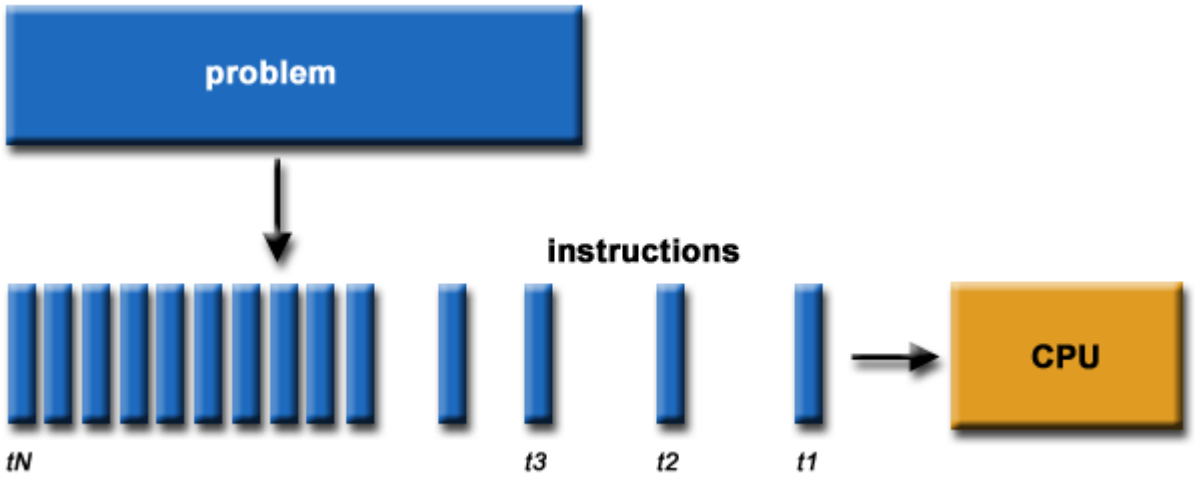
Distributed, parallel, concurrent, High-Performance Computing

Philippe Esling.
Associate professor, UPMC
esling@ircam.fr

Parallel Computing ?

- Software are written for ***serial*** computation:
  - Single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.



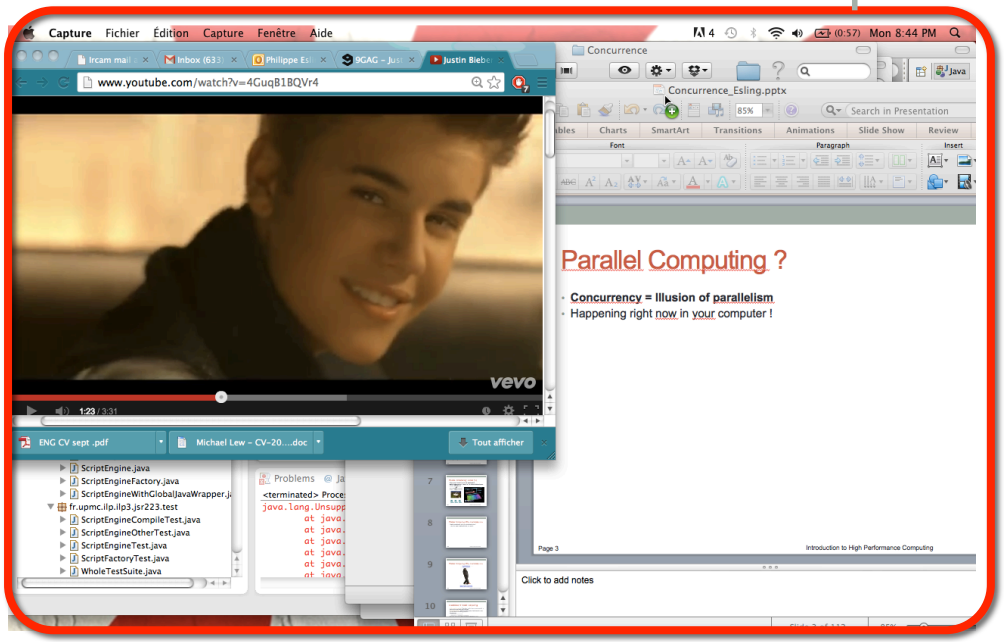
Parallel Computing ?

- **Concurrency = Illusion of parallelism**
- Happening right now in your computer !

Parallel Computing ?

- **Concurrency = Illusion of parallelism**
- Happening right now in your computer !

You working on your computer while …

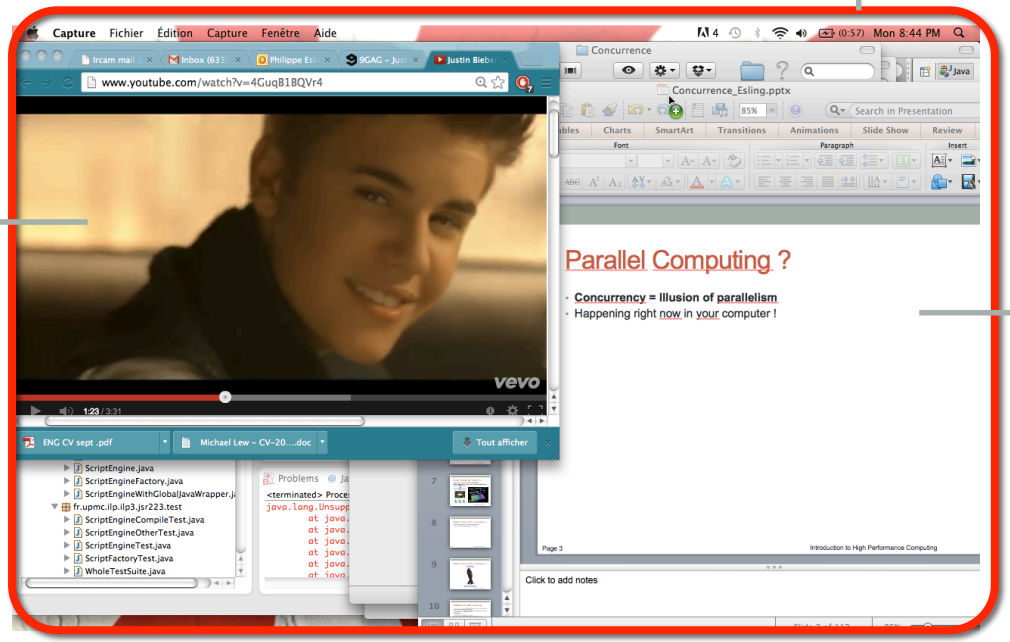


Parallel Computing ?

- **Concurrency = Illusion of parallelism**
- Happening right now in your computer !

You working on your computer while …

Listening to your favorite female singer



+ Preparing your course (Powerpoin tception)

Parallel Computing ?

- **Concurrency = Illusion of parallelism**
- Happening right now in your computer !

10 ms

Parallel Computing ?

- **Concurrency = Illusion of parallelism**
- Happening right now in your computer !

10 ms

Parallel Computing ?

- **Concurrency = Illusion of parallelism**
- Happening right now in your computer !

10 ms

Parallel Computing ?

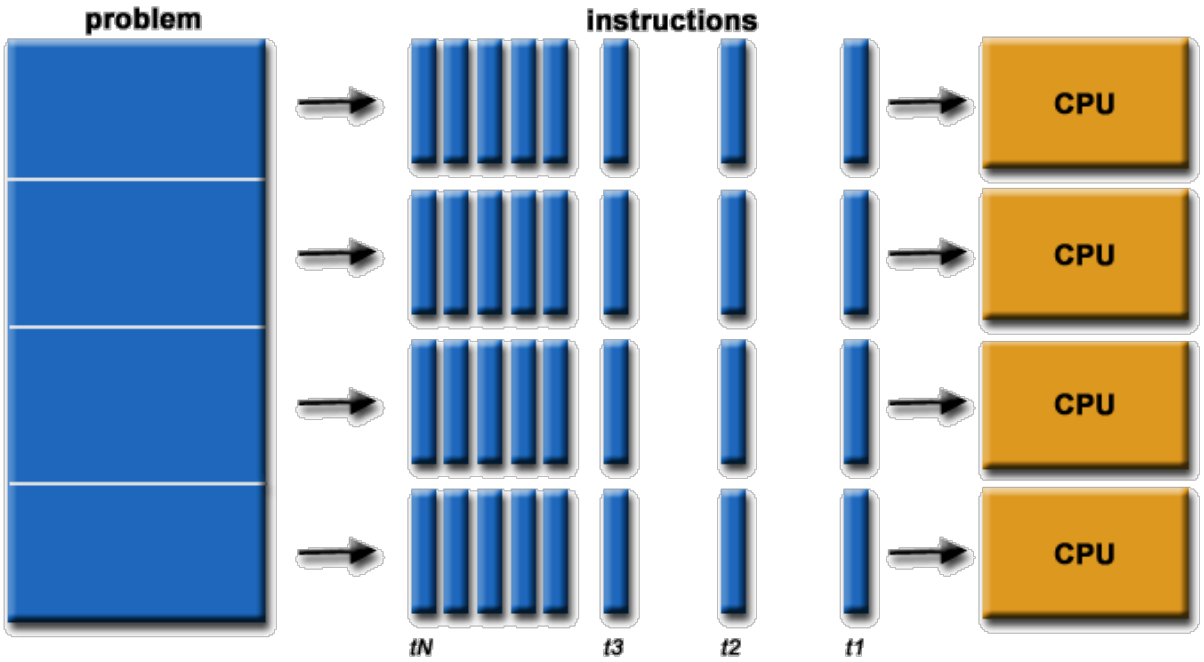
- **Concurrency = Illusion of parallelism**
- Happening right now in your computer !

Parallel Computing ?

- **Concurrency = Illusion of parallelism**
- Happening right now in your computer !

Parallel Computing ?

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



Parallel Computing: *Design*

- *Computing resources* can include:
  - A single computer with multiple processors;
  - A single computer with (multiple) processor(s) and some specialized computer resources (GPU, FPGA …)
  - An arbitrary number of computers connected by a network (**HPC**)
- Computational *problem needs*
  - Discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Solved in less time with multiple compute resources.
- Synchronization, memory sharing and limitations

Parallel Computing … Why ?

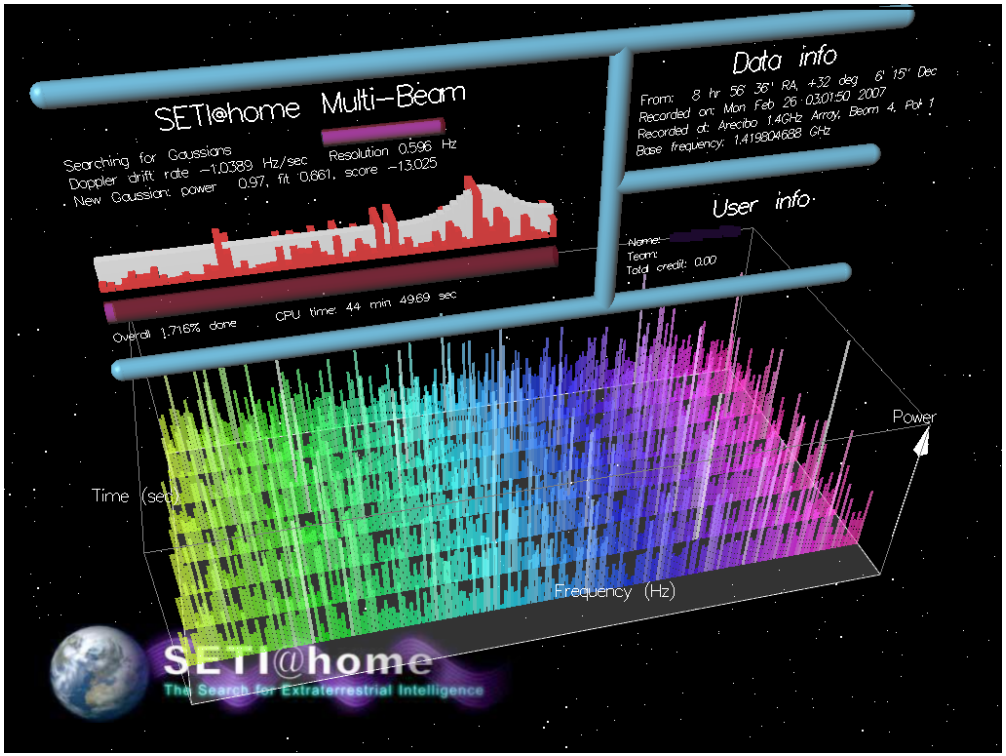
- Legitimate question … Parallel computing is complex !

- The primary reasons for using parallel computing:
  - Save time - wall clock time
  - Solve larger problems
  - Provide concurrency (do multiple things at the same time)

- Other reasons might include:
  - Non-local resources – using resources on wide area networks;
  - Cost savings - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
  - Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

Parallel Computing: *Aliens (?!)*

- *Brilliant case of parallel computing* ***is alien hunting***
- ***NASA, Berkeley project : Search for Extra-Terrestrial Intelligence at home (SETI@home)***



Parallel Computing: *Why Ircam hates me*

- *Parallel computing can help you get your thesis done !*

Name	Topic	F.	C.	S.
Arabic digit	Spoken arabic digits	13	10	8800
Artificial characters	Character recognition	2	10	6000
Australian signs	Hand sign recognition	10	95	6650
Australian signs (HQ)	Hand signs (HQ)	20	95	2565
BciIII-03a-Graz	Brain-Computer	60	4	840
BciIV-01-Berlin	Brain-Computer	64	3	1400
BciIV-03-Freiburg	Brain-Computer	10	4	480
Biomag-2010	EEG analysis	274	2	780
Challenge-2011	Cardiology	12	2	2000
Character-trajectories	Character recognition	3	20	2858
Dachstein	High altitude medicine	3	2	698
Eeg-alcoholism	Medical analysis	64	6	650
Forte-2	Climatology	2	7	121
Forte-6	Climatology	6	7	121
Gaitpdb	Gait analysis	18	2	306
Handwritten	Character recognition	2	183	8235
Ionosphere	Radar analysis	34	2	358
Japanese-vowels	Speech analysis	12	9	640
Libras	Movement recognition	2	15	360

Name	Topic	F.	C.	S.
Pen-chars-35	Character recognition	2	62	1364
Pen-chars-97	Character recognition	2	97	11640
Person activity	Movement analysis	12	11	164860
Physical action	Movement analysis	8	20	80
Ptbdb-1	Cardiology	15	9	2750
Ptbdb-2	Cardiology	15	9	2750
Ptbdb-5	Cardiology	15	9	2750
Robot failures-1	Robotics	6	4	88
Robot failures-2	Robotics	6	5	47
Robot failures-3	Robotics	6	4	47
Robot failures-4	Robotics	6	3	117
Robot failures-5	Robotics	6	5	164
Slpdb	Sleep apnea analysis	7	7	4085
Sonar	Sonar analysis	60	2	208
Synemp	Climatology	2	2	20000
Vfdb	Cardiology	2	15	600
Vicon physical	Physiological analysis	26	20	2000
Wall-robot-4	Robotics	28	4	5460
Wall-robot-24	Robotics	28	24	5460

Parallel Computing: *Why Ircam hates me*

- *Parallel computing can help you get your thesis done !*
- *… And also make the whole lab hate your guts ☺*

```
# Sniffing part
user="esling"
machines=()
for ((i = 1; i < 5000; i++))
do
    ssh $user@m$i -o "ConnectTimeout 1" "ls -la /Applications/ | grep MATLAB | grep -v grep" 2> /dev/null
    if [ $? -eq 0 ]
    then
        machines += (m$i)
    fi
done
# Safe-checking part
echo "Testing on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2010b.app/bin/matlab -n | head -n 3 && exit"
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2008b.app/bin/matlab -n | head -n 3 && exit"
    if [ $? -eq 0 ]
    then
        echo "[OK] - m$i"
    fi
done
# Computation part
matlabWorkDir="Desktop/HVMOTS_Datasets/datasets_Matlab"
echo "Launching on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    echo "[$i / ${#machines[@]}] - Testing m${machines[$i]} availability"
    matlabA="/Applications/MATLAB_R${matlab[$i]}/bin/matlab -nojvm -nodisplay -nodesktop -r pipelineSSH\($i,${#machines[@]}\) </dev/null"
    ssh $user@m${machines[$i]} "cd $matlabWorkDir ; $matlabA ; exit" &
    ssh $user@m${machines[$i]} "watchDogMonitor --command $matlabA ; exit" &
done
```

Parallel Computing: *Why Ircam hates me*

- *Parallel computing can help you get your thesis done !*
- *… And also make the whole lab hate your guts ☺*

```
# Sniffing part
user="esling"
machines=()
for ((i = 1; i < 5000; i++))
do
    ssh $user@m$i -o "ConnectTimeout 1" "ls -la /Applications/ | grep MATLAB | grep -v grep" 2> /dev/null
    if [ $? -eq 0 ]
    then
        machines += (m$i)
    fi
done
# Safe-checking part
echo "Testing on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2010b.app/bin/matlab -n | head -n 3 && exit"
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2008b.app/bin/matlab -n | head -n 3 && exit"
    if [ $? -eq 0 ]
    then
        echo "[OK] - m$i"
    fi
done
# Computation part
matlabWorkDir="Desktop/HVMOTS_Datasets/datasets_Matlab"
echo "Launching on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    echo "[$i / ${#machines[@]}] - Testing m${machines[$i]} availability"
    matlabA="/Applications/MATLAB_R${matlab[$i]}/bin/matlab -nojvm -nodisplay -nodesktop -r pipelineSSH\($i,${#machines[@]}\) </dev/null"
    ssh $user@m${machines[$i]} "cd $matlabWorkDir ; $matlabA ; exit" &
    ssh $user@m${machines[$i]} "watchDogMonitor --command $matlabA ; exit" &
done
```

Sniffer

Parallel Computing: *Why Ircam hates me*

- *Parallel computing can help you get your thesis done !*
- *… And also make the whole lab hate your guts ☺*

```
# Sniffing part
user="esling"
machines=()
for ((i = 1; i < 5000; i++))
do
    ssh $user@m$i -o "ConnectTimeout 1" "ls -la /Applications/ | grep MATLAB | grep -v grep" 2> /dev/null
    if [ $? -eq 0 ]
    then
        machines += (m$i)
    fi
done
# Safe-checking part
echo "Testing on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2010b.app/bin/matlab -n | head -n 3 && exit"
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2008b.app/bin/matlab -n | head -n 3 && exit"
    if [ $? -eq 0 ]
    then
        echo "[OK] - m$i"
    fi
done
# Computation part
matlabWorkDir="Desktop/HVMOTS_Datasets/datasets_Matlab"
echo "Launching on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    echo "[$i / ${#machines[@]}] - Testing m${machines[$i]} availability"
    matlabA="/Applications/MATLAB_R${matlab[$i]}/bin/matlab -nojvm -nodisplay -nodesktop -r pipelineSSH\($i,${#machines[@]}\) </dev/null"
    ssh $user@m${machines[$i]} "cd $matlabWorkDir ; $matlabA ; exit" &
    ssh $user@m${machines[$i]} "watchDogMonitor --command $matlabA ; exit" &
done
```

Sniffer

Checker

Parallel Computing: *Why Ircam hates me*

- *Parallel computing can help you get your thesis done !*
- *… And also make the whole lab hate your guts ☺*

```
# Sniffing part
user="esling"
machines=()
for ((i = 1; i < 5000; i++))
do
    ssh $user@m$i -o "ConnectTimeout 1" "ls -la /Applications/ | grep MATLAB | grep -v grep" 2> /dev/null
    if [ $? -eq 0 ]
    then
        machines += (m$i)
    fi
done
# Safe-checking part
echo "Testing on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2010b.app/bin/matlab -n | head -n 3 && exit"
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2008b.app/bin/matlab -n | head -n 3 && exit"
    if [ $? -eq 0 ]
    then
        echo "[OK] - m$i"
    fi
done
# Computation part
matlabWorkDir="Desktop/HVMOTS_Datasets/datasets_Matlab"
echo "Launching on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    echo "[$i / ${#machines[@]}] - Testing m${machines[$i]} availability"
    matlabA="/Applications/MATLAB_R${matlab[$i]}/bin/matlab -nojvm -nodisplay -nodesktop -r pipelineSSH\($i,${#machines[@]}\) </dev/null"
    ssh $user@m${machines[$i]} "cd $matlabWorkDir ; $matlabA ; exit" &
    ssh $user@m${machines[$i]} "watchDogMonitor --command $matlabA ; exit" &
done
```

Sniffer

Checker

Computing

Parallel Computing: *Why Ircam hates me*

- *Parallel computing can help you get your thesis done !*
- *… And also make the whole lab hate your guts ☺*

```
# Sniffing part
user="esling"
machines=()
for ((i = 1; i < 5000; i++))
do
    ssh $user@m$i -o "ConnectTimeout 1" "ls -la /Applications/ | grep MATLAB | grep -v grep" 2> /dev/null
    if [ $? -eq 0 ]
    then
        machines += (m$i)
    fi
done
# Safe-checking part
echo "Testing on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2010b.app/bin/matlab -n | head -n 3 && exit"
    ssh $user@m${machines[$i]} "nice +5 /Applications/MATLAB_R2008b.app/bin/matlab -n | head -n 3 && exit"
    if [ $? -eq 0 ]
    then
        echo "[OK] - m$i"
    fi
done
# Computation part
matlabWorkDir="Desktop/HVMOTS_Datasets/datasets_Matlab"
echo "Launching on grid : ${#machines[@]} computers"
for ((i = 0; i < ${#machines[@]}; i++))
do
    echo "[$i / ${#machines[@]}] - Testing m${machines[$i]} availability"
    matlabA="/Applications/MATLAB_R${matlab[$i]}/bin/matlab -nojvm -nodisplay -nodesktop -r pipelineSSH\($i,${#machines[@]}\) </dev/null"
    ssh $user@m${machines[$i]} "cd $matlabWorkDir ; $matlabA ; exit" &
    ssh $user@m${machines[$i]} "watchDogMonitor --command $matlabA ; exit" &
done
```

Sniffer

Checker

Computing

Watchdog ! (auto-restart if stopped)

Parallel Computing: *Why Ircam hates me*

Limitations of Serial Computing

- Limits to serial computing - Physical and practical reasons pose constraints to simply building ever faster serial computers.
- Transmission speeds – Speed of a serial computer directly dependent upon how fast data can move through hardware.
- Absolute limits
  - speed of light (30 cm/nanosecond)
  - transmission limit of copper wire (9 cm/nanosecond).
- Increasing speeds necessitate increasing proximity of processors.
- Limits to miniaturization - increasing number of transistors on chip.
- Even molecular or atomic-level components (quantum computer), limit will be reached on how small components.
- Economic limitations – Increasingly expensive for faster single processor.
- Inexpensive to use larger number of lame processors to achieve same (or better) performance.

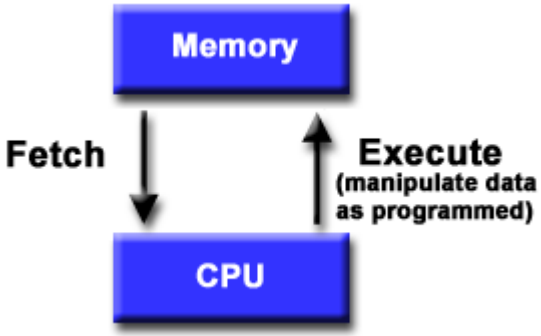
The future

- The past 10 years, the trends are indicated
  - Ever faster networks
  - Distributed systems
  - Multi-processor computer architectures
- ***Parallelism is the future of computing***.
- Will be multi-forms, mixing general purpose solutions (your PC) and very speciliazed solutions …
- Cf. PS3 HPC in Top500 ☺

Von Neumann Architecture

- All computers have followed the common machine model known as the von Neumann computer.
- *Stored-program* concept : The CPU executes a stored program with a sequence of read and write on memory.

- Basic design
  - Memory used to store program and data
  - Program instructions are coded data
  - Data is simply information to be used
- CPU gets instructions and/or data from memory, decodes the instructions and then ***sequentially*** performs them.



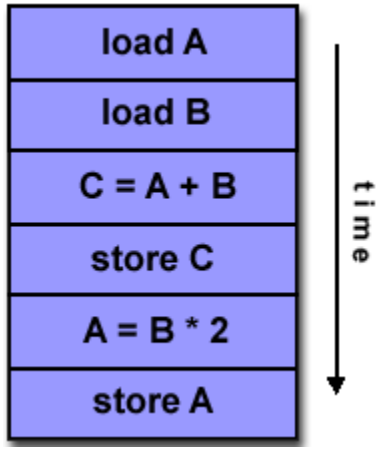
Flynn's Classical Taxonomy

- Way to classify parallel computers.
- Distinguishes multi-processor computer architectures along two independent dimensions: ***Instruction*** and ***Data***.

S I S D **Single Instruction, Single Data**	**S I M D** **Single Instruction, Multiple Data**
M I S D **Multiple Instruction, Single Data**	**M I M D** **Multiple Instruction, Multiple Data**

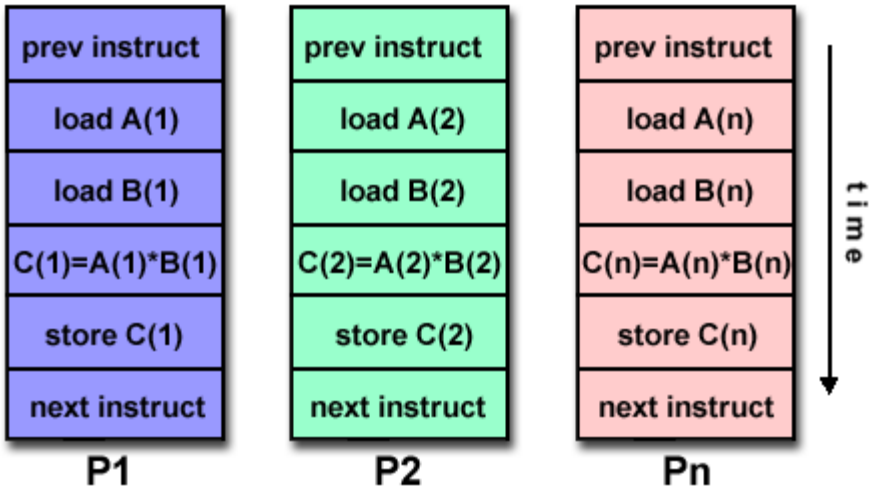
Single Instruction, Single Data (SISD)

- A serial (non-parallel) computer
- Single instruction: one instruction is processed by the CPU at each clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer



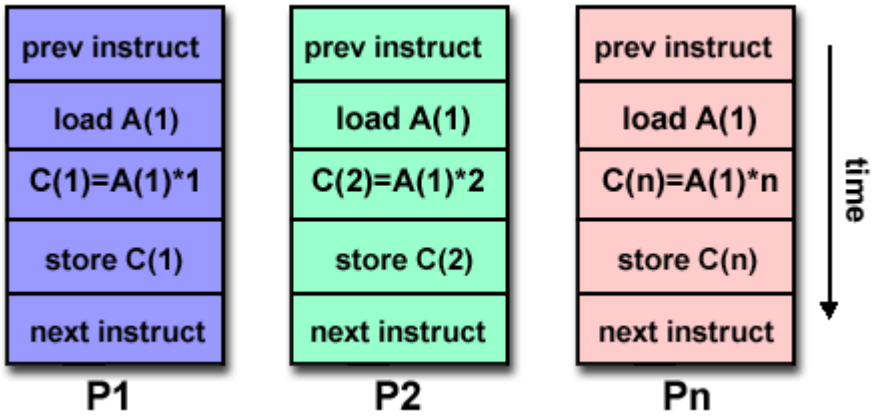
Single Instruction, Multiple Data (SIMD)

- Single instruction: All processing units execute the same instruction at any given clock cycle
- Multiple data: Each processing unit operate on different data
- Suited for problems with high regularity (image processing).
- Synchronous (lockstep) and deterministic execution



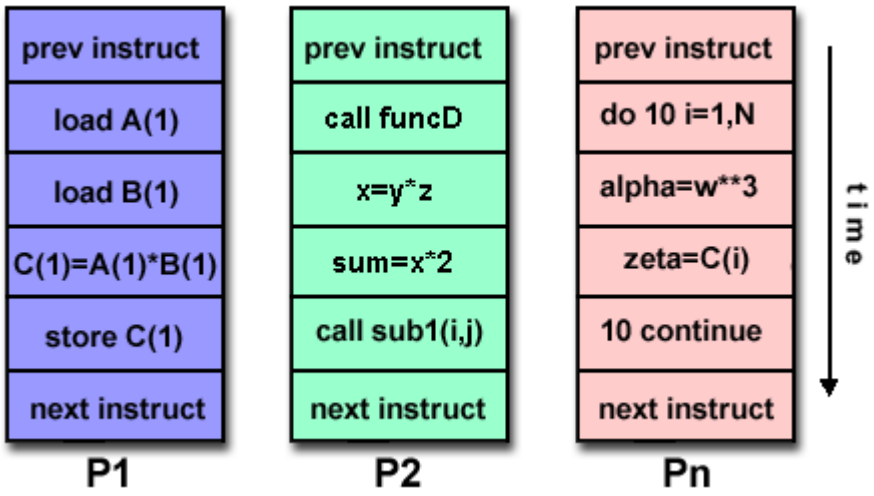
Multiple Instruction, Single Data (MISD)

- Single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently.
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
- multiple cryptography algorithms attempting to crack.



Multiple Instruction, Multiple Data (MIMD)

- Most common type of parallel computer.
- Multiple Instruction: every processor execute different instructions
- Multiple Data: every processor work with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Examples: most current supercomputers (HPC), networked parallel computer "grids" and multi-processor SMP computers.



General Parallel Terminology

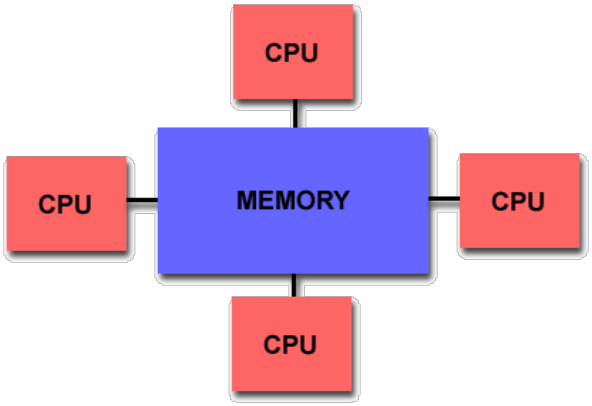
- **Task**
  - Discrete section of computational work. Typically a set of instructions.
- **Parallel Task**
  - A task that can be executed by multiple processors safely
- **Serial Execution**
  - Execution of a program sequentially (what happens on one processor)
- **Parallel Execution**
  - Execution of a program by more than one task each execute at the same moment.
- **Shared Memory**
  - Computer architecture with direct access to common physical memory.
- **Distributed Memory**
  - Network-based memory access for non-common physical memory.
- **Communication**
  - Parallel tasks need to exchange data (shared memory, NFS)
- **Synchronization**
  - The coordination of parallel tasks in real time (wait, signal, cooperate)

General Parallel Terminology

- **Granularity**
  - Qualitative measure of the ratio of computation to communication.
  - ***Coarse:*** large amounts of computation done between communication
  - ***Fine:*** Small amounts of computation between communication events
- **Observed Speedup**
  - Indicators for a parallel program's performance.
- **Parallel Overhead**
  - Amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
    - Task start-up time; Synchronizations; Data communications; Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
- **Massively Parallel**
  - Hardware for parallel system - many processors.
- **Scalability**
  - Ability of the system to demonstrate a proportionate increase in parallel speedup with the addition of more processors

Shared Memory

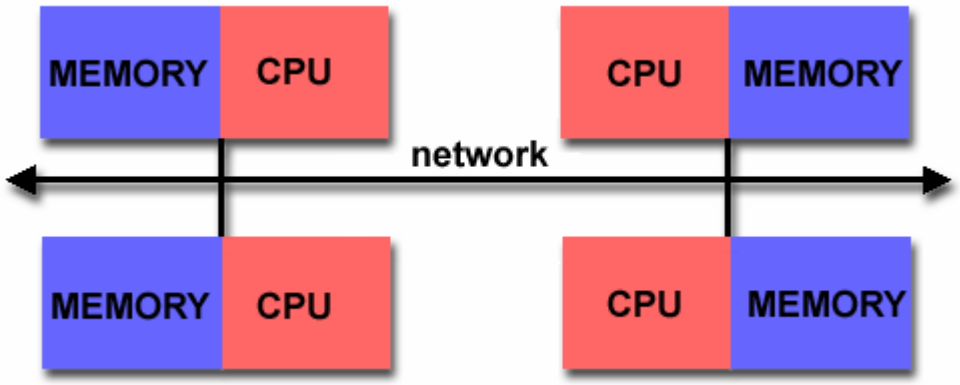
- Single memory as global address space.
- Processors operate independently but share the same memory.
- Changes in a memory location effected by one processor are visible to all others.



- Advantages
  - User-friendly programming perspective to memory
  - Data sharing between tasks is both fast and uniform (proximity)
- Disadvantages:
  - Lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path.
  - **Mandatory synchronization constructs** that insure "correct" access of global memory. (cf. **mutex, semaphore, conditions**)
  - Increasingly difficult and expensive to design

Distributed Memory

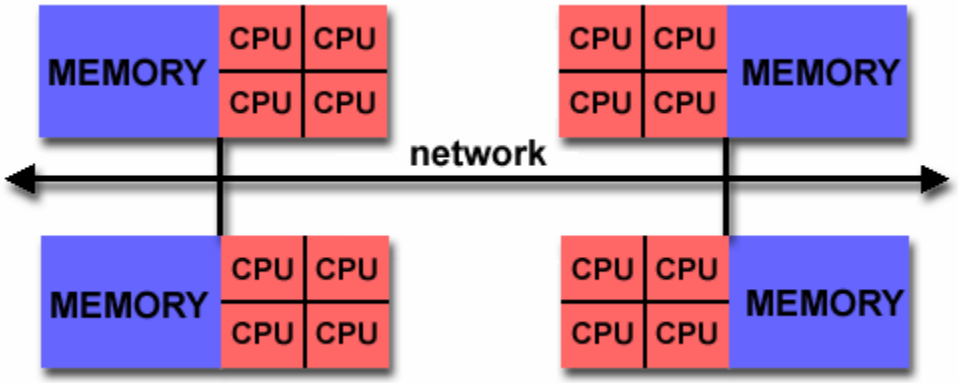
- Distributed memory require communication network between processor memory.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor.
- Because each processor has its own local memory, it operates independently.
- Concept of cache coherency does not apply.
- Needs to explicitly define how and when data is communicated.
- Synchronization between tasks is required



Distributed Memory: Pro and Con

- Advantages
  - Memory is scalable with number of processors.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use crappy old processors and networking.
  - Cf. **Beowulf cluster**
- Disadvantages
  - Requires data communication and synchronization between processors.
  - Difficult to map existing data structures, based on global memory.
  - Non-uniform memory access (NUMA) times

Hybrid Distributed-Shared Memory



- Shared memory is usually a cache coherent machine.
- Processors on one can address that machine's memory as global.
- Distributed memory component is the
- Current trends as this memory architecture prevails.

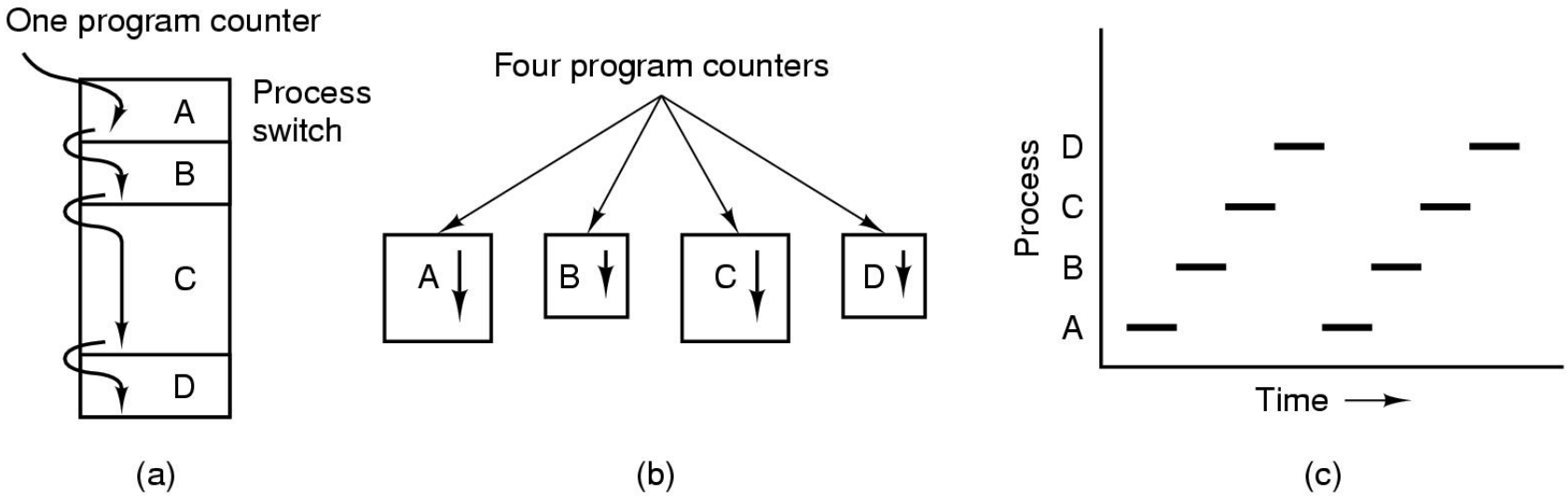
Parallel programming models

- Several parallel programming models in common use:
  - *Shared Memory*
  - *Threads*
  - *Message Passing*
  - *Data Parallel*
  - *Hybrid*
- Models are abstraction above hardware and memory architectures.
- Models are not specific to a particular type of machine or memory architecture.
- Which model should I use ?...
- There is no "best" model but some are **more appropriate depending on the certain problem**

Shared Memory Model

- Shared-memory model :
  - Tasks share a common address space
  - Read and write operations are asynchronous.
- Mechanisms (locks / semaphores) are required to control access to shared memory.
- Advantages of this model
  - No notion of data "ownership”
  - No need to specify explicitly the communication of data between tasks.
  - Communication is implicit (through shared memory)
- Important disadvantage in terms of performance : more difficult to understand and manage data locality.

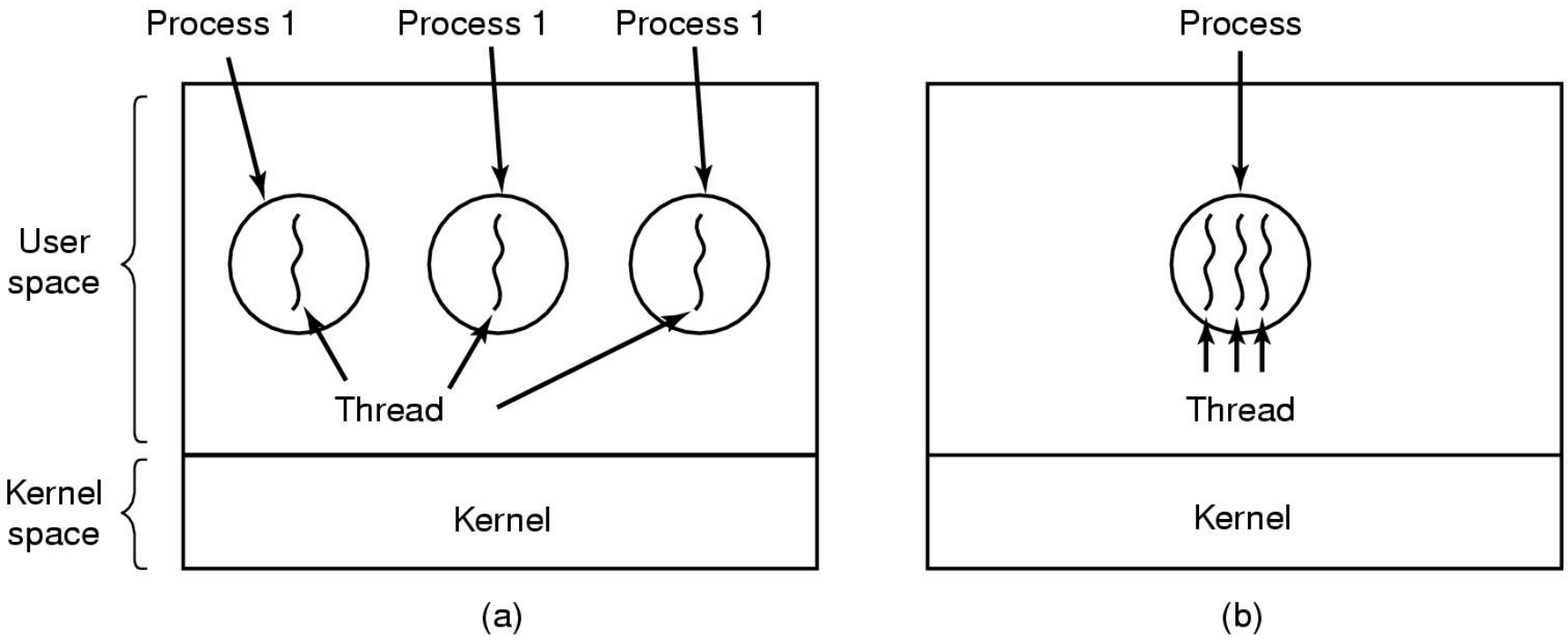
Threads Model



Single process can have multiple, concurrent execution paths.

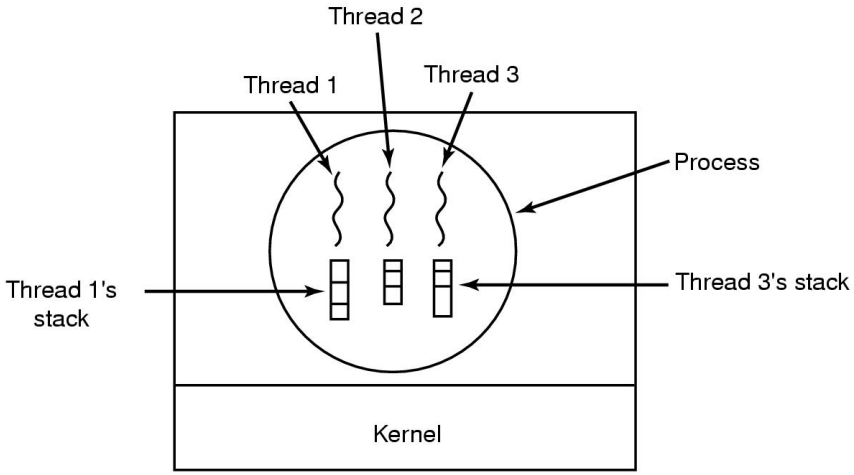
- Most simple analogy : single program that includes a number of subroutines:
  - Main program is scheduled to run by the native operating system.
  - Master **creates a number of tasks (threads)** that can be scheduled concurrently.
  - **Each thread has local data**, but also, **shares the entire resources**.
  - Saves the overhead associated with replicating a program's resources.
  - Each thread also benefits from a global memory view.
  - A thread's work may best be described as a subroutine within the main program.
  - **Threads communicate** with each other **through global memory**.

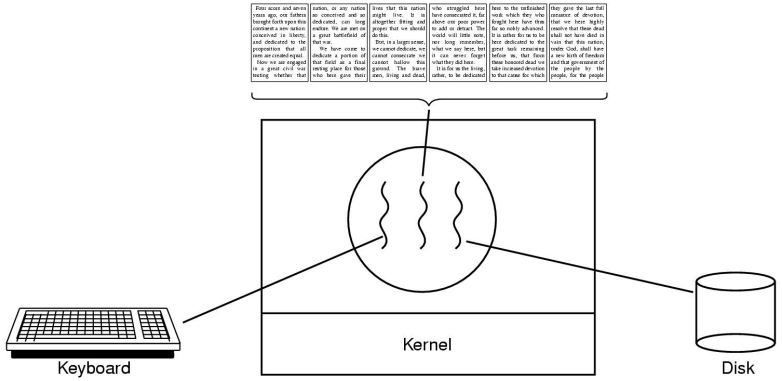
Threads Model

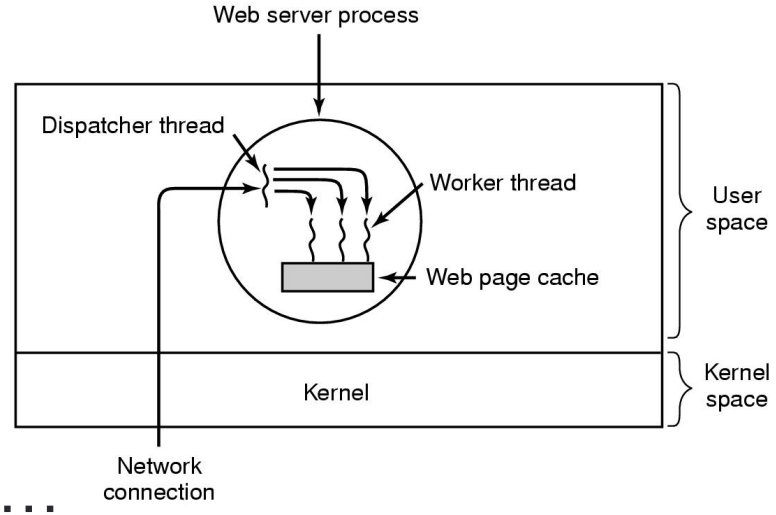


(a) Three processes with each a single thread
(b) One process with three threads

Threads Model







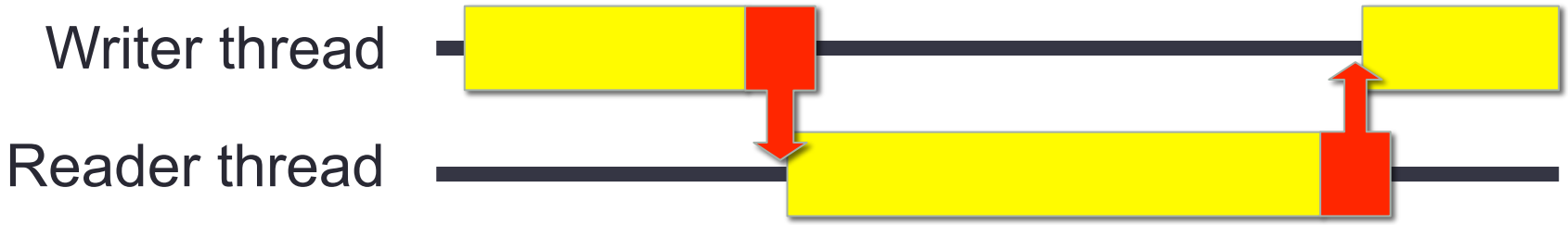
- Each thread has its own stack …
- But **global memory is shared**
- **Communication is *implicit*** (shared memory)
- **Synchronization must be *explicit***

Threads Model : Critical region

2 huge problems with the threads model :

- **Shared memory** between threads
- Evaluation is **completely non-deterministic**

Give rise to the problem of memory coherence, known as the classical **writer / reader problem** (also called producer / consumer)



Threads Model : Critical region

2 huge problems with the threads model :

- **Shared memory** between threads
- Evaluation is **completely non-deterministic**

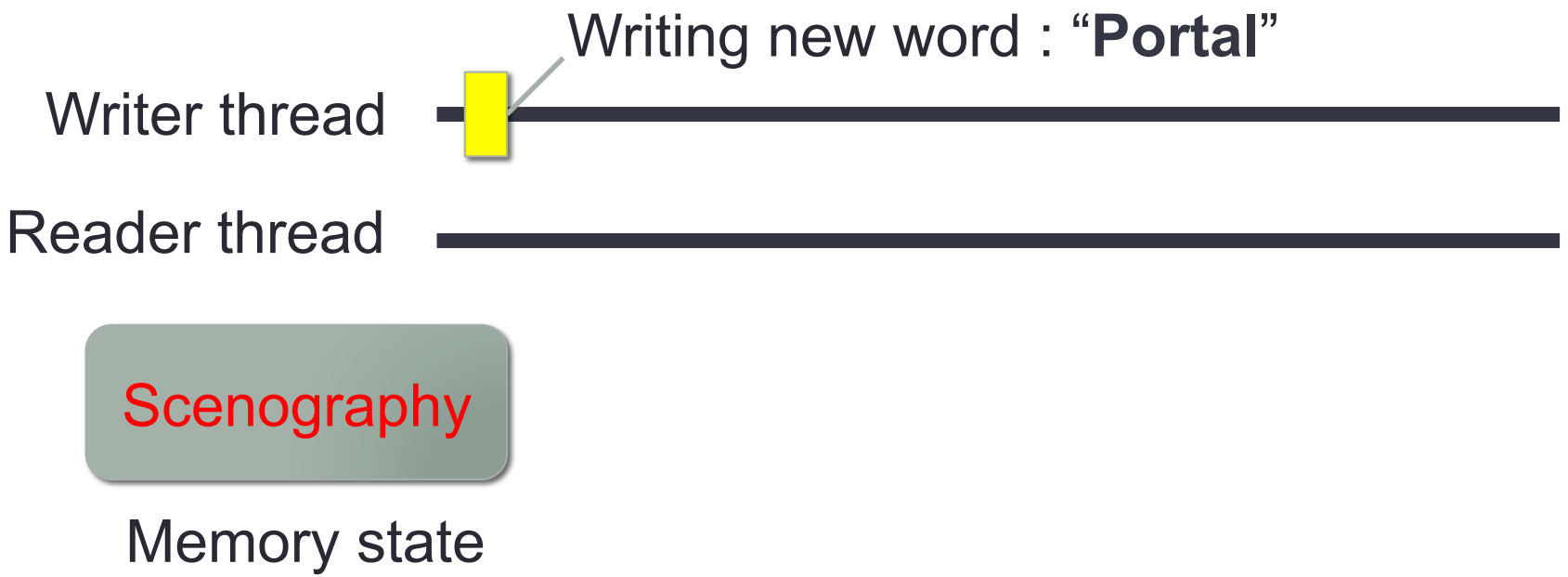
Give rise to the problem of memory coherence, known as the classical **writer / reader problem** (also called producer / consumer)

Threads Model : Critical region

2 huge problems with the threads model :

- **Shared memory** between threads
- Evaluation is **completely non-deterministic**

Give rise to the problem of memory coherence, known as the classical **writer / reader problem** (also called producer / consumer)



Threads Model : Critical region

2 huge problems with the threads model :

- **Shared memory** between threads
- Evaluation is **completely non-deterministic**

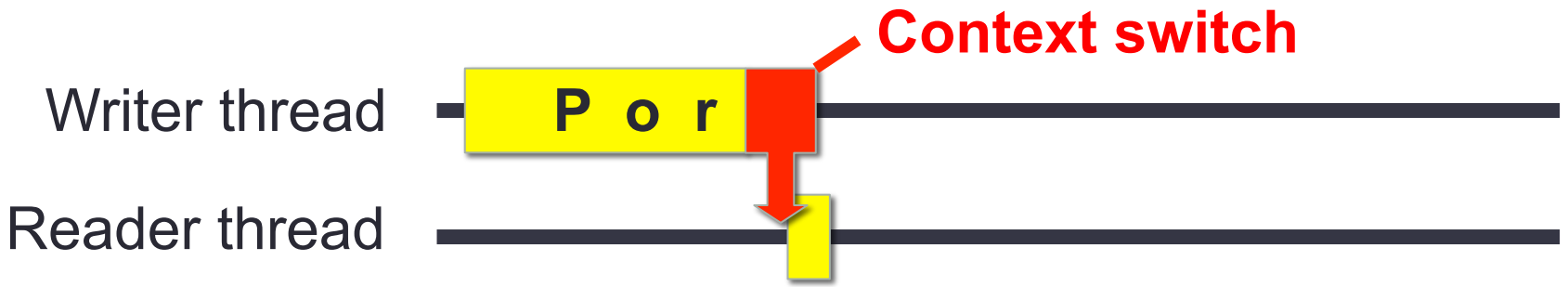
Give rise to the problem of memory coherence, known as the classical **writer / reader problem** (also called producer / consumer)

Threads Model : Critical region

2 huge problems with the threads model :

- **Shared memory** between threads
- Evaluation is **completely non-deterministic**

Give rise to the problem of memory coherence, known as the classical **writer / reader problem** (also called producer / consumer)



Scenography

Memory state ?

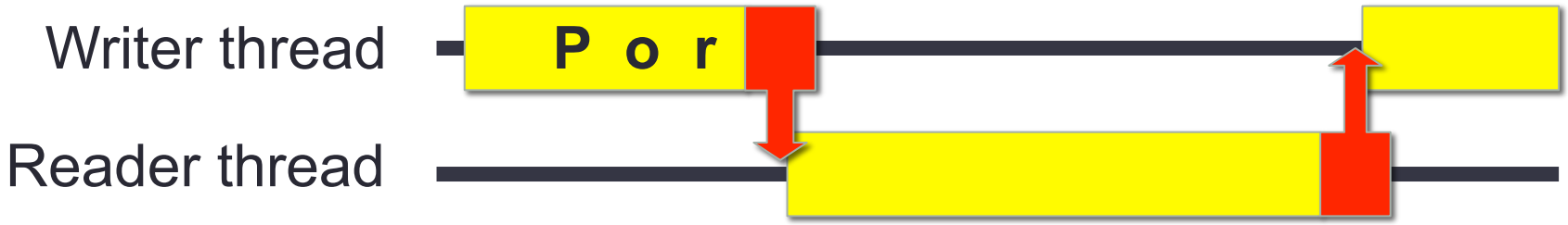
Memory state

Threads Model : Critical region

2 huge problems with the threads model :

- **Shared memory** between threads
- Evaluation is **completely non-deterministic**

Give rise to the problem of memory coherence, known as the classical **writer / reader problem** (also called producer / consumer)

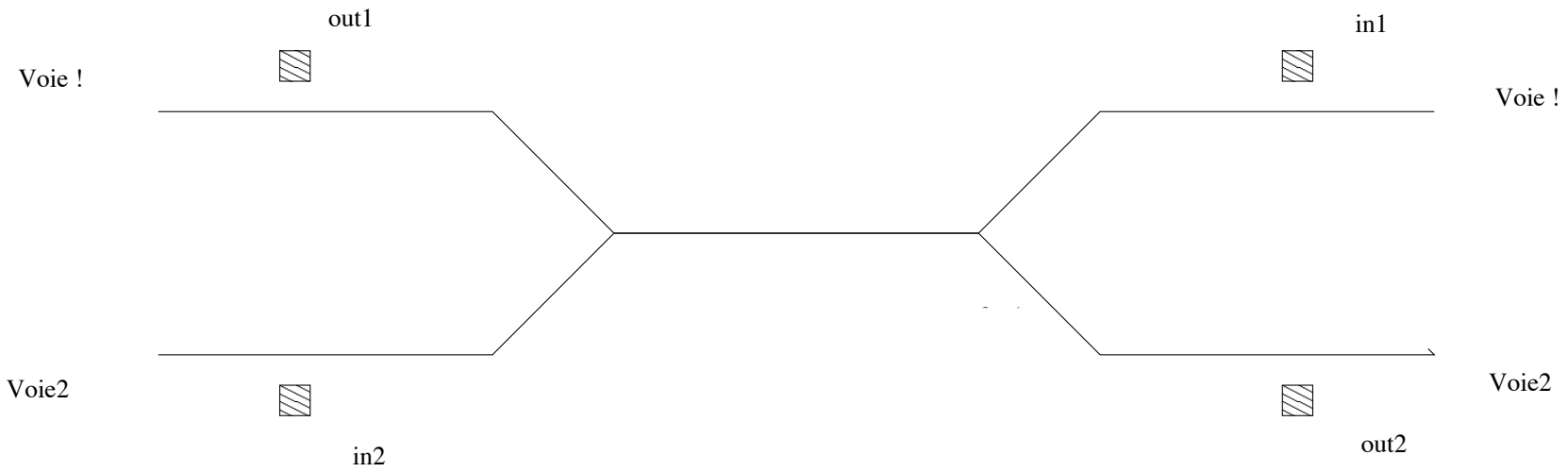


Scenography

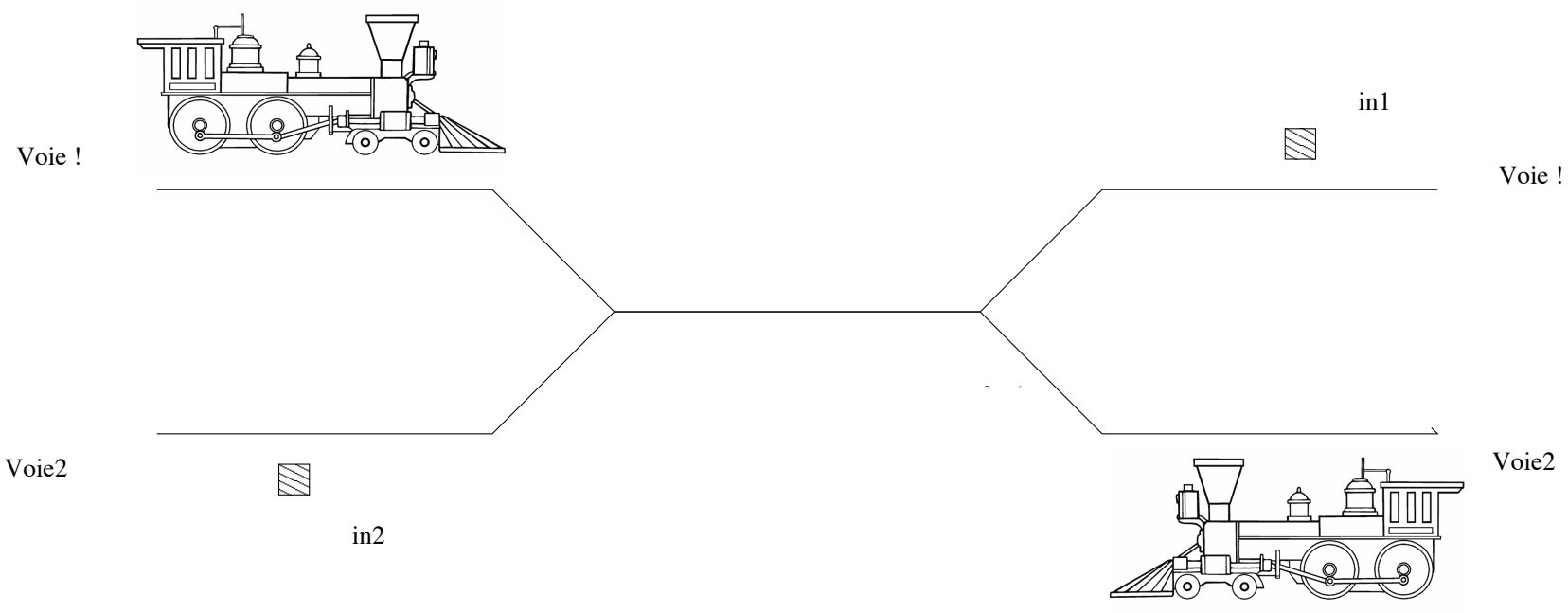
Pornography

Memory state

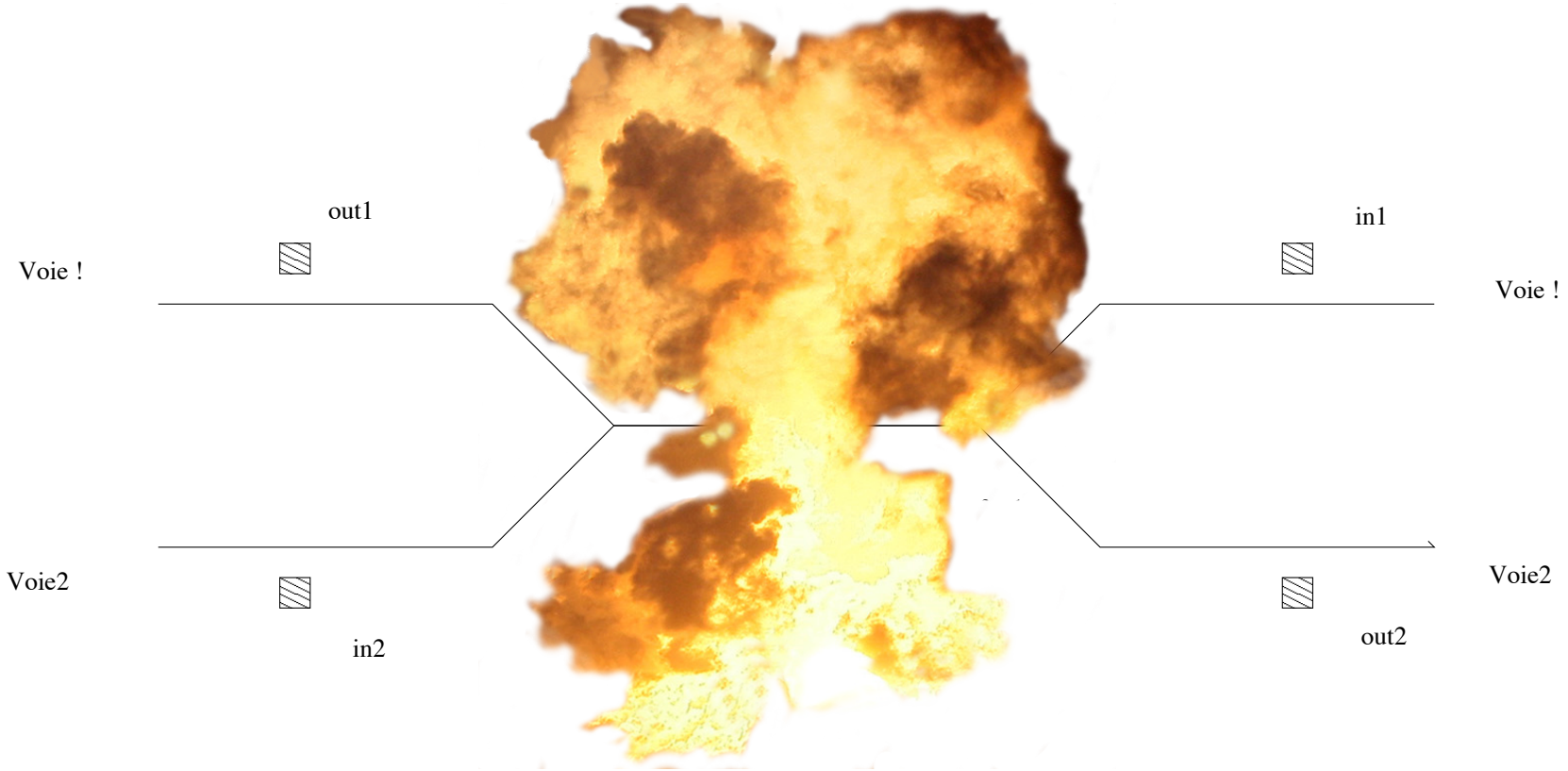
Threads Model : Critical region



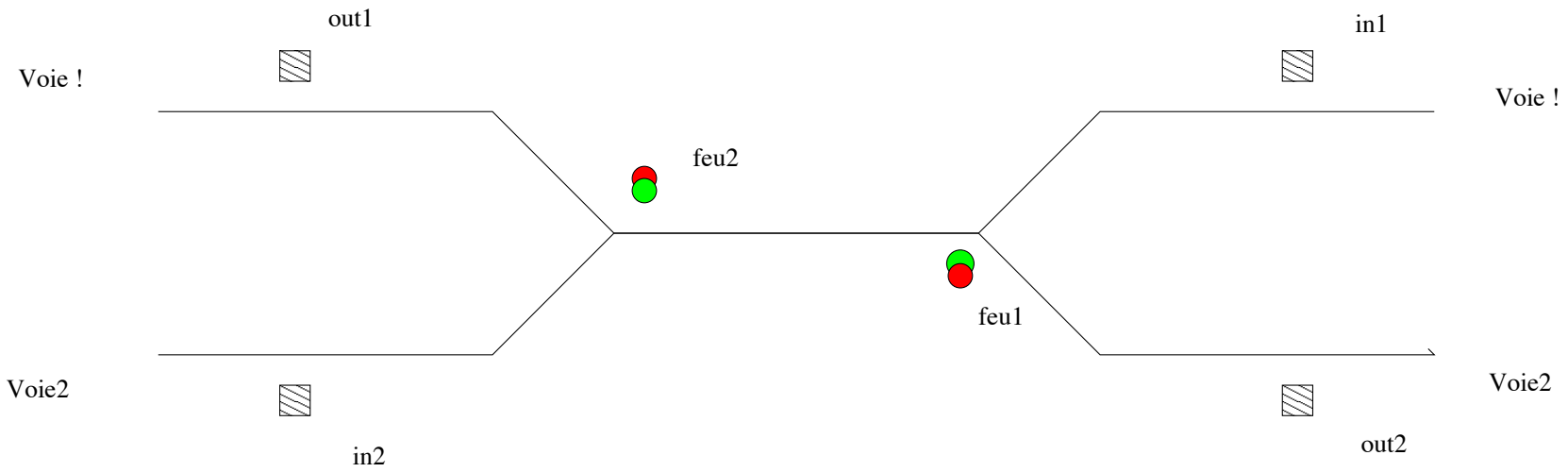
Threads Model : Critical region



Threads Model : Critical region



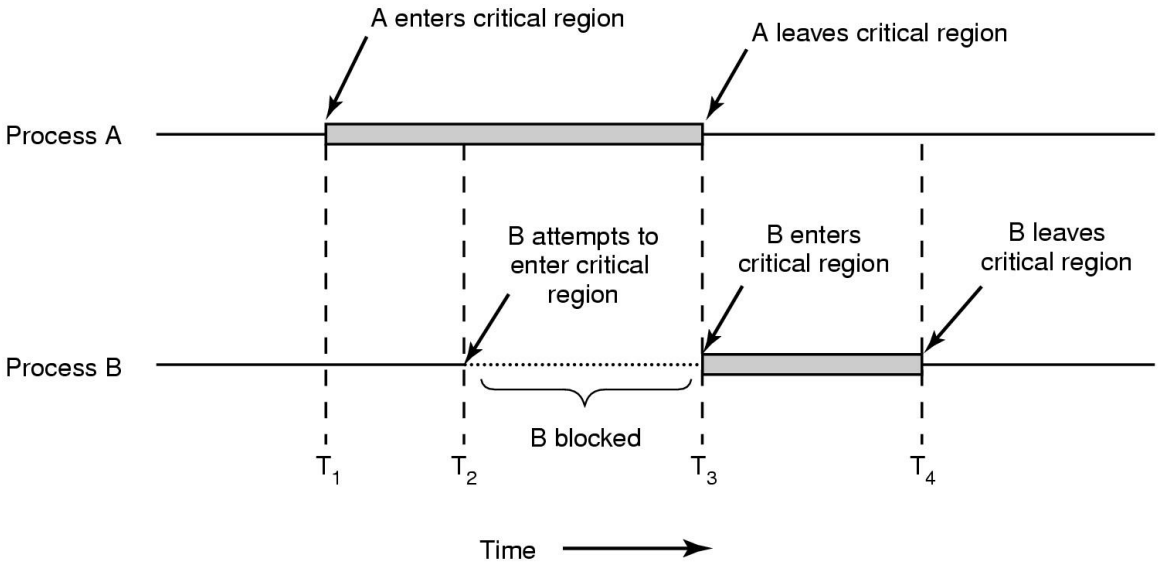
Threads Model : Critical region



Threads Model : Mutual exclusion

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region (**safety**)
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside critical region may block another
4. No process must wait forever to enter its critical region (**liveness**)



Mutual exclusion through **critical regions**

Threads Model : Mutual exclusion

Busy waiting solution (not that good)

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

- Busy waiting situation !
- We use several processor cycles for nothing
- Best if the thread sleeps, waiting for its turn

Threads Model : Mutual exclusion

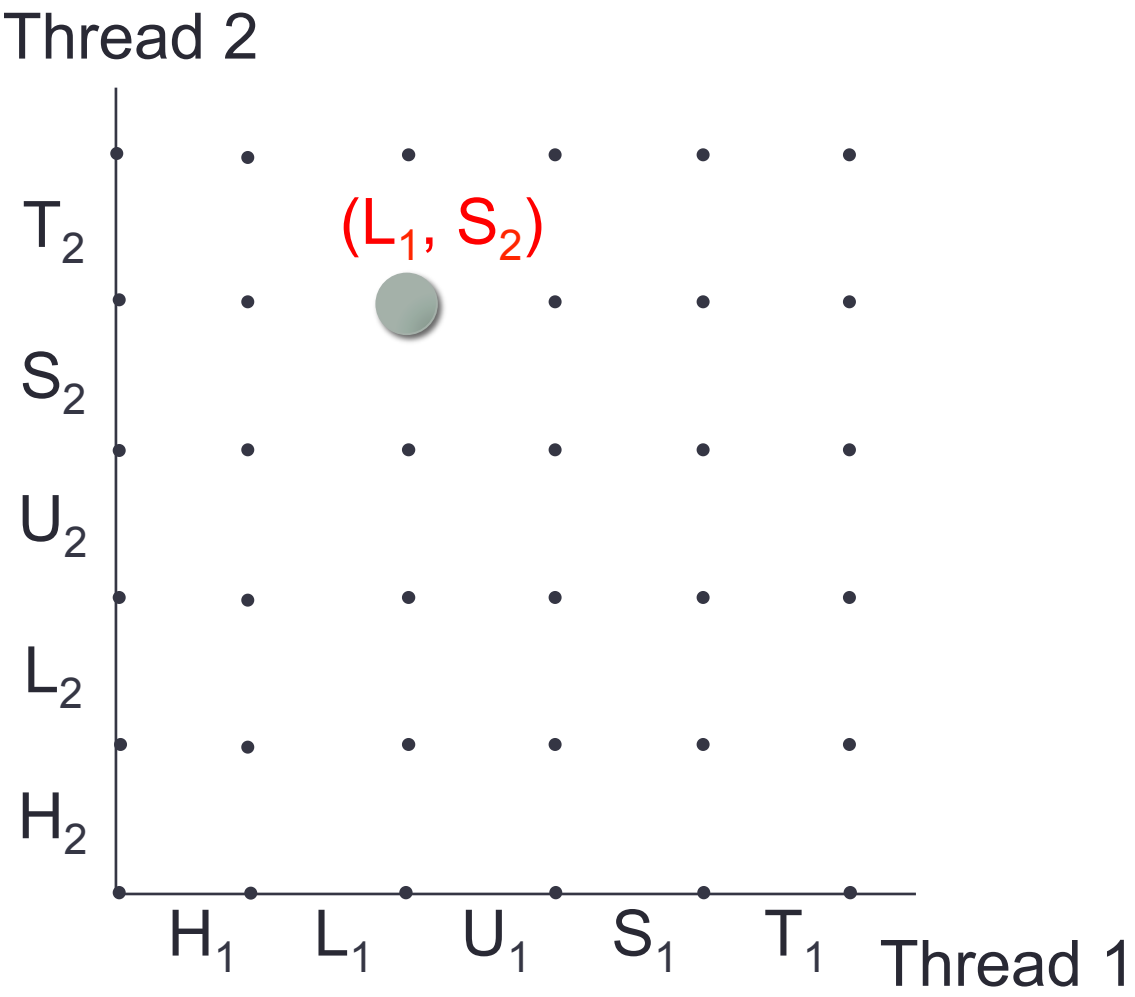
```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

We avoid busy waiting through

1. Sleep
2. Wakeup

(Conditions)

Progress Graphs



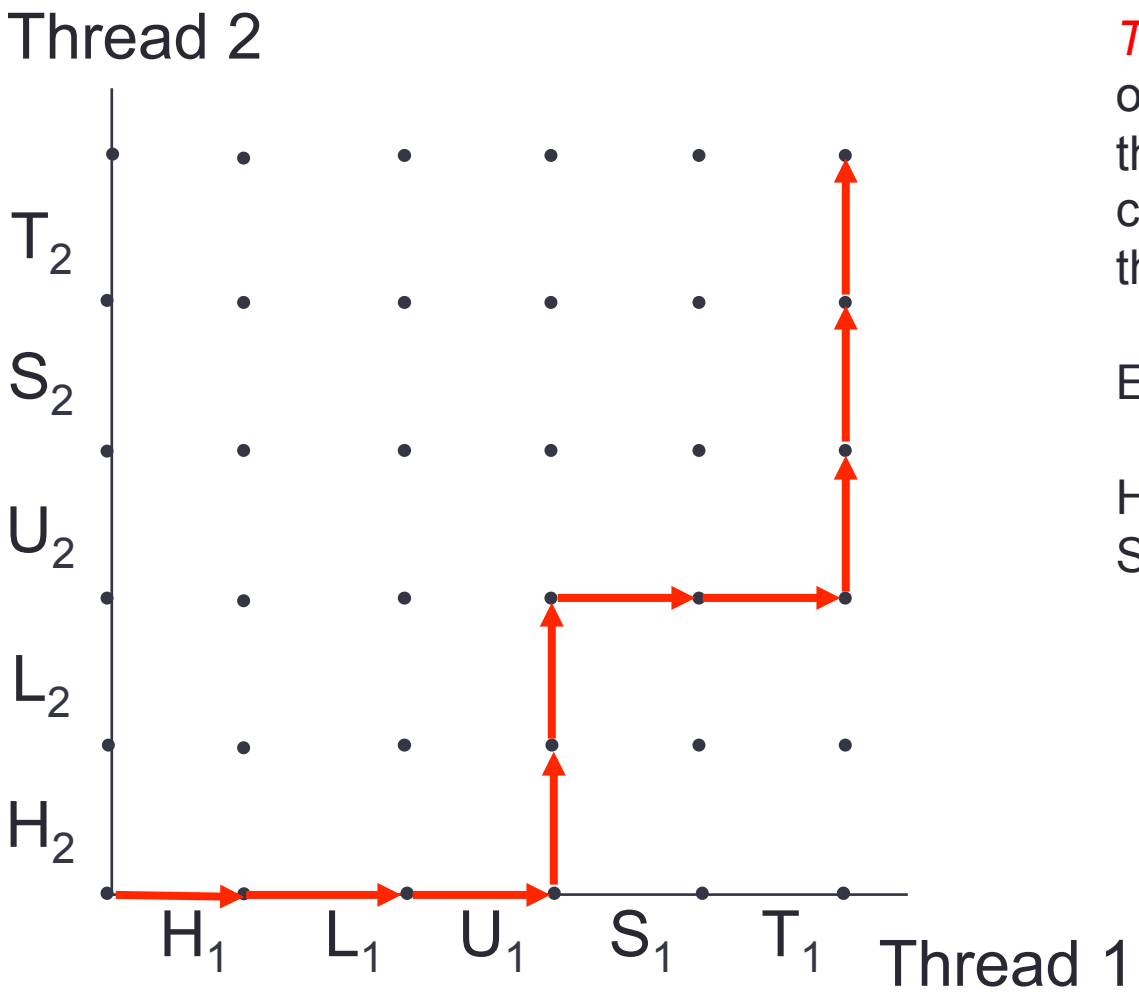
Progress graph depicts discrete *execution state space* of concurrent threads

Each axis corresponds to sequential order of instructions in a thread

Each point corresponds to a possible *execution state* ($Inst_1$, $Inst_2$)

E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2

Trajectories in Progress Graphs

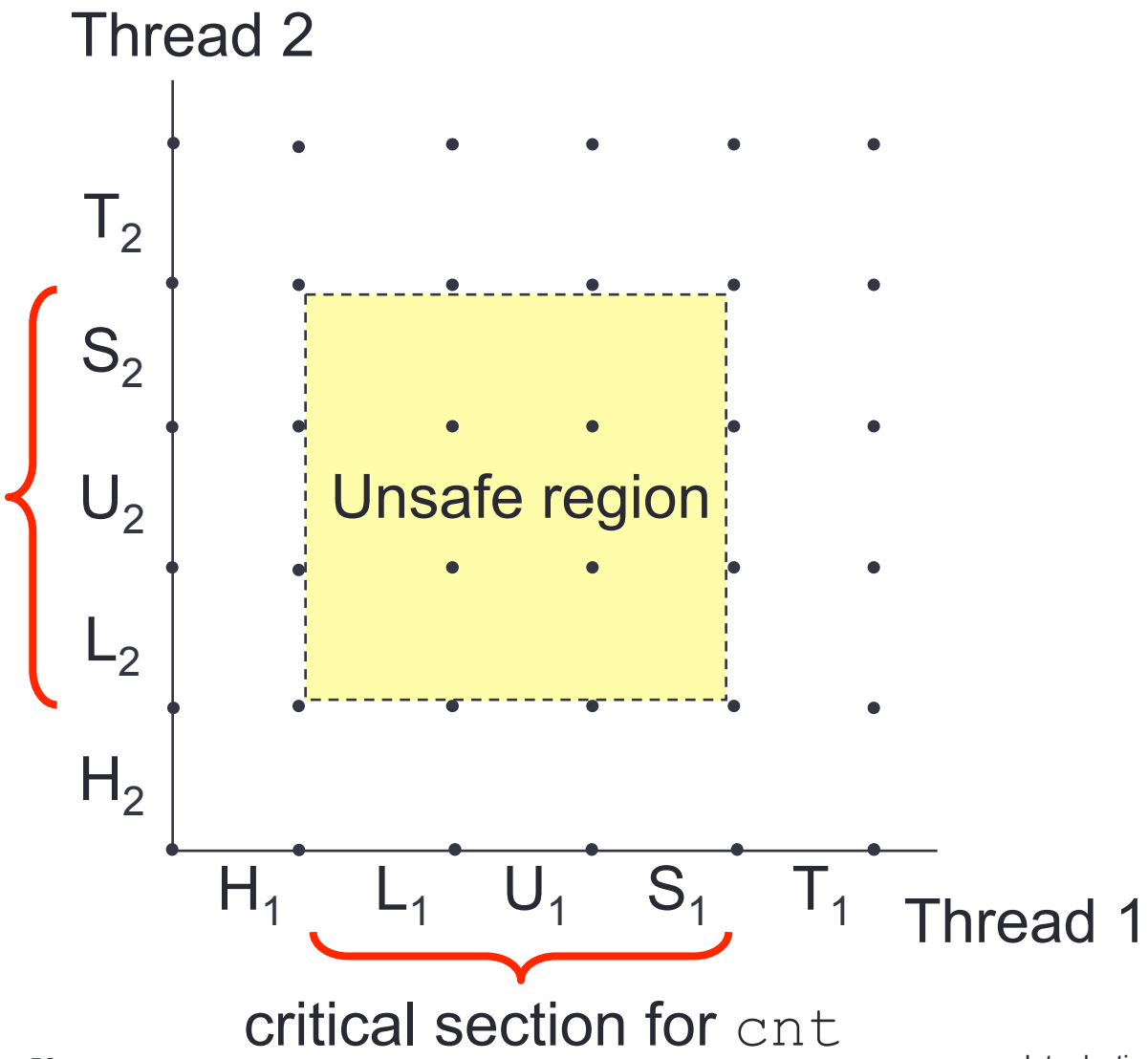


Trajectory is sequence of legal state transitions that describes one possible concurrent execution of the threads

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical Sections and Unsafe Regions

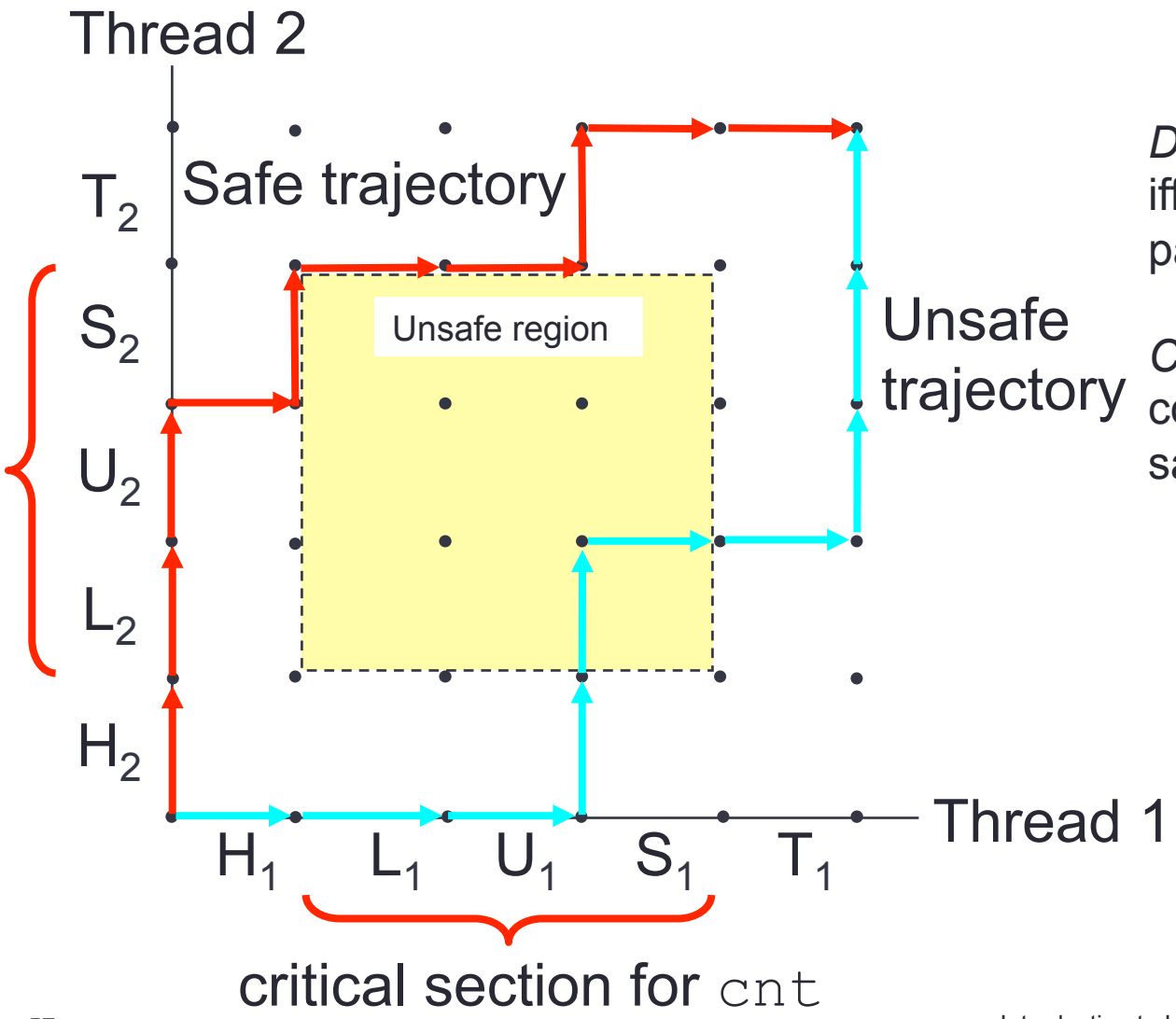


L, U, and S form a *critical section* with respect to the shared variable `cnt`

Instructions in critical sections (w.r.t. to some shared variable) should not be interleaved

Sets of states where such interleaving occurs form *unsafe regions*

Safe and Unsafe Trajectories



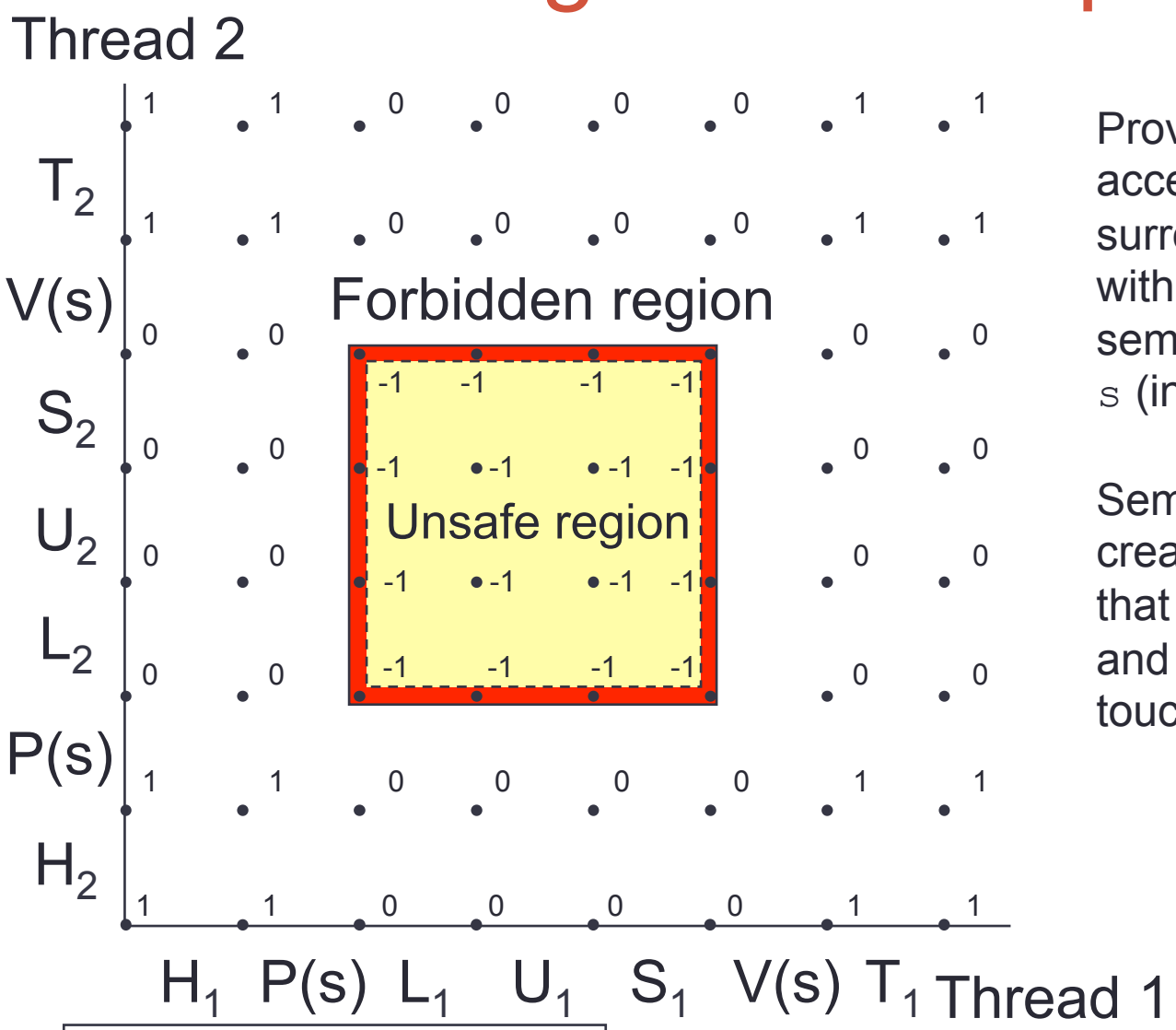
Def: A trajectory is *safe* iff it doesn't enter any part of an unsafe region

Claim: A trajectory is correct (wrt `cnt`) iff it is safe

Semaphores

- *Question:* How can we guarantee a safe trajectory?
  - *Synchronize* threads so they never enter unsafe state
- *Classic solution*: Dijkstra's P/V operations on *semaphores*
  - *Semaphore:* non-negative integer synchronization variable
    - P(s): `while(1){[ if (s>0) {s--; break}] wait_a_while()}`
      - Dutch for “test” ("Proberen“)
    - V(s): [`s++;`]
      - Dutch for “increment” ("Verhogen")
  - OS guarantees that operations between brackets [ ] are executed indivisibly
    - Only one P or V operation at a time can modify s
    - When `while` loop in process X terminates, only that process has decremented s
- Semaphore invariant: *(s >= 0)*

Safe Sharing With Semaphores

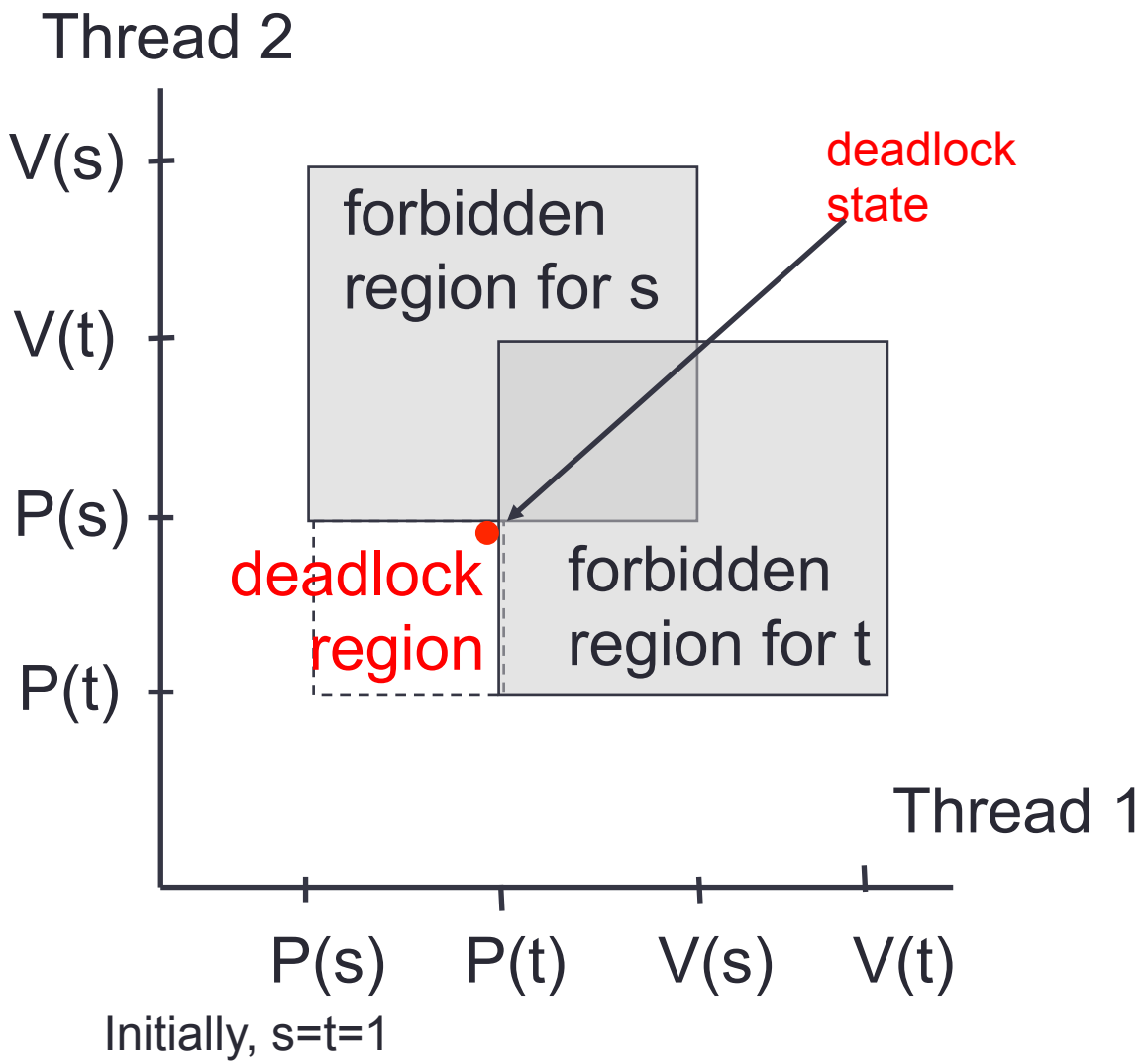


Initially s = 1

Provide mutually exclusive access to shared variable by surrounding critical section with P and V operations on semaphore s (initially set to 1)

Semaphore invariant creates *forbidden region* that encloses unsafe region and is never touched by any trajectory

Deadlock



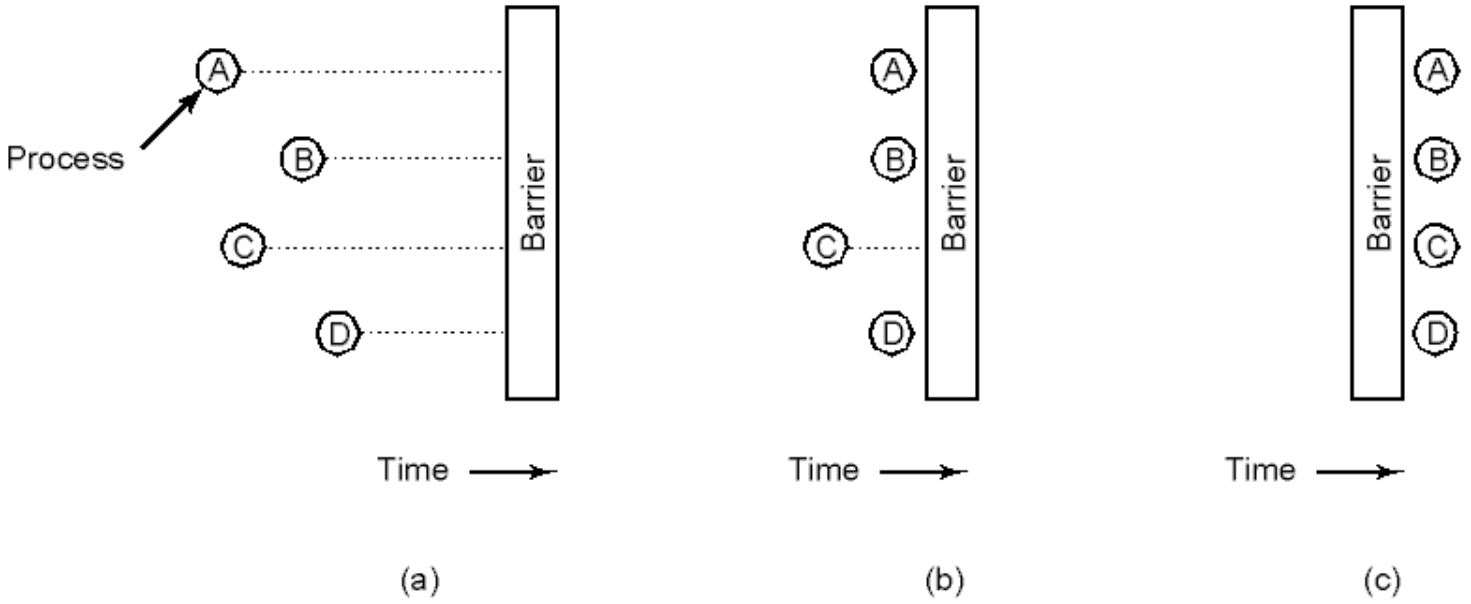
Locking introduces potential for *deadlock:* waiting for a condition that will never be true.

Any trajectory that enters *deadlock region* will eventually reach *deadlock state*, waiting for either `s` or `t` to become nonzero.

Other trajectories luck out and skirt deadlock region.

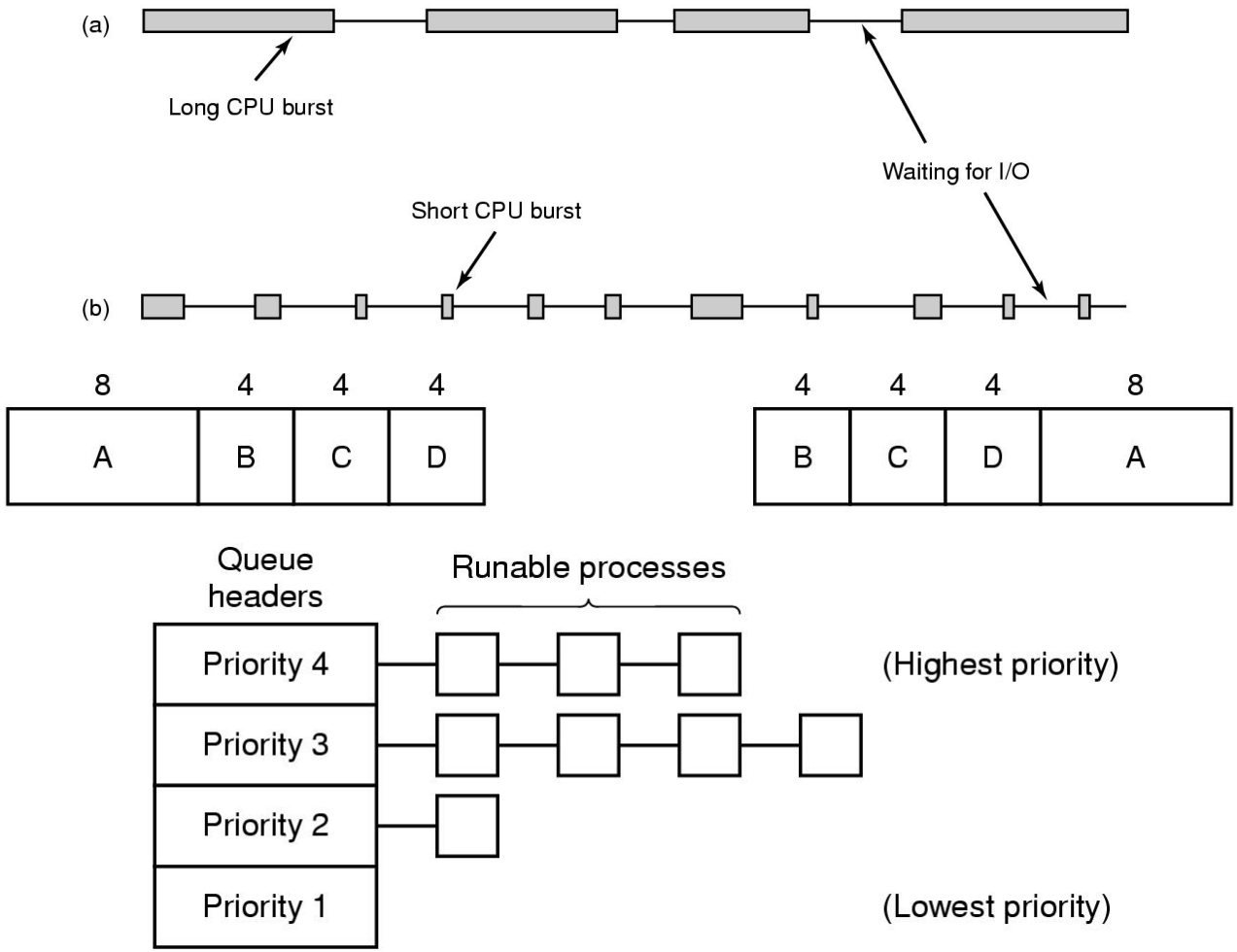
Unfortunate fact: deadlock is often non-deterministic (thus hard to detect).

Threads Model : Barriers (join)



- Use of a barrier
  - (a) processes approaching a barrier
  - (b) all processes but one blocked at barrier
  - (c) last process arrives, all are let through

Threads Model : Scheduling



Threads Model Implementations

- Threads implementations commonly comprise:
  - Library of subroutines called within parallel source code
  - Set of compiler directives embedded in serial or parallel code
- In both cases, programmer is responsible for determining all parallelism.
- Threaded implementations are not new in computing.
- Standardization efforts resulted in three very different implementations of threads:
  - ***POSIX Threads***
  - ***OpenMP***.
  - ***Fair Threads***
- **POSIX Threads**
  - **Pre-emptive thread model**
  - Library based; requires parallel coding (referred to as Pthreads).
  - Very explicit parallelism;
  - Significant attention to detail.
- **OpenMP**
  - Extremely simple to code through **set of pragmas**
  - … However no fine-grain control over implementation
- **Fair Threads**
  - Enables **fairness** and **thread scheduling**
  - Opposed to the **pre-emptive model**
  - Fully determinist, no concurrency
  - Threads must **explicitly cooperate**

Threads Model: POSIX Threads

- Standard interface ~60 functions for threads in C programs
  - **Creating and reaping threads**
    - `pthread_create, pthread_join`
  - **Determining your thread ID**
    - `pthread_self`
  - Terminating threads
    - `pthread_cancel, pthread_exit`
    - `exit` [terminates all threads] , `return` [terminates current thread]
  - **Synchronizing access to shared variables**
    - `pthread_mutex_init, pthread_mutex_[un]lock`
    - `pthread_cond_init, pthread_cond_[timed]wait`

Threads Model: OpenMP

- **OpenMP**
  - Compiler directive based; can use serial code
  - Portable / multi-platform, C, C++, etc…
  - Set of **pragmas functions** easy to add to existing code
  - **So easy and simple to use** - provides for "incremental parallelism"
  - … Almost cheating ☺

Threads Model: OpenMP

- **OpenMP**
  - Compiler directive based; can use serial code
  - Portable / multi-platform, C, C++, etc…
  - Set of **pragmas functions** easy to add to existing code
  - **So easy and simple to use** - provides for "incremental parallelism"
  - … Almost cheating ☺

```
for (i=0; i<max; i++)
    zero[i] = 0;
```

Threads Model: OpenMP

- **OpenMP**
  - Compiler directive based; can use serial code
  - Portable / multi-platform, C, C++, etc…
  - Set of **pragmas functions** easy to add to existing code
  - **So easy and simple to use** - provides for "incremental parallelism"
  - … Almost cheating ☺

```
for (i=0; i<max; i++)
    zero[i] = 0;
```

```
#pragma omp parallel for
for (i=0; i<max; i++)
    zero[i] = 0;
```

Threads Model: OpenMP

- **OpenMP**
  - Compiler directive based; can use serial code
  - Portable / multi-platform, C, C++, etc…
  - Set of **pragmas functions** easy to add to existing code
  - **So easy and simple to use** - provides for "incremental parallelism"
  - … Almost cheating ☺

```
for (i=0; i<max; i++)
    zero[i] = 0;
```

```
#pragma omp parallel for
for (i=0; i<max; i++)
    zero[i] = 0;
```

How many threads will OpenMP create ?

- Defined by OMP_NUM_THREADS environment variable
- Set this variable to the maximum number of threads you want

Threads Model: OpenMP

- Make some **variables private to each thread** (local copy)
  - **#pragma omp parallel for private(j)**
- Decide (slightly) how to handle parallelism
  - **schedule(type [,chunk])**
- Make a **whole block of code parallel**
  - **#pragma omp parallel { … }**
- **Forbid thread switching** (protect memory coherence)
  - **#pragma omp atomic**
- **#pragma omp for [clause …]**
- **#pragma omp section [clause …]**
- **#pragma omp single [clause …]**
- **Clause : private(var1, var2)**
- **Clause : shared(var1, var2)**
- **Clause : public(var1, var2)**

Threads Model: Fair Threads

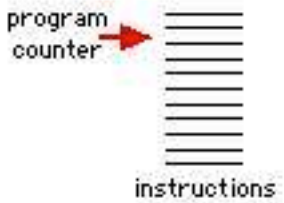


Figure 1: A Thread

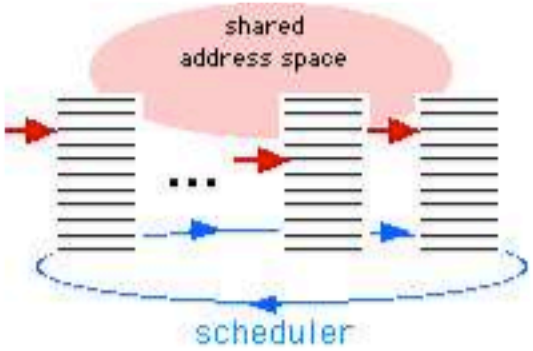


Figure 2: A Scheduler

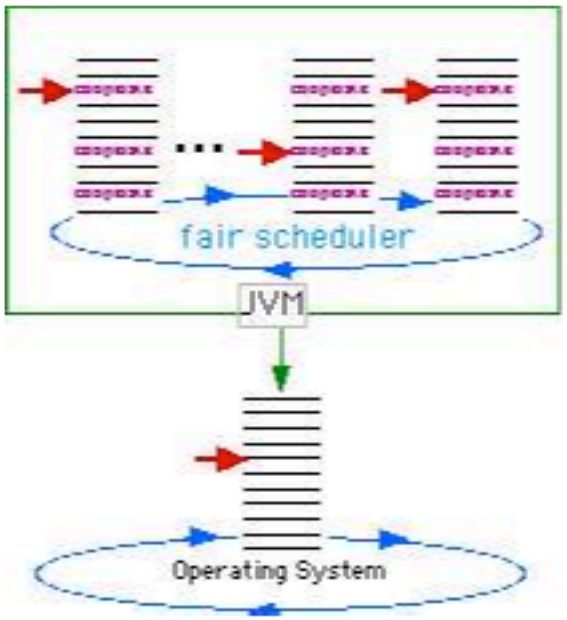


Figure 7: Implementation

- Explicit cooperation required by threads
- No preemption made by fair scheduler
- Completely determinist

- **Fair threads** library available in C, C++, Java, Fortran, …
- Allows extremely **simple implementation of scheduling**

Message Passing Model

- Message passing has the following characteristics:
  - Set of tasks that use their own local memory during computation.
  - Tasks on same machine or across arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process.
  - For example, a send operation must have a matching receive operation.

Exact complementary model to threads

- **Communication is *explicit***
- **Synchronization is *implicit*** (through communication)

Message passing producer / consumer

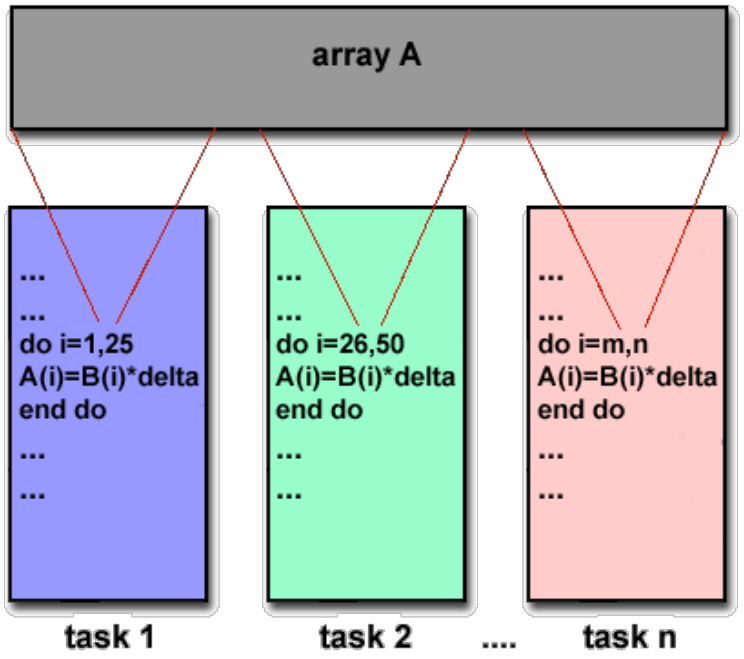
```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Implementations: MPI

- Programming perspective
  - Library of subroutines embedded in source code.
  - Programmer is responsible for determining all parallelism.
- Variety of message passing libraries available since the 1980s.
- Implementations differed substantially from each other
- In 1992, MPI Forum formed with the primary goal of establishing a standard interface for message passing implementations.
- **Message Passing Interface (MPI)** released in 1994 (p2 in 1996)
- Both MPI specifications are available on the web at www.mcs.anl.gov/Projects/mpi/standard.html.

Message Passing Model : MPI

- MPI = industry standard for message passing
- All parallel computing platforms offer at least one implementation.
- On every HPC cluster around the world !
- In shared memory architectures : no network for task communications.



Message Passing Model : MPI

In C:

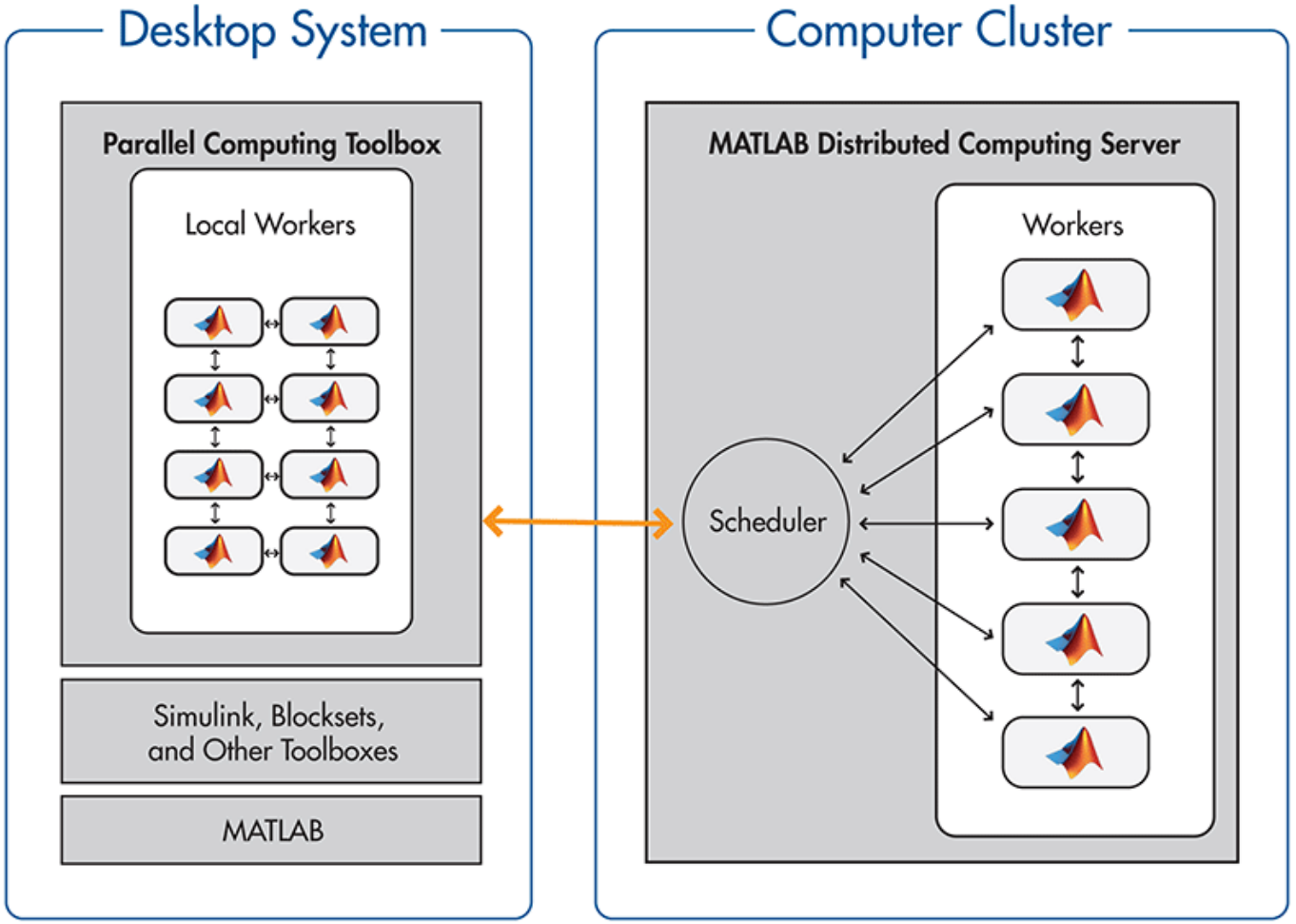
```
#include "mpi.h"
#include <stdio.h>

int main(int argc,
         char *argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello, world!\n");
  MPI_Finalize();
  return 0;
}
```

- Two important questions in parallel program
  - How many processes are participating ?
  - Which one am I ?
- MPI provides functions to answer these
  - **`MPI_Comm_size`** : number of processes.
  - **`MPI_Comm_rank`** : *rank* (0…size-1), identifying the calling process

MPI_SEND (start, count, datatype, dest, tag, comm)

MPI_RECV(start, count, datatype, source, tag, comm, status)

Matlab Parallel Toolbox



Matlab Parallel Toolbox

matlabpool	Open,close pool of MATLAB sessions for parallel computation
parfor	Execute code loop in parallel
spmd	Execute code in parallel on MATLAB pool
batch	Run MATLAB script as batch job

- **Parallel for-Loops (parfor)**

```
parfor (i = 1 : n)
        % do something with i
end
```

- Mix parallel and serial code in same function
- Run loops on pool of MATLAB resources
- Iterations must be order-independent

(**cf. data dependence**)

- **Single Program Multiple Data (spmd)**

```
spmd (n)
        <statements>
end
```

For example, create a random matrix on four labs:

```
matlabpool open
spmd (2)
   R = rand(4,4);
end
matlabpool close
```

Matlab Parallel Toolbox

- **Job Creation**

createJob	Create job object in scheduler and client
createTask	Create new task in job
dfeval	Evaluate function using cluster

- **Interlab Communication Within a Parallel Job**

labBarrier	Block execution until all labs reach this call
labBroadcast	Send data to all labs or receive data sent to all labs
labindex	Index of this lab
labReceive	Receive data from another lab
labSend	Send data to another lab
numlabs	Total number of labs operating in parallel on current job

- **Job Management**

cancel	Cancel job or task
destroy	Remove job or task object from parent and memory
getAllOutputArguments	Output arguments from evaluation of all tasks in job
submit	Queue job in scheduler
wait	Wait for job to finish or change states

High-level synchronous languages

- Up to now we have seen **parallel extensions** to languages
- … But some are developed especially for parallel tasks !
- Usually implies extremely simple sequential / parallel instructions
- Sequential instructions = **A ; B**
- Parallel execution = **A || B**
- Allow to write parallel code in seconds :

```
separateData;
        [performMethodA
        || performMethodB; postProcessB
        || preProcessC; performC]
```

Concurrent Constraint Programming (CCP)

- Actor statements

```
a ::= new x in a
      tell c
      if c1→a1 [] c2→a2
      a1 | a2
      a1 + a2
      p(x)
```

- Procedure definitions

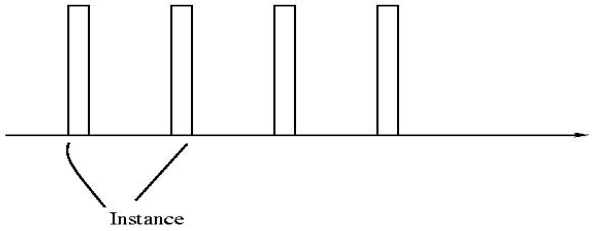
```
p(x) :- a
```

Esterel

- Extremely simple to define automaton behavior

```
loop
   abort
      await SET(t);
      trap T in
         loop
            [ if zorn(t) then exit T
              else nothing
            ||
              await SECOND;
              call dec(t);
            ]
         end
      end;
      emit ALARM;
   when RESET;
end
end module.
```

Execution is series of **reaction**

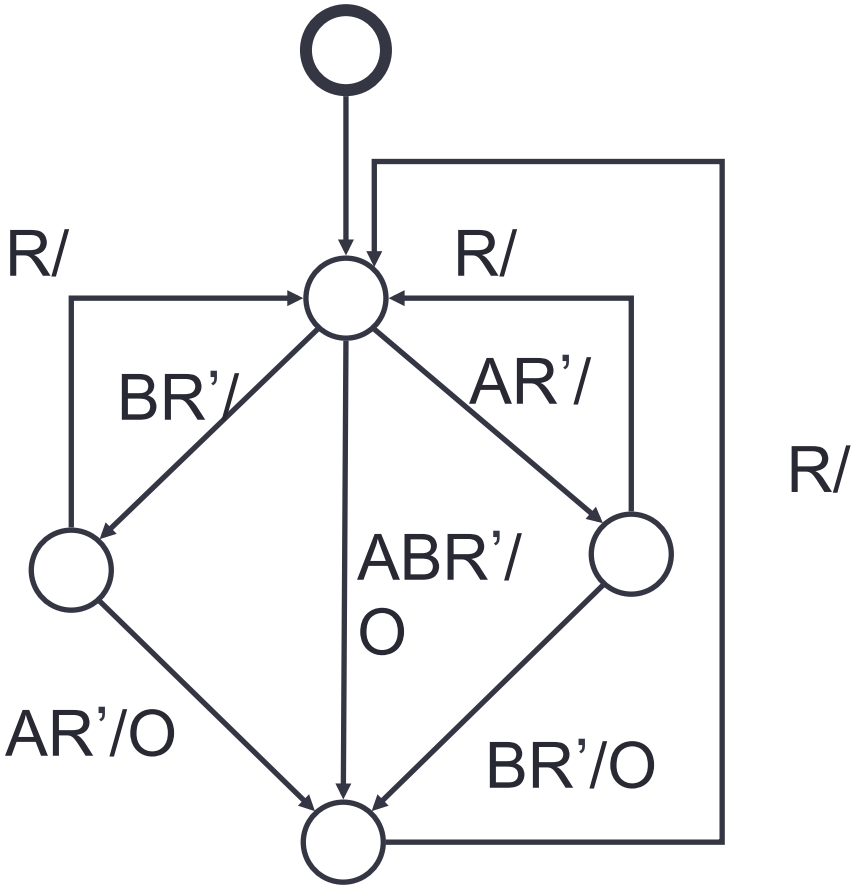


Synchronous parallelism

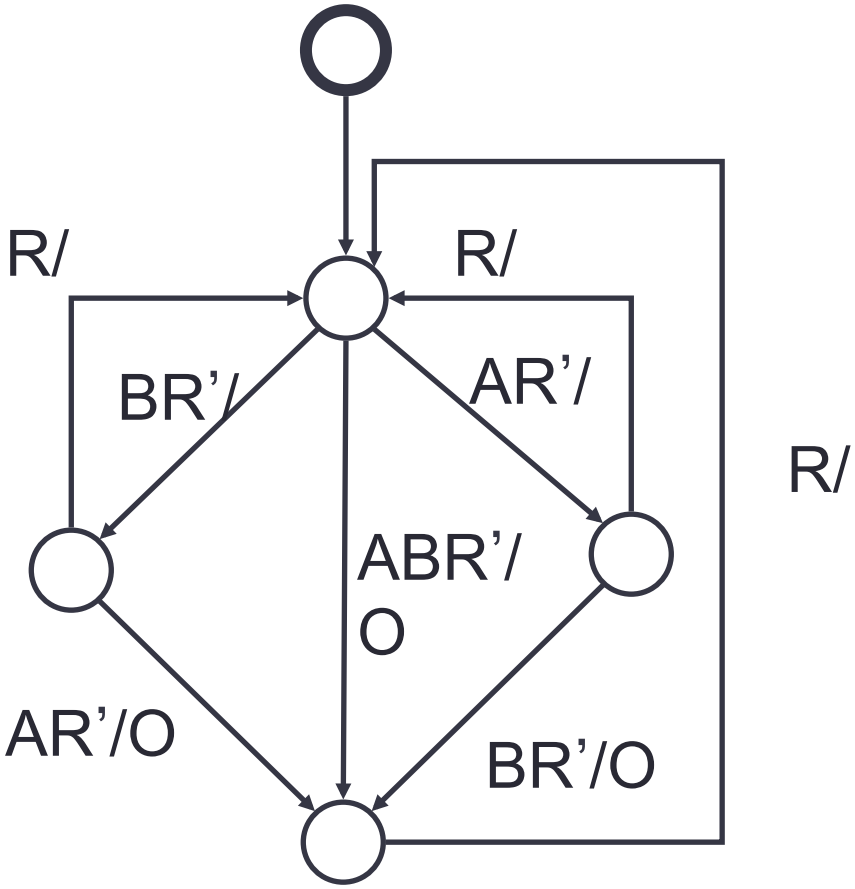
[stat1 || stat2 || stat3]

Fundamentally a **reactive programming** approach

Esterel



Esterel




```
module ABRO
input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O
each R

end module
```

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming

Automatic vs. manual

- Developing parallel programs is usually manual process.
- Programmer must first identify then actually implement parallelism.
- Manually developing parallel codes is a time consuming, complex, error-prone and ***iterative*** process.
- Various tools have been available to assist with converting serial programs into parallel programs.
- Most common tool is a parallelizing compiler or pre-processor.
  - **Matlab parallel toolbox**
  - **OpenMP pragmas**

Automatic vs. manual

- A parallelizing compiler generally works in two different ways:
  - Fully Automatic
    - Compiler analyzes source code and identifies opportunities for parallelism.
    - Analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
    - Loops (do, for) loops are the most frequent target for automatic parallelization.
  - Programmer Directed
    - “Compiler directives" or flags to explicitly tell the compiler how to parallelize code.
    - Used in conjunction with some degree of automatic parallelization.
- Several flaws of automatic parallelism :
  - Wrong results may be produced
  - Performance may actually degrade
  - Much less flexible than manual parallelization
  - Limited to a subset (mostly loops) of code
  - May actually not parallelize code if the analysis suggests there are inhibitors or the code is too complex

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

- First step in developing parallel software is to understand the problem that you wish to solve in parallel.
- Before spending time, determine whether or not the problem is one that can actually be parallelized.
- Identifying **hotspots** (parallel sections)
- Identifying bottlenecks (**inhibitors**)

Example of Parallelizable Problem

Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.

This problem can be solved in parallel

Each molecular conformation is determined independently

Even called an **embarassingly parallel situation !**

Example of a Non-parallelizable Problem

Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula:

```
F(k + 2) = F(k + 1) + F(k)
```

- Non-parallelizable problem
- Calculation of the Fibonacci sequence entail dependent calculations rather than independent ones.
- Calculation of the k + 2 value uses both k + 1 and k.
- Terms cannot be calculated independently and

Identify the program's *hotspots*

- Where most of the real work is being done ?
- Majority of scientific programs accomplish most of their work in a few places.
- Profilers and performance analysis tools can help here
- Focus on parallelizing the hotspots
- Ignore program sections that account for little CPU usage.

Identify ***bottlenecks*** in the program

- Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred ?
- For example, I/O slows a program down.
- In these cases :
  - Restructure the program
  - Use a different algorithm
  - Reduce or eliminate unnecessary slow areas

Other considerations

- Identify inhibitors to parallelism.
- One common class of inhibitor is *data dependence* (cf. Fibonacci sequence).
- Investigate other algorithms if possible.
- This may be the single most important consideration when designing a parallel application.

Agenda

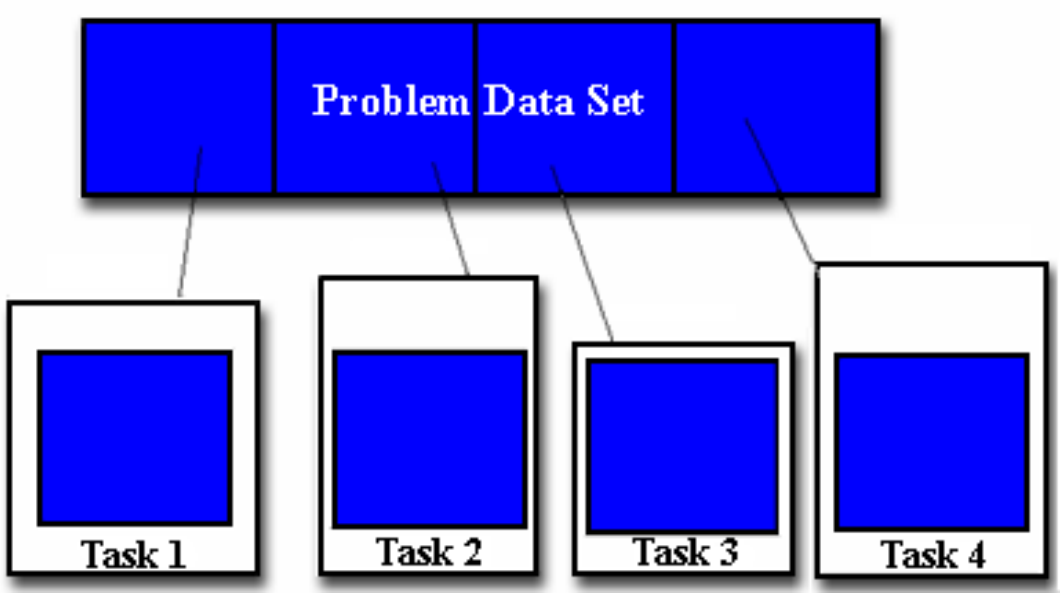
- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Partitionning

- First steps in designing a parallel program is to break the problem into discrete "chunks" of work
- These chunks can be distributed to multiple tasks.
- This is known as *decomposition* or *partitioning*.
- Two basic ways to partition computational work
  - ***Domain decomposition***
  - ***Functional decomposition***

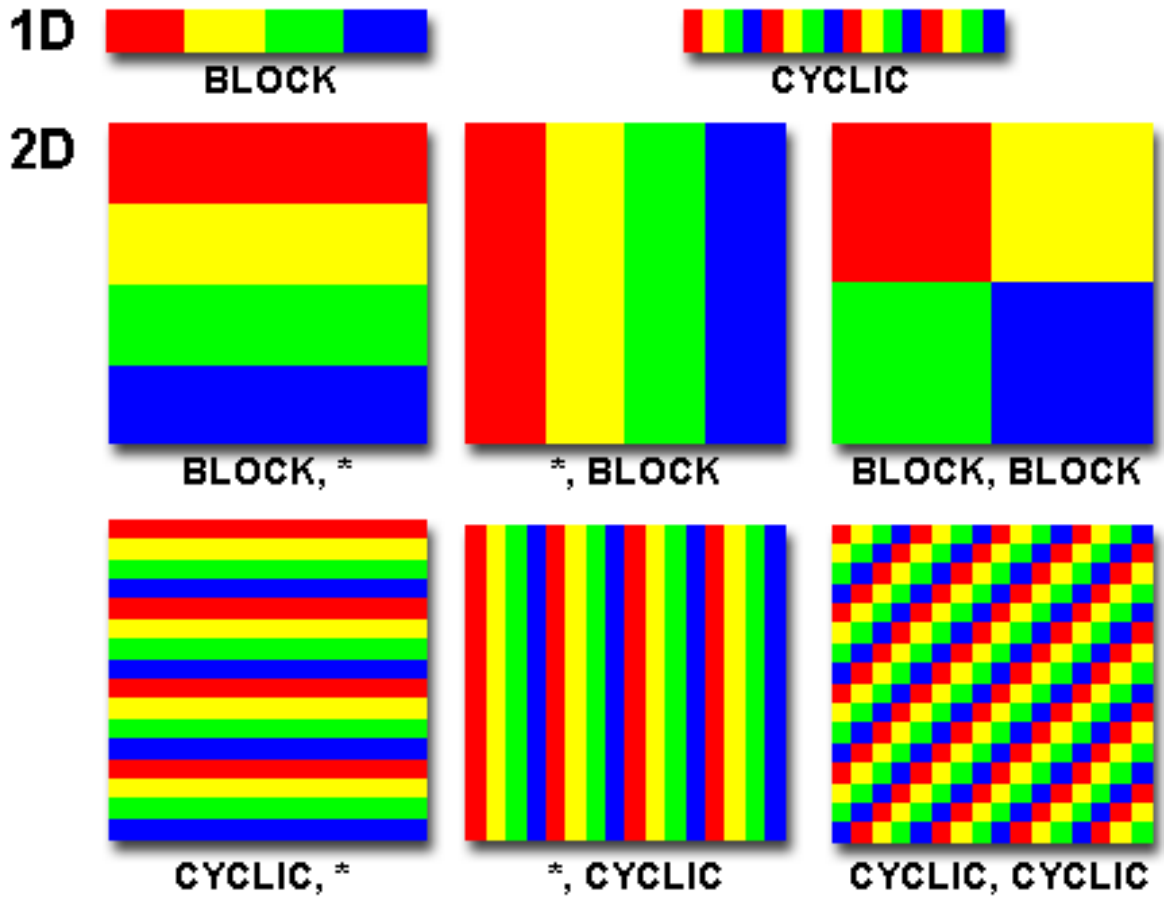
Domain Decomposition

- Data associated with a problem is decomposed.
- Each parallel task then works on a portion of the data.



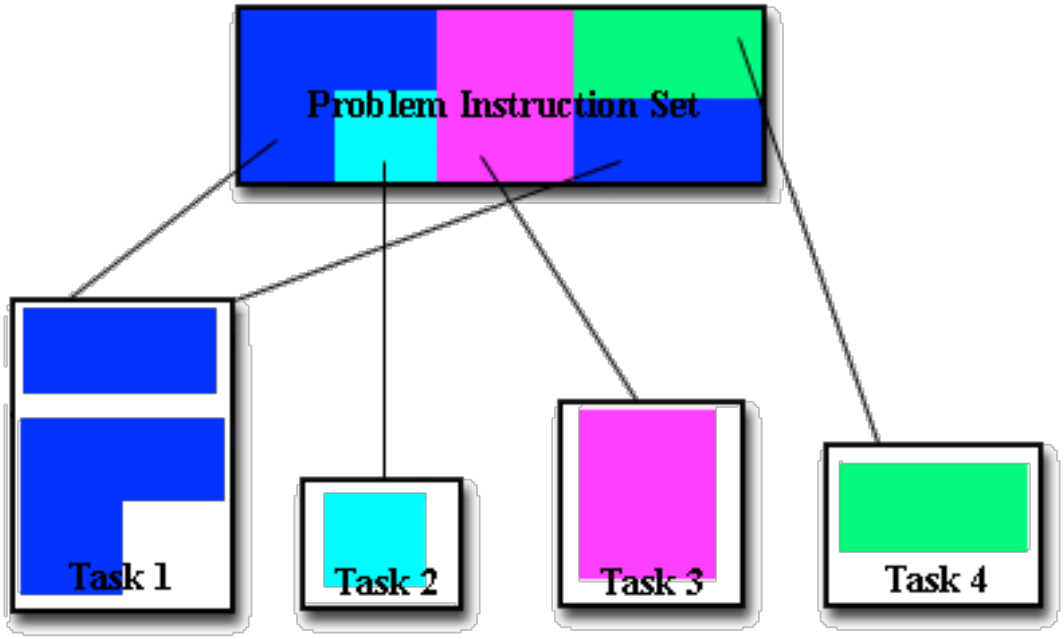
Partitioning Data

- Different ways to partition data



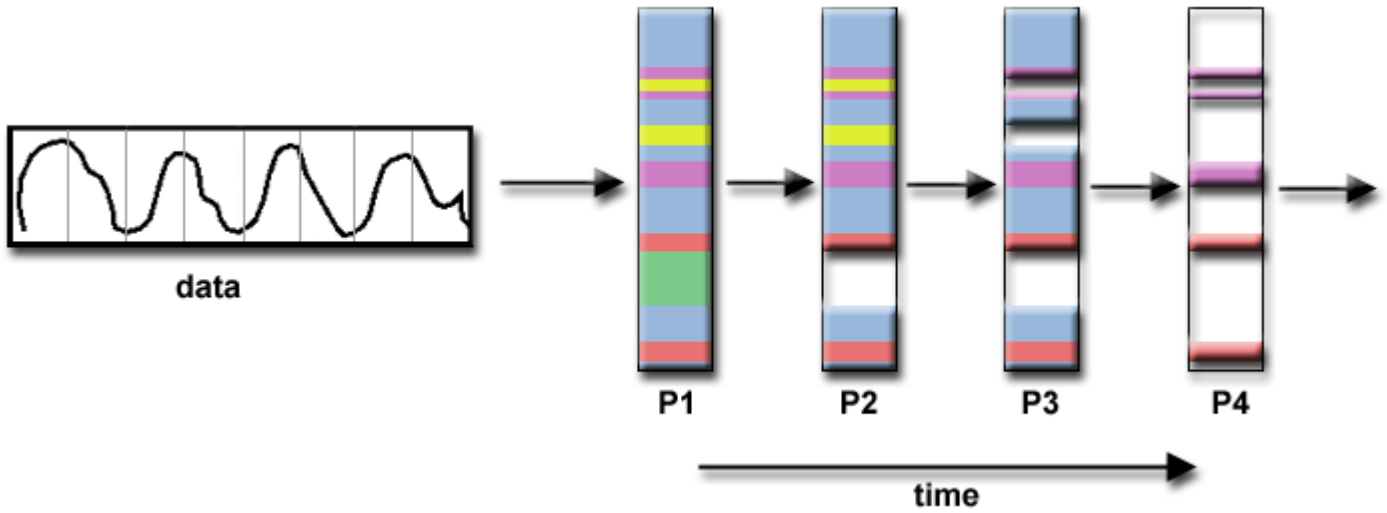
Functional Decomposition

- Here focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- Problem is decomposed according to the work that must be done.
- Each task then performs a portion of the overall work.
- Functional decomposition lends itself well to problems that can be split into different tasks (eg. **Signal processing** !)



Signal Processing

- Audio signal data set passed through four distinct filters.
- Each filter is a separate process.
- First segment of data must pass through first filter before second.
- When it does, second segment of data passes through the first filter.
- By the time the fourth segment of data is in the first filter, all four tasks are busy.



Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Who Needs Communications?

- The need for communications between tasks depends upon your problem
- **You DON'T need communications**
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data.
  - For example, image processing operation where every pixel in a black and white image needs to have its color reversed.
  - These types are called ***embarrassingly parallel*** because they are so straight-forward. Very little inter-task communication is required.
- **You DO need communications**
  - Most parallel applications are not quite so simple,
  - Do require tasks to share data with each other.
  - For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.

Factors to Consider

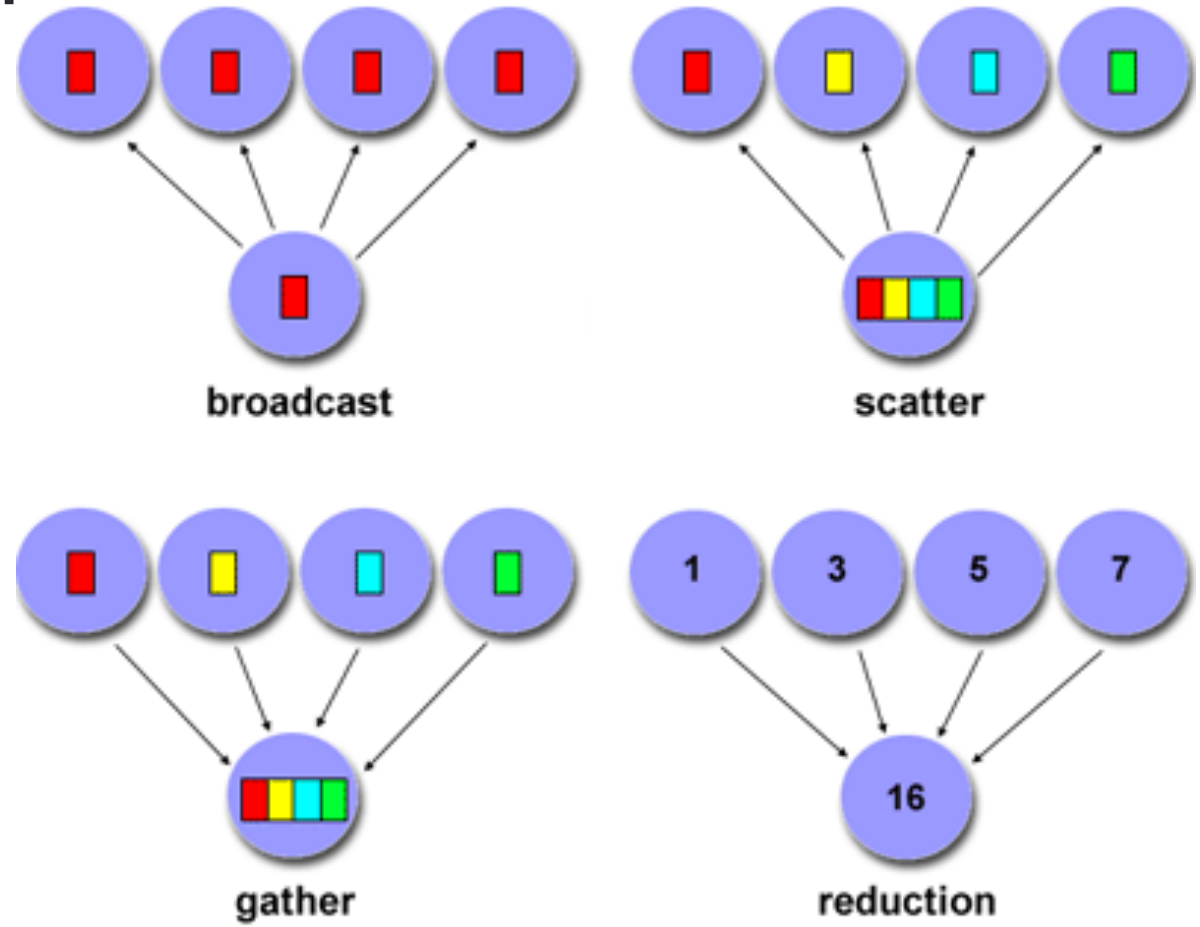
- Important factors to consider in inter-task communications
- **Cost of communications**
  - Inter-task communication always implies overhead.
  - Resources could be computation but instead transmit data.
  - Communications frequently require some type of synchronization between tasks, (can result in tasks spending time "waiting”).
  - Communication can saturate the available network bandwidth.
- **Latency vs. Bandwidth**
  - ***latency*** is the time it takes to send a minimal (0 byte) message.
  - ***Bandwidth***: amount of data that can be communicated per time
  - Sending many small messages can cause latency to dominate communication overheads.

Factors to Consider

- **Visibility of communications**
  - With the Message Passing Model, communications are explicit.
  - With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory
- **Synchronous vs. asynchronous communications**
  - Synchronous communications require connection between tasks.
  - Synchronous communications are often referred to as ***blocking***.
  - Asynchronous communications allow tasks to transfer data independently.
  - Asynchronous communications are often referred to as ***non-blocking***.
- **Scope of communications**
  - Knowing which tasks must communicate with each other is critical during the design stage of a parallel code.
  - ***Point-to-point*** - one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
  - ***Collective*** - data sharing between more than two tasks, which are often specified as being members in a common group, or collective.

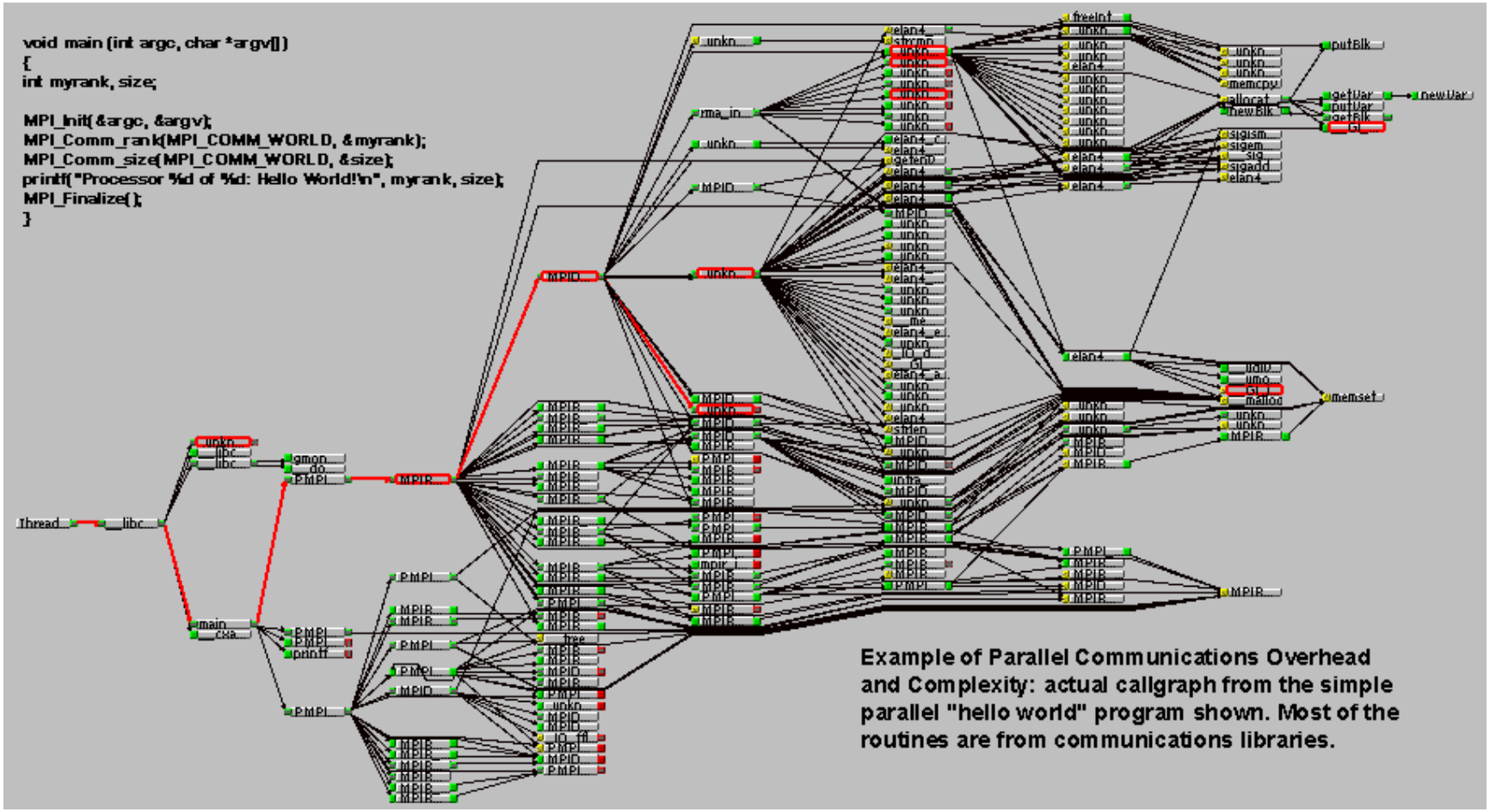
Collective Communications

- **Examples**



Factors to Consider

- **Overhead and Complexity**



Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Types of Synchronization

- **Barrier**
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized.
  - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.
- **Lock / semaphore**
  - Can involve any number of tasks
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking
- **Synchronous communication operations**
  - Involves only those tasks executing a communication operation
  - When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.
  - Discussed previously in the Communications section.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Definitions

- ***Dependence*** = order of statement execution affects the results of the program.
- ***Data dependence*** = multiple use of the same location(s) in storage by different tasks.
- Important to parallel programming because they are one of the primary inhibitors to parallelism.

Loop-carried data dependence

```
for J = 1:10000
      A(J) = A(J-1) * 2.0500
end
```

- Value of A(J-1) must be computed before A(J),
- A(J) exhibits a data dependency on A(J-1).
- Parallelism is inhibited.

Loop independent data dependence

```
task 1          task 2
------          ------
X = 2           X = 4
.               .
.               .
Y = X**2        Y = X**3
```

- Parallelism is inhibited. The value of Y is dependent on:
  - Distributed memory architecture –X is communicated between tasks.
  - Shared memory architecture - which task last stores the value of X.
- All data dependencies are important to identify
- Loop carried dependencies are particularly important
- Loops are the most common target of parallelization efforts.

How to Handle Data Dependencies?

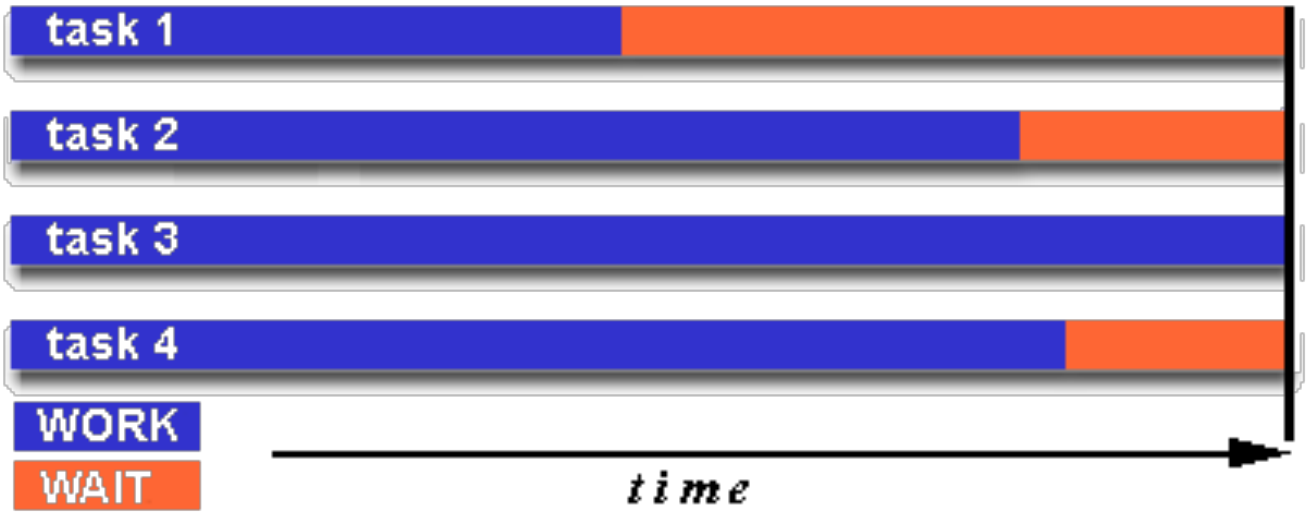
- Distributed memory architectures
- Communicate required data at synchronization points.
- Shared memory architectures
- Synchronize read/write operations between tasks.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Load balancing

- Distributing work among tasks so that ***all*** are kept busy ***all*** of the time.
- Minimization of task idle time.
- Load balancing is important to parallel programs for performance
- If all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



How to Achieve Load Balance ?

- **Equally partition the work each task receives**
  - For array/matrix operations, evenly distribute data set among tasks.
  - For loop iterations, evenly distribute the iterations across the tasks.
  - If heterogeneous mix of machines with varying performance, use analysis tool to detect load imbalances.
- **Use dynamic work assignment**
  - Certain problems result in load imbalances even if data is evenly distributed
    - Sparse arrays - some tasks will have actual data to work on while others have "zeros".
    - Adaptive grid methods - some tasks may need to refine their mesh while others don't.
  - When amount of work is intentionally variable, helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get assigned.
  - Become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

Agenda

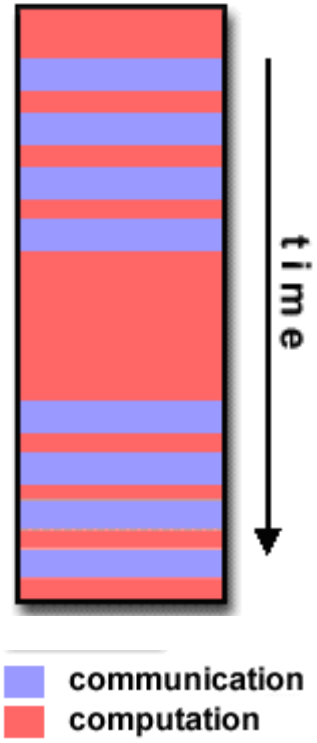
- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Definitions

- Computation / Communication Ratio:
  - Granularity: measure of the ratio of computation to communication.
  - Computation typically separated from periods of communication by synchronization events.
- Fine grain parallelism
- Coarse grain parallelism

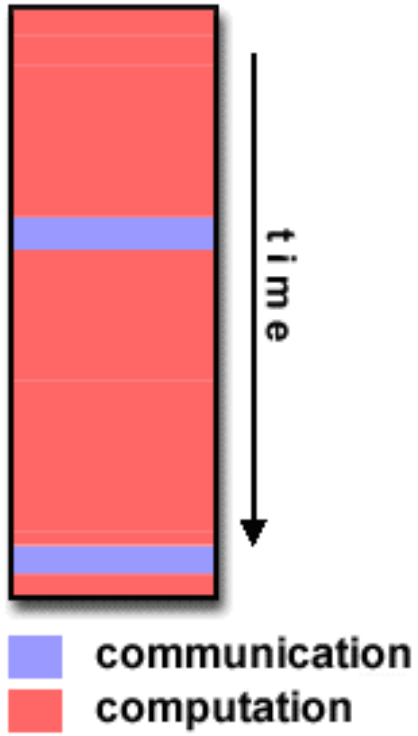
Fine-grain Parallelism

- Small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine then the overhead required for communications and synchronization takes longer than the computation.



Coarse-grain Parallelism

- Large amounts of computational work are done between communication/ synchronization events
- High computation to communication ratio
- More opportunity for performance increase
- Harder to load balance efficiently



Which is Best ?

- Most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
- Coarse granularity reduce the overhead associated with communications and synchronization.
- Fine-grain parallelism can help reduce load imbalance.

Agenda

- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

The bad News

- I/O operations are regarded as inhibitors to parallelism
- Parallel I/O systems are immature …
- In an environment where all tasks see the same filespace, write operations will result in file overwriting
- Read operations will be affected by the fileserver's ability to handle multiple read requests at the same time
- I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks

The good News

- Some parallel file systems are available. For example:
  - GPFS: General Parallel File System for AIX (IBM)
  - Lustre: for Linux clusters (Cluster File Systems, Inc.)
  - PVFS/PVFS2: Parallel Virtual File System for Linux clusters (Clemson/ Argonne/Ohio State/others)
  - PanFS: Panasas ActiveScale File System for Linux clusters (Panasas, Inc.)
  - HP SFS: HP StorageWorks Scalable File Share. Lustre based parallel file system (Global File System for Linux) product from HP
- The parallel I/O programming interface specification for MPI has been available since 1996 as part of MPI-2. Vendor and "free" implementations are now commonly available.

Some Options

- If you have access to a parallel file system, investigate using it …
- Rule #1: Reduce overall I/O as much as possible
- Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks.
  - read all input file and then communicate required data to other tasks.
  - perform write operation after receiving required data from all other tasks.
- For distributed memory systems with shared filespace, perform I/O in local, non-shared filespace.
- Local I/O always more efficient than network I/O
- Create unique filenames for each tasks' input/output file(s)

Agenda

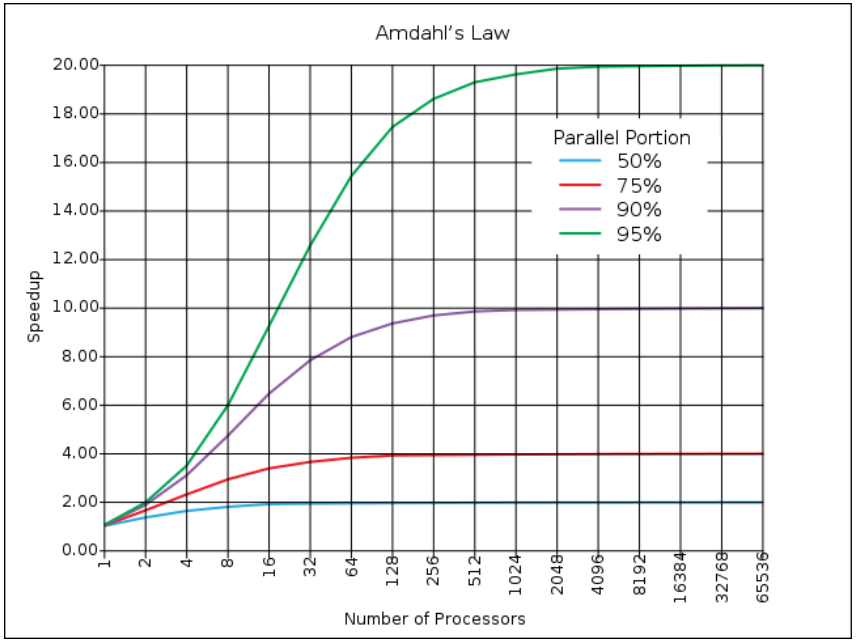
- Automatic vs. Manual Parallelization
- Understand the Problem and the Program
- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O
- Limits and Costs of Parallel Programming
- Performance Analysis and Tuning

Amdahl's Law

- Amdahl's Law : potential speedup is defined by the fraction of code (B) that **is serial** :

$$\phi(n) = \frac{1}{B + \frac{1}{n}(1 - B)}$$

n : Number of processors

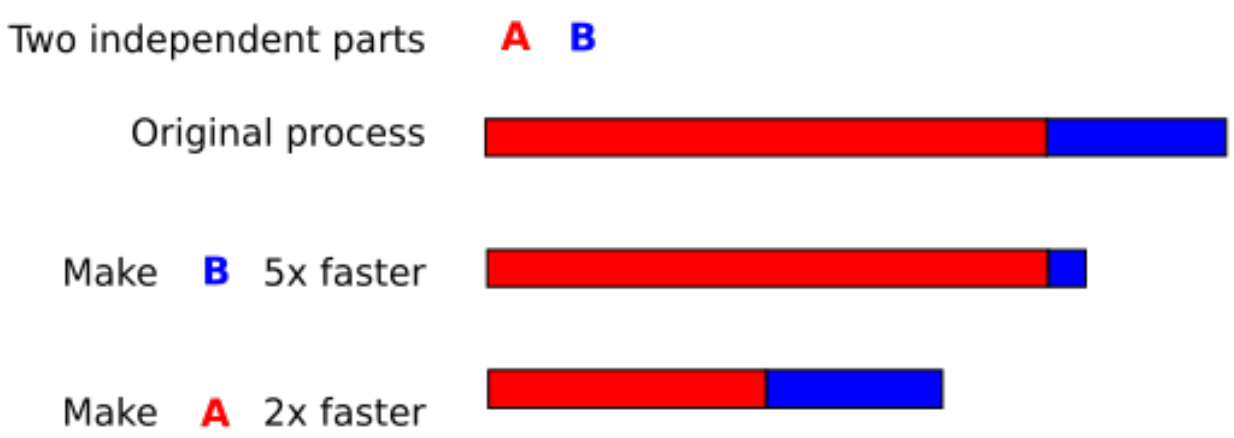


- None parallelized (B = 1) : speedup = 1 (no speedup).
- 50% of the code is parallel, maximum speedup = 2.

Amdahl's Law

- Obvious limits to the scalability of parallelism !

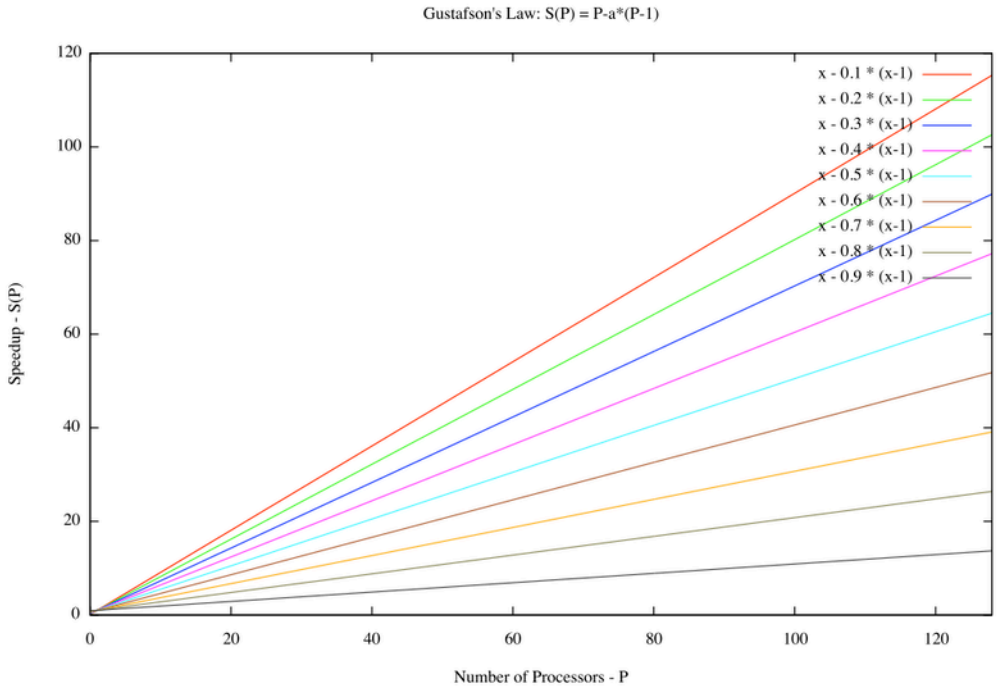
	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02



Gustaffson's Law

- Certain problems violates the Amdahl's Law
- Problems that **increase the percentage of parallel time with their size** are more ***scalable*** than problems with a fixed percentage of parallel time.

$$\phi(n) = n - B(n-1)$$



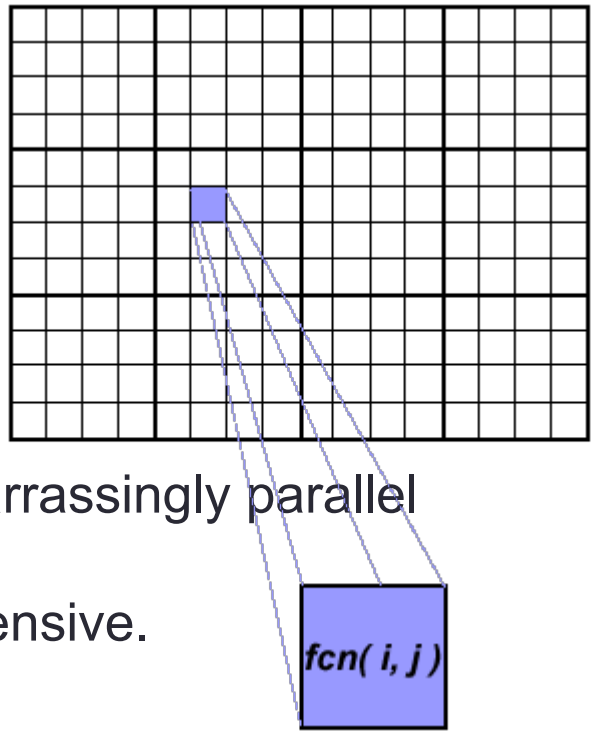
Parallel examples

- **Array Processing**
- **PI Calculation**
- **Simple Heat Equation**
- **1-D Wave Equation**

Array Processing

- Calculations on 2-dimensional array, with computation on each array element being independent from other array elements.
- Serial program calculates one element at a time in sequential.
- Serial code could be of the form:

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



- Computations are independent = embarrassingly parallel situation.
- Problem should be computationally intensive.

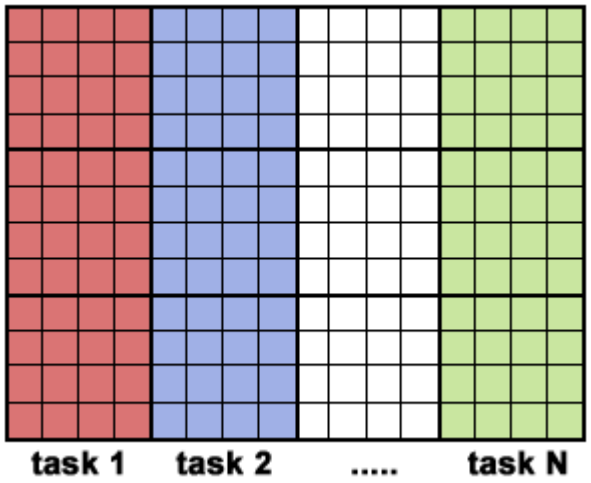
Array Processing Solution

- Arrays elements are distributed, each processor computes subarray.
- Independent calculation insures there is no need for communication.
- Distribution scheme is chosen by other criteria, (unit stride)
- Unit stride maximizes cache/memory usage.
- Choice of the strides depends on programming language.

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

```
parfor j = 1:m
   parfor i = 1,n
      a(i,j) = fcn(i,j)
   end
end
```

In Matlab …
Add 6 chars !



Array Processing Solution 1
One possible implementation

- Implemented as master / worker model.
- **Master process**
  - initializes array
  - sends info to worker processes
  - receives results.
- **Worker process**
  - receives info
  - performs its share of computation
  - Send results to master.

Array Processing Solution 2: Pool of Tasks

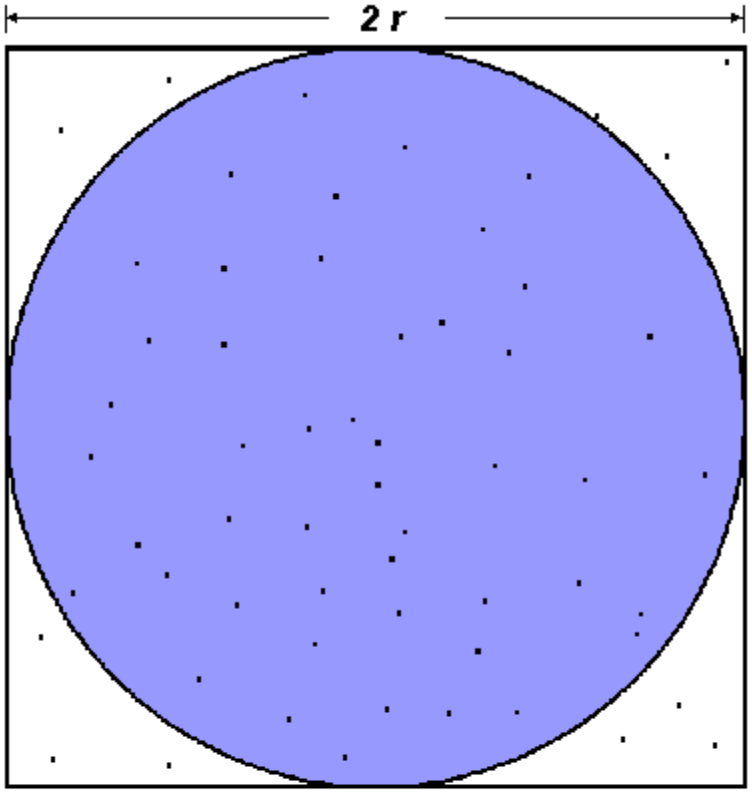
- Previous solution demonstrated static load balancing:
  - Each task has a fixed amount of work to do
  - May be significant idle time for faster or more lightly loaded processors - slowest tasks determines overall performance.
- Static load balancing not usually a major concern if all tasks are performed on identical machines.
- If load balance problem (tasks work faster than others), you may benefit by using a "pool of tasks" scheme.

Pi Calculation

- Value of PI can be calculated in a number of ways.
- Consider the following method of approximating PI
  - Inscribe a circle in a square
  - Randomly generate points in the square
  - Determine the number of points in the circle
  - r = number of points in circle / number of points in the square
  - PI ~ 4 r
- Note that the more points generated, the better the approximation

Algorithm

```
npoints = 10000
circle_count = 0
do j = 1,npoints
  generate 2 random numbers between 0
  and 1
  xcoordinate = random1 ; ycoordinate
  = random2
  if (xcoordinate, ycoordinate) inside
  circle  then circle_count =
  circle_count + 1
end do
PI = 4.0*circle_count/npoints
```



$$A_S = (2r)^2 = 4r^2$$

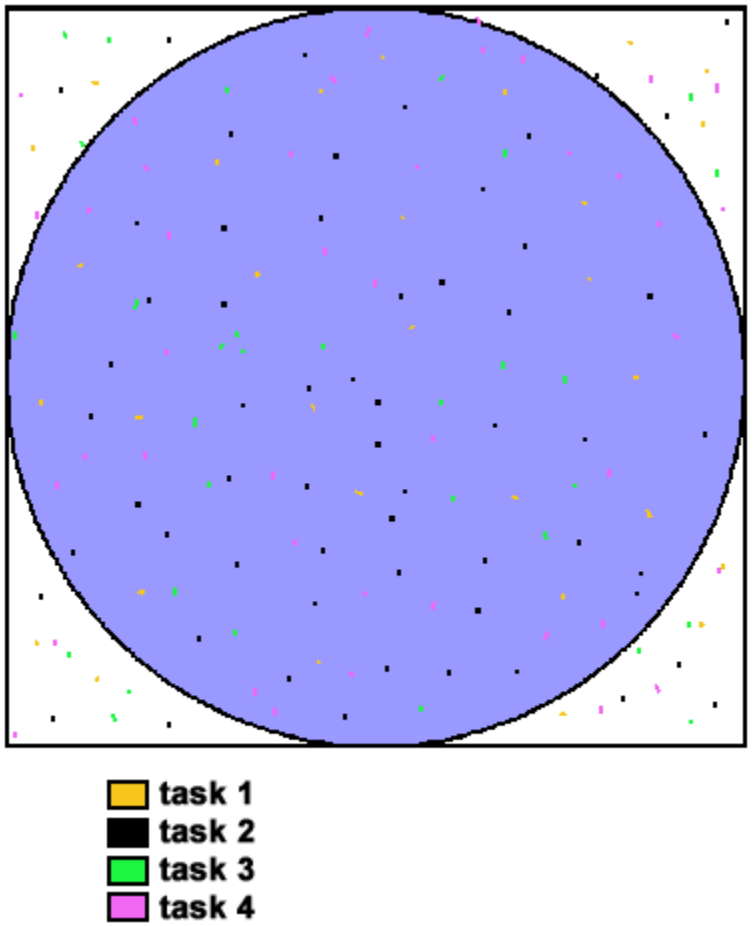
$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

- Note that most of the time in running this program would be spent executing the loop
- Leads to an embarrassingly parallel solution
  - Computationally intensive
  - Minimal communication
    - Minimal I/O

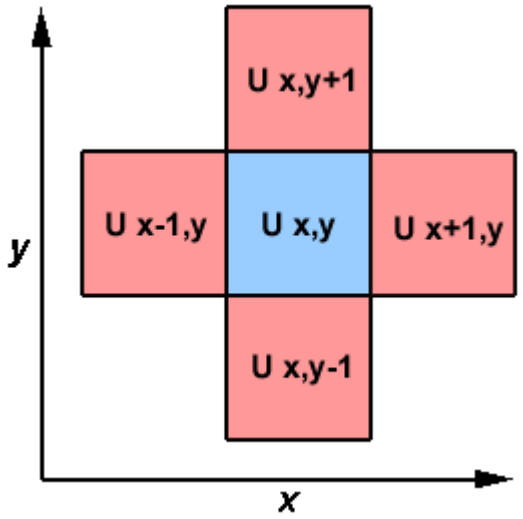
PI Calculation
Parallel Solution

- Parallel strategy: break the loop into portions that are executed by tasks.
- For the task of approximating PI:
  - Each task executes its portion of the loop a number of times.
  - Each task do its work without requiring any information from other tasks.
  - Uses the master/worker model. One task acts as master and collects the results.



Simple Heat Equation

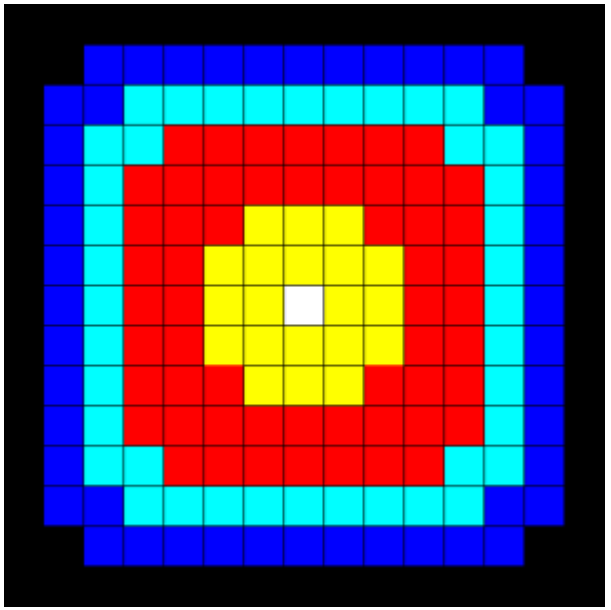
- Most problems require communication among the tasks.
- A number of common problems require communication with "neighbor" tasks.
- Heat equation describes the temperature change over time, given initial temperature distribution and boundary conditions.
- A finite differencing scheme is employed to solve the heat equation on a square region.
- The initial temperature is zero on the boundaries and high in the middle.
- The boundary temperature is held at zero.
- For the fully explicit problem, a time stepping algorithm is used. The elements of a 2-dimensional array represent the temperature at points on the square.



Simple Heat Equation

- The calculation of an element dependent upon neighbor values

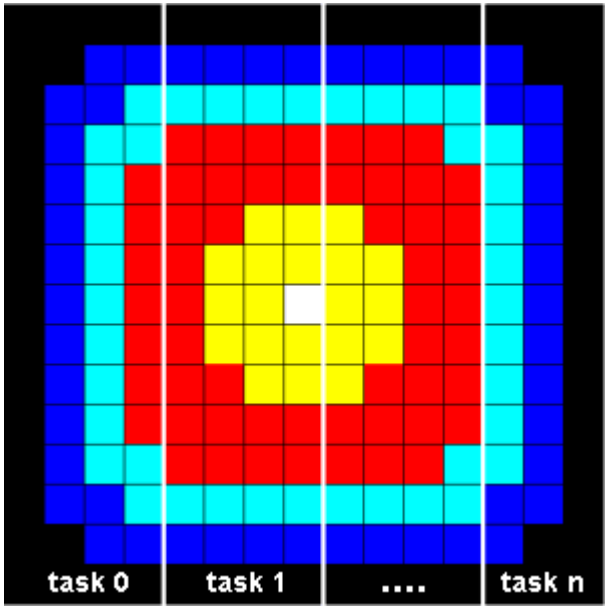
$$U_{x,y} = U_{x,y} + C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy}) + C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$



- A serial program :

```
do iy = 2, ny - 1
do ix = 2, nx - 1
  u2(ix, iy) =
    u1(ix, iy)  +
      cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) +
      cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
end do
end do
```

Parallel Solution 1



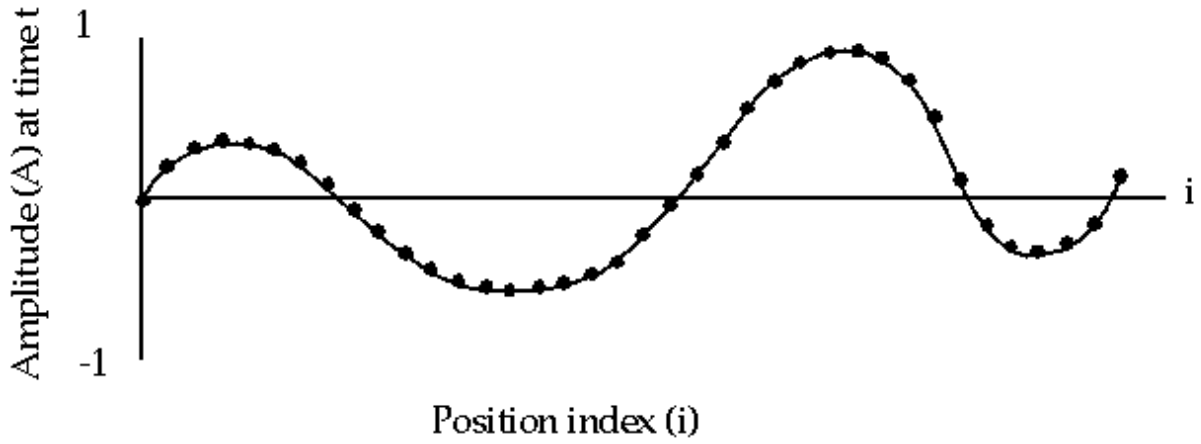
- Implement as an SPMD model
- Array is partitioned and distributed as subarrays to all tasks.
- Each task owns a portion of the total array.
- Determine data dependencies
  - interior elements are independent of other tasks
  - border elements are dependent upon a neighbor task's data.
- Master process sends initial info to workers, checks for convergence and collects results
- Worker process calculates solution, communicating as necessary with neighbor processes
- Pseudo code solution: **red** highlights changes for parallelism.

Parallel Solution 2
Overlapping Communication and Computation

- Blocking communications is assumed for the worker tasks.
- Blocking wait for the communication to complete before continuing
- Neighbor tasks communicated border data, then each process updated its portion of the array.
- Computing times be reduced by using non-blocking communication.
- Non-blocking communications allow work to be performed while communication is in progress.
- Each task could update the interior of its part of the solution array while the communication of border data is occurring, and update its border after communication has completed.

1-D Wave Equation

- Amplitude along a uniform, vibrating string is calculated after a specified amount of time has elapsed.
- Calculation involves:
  - amplitude on the y axis
  - i as the position index along the x axis
  - node points imposed along the string
  - update of the amplitude at discrete time steps.



1-D Wave Equation

- Solve the one-dimensional wave equation:

```
A(i,t+1) = (2.0 * A(i,t)) - A(i,t-1)
     + (c * (A(i-1,t) - (2.0 * A(i,t)) + A(i+1,t)))
```

where c is a constant

- Amplitude will depend on :
  - previous timesteps (t, t-1)
  - neighboring points (i-1, i+1).
  - Data dependence = a parallel solution will involve communications.

1-D Wave Equation
Parallel Solution

- Amplitude array is partitioned and distributed as subarrays
- Each task owns a portion of the total array.
- Load balancing: all points require equal work, so the points should be divided equally
- Block decomposition would have the work partitioned into the number of tasks as chunks, allowing each task to own mostly contiguous data points.
- Communication need only occur on data borders. The larger the block size the less the communication.