Concurrency and Performance

ICS432 - Fall 2008
Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Concurrency and Performance

- You want to make your program concurrent to exploit the concurrent features of your computer architecture as best as possible
  - With the goal of accelerating the computation
- Let us define measures of "parallel performance"
  - The terms parallel and concurrent are basically synonyms (although for multi-threading the terms concurrent is preferred)

Parallel Speedup

- The simplest way to measure how well a parallel program performs is to compute its *parallel speedup*
- Parallel program takes time T_1 on 1 core
- Parallel program takes time T_p on p cores

$$Speedup(p) = T_1 / T_p$$

Definition for T_1?

- What is T_1?
  - The time to run a sequential version of the program?
  - The time to run the parallel version, but using one processor?
- The two can be different
  - e.g., Writing a code so that it can work in parallel can require a less efficient algorithm
- In practice, most people use the second definition above

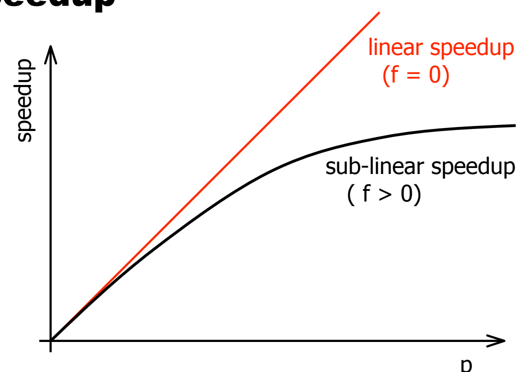
Amdahl's Law

- A parallel program always has a sequential part (e.g., I/O) and a parallel part
  - $T_1 = f\,T_1 + (1-f)T_1$
  - $T_p = f\,T_1 + (1-f)T_1 / p$
- Therefore:

$$Speedup(p) = 1 / (f + (1-f)/p) = p / (f\,p + 1 - f) \leq 1 / f$$

- Example: if a code is 10% sequential (i.e., f = .10), the speedup will always be lower than 1 + 90/10 = 10, no matter how many cores are used

Speedup



Parallel Efficiency

- $Eff_p = S_p / p$
- Generally lower than 1
  - We'll talk about "superlinear" speedups in a later lecture if we have time
- Used to measure how well the cores are utilized
  - If increasing the number of cores by a factor 10 increases the speedup by a factor 2, perhaps it's not worth it: efficiency drops by a factor 5
  - Typically one "pays" for adding computing resources
    - e.g., each extra core is expensive

A Trivial Example

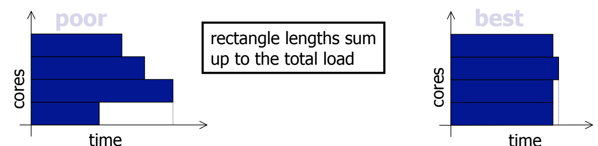
- Let's say I want to compute $\sum_{i=0}^{n-1} f(a[i])$
- I have p cores P0, P1, ... Pp-1 (p > n)
- I first need to figure out how to partition the computation among the cores
  - Which cores does what?
- There are multiple options:
  - A bad option
    - core 1 computes 1 value
    - core 2 computes 1 value
    - core p-1 computes 1 value
    - core p computes the remaining n-p+1 values and adds them up all together
  - It is a bad option because of "load balancing"
    - Core p has more work to do than the others

A Trivial Example

- Load Balancing = balancing the amount of computation done by all the cores
  - "load" often used for "amount of computation to perform"
- Perfect Load Balance: all cores have the same load to compute
  - This way one doesn't have to wait for the "slowest" core, i.e., the one that was given the biggest load.

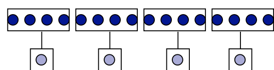
A Trivial Example

- Load Balancing = balancing the amount of computation done by all the cores
  - "load" often used for "amount of computation to perform"
- Perfect Load Balance: all cores have the same load to compute
  - This way one doesn't have to wait for the "slowest" core, i.e., the one that was given the biggest load.



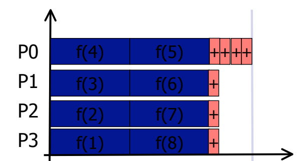
A Trivial Example

- Why is a good partitioning of the computation to compute? $\sum_{i=0}^{n-1} f(a[i])$
  - n evaluations of function f
  - n-1 sums
- Partial sum partitioning
  - Each core computes n/p values of f
  - Each core adds the value it has computed
  - One core, say P0, adds all these values together
- Assuming that p divides n
  - Core P0: n/p values of f, n/p additions, p-1 additions
  - Core P1, P2, ..., Pp-1: n/p values of f, n/p additions



A Trivial Example

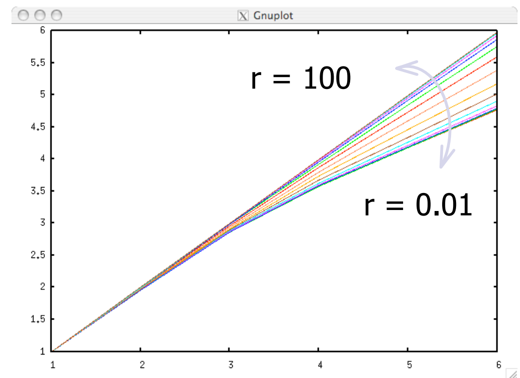
- Let α be the time taken to evaluate f
- Let β be the time to perform an addition
- Execution time = $\alpha n/p + \beta(n/p - 1) + \beta(p-1))$
- Execution time= $\alpha n/p + \beta(n/p + p - 2)$
  - The execution completes when the last core has finished computing



A Trivial Example

- What is the speedup?
  - Speed up = execution time on 1 core / execution time on p core
- Speedup = $(\alpha n + \beta n) / (\alpha n/p + \beta(n/p + p\text{-}2))$
- Speedup = $((\alpha/\beta)n + n)/((\alpha/\beta)n/p + n/p + p\text{-}2)$
  - I introduced the α/β ratio, which denotes how much more expensive is an evaluation of function f when compared to an addition
- What does the speedup look like?
  - Let's fix n = 100
  - Let's vary p from 1 to 6
  - Let's try a few values of $r = \alpha/\beta$

A Trivial Example

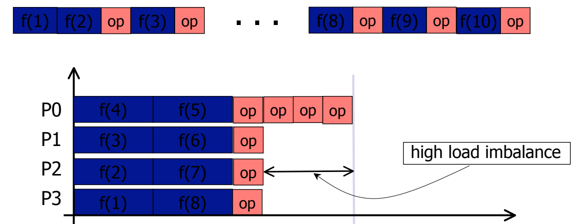


A Trivial Example

- If (α/β) is high, nearly optimal speed-up
  - meaning we had a nearly optimal load balance
  - meaning that we had a nearly optimal decomposition of the computation
  - should be the case for our example, as adding two number should be fast
- If (α/β) is very low, we don't do so well
  - could happen if instead of '+' we have some other operator, in case function f returns matrices as opposed to individual numbers
  - we have a poor computation decomposition because one processor does more of the expensive operations than the others

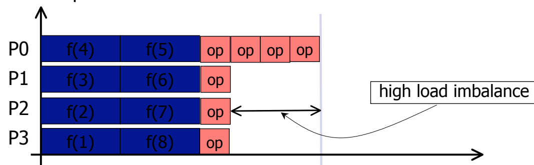
Trivial Example

- What does the execution look like when (α/β) is low?
- Example
  - Let's say that n = 8 and p = 4
  - We apply a commutative/associative operator op



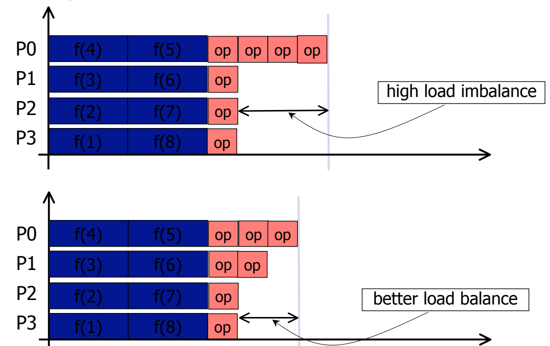
Trivial Example

- Better Decomposition



Trivial Example

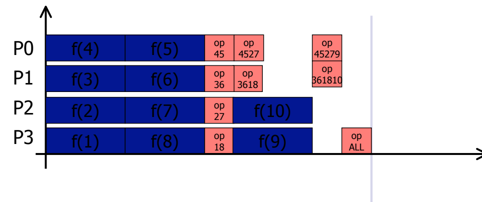
- Better Decomposition



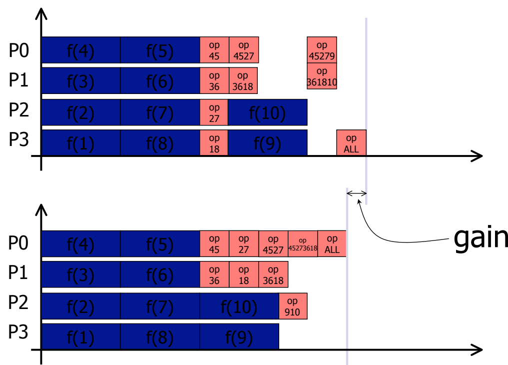
A Trivial Example

- Note that we assumed that p divides n. If it doesn't, then some cores will compute more values of f than some others
  - There will be n mod p "left over" function evaluations
  - Given each to a different core
  - Maybe use the core that doesn't have any left over evaluation do the global sum?
- Example
  - Let's say that n = 10 and p = 4
  - We apply a commutative/associative operator op

A Trivial Example?



A Trivial Example?



So what? (2)

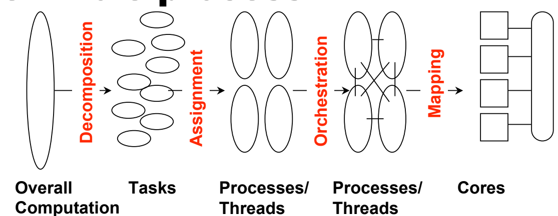
- What we have NOT learned from the "trivial" example: How do we write code that allows us to "shuffle" computations among different processors

it depends of the programming model

Creating a Parallel Program

- "Parallelizing"
  - identify pieces of work that can be done in parallel
  - assign them to threads of control
  - orchestrate data access, communication, synchronization
  - map threads of control to processors
- Goal: maximize parallel speedup

The whole process



- **Task:** defined pieces of work that are the basic concurrent unit
  - fine grain, coarse grain
- **Thread (of control):** abstract entity that performs tasks
  - tasks are assigned to them
  - they must coordinate
- **Cores:** physical entity that executes a thread of control
  - threads of control must be assigned to them

Conclusion

- Parallelization can be very straightforward, or very difficult
  - Typically difficult when the code was written without concurrency in mind
  - Generations of programmers have never been trained to think of concurrency
  - This has to change
- In the next lecture we look at some standard parallelization schemes