Parallel Computing Basics

Nima Honarmand

Reading assignments

- For Thursday, 9/3, read and discuss all the papers in the first batch (both required and optional)
  - Except the “Referee” paper; just read it. No discussion needed on that one.
- Each student should discuss each paper with at least 2 posts-per paper
- DISCUSS! Do not summarize!

Note

- Most of the theoretical concepts presented in this lecture were developed in the context of HPC (high performance computing) and Scientific applications
- Hence, they are less useful when reasoning about server and datacenter workloads
- A lot more fundamental work is needed in that domain
  - Especially in terms of computation models and performance debugging and tuning techniques
- Yeay, research opportunity!!!

Task Dependence Graph (TDG)

- Let's model a computation as a DAG
  - DAG = Directed Acyclic Graph
- Classical view of parallel computations; still useful in many areas
- Nodes are tasks
- Edges are dependences between task
- Each tasks is a sequential unit of computation
  - Can be an instruction, or a function, or something bigger
- Each task has a weight, representing the time it takes to execute

Task Decomposition

- ***Task Decomposition***: dividing the work into multiple tasks
  - Often, there are many valid decompositions (TDGs) for a given computation

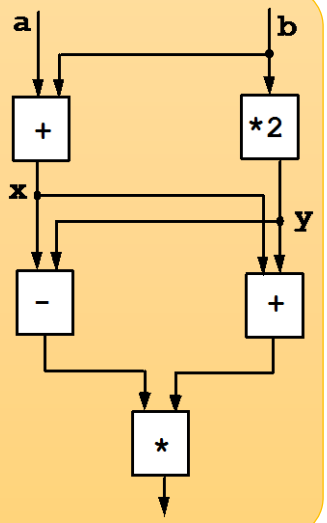
Static vs. ***dynamic***

- ***Static***: decide the decomposition at the beginning of the program computation
- **Dynamic**: decide the decomposition dynamically, based on the input characteristics
  - E.g., when exploring a graph whose shape

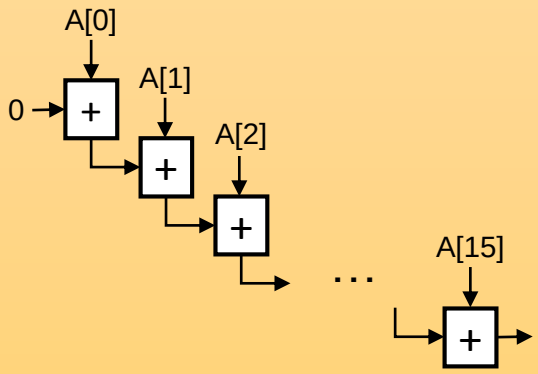
Task Decomposition Granularity

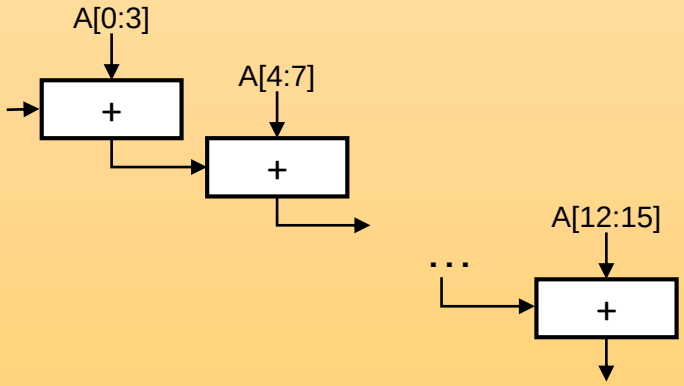
- Granularity = task size
  - depends on the number of tasks
- Fine-grain = large # of tasks
- Coarse-grain = small # of tasks

```
x = a + b;
y = b * 2
z =(x-y) * (x+y)
```



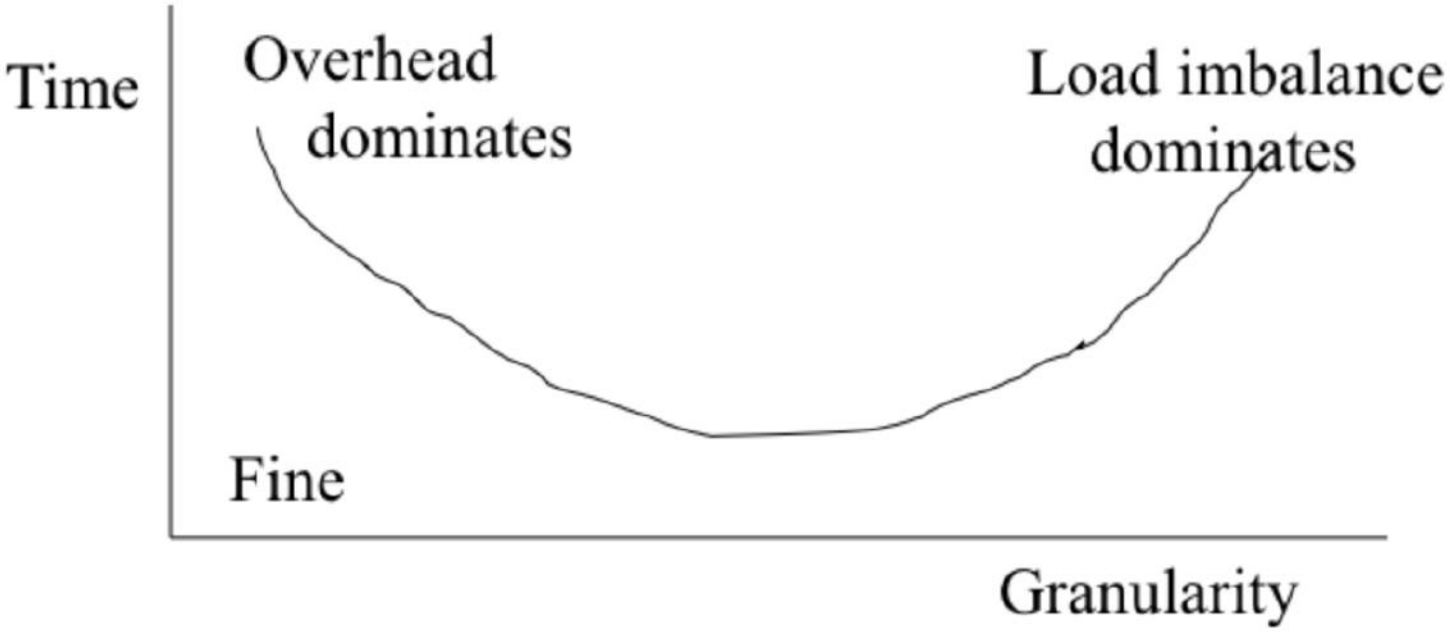

```
c = 0;
For (i=0; i<16; i++)
    c = c + A[i]
```





Bathtub Graph

- Typical graph of execution time using *p* processors
  - Overhead = communication + synchronization + excess work



Mapping and Scheduling (M&S)

- **Mapping and Scheduling**: determine the assignment of the tasks to processing elements (mapping) and the timing of their execution (scheduling)

Static vs. Dynamic M&S

- Sometimes, one can statically assign tasks to processors (reduce overhead)
  - if grain size is constant and the number of tasks is known
- Otherwise, one needs some dynamic assignment
  - task queue
  - self-scheduled loop, …

Goals of Decomposition and M&S

- Maximize parallelism, i.e., number of tasks that can be executed in parallel at any point of time
- Minimize communication
- Minimize load imbalance
  - ***Load imbalance*** : assigning different amount of work to different processors
  - Metric: total idle time across all processors
- Typically opposing goals ☹
  - parallelism↑ vs. communication↓
  - load imbalance↓ vs. communication↓
  - However, parallelism↑ and load imbalance↓ often compatible

Basic Measures of Parallelism

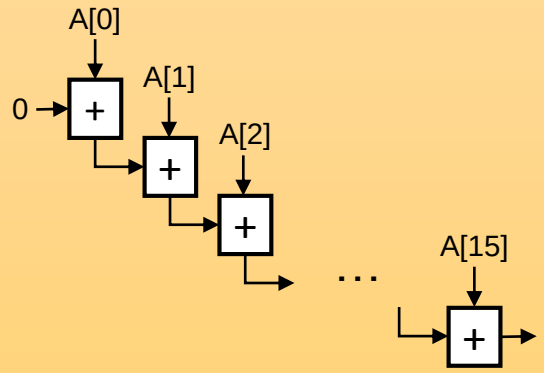
Work and Depth

- Algorithmic complexity measures
  - ignoring communication overhead
- ***Work***: total amount of work in the TDG
  - Work = T_1: time to execute TDG sequentially
- ***Depth***: time it takes to execute the critical path
  - Depth = T_∞ : time to execute TDG on an infinite number of processors
  - Also called *span*
- ***Average Parallelism***:
  - $P_{avg} = T_1 / T_\infty$
- What about time on *p* processors?
  - Depends on how we schedule the operations on the processors
  - $T_p(S)$: time to execute TDG on *P* processors using scheduler *S*
  - T_p : time to execute TDG on *P* processors with the best scheduler

Work and Depth

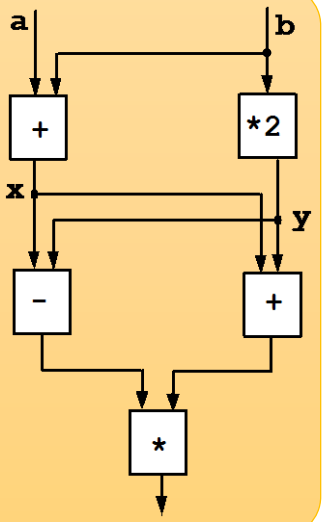
- Work = 16
- Depth = 16
- Average Par = 1

```
c = 0;
For (i=0; i<16; i++)
    c = c + A[i]
```



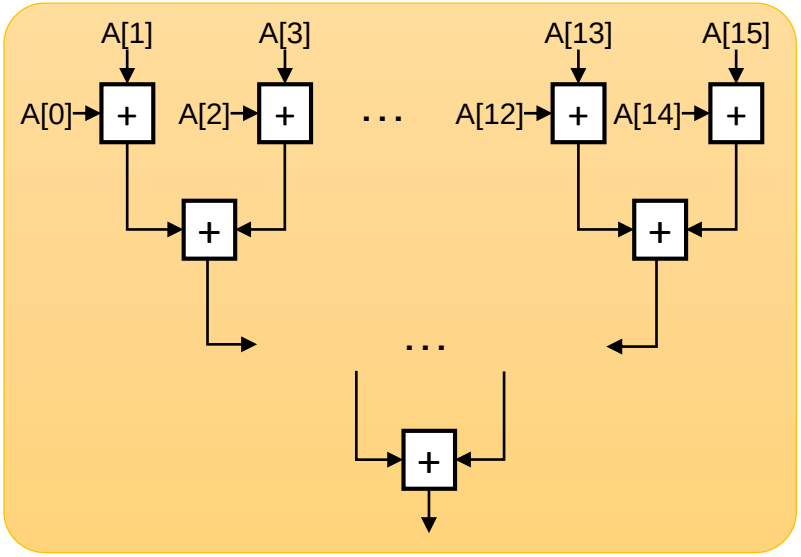
- Work = 5
- Depth = 3
- Average Par = 5/3

```
x = a + b;
y = b * 2
z =(x-y) * (x+y)
```



Inexact vs. Exact Parallelization

- Exact parallelization: parallel execution maintains all the dependences
- Inexact parallelization: parallel execution can change the dependences in a reasonable fashion
  - Reasonable fashion: depends on the problem domain
- Inexact parallelism may or may not change the final result
  - Often it does



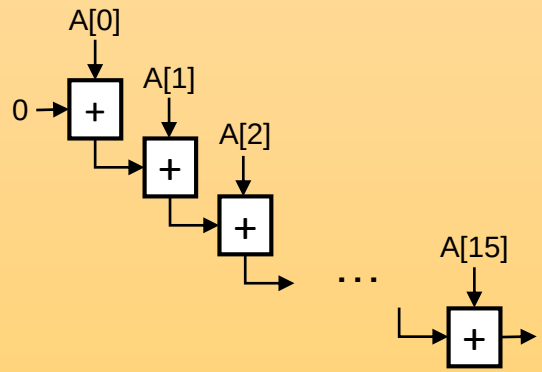
- Result the same if "+" is associative
  - Like integer "+"
  - Unlike floating-point "+"

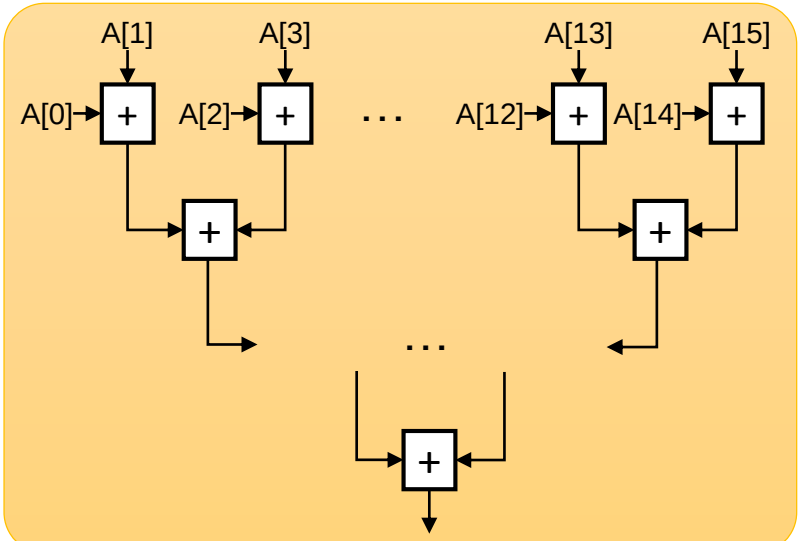
Inexact vs. Exact Parallelization

- Work = 16
- Depth = 16
- Average Par = 1

- Work = 15
- Depth = 4
- Average Par = 15/4

```
c = 0;
For (i=0; i<16; i++)
    c = c + A[i]
```





Often, efficient parallelization needs algorithmic changes

Speed Up and Efficiency

- ***Speed up***: sequential time / parallel time
  - $S_p = T_1 / T_p$
- ***Work efficiency***: a measure of how much extra work the parallel execution does
  - $E_p = S_p / p = T_1 / (p \times T_p)$

Work Law

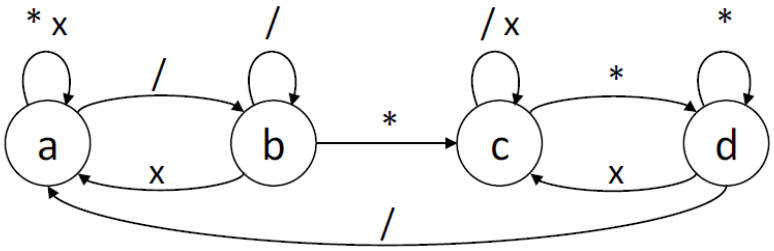
- For the same TDG, you cannot avoid work by parallelizing
- Thus, in theory
  - $T_1 / p \leq T_p$
  - Equivalently (in terms of speedup), $S_p \leq p$
- How about in practice?
  - If $S_p > p$, we say the speedup is *superlinear*
  - Is it possible?
- Yes, it is
  - Due to caching effects (locality rocks!)
  - Due to exploratory task decomposition

Depth Law

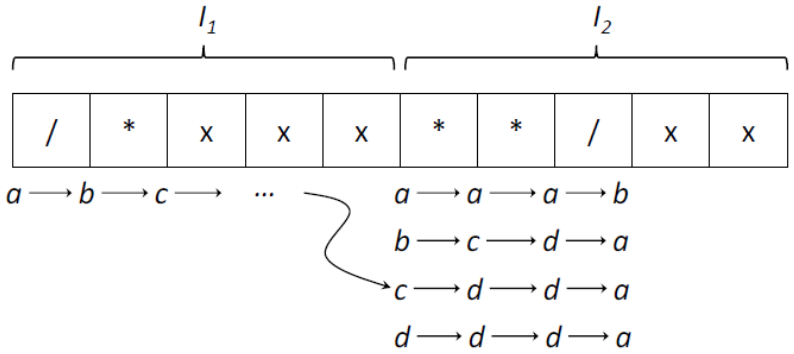
- More resources should make things faster
  - However, you are limited by the sequential bottleneck
- Thus, in theory
  - $S_p = T_1 / T_p \leq T_1 / T_\infty$
  - ***Speedup*** is bounded from above by ***average parallelism***
- What about in practice?
  - Is it possible to execute faster than the critical path?
- Yes, it is
  - Through ***speculation***
  - Might (often does) reduces work efficiency

Speculation to Decrease Depth

- Example: parallel execution of FSMs over input sequences
  - Todd Mytkowicz et al., "*Data-Parallel Finite-State Machines*", ASPLOS 2014



An 4-state FSM that accepts C-style comments, delineated by /* and */. "x" represents all characters other than / and *.



Parallel execution of the FSM over the given input.

Performance of Greedy Scheduling

- Greedy scheduling: At each time step,
  - If more than *P* nodes are ready, pick and run any subset of size *P*
  - Otherwise, run all the ready nodes
    - A node is "ready" if all its dependences are resolved
- **Theorem**: any greedy scheduler *S* achieves

$$T_p(S) \leq T_1 / p + T_\infty$$

- Proof?
- **Corollary**: Any greedy scheduler is 2-optimal, i.e.,

$$T_p(S) \leq 2T_p$$

- Food for thought: the corollary implies that scheduling is asymptotically irrelevant → Only decomposition matters!!!
  - Does it make sense? Is something amiss?

Scalability

Amdahl's Law

- Depth Law is a special case of Amdahl's law
  - Due to Gene Amdahl, a legendary Computer Architect
- If a change improves a fraction ***f*** of the workload by a factor ***K***, the total speedup is:

  Speedup = $1 / ((1 - f) + f / K)$

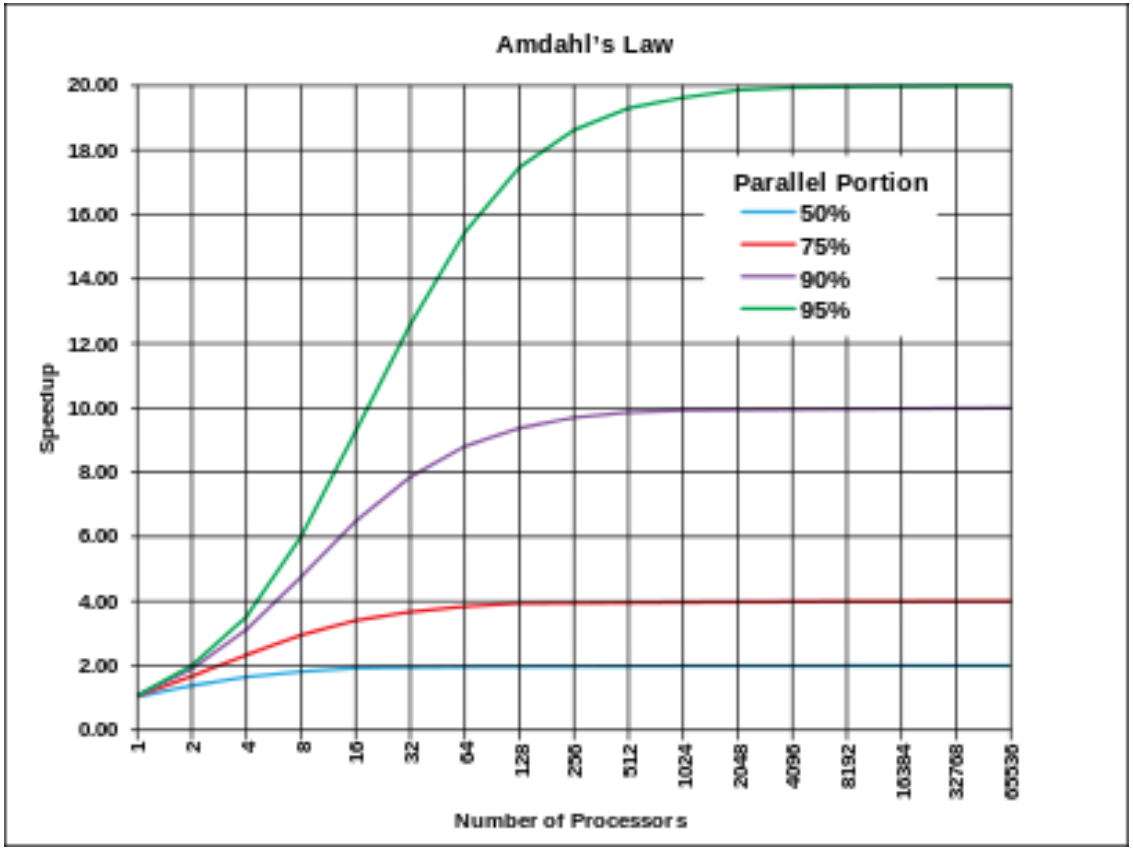
  Hence, $S_\infty = 1 / (1 - f)$
- In our case:
  - ***f*** is the fraction that can be run in parallel
  - Fraction ***1 - f*** should be run sequentially

→ Look for algorithms with large ***f***

  - Otherwise, do not bother with parallelism for performance

Amdahl's Law

- Speed up for different values of ***f***



Source: wikipedia

Lesson

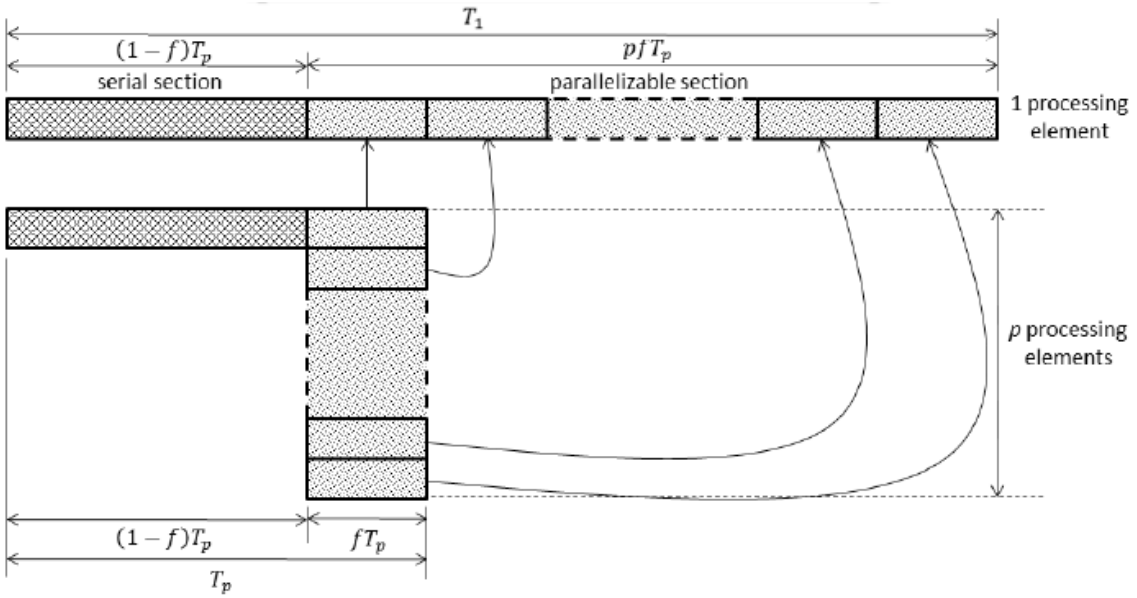
- Speedup is limited by sequential code
- Even a small percentage of sequential code can greatly limit potential speedup
  - That's why speculation is important

Counterpoint: Gustafson-Barsis' Law

- Amdahl's law keeps the problem size fixed
- What if we fix the exec. time and let the problem size grow?
  - We often use more processors to solver larger problems
- f is the fraction of execution time that's parallel
- $S_p = p\,f + (1 - f)$

→ S_p can grow unboundedly.

  - If f does not shrink too rapidly.



Any sufficiently large problem can be effectively parallelize

Scalability

- "The Program should scale up to use a large number of processors"
  - But what does that really mean?
- One formulation: How does parallel efficiency (E_P) change as P grows?

A (not so good) measure of scalability:

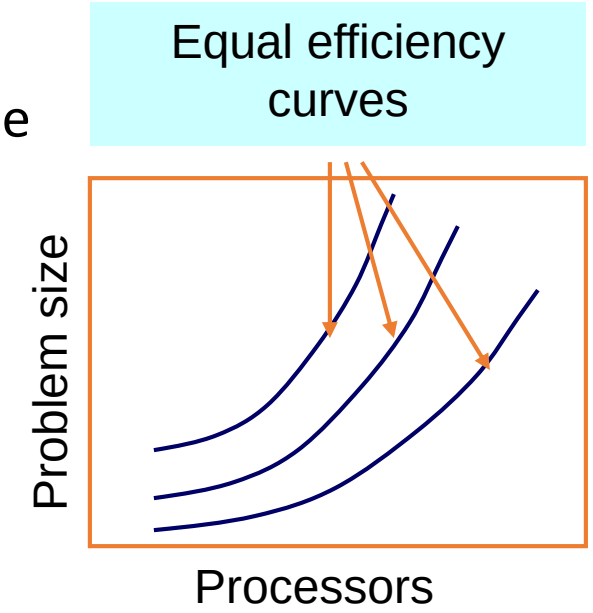
- ***Strong Scaling***: How does E_P vary with P when the problem size is fixed?
  - Not a reasonable measure
  - Any fixed-size computation is only scalable up to a certain processor count

Better measures:

- ***Weak Scaling***: How does E_P vary with P when the problem size per processor is fixed.
  - i.e., the problem size grows linearly with P
  - N/P = constant
- ***Isoefficiency***: How should N vary with P to maintain keep E_P fixed?

Scalability

- A parallel algorithm called *scalable* if E_P can be kept constant by increasing the problem size as P grows
- ***Isoefficiency***: Equation for equal-efficiency curves
- Solve $E(P, N) = E(x.P, y.N)$
  - If no solution, the algorithm is not scalable
- Food for thought:
  What does the shape of the curve signify?



What about Communication and Synchronization?

Communication and Synchronization

- Parallel Time = Computation + Communication + Idle
  - Idle: due to synchronization, load imbalance and sequential sections (a form of load imbalance IMO)
  - Synchronization typically uses communication mechanisms
    - However, it's for control purposes
- In modern machines, communication is much more expensive than computation
  - Both in terms of performance and power
- But how to quantify communication?
  - Very difficult for several reasons

Difficulties with Communication (1)

- There are different types of communication
  - Point-to-point
  - Global Synchronization
    - Barriers, scalar reductions, …)
  - Vector reductions
    - Data size is significant
  - Broadcasts
    - Small (Signals)
    - Large
  - Global (Collective) operations
    - All-to-all operations, gather, scatter

Difficulties with Communication (2)

- There different scales
  - Within a core (in-cache)
  - Within a chip (between caches)
  - Within a machine (across sockets)
  - Within a switch
  - Across switches
- Not always, statically obvious which one a given communication operations is going to be
  - Especially in shared-memory programming where communication is implicit
  - Even in message-passing programming where communication is explicit
- Made even more complex by dynamic mapping and decomposition

Difficulties with Communication (3)

- Often, communication overlaps with computation
  - In message passing:
    - can send a message and do computation while the message is being sent
    - initiate a recv, do work and then poll to see if it is done
  - In shared-memory:
    - Often memory requests are overlapped with other instructions if there is enough work to do

Quantifying Communication

- One used measure is ***computation-to-communication ratio***
  - In other words, the communication grain size
  - Operations per byte
- Ignores most of the difficulties mentioned previously
  - But still useful as it provides a first-order understanding of the communication complexity of an algorithm
- In message passing it's the total data sent and recv'd
  - Easier to calculate based on program and input size
- What about in shared-memory?
  - Once measure: total amount of data moved to the local memory (e.g., cache)
    - Often, very difficult to calculate

Performance Tuning Techniques

Computation

- Analyze the Work and Depth of your algorithm
- Parallelism is Work/Depth
- Try to decrease Depth
  - the critical path
  - a *sequential* bottleneck
- If you increase Depth
  - better increase Work by a lot more!

Synchronization and load imbalance

- Reduce sharing degree of heavily-used data structures by using distributed versions instead of centralized ones
  - Example: per-thread heaps instead of a global heap
  - Example: distributed task queues versus centralized queue
- Use lock-free and synchronization-free algorithms
  - We'll see a bunch later
- Avoid coarse-grained decomposition
- Give higher priority to more critical jobs

Communication

- Locality is your friend
  - Once communicated, use the data (of instructions) as much as possible before moving to the next piece
- Sometimes it might be okay to use “stale” data
  - Especially, for iterative algorithms that will eventually converge no matter what
  - Or problems that can tolerate approximate solutions
- Might be beneficial to recomputed instead of communicate
  - Lose computation performance to gain communication performance
- Overlap communication with computation whenever possible
  - To hid communication delay

Much Easier Said than Done!

- Yes, that's why parallel computing is still a major challenge.
- Add to all of this the challenges of
  - huge and unstructured data sets,
  - heterogeneity in hardware and software,
  - need for integration & cooperation over a vast spectrum (wearable devices to data centers),
  - lack of proper foundational models for non-scientific computing,
  - need for balancing speed, power and dollar cost,
  - Failures and reliability issues in large computer systems, …
- Lots of research still needed. Hence this course!