# UVA HPC & BIG DATA COURSE

## INTRODUCTORY LECTURES

Adam Belloum

# Computing facilities SURF/UvA/VU

SURFspot SURFdashboard My SURFmarket SURFdrive SURFfilesender edu.nl Nederlands

SURF

ICT facilities Education and ICT Research & ICT About SURF

Driving innovation together

## Research & ICT

Do you encounter limitations with your own systems? SURF offers researchers a wide range of services in the field of high performance computing (HPC): tens of thousands of times faster than your PC.

Team SURFsara

Questions? Get in touch

Choose your compute service | Consultancy & training | Knowledge & inspiration | Innovation | More

### Choose your compute service

**Which compute service for which research question?**

SURF offers several services in the area of High Performance Computing (HPC). Which one do you need for your research?

Learn more

**Request access**

Apply directly for cloud and cluster computing on HPC Cloud, Lisa, Cartesius Supercomputer and Grid.

Apply >

### Lisa Compute Cluster: extra processing power for research

Lisa Compute Cluster combines processing power with user friendliness. Are the limits of your own system inhibiting your research? This service lets you upscale to a higher level. Lisa Compute Cluster is preconfigured with a range of software packages, meaning you can start working right away.

Request access

### Dutch National Supercomputer Cartesius

The Dutch National Supercomputer, Cartesius, is the flagship of SURF's research services. It is the most comprehensive system in the field of capability computing in the Netherlands. Cartesius is especially in high demand for its combination of fast processors and internal network, large storage capacity and the ability to process large datasets.

Request access

### HPC Cloud: your flexible compute infrastructure

HPC Cloud gives you and your project team complete control over your computing infrastructure. The infrastructure ranges from a single work station to a complete cluster and can be expanded to suit your needs. You can use your own operating system and analysis software. HPC Cloud is housed in SURF's own data centre.

Request access to HPC Cloud

### Visualisation: more insight into your data

Our Visualisation service allows you to visualise your own datasets on your desktop. This makes it easy to identify connections between data or gain other insight into your datasets. SURF offers a powerful remote visualisation service that combines high performance with ease of use.

DAS 5

ASCI Homepage

SURF NET NWO Universiteit van Amsterdam CONNECT/ Science Universiteit Leiden ASTRON TU Delft

Distributed A

Home About DAS-5 Research Users Sponsors

## DAS-5 Overview

DAS-5 (The Distributed ASCI Supercomputer 5) is a six-cluster wide-area distributed system designed by the Advanced School for Computing and Imaging (ASCI). DAS-5 is funded by NWO/NCF (the Netherlands Organization for Scientific Research), and the participating universities and organizations (see below). As one of its distinguishing features, DAS-5 employs a number of HPC Accelerators (e.g., currently various GPU types, FPGA's are also planned) and a internal wide-area OpenFlow interconnect based on light paths.

The goal of DAS-5 is to provide a common computational infrastructure for researchers within ASCI, who work on various aspects of parallel, distributed, grid and cloud computing, and large-scale multimedia content analysis. The following institutes and organisations are directly involved in the realization and running of DAS-5:

- VU University, Amsterdam (VU)
- Leiden University (LU)
- University of Amsterdam (UvA)
- Delft University of Technology (TUD)
- The MultimediaN Consortium (UvA-MN)
- Netherlands Institute for Radio Astronomy (ASTRON)
- Netherlands e-Science Center (NLeSC)

Announcements

**9 October 2018**

The next generation DAS, DAS-6, will get funding! For details, see the DAS Achievements page. DAS-6 is expected to become operational in the second half of 2020.

**2 Jan 2017**

DAS-5/VU has been extended with 4 TitanX-Pascal GPUs.

**May, 2016**

IEEE Computer publishes paper about 20 years of Distributed ASCI Supercomputer. See the DAS Achievements page.

**28 Sep 2015**

DAS-5/VU has been extended with 16 GTX TitanX GPUs.

**DAS-5 cluster at VU University, Amsterdam**

**DAS-5 clusters at UvA/UvA-MN, Amsterdam**

Landscape

… From ~ 1986

to ~ 2023…

# From mono-core to exa-scale computer

Aggregation

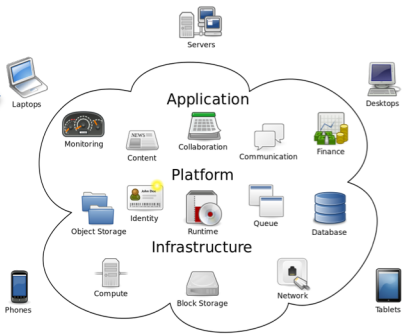
Unreliable not very fast network


Abstraction

No central control

central control

Virtualisation

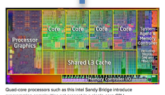



Cloud computing

Reliable fast network

central control

central control



Bigger systems

# Introduction to Parallel programming distributed systems

- Parallel programming  MPI/openMP/RMI …
- Service Oriented Architecture and Web Service
- Grid computing / Virtualisation
- Cloud Computing
- Workflow
- Discussions

# BigData

- General introduction to BigData
- MapReduce and Beyond
- Analytics of BigData
- Technology for Big Data

# If you know these concepts you are attending the wrong class …

- Supercomputing / High Performance Computing (HPC)
- Node
- CPU / Socket / Processor / Core
- Task
- Pipelining
- Shared Memory
- Symmetric Multi-Processor (SMP)
- Distributed Memory
- Communications
- Synchronization
- Granularity
- Observed Speedup
- Parallel Overhead
- Massively Parallel
- Embarrassingly Parallel
- Scalability

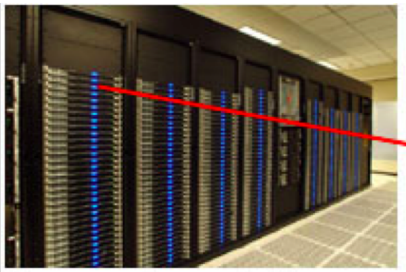
# Content

- Computer Architectures
- High Performance Computing (HPC)
- Speed up
- Parallel programming models

# Computer Architecture

- supercomputers use many CPUs to do the work
- All supercomputing architectures have
  - processors and some combination cache
  - some form of memory and input/ouput IO
  - the processors are separated from every other processors by some distance
- there are major differences in the way these parts are connected

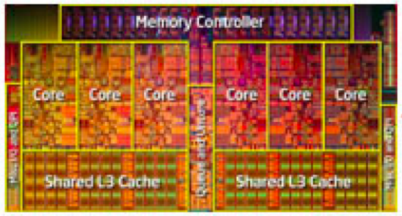
some scientific problems fit better some architectures better than others

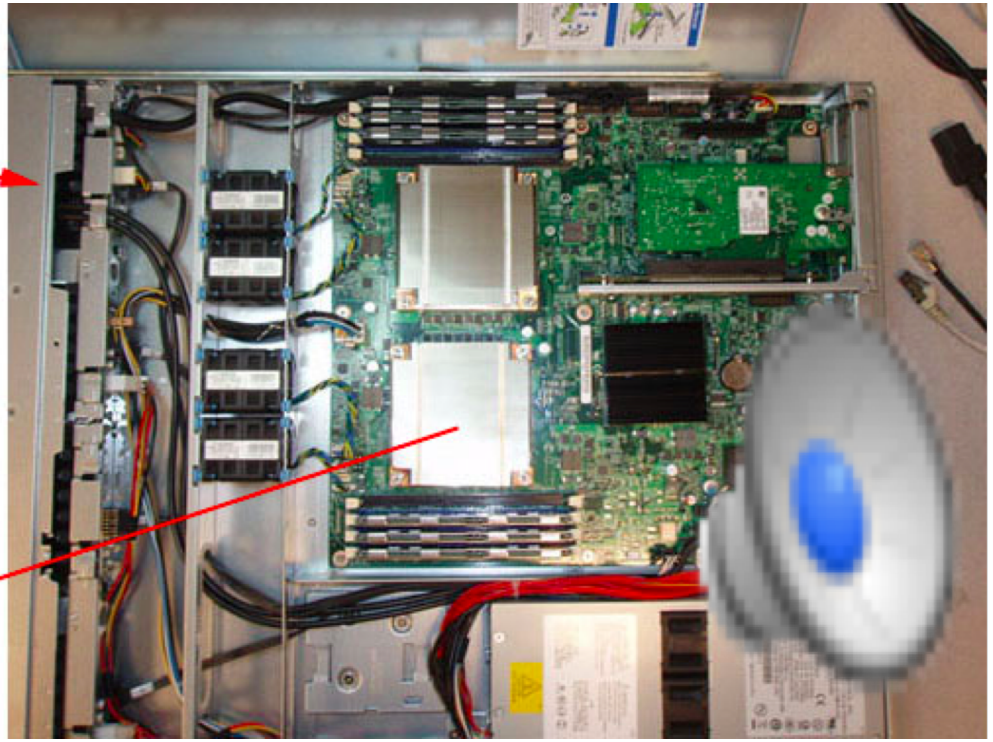


Supercomputer - each blue light is a node

Node - standalone Von Neumann computer

CPU / Processor / Socket - each has multiple cores / processors.



- How CPU works http://www.youtube.com/watch?v=cNN_tTXABUA
- How Computers Add Numbers In One Lesson: http://www.youtube.com/watch?v=VBDoT8o4q00&feature=fvwp
- Computer Architecture Lesson 1: Bits and Byteshttp://www.youtube.com/watch?v=UmSelKbP4sc
- Computer Architecture Lesson 2: Memory addresses http://www.youtube.com/watch?v=yF_txERujps&NR=1&feature=episodic
- Richard Feynman Computer Heuristics Lecture http://www.youtube.com/watch?v=EKWGGDXe5MA

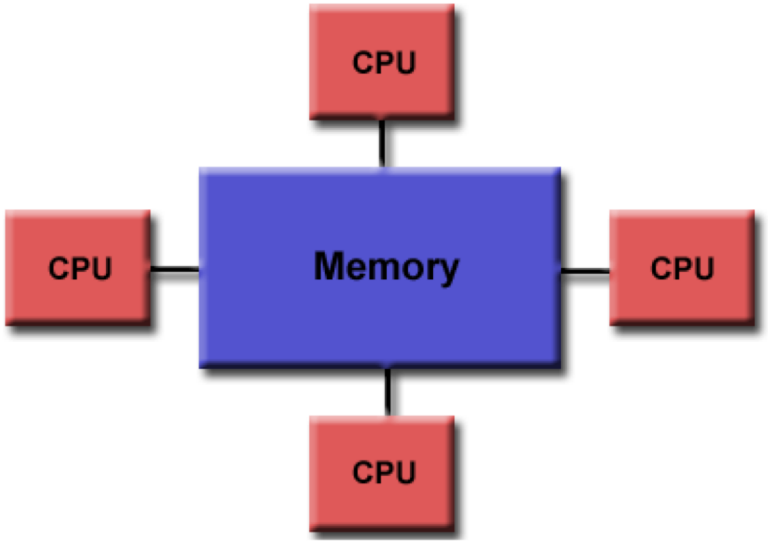
# Parallel Computer Memory Architectures

- we can also **classify** supercomputers according to how the processors and memory are connected
  - couple of processors to a single large memory address space
  - couple of computers, each with its own memory address space

# Parallel Computer Memory Architectures

## Shared Memory

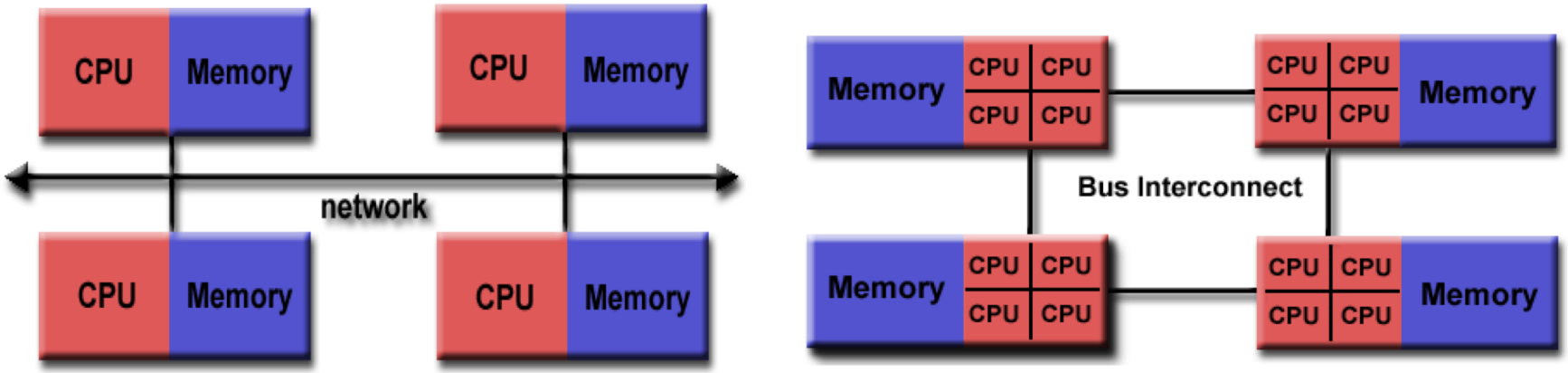
- Uniform Memory Access (UMA)
- Non-Uniform Memory Access (NUMA)



# Parallel Computer Memory Architectures

## Distributed Memory Multiprocessor

- Processors have their own local memory
- Changes it makes to its local memory have no effect on the memory of other processors.



# Why Use supercomputers?

- To solve larger problems
- To use of non-local resources
- To save time and/or money
- Because they enable concurrency

**DreamWorks Presents the Power of Supercomputing**
http://www.youtube.com/watch?v=TGSRvV9u32M&feature=fvwp

https://computing.llnl.gov/tutorials/parallel_comp/

# High Performance Computing

- What does *High-Performance Computing (HPC)* mean?
  - High-performance computing (HPC) is **the use** of super computers and parallel processing techniques for solving complex computational problems.
  - *HPC technology* focuses on developing parallel processing systems by incorporating both **administration** and **parallel computational** techniques.

The terms high-performance computing and supercomputing are sometimes used interchangeably.

http://www.techopedia.com/definition/4595/high-performance-computing-hpc

# Content

- High Performance Computing
- Computer Architectures
- Speed up
- Parallel programming models
- Example of Parallel programs

# Speedup

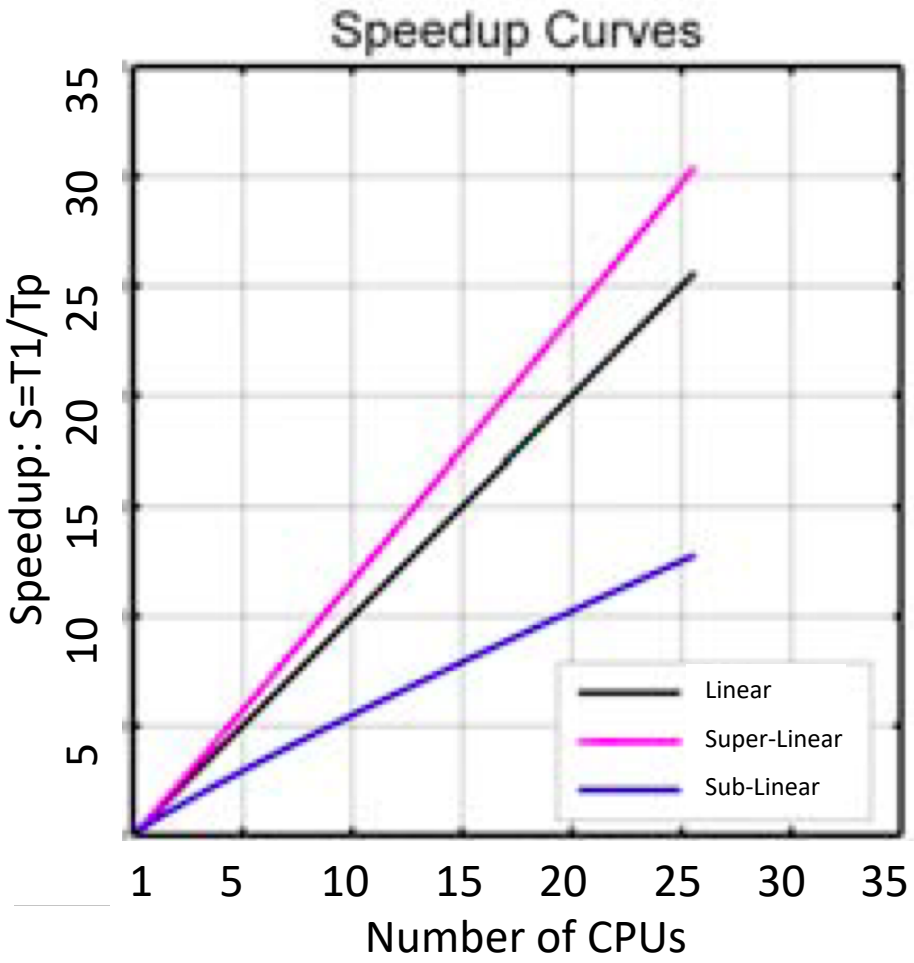
- How can we measure how much faster our program runs when using more than one processor?
- Define Speedup S as:
  - the ratio of 2 program execution times
  - constant problem size
    - $T_1$ is the execution time for the problem on a single processor (use the "best" serial time)
    - $T_P$ is the execution time for the problem on P processors

$$S = \frac{T_1}{T_P}$$

# Speedup

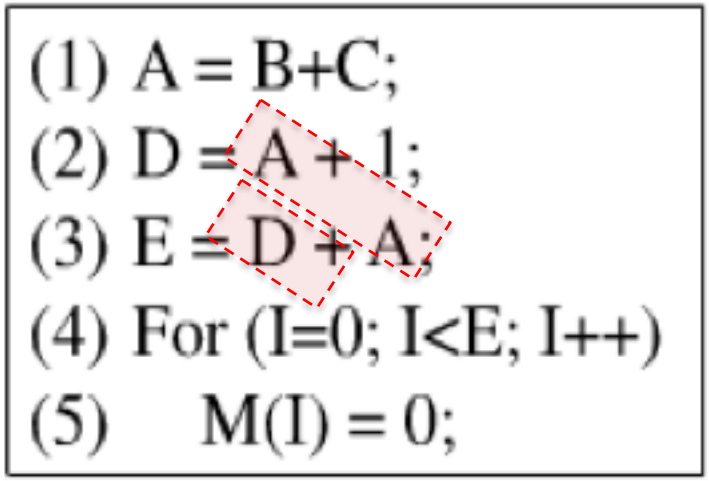
- Linear speedup
- Sublinear speedup
- Superlinear speedup

- why do a speedup test?



# Speedup: Limit of Parallel programming

- A program always has a sequential part and a parallel part




- the best you can do is to sequentially execute 4 instructions no mater how many processors you get

http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming

# Speedup: Implication

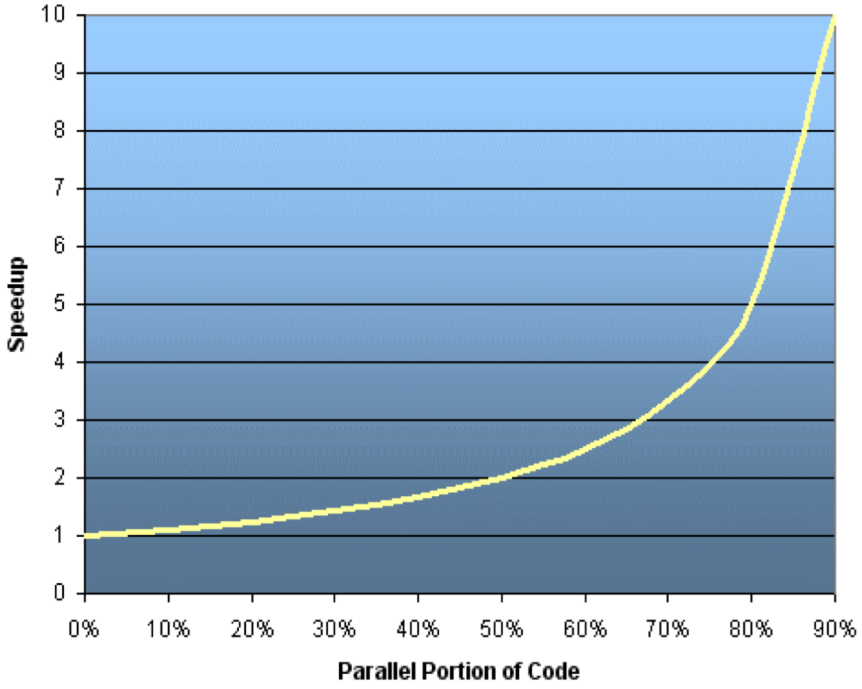
- Parallel programming is great for programs with a lot of parallelism
  - Jacobi, scientific applications (weather prediction, DNA sequencing, etc)
- Parallel programming may not be that great some traditional applications:
    - Computing Fibonacci series F(K+2)=F(k+1) + F(k)

http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming

# Speedup: Amdahl's Law (1967)

- Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

$$\text{speedup} = \frac{1}{1 - P}$$



# Speedup: Amdahl's Law (1967)

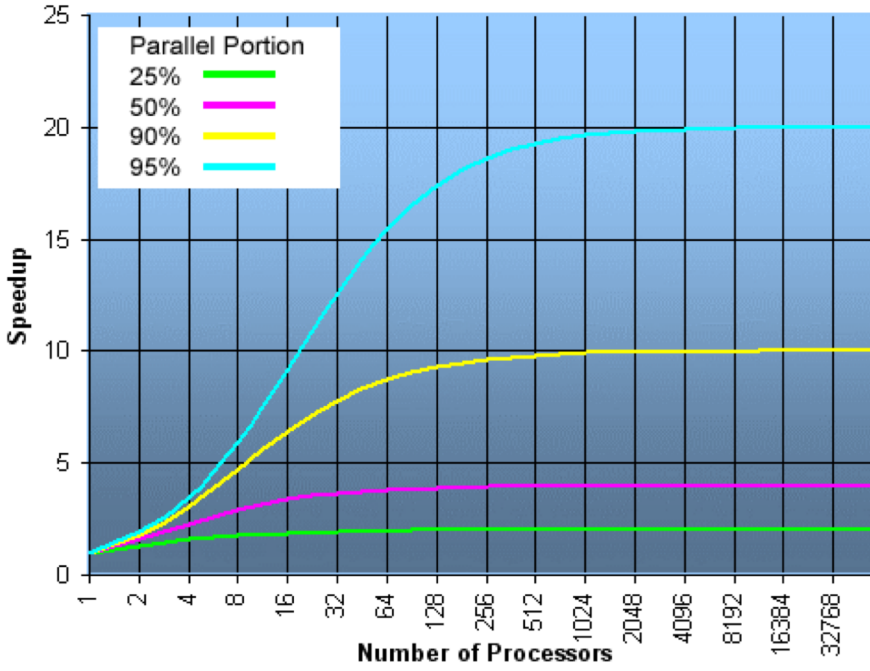
- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$Speedup = \frac{1}{\frac{P}{N} + s}$$

P = parallel fraction of the code

S = Section fraction of the code

N= Number of Processors



# Gustafson's law

- Amdahl's law assumes a constant problem size
- Now, let's allow scaling of the problem size
- Gustafson's law considers the execution time of a parallel program using
  - *n* processors
  - with a sequential fraction s (with s in [0,1])

$$\text{Speedup} = \frac{s + n(1-s)}{s + (1-s)}$$

*Translates into:* Speedup = s+n(1-s) = n–s(n-1)

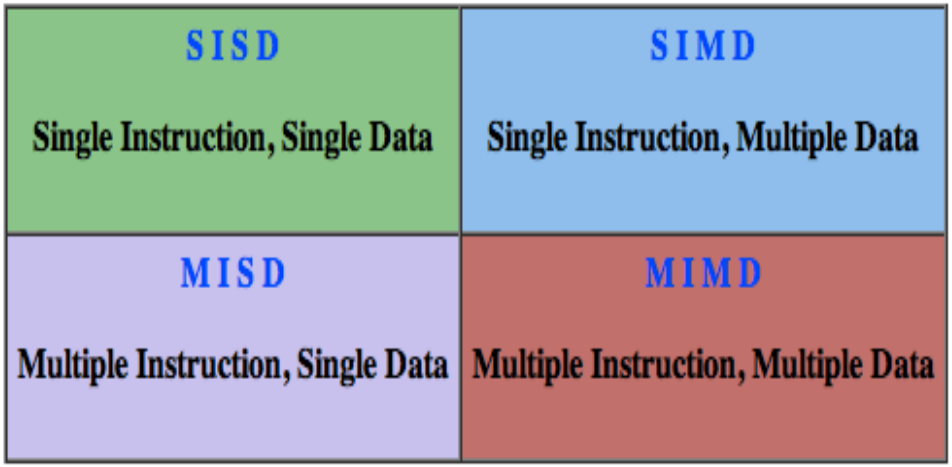
# Content

- High Performance Computing
- Computer Architectures
- Speed up
- How to design Parallel programs
- Parallel programming models
- Example of Parallel programs

# Architectures: Michael J. Flynn (1972)

- Flynn's taxonomy distinguish multi-processor computer according to independent dimensions
  - Instruction
  - Data

- Each dimension
  - Single
  - Multiple



# Design Parallel programs

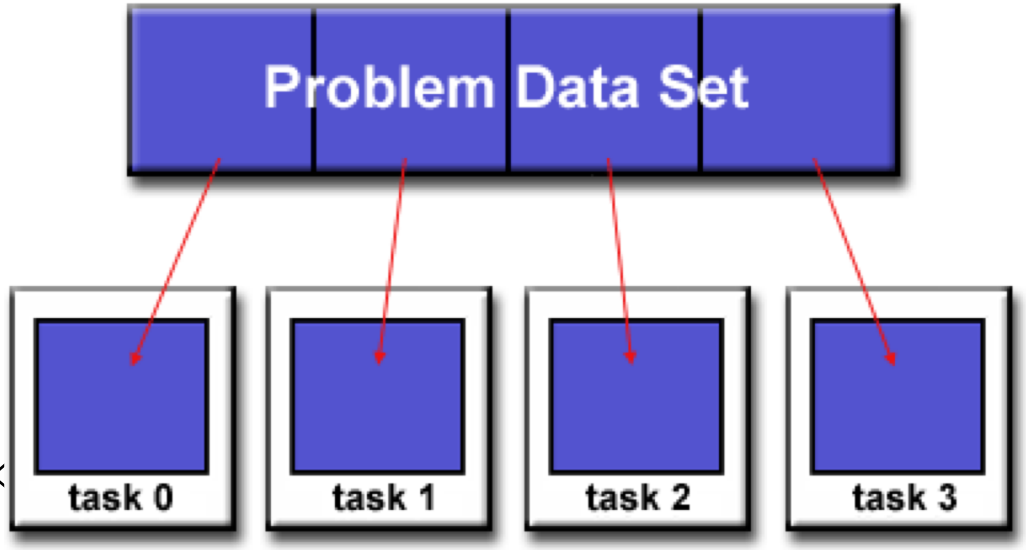
- Domain decomposition and functional decomposition
  - **Domain decomposition**: DATA associate with a problem is decomposed.
    - Each parallel **task** then **works** on a portion of data
  - **Functional deposition**: focus on the computation that is be performed. The problem is decomposed according to the work that must be done.
    - Each **task** then **performs** a portion of the overall work

http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming

# Data parallelism

Domain decomposition:

- Also Called data parallelism
- DATA associate with a problem is decomposed.
- Each parallel task then work on a portion of data
- Example: MapReduce

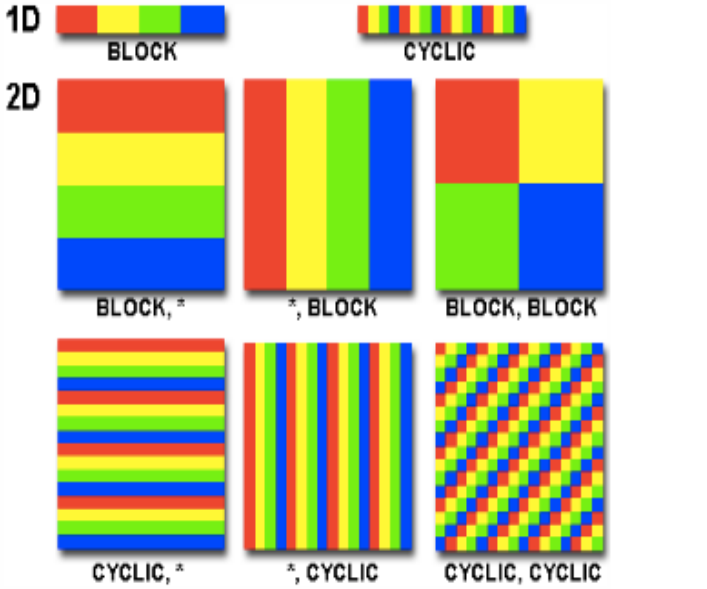



http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming

# Data parallelism

Domain decomposition methods:

- Same datum may be needed by **multiple tasks**
- Decompose the data in such a manner that the required communication is minimized
- Ensure that the computational loads on processes are balanced




http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming

# Data parallelism

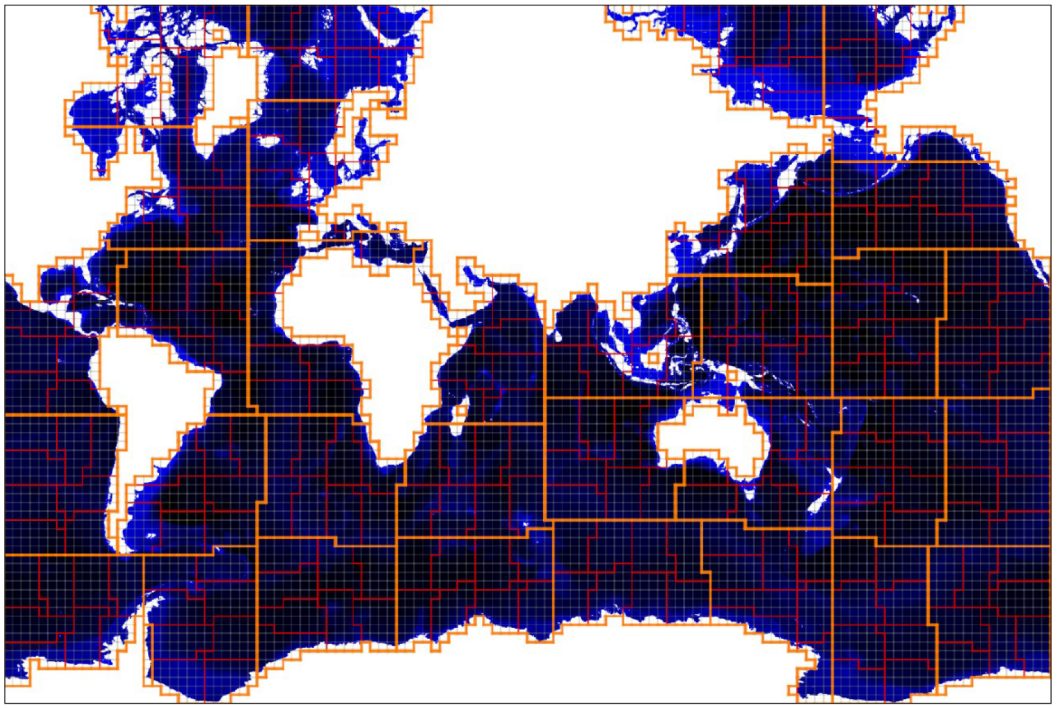
## Hierarchical work distribution

To solve this problem we designed a hierarchical work distribution algorithm that recursively splits the world into blocks (top-down).

Example:

Split each result into 32 (nodes)
Split each result into 16 (cores)

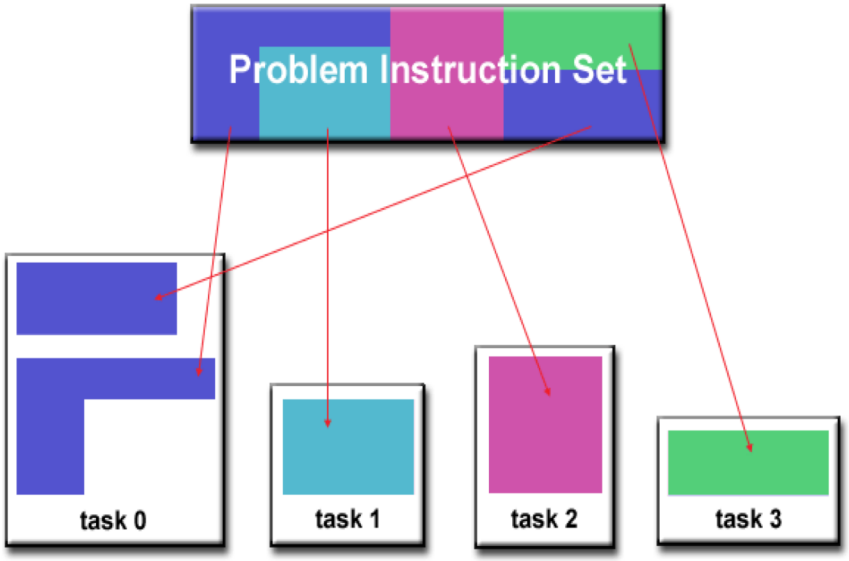
Maximize load balance and minimize communication at each level!



Source: A Distributed Approach to Improve the Performance of the Parallel Ocean Program
Ben van Werkhoven et al., Geoscientific Model Development, 7, 257-281, 2014

# Functional deposition

- the focus is on the **computation** that is to be performed rather than on the data manipulated by the computation.
- The problem is decomposed according to the **work that must be** done.
- Each task then **performs** a portion of the overall work.

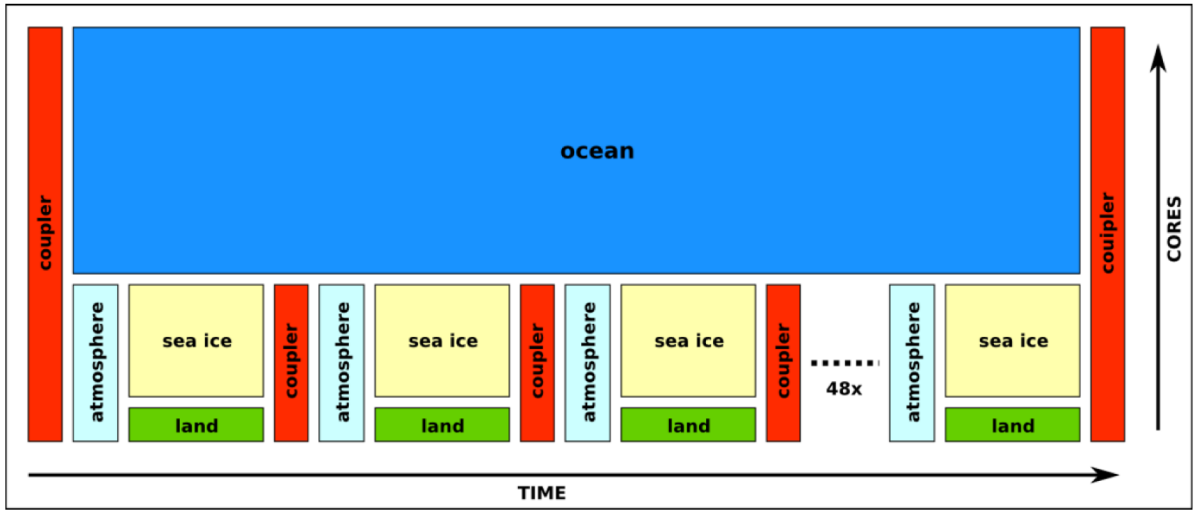



http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming

# CESM model coupling

Data dependencies allow some models to run concurrently, while others must run in sequence.

(balancing their run times is black magic)




Atmosphere, land and sea ice exchange data every 30 model minutes, but ocean only 1x to 4x each model day!

# Data Dependence

- A dependence exists between programs when the order of statement execution affects the results of the program.

- A data dependence results from multiple use of the same location(s) in storage by different tasks.

  (task 1) – (task2)

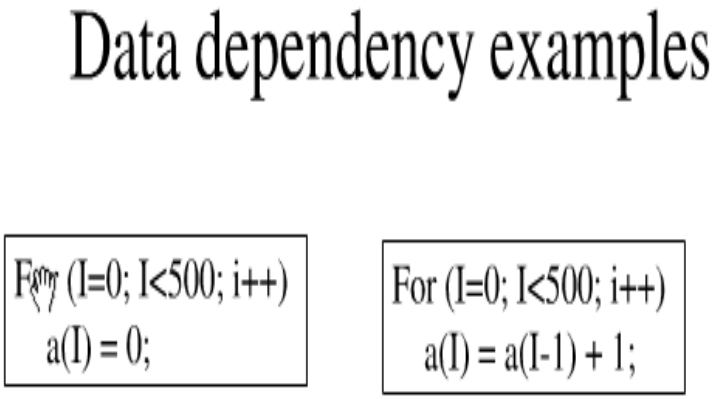
  - True dependence: Write X – Read X
  - Output dependence: Write X – Write X
  - Anti dependence: Read X – Write X

Dependencies: are important to parallel programming because the are one of the inhibitors to parallelism.

http://www.docstoc.com/docs/2698537/Introduction-to-Parallel-Programming

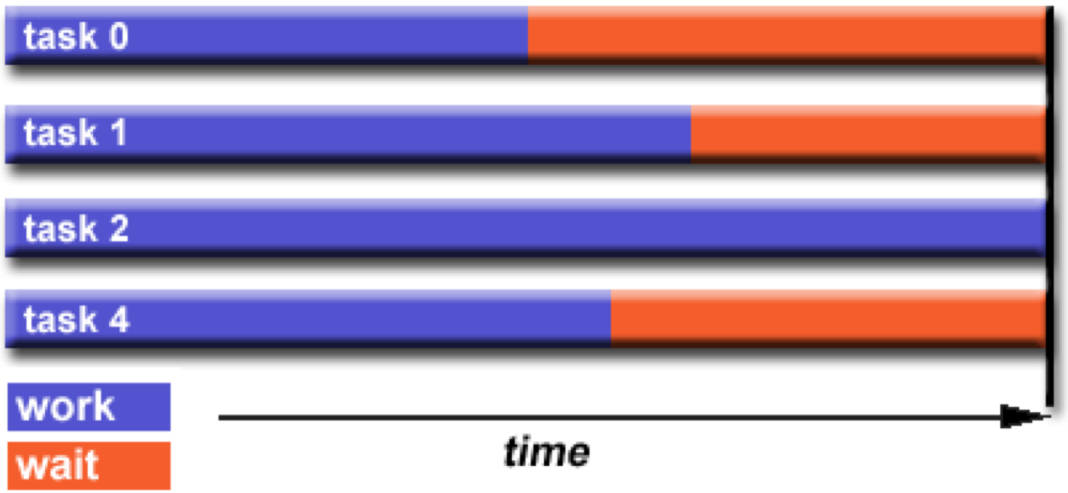
# Data Dependence

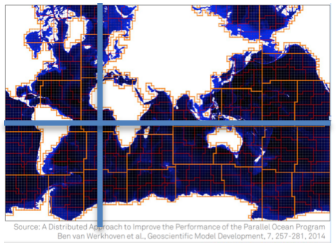
- The value of a(I-1) must be computed before the value of a(I)
- A(I) exhibits **a data dependency** on a(I-1).
- Parallelism is inhibited.



# Load balancing

- Distribute the **computation/communication** such that all the processor are busy all the time.
- At a synchronization point, the worst case performance is the real performance



# Communications

- Parallel applications that do not need communications are called embarrassingly parallel programs
  - Monte carlo method, Seti at home
  - Most programs (e.g. Jacobi) are not like that
  - Communication is inherent to exploit parallelism in a program

# Communications

- Factors to consider:
  - Cost of the communication
  - Latency and bandwidth
  - Synchronous and asynchronous
  - Point to point or collective

# Overlapping communication and computation

- Make processors busy when waiting for communication results
  - Usually achieved by using non-blocking communicating primitives

Loading balancing, minimizing communication and overlapping communication with computation are keys to develop efficient parallel applications

# Some basic load balancing techniques

- Equally partition the work each task receives
  - For array/matrix operations where each task performs similar work, evenly **distribute the data** set among the tasks.
  - For loop iterations where the work done in each iteration is similar, evenly **distribute the iterations** across the tasks.
- Use dynamic work assignment
  - Sparse arrays
  - Adaptive grid method
  - If a **heterogeneous mix of machines** with **varying performance**
    - → *scheduler* - *task pool* approach

# Granularity

- Computation/ Communication
  - In parallel programming, granularity is a qualitative measure of the ratio of the **computation** to **communication**.
  - **Periods of computation** are typically separated form **periods of communication** by synchronization events
    - Computation phase and communication phase

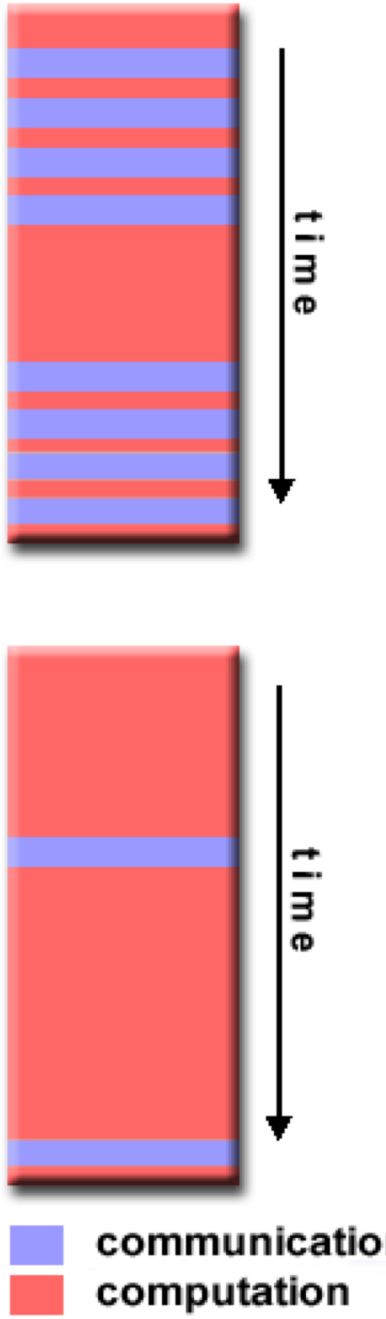
# Granularity

- Fine-grain parallelism
  - Relatively small **amount of computational** work are done between communication events

Low computation to communication ratio Implies high commutation over head and less opportunity for performance enhancement

- Coarse-grain parallelism
  - Relatively large **amounts of computation** work are done between communication/synchronization events

High computation to communication ratio Implies more opportunity for performance increase Harder to load balance efficiently



# Deadlock/Livelock

- Deadlock appears  when two or more programs are waiting and none can make progress

- Livelock results from indefinite loop.

# content

- High Performance Computing
- Computer Architectures
- Speed up
- How to design parallel applications
- Parallel programming models
- Example of Parallel programs

# Parallel Programming models

Data Parallelism/task parallelism

- Shared Memory (without threads/Threads)
- Distributed Memory / Message Passing
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

https://computing.llnl.gov/tutorials/parallel_comp/#ModelsOverview

# Parallel programming

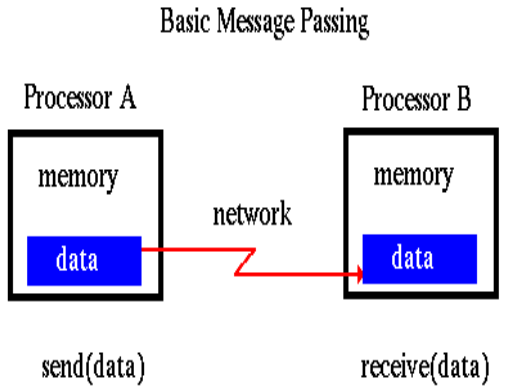
- need to do something to your program to use multiple processors
- need to incorporate commands into your program which allow **multiple threads** to run
  - one thread per processor
  - each thread gets a piece of the work

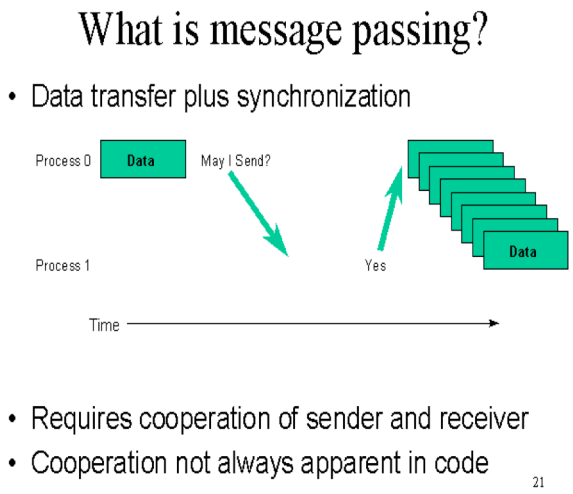
- several ways (APIs) to do this …

# Parallel programming

**Message Passing Interface (MPI)**

- **Interprocess** communication which have separate address spaces
- Data is **explicitly** sent by one process and received by another
  - Data transfer usually requires cooperative operations to be performed by each process.
  - For example, a send operation must have a matching receive operation



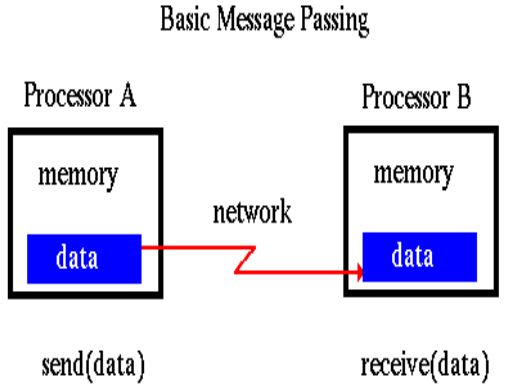



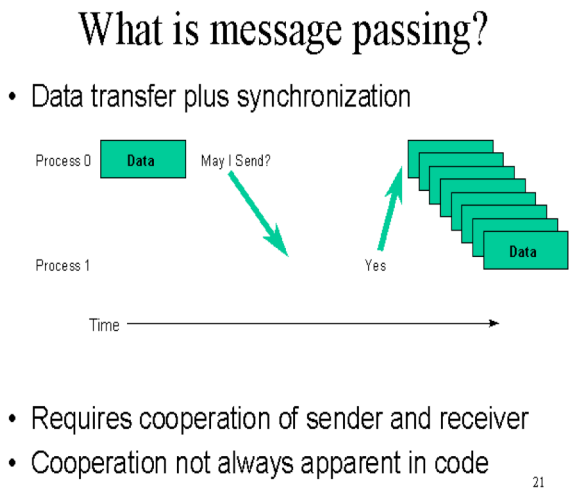
# Parallel programming

## Message Passing Interface (MPI)

- What is MPI?
  - A message-Passing Library specification
  - Not a language or compiler specification
  - Not a specific implementation or product

- For parallel computers, clusters, and heterogeneous networks.
  - Designed to provide access to advanced parallel hardware for:
    - End users, library writers, tools developers

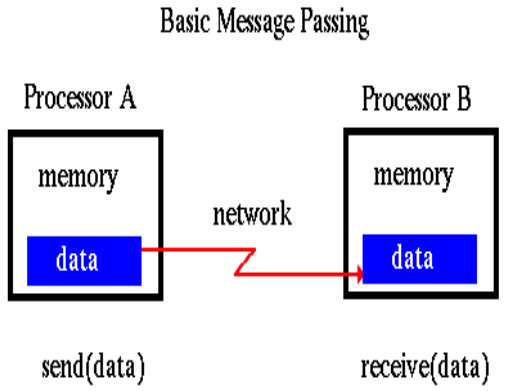


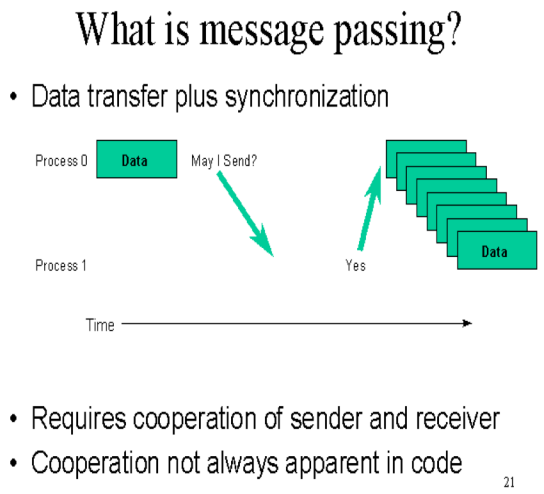



# Parallel programming

## Message Passing Interface (MPI)

- Why use MPI?
  - Optimized for performance
  - Will take advantage of fastest transport found
    - Shared memory (within a computer)
    - Fast cluster interconnects (Infiniband, Myrinet..) between computers (nodes)
    - TCP/IP if all else fails



# Parallel programming

**Message Passing Interface (MPI)**

Deadlocks?

- Send a large message from proc A to proc B
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

- What will happen? (unsafe)
  - Process 0

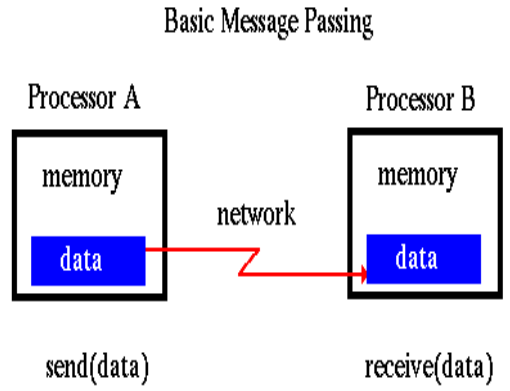
    Send(1)

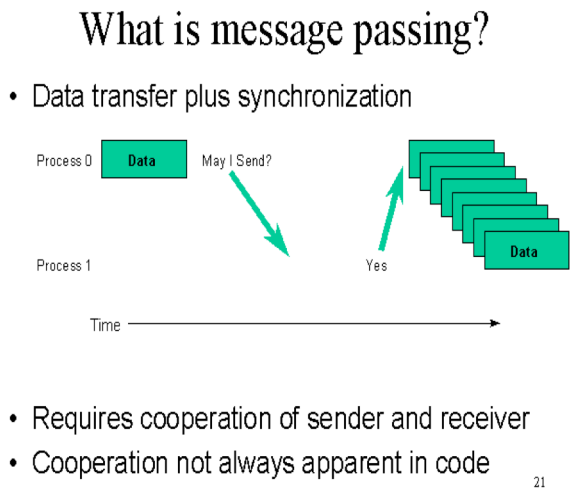
    Recv(1)

    Process 1

    Send(0)

    Recv(0)

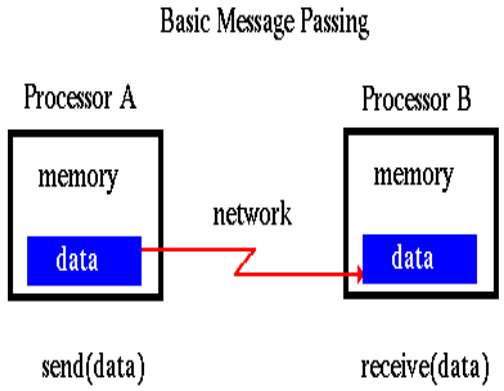


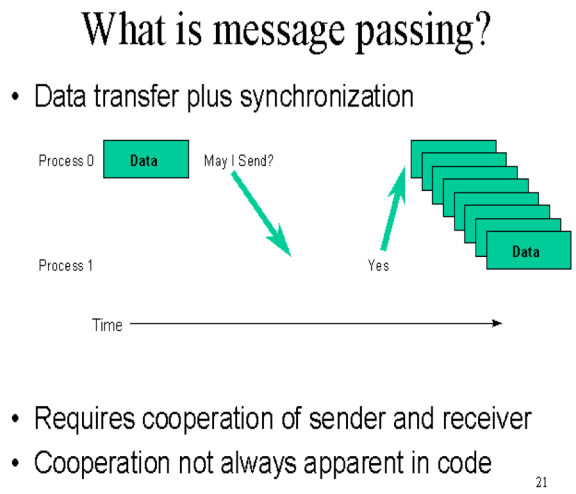



# Parallel programming

## Message Passing Interface (MPI)

- Very good for distributing large computations across reliable network
- Would be terrible for a distributed internet chat client or BitTorrent server



# Example MPI Hello World

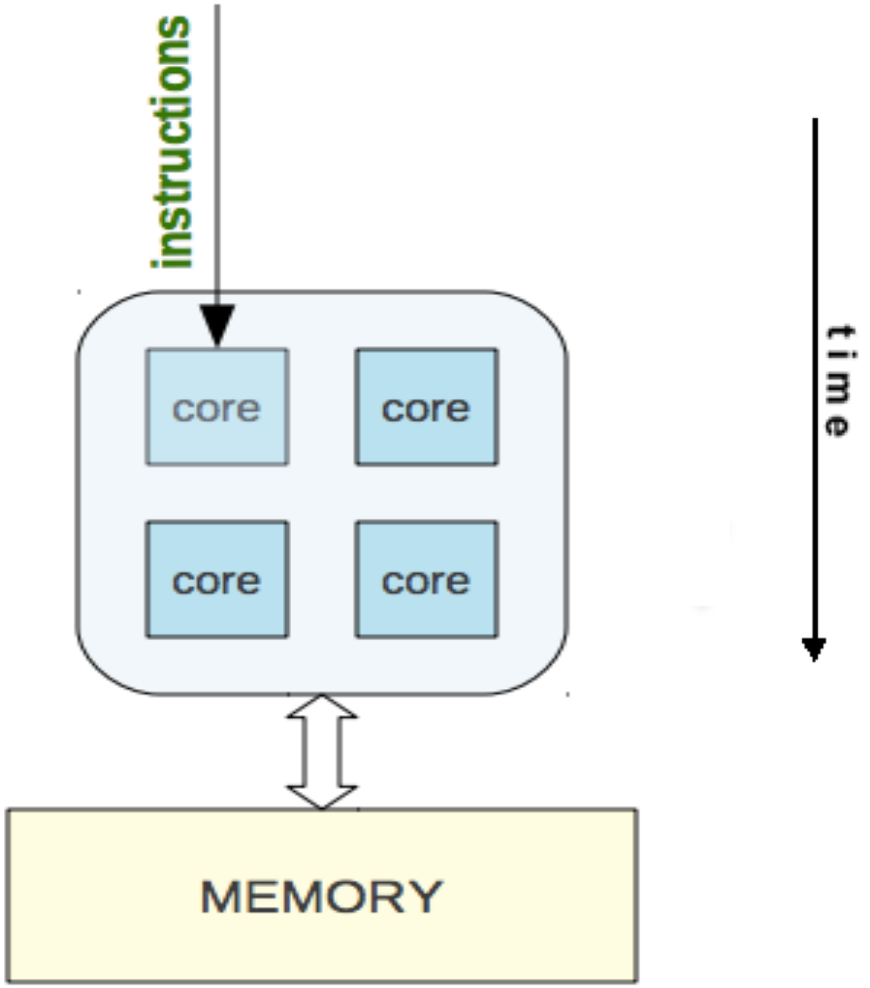
```
#include <mpi.h>;

int main(int argc, char** argv) {
  // Initialize the MPI environment

  // Print off a hello world message
  printf("Hello world from processor );

}
```

```
>>> export MPIRUN=/home/kendall/bin/mpirun
>>> export MPI_HOSTS=host_file
>>> ./run.perl mpi_hello_world
/home/kendall/bin/mpirun -n 4 -f host_file ./mpi_hello_world
Hello world from processor cetus2, rank 1 out of 4 processors
Hello world from processor cetus1, rank 0 out of 4 processors
Hello world from processor cetus4, rank 3 out of 4 processors
Hello world from processor cetus3, rank 2 out of 4 processors
```

# Threads

- threads model of parallel programming, a single process can have multiple, concurrent execution paths
- Each thread has local data, but also, shares the entire resources of executable a.out.

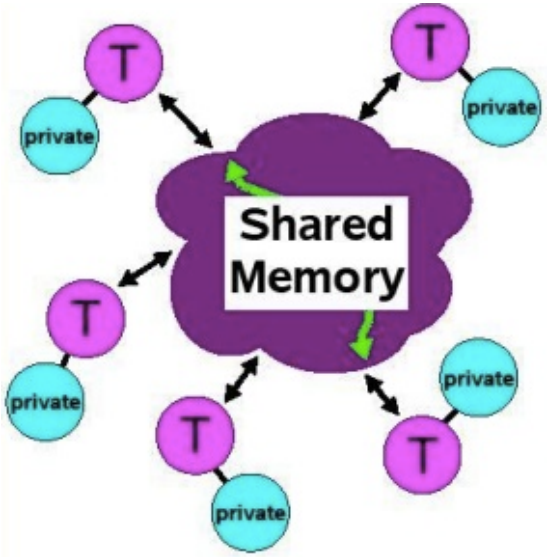
- Threads **communicate** with each other through global memory

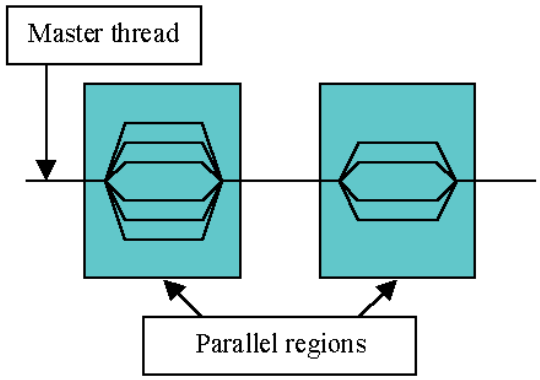


# Parallel programming

Open MultiProcessing (OpenMP)

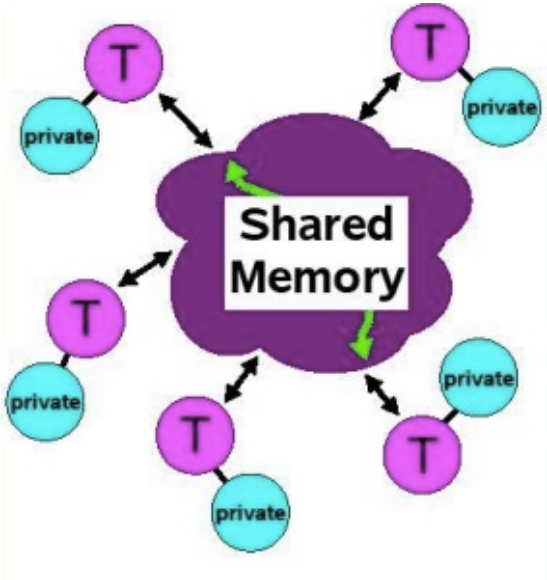
- What is OpenMP?
  - is a library that supports parallel programming in shared-memory parallel machines.
  - allows for the parallel execution of code (*parallel DO loop*), the definition of shared data (*SHARED*), and synchronization of processes



# Parallel programming

- Open MultiProcessing (OpenMP)
  - What is the programming model?
    - All threads have access to the same, globally shared, memory
    - Data can be **shared** or **private**
      - Shared data is accessible by all threads
      - Private data can be accessed only by the threads that owns it




Data transfer is **transparent** to the **programmer**
**Synchronization** takes place, but it is mostly **implicit**

# Example OpenMP Hello World

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {

 printf("Number of threads = “);

}
```

```
$ icc -o omp_helloc -openmp omp_hello.c
omp_hello.c(22): (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
$ export OMP_NUM_THREADS=3
$ ./omp_helloc
Hello World from thread = 0
Hello World from thread = 2
Hello World from thread = 1
Number of threads = 3
```

# Parallel programming

## Pros/Cons of OpenMP

- ✓ easier to program and debug than MPI
- ✓ directives can be added incrementally - gradual parallelization
- ✓ can still run the program **as a serial code**
- ✓ serial code statements usually don't need modification
- ✓ code is easier to **understand** and maybe more easily maintained

- ➢ can only be run in shared memory computers
- ➢ requires a compiler that supports OpenMP
- ➢ mostly used for loop parallelization

## Pros/Cons of MPI

- ✓ runs on either **shared** or **distributed** memory architectures
- ✓ can be used on **a wider range of problems** than OpenMP
- ✓ each process has its **own local variables**
- ✓ distributed memory computers are less expensive than large shared memory computers

- ➢ requires more programming changes to go from serial to parallel version
- ➢ can be harder to debug
- ➢ performance is limited by the communication network between the nodes

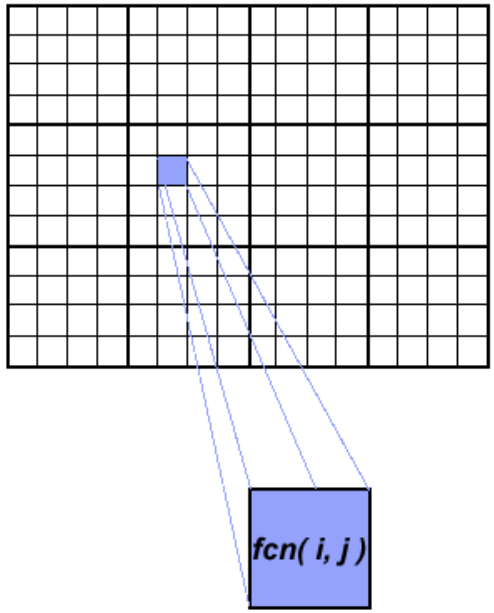
# content

- High Performance Computing
- Computer Architectures
- Speed up
- How to design parallel applications
- Parallel programming models
- Example of Parallel programs

# calculations on 2-dimensional array elements

- The serial program calculates one element at a time in sequential order.
- Serial code could be of the form:

```
do j = 1,n
do i = 1,n
  a(i,j) = fcn(i,j)
end do
end do
```

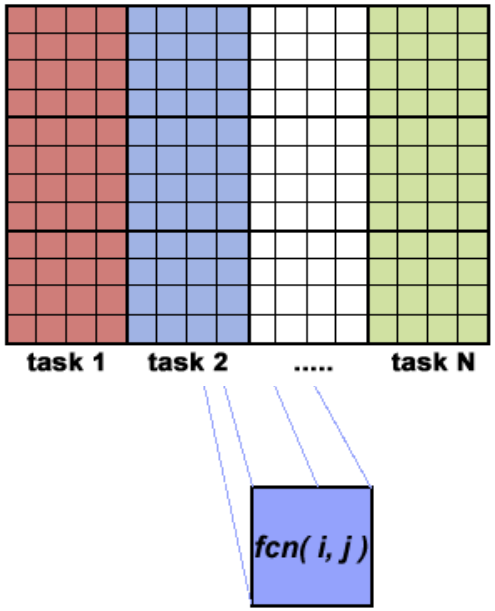



https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch

# calculations on 2-dimensional array elements: solution 1

- Implement as a Single Program Multiple Data (SPMD) model.
- each task executes the portion of the loop corresponding to the data it owns.

```
do j = mystart, myend
do i = 1,n
  a(i,j) = fcn(i,j)
end do
end do
```




https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch

# calculations on 2-dimensional array elements: implementation

- Implement as a Single Program Multiple Data (SPMD) model.
- **Master process** initializes array, sends info to worker processes and receives results.
- **Worker process** receives info, performs its share of computation and sends results to master.

https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch

# calculations on 2-dimensional array elements: implementation

```
find out if I am MASTER or WORKER

if I am MASTER

   initialize the array
   send each WORKER info on part of array it owns
   send each WORKER its portion of initial array

   receive from each WORKER results

else if I am WORKER
   receive from MASTER info on part of array I own
   receive from MASTER my portion of initial array

   # calculate my portion of array
   do j = my first column,my last column
   do i = 1,n
     a(i,j) = fcn(i,j)
   end do
   end do

   send MASTER results

endif
```

https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch

# calculations on 2-dimensional array elements: solution 2

- Solution I : demonstrated static load balancing:
  - Each task has a fixed amount of work to do
  - May be significant idle time for faster or more lightly loaded processors - slowest tasks determines overall performance.
- If you have a load balance problem (some tasks work faster than others),
  - you may benefit by using a "pool of tasks" scheme.

https://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch

# calculations on 2-dimensional array elements: implementation

- Master Process:
  - Holds pool of tasks for worker processes to do
  - Sends worker a task when requested
  - Collects results from workers
- Worker Process: repeatedly does the following
  - Gets task from master process
  - Performs computation
  - Sends results to master

# calculations on 2-dimensional array elements: implementation

```
find out if I am MASTER or WORKER

if I am MASTER

  do until no more jobs
    if request send to WORKER next job
    else receive results from WORKER
  end do

else if I am WORKER

  do until no more jobs
    request job from MASTER
    receive from MASTER next job

    calculate array element: a(i,j) = fcn(i,j)

    send results to MASTER
  end do

endif
```

# References


1. Introduction to Parallel Computinghttps://computing.llnl.gov/tutorials/parallel_comp/#MemoryArch
2. Intro to Parallel Programming . Lesson 2, pt. 1- Shared Memory and threads http://www.youtube.com/watch?v=6sL4C2SwszM
3. Intro to Parallel Programming . Lesson 2, pt. 2- Shared Memory and threads http://www.youtube.com/watch?v=ydG8cOzJjLA
4. Intro to Parallel Programming . Lesson 2, pt. 3- Shared Memory and threads http://www.youtube.com/watch?v=403LWbrA5oU

# TODO (for Students)

| Topics | Organizers | Type/duration |
|---|---|---|
| Intro to distributed sys & BigData | (Adam Belloum, UvA) | Lectures/6 hours |

- Foster et al. "Cloud Computing and Grid Computing 360-Degree Compared," Grid Computing Environments Workshop, 2008. GCE '08 , vol., no., pp.1,10, 12-16 Nov. 2008 doi: 10.1109/GCE.2008.4738445

- Adam Jacobs "The pathologies of big data", Magazine Communications of the ACM ,Vol. 52 Issue 8, Aug. 2009. doi:10.1145/1536616.1536632