Lecture 4

The Theory: Programming Models

Christopher Mitchell, Ph.D.
cmitchell@cs.nyu.edu || http://z80.me
Lecture 3 Outline

- C++ Templating Aside
- Classifying Hardware
- Matching Software and Hardware
- Programming Models
  - Why Programming Models?
  - Serial Model
  - PRAM
  - LogP
  - BSP and Multi-BSP
C++ Templating Aside

Normally:

foo.hpp:
```cpp
class MyClass {
    // ...
    int myMethod();
};
```

foo.cc:
```cpp
#include "foo.hpp"
int MyClass::myMethod() {
    // ...
}
```

With templates:

foo.hpp:
```cpp
template<typename A>
class MyClass {
    // ...
    int myMethod(const A& a);
};
```

#include foo-impl.hpp

foo-impl.hpp:
```cpp
template<typename A>
int MyClass<A>::myMethod(const A& a) {
    // ...
}
```
Classifying Hardware
Last Week: Parallelism
How about the Hardware?

Latency Vs Throughput

Symmetric Multiprocessing: Multiple CPUs
Chip-Level Multiprocessing: Multiple Cores
Classifying the Hardware

- Data parallelism
- Instruction parallelism
- Memory model
Flynn Classification

• A taxonomy of computer architecture
• Proposed by Michael Flynn in 1966
• Classifies by:
  • Instruction parallelism
  • Data parallelism

<table>
<thead>
<tr>
<th></th>
<th>Single instruction</th>
<th>Multiple instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
Instruction Parallelism: Single Data Architectures

PU = Processing Unit
Instruction Parallelism: Multiple Data Architectures

SIMD
Instruction Pool

MIMD
Instruction Pool

PU = Processing Unit
MIMD Memory Models: Shared Memory
MIMD Memory Models: Distributed Memory
MIMD Memory Models: Hybrid
Multicore and Manycore

“We have arrived at many-core solutions not because of the success of our parallel software but because of our failure to keep increasing CPU frequency.”

-Tim Mattson
Parallel Computing @ Intel

• Dilemma:
  • Parallel hardware is ubiquitous
  • Parallel software is not!

• After more than 25 years of research, we are not closer to solving the parallel programming model!
# The Mentality of Yet Another Programming Language

The Mentality of Yet Another Programming Language

<table>
<thead>
<tr>
<th>ABCPL</th>
<th>CORRELATE</th>
<th>GLU</th>
<th>GUARD</th>
<th>Legion</th>
<th>Meta Chaos</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>CPS</td>
<td>HAI</td>
<td>Haskell</td>
<td>Midway</td>
<td>Millipede</td>
</tr>
<tr>
<td>ACT++</td>
<td>CRL</td>
<td>HPC++</td>
<td>ParC</td>
<td>Parallaxis</td>
<td>ParLib++</td>
</tr>
<tr>
<td>Active messages</td>
<td>CSP</td>
<td>JAVAR</td>
<td>Millipede</td>
<td>ParLib++</td>
<td>Parmacs</td>
</tr>
<tr>
<td>Adl</td>
<td>Cthreads</td>
<td>HORUS</td>
<td>Mirge</td>
<td>ParLib++</td>
<td>Parex++</td>
</tr>
<tr>
<td>Adsmith</td>
<td>CUMULVS</td>
<td>HPC</td>
<td>MpC</td>
<td>Parafrase2</td>
<td>pC++</td>
</tr>
<tr>
<td>ADDAP</td>
<td>DAGGER</td>
<td>PGM</td>
<td>Mentat</td>
<td>Paralation</td>
<td>SCHEDULE</td>
</tr>
<tr>
<td>AFAPI</td>
<td>DAPPLE</td>
<td></td>
<td></td>
<td>SciTL</td>
<td>SDDA</td>
</tr>
</tbody>
</table>

---

We don’t want to scare away the programmers ... Only add a new API/language if we can’t get the job done by fixing an existing approach.

<table>
<thead>
<tr>
<th>Blaze</th>
<th>DSM-Threads</th>
<th>Paras</th>
<th>Nemo</th>
<th>Nimrod</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSP</td>
<td>Ease</td>
<td>KOAN/Fortran-S</td>
<td>LAM</td>
<td>NOW</td>
</tr>
<tr>
<td>BlockComm</td>
<td>ECO</td>
<td>Lilac</td>
<td>Lepa</td>
<td>Objective Linda</td>
</tr>
<tr>
<td>C*</td>
<td>Eiffel</td>
<td>Linda</td>
<td>JADA</td>
<td>Occam</td>
</tr>
<tr>
<td>&quot;C* in C&quot;</td>
<td>Eilean</td>
<td>WWWinda</td>
<td>ISETL-Linda</td>
<td>Omega</td>
</tr>
<tr>
<td>C++</td>
<td>Emerald</td>
<td>ParLin</td>
<td>ParLin</td>
<td>OpenMP</td>
</tr>
<tr>
<td>CarIoS</td>
<td>EPL</td>
<td>Eileen</td>
<td>Eileen</td>
<td>Orca</td>
</tr>
<tr>
<td>Cashmere</td>
<td>Excalibur</td>
<td>P4-Linda</td>
<td>P4-Linda</td>
<td>OOF90</td>
</tr>
<tr>
<td>C4</td>
<td>Express</td>
<td>POSTBL</td>
<td>POSTBL</td>
<td>P++</td>
</tr>
<tr>
<td>CC++</td>
<td>Falcon</td>
<td>Objective-Linda</td>
<td>Objective-Linda</td>
<td>P3L</td>
</tr>
<tr>
<td>Chu</td>
<td>Filaments</td>
<td>LiPS</td>
<td>Locust</td>
<td>Pablo</td>
</tr>
<tr>
<td>Charlotte</td>
<td>FM</td>
<td>Locus</td>
<td>Locust</td>
<td>Pade</td>
</tr>
<tr>
<td>Charm</td>
<td>FLASH</td>
<td>Links</td>
<td>Links</td>
<td>Padre</td>
</tr>
<tr>
<td>Charm++</td>
<td>The FORCE</td>
<td>Maisie</td>
<td>Maisie</td>
<td>Panda</td>
</tr>
<tr>
<td>Cid</td>
<td>Fork</td>
<td>Manifold</td>
<td>Manifold</td>
<td>Papers</td>
</tr>
<tr>
<td>Cilk</td>
<td>Fortran-M</td>
<td>AFAPI</td>
<td>AFAPI</td>
<td>Sage++</td>
</tr>
<tr>
<td>CM-Fortran</td>
<td>FX</td>
<td>GA</td>
<td>GA</td>
<td>SCANDAL</td>
</tr>
<tr>
<td>Converse</td>
<td>GAMMA</td>
<td>Glenda</td>
<td>Glenda</td>
<td>SAM</td>
</tr>
<tr>
<td>Code</td>
<td>COOL</td>
<td>Glenda</td>
<td>Glenda</td>
<td>Zounds</td>
</tr>
<tr>
<td>COOL</td>
<td></td>
<td></td>
<td></td>
<td>ZPL</td>
</tr>
</tbody>
</table>
Programming Models and Performance
Writing a [Parallel] Program

Original Problem

Decomposition

Tasks, shared and local data

Algorithm strategy

Implementation & building blocks

Units of execution + new shared data for extracted dependencies

Corresponding source code

Source: “Many Core Processors ... Opportunities and Challenges” by Tim Mattson
Designing Your Algorithm

• Does your knowledge of the underlying hardware change your task dependency graph? Your algorithm? If yes, how?

• Suppose you have several candidate algorithms for solving a problem, how do you pick?
Wishlist for a Good Algorithm

1. Good performance
2. On a wide range of parallel machines
3. Minimal tuning to hardware in early stage

We need an analytical model that can predict the performance of our algorithm on a wide range of machines and must strike a balance between detail and simplicity.
Why Programming Models?

- Abstract away complexity of real hardware
- Abstract away implementation details
  - Architecture
  - Language
  - OS
  - Libraries
- Focus on algorithmic choices
  - Don’t worry about implementation yet
- Define/compare asymptotic complexity
Classifying Parallel Programming Models

Parallel Programming Models

- Task Parallel
  - Message-Passing
  - Locking
- Data Parallel
  - Message Passing

MIMD

SPMD
Defining a Programming Model

- **Control**
  - How is parallelism created?
  - How are *dependencies* enforced?

- **Data**
  - Shared or private?
  - How is shared data accessed or private data communicated?

- **Synchronization / Communication**
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?
Any Paradigm on Any Hardware

- You can run any paradigm on any hardware (e.g. an MPI on shared-memory)
- The hardware itself can be heterogeneous

The whole challenge of parallel programming is to make the best use of the underlying hardware to exploit the different types of parallelism
Example

We have a matrix $A$. We need to form another matrix $A_{sqr}$ that contains the square of each element of $A$. Then we need to calculate $S$, which is the sum of the elements in $A_{sqr}$.

Thanks to Katherine Yelick.
Example

We have a matrix $A$. We need to form another matrix $A_{sqr}$ that contains the square of each element of $A$. Then we need to calculate $S$, which is the sum of the elements in $A_{sqr}$.

$A_{sqr}$: sum

$s$: square

• How can we parallelize this?
• How long will it take if we have unlimited number of processors?
Example

• First, decompose your problem into a set of tasks
  • There are many ways of doing it.
  • Tasks can be of the same, different, or undetermined sizes.

• Draw a task-dependency graph (do you remember the DAG we saw last time?)
  • A directed graph with nodes corresponding to tasks
  • Edges indicating dependencies, that the result of one task is required for processing the next.
Example

\[ A: \]
\[ A_{sqr} : \]
\[ s: \]
\[ \text{square} \]
\[ \text{sum} \]

\[ \text{sqr} (A[0]) \quad \text{sqr}(A[1]) \quad \text{sqr}(A[2]) \quad \ldots \quad \text{sqr}(A[n]) \]
\[ \text{sum} \]
Serial Programming Model

- The easy one we already know
  - aka RAM/Von Neumann/Turing
- One unlimited memory, constant access time
- Basis for classical algorithmic complexity theory
  - Complexity: number of instructions executed as a function of problem size
Serial Algorithms

• Can you give examples of serial algorithms and their complexity?
Serial Algorithms

- Can you give examples of serial algorithms and their complexity?
  - Binary search: $O(\log n)$
  - Quicksort: $O(n \log n)$
  - Factorization of $n$-digit number: $O(e^n)$
  - Multiplication of $n$-digit number:
    $O(n \log n) < O(n \log n 2^{O(\log^* n)}) < O(n \log n \log \log n)$
    - Conjectured lower bound
    - Furer 2007
    - Schonhage-Strassen 1971

- For parallel machines, need parallel models
Parallel Programming Models

- PRAM
- LogP
- BSP
- Multi-BSP
PRAM Model

- Parallel Random Access Machine
- 1 shared memory, N processors
- A synchronous MIMD
Access Conflicts: PRAM Machine Types

• What if processors try to read/write the same memory?

• Read:
  • Exclusive: Processors have to read different locations
  • Concurrent: Two processors can read the same location

• Write
  • Exclusive: Processors have to write different locations
  • Concurrent: Allow *reduction* operators (sum, AND, OR, ...)

• Types of PRAM machines: EREW, CREW, CRCW

• We’ll assume CREW for now
Reduction Example

• Given a vector $a$ of size $n$, compute the sum of elements with $n$ processors.
• Sequential algorithm

```plaintext
for i in [0 : n-1]:
    sum ← sum + a[i]
```

• All operations dependent: no parallelism
• Need to use associativity to reorder the sum
Parallel Reduction: Dependency Graph

- Compute $r = \ldots (a_0 \oplus a_1) \oplus (a_2 \oplus a_3) \oplus \ldots \oplus a_n) \ldots$
Parallel Reduction: PRAM Algorithm

- First step

Processor Pi:
\[ a[i] = a[2*i] + a[2*I + 1] \]
Parallel Reduction: PRAM Algorithm

- Second step
  - Reduction on n/2 processors
  - Other processors idle

Processor Pi:

\[ n' = \frac{n}{2} \]
\[ \text{if } i < n': \]
\[ a[i] = a[2*i] + a[2*I + 1] \]
Parallel Reduction: PRAM Algorithm

• Complete algorithm

Processor Pi:

while n > 1:
  if i < n:
    a[i] = a[2*i] + a[2*I + 1]
  n = n/2

• Complexity: 0(log(n)) for n CPUs
Brent’s Theorem

• Generalization of any PRAM algorithm of complexity $C(n)$ with $n$ processors.

• Derive an algorithm for $n$ elements, using $p$ processors

• Idea: each processor emulates $n/p$ virtual processors
  • Each emulated step takes $n/p$ actual steps

• Complexity: $O(n/p \times C(n))$
Interpretation of Brent’s Theorem

• We can design algorithms for a variable (unlimited) number of processors
  • And can then run them on any machine with fewer processors, with computable complexity
  • There may exist a more efficient algorithm when fewer processors are used
PRAM Model

• Can emulate a message-passing machine by partitioning memory into private memories.
• No communication cost (i.e. infinite bandwidth and zero latency).
• Infinite memory, unlimited parallelism – never simulate finite $P_n$!
• Different protocols can be used for reading and writing shared memory.
  • EREW - exclusive read, exclusive write: A program isn’t allowed to have two processors access the same memory location at the same time.
  • CREW - concurrent read, exclusive write
  • CRCW - concurrent read, concurrent write: Needs protocol for arbitrating write conflicts
  • CROW – concurrent read, owner write: Each memory location has an official “owner”
Pros/Cons of PRAM

+ Simple to use → can model many processors/hardware
- Unrealistic → performance prediction is inaccurate, very far from current implementations
BSP Model

• Bulk Synchronous Parallel
• N local memories, N processors, communication
  • Can also map to shared memory machines and GPUs
• Processors need to synchronize with each other
BSP Model

- BSP programs composed of supersteps
  - Each superstep consists of three ordered stages
- BSP needs a barrier synchronization method
BSP Model
Reduction: BSP

P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7

Computation
Communication
Sync
Computation
Communication
Sync
Computation
Communication
Sync
r
(Better) Reduction: BSP

![Diagram showing parallel processing and communication]

- Processes: $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7$
- Arrays: $a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7$
- Computation and communication synchronization

Sync
Communication
Computation
Sync
Communication
Computation
Sync
Communication
Computation
Sync

BSP Model

- **Variables**
  - \( p \): number of processors
  - \( s \): processor computation speed (flops/s)
  - \( h \): the maximum number of incoming or outgoing messages per processor
  - \( g \): the cost of sending a message.
  - \( l \): time to do a barrier synchronization

- Assume \( w_i \) is the computation time for work on processor \( i \) during a superstep.

- Cost of a superstep: ?
BSP Model

- **Variables**
  - \( p \): number of processors
  - \( s \): processor computation speed (flops/s)
  - \( h \): the maximum number of incoming or outgoing messages per processor
  - \( g \): the cost of sending a message.
  - \( l \): time to do a barrier synchronization

- Assume \( w_i \) is the computation time for work on processor \( i \) during a superstep.

- Cost of a superstep: \( \max_{i=1}^{p}(w_i) + \max_{i=1}^{p}(h_i g) + l \)
Pros/Cons of BSP Model

+ Simple
+ Predictable performance: accounts for synchronization cost
- Not very good if locality is important
- BSP does not distinguish between sending 1 message of length m, or m messages of length 1.
Multi-BSP Model

- Minimize communication cost on hierarchical platforms
  - e.g., cache sharing
Pros/Cons of Multi-BSP Model

+ Closer to modern architectures
+ Better for locality than BSP
- Similar communication cost estimate problems as BSP
LogP Model

- Distributed memory
- No specification of interconnection network
- Based on:
  - Latency of communication
  - Overhead in processing transmitted/received messages
  - Gap between consecutive transmissions (i.e., bandwidth limitation)
  - Processing power
LogP Model

Limited Volume ($L/g$ to or from a proc)

P (processors)

o (overhead)

L (latency)

Interconnection Network

g (gap)
Using the LogP Model

• One processor sends $n$ words to another processor
  
  $2o + L + g(n-1)$
Pros/Cons of the LogP Model

+ Simple, 4 parameters
+ Can easily be used to guide the algorithm development
- Does not take contention into account → can sometimes underestimate communication time.

There are many variations to the LogP model, making it more accurate but more complex (e.g. LogGP, logGPC, pLogP, ...).
Be Careful!

- All these models are just approximations.
- They do not model memory (finite memory, memory speed, cache), which can greatly affect their predictions.
  - There are memory models though.
- An implementation of a good parallel algorithm on a specific machine will surely require tuning. But first, pick/design an algorithm based on one of the models discussed.
Conclusions

• Modeling the algorithm for a general class of system simplifies choosing the best approach
  • PRAM, BSP, LogP are common models

• Even programming models allow optimization for memory usage, communication, etc.