Lecture 2: Concurrency and Parallelism

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
• **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. 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

Concurrency with parallelism

Performance tuning technique number 106: Concurrency vs. Parallelism

Copyright © Fasterj.com Limited
Concurrency + Parallelism = 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 $\frac{1}{F+(1-F)/P}$
What Was Amdahl Trying to Say?

• Don’t invest blindly on large number of processors.
• 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.
- 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 for Multithreading

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

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

**Work Law:**

\[ T_p \geq T_1/P \]
DAG Model for Multithreading

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?

\[
\text{Work: } T_1 = 50 \\
\text{Span: } T_\infty = 8 \\
\text{Parallelism: } T_1 / T_\infty = 6.25
\]
At What Level Can We Reason About Parallelism (algorithm, high-level language, assembly)?
Is Thread The Only Parallelism Granularity?

• Instruction level parallelism (ILP)
  – 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
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!

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
The Mentality of Yet Another Programming Language ... Doesn’t work!
Parallel Programming Models

- Parallel Programming Models
  - Control Flow
    - Message-Passing
    - Locking
  - Data-Parallel
    - Message Passing

MIMD

SPMD or SIMD (GPUs!)
**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 type of parallelisms
Example

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

• 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

A:

Asqr:

slide derived from Katherine Yelick
Does your knowledge of the underlying hardware change your task dependency graph? If yes, how?
Conclusions

• Concurrency and parallelism are not exactly the same thing.
• There is parallelism at different granularities, with methods to exploit each parallelism granularity.
• You need to know the difference among: threads/processors/tasks.
• Knowing the hardware will help you generating a better task dependency graph.