Lecture 2: Concurrency and Parallelism

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Same Meaning?

- **Concurrency**: At least two tasks are making progress at the same time frame.
  - Not necessarily at the same time
  - Include techniques like time-slicing
  - Can be implemented on a single processing unit
  - Concept more general than parallelism

- **Parallelism**: At least two tasks execute literally at the same time.
  - Requires hardware with multiple processing units
A Quick Example

This Figure shows a server receiving two requests at time 0. How will this server react if it is serial and if it is concurrent? [Hint: look at total completion time and average completion time]
Concurrency without parallelism

After you
No, no, after you!

Profiler

Yo Bill

Concurrency with parallelism

Hey Joe

Performance tuning technique number 106: Concurrency vs. Parallelism

Copyright © Fasterj.com Limited
Simply Speaking

Concurrency + Parallelism = High Performance
Questions!

If we have as much hardware as we want, do we get as much parallelism as we wish?

If we have 2 cores, do we get 2x speedup?
Amdahl's Law

• How much of a speedup one could get for a given parallelized task?

If $F$ is the fraction of a calculation that is sequential then the maximum speed-up that can be achieved by using $P$ processors is $\frac{1}{F + \frac{1-F}{P}}$. 

Gene M. Amdahl
What Was Amdahl Trying to Say?

• Don’t invest blindly on large number of processors.
• Sometimes having faster cores (or processor at his time) makes more sense than having many cores.

Was he right?
• At his days (the law appeared 1967) many programs had long sequential parts.
• This is not necessarily the case nowadays.
• It is not very easy to find F (sequential portion)
So …

• Decreasing the serialized portion is of greater importance than adding more cores blindly.

• Only when a program is mostly parallelized, does adding more processors help more than parallelizing the remaining rest.

• **Gustafson’s law**: computations involving arbitrarily large data sets can be efficiently parallelized.

• Both Amdahl and Gustafson do not take into account:
  – The overhead of synchronization, communication, OS, etc.
  – Load may not be balanced among cores.

• So you have to use these laws as guideline and theoretical bounds only.
**DAG Model**

**Work**: total amount of time spent on all instructions

\[ T_p = \text{The fastest possible execution time on P processors} \]

**Work Law**: \[ T_p \geq \frac{T_1}{P} \]
DAG Model

**Span**: The longest path of dependence in the DAG = $T_\infty$

**Span Law**: $T_p \geq T_\infty$
Can We Define Parallelism Now?

How about? \( \frac{T_1}{T_\infty} \)

Ratio of work to span
Can We Define Parallelism Now?

**Work:** $T_1 = 50$

**Span:** $T_\infty = 8$

**Parallelism:** $\frac{T_1}{T_\infty} = 6.25$
At What Level Can We Reason About Parallelism (algorithm, high-level language, assembly, hardware)?
Parallelism Granularity

• Instruction level parallelism (ILP)
  – Pipelining
  – Superscalar
  – Out-of-order execution
  – Speculative execution

• **Thread** level parallelism
  – Hyperthreading technology (aka SMT)
  – Multicore

• **Process** level parallelism
  – Multiprocessor system
  – Hyperthreading technology (aka SMT)
  – Multicore
That Was The Software
How about the Hardware?

Latency Vs Throughput
Flynn Classification

• A taxonomy of computer architecture
• Proposed by Michael Flynn in 1966
• It is based on two things:
  – Instructions
  – Data

<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>
PU = Processing Unit
More About MIMD

Shared-Memory

Distributed-Memory

Or 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

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 ... Doesn't work!
The Mentality of Yet Another Programming Language ... Doesn't work!

<table>
<thead>
<tr>
<th>Mentality of Yet Another Programming Language</th>
<th>Doesn't work!</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ABCPL</th>
<th>CORRELATE</th>
<th>GLU</th>
<th>GUARD</th>
<th>Legion</th>
<th>Mentat</th>
<th>Parafrase2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>CPS</td>
<td>GUARD</td>
<td>HASKELL</td>
<td>HPC++</td>
<td>IMPACT</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>ACT++</td>
<td>CRL</td>
<td>HASSL</td>
<td>JAVAR.</td>
<td>JAVAR.</td>
<td>ISIS.</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>Active messages</td>
<td>CSP</td>
<td>HPC</td>
<td>MPc</td>
<td>MOSIX</td>
<td>Modulea-P</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>Adl</td>
<td>Chthreads</td>
<td>IMPACT</td>
<td>MIRAGE</td>
<td>Modulea-P</td>
<td>Multinol</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>Adsmith</td>
<td>CUMULUS</td>
<td>IMPACT</td>
<td>MIRAGE</td>
<td>Modulea-P</td>
<td>Multinol</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>ADDAP</td>
<td>DAGGER</td>
<td>HPC</td>
<td>MPc</td>
<td>MOSIX</td>
<td>Modulea-P</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>AFAPI</td>
<td>DAPPLE</td>
<td>IMPACT</td>
<td>MIRAGE</td>
<td>Modulea-P</td>
<td>Multinol</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>ALWAN</td>
<td>Data Parallel C</td>
<td>IMPACT</td>
<td>MIRAGE</td>
<td>Modulea-P</td>
<td>Multinol</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>AM</td>
<td>DC++</td>
<td>IMPACT</td>
<td>MIRAGE</td>
<td>Modulea-P</td>
<td>Multinol</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>AMDC</td>
<td>DCE++</td>
<td>IMPACT</td>
<td>MIRAGE</td>
<td>Modulea-P</td>
<td>Multinol</td>
<td>Parafrase2</td>
</tr>
<tr>
<td>ArmLaS</td>
<td>DDD</td>
<td>IMPACT</td>
<td>MIRAGE</td>
<td>Modulea-P</td>
<td>Multinol</td>
<td>Parafrase2</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>C</th>
<th>Eileen</th>
<th>Linda</th>
<th>Occam</th>
<th>PRESTO</th>
<th>TreadMarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;C&quot; in C</td>
<td>Emerald</td>
<td>JADA</td>
<td>Omega</td>
<td>P-RIO</td>
<td>TRAPPER</td>
</tr>
<tr>
<td>C++</td>
<td>EPL</td>
<td>WWWinda</td>
<td>OpenMP</td>
<td>Prospero</td>
<td>uC++</td>
</tr>
<tr>
<td>CarlOS</td>
<td>Excalibur</td>
<td>ISETL-Linda</td>
<td>Orca</td>
<td>Proteus</td>
<td>UNITY</td>
</tr>
<tr>
<td>Cashmere</td>
<td>Express</td>
<td>ParLin</td>
<td>OOF90</td>
<td>QPC++</td>
<td>UC</td>
</tr>
<tr>
<td>C4</td>
<td>Falcon</td>
<td>Eileen</td>
<td>P++</td>
<td>PVM</td>
<td>V</td>
</tr>
<tr>
<td>CC++</td>
<td>Filaments</td>
<td>P4-Linda</td>
<td>P3L</td>
<td>PSI</td>
<td>ViC*</td>
</tr>
<tr>
<td>Chu</td>
<td>FM</td>
<td>POSYBL</td>
<td>Pablo</td>
<td>PSDKM</td>
<td>Visifold V-NUS</td>
</tr>
<tr>
<td>Charlotte</td>
<td>FLASH</td>
<td>Objective-Linda</td>
<td>PADE</td>
<td>Quake</td>
<td>VPE</td>
</tr>
<tr>
<td>Charm</td>
<td>The FORCE</td>
<td>LiPS</td>
<td>PADRE</td>
<td>Quark</td>
<td>Win32 threads</td>
</tr>
<tr>
<td>Charm++</td>
<td>Fork</td>
<td>Locust</td>
<td>Panda</td>
<td>Quick Threads</td>
<td>WinPar</td>
</tr>
<tr>
<td>Cid</td>
<td>Filament</td>
<td>Lpars</td>
<td>Papers</td>
<td>Sage++</td>
<td>XENOOPS</td>
</tr>
<tr>
<td>Cilk</td>
<td>Fortran-M</td>
<td>Lucid</td>
<td>Papers</td>
<td>SCANDAL</td>
<td>XPC</td>
</tr>
<tr>
<td>CM-Fortran</td>
<td>FX</td>
<td>Maisie</td>
<td>Papers</td>
<td>SAM</td>
<td>Zounds</td>
</tr>
<tr>
<td>Converse</td>
<td>GA</td>
<td>Manifold</td>
<td>Papers</td>
<td>SAM</td>
<td>ZPL</td>
</tr>
<tr>
<td>Code</td>
<td>GAMMA</td>
<td>Glenda</td>
<td>PARA++</td>
<td>Paradigm</td>
<td>ZPL</td>
</tr>
<tr>
<td>COOL</td>
<td>Glenda</td>
<td>Glenda</td>
<td>PARA++</td>
<td>Paradigm</td>
<td>ZPL</td>
</tr>
</tbody>
</table>
Programming Model

- **Definition**: the languages and libraries that create an abstract view of the machine
- **Control**
  - How is parallelism created?
  - How are dependencies enforced?
- **Data**
  - Shared or private?
  - How is shared data accessed or private data communicated?
- **Synchronization**
  - What operations can be used to coordinate parallelism
  - What are the atomic (indivisible) operations?
It Is Important to Note

• 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 $A_{sq}$ that contains the square of each element of $A$. Then we need to calculate $S$, which is the sum of the elements in $A_{sq}$.

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

slide derived from Katherine Yelick
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 earlier?)
  – A directed graph with Nodes corresponding to tasks
  – Edges indicating dependencies, that the result of one task is required for processing the next.

slide derived from Katherine Yelick
Example

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

slide derived from Katherine Yelick
Source: “Many Core Processors ... Opportunities and Challenges” by Tim Mattson
Does your knowledge of the underlying hardware change your task dependency graph? If yes, how?
Suppose you have several candidate algorithms for solving a problem, how do you pick?
Wish List for a Good Algorithm for Parallelism

- Good performance
- On a wide range of parallel machines
- Without going into many details about the hardware at that early stage
- Scalable

This means: We need a computational model that can predict the performance of our algorithm on a wide range of machines and must strike a balance between detail and simplicity.
Three Main Computational Models

- PRAM
- BSP
- LogP
• Parallel Random Access Machine
• Shared memory
• A synchronous MIMD
PRAM Model

- No communication cost (i.e. infinite bandwidth and zero latency).
- Infinite memory
- Can emulate a message-passing machine by partitioning memory into private memories.
- 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
- Unrealistic → performance prediction is inaccurate
**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

\[ P \ (\text{processors}) \]

\[ o \ (\text{overhead}) \]

\[ L \ (\text{latency}) \]

\[ g \ (\text{gap}) \]

Interconnection Network
Using the LogP model

- Two processors send $n$ words to each other:
  $$2o + L + g(n-1)$$
Pros/Cons of the LogP model

+ Simple, 4 parameters
+ Can easily be used to guide the algorithm development
- Does not take contention into account \(\rightarrow\) 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, ...)

BSP Model

- Bulk Synchronous Parallel

A BSP computer consists of
- A set of processor-memory pairs
- A communication network that delivers messages in a point-to-point manner
- Mechanism for barrier synchronization for all or a subset of the processes

BSP programs composed of supersteps
- Each superstep consists of three ordered stages:
  - Computation
  - Communication
  - Barrier synchronization
BSP Model

Virtual Processors

Local Computation

Global Communication

Barrier Synchronization
BSP Model

- $p$: 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
Using BSP Model

• 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
- Not very good if locality is important
- BSP does not distinguish between sending 1 message of length m, or m messages of length 1.
Be Careful!

- All these models are just approximations.
- They do not model memory 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

- Concurrency and parallelism are not exactly the same thing.
- There is parallelism at different granularities, with methods to exploit each parallelism granularity.
- Knowing the hardware will help you generating a better task dependency graph.