

Principles of High Performance Computing (ICS 632)

Concurrency and Measures of Performance

Concurrency and Performance

- You have an application
- You have identified expensive functions
- You have removed optimization blockers
- You must now **make your program concurrent** to exploit the concurrent features of your computer architecture as best as possible
- We mean measures of “parallel performance”
 - The terms parallel and concurrent are basically synonyms (although for multi-threading the terms concurrent is sometimes 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 processor
- Parallel program takes time T_p on p processors

$$\text{Speedup}(p) = T_1 / T_p$$

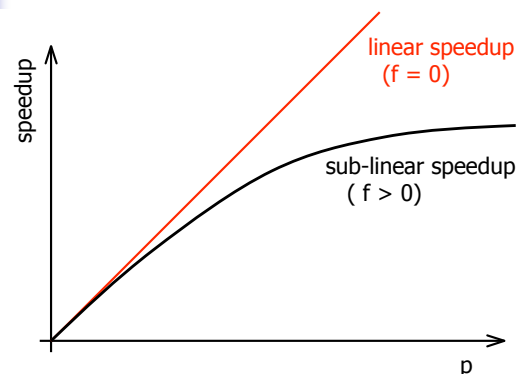
Definition for T_1 ?

- 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:
$$\begin{aligned} \text{Speedup}(p) &= 1 / (f + (1-f)/p) \\ &= p / (f p + 1 - f) \\ &\leq 1 / f \end{aligned}$$
- 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 processors are used

Speedup



Parallel Efficiency

- Eff_p = S_p / p
- Generally lower than 1
 - We'll talk about "superlinear" speedups in a later lecture
- Used to measure how well the processors are utilized
 - If increasing the number of process 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

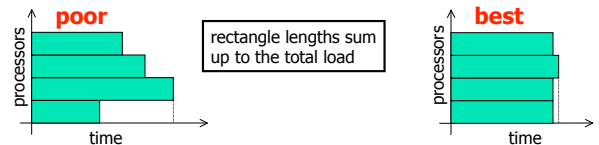
- Let's say I want to compute $\sum_{i=0}^{n-1} f(a[i])$
- I have p processors P0, P1, ... Pp-1 (p < n)
- I first need to figure out how to partition the computation among the processors
 - Which processor does what?
- There are multiple options:
 - A bad option
 - processor 1 computes 1 value
 - processor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

A Trivial Example

- Load Balancing = balancing the amount of computation done by all the processors
 - "load" often used for "amount of computation to perform"
 - probably one of the most overloaded terms in computer science
- Perfect Load Balance: all processors have the same load to compute
 - This way one doesn't have to wait for the "slowest" processor, i.e., the one that was given the biggest load.

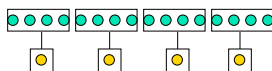
A Trivial Example

- Load Balancing = balancing the amount of computation done by all the processors
 - "load" often used for "amount of computation to perform"
 - probably one of the most overloaded terms in computer science
- Perfect Load Balance: all processors have the same load to compute
 - This way one doesn't have to wait for the "slowest" processor, i.e., the one that was given the biggest load.



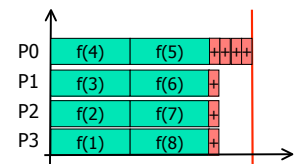
A Trivial Example

- 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 processor computes n/p values of f
 - Each processor adds the value it has computed
 - One processor, say P0, adds all these values together
- Assuming that p divides n
 - Processor P0: n/p values of f, n/p additions, p-1 additions
 - Processor P1, P2, ..., Pp-1: n/p values of f, n/p additions



A Trivial Example

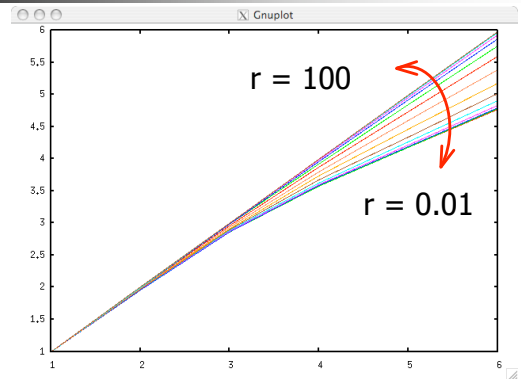
- Let α be the time taken to evaluate f
- Let β be the time to perform an addition
- Execution time = $\alpha n/p + \beta(n/p - 1) + \beta(p-1)$
- Execution time = $\alpha n/p + \beta(n/p + p - 2)$
 - The execution completes when the last processor has finished computing



A Trivial Example

- What is the speedup?
 - Speed up = execution time on 1 processors / execution time on p processor
- Speedup = $(\alpha n + \beta(n-1)) / (\alpha n/p + \beta(n/p + p-2))$
- Speedup = $((\alpha/\beta)n + n - 1) / ((\alpha/\beta)n/p + n/p + p-2)$
 - 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

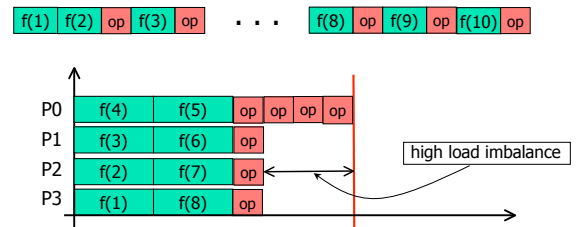


A Trivial Example

- 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

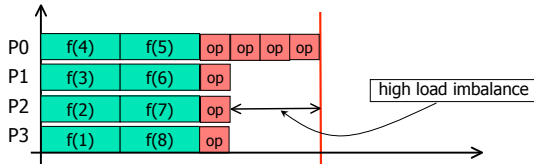
Trivial Example

- 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



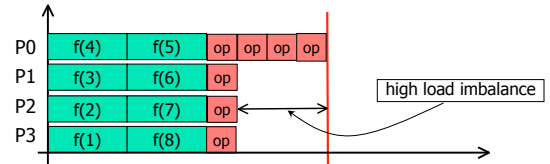
Trivial Example

- Naïve version

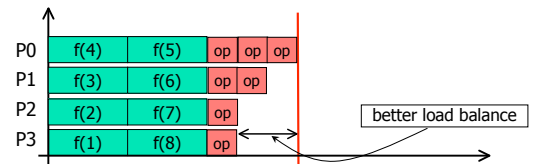


Trivial Example

- Naïve version



- Better version

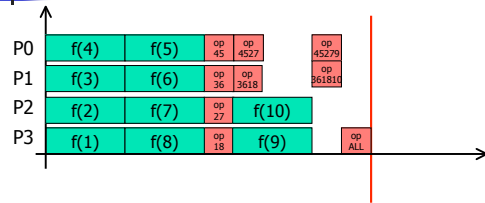


A Trivial Example

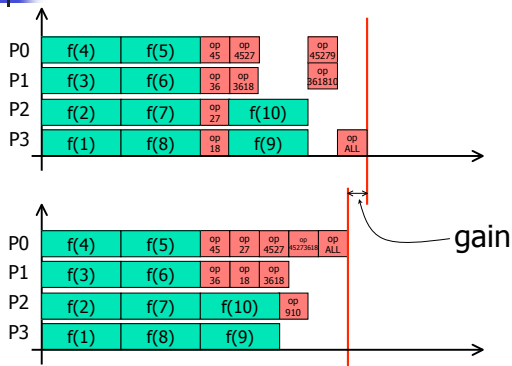
- Note that we assumed that p divides n . If it doesn't, then some processors will compute more values of f than some others
 - There will be $n \bmod p$ "left over" function evaluations
 - Given each to a different processor
 - Maybe use the processor that doesn't have any left over evaluation do the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

f(1) f(2) op f(3) op . . . f(8) op f(9) op f(10) op

A Trivial Example?



A Trivial Example?



So what?

- What have we learned from the "trivial" example?
- The objective is to assign units of computation to processors (*without disrupting the correctness of the program*) in a way that leads to good performance
 - called **computation decomposition**
- Things can become non-trivial very quickly
 - not so trivial question: given a , b , p , and n , what is the optimal execution data decomposition?
 - or: is there a computation decomposition that gets within 5% of optimal no matter what the values of a , b , p , and n are?
 - Answers to such questions may require a lot of work, and these problems are often NP-hard
 - And this is for a very straightforward application
 - The more theoretical side of parallel computing is not easy
 - Luckily, we'll look mostly at practice and not theory

So what? (2)

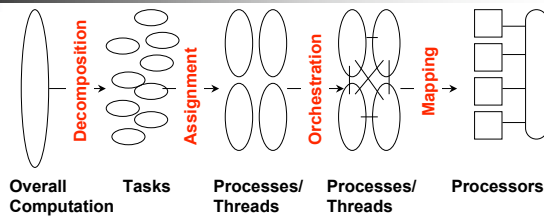
- What we have NOT learned from the "trivial" example: How do we write code that allows us to "shuffle" computations among different processors

it depends of the programming model

Creating a Parallel Program

- Pieces of the job
 - 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
- **Processor:** physical entity that executes a thread of control
 - threads of control must be assigned to them

Identify Tasks

- Break the overall computation into tasks
 - identify concurrency and decide at what level to exploit it
 - concurrency may be statically identifiable or may vary dynamically!
 - it may depend on program size, on data structures, on algorithms.
- Goal:
 - enough tasks to keep all processors busy
 - not so many tasks that they are all tiny, which could lead to high overheard

Assigning tasks

- Determine mechanism to assign tasks to threads of control
 - Functional partitioning: assign logically distinct aspects of work to different thread of control
 - e.g., pipelining
 - Structural partitioning: assign a subset of the set of (identical) tasks to each thread of control
 - e.g., our trivial example
 - Data/Domain decomposition: break up data structures into regions and assign work the threads of control
 - e.g., parts of a physical system being simulated
- Goal:
 - Load balancing

Orchestration

- Provide means to
 - name and access shared data
 - communicate and coordinate among threads of control
- Goal
 - correctness
 - avoid serialization (Amdahl's law)
 - reduce cost of synchronization
 - preserve locality of reference as much as possible to reuse caches

The trivial example

- $s = f(A[1]) + \dots + f(A[n])$
- Decomposition
 - computing each $f(A[j])$
 - n-fold parallelism, where n may be $\gg p$
 - computing sum s
- Assignment
 - thread k sums $s_k = f(A[k*n/p]) + \dots + f(A[(k+1)*n/p-1])$
 - thread 1 sums $s = s_1 + \dots + s_p$ (for simplicity of this example)
 - thread 1 communicates s to other threads
- Orchestration
 - starting up threads
 - communicating, synchronizing with thread 1
- Mapping
 - processor j runs thread j

Conclusion

- The same principles will apply for distributed-memory computing
 - Just more constraints and more complicated models to account for the network
- Ultimately, these types of problems will lead us to the issue of *scheduling*