# Introduction & Basics of parallelism

Basic computer architecture
Basic approaches to parallelism
Speedup and Efficiency

Material in this presentation from textbook:
Georg Hager and Gerhard Wellein, Introduction to High Performance Computing for Scientists and Engineers, Chapman & Hall/CRC Computational Science Series ISBN 978-1-4398-1192-4



## Outline

- Quick intro do computer architecture
- Why parallelize?
- Parallelism
- Speedup
  - Strong scaling
  - Weak scaling
- Parallel program design



## What Is a Supercomputer?

- Many supercomputers are one large computer made up of many smaller computers and processors – a “cluster”
- With a supercomputer, all these different computers talk to each other through a communications network
  - On new Yeti – InfiniBand
- Each different computer is called a **node**
- Each node has processors/cores
  - Carry out the instructions of the computer

## Why Use a Supercomputer?

- Supercomputers give you the opportunity to solve problems that are too complex for the desktop
  - Might take hours, days, weeks, months, years
  - If you use a supercomputer, might only take minutes, hours, days, or weeks
- Useful for problems that require large amounts of memory



## Computers and Cars - Analogy

≈



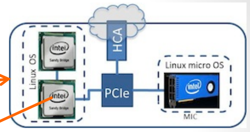
## Computers and Cars - Analogy

≈

Image from cray.com

# Computer Architecture 101

Yeti addition




Yeti addition has 60 nodes
40 GB/s QDR Infiniband interconnect

Yeti compute node :

- 2 Sockets per Node → 2 Xeon Ivy bridge processors
- 128 GB Memory
- 250 GB Disk


Socket:

- 2.2 GHz
- 10 Cores
- 8 DP FP operations per clock cycle
- 64 GB L1 Cache/core
- Vector width: 4 double precision items



# Floating Point Performance

$$P = n_{\text{core}} * F * S * \nu$$

- Example: Intel Xeon E5 on Stampede
  - Number of cores: 8 $n_{\text{core}}$
  - FP instructions per cycle: 2 (1 Multiply and 1 add) $F$
  - FP operations / instruction (SIMD): 4 (dp) / 8 (sp) $S$
  - Clock speed: 2.7 GHZ $\nu$

$$P = 173\ GF/s\ (dp) \quad \text{or} \quad 346\ GF/s\ (sp)$$

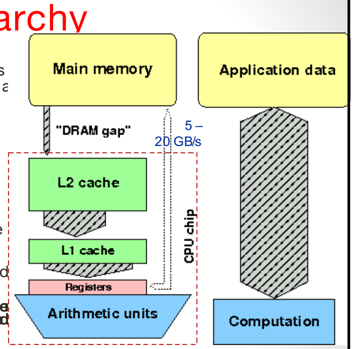
- But: P= 5.4 GF/s (dp) for serial, non-SIMD code



# Memory hierarchy

1. CPU/Arithmetic unit issues a load request to transfer a data item to a register
2. Cache logic automatically checks all cache levels if data item is already in cache
3. If data item is in cache "cache hit" it is loaded to register.
4. If data item is in no cache level ("cache miss") data item is loaded from main memory and a copy is held in cache

**If cache is already full another cache line must be invalidated or "evicted" in 4**



# Why parallelize

- Single core too slow for solving the problem in a "reasonable" time
  - "Reasonable" time: overnight, over lunch, duration of a PhD theses
- Memory requirements
  - Larger problem
  - More physics
  - More particles



# Parallelism

- For multi-core or multi-node computers
- Data parallelism
  - Single Program Multiple Data (SPMD)
  - Same code is executed on all processors
  - Data is different on the nodes
- Functional parallelism
  - Splitting problem in separate subtasks
  - Multiple Program Multiple Data (MPMD)
  - Subtask could be SPMD
  - Difficult to load balance if subtasks have different performance properties

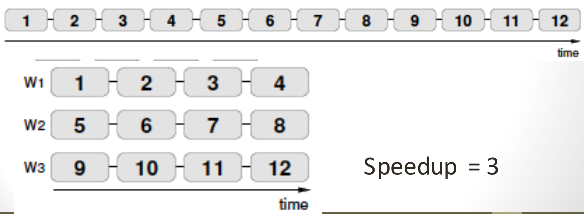


# Examples Data Parallelism

<table>
<tr><td>P1</td><td><pre>do i=1,500
  a(i)=c*b(i)
enddo</pre></td><td rowspan="2"><pre>do i=1,1000
  a(i)=c*b(i)
enddo</pre></td></tr>
<tr><td>P2</td><td><pre>do i=501,1000
  a(i)=c*b(i)
enddo</pre></td></tr>
</table>

## Speedup Formula

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$




Speedup = 3



## Execution Time Components

- Inherently sequential computations: $s(n)$
- Potentially parallel computations: $p(n)$
- Communication operations $c(n, \ p)$

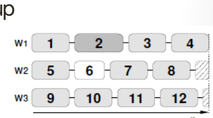
- Speedup expression:

$$S \leq \frac{s+p}{s(n)+p/N+c}$$



## Parallel Overhead

- Overhead because of
  - Startup time
  - Synchronizations
  - Communication
  - Overhead by libraries, compilers
  - Termination time
- Other barriers to perfect speedup
  - Not perfectly load balanced



# Efficiency

$$\text{Efficiency} = \frac{\text{Sequential execution time}}{\text{Processors} \times \text{Parallel execution time}}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Processors}}$$



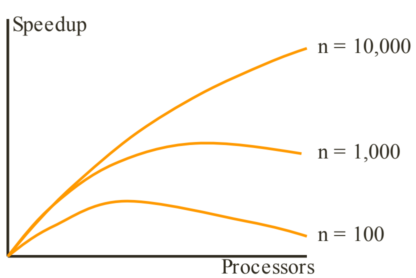
# Strong Scaling

- Keep problem size the same
- Increase the number of processors


http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980?pgno=2



# Effect of Problem Size

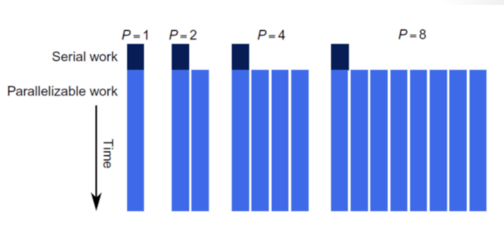


## Another Perspective

- We often use faster computers to solve larger problem instances
- Let's treat time as a constant and allow problem size to increase with number of processors
- "…speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size" – John Gustafson



## Weak Scaling




http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980?pgno=2



## Summary

- Access to main memory is most of the times your bottleneck
- Speedup
- Strong Scaling
- Weak Scaling

http://www.mcs.anl.gov/~itf/dbpp/text/book.html

# PARALLEL PROGRAM DESIGN



## Parallel Program Design

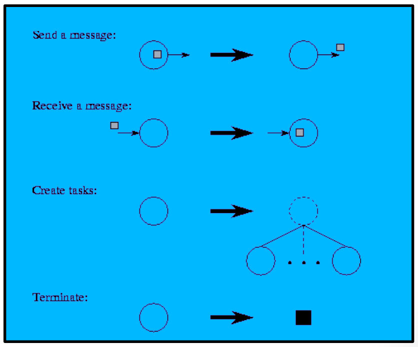
- Parallel Program Development
- Reference
  - Ian Foster
    - http://www.mcs.anl.gov/~itf/dbpp/text/book.html



## Task/Channel Model

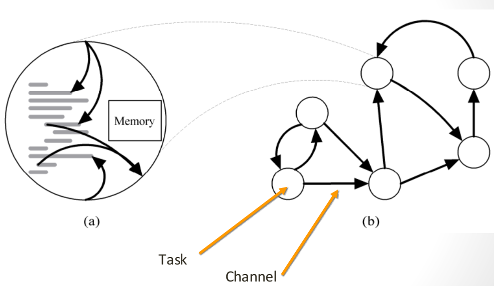
- Parallel computation = set of tasks
- Task
  - Program
  - Local memory
  - Collection of I/O ports
- Tasks interact by sending messages through channels

## Task - Channel






## Task/Channel Model






## Parallel Program

- One or more tasks executing concurrently
- Task
  - Local memory
  - Serial program
  - Input and output ports
- Asynchronous send
- Synchronous receive
- Channel – Input/Output message queues
- Mapping to physical processors does not affect semantics of program

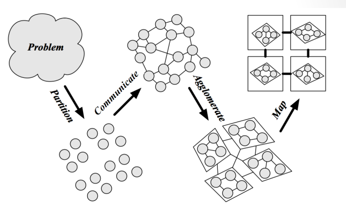
## Algorithm Design Methodology

- *Partition*
  Decompose computation and its data into small tasks.
- *Communicate*
  Determine requirements to coordinate task execution.
- *Agglomerate*
  combine tasks to improve performance or to reduce development costs.
- *Map*
  Assign tasks to processors.



## Algorithm Design Methodology

- *Partition.*
- *Communicate.*
- *Agglomerate.*
- *Map.*



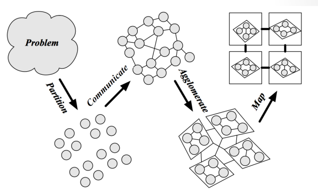



## What to aim for in partitioning

- Maximum possible concurrency (performance)
- Many more tasks than processors (flexibility)
- Number of tasks, rather than size of each task, grows as overall problem size increases (allows larger problems)
- Tasks reasonably uniform in size (load balance)
- Redundant computation or storage avoided (scalability)
- Alternative partitionings (flexibility)

## Algorithm Design Methodology

- *Partition.*
- *Communicate.*
- *Agglomerate.*
- *Map.*






## Communication Patterns

- Communication pattern determined by data dependences among tasks
- Communication pattern may be
  - local or global
  - structured or random
  - persistent or dynamically changing
  - synchronous or sporadic



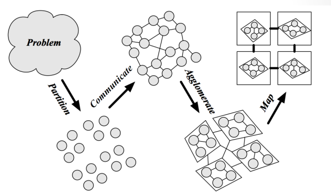
## What to aim for in communication

- Frequency and volume minimized
- Highly localized (between neighboring tasks)
- Reasonably uniform across channels and between processors
- Network resources used concurrently
- Does not inhibit concurrency of tasks
- Overlapped with computation as much as possible

All pertain to efficiency and scalability

## Algorithm Design Methodology

- *Partition.*
- *Communicate.*
- *Agglomerate.*
- *Map.*






## What to aim for in agglomeration

- Increased locality reduces communication costs
- Benefits that outweigh costs of replicating data or computation for range of problem sizes and processor counts
- Tasks that have similar computation and communication costs (load balance)
- Number of tasks scales with problem size (larger problems ok)
- When translating serial program, consider alternative agglomeration strategies that increase opportunities for code reuse (development cost)

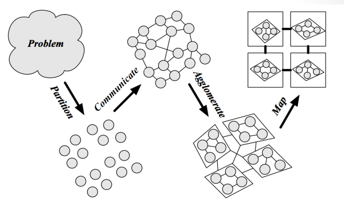


## The moral of the agglomeration story

Programs with fewer larger-grained tasks are often simpler and more efficient than those that create many fine-grained tasks

## Algorithm Design Methodology

- *Partition.*
- *Communicate.*
- *Agglomerate.*
- *Map.*



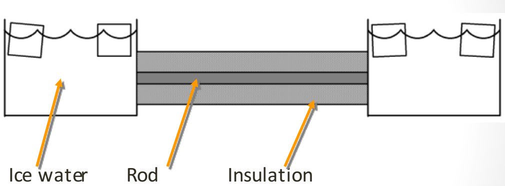



## Case Study

- Boundary value problem

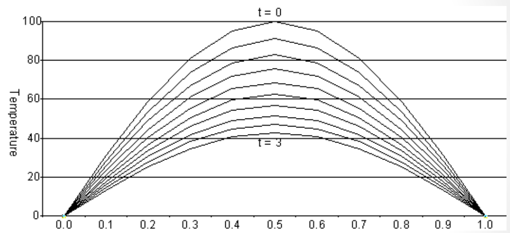


## Boundary Value Problem




$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

## Rod Cools as Time Progresses






## Discretize Heat Equation

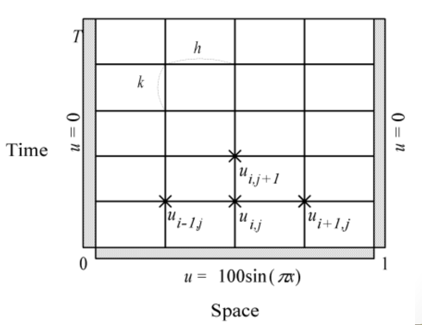
- 1st order $$\left.\frac{\partial u}{\partial t}\right)_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{\Delta t} + O(\Delta t)$$
- 2nd order $$\left.\frac{\partial^2 u}{\partial x^2}\right)_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + O(\Delta x^2)$$

$$u_{i,j+1} = u_{i,j} + \frac{\alpha \Delta t}{(\Delta x)^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right) + O(\Delta t, \Delta x^2)$$

- Stability: $$0 \leq \; r \; \leq \frac{1}{2}$$ $$r \; = \; \frac{\alpha \Delta t}{(\Delta x)^2}$$



## Finite Difference Approximation



## Partitioning

- One data item per grid point
- Associate one primitive task with each grid point
- Two-dimensional domain decomposition



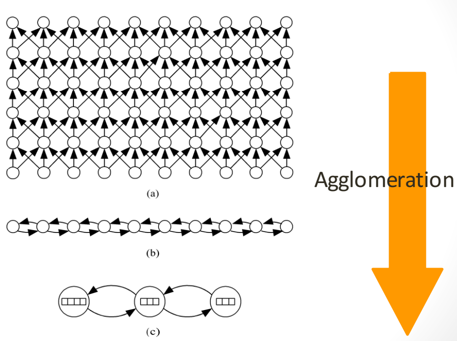
## Communication

- Identify communication pattern between primitive tasks
- Each interior primitive task has three incoming and three outgoing channels

$$u_{i,j+1} = u_{i,j} + \frac{\alpha \Delta t}{(\Delta x)^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right) + O(\Delta t, \Delta x^2)$$



## Agglomeration and Mapping



## Summary: Task/channel Model

- Parallel computation
  - Set of tasks
  - Interactions through channels
- Good designs
  - Maximize local computations
  - Minimize communications
  - Scale up



## Summary: Design Steps

- Partition computation
- Agglomerate tasks
- Map tasks to processors
- Goals
  - Maximize processor utilization
  - Minimize inter-processor communication