Parallel programming paradigms and their performances

A high level exploration of the HPC world

George Bosilca
University of Tennessee, Knoxville
Innovative Computer Laboratory

Overview

• Definition of parallel application
• Architectures taxonomy
• Laws managing the parallel domain
• Models in parallel computation
• Examples

Formal definition

Bernstein
\{ I_1 \cap O_2 = \emptyset \text{ and } I_2 \cap O_1 = \emptyset \text{ and } O_1 \cap O_2 = \emptyset \}

General case: P_1\ldots P_n are parallel if and only if each for each pair P_i, P_j we have P_i \parallel P_j.

3 limit to the parallel applications:
1. Data dependencies
2. Flow dependencies
3. Resources dependencies
Data dependencies

I1: \( A = B + C \)
I2: \( E = D + A \)
I3: \( A = F + G \)

- Dataflow dependency
- Anti-dependency
- Output dependency

How to avoid them?
Which can be avoided?

Flow dependencies

I1: \( A = B + C \)
I2: if( A ) {
I3: \( D = E + F \) }
I4: \( G = D + H \)

- Dataflow dependency
- Flow dependency

How to avoid?

Resources dependencies

I1: \( A = B + C \)
I2: \( G = D + H \)

How to avoid?
Flynn Taxonomy

- Computers classified by instruction delivery mechanism and data stream
- 4 characters code: 2 for instruction stream and 2 for data stream

<table>
<thead>
<tr>
<th>1 Instruction flow</th>
<th>&gt; 1 Instruction flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 data stream</td>
<td>SISD: Von Neumann</td>
</tr>
<tr>
<td></td>
<td>MISD: pipeline</td>
</tr>
<tr>
<td>&gt; 1 data stream</td>
<td>SIMD</td>
</tr>
<tr>
<td></td>
<td>MIMD</td>
</tr>
</tbody>
</table>

Flynn Taxonomy: Analogy

- SISD: lost people in the desert
- SIMD: rowing
- MISD: pipeline in the car construction chain
- MIMD: airport facility, several desks working at their own pace, synchronizing via a central database.

Amdahl Law

- First law of parallel applications (1967)
- Limit the speedup for all parallel applications

\[
\text{speedup} = \frac{s + p}{s + p/N}
\]

\[
\text{speedup} = \frac{1}{a + (1-a)/N}
\]

\(s\) = number of processors

\(N\) = number of processors
Amdahl Law

- Speedup is bound by $1/a$.

**Bad news for parallel applications**

- 2 interesting facts:
  - We should limit the sequential part
  - A parallel computer should be a fast sequential computer to be able to resolve the sequential part quickly

- What about increasing the size of the initial problem?

Gustafson Law

- Less constraints than the Amdahl law.
- In a parallel program the quantity of data to be processed increase, so the sequential part decrease.

\[
\begin{align*}
\frac{t}{P} & = \frac{s + P/n}{a^n} \\
\text{speedup} & = \frac{s + a^n n}{s + a} \\
& \rightarrow \infty \text{ } \Rightarrow \text{ speedup } \rightarrow n
\end{align*}
\]
Gustafson Law

- The limit of Amdahl Law can be transgressed if the quantity of data to be processed increase.

\[ \text{speedup} \leq n + (1 - n)s \]

Rule stating that if the size of most problems is scaled up sufficiently, then any required efficiency can be achieved on any number of processors.

Speedup

- Superlinear speedup?

Sometimes superlinear speedups can be observed!

- Memory/cache effects
- More processors typically also provide more memory/cache.
- Total computation time decreases due to more page/cache hits.
- Search anomalies
- Parallel search algorithms.
- Decomposition of search range and/or multiple search strategies.
- One task may be "lucky" to find result early.

Parallel execution models

- Amdahl and Gustafson laws define the limits without taking in account the properties of the computer architecture.
- They cannot be used to predict the real performance of any parallel application.
- We should integrate in the same model the architecture of the computer and the architecture of the application.
What are models good for?

- Abstracting the computer properties
  - Making programming simple
  - Making programs portable?
- Reflecting essential properties
  - Functionality
  - Costs
- What is the von-Neumann model for parallel architectures?

Parallel Random Access Machine

- One of the most studied
- World described as a collection of synchronous processors which communicate with a global shared memory unit.

\[
P_1 \quad P_2 \quad P_3 \quad P_4
\]

Shared memory

How to represent the architecture

- 2 resources have a major impact on the performances:
  - The couple (processor, memory)
  - The communication network.
- The application should be described using those 2 resources.
  \[
  T_{\text{app}} = T_{\text{comp}} + T_{\text{comm}}
  \]
Models

- 2 models are often used.
- They represent the whole system as composed by \( n \) identical processors, each of them having his own memory.
- They are interconnected with a predictable network.
- They can realize synchronizations.

Bulk Synchronous Parallel – BSP

- Distributed-memory parallel computer
- Global vision as a number of processor/memory pairs interconnected by a communication network

\[
\begin{array}{cccc}
\text{P/M} & \text{P/M} & \text{P/M} & \text{P/M} \\
\end{array}
\]

- Each processor can access his own memory without overhead and have a uniform slow access to remote memory

BSP

- Applications composed by Supersteps separated by global synchronizations.
- One superstep include:
  - A computation step
  - A communication step
  - A synchronization step

Synchronization used to insure that all processors complete the computation + communication steps in the same amount of time.
BSP

\[ T_{\text{superstep}} = w + g \times h + l \]

Where:
- \( w = \text{max of computation time} \)
- \( g = 1/(\text{network bandwidth}) \)
- \( h = \text{max of number of messages} \)
- \( l = \text{time for the synchronization} \)

Sketch the communications.

BSP

- An algorithm can be described using only \( w, h \) and the \textit{problem size}.
- Collections of algorithms are available depending on the computer characteristics.
  - Small \( L \)
  - Small \( g \)
- The best algorithm can be selected depending on the computer properties.
**BSP - example**

- Numerical solution to Laplace’s equation

\[
U_{ij}^{(n+1)} = \frac{1}{4} \left( U_{i+1,j}^{(n)} + U_{i-1,j}^{(n)} + U_{i,j+1}^{(n)} + U_{i,j-1}^{(n)} \right)
\]

for \( j = 1 \) to \( j_{\text{max}} \)

for \( i = 1 \) to \( i_{\text{max}} \)

\[
U_{\text{new}}(i,j) = 0.25 \times ( U(i-1,j) + U(i+1,j) \\
+ U(i,j-1) + U(i,j+1) )
\]

end for

end for

--

**BSP - example**

- The approach to make it parallel is by partitioning the data

Overlapping the data boundaries allow computation without communication for each superstep.

On the communication step each processor updates the corresponding columns on the remote processors.
BSP - example

\[\begin{align*}
  &\text{for } j = 1 \text{ to } j_{\text{max}} \\
  &\text{for } i = 1 \text{ to } i_{\text{max}} \\
  &\quad \text{unew}(i,j) = 0.25 \times (U(i-1,j) + U(i+1,j) \\
  &\quad \quad + U(i,j-1) + U(i,j+1)) \\
  &\text{end for} \\
  &\text{end for} \\
  &\text{if } \text{me} \neq 0 \text{ then} \\
  &\quad \text{bsp\_put( to the left )} \\
  &\text{endif} \\
  &\text{if } \text{me} \neq \text{NPROCS} - 1 \text{ then} \\
  &\quad \text{bsp\_put( to the right )} \\
  &\text{Endif} \\
  &\text{bsp\_sync()} 
\end{align*}\]

BSP - example

\[T_{\text{superstep}} = w + g \times h + l\]

\[h = \text{max number of messages}\]
\[= l \text{ values to the left} + l \text{ values to the right}\]
\[= 2 \times l \text{ (ignoring the inverse communication!)}\]

\[w = 4 \times l \times l / p\]

\[T_{\text{superstep}} = \frac{4 \times l \times l + 2 \times g \times l + l}{p}\]

BSP - example

- BSP parameters for a wide variety of architectures has been published.

<table>
<thead>
<tr>
<th>Machine</th>
<th>s</th>
<th>p</th>
<th>l</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin 2000</td>
<td>101</td>
<td>4</td>
<td>1789</td>
<td>39057</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>46.7</td>
<td>4</td>
<td>357</td>
<td>751</td>
</tr>
<tr>
<td>Pentium 10Mbit</td>
<td>81</td>
<td>4</td>
<td>139891</td>
<td>826054</td>
</tr>
<tr>
<td>Pentium II</td>
<td>38</td>
<td>4</td>
<td>27583</td>
<td>38788</td>
</tr>
</tbody>
</table>
A more sophisticated model
LogP

• Tend to be more empirical and network-related.

A more sophisticated model
LogP

• Tend to be more empirical and network-related.

LogP

• Decompose the communications in 3 elements:
  – Latency: small message cross the network
  – Overhead: lost time in communication
LogP

- Decompose the communications in 3 elements:
  - Latency: small message cross the network
  - Overhead: lost time in communication
  - Gap: between 2 consecutive messages
- And P the number of processors.

Both g and o matter!

\[ g > o \quad g < o \]

LogP

- The total time for a message to go from the processor A to the processor B is:
  \[ L + 2 \times o \]
- There is no model for the application
- We can describe the application using the same approach as for BSP: supersteps

\[ T_{\text{superstep}} = w + h \times (L + 2o) + l \]

LogP

- The P parameter does not interfere in the superstep computation?
- When the number of processors is not fixed:
  - The time of the computation change \( w(p) \)
  - The number of messages change \( h(p) \)
  - The synchronization time change \( l(p) \)
LogP

- Allow/encourage the usage of general techniques of designing algorithms for distributed memory machines: exploiting locality, reducing communication complexity and overlapping communication and computation.
- Balanced communication to avoid overloading the processors.

- Interesting concept: idea of finite capacity of the network. Any attempt to transit more than a certain amount of data will stall the processor.
- This model does not address on the issue of message size, even the worst is the assumption of all messages are of "small" size.
- Does not address the global capacity of the network.

Design a LogP program

- Execution time is the time of the slowest process
- Implications for algorithms:
  - Balance computation
  - Balance communications
  - Remember the capacity constraint
LogP Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>L</th>
<th>φ</th>
<th>g</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM-5</td>
<td>6</td>
<td>2.2</td>
<td>4</td>
<td>512</td>
</tr>
<tr>
<td>Melko CS-2</td>
<td>8.6</td>
<td>1.7</td>
<td>14.2 + 0.03x</td>
<td>64</td>
</tr>
<tr>
<td>Power Xplorer</td>
<td>21 - 0.02x</td>
<td>70 + x</td>
<td>115 + 1.45x</td>
<td>8</td>
</tr>
<tr>
<td>Para-Station</td>
<td>50 - 0.10x</td>
<td>3 + 0.112x</td>
<td>3 + 0.115x</td>
<td>4</td>
</tr>
<tr>
<td>IBM SP-2</td>
<td>13 - 0.003x</td>
<td>8 + 0.006x</td>
<td>10 + 0.01x</td>
<td>128</td>
</tr>
<tr>
<td>IBM SP-2</td>
<td>17 - 0.005x</td>
<td>8 + 0.008x</td>
<td>10 + 0.01x</td>
<td>256</td>
</tr>
</tbody>
</table>

Improving LogP

- First model to break the synchrony of parallel execution
- LogGP: augments the LogP model with a linear model for long messages
- LogGPC model extends the LogGP model to include contention analysis using queuing model on the k-ary n-cubes network
- LogPQ model augments the LogP model on the stalling issue of the network constraint by adding buffer queues in the communication lines.

The CCM model

- Collective Computing Model transform the BSP superstep framework to support high-level programming models as MPI and PVM.
- Remove the requirement of global synchronization between supersteps, but combines the message exchanges and synchronization properties into the execution of a collective communication.
- Prediction quality usually high.
POSIX Threads & RPC: 2 parallel programming models

George Bosilca
bosilca@cs.utk.edu

Process vs. Thread

- A process is a collection of virtual memory space, code, data, and system resources.
- A thread (lightweight process) is code that is to be serially executed within a process.
- A process can have several threads.

Threads executing the same block of code maintain separate stacks. Each thread in a process shares that process's global variables and resources.

Possible to create more efficient applications?

Process vs. Thread

- Multithreaded applications must avoid two threading problems: deadlocks and races.
- A deadlock occurs when each thread is waiting for the other to do something.
- A race condition occurs when one thread finishes before another on which it depends, causing the former to use a bogus value because the latter has not yet supplied a valid one.
The key is synchronization

• Synchronization = gaining access to a shared resource.
• Synchronization REQUIRE cooperation.

POSIX Thread

• What’s POSIX ?
  – Widely used UNIX specification
  – Most of the UNIX flavor operating systems

POSIX is the Portable Operating System Interface, the open operating interface standard accepted world-wide. It is produced by IEEE and recognized by ISO and ANSI.

Mutual exclusion

• Simple lock primitive with 2 states: lock and unlock
• Only one thread can lock the mutex.
• Several politics: FIFO, random, recursive
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive

![Diagram showing lock and unlock operations across threads]

Thread 1

Active threads

Thread 2

Thread 3

Mutex

Sleeping threads

Thread 2

Thread 3

Thread 2

Thread 3

Thread 3

Thread 3

Thread 2

Thread 2

Thread 1

Thread 1

Thread 1
**Mutual exclusion**

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive

---

**Spin vs. sleep?**

- What’s the desired lock grain?
  - Fine grain – spin mutex
  - Coarse grain – sleep mutex

- Spin mutex: use CPU cycles and increase the memory bandwidth, but when the mutex is unlock the thread continue his execution immediately.
Shared/Exclusive Locks

- **ReadWrite Mutual exclusion**
- Extension used by the reader/writer model
- 4 states: write_lock, write_unlock, read_lock and read_unlock.
- multiple threads may hold a shared lock simultaneously, but only one thread may hold an exclusive lock.
- if one thread holds an exclusive lock, no threads may hold a shared lock.

---

### Legend

- **Active thread**
- **Sleeping thread**

---

### Step 1

- **Writer 1**
  - `rw_lock`
  - `rw_unlock`

- **Writer 2**
  - `rw_lock`
  - `rw_unlock`

- **Reader 1**
  - `rd_lock`
  - `rd_unlock`

- **Reader 2**
  - `rd_lock`
  - `rd_unlock`

---

### Step 2

- **Writer 1**
  - `rw_lock`
  - `rw_unlock`

- **Writer 2**
  - `rw_lock`
  - `rw_unlock`

- **Reader 1**
  - `rd_lock`
  - `rd_unlock`

- **Reader 2**
  - `rd_lock`
  - `rd_unlock`

---

### Step 3

- **Writer 1**
  - `rw_lock`
  - `rw_unlock`

- **Writer 2**
  - `rw_lock`
  - `rw_unlock`

- **Reader 1**
  - `rd_lock`
  - `rd_unlock`

- **Reader 2**
  - `rd_lock`
  - `rd_unlock`

---

### Step 4

- **Writer 1**
  - `rw_lock`
  - `rw_unlock`

- **Writer 2**
  - `rw_lock`
  - `rw_unlock`

- **Reader 1**
  - `rd_lock`
  - `rd_unlock`

- **Reader 2**
  - `rd_lock`
  - `rd_unlock`
<table>
<thead>
<tr>
<th>Step</th>
<th>Active threads</th>
<th>Sleeping threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Thread 1</td>
<td>Thread 2, Thread 3</td>
</tr>
<tr>
<td></td>
<td>Active threads</td>
<td>Sleeping threads</td>
</tr>
<tr>
<td>6</td>
<td>Thread 1, Thread 2</td>
<td>Thread 3</td>
</tr>
</tbody>
</table>

### Shared/Exclusive Locks

<table>
<thead>
<tr>
<th>Legend</th>
<th>Writer 1</th>
<th>Writer 2</th>
<th>Reader 1</th>
<th>Reader 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active thread</td>
<td>rw_lock</td>
<td>rw_lock</td>
<td>rd_lock</td>
<td>rd_lock</td>
</tr>
<tr>
<td>Sleeping thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Condition Variable

- Block a thread while waiting for a condition
- `Condition_wait / condition_signal`
- Several threads can wait for the same condition, they all get the signal

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active threads</td>
<td>wait</td>
<td>wait</td>
</tr>
<tr>
<td>Signal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active threads</td>
<td>wait</td>
<td>wait</td>
</tr>
<tr>
<td>Signal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active threads</td>
</tr>
<tr>
<td>Signal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active threads</td>
</tr>
<tr>
<td>Signal</td>
</tr>
</tbody>
</table>

---

**Legend**
- `rw_lock`: Read/Write lock
- `rw_unlock`: Read/Write unlock
- `rd_lock`: Read lock
- `rd_unlock`: Read unlock

**Step 5**
- Writer 1
- Writer 2
- Reader 1
- Reader 2

**Step 6**
- Writer 1
- Writer 2
- Reader 1
- Reader 2
Condition Variable

- Block a thread while waiting for a condition
- Condition_wait / condition_signal
- Several thread can wait for the same condition, they all get the signal
Semaphores

- simple counting mutexes
- The semaphore can be hold by as many threads as the initial value of the semaphore.
- When a thread get the semaphore it decrease the internal value by 1.
- When a thread release the semaphore it increase the internal value by 1.
**Atomic instruction**

- Is any operation that a CPU can perform such that all results will be made visible to each CPU at the same time and whose operation is safe from interference by other CPUs
  - TestAndSet
  - CompareAndSwap
  - DoubleCompareAndSwap
  - Atomic increment
  - Atomic decrement