>?</td>
<td>2/32 = 1/16</td>
<td>sqrt(2)/sqrt(32) = 1/4</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>?</td>
<td>16/128 = 1/8</td>
<td>sqrt(16)/sqrt(128) = 1/(2 * sqrt(2))</td>
</tr>
<tr>
<td>128</td>
<td>32</td>
<td>?</td>
<td>32/128 = 1/4</td>
<td>sqrt(32)/sqrt(128) = 1/2</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>?</td>
<td>128/256 = 1/2</td>
<td>sqrt(128)/sqrt(256) = 1/2</td>
</tr>
</tbody>
</table>

From the table, if we call the base time of computation $x$, and the base time of communication $y$ then we get (using minutes to be consistent):

(1) \[ 58 = 1x + 0y \]

(2) \[ 40 = 0.5x + 1/(\sqrt{2})y \]

From the first equation, we now know that $x=58$ min. From this, we can solve for $y$ in equation 2, which is $y=12 \times \sqrt{2}$. 
From this we can solve for each of the times (Note: $T_{p,n}$ where p is #processors and n is size):

$$T_{32,2} = \frac{1}{16}x + \frac{1}{4}y$$
$$= \frac{1}{16}(58) + \frac{1}{4}(12*\sqrt{2})$$
$$= 3.625 + 4.24$$
$$= 8\text{min 26sec} = 8 \text{ min}$$

$$T_{128,16} = \frac{1}{8}x + \frac{1}{(2*\sqrt{2}))}y$$
$$= \frac{1}{8}(58) + \frac{1}{(2*\sqrt{2}))}(12*\sqrt{2})$$
$$= 7.25 + 6$$
$$= 13\text{ min 25 sec} = 13 \text{ min}$$

<table>
<thead>
<tr>
<th>Processors (p)</th>
<th>Size(n)</th>
<th>Time (min)</th>
<th>Computation</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>58</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>40</td>
<td>(\frac{1}{2})</td>
<td>(\frac{\sqrt{1}}{\sqrt{2}})</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>8</td>
<td>(\frac{2}{32} = \frac{1}{16})</td>
<td>(\frac{\sqrt{2}}{\sqrt{32}} = \frac{1}{4})</td>
</tr>
<tr>
<td>128</td>
<td>16</td>
<td>13</td>
<td>(\frac{16}{128} = \frac{1}{8})</td>
<td>(\frac{\sqrt{16}}{\sqrt{128}} = \frac{1}{(2 * \sqrt{2})})</td>
</tr>
<tr>
<td>128</td>
<td>32</td>
<td>23</td>
<td>(\frac{32}{128} = \frac{1}{4})</td>
<td>(\frac{\sqrt{32}}{\sqrt{128}} = \frac{1}{2})</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>37</td>
<td>(\frac{128}{256} = \frac{1}{2})</td>
<td>(\frac{\sqrt{128}}{\sqrt{256}} = \frac{1}{2})</td>
</tr>
</tbody>
</table>
Synchronization performance challenges:

Example: Suppose 10 processors on a bus try to lock a variable simultaneously. Assume each bus transaction read/write miss is 100 clock cycles long. Determine the time required for all 10 processors to acquire the lock, assuming they are all spinning when the lock is released at time 0.
Example continued

<table>
<thead>
<tr>
<th>lockit:</th>
<th>LL</th>
<th>R2,0(R1)</th>
<th>;load linked</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BNEZ</td>
<td>R2,lockit</td>
<td>;not available-spin</td>
</tr>
<tr>
<td></td>
<td>DADDUI</td>
<td>R2,R0,#1</td>
<td>;locked value</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>R2,0(R1)</td>
<td>;store</td>
</tr>
<tr>
<td></td>
<td>BEQZ</td>
<td>R2,lockit</td>
<td>;branch if store fails</td>
</tr>
</tbody>
</table>

When \( i \) processes are contending for the lock, they perform the following sequence of actions, each of which generates a bus transaction:

- \( i \) load linked operations to access the lock
- \( i \) store conditional operations to try to lock the lock
- 1 store (to release the lock)

Thus for \( i \) processes, there are a total of \( 2i + 1 \) bus transactions.

Thus, for \( n \) processes, the total number of bus operations is:

\[
\sum_{i=1}^{n} (2i + 1) = n(n + 1) + n = n^2 + 2n
\]

For 10 processes there are 120 bus transactions requiring 12,000 clock cycles or 120 clock cycles per lock acquisition!
• A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes.

• A typical implementation of barrier can be done with two spin locks (lock and unlock notation):
  - One to protect a counter that tallies the processes arriving at the barrier
  - One to hold the processes until the last process arrives at the barrier

• Barrier uses the ability to spin on a variable until it satisfies a test; we use the spin(condition) notation.
Simple Barrier Code

```
lock (counterlock); /* ensure update atomic */
if (count == 0) release=0; /* first => reset release */
count = count + 1; /* count arrivals */
unlock(counterlock); /* release lock */
if (count == total) { /* all arrived */
   count=0; /* reset counter */
   release=1; /* release processes */
}
else { /* more to come */
   spin (release=1); /* wait for arrivals */
}
```

- total = No of processes that must reach the barrier
- count = Tally of how many processes have reached the barrier
- Lock and unlock are basic spin locks
- Release is used to hold the processes until the last one reaches the barrier
If a process races ahead to the next instance of this barrier while some other processes are still in the barrier, the fast process cannot trap the other processes, since it does not reset the value of release as it did before.
Synchronization Mechanisms for Large-Scale Multiprocessors

- Software Implementations:
  - Spin Lock with exponential back-off

- Queuing Locks: Construct a queue of waiting processors; whenever processor frees up the lock it causes the next processor in the queue to attempt access

```assembly
lockit:
DADDUI R3,R0,#1 ;R3 = initial delay
LL R2,0(R1) ;load linked
BNEZ R2,lockit ;not available-spin
DADDUI R2,R2,#1 ;get locked value
SC R2,0(R1) ;store conditional
BNEZ R2,gotit ;branch if store succeeds
DSLL R3,R3,#1 ;increase delay by factor of 2
PAUSE R3 ;delays by value in R3
J lockit

gotit: use data protected by lock
```
Synchronization Mechanisms for Large-Scale Multiprocessors

- Hardware Implementation:
  - Queuing Lock:

Has a synchronization controller. If the lock is free, it is simply returned to the processor. If the lock is unavailable the controller creates a record of the node’s request and sends the processor back a locked value for the variable which the processor then spins on. When lock is freed, controller selects a processor to go ahead from the list of waiting processors. It can then either update the lock variable in the selected processors cache or invalidate the copy, causing a miss and then fetch the available copy of the lock.
Performance of Scientific Workload on Shared-Memory Multiprocessors

• **Variables**
  – Processor Count
  – Cache Size
  – Block Size

• **Metrics**
  – Coherence Misses
  – Uniprocessor Misses
    • Capacity Misses
    • Conflict Misses
    • Compulsory Misses
Varying Processor Count

**FFT**
- Processor count: 1, 2, 4, 8, 16
- Miss rate:
  - Coherence miss rate
  - Capacity miss rate

**LU**
- Processor count: 1, 2, 4, 8, 16
- Miss rate:
  - Coherence miss rate
  - Capacity miss rate

**Barnes**
- Processor count: 1, 2, 4, 8, 16
- Miss rate:
  - Coherence miss rate
  - Capacity miss rate

**Ocean**
- Processor count: 1, 2, 4, 8, 16
- Miss rate:
  - Coherence miss rate
  - Capacity miss rate
Varying Cache Size

FFT

Miss rate

<table>
<thead>
<tr>
<th>Cache size (KB)</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss rate</td>
<td>8%</td>
<td>6%</td>
<td>4%</td>
<td>2%</td>
</tr>
</tbody>
</table>

LU

Miss rate

<table>
<thead>
<tr>
<th>Cache size (KB)</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss rate</td>
<td>2.5%</td>
<td>2%</td>
<td>1.5%</td>
<td>1%</td>
</tr>
</tbody>
</table>

Barnes

Miss rate

<table>
<thead>
<tr>
<th>Cache size (KB)</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss rate</td>
<td>1.5%</td>
<td>1%</td>
<td>0.5%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Ocean

Miss rate

<table>
<thead>
<tr>
<th>Cache size (KB)</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss rate</td>
<td>14%</td>
<td>12%</td>
<td>10%</td>
<td>8%</td>
</tr>
</tbody>
</table>

Legend:
- Coherence miss rate
- Capacity miss rate
Varying Block Size

**FFT**

- Miss rate
- Block size (bytes)

- Coherence miss rate
- Capacity miss rate

**LU**

- Miss rate
- Block size (bytes)

**Barnes**

- Miss rate
- Block size (bytes)

**Ocean**

- Miss rate
- Block size (bytes)
Bus Traffic for data misses

Bytes per data reference vs. Block size (bytes)

- FFT
- LU
- Barnes
- Ocean
Performance of Scientific Workload on Distributed-Memory Multiprocessors

• Variables
  – Processor Count
  – Cache Size
  – Block Size

• Metrics
  – Local Misses
  – Remote Misses
Varying Processor Count

**FFT**
- Processor count: 8, 16, 32, 64
- Miss rate: 0%, 1%, 2%, 3%, 4%, 5%, 6%
- Local misses: dark gray
- Remote misses: light gray

**LU**
- Processor count: 8, 16, 32, 64
- Miss rate: 0%, 0.5%, 1.0%
- Local misses: dark gray
- Remote misses: light gray

**Barnes**
- Processor count: 8, 16, 32, 64
- Miss rate: 0%, 0.5%
- Local misses: dark gray
- Remote misses: light gray

**Ocean**
- Processor count: 8, 16, 32, 64
- Miss rate: 0%, 2%, 4%, 6%, 8%
- Local misses: dark gray
- Remote misses: light gray
Varying Cache Size

![Graphs showing varying cache size for FFT, LU, Barnes, and Ocean benchmarks.](image)

- **FFT**
  - Miss rate vs. cache size (KB)
  - Local misses and Remote misses indicated.

- **LU**
  - Miss rate vs. cache size (KB)
  - Local misses and Remote misses indicated.

- **Barnes**
  - Miss rate vs. cache size (KB)
  - Local misses and Remote misses indicated.

- **Ocean**
  - Miss rate vs. cache size (KB)
  - Local misses and Remote misses indicated.

Legend:
- **Local misses**
- **Remote misses**
Varying Block Size

- FFT
- LU
- Barnes
- Ocean

Legend:
- Local misses
- Remote misses
Effective Latency

Graphs showing effective latency for FFT, LU, Barnes, and Ocean benchmarks across different processor counts. The graphs represent average cycles per reference, with different colors indicating cache hit, local miss, remote miss, and three-hop miss to remote cache.
Implementing Cache Coherence

- Snoopy protocol (handling of upgrade miss)
  - Detect Miss and compose invalidate message
  - Lock bus and broadcast the message
  - Upgrade the Cache Block

- Contention to send invalidate message!
- e.g. processors P1 and P2 are attempting to upgrade the same cache block at the same time!
Implementing Cache Coherence

• Race winner is decided by the ordering imposed by the medium.
• What about the looser?
• Need to impose the condition that the looser (or any other processor contending) must handle any pending invalidates before it can generate its own invalidate i.e. P2 should handle any pending invalidates.
Implementing Cache Coherence in DSM-M’s

• No broadcast medium
• Need to overcome problems related to non-atomic actions in snoopy protocols (W/O broadcast medium).
• Write requests are easily serviced as the unique directory which holds the block processes requests and informs the requester about success
• NAK is sent to the loser's in the race. This causes the loser to regenerate the request.
Implementing Directory Controller

• Need to have the same capabilities as that in the snoopy case
• Need to handle requests for independent blocks while awaiting response to request from a local processor
• Process requests in order.
• Directory should be multithreaded i.e. handle requests for multiple blocks independently.
Implementing Directory Controller

- Directory controller should be reentrant i.e. capable of suspending its execution while waiting for reply and accept another transaction.
- The major implementation difficulty is to handle NAK’s (keep track of outstanding transactions and the corresponding NAK’s).
- Usually there is a reply slot for each request. It can hold ACK or a NAK.
Blue Gene/L

• Scalable distributed memory message-passing supercomputer.
• Focus on Power consumption. Has the highest throughput/cubic foot.
• The first Super Computer to give 100 TFLOPS sustained on real-world applications like 3-D molecular dynamics code, simulating solidification of molten metal under high pressure and temperature conditions.
Blue Gene/L node

- Customized processing node, each containing two PowerPC 4000 chips.
- Each modestly clocked at 700 MHz (to reduce power consumption)
- 2-issue superscalar and 7 stage pipeline.
- Consists of up to 64 K nodes, organized in 32 racks each taking approx. 50 cubic feet.
Blue Gene/L node

- **32K/32K L1**: PPC 440 CPU, Double-issue FPU
- **32K/32K L1**: PPC 440 CPU, Double-issue FPU

**Shared L3 directory for embedded DRAM** includes error correction control (ECC)

**4 MB embedded DRAM**

- **144 ECC**: 22 GB/sec
- **1024**: 1.44 GB/sec

**Ethernet Gbit**

- **IEEE 1149.1 (JTAG)**
- **6 out and 6 in, each at 1.4 GB/sec link**

**JTAG access**

- **3 out and 3 in, each at 2.8 GB/sec link**

**Torus**

- **5.5 GB/sec**

**Collective**

- **4 global barriers or interrupts**

**Global interrupt/lockbox**

- **144-bit-wide DDR**: 256/512 MB

**DDR control with ECC**

**Gigabit Ethernet**

**IEEE 1149.1**

**JTAG**
Blue Gene/L inter-connection network

- 3-D torus network - nodes connect to 6 nearest neighbors in a 3-D mesh.
- Provides high bandwidth nearest neighbor connectivity.
- Cost-effective, scalable and directly applicable to many scientific applications.
- Provides both dynamic and deterministic routing with virtual buffering and cut-through capability.
Blue Gene/L Performance on LINPACK

Theoritical  Sustained

2004  2007 (number of cabinets doubled to 32)
• Into Wide Blue Yonder with BlueGene/L (https://www.llnl.gov/str/April05/Seager.html)
• BlueGene/L torus interconnection network (http://www.research.ibm.com/journal/rd/492/adiga.html)
• The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors By-Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy https://eprints.kfupm.edu.sa/70813/1/70813.pdf
• CSUSB, K. E. Schubert http://ftp.csci.csusb.edu/schubert/tutorials/csci610/w05/DTD_Ocean.pdf
Thank You!