Lecture 6: Parallel Software: Advanced

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Concurrency Vs Parallelism: 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
Concurrency without parallelism

After you
No, no, after you!

Concurrency with parallelism

Profiler
Yo Bill

Performance tuning technique number 106: Concurrency vs. Parallelism

Copyright © Fasterj.com Limited
Simply Speaking

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 core (or processor at his time) makes more sense than having many cores.

Was he right?
- At his days (the law appeared 1967) many programs have 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.
- Amdahl does not take into account:
  - The overhead of synchronization, communication, OS, etc.
  - Load may not be balanced among cores.
- So you have to use this law 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?

- **Work**: $T_1 = 50$
- **Span**: $T_\infty = 8$
- **Parallelism**: $T_1/T_\infty = 6.25$
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: \[ \text{square} \]

Asqr: \[ \text{sum} \]

\[ s: \]

\[ \text{sum} \]

\[ \text{slide derived from Katherine Yelick} \]
Does your knowledge of the underlying hardware change your task dependency graph? If yes, how?
Never Forget ...

- Parallel programming is not only about finding a lot of parallelism.
- Critical section and atomic operations
  - Race condition
  - Correctness vs performance loss
- Know your tools: language, compiler and hardware
Sources of Performance Loss in Parallel Programs

- Extra overhead
  - load
  - synchronization
  - communication
- Artificial dependencies
  - Hard to find
  - May introduce more bugs
  - A lot of effort to get rid of
- Contention due to hardware resources
- Coherence
- Load imbalance
Artificial Dependencies

```c
int result;
//Global variable
for (...) // The OUTER loop
    modify_result(...);
    if(result > threshold)
        break;

void modify_result(...) {
    ...
    ...
    result = ...
```

What is wrong with that program when we try to parallelize it?
Coherence

- Extra bandwidth (scarce resource)
- Latency due to the protocol
- False sharing
Load Balancing

Time

synchronization

work

work

work

idle

idle

synchronization
Load Balancing

• Assignment of work not data is the key
• If you cannot eliminate it, at least reduce it.
• Static assignment
• Dynamic assignment
  – Has its overhead
Patterns in Parallelism

- Task-level (e.g. Embarrassingly parallel)
- Divide and conquer
- Pipeline
- Iterations (loops)
- Client-server
- Geometric (usually domain dependent)
- Hybrid (different program phases)
Task Level

Independent Tasks

A  B  C  D  E

A  B  C  E  D
Client-Server/ Repository

Compute A
Compute B
Compute C
Compute D
Compute E

Asynchronous Function calls
Assume we have a large array and we want to compute its minimum (T1), average (T2), and maximum (T3).

```c
#define maxN 1000000000
int m[maxN];
int i;
int min = m[0];
int max = m[0];
double avrg = m[0];
for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
    avrg = avrg + m[i];
    if(m[i] > max)
        max = m[i];
}  
avrg = avrg / maxN;
```
Divide-And-Conquer

- Compute subproblem
- Compute subproblem
- Compute subproblem
- Compute subproblem

- Split
- Split
- Split

- Merge
- Merge
- Merge

- Subproblem
- Subproblem
- Subproblem
- Subproblem

- Solution
A series of ordered but independent computation stages need to be applied on data.
Pipeline

• Useful for
  – streaming workloads
  – Loops that are hard to parallelize
    • due inter-loop dependence
• Usage for loops: split each loop into stages so that multiple iterations run in parallel.
• Advantages
  – Expose intra-loop parallelism
  – Locality increases for variables uses across stages
• How shall we divide an iteration into stages?
  – number of stages
  – inter-loop vs intra-loop dependence
Conclusions

• Concurrency and parallelism are not exactly the same thing.
• You need to know the difference among: threads/processors/tasks.
• Knowing the hardware will help you generate a better task dependency graph → dependency graph in turn helps you reason about parallelism in your code.