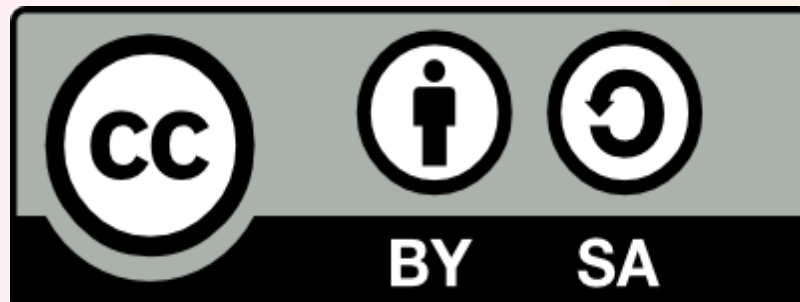


Parallel Processing

Dr. Pierre Vignéras

<http://www.vigneras.name/pierre>



This work is licensed under a Creative Commons
Attribution-Share Alike 2.0 France.

See

<http://creativecommons.org/licenses/by-sa/2.0/fr/>
for details

Text & Reference Books

- *Fundamentals of Parallel Processing*

Harry F. Jordan & Gita Alaghband

Pearson Education

ISBN: 81-7808-992-0

- *Parallel & Distributed Computing Handbook*

Albert Y. H. Zomaya

McGraw-Hill Series on Computer Engineering

ISBN: 0-07-073020-2

- *Parallel Programming (2nd edition)*

Barry Wilkinson and Michael Allen

Pearson/Prentice Hall

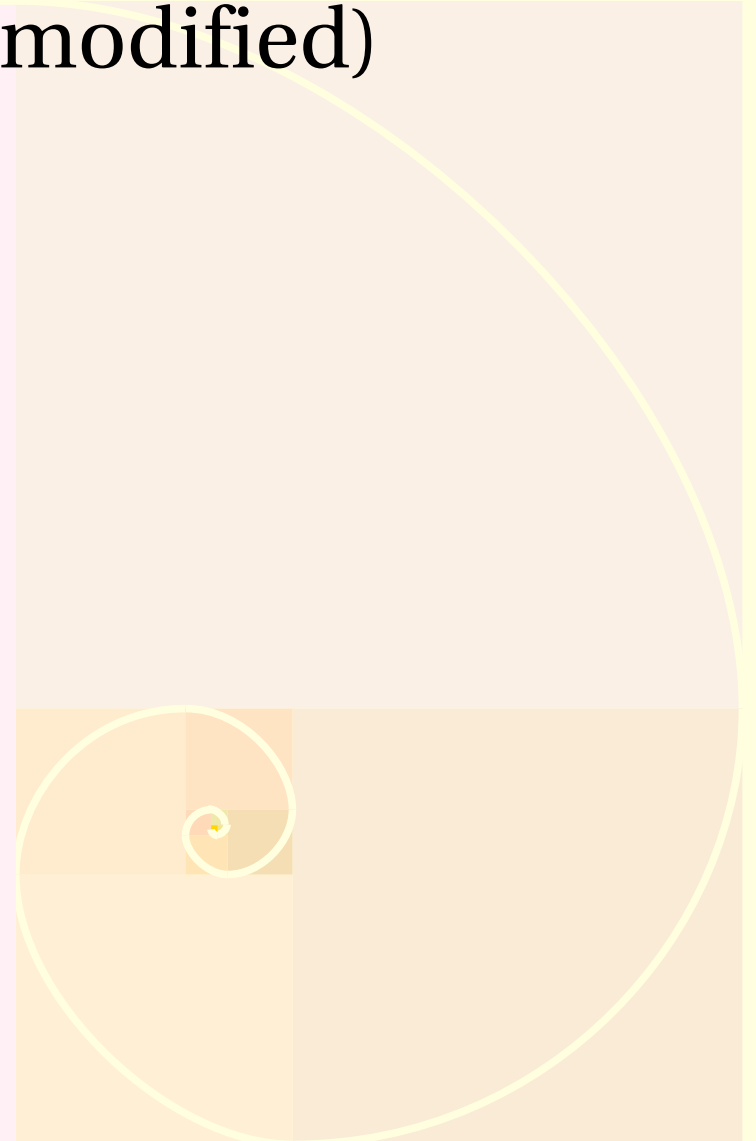
ISBN: 0-13-140563-2

Class, Quiz & Exam Rules

- No entry after the first 5 minutes
- No exit before the end of the class
- Announced Quiz
 - At the beginning of a class
 - Fixed timing (you may suffer if you arrive late)
 - Spread Out (do it quickly to save your time)
 - Papers that are not strictly in front of you will be considered as done
 - Cheaters will get '-1' mark
 - Your head should be in front of your paper!

Grading Policy

- (Temptative, can be slightly modified)
 - Quiz: 5 %
 - Assignments: 5%
 - Project: 15%
 - Mid: 25 %
 - Final: 50 %



Outline

(Important points)

- Basics
 - *[SI|MI][SD|MD],*
 - *Interconnection Networks*
 - *Language Specific Constructions*
- Parallel Complexity
 - *Size and Depth of an algorithm*
 - *Speedup & Efficiency*
- SIMD Programming
 - *Matrix Addition & Multiplication*
 - *Gauss Elimination*

Basics

- ***Flynn's Classification***
- ***Interconnection Networks***
- ***Language Specific Constructions***

History

- Since
 - Computers are based on the Von Neumann Architecture
 - Languages are based on either Turing Machines (imperative languages), Lambda Calculus (functional languages) or Boolean Algebra (Logic Programming)
- Sequential processing is the way both computers and languages are designed
- But doing several things at once is an obvious way to *increase speed*

Definitions

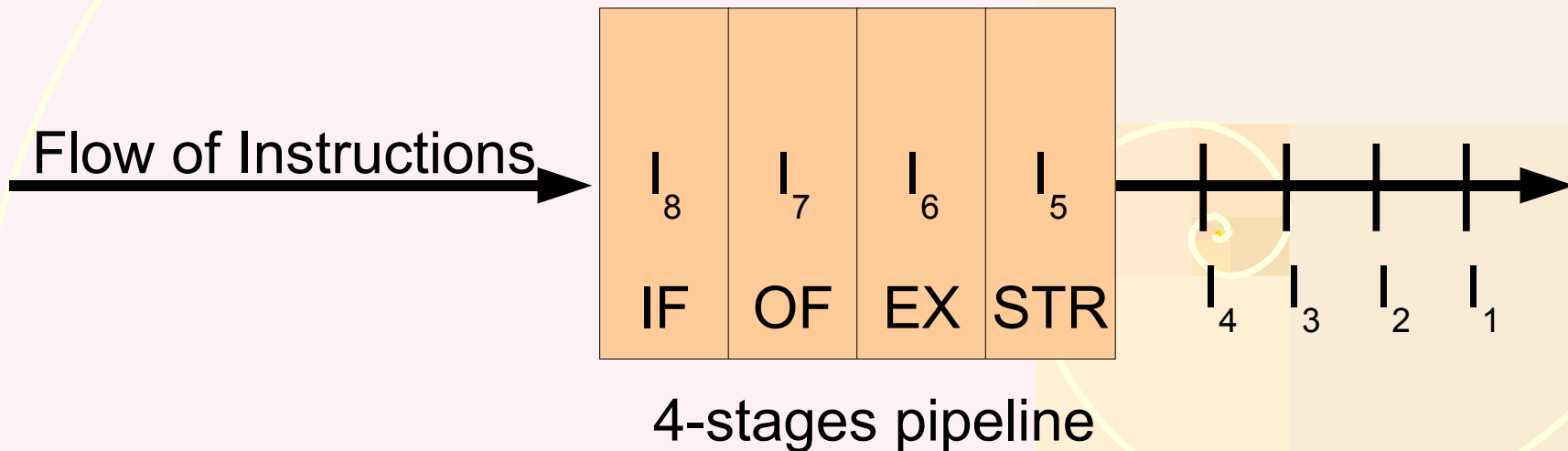
- Parallel Processing objective is *execution speed improvement*
- The manner is by doing several things at once
- The instruments are
 - Architecture,
 - Languages,
 - Algorithm
- Parallel Processing is an old field
 - but always limited by the wide sequential world for various reasons (financial, psychological, ...)

Flynn Architecture Characterization

- Instruction Stream (what to do)
 - S: Single, M: Multiple
- Data Stream (on what)
 - I: Instruction, D: Data
- Four possibilities
 - SISD: usual computer
 - SIMD: “Vector Computers” (Cray)
 - MIMD: Multi-Processor Computers
 - MISD: Not very useful (sometimes presented as pipelined SIMD)

SISD Architecture

- Traditional Sequential Computer
- Parallelism at the Architecture Level:
 - Interruption
 - IO processor
 - *Pipeline*

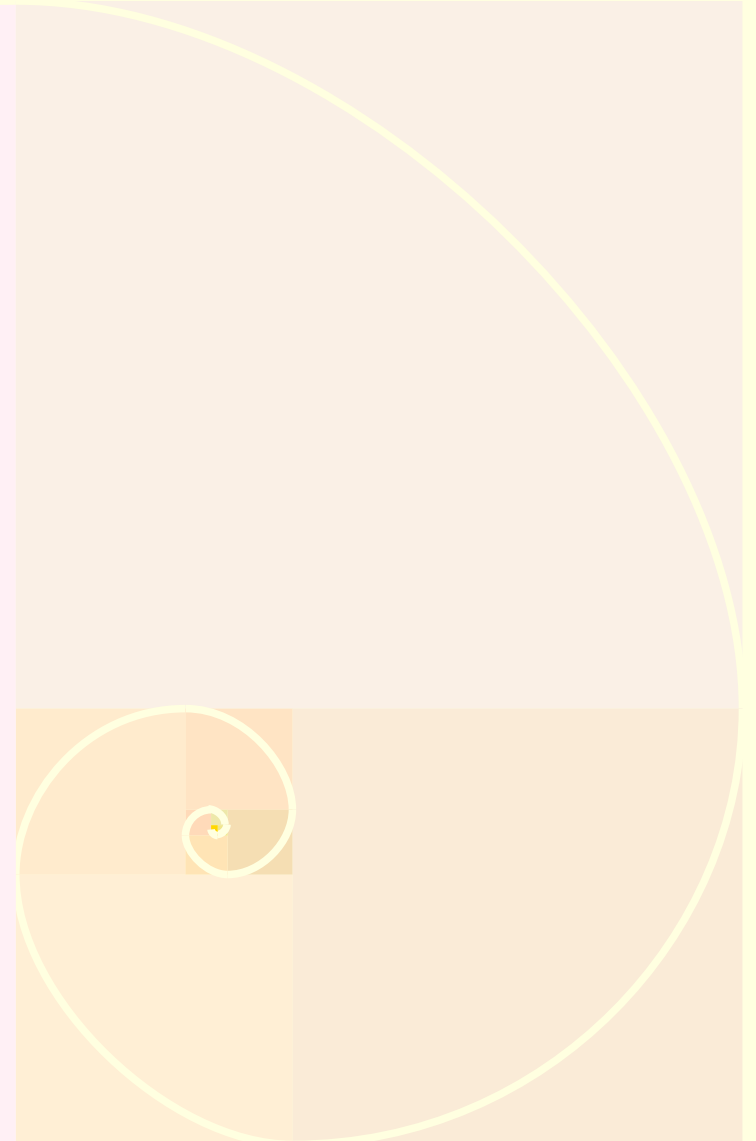


Problem with pipelines

- Conditional branches need complete flush
 - The next instruction depends on the final result of a condition
- Dependencies
 - Wait (inserting NOP in pipeline stages) until a previous result becomes available
- Solutions:
 - Branch prediction, out-of-order execution, ...
- Consequences:
 - Circuitry Complexities (cost)
 - Performance

Exercises: Real Life Examples

- Intel:
 - Pentium 4:
 - Pentium 3 (=M=Core Duo):
- AMD
 - Athlon:
 - Opteron:
- IBM
 - Power 6:
 - Cell:
- Sun
 - UltraSparc:
 - T1:

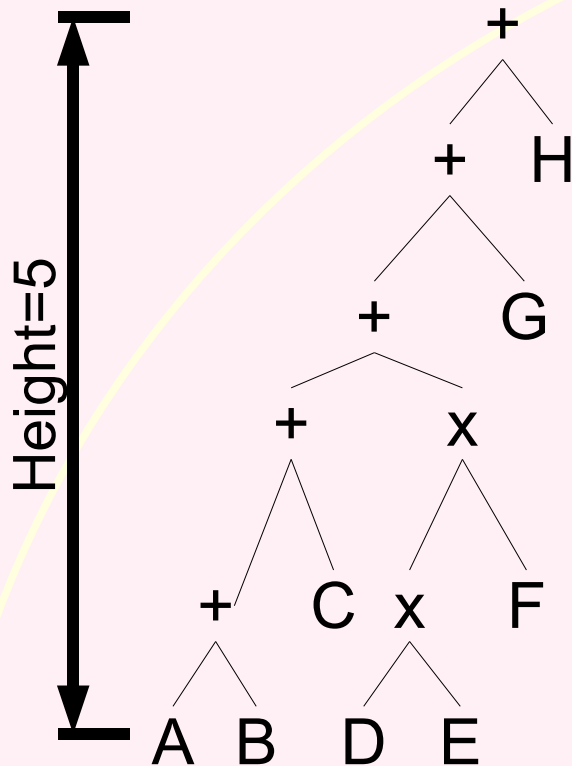


Instruction Level Parallelism

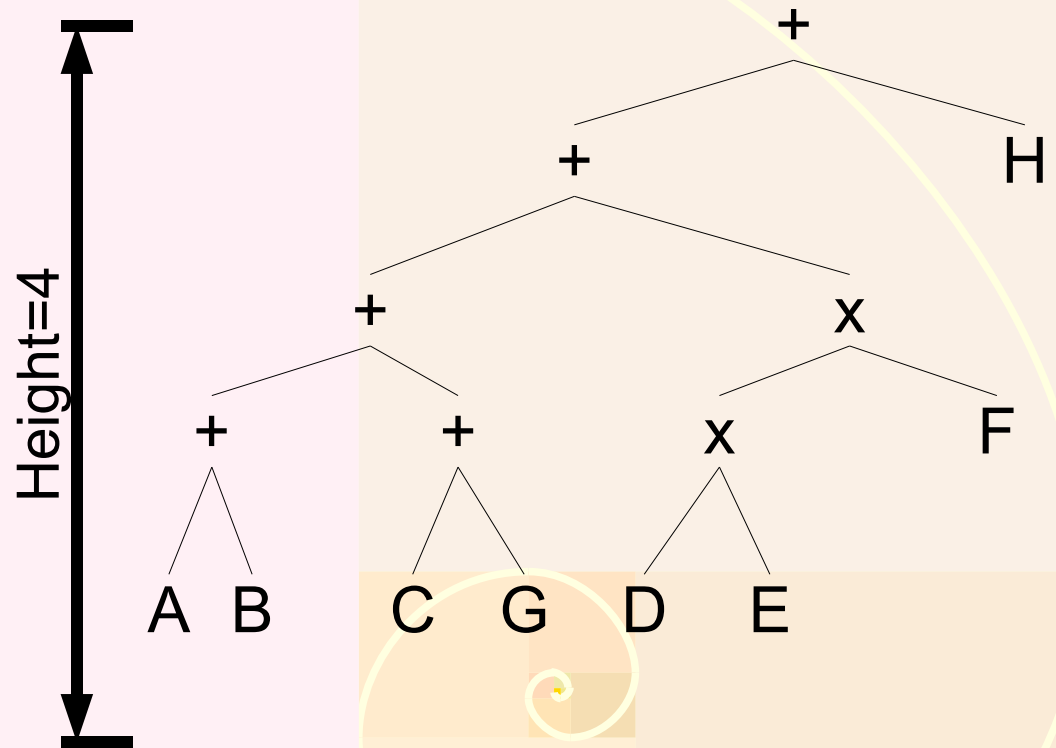
- Multiple Computing Unit that can be used simultaneously (scalar)
 - Floating Point Unit
 - Arithmetic and Logic Unit
- Reordering of instructions
 - By the compiler
 - By the hardware
- Lookahead (prefetching) & Scoreboarding (resolve conflicts)
 - Original Semantic is Guaranteed

Reordering: Example

- $EXP = A + B + C + (D \times E \times F) + G + H$



5 Steps required if hardware can do 1 add and 1 multiplication simultaneously



4 Steps required if hardware can do 2 adds and 1 multiplication simultaneously

SIMD Architecture

- Often (especially in scientific applications), one instruction is applied on different data

```
for (int i = 0; i < n; i++) {  
    r[i]=a[i]+b[i];  
}
```

- Using vector operations (add, mul, div, ...)
- Two versions
 - True SIMD: n Arithmetic Unit
 - n high-level operations at a time
 - Pipelines SIMD: Arithmetic Pipelined (depth n)
 - n non-identical low-level operations at a time

Real Life Examples

- Cray: pipelined SIMD vector supercomputers
- Modern Micro-Processor Instruction Set contains SIMD instructions
 - Intel: MMX, SSE{1,2,3}
 - AMD: 3dnow
 - IBM: AltiVec
 - Sun: VSI
- Very efficient
 - 2002: winner of the TOP500 list was the Japanese Earth Simulator, a vector supercomputer

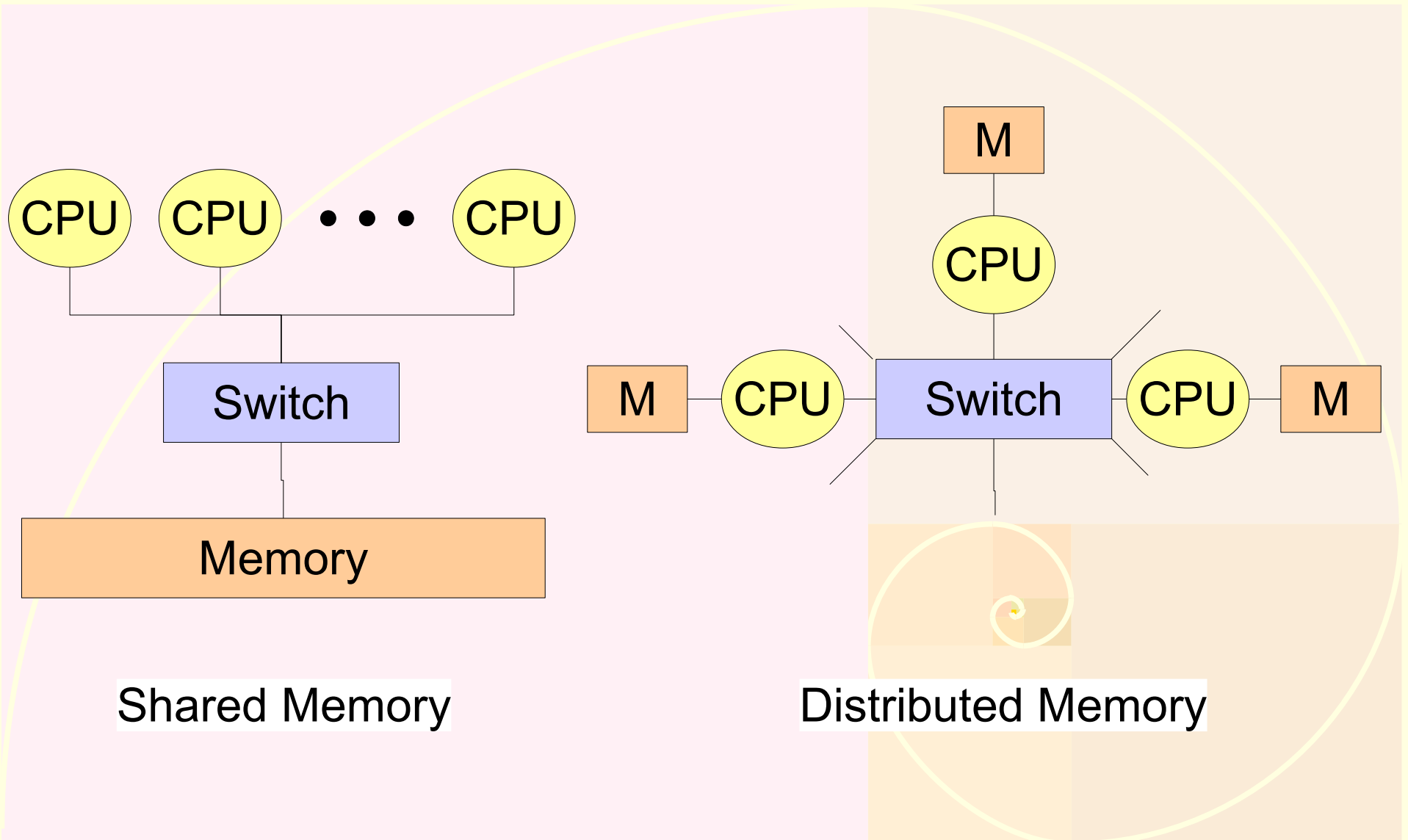
MIMD Architecture

- Two forms
 - Shared memory: any processor can access any memory region
 - Distributed memory: a distinct memory is associated with each processor
- Usage:
 - Multiple sequential programs can be executed simultaneously
 - Different parts of a program are executed simultaneously on each processor
 - Need cooperation!

Warning: Definitions

- Multiprocessors: computers capable of running multiple instructions streams simultaneously to cooperatively execute a single program
- Multiprogramming: sharing of computing resources by independent jobs
- Multiprocessing:
 - running a possibly sequential program on a multiprocessor (not our concern here)
 - running a program consisting of multiple cooperating processes

Shared vs Distributed

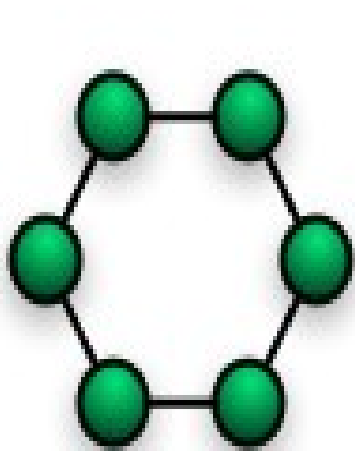


MIMD

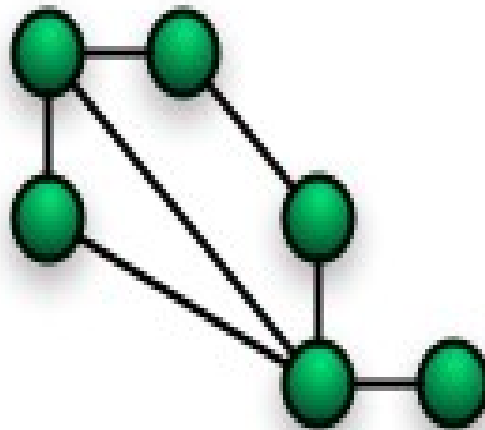
Cooperation/Communication

- Shared memory
 - read/write in the shared memory by hardware
 - synchronisation with locking primitives (semaphore, monitor, etc.)
 - Variable Latency
- Distributed memory
 - explicit message passing in software
 - synchronous, asynchronous, grouped, remote method invocation
 - Large message may mask long and variable latency

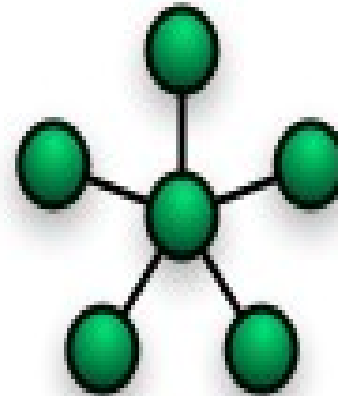
Interconnection Networks



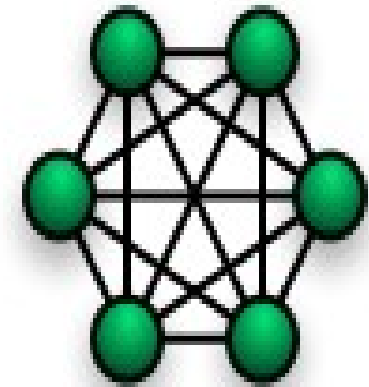
Ring



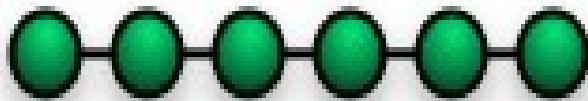
Mesh



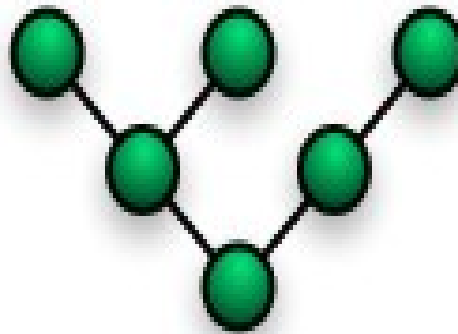
Star



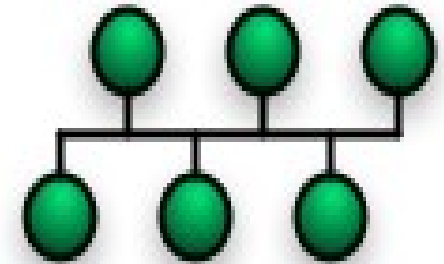
Fully Connected



Line



Tree



Bus

“Hard” Network Metrics

- Bandwidth (B): maximum number of bytes the network can transport
- Latency (L): transmission time of a single message
 - 4 components
 - software overhead (so): msg packing/unpacking
 - routing delay (rd): time to establish the route
 - contention time (ct): network is busy
 - channel delay (cd): msg size/bandwidth
 - so+ct depends on program behaviour
 - rd+cd depends on hardware

“Soft” Network Metrics

- Diameter (r): longest path between two nodes
- Average Distance (d_a):
$$\frac{1}{N-1} \sum_{d=1}^r d \cdot N_d$$

where d is the distance between two nodes defined as the number of links in the shortest path between them, N the total number of nodes and N_d , the number of nodes with a distance d from a given node.

- Connectivity (P): the number of nodes that can be reached from a given node in one hop.
- Concurrency (C): the number of independent connections that a network can make.
 - bus: $C = 1$; linear: $C = N-1$ (if processor can send and receive simultaneously, otherwise, $C = \lfloor N/2 \rfloor$)

Influence of the network

- Bus
 - low cost,
 - low concurrency degree: 1 message at a time
- Fully connected
 - high cost: $N(N-1)/2$ links
 - high concurrency degree: N messages at a time
- Star
 - Low cost (high availability)
 - Exercice: Concurrency Degree?

Parallel Responsibilities

Human writes a
Human implements using a
Compiler translates into a

Runtime/Libraries are required in a
Human & Operating System map to the
The hardware is leading the

Parallel Algorithm
Language
Parallel Program

Parallel Process
Hardware
Parallel Execution

Many actors are involved in the execution of a program (even sequential). Parallelism can be introduced automatically by compilers, runtime/libraries, OS and in the hardware

We usually consider **explicit parallelism**: when parallelism is introduced at the **language level**

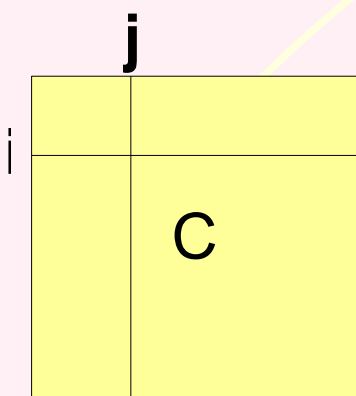
SIMD Pseudo-Code

- SIMD

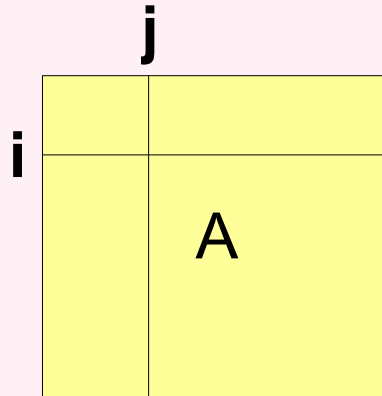
`<indexed variable> = <indexed expression>, (<index range>);`

Example:

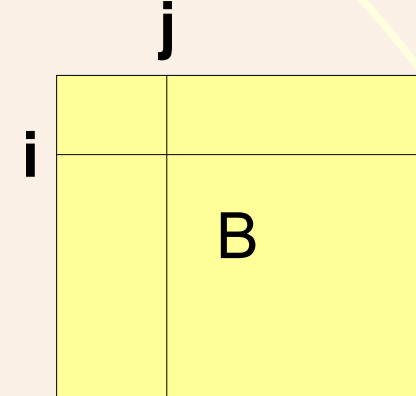
`C[i,j] == A[i,j] + B[i,j], (0 ≤ i ≤ N-1)`



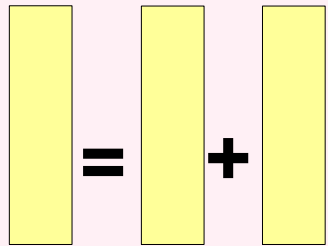
=



+

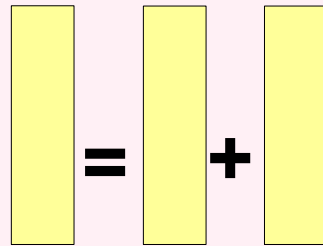


j = 0



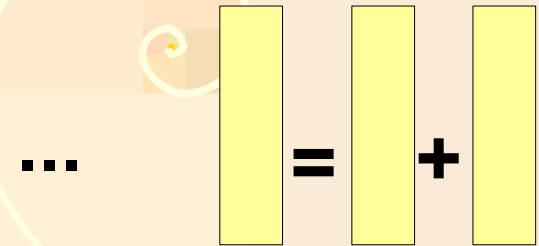
C[i] A[i] B[i]

j = 1



C[i] A[i] B[i]

j = N-1



C[i] A[i] B[i]

SIMD Matrix Multiplication

- General Formula

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj}$$

```
for (i = 0; i < N; i++) {  
    // SIMD Initialisation  
    C[i,j] = 0, (0 <= j < N);  
    for (k = 0; k < N; k++) {  
        // SIMD computation  
        C[i,j]=C[i,j]+A[i,k]*B[k,j], (0 <= j < N);  
    }  
}
```

SIMD C Code (GCC)

- gcc vector type & operations

- Vector of 8 bytes long, composed of 4 short elements

```
typedef short v4hi __attribute__((vector_size (8)));
```

- Use union to access individual elements

```
union vec { v4hi v, short s[4]; };
```

- Usual operations on vector variables

```
v4hi v, w, r; r = v+w*w/v;
```

- gcc builtins functions

- #include <mmintrin.h> // MMX only

- Compile with -m32

- Use the provided functions

```
_mm_setr_pi16 (short w0, short w1, short w2, short w3);
```

```
_mm_add_pi16 (__m64 m1, __m64 m2);
```

```
...
```

MIMD Shared Memory Pseudo Code

- New instructions:
 - `fork <label>` Start a new process executing at `<label>`;
 - `join <integer>` Join `<integer>` processes into one;
 - `shared <variable list>` make the storage class of the variables shared;
 - `private <variable list>` make the storage class of the variables private.
- Sufficient? What about synchronization?
 - We will focus on that later on...

MIMD Shared Memory Matrix Multiplication

```
• private i,j,k;
  shared A[N,N], B[N,N], C[N,N], N;
  for (j = 0; j < N - 2; j++) fork DOCOL;
  // Only the Original process reach this point
  j = N-1;
  DOCOL: // Executed by N process for each column
  for (i = 0; i < N; i++) {
    C[i,j] = 0;
    for (k = 0; k < N; k++) {
      // SIMD computation
      C[i,j]=C[i,j]+A[i,k]*B[k,j];
    }
  }
  join N; // Wait for the other process
```

MIMD Shared Memory C Code

- Process based:
 - fork()/wait()
- Thread Based:
 - Pthread library
 - Solaris Thread library
- Specific Framework: OpenMP
- Exercice: read the following articles:
 - “Imperative Concurrent Object-Oriented Languages”, M.Philippsen, 1995
 - “Threads Cannot Be Implemented As a Library”, Hans-J.Boehm, 2005

MIMD Distributed Memory Pseudo Code

- Basic instructions `send()` / `receive()`
- Many possibilities:
 - non-blocking send needs message buffering
 - non-blocking receive needs failure return
 - blocking send and receive both needs termination detection
- Most used combinations
 - blocking send and receive
 - non-blocking send and blocking receive
- Grouped communications
 - scatter/gather, broadcast, multicast, ...

MIMD Distributed Memory C Code

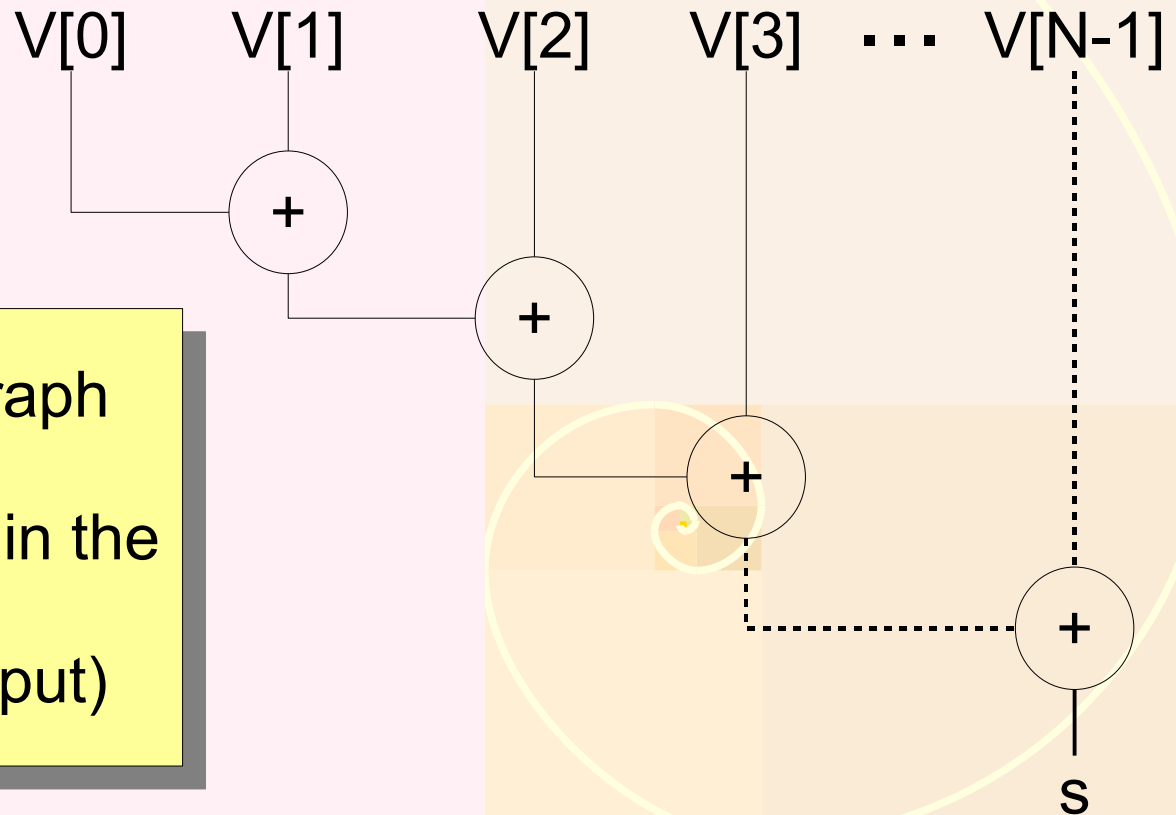
- Traditional Sockets
 - Bad Performance
 - Grouped communication limited
- Parallel Virtual Machine (PVM)
 - Old but good and portable
- Message Passing Interface (MPI)
 - The standard used in parallel programming today
 - Very efficient implementations
 - Vendors of supercomputers provide their own highly optimized MPI implementation
 - Open-source implementation available

Performance

- ***Size and Depth***
- ***Speedup and Efficiency***

Parallel Program Characteristics

```
• s = V[0];  
for (i = 1; i < N; i++) {  
    s = s + V[i];  
}
```

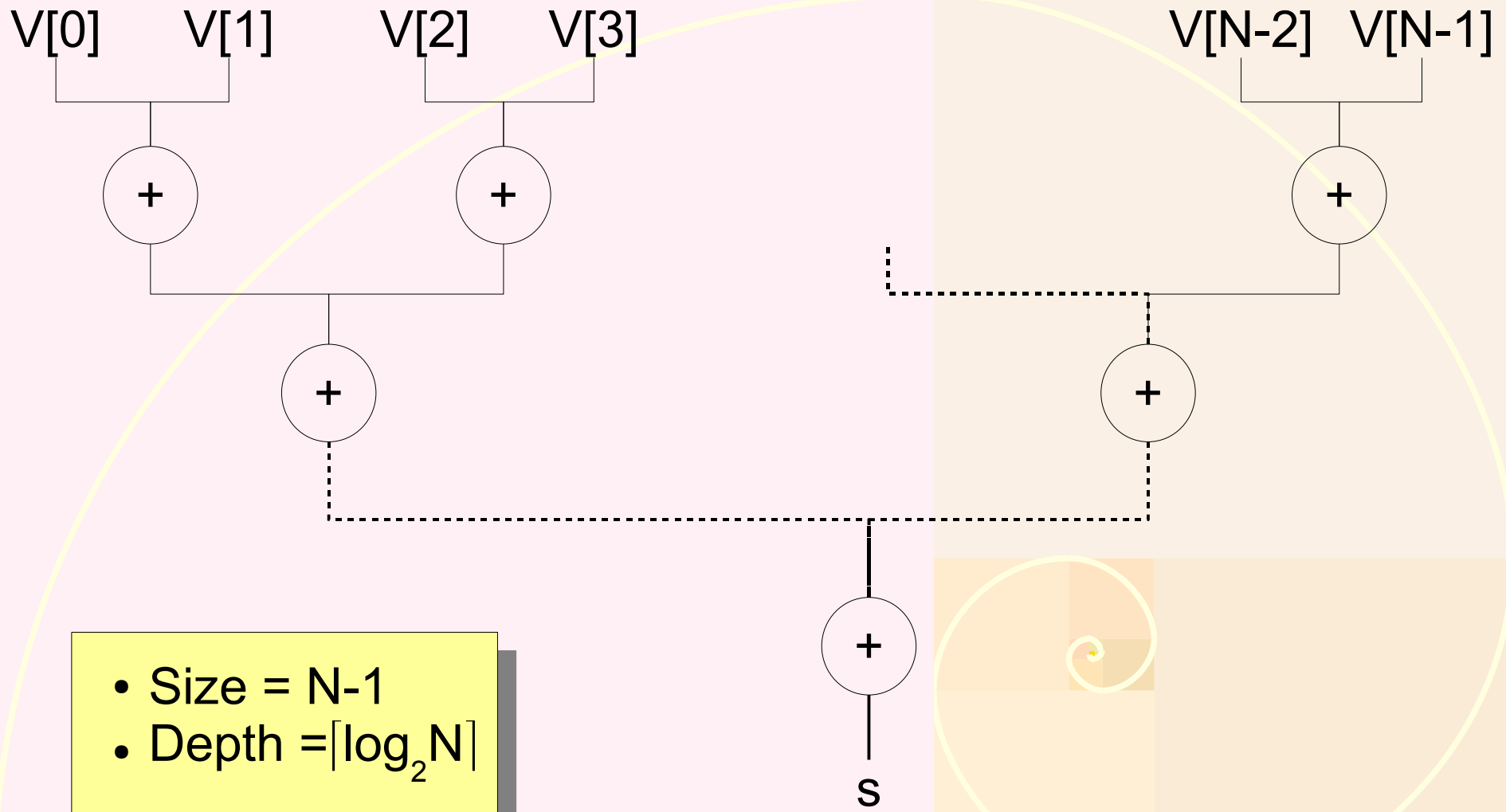


Data Dependence Graph

- Size = $N-1$ (nb op)
- Depth = $N-1$ (nb op in the longest path from any input to any output)

Parallel Program Characteristics

I. Performance

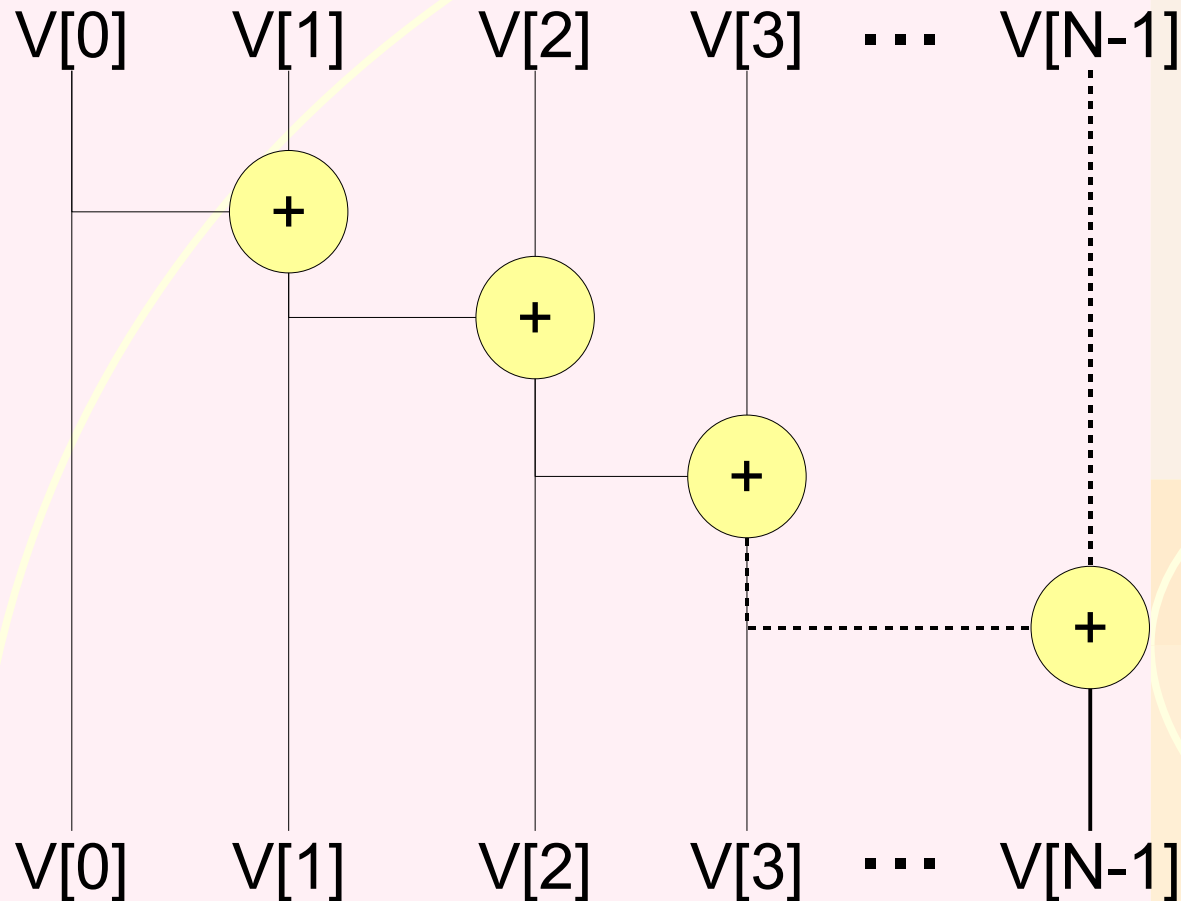


Parallel Program Characteristics

- Size and Depth does not take into account many things
 - memory operation (allocation, indexing, ...)
 - communication between processes
 - read/write for shared memory MIMD
 - send/receive for distributed memory MIMD
 - Load balancing when the number of processors is less than the number of parallel task
 - etc...
- Inherent Characteristics of a Parallel Machine independent of any specific architecture

Prefix Problem

```
• for (i = 1; i < N; i++) {  
    V[i] = V[i-1] + V[i];  
}
```



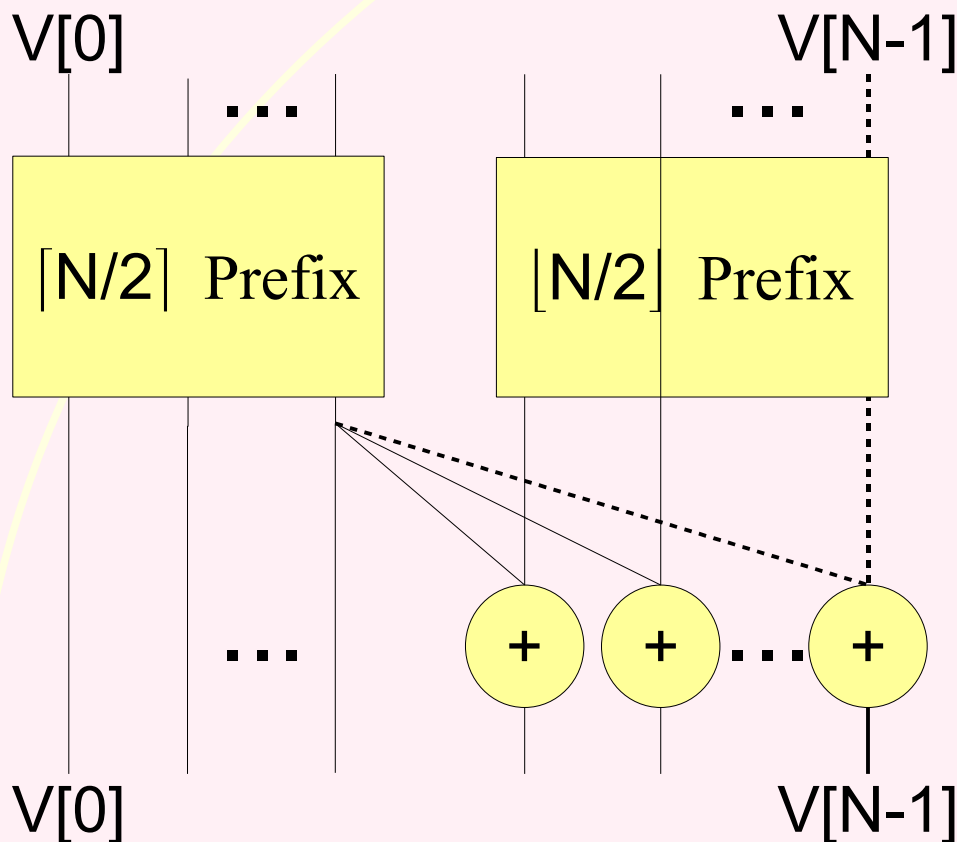
- Size = N-1
- Depth = N-1

How to make this program parallel?

Prefix Solution 1

Upper/Lower Prefix Algorithm

- Divide and Conquer
 - Divide a problem of size N into 2 equivalent problems of size $N/2$



Apply the same algorithm for sub-prefix.
When $N=2$, use the direct, sequential implementation

Size has increased!
 $N-2+N/2 > N-1$ ($N>2$)

Depth = $\lceil \log_2 N \rceil$
Size ($N=2^k$) = $k \cdot 2^{k-1}$
= $(N/2) \cdot \log_2 N$

Analysis of P^{ul}

- Proof by induction
- Only for N a power of 2
 - Assume from construction that size increase smoothly with N
- Example: $N=1,048,576=2^{20}$
 - Sequential Prefix Algorithm:
 - 1,048,575 operations in 1,048,575 time unit steps
 - Upper/Lower Prefix Algorithm
 - 10,485,760 operations in 20 time unit steps

Other Prefix Parallel Algorithms

- P^{ul} requires $N/2$ processors available!
 - In our example, 524,288 processors!
- Other algorithms (exercice: study them)
 - Odd/Even Prefix Algorithm
 - Size = $2N - \log_2(N) - 2$ ($N=2^k$)
 - Depth = $2\log_2(N) - 2$
 - Ladner and Fischer's Parallel Prefix
 - Size = $4N - 4.96N^{0.69} + 1$
 - Depth = $\log_2(N)$
- Exercice: implement them in C

Benchmarks

(Inspired by Pr. Ishfaq Ahmad Slides)

- A benchmark is a performance testing program
 - supposed to capture processing and data movement characteristics of a class of applications.
- Used to measure and to predict performance of computer systems, and to reveal their architectural weakness and strong points.
- Benchmark suite: set of benchmark programs.
- Benchmark family: set of benchmark suites.
- Classification:
 - scientific computing, commercial, applications, network services, multimedia applications, ...

Benchmark Examples

Type	Name	Measuring
Micro-Benchmark	LINPACK	Numerical computing (linear algebra)
	LMBENCH	System calls and data movement operations in Unix
	STREAM	Memory bandwidth
Macro-Benchmark	NAS	Parallel computing (CFD)
	PARKBENCH	Parallel computing
	SPEC	A mixed benchmark family
	Splash	Parallel computing
	STAP	Signal processing
	TPC	Commercial applications

SPEC Benchmark Family

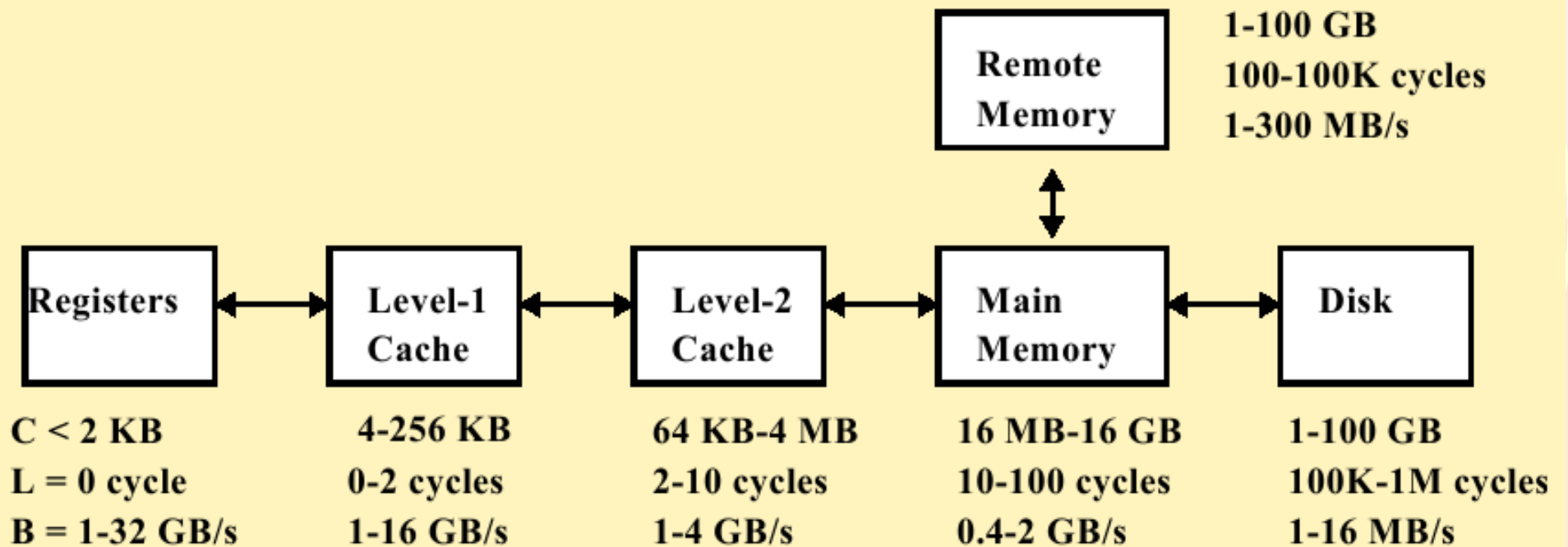
- Standard Performance Evaluation Corporation
 - Started with benchmarks that measure CPU performance
 - Has extended to client/server computing, commercial applications, I/O subsystems, etc.
- Visit <http://www.spec.org/> for more informations

Performance Metrics

- Execution time
 - Generally in seconds
 - Real time, User Time, System time?
 - Real time
- Instruction Count
 - Generally in MIPS or BIPS
 - Dynamic: number of executed instructions, not the number of assembly line code!
- Number of Floating Point Operations Executed
 - Mflop/s, Gflop/s
 - For scientific applications where flop dominates

Memory Performance

- Three parameters:
 - Capacity,
 - Latency,
 - Bandwidth



Parallelism and Interaction Overhead

- Time to execute a parallel program is

$$T = T_{\text{comp}} + T_{\text{par}} + T_{\text{inter}}$$

- T_{comp} : time of the effective computation

- T_{par} : parallel overhead

- Process management (creation, termination, context switching)

- Grouping operations (creation, destruction of group)

- Process Inquiry operation (Id, Group Id, Group size)

- T_{inter} : interaction overhead

- Synchronization (locks, barrier, events, ...)

- Aggregation (reduction and scan)

- Communication (p2p, collective, read/write shared variables)

Measuring Latency Ping-Pong

```
for (i=0; i < Runs; i++)
  if (my_node_id == 0) {      /* sender */
    tmp = Second();
    start_time = Second();
    send an m-byte message to node 1;
    receive an m-byte message from node 1;
    end_time = Second();
    timer_overhead = start_time - tmp ;
    total_time = end_time - start_time - timer_overhead;
    communication_time[i] = total_time / 2 ;
  } else if (my_node_id ==1) { /* receiver */
    receive an m-byte message from node 0;
    send an m-byte message to node 0;
  }
}
```

Measuring Latency

Hot-Potato (fire-brigade) Method

- n nodes are involved (instead of just two)
- Node 0 sends an m -bytes message to node 1
- On receipt, node 1 immediately sends the same message to node 2 and so on.
- Finally node $n-1$ sends the message back to node 0
- The total time is divided by n to get the point-to-point average communication time.

Collective Communication Performance

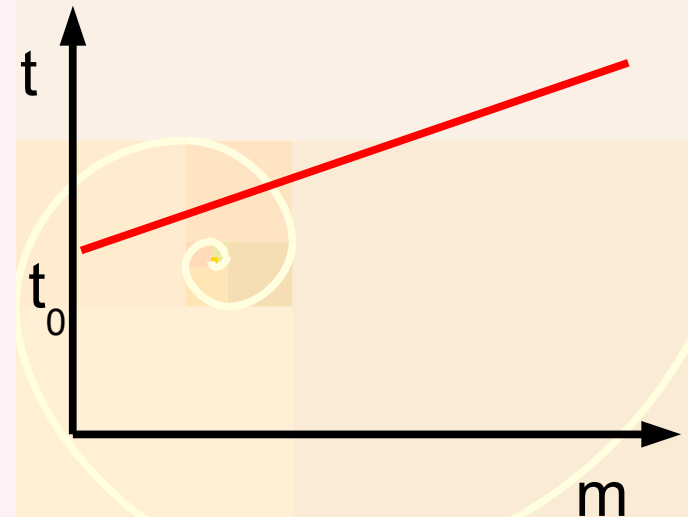
```
for (i = 0; i < Runs; i++) {  
    barrier synchronization;  
    tmp = Second();  
    start_time = Second();  
    for (j = 0; j < Iterations; j++)  
        the_collective_routine_being_measured;  
    end_time = Second();  
    timer_overhead = start_time - tmp;  
    total_time = end_time - start_time - timer_overhead;  
    local_time = total_time / Iterations;  
    communication_time[i] = max {all n local_time};  
}
```

How to get all n local time?

Point-to-Point Communication Overhead

- Linear function of the message length m (in bytes)
 - $t(m) = t_0 + m/r_\infty$
 - t_0 : start-up time
 - r_∞ : asymptotic bandwidth

- Ideal Situation!
- Many things are left out
 - Topology metrics
 - Network Contention



Collective Communication

- Broadcast: 1 process sends an m -bytes message to all n process
- Multicast: 1 process sends an m -bytes message to $p < n$ process
- Gather: 1 process receives an m -bytes from each of the n process (mn bytes received)
- Scatter: 1 process sends a distinct m -bytes message to each of the n process (mn bytes sent)
- Total exchange: every process sends a distinct m -bytes message to each of the n process (mn^2 bytes sent)
- Circular Shift: process i sends an m -bytes message to process $i+1$ (process $n-1$ to process 0)

Collective Communication Overhead

- Function of both m and n
 - start-up latency depends only on n

$$T(m,n) = t_0(n) + m/r_\infty(n)$$

- Many problems here! For a broadcast for example, how is done the communication at the link layer?
 - Sequential?
 - Process 1 sends 1 m-bytes message to process 2, then to process 3, and so on... (n steps required)
 - Truly Parallel (1 step required)
 - Tree based ($\log_2(n)$ steps required)
- Again, many things are left out
 - Topology metrics
 - Network Contention

Speedup (Problem-oriented)

- For a given algorithm, let T_p be time to perform the computation using p processors
 - T_∞ : depth of the algorithm
- How much time we gain by using a parallel algorithm?
 - Speedup: $S_p = T_1 / T_p$
 - Issue: what is T_1 ?
 - Time taken by the execution of the **best sequential algorithm** on the same input data
 - We note $T_1 = T_s$

Speedup consequence

- Best Parallel Time Achievable is: T_s / p
- Best Speedup is bounded by $S_p \leq T_s / (T_s / p) = p$
 - Called Linear Speedup
- Conditions
 - Workload can be divided into p equal parts
 - No overhead at all
- Ideal situation, usually impossible to achieve!
- Objective of a parallel algorithm
 - Being as close to a linear speedup as possible

Biased Speedup (Algorithm-oriented)

- We usually don't know what is the best sequential algorithm except for some simple problems (sorting, searching, ...)
- In this case, T_1
 - is the time taken by the execution of **the same parallel algorithm** on 1 processor.
- Clearly biased!
 - The parallel algorithm and the best sequential algorithm are often very, very different
- This speedup should be taken with that limitation in mind!

Biased Speedup Consequence

- Since we compare our parallel algorithm on p processors with a suboptimal sequential (parallel algorithm on only 1 processor) we can have: $S_p > p$
 - Superlinear speedup
- Can also be the consequence of using
 - a unique feature of the system architecture that favours parallel formation
 - indeterminate nature of the algorithm
 - Extra-memory due to the use of p computers (less swap)

Efficiency

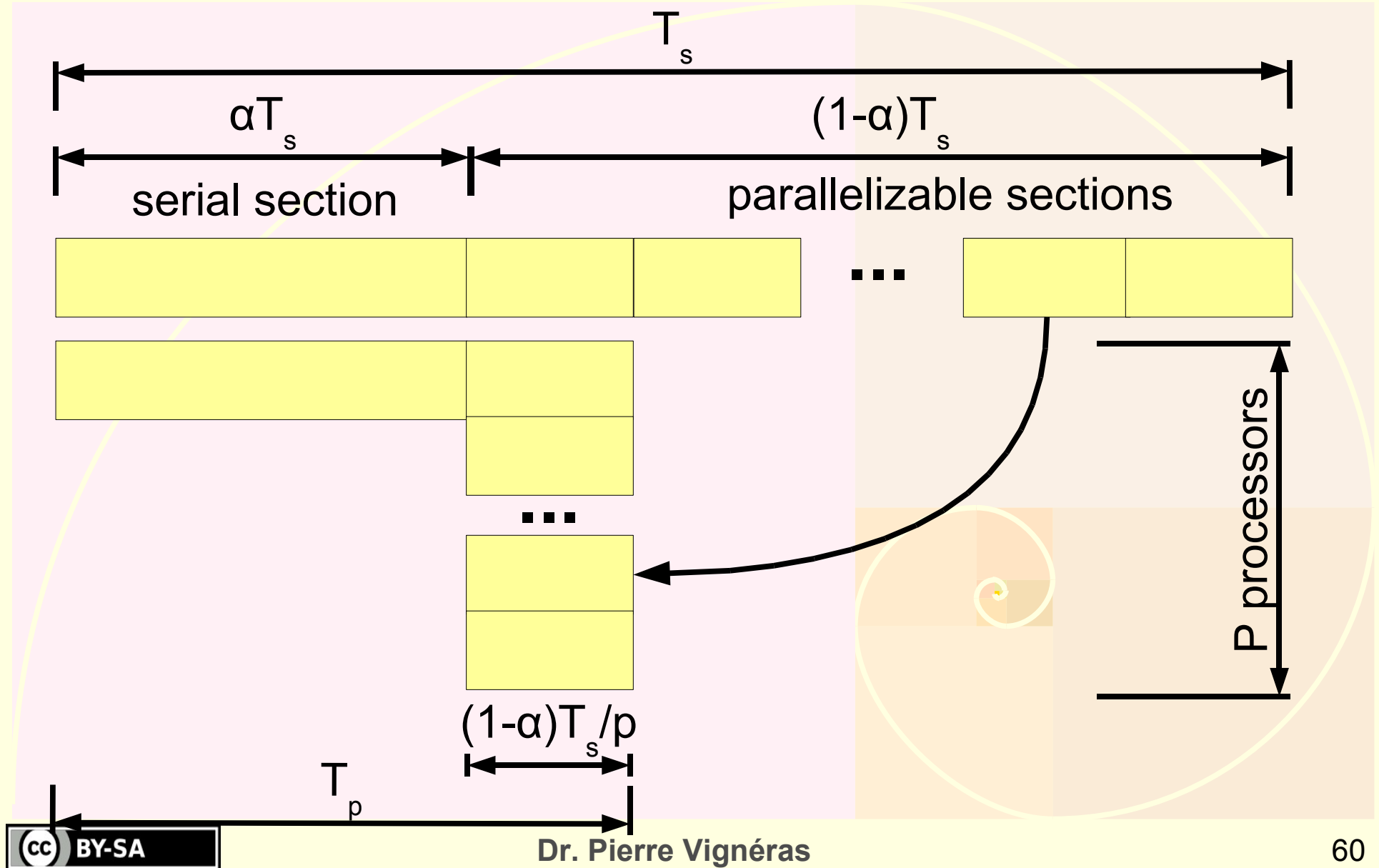
- How long processors are being used on the computation?
 - Efficiency: $E_p = S_p / p$ (expressed in %)
 - Also represents how much benefits we gain by using p processors
- Bounded by 100%
 - Superlinear speedup of course can give more than 100% but it is **biased**

Amdhal's Law: fixed problem size (1967)

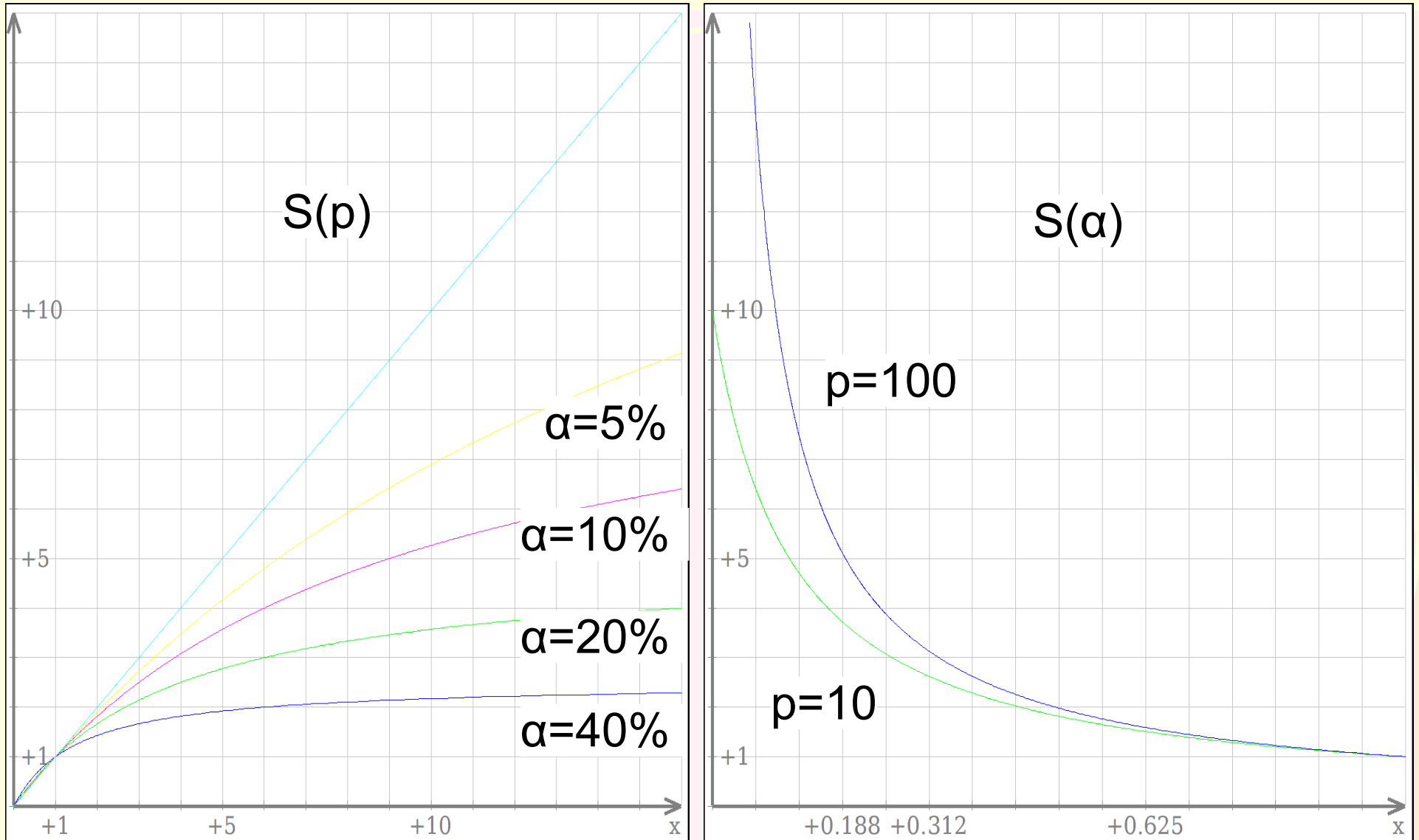
- For a problem with a workload W , we assume that we can divide it into two parts:
 - $W = \alpha W + (1-\alpha)W$
 α percent of W **must be executed sequentially**, and the remaining $1-\alpha$ **can be executed** by n nodes simultaneously with 100% of efficiency
 - $T_s(W) = T_s(\alpha W + (1-\alpha)W) = T_s(\alpha W) + T_s((1-\alpha)W) = \alpha T_s(W) + (1-\alpha)T_s(W)$
 - $T_p(W) = T_p(\alpha W + (1-\alpha)W) = T_s(\alpha W) + T_p((1-\alpha)W) = \alpha T_s(W) + (1-\alpha)T_s(W)/p$

$$S_p = \frac{T_s}{\alpha T_s + \frac{(1-\alpha)T_s}{p}} = \frac{p}{1 + (p-1)\alpha} \rightarrow \frac{1}{\alpha} \quad (p \rightarrow \infty)$$

Amdhal's Law: fixed problem size



Amdhal's Law Consequences



Amdhal's Law Consequences

- For a fixed workload, and without any overhead, the maximal speedup as an upper bound of $1/\alpha$
- Taking into account the overhead T_o
 - Performance is limited not only by the sequential bottleneck but also by the average overhead

$$S_p = \frac{T_s}{\alpha T_s + \frac{(1-\alpha)T_s}{p} + T_o} = \frac{p}{1 + (p-1)\alpha + \frac{pT_o}{T_s}}$$

$$S_p \rightarrow \frac{1}{\alpha + \frac{T_o}{T_s}} \quad (p \rightarrow \infty)$$

Amdhal's Law Consequences

- The sequential bottleneck cannot be solved just by increasing the number of processors in a system.
 - Very pessimistic view on parallel processing during two decades .
- Major assumption: the problem size (workload) is fixed.

Conclusion don't use parallel programming on a small sized problem!

Gustafson's Law: fixed time (1988)

- As the machine size increase, the problem size may increase with execution time unchanged
- Sequential case: $W = \alpha W + (1 - \alpha)W$
 - $T_s(W) = \alpha T_s(W) + (1 - \alpha)T_s(W)$
 - $T_p(W) = \alpha T_s(W) + (1 - \alpha)T_s(W) / p$
 - To have a constant time, the parallel workload should be increased
- Parallel case (p-processor): $W' = \alpha W + p(1 - \alpha)W$.
 - $T_s(W') = \alpha T_s(W) + p(1 - \alpha)T_s(W)$
- Assumption: sequential part is constant: it does not increase with the size of the problem

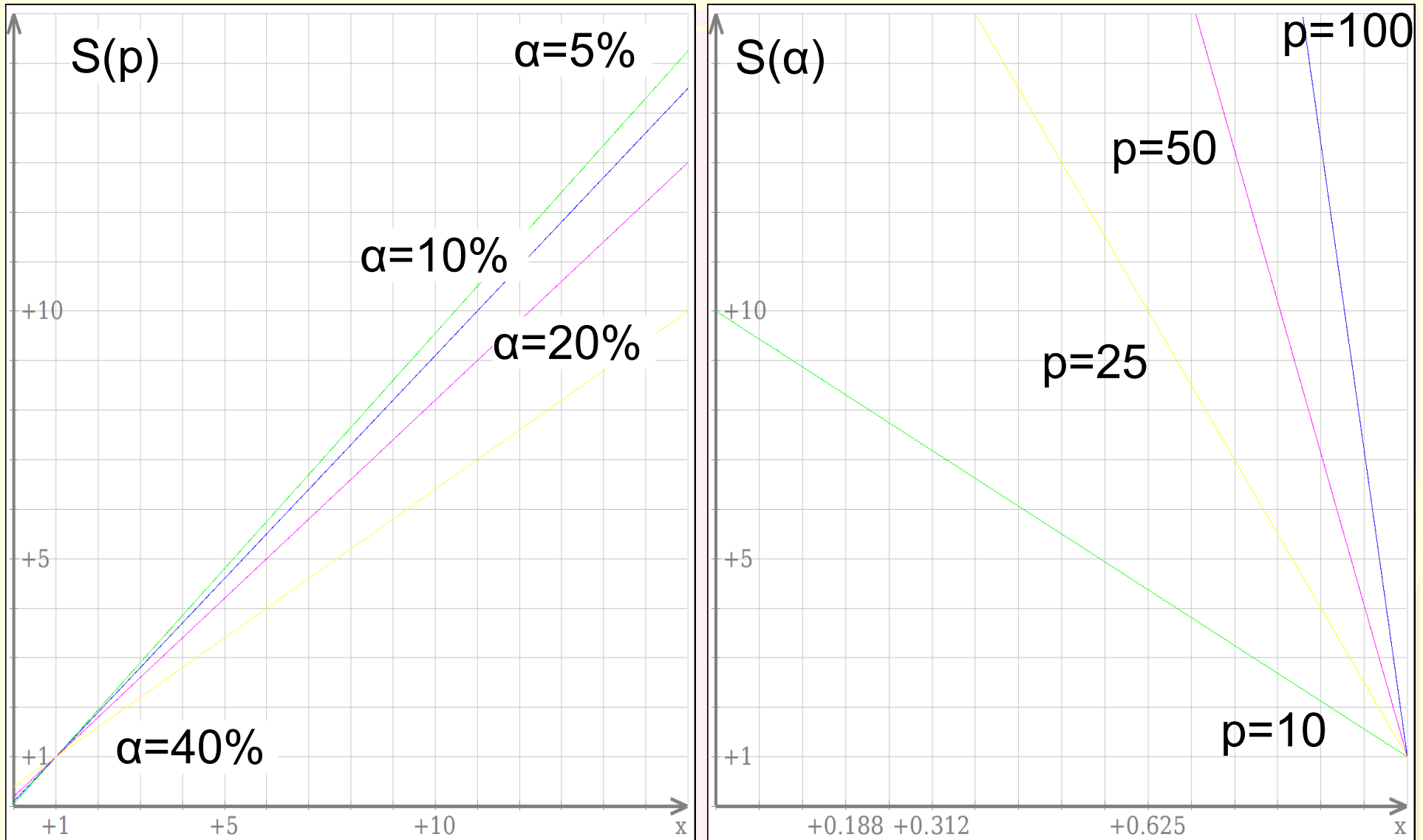
Fixed time speedup

$$S(p) = \frac{T_s(W')}{T_p(W')} = \frac{T_s(W')}{T_s(W)} = \frac{\alpha T_s(W) + p(1-\alpha)T_s(W)}{\alpha T_s(W) + (1-\alpha)T_s(W)}$$

$$S(p) = \alpha + (1-\alpha)p$$

Fixed time speedup is a linear function of p ,
if the workload is scaled up
to maintain a fixed execution time

Gustafson's Law Consequences



Fixed time speedup with overhead

$$S(p) = \frac{T_s(W')}{T_p(W')} = \frac{T_s(W')}{T_s(W) + T_o(p)} = \frac{\alpha + (1 - \alpha)p}{1 + \frac{T_o(p)}{T_s(W)}}$$

- Best fixed speedup is achieved with a low overhead as expected
- T_o : function of the number of processors
- Quite difficult in practice to make it a ***decreasing function***

SIMD Programming

- *Matrix Addition & Multiplication*
- *Gauss Elimination*

Matrix Vocabulary

- A matrix is dense if most of its elements are non-zero
- A matrix is sparse if a significant number of its elements are zero
 - Space efficient representation available
 - Efficient algorithms available
- Usual matrix size in parallel programming
 - Lower than 1000×1000 for a dense matrix
 - Bigger than 1000×1000 for a sparse matrix
 - Growing with the performance of computer architectures

Matrix Addition

$$C = A + B, \quad c_{i,j} = a_{i,j} + b_{i,j} \quad 0 \leq i \leq n, 0 \leq j \leq m$$

```
// Sequential
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        c[i,j] = a[i,j]+b[i,j];
    }
}
```

Need to map the vectored addition of m elements $a[i]+b[i]$ to our underlying architecture made of p processing elements

```
// SIMD
for (i = 0; i < n; i++) {
    c[i,j] = a[i,j]+b[i,j]; \
    (0 <= j < m)
}
```

Complexity:

- Sequential case: $n.m$ additions
- SIMD case: $n.m/p$

Speedup: $S(p)=p$

Matrix Multiplication

$$C = AB, A[n, l]; B[l, m]; C[n, m]$$
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j} \quad 0 \leq i \leq n, 0 \leq j \leq m$$

```
for (i = 0; i < n; i++) {
  for (j = 0; j < m; j++) {
    c[i,j] = 0;
    for (k = 0; k < l; k++) {
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
  }
}
```

Complexity:

- n.m initialisations
- n.m.l additions and multiplications

Overhead: Memory indexing!

- Use a temporary variable sum for c[i,j]!

SIMD Matrix Multiplication

```
for (i = 0; i < n; i++) {  
    c[i,j] = 0; (0 <= j < m) // SIMD Op  
    for (k = 0; k < l; k++) {  
        c[i,j] = c[i,j] + a[i,k]*b[k,j];\  
        (0 <= j < m)  
    }  
}
```

Complexity ($n = m = l$ for simplicity):

- n^2/p initialisations ($p \leq n$)
- n^3/p additions and multiplications ($p \leq n$)

Overhead: Memory indexing!

- Use a temporary vector for $c[i]$!

Algorithm in $O(n)$ with $p=n^2$

- Assign one processor for each element of C
- Initialisation in $O(1)$
- Computation in $O(n)$

Algorithm in $O(\lg(n))$ with $p=n^3$

- Parallelize the inner loop!
- Lowest bound

Gauss Elimination

- Matrix (n x n) represents a system of n linear equations of n unknowns

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

$$a_{n-2,0}x_0 + a_{n-2,1}x_1 + a_{n-2,2}x_2 + \dots + a_{n-2,n-1}x_{n-1} = b_{n-2}$$

...

$$a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n-1}x_{n-1} = b_1$$

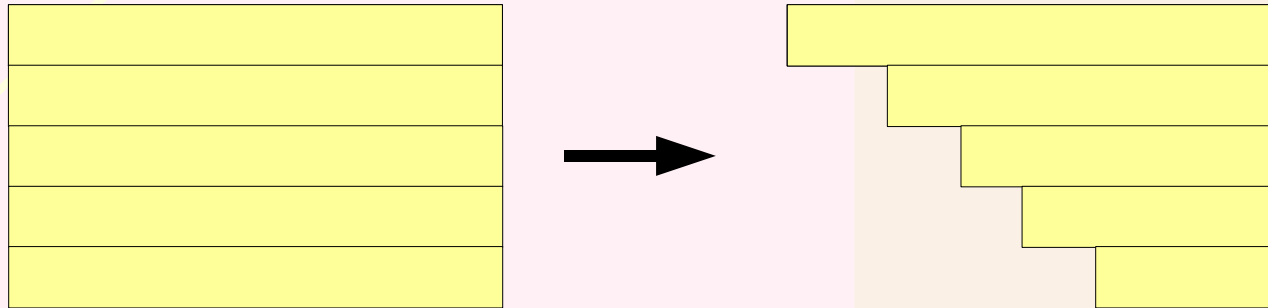
$$a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 + \dots + a_{0,n-1}x_{n-1} = b_0$$

Represented by $M =$

$$\begin{pmatrix} a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \dots & a_{n-1,n-1} & b_{n-1} \\ a_{n-2,0} & a_{n-2,1} & a_{n-2,2} & \dots & a_{n-2,n-1} & b_{n-2} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{1,0} & a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & b_1 \\ a_{0,0} & a_{0,1} & a_{0,2} & \dots & a_{0,n-1} & b_0 \end{pmatrix}$$

Gauss Elimination Principle

- Transform the linear system into a triangular system of equations.



At the i th row, each row j below is replaced by:
 $\{\text{row } j\} + \{\text{row } i\} \cdot (-a_{j,i} / a_{i,i})$

Gauss Elimination Sequential & SIMD Algorithms

```
for (i=0; i<n-1; i++){ // for each row except the last
  for (j=i+1; j<n; j++) // step through subsequent rows
    m=a[j,i]/a[i,i]; // compute multiplier
    for (k=i; k<n; k++) {
      a[j,k]=a[j,k]-a[i,k]*m;
    }
}
```

Complexity: $O(n^3)$

```
for (i=0; i<n-1; i++){
  for (j=i+1; j<n; j++)
    m=a[j,i]/a[i,i];
    a[j,k]=a[j,k]-a[i,k]*m; (i<=k<n)
}
```

SIMD Version Complexity: $O(n^3/p)$

Other SIMD Applications

- Image Processing
- Climate and Weather Prediction
- Molecular Dynamics
- Semi-Conductor Design
- Fluid Flow Modelling
- VLSI-Design
- Database Information Retrieval
- ...

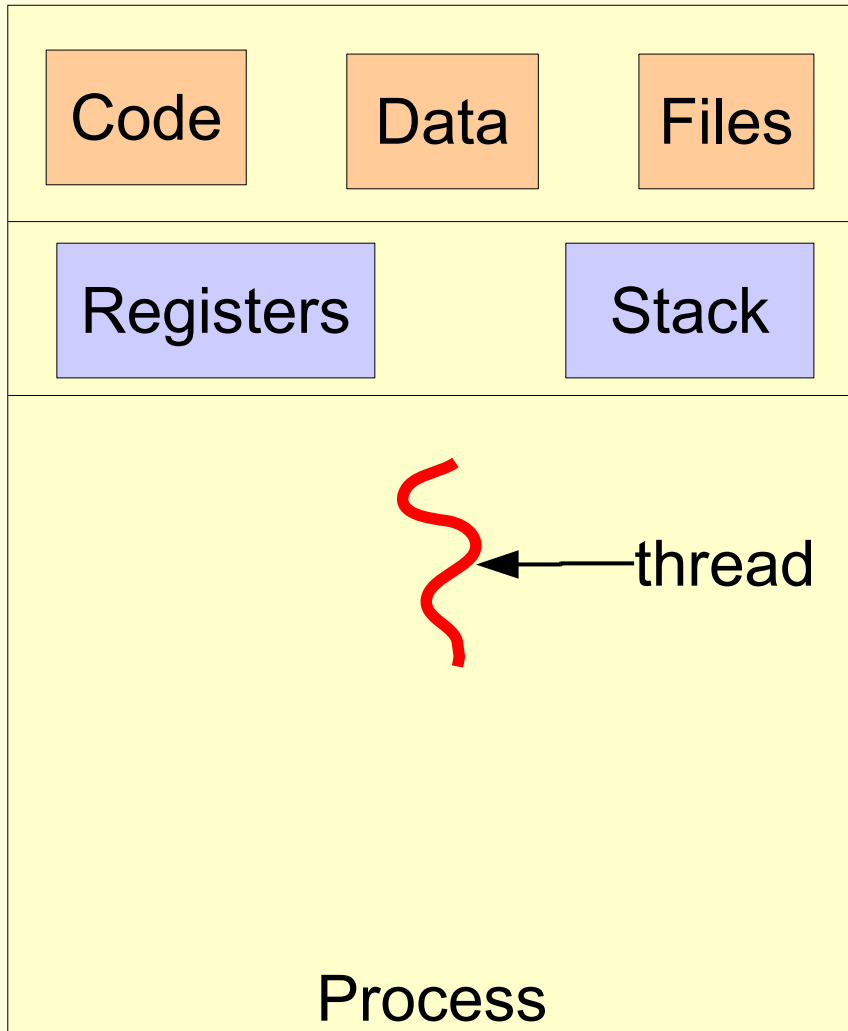
MIMD SM Programming

- ***Process, Threads & Fibers***
- ***Pre- & self-scheduling***
- ***Common Issues***
- ***OpenMP***

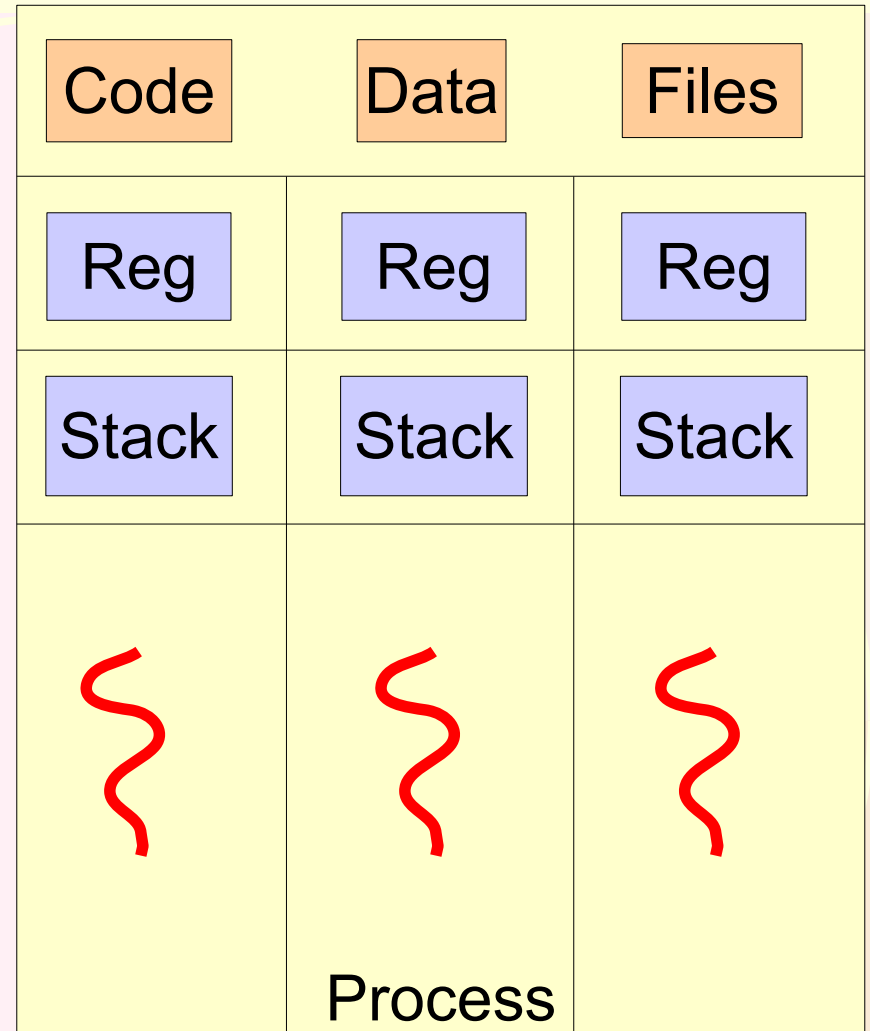
Process, Threads, Fibers

- Process: heavy weight
 - Instruction Pointer, Stacks, Registers
 - memory, file handles, sockets, device handles, and windows
- (kernel) Thread: medium weight
 - Instruction Pointer, Stacks, Registers
 - Known and scheduled preemptively by the kernel
- (user thread) Fiber: light weight
 - Instruction Pointer, Stacks, Registers
 - Unkwown by the kernel and scheduled cooperatively in user space

Multithreaded Process



Single-Threaded



Multi-Threaded

Kernel Thread

- Provided and scheduled by the kernel
 - Solaris, Linux, Mac OS X, AIX, Windows XP
 - Incompatibilities: POSIX PThread Standard
- Scheduling Question: Contention Scope
 - The time slice q can be given to:
 - the whole process (PTHREAD_PROCESS_SCOPE)
 - Each individual thread (PTHREAD_SYSTEM_SCOPE)
- Fairness?
 - With a PTHREAD_SYSTEM_SCOPE, the more threads a process has, the more CPU it has!

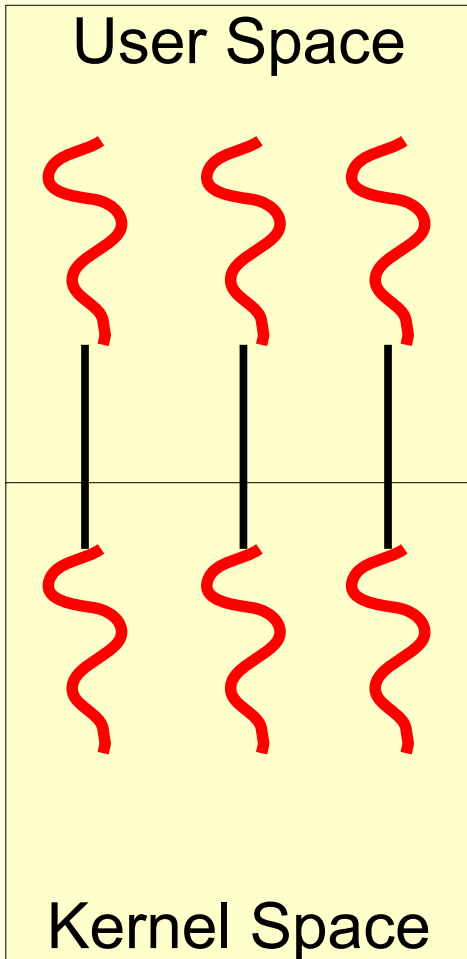
Fibers (User Threads)

- Implemented entirely in user space (also called green threads)
- Advantages:
 - Performance: no need to cross the kernel (context switch) to schedule another thread
 - Scheduler can be customized for the application
- Disadvantages:
 - Cannot directly use multi-processor
 - Any system call blocks the entire process
 - Solutions: Asynchronous I/O, Scheduler Activation
- Systems: NetBSD, Windows XP (FiberThread)

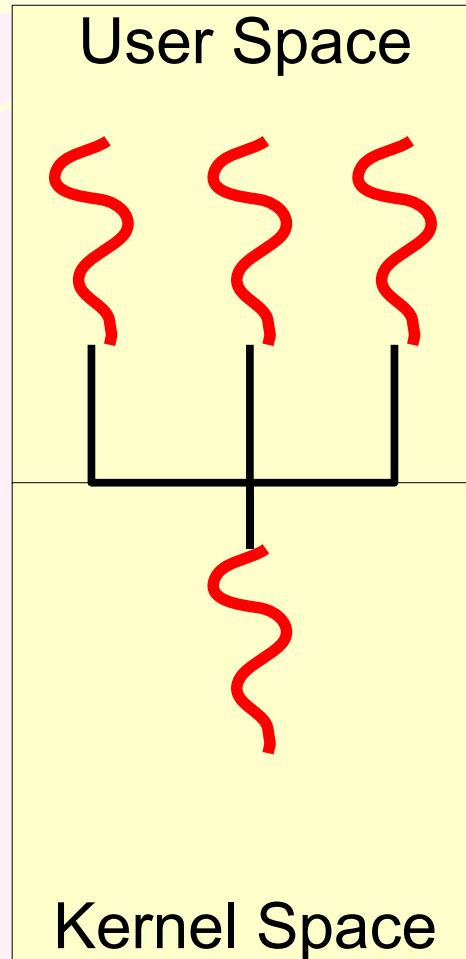
Mapping

- Threads are expressed by the application in user space.
- 1-1: any thread expressed in user space is mapped to a kernel thread
 - Linux, Windows XP, Solaris > 8
- n-1: n threads expressed in user space are mapped to one kernel thread
 - Any user-space only thread library
- n-m: n threads expressed in user space are mapped to m kernel threads ($n \gg m$)
 - AIX, Solaris < 9, IRIX, HP-UX, TRU64 UNIX

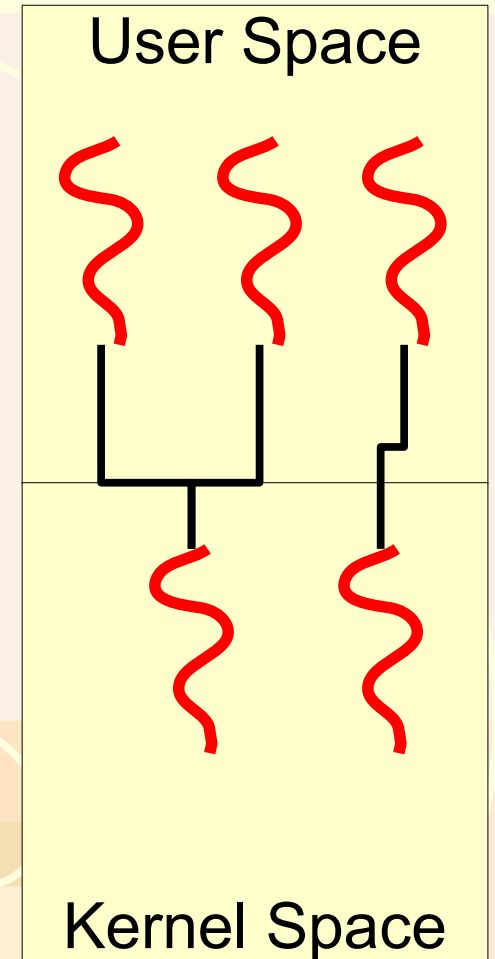
Mapping



1-1



n-1



n-m

MIMD SM Matrix Addition Self-Scheduling

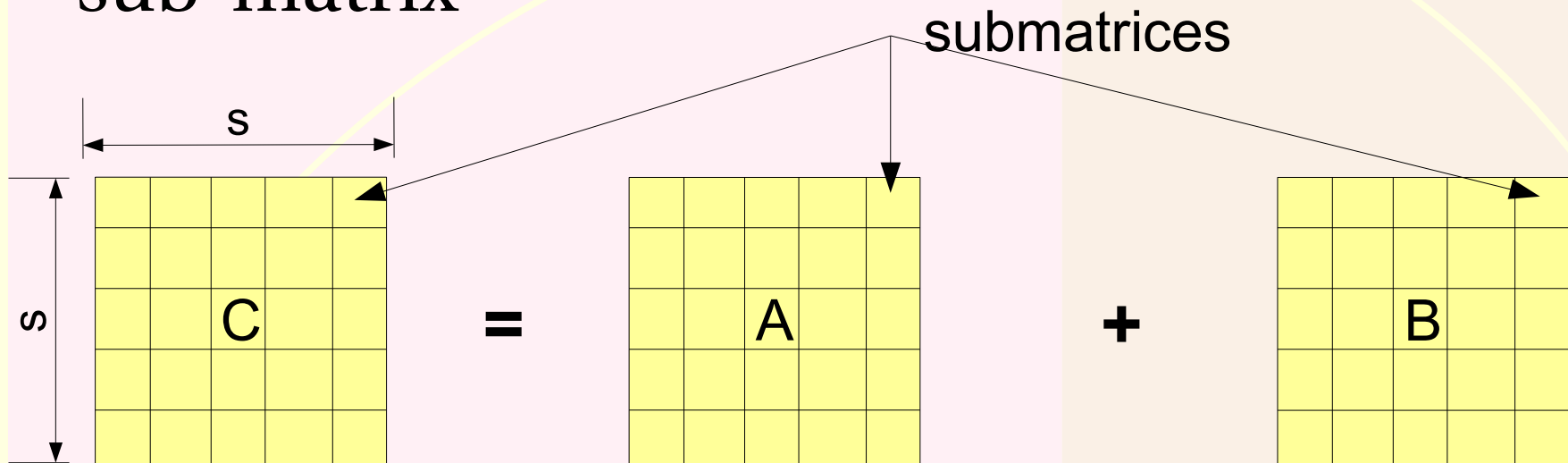
```
• private i,j,k;
  shared A[N,N], B[N,N], C[N,N], N;
  for (i = 0; i < N - 2; i++) fork DOLINE;
  // Only the Original process reach this point
  i = N-1; // Really Required?
  DOLINE: // Executed by N process for each line
  for (j = 0; j < N; j++) {
    // SIMD computation
    C[i,j]=A[i,j]+B[i,j]
  }
}
join N; // Wait for the other process
```

Complexity: $O(n^2/p)$
If SIMD is used: $O(n^2/(p.m))$
m: number of SIMD PEs

Issue: if $n > p$, self-scheduling.
The OS decides who will do what. (Dynamic)
In this case, overhead

MIMD SM Matrix Addition Pre-Scheduling

- Basic idea: divide the whole matrix by $p=s*s$ sub-matrix



Each of the p threads compute:

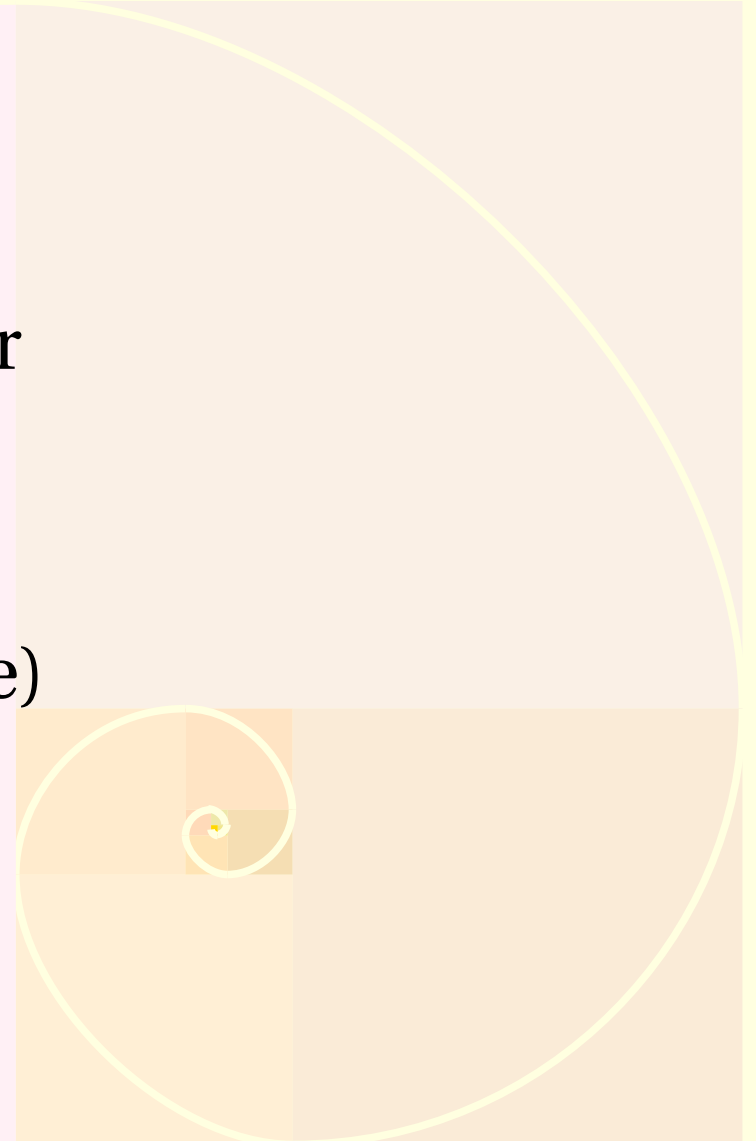
$$\square = \square + \square$$

Ideal Situation! No communication Overhead!
Pre-Scheduling: decide *statically* who will do what.

Parallel Primality Testing

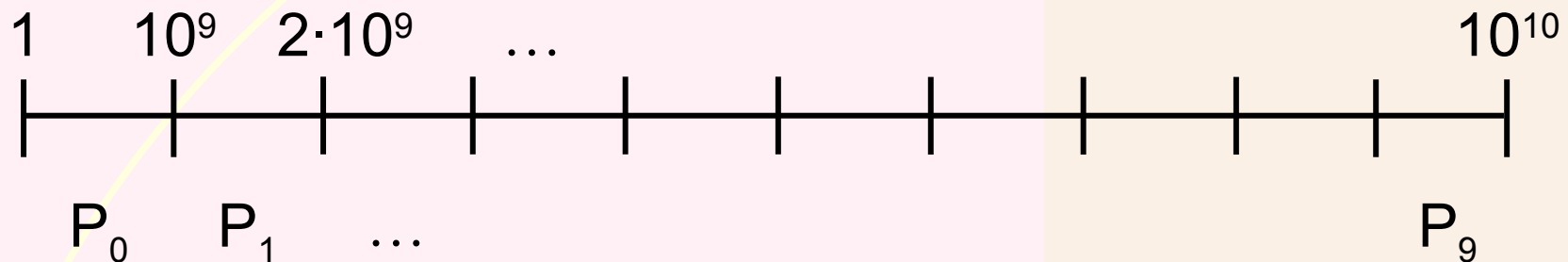
(from Herlihy and Shavit Slides)

- Challenge
 - Print primes from 1 to 10^{10}
- Given
 - Ten-processor multiprocessor
 - One thread per processor
- Goal
 - Get ten-fold speedup (or close)



Load Balancing

- Split the work evenly
- Each thread tests range of 10^9



Code for thread i :

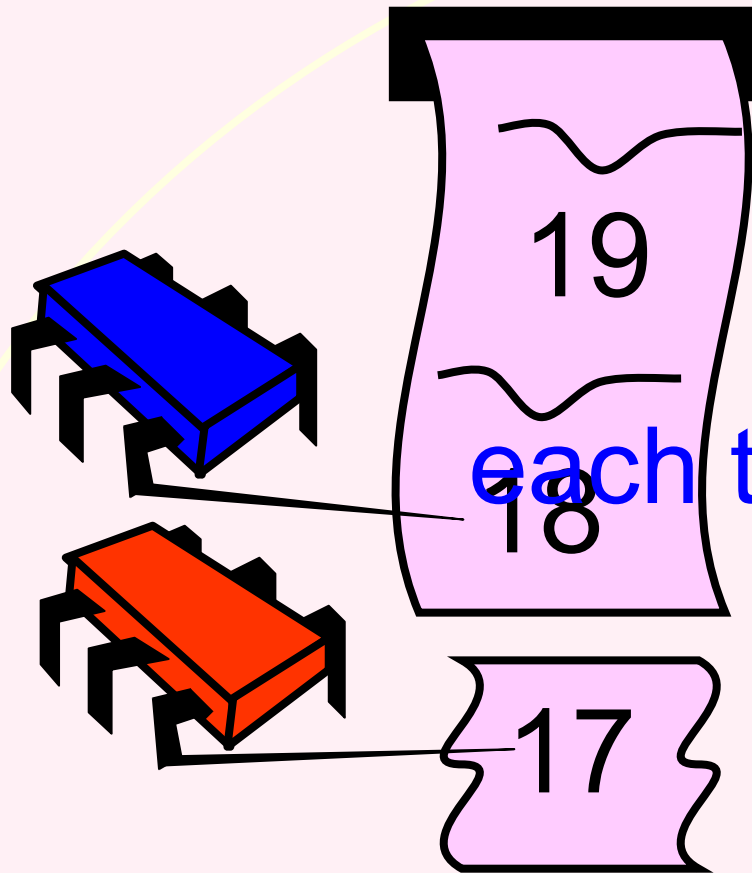
```
void thread(int i) {  
    for (j = i*109+1, j<(i+1)*109; j++) {  
        if (isPrime(j)) print(j);  
    }  
}
```

Issues

- Larger Num ranges have fewer primes
- Larger numbers harder to test
- Thread workloads
 - Uneven
 - Hard to predict
- Need *dynamic* load balancing

rejected

Shared Counter



each thread takes a number

Procedure for Thread i

```
// Global (shared) variable  
static long value = 1;  
  
void thread(int i) {  
    long j = 0;  
    while (j < 1010) {  
        j = inc();  
        if (isPrime(j)) print(j);  
    }  
}
```

Counter Implementation

```
// Global (shared) variable  
static long value = 1;  
  
long inc() {  
    return value++;  
}
```

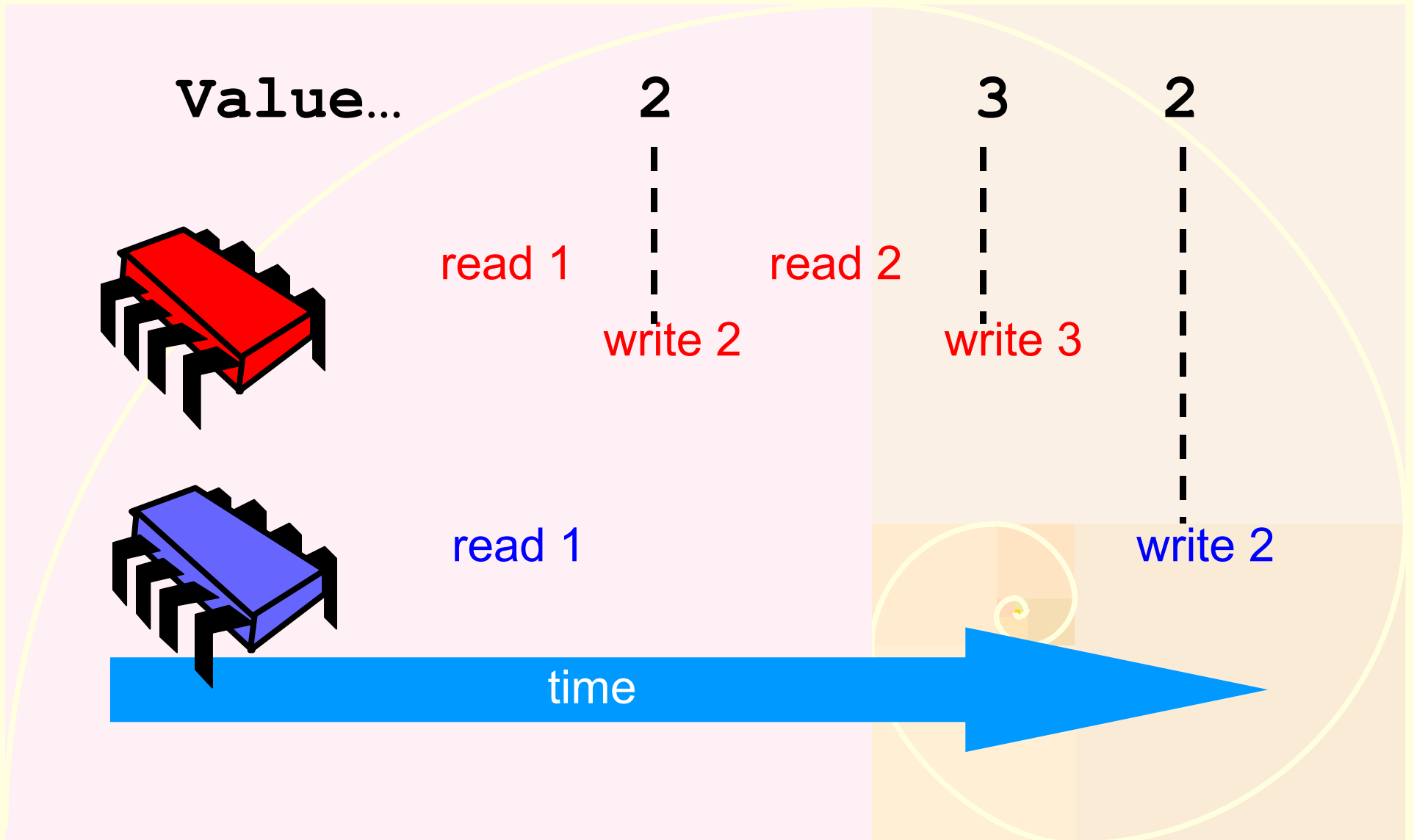
OK for uniprocessor,
not for multiprocessor

Counter Implementation

```
// Global (shared) variable  
static long value = 1;
```

```
long inc() {  
    return v; temp = value;  
}          value = value + 1;  
          return temp;
```

Uh-Oh



Challenge

```
// Global (shared) variable  
static long value = 1;  
  
long inc() {  
    temp = value;  
    value = temp + 1;  
    return temp;  
}
```

Make these steps *atomic* (indivisible)

An Aside: C/Posix Threads API

```
// Global (shared) variable
static long value = 1;
static pthread_mutex_t mutex;
// somewhere before
pthread_mutex_init(&mutex, NULL);

long inc() {
    pthread_mutex_lock(&mutex);
    temp = value;
    value = temp + 1;
    pthread_mutex_unlock(&mutex);
    return temp;
}
```

Critical section

Correctness

- Mutual Exclusion
 - Never two or more in critical section
 - This is a *safety* property
- No Lockout (lockout-freedom)
 - If someone wants in, someone gets in
 - This is a *liveness* property

Multithreading Issues

- Coordination
 - Locks, Condition Variables, Volatile
- Cancellation (interruption)
 - Asynchronous (immediate)
 - Equivalent to 'kill -KILL' but much more dangerous!!
 - What happens to lock, open files, buffers, pointers, ...?
 - Avoid it as much as you can!!
 - Deferred
 - Target thread should check periodically if it should be cancelled
 - No guarantee that it will actually stop running!

General Multithreading Guidelines

- Use good software engineering
 - Always check error status
- Always use a while(condition) loop when waiting on a condition variable
- Use a thread pool for efficiency
 - Reduce the number of created thread
 - Use thread specific data so one thread can holds its own (private) data
- The Linux kernel schedules *tasks* only,
 - Process == a task that shares nothing
 - Thread == a task that shares everything

General Multithreading Guidelines

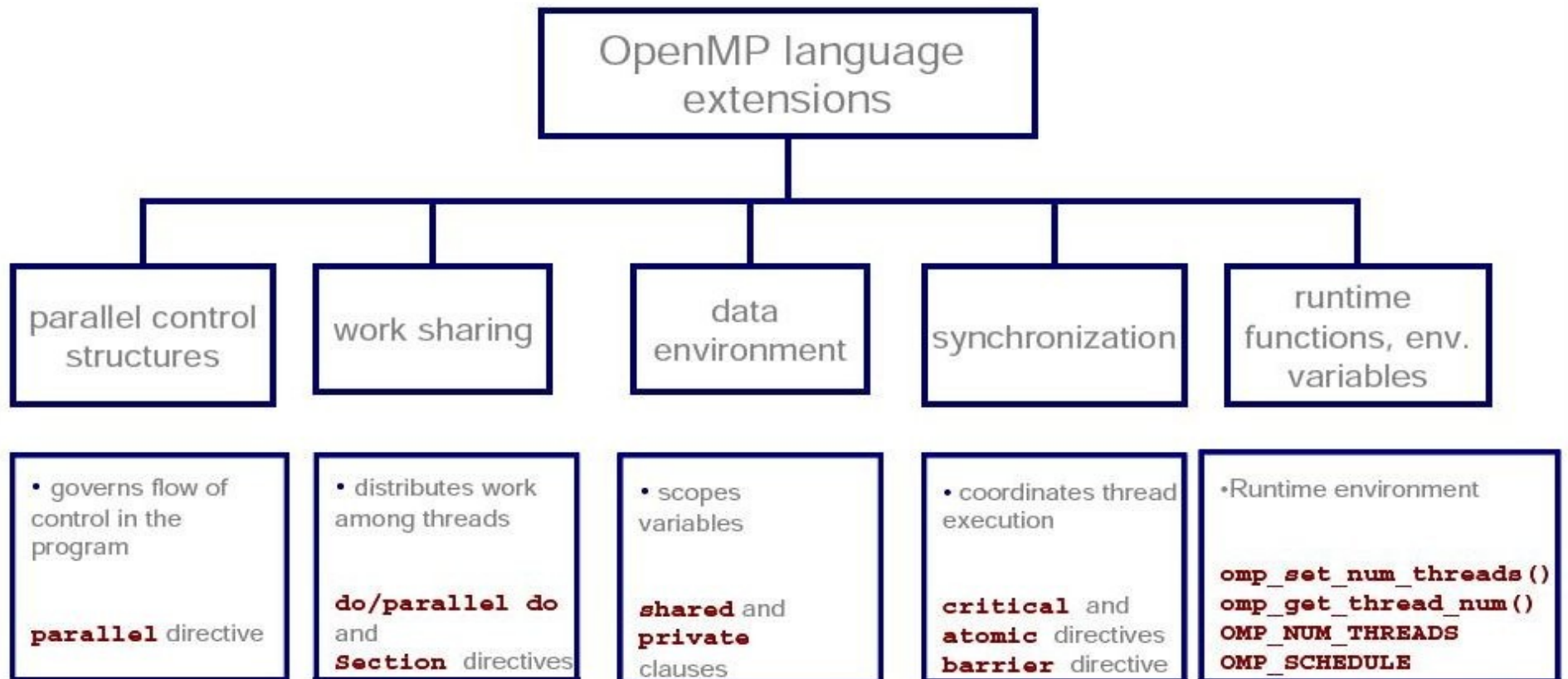
- Give the biggest priorities to I/O bounded threads
 - Usually blocked, should be responsive
- CPU-Bounded threads should usually have the lowest priority
- Use Asynchronous I/O when possible, it is usually more efficient than the combination of threads+ synchronous I/O
- Use event-driven programming to keep indeterminism as low as possible
- Use threads when you really need it!

OpenMP

- Design as superset of common languages for MIMD-SM programming (C, C++, Fortran)
 - UNIX, Windows version
- `fork()/join()` execution model
 - OpenMP programs always start with one thread
- relaxed-consistency memory model
 - each thread is allowed to have its own temporary view of the memory
- Compiler directives in C/C++ are called **pragma** (pragmatic information)
 - Preprocessor directives:
`#pragma omp <directive> [arguments]`

OpenMP

OpenMP Constructs



OpenMP

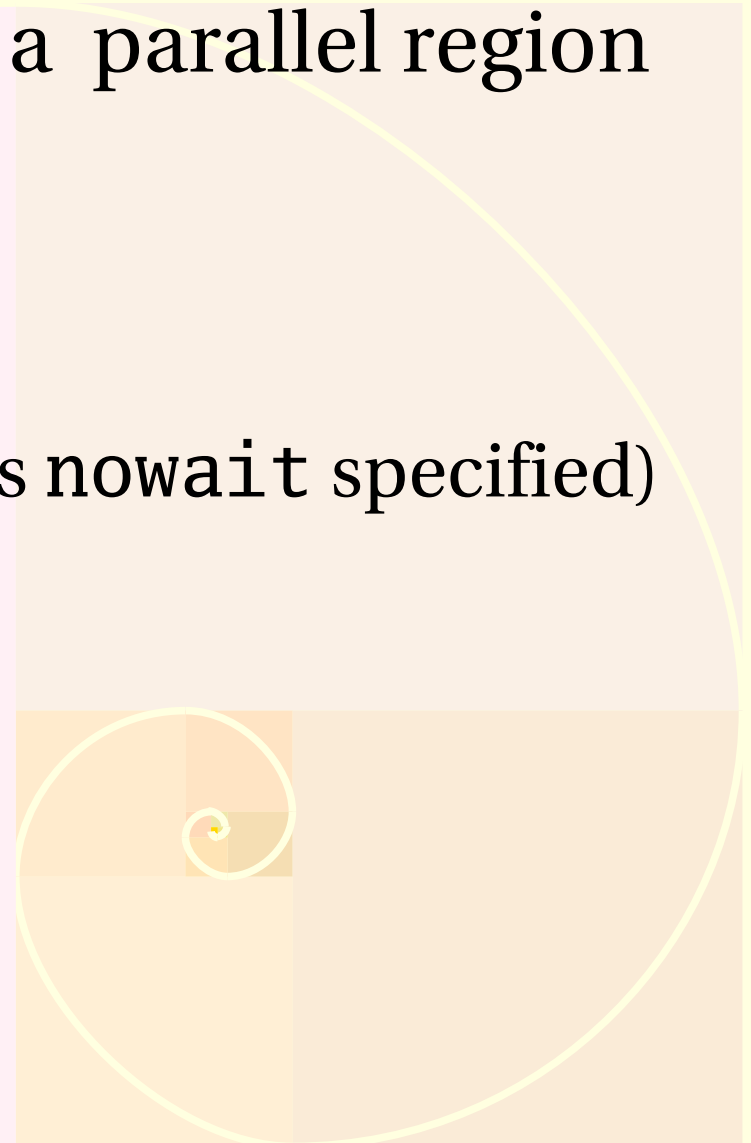
Thread Creation

- `#pragma omp parallel [arguments]`
`<C instructions> // Parallel region`
- A team of thread is created to execute the parallel region
 - the original thread becomes the master thread (thread ID = 0 for the duration of the region)
 - numbers of threads determined by
 - Arguments (if, `num_threads`)
 - Implementation (nested parallelism support, dynamic adjustment)
 - Environment variables (`OMP_NUM_THREADS`)

OpenMP

Work Sharing

- Distributes the execution of a parallel region among team members
 - Does not launch threads
 - No barrier on entry
 - Implicit barrier on exit (unless `nowait` specified)
- Work-Sharing Constructs
 - Sections
 - Loop
 - Single
 - Workshare (Fortran)



OpenMP

Work Sharing - Section

```
double e, pi, factorial, product;
int i;
e = 1;    // Compute e using Taylor expansion
factorial = 1;
for (i = 1; i < num_steps; i++) {
    factorial *= i;
    e += 1.0/factorial;
} // e done
pi = 0; // Compute pi/4 using Taylor expansion
for (i = 0; i < num_steps*10; i++) {
    pi += 1.0/(i*4.0 + 1.0);
    pi -= 1.0/(i*4.0 + 3.0);
}
pi = pi * 4.0; // pi done
return e * pi;
```

Independent Loops
Ideal Case!

OpenMP

Work Sharing - Section

```
double e, pi, factorial, product; int i;
#pragma omp parallel sections shared(e, pi) {
    #pragma omp section {
        e = 1; factorial = 1;
        for (i = 1; i < num_steps; i++) {
            factorial *= i; e += 1.0/factorial;
        }
    } /* e section */
    #pragma omp section {
        pi = 0;
        for (i = 0; i < num_steps*10; i++) {
            pi += 1.0/(i*4.0 + 1.0);
            pi -= 1.0/(i*4.0 + 3.0);
        }
        pi = pi * 4.0;
    } /* pi section */
} /* omp sections */
return e * pi;
```

Private Copies
Except for
shared variables

Independent
sections are
declared

Implicit Barrier
on exit

Talyor Example Analysis

- OpenMP Implementation: Intel Compiler
- System: Gentoo/Linux-2.6.18
- Hardware: Intel Core Duo T2400 (1.83 Ghz) Cache L2 2Mb (SmartCache)

Reason: Core Duo Architecture?

Gcc:

Standard: 13250 ms,
Optimized (-Os -msse3
-march=pentium-m
-mfpmath=sse): 7690 ms

Intel CC:

Standard: 12530 ms
Optimized (-O2 -xW
-march=pentium4): 7040 ms

OpenMP Standard: 15340 ms
OpenMP Optimized: 10460 ms

Speedup: 0.6
Efficiency: 3%

OpenMP

Work Sharing - Section

- Define a set of structured blocks that are to be divided among, and executed by, the threads in a team
- Each structured block is executed once by one of the threads in the team
- The method of scheduling the structured blocks among threads in the team is implementation defined
- There is an implicit barrier at the end of a sections construct, unless a `nowait` clause is specified

OpenMP

Work Sharing - Loop

```
// LU Matrix Decomposition
for (k = 0; k<SIZE-1; k++) {
    for (n = k; n<SIZE; n++) {
        col[n] = A[n][k];
    }
    for (n = k+1; n<SIZE; n++) {
        A[k][n] /= col[k];
    }
    for (n = k+1; n<SIZE; n++) {
        row[n] = A[k][n];
    }
    for (i = k+1; i<SIZE; i++) {
        for (j = k+1; j<SIZE; j++) {
            A[i][j] = A[i][j] - row[i] * col[j];
        }
    }
}
```

OpenMP

Work Sharing - Loop

```
// LU Matrix Decomposition
for (k = 0; k<SIZE-1; k++) {
    ... // same as before
    #pragma omp parallel for shared(A, row, col)
    for (i = k+1; i<SIZE; i++) {
        for (j = k+1; j<SIZE; j++) {
            A[i][j] = A[i][j] - row[i] * col[j];
        }
    }
}
```

Private Copies
Except for
shared variables
(Memory Model)

Implicit Barrier
at the end of a loop

Loop should be in
canonical form
(see spec)

OpenMP

Work Sharing - Loop

- Specifies that the iterations of the associated loop will be executed in parallel.
- Iterations of the loop are distributed across threads that already exist in the team
- The for directive places restrictions on the structure of the corresponding for-loop
 - Canonical form (see specifications):
 - for (i = 0; i < N; i++) {...} is ok
 - For (i = f(); g(i);) {... i++} is not ok
- There is an implicit barrier at the end of a loop construct unless a `nowait` clause is specified.

OpenMP

Work Sharing - Reduction

- `reduction(op: list)`
e.g: `reduction(*: result)`
- A private copy is made for each variable declared in 'list' for each thread of the parallel region
- A final value is produced using the operator 'op' to combine all private copies

```
#pragma omp parallel for reduction(*: res)
  for (i = 0; i < SIZE; i++) {
    res = res * a[i];
  }
}
```

OpenMP

Synchronisation -- Master

- Master directive:
`#pragma omp master`
`{...}`
- Define a section that is only to be executed by the master thread
- No implicit barrier: other threads in the team do not wait

OpenMP

Synchronisation -- Critical

- Critical directive:

```
#pragma omp critical [name]  
{...}
```
- A thread waits at the beginning of a critical region until no other thread is executing a critical region with the *same name*.
- Enforces exclusive access with respect to all critical constructs with the same name in *all threads*, not just in the current team.
- Constructs without a name are considered to have the same unspecified name.

OpenMP

Synchronisation -- Barrier

- Critical directive:

```
#pragma omp barrier  
{...}
```

- All threads of the team must execute the barrier before any are allowed to continue execution

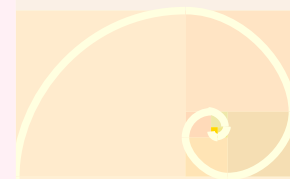
- Restrictions

- Each barrier region must be encountered by all threads in a team or by none at all.

- Barrier is not part of C!

- If ($i < n$)

```
    #pragma omp barrier  
    // Syntax error!
```



OpenMP

Synchronisation -- Atomic

- Critical directive:
`#pragma omp atomic`
`expr-stmt`
 - Where `expr-stmt` is:
`x binop= expr, x++, ++x, x--, --x` (`x` scalar, `binop` any non overloaded binary operator: `+, /, -, *, <<, >>, &, ^, |`)
- Only load and store of the object designated by `x` are atomic; evaluation of `expr` is not atomic.
- Does not enforce exclusive access of same storage location `x`.
- Some restrictions (see specifications)

OpenMP

Synchronisation -- Flush

- Critical directive:
`#pragma omp flush [(list)]`
- Makes a thread's temporary view of memory consistent with memory
- Enforces an order on the memory operations of the variables explicitly specified or implied.
- Implied flush:
 - barrier, entry and exit of parallel region, critical
 - Exit from work sharing region (unless `nowait`)
 - ...
- Warning with pointers!

MIMD DM Programming

- *Message Passing Solutions*
- *Process Creation*
- *Send/Receive*
- *MPI*

Message Passing Solutions

- How to provide Message Passing to programmers?
 - Designing a special parallel programming language
 - Sun Fortress, E language, ...
 - Extending the syntax of an existing sequential language
 - ProActive, JavaParty
 - Using a library
 - PVM, MPI, ...

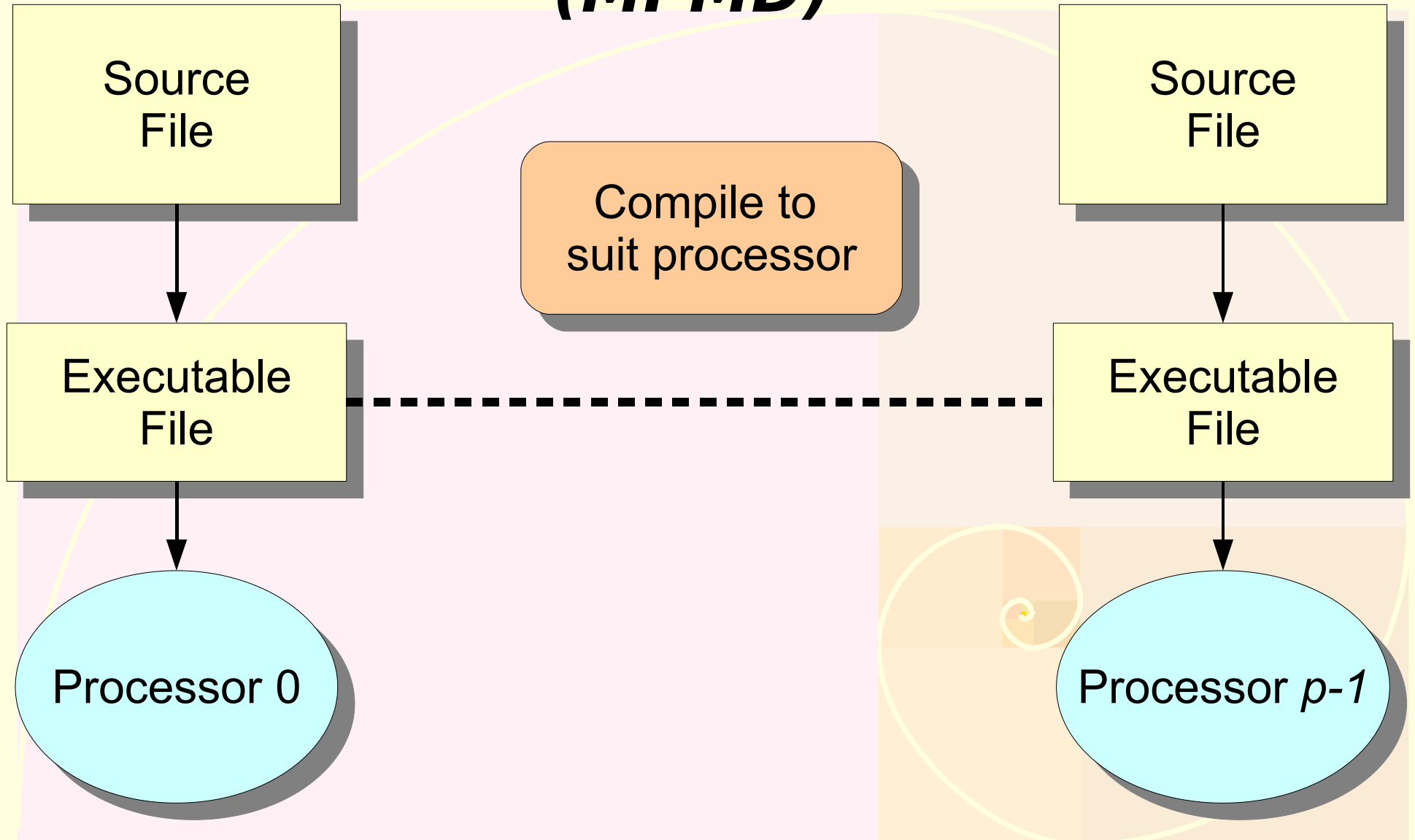


Requirements

- Process creation
 - Static: the number of processes is fixed and defined at start-up (e.g. from the command line)
 - Dynamic: process can be created and destroyed at will during the execution of the program.
- Message Passing
 - Send and receive efficient primitives
 - Blocking or Unblocking
 - Grouped Communication

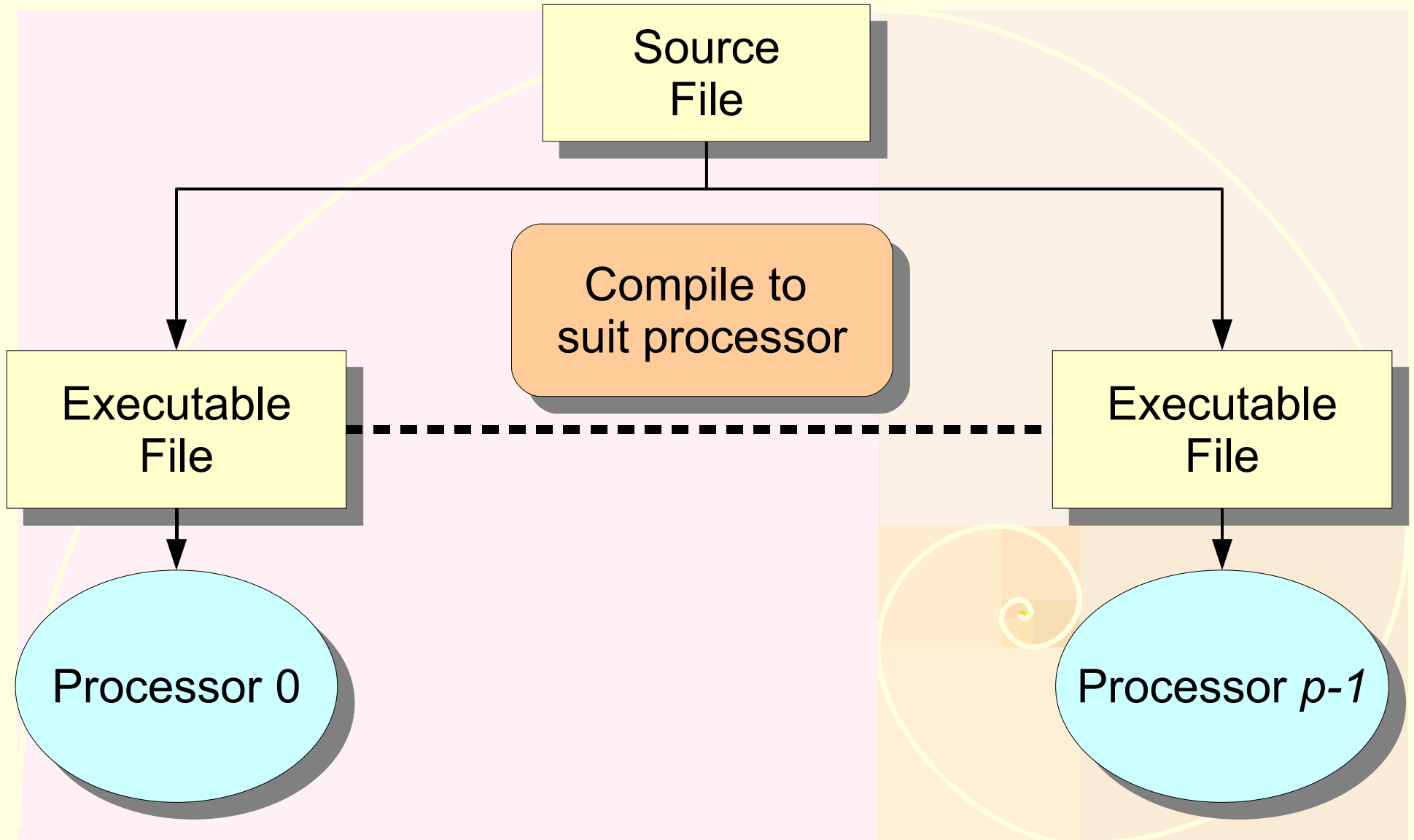
Process Creation

Multiple Program, Multiple Data Model (MPMD)



Process Creation

Single Program, Multiple Data Model (SPMD)



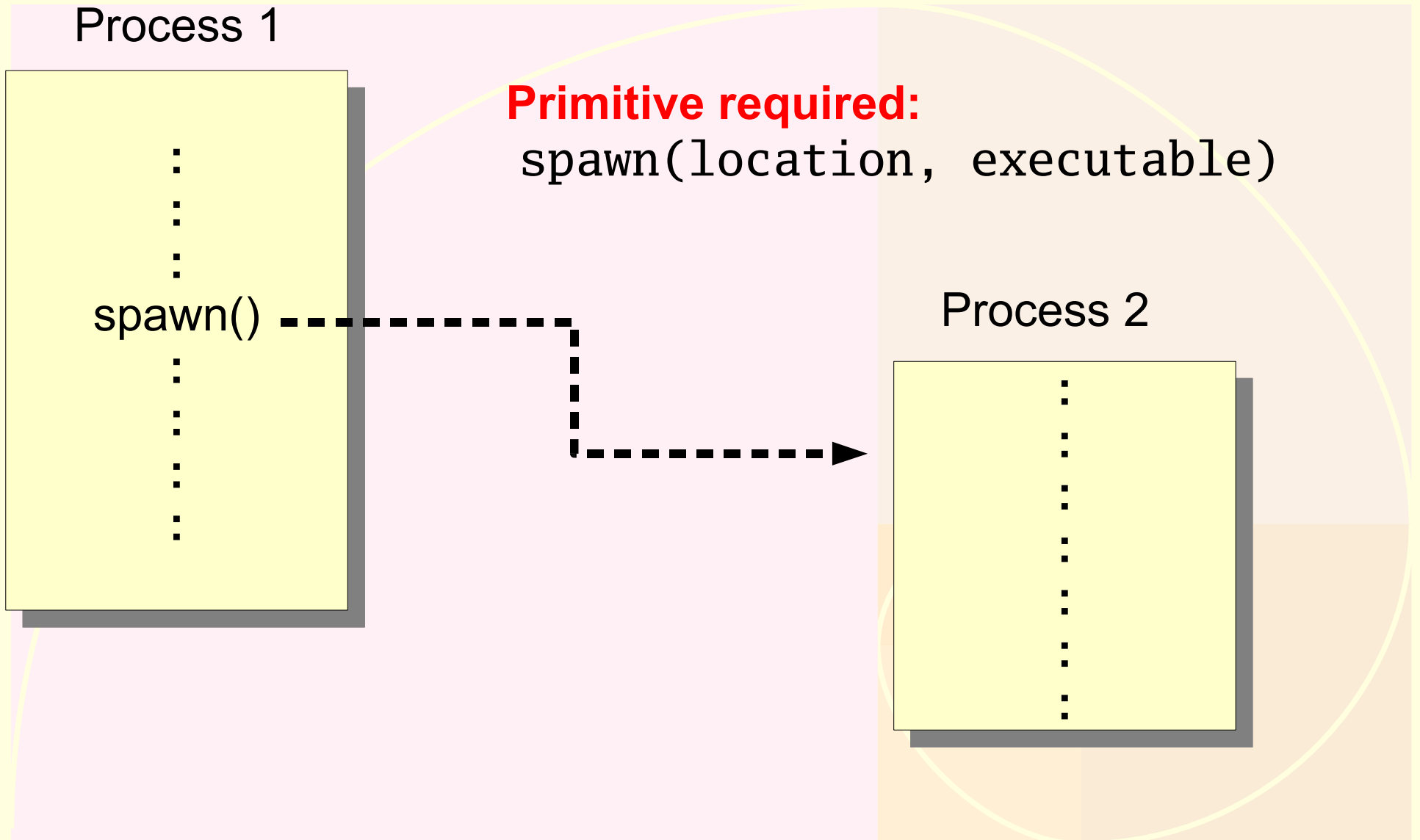
SPMD Code Sample

```
int pid = getPID();  
if (pid == MASTER_PID) {  
    execute_master_code();  
}else{  
    execute_slave_code();  
}
```

- Compilation for heterogeneous processors still required

- Master/Slave Architecture
- GetPID() is provided by the parallel system (library, language, ...)
- MASTER_PID should be well defined

Dynamic Process Creation

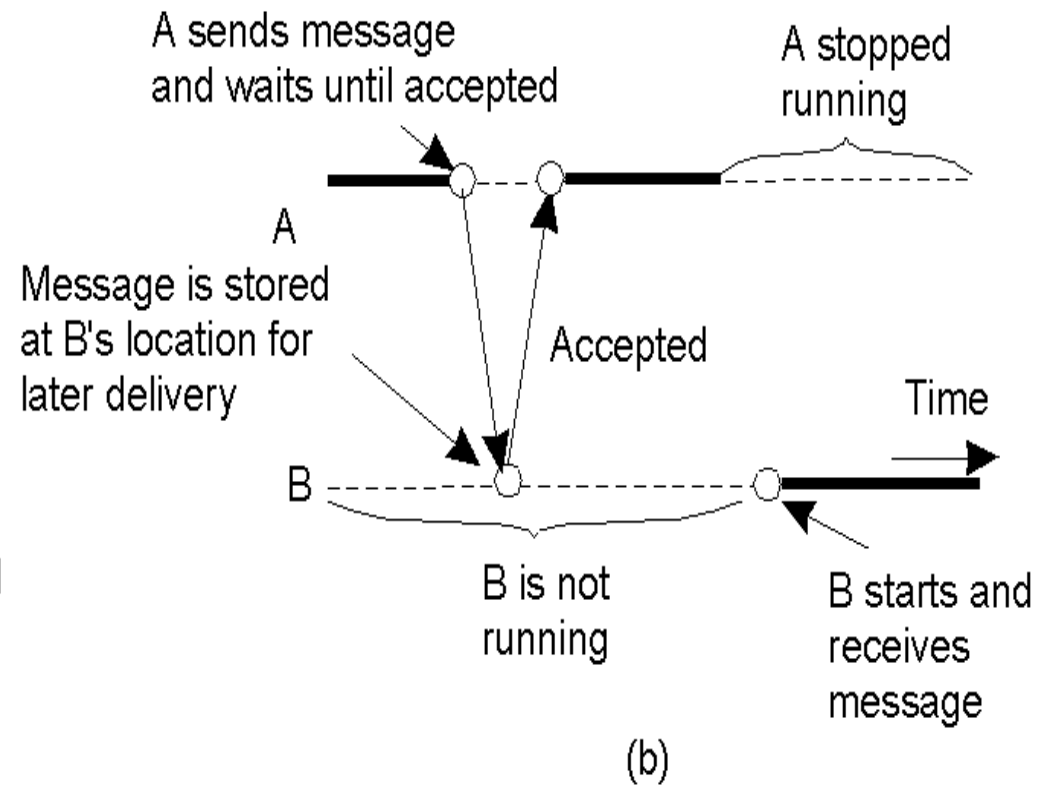
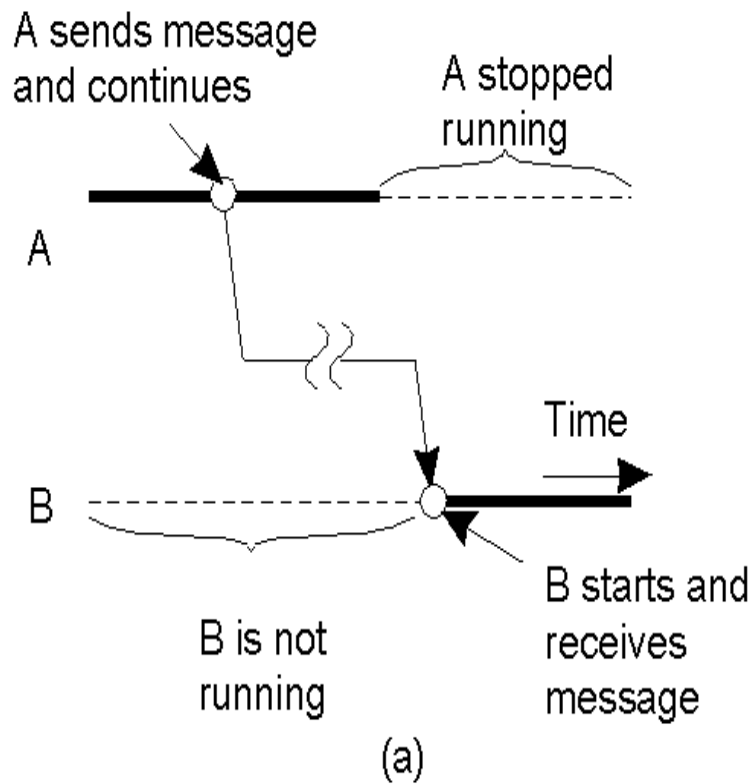


Send/Receive: Basic

- Basic primitives:
 - `send(dst, msg);`
 - `receive(src, msg);`
- Questions:
 - What is the type of: `src`, `dst`?
 - IP:port --> Low performance but portable
 - Process ID --> Implementation dependent (efficient non-IP implementation possible)
 - What is the type of: `msg`?
 - Packing/Unpacking of complex messages
 - Encoding/Decoding for heterogeneous architectures

Persistence and Synchronicity in Communication (3)

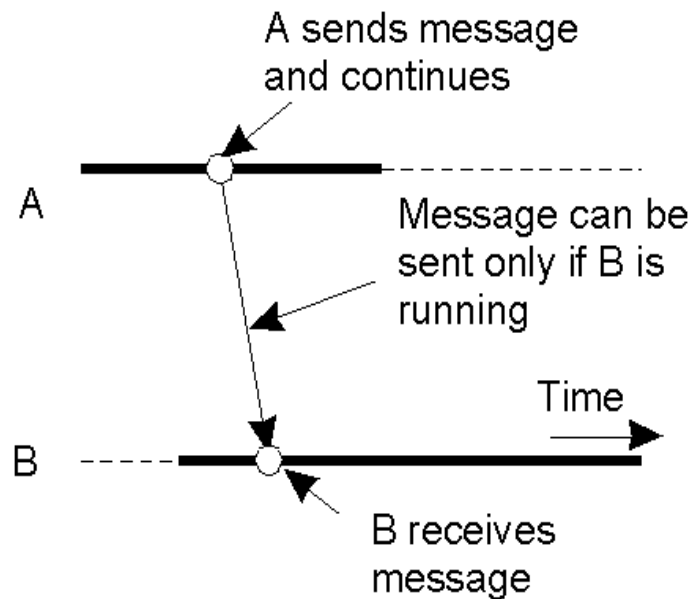
(from Distributed Systems -- Principles and Paradigms, A. Tanenbaum, M. Steen)



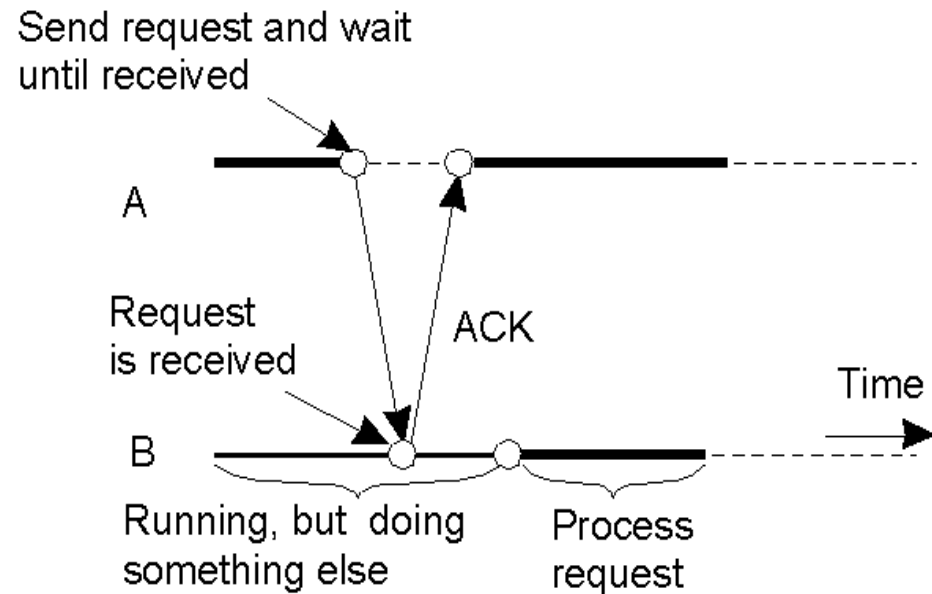
- a) Persistent asynchronous communication
- b) Persistent synchronous communication

Persistence and Synchronicity in Communication (4)

(from Distributed Systems -- Principles and Paradigms, A. Tanenbaum, M. Steen)



(c)

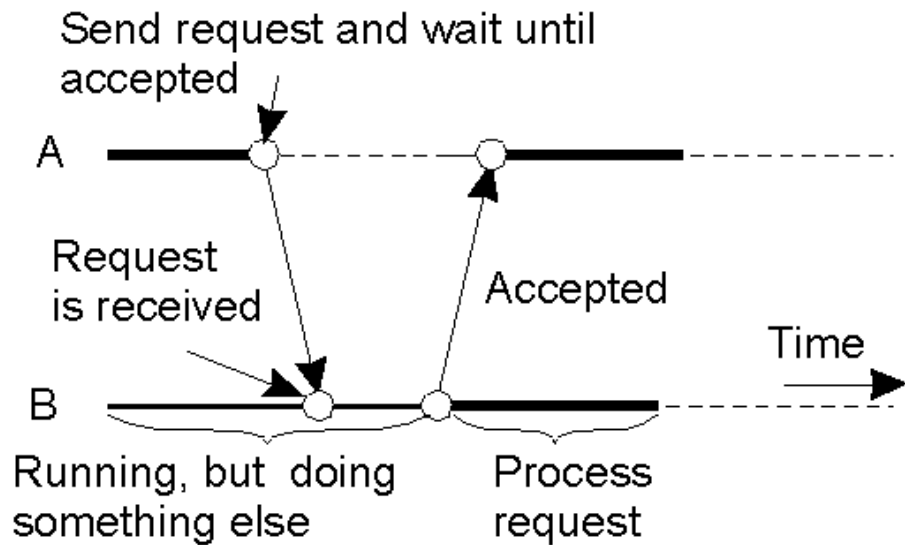


(d)

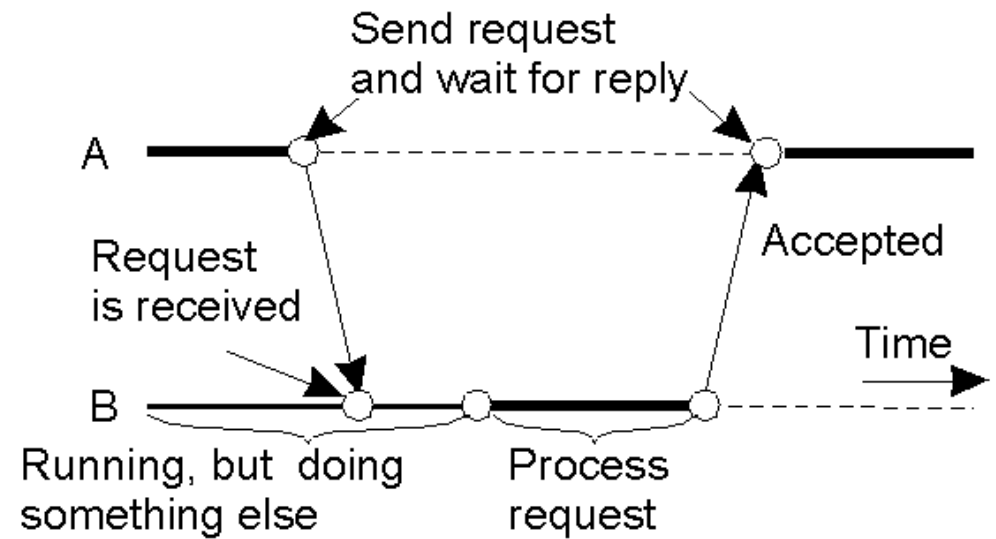
- a) Transient asynchronous communication
- b) Receipt-based transient synchronous communication

Persistence and Synchronicity in Communication (5)

(from Distributed Systems -- Principles and Paradigms, A. Tanenbaum, M. Steen)



(e)



(f)

- a) Delivery-based transient synchronous communication at message delivery
- b) Response-based transient synchronous communication

Send/Receive: Properties

Message Synchronization	System Requirements	Precedence Constraints
Send: non-blocking Receive: non-blocking	Message Buffering Failure return from receive	None, unless message is received successfully
Send: non-blocking Receive: blocking	Message Buffering Termination Detection	Actions preceding send occur before those following send
Send: blocking Receive: non-blocking	Termination Detection Failure return from receive	Actions preceding send occur before those following send
Send: blocking Receive: blocking	Termination Detection Termination Detection	Actions preceding send occur before those following send

Receive Filtering

- So far, we have the primitive:
`receive(src, msg);`
- And if one to receive a message:
 - From any source?
 - From a group of process only?
- Wildcards are used in that cases
 - A wildcard is a special value such as:
 - IP: broadcast (e.g: 192.168.255.255), multicast address
 - The API provides well defined wildcard (e.g: `MPI_ANY_SOURCE`)

The Message-Passing Interface (MPI)

(from Distributed Systems -- Principles and Paradigms, A. Tanenbaum, M. Steen)

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isead	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there are none
MPI_irecv	Check if there is an incoming message, but do not block

Some of the most intuitive message-passing primitives of MPI.

MPI Tutorial

(by William Gropp)

See William Gropp Slides

Parallel Strategies

- ***Embarrassingly Parallel Computation***
- ***Partitionning***

Embarrassingly Parallel Computations

- Problems that can be divided into different independent subtasks
 - No interaction between processes
 - No data is really shared (but copies may exist)
 - Result of each process have to be combined in some ways (reduction).
- Ideal situation!
- Master/Slave architecture
 - Master defines tasks and send them to slaves (statically or dynamically)

Embarrassingly Parallel Computations Typical Example: Image Processing

- Shifting, Scaling, Clipping, Rotating, ...
- 2 Partitioning Possibilities

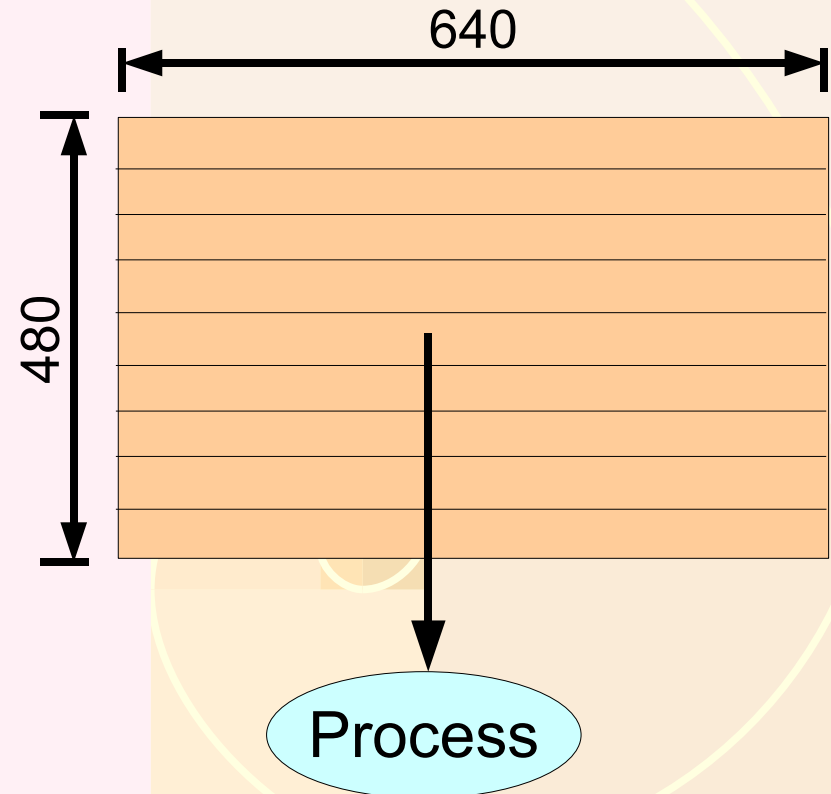
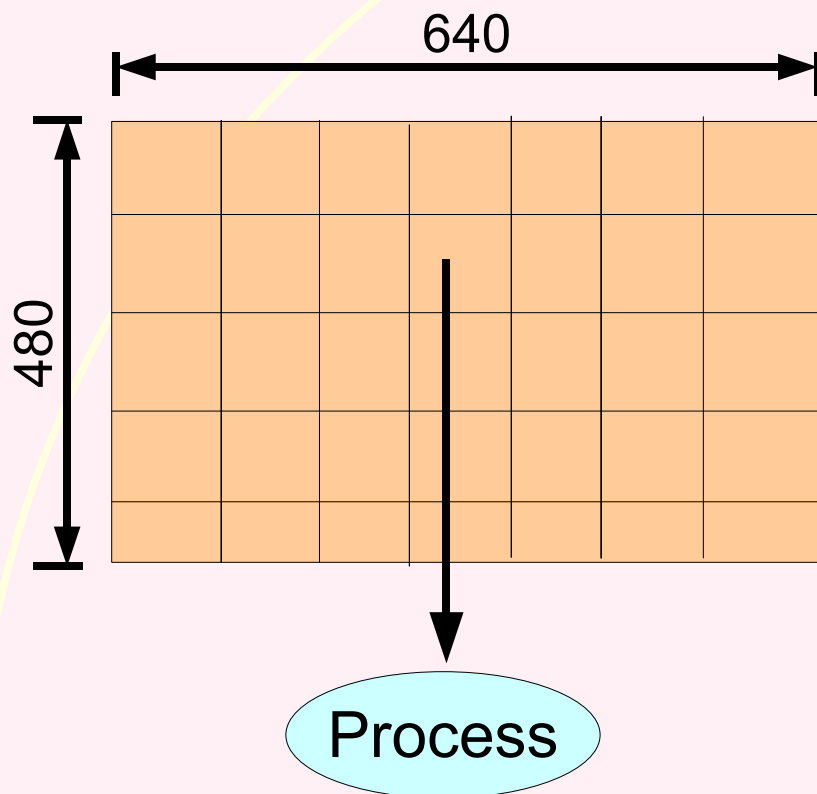


Image Shift Pseudo-Code

- Master

```
h=Height/slaves;
for(i=0,row=0;i<slaves;i++,row+=h)
    send(row, Pi);
for(i=0; i<Height*Width;i++){
    recv(oldrow, oldcol,
        newrow, newcol, PANY);
    map[newrow,newcol]=map[oldrow,oldcol];
}
```

- Slave

```
recv(row, Pmaster)
for(oldrow=row;oldrow<(row+h);oldrow++)
    for(oldcol=0;oldcol<Width;oldcol++) {
        newrow=oldrow+delta_x;
        newcol=oldcol+delta_y;
        send(oldrow,oldcol,newrow,newcol,Pmaster);
    }
```

Image Shift Pseudo-Code Complexity

- Hypothesis:
 - 2 computational steps/pixel
 - $n*n$ pixels
 - p processes
- Sequential: $T_s = 2n^2 = O(n^2)$
- Parallel: $T_{//} = O(p+n^2) + O(n^2/p) = O(n^2)$ (p fixed)
 - Communication: $T_{\text{comm}} = T_{\text{startup}} + mT_{\text{data}}$
 $= p(T_{\text{startup}} + 1 * T_{\text{data}}) + n^2(T_{\text{startup}} + 4T_{\text{data}}) = O(p+n^2)$
 - Computation: $T_{\text{comp}} = 2(n^2/p) = O(n^2/p)$

Image Shift Pseudo-Code Speedup

- Speedup $\frac{T_s}{T_p} = \frac{2n^2}{2n^2/p + p(T_{startup} + T_{data}) + n^2(T_{startup} + 4T_{data})}$

- Computation/Communication:

$$\frac{2n^2/p}{p(T_{startup} + T_{data}) + n^2(T_{startup} + 4T_{data})} = O\left(\frac{n^2/p}{p + n^2}\right)$$

- Constant when p fixed and n grows!

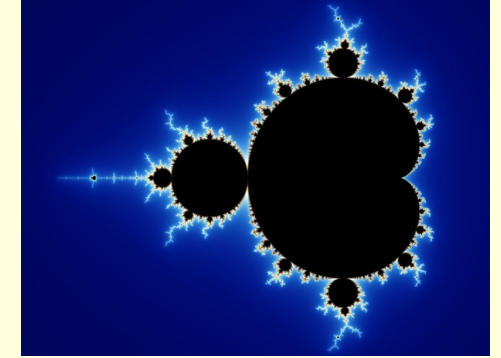
- Not good: the largest, the better

- Computation should hide the communication

- Broadcasting is better

- This problem is for Shared Memory!

Computing the Mandelbrot Set



• Problem: we consider the limit of the following sequence in the complex plane:

$$z_0 = 0$$

$$z_{k+1} = z_k^2 + c$$

- The constant $c = a + bi$ is given by the point (a, b) in the complex plane
- We compute z_k for $0 < k < n$
 - if for a given k , $(|z_k| > 2)$ we know the sequence diverge to infinity – we plot c in color(k)
 - if for $(k == n)$, $|z_k| < 2$ we *assume* the sequence converge we plot c in a black color.

Mandelbrot Set: sequential code

```
int calculate(double cr, double ci) {
    double bx = cr, by = ci;
    double xsq, ysq;
    int cnt = 0;

    while (true) {
        xsq = bx * bx;
        ysq = by * by;
        if (xsq + ysq >= 4.0) break;
        by = (2 * bx * by) + ci;
        bx = xsq - ysq + cr;
        cnt++;
        if (cnt >= max_iterations) break;
    }

    return cnt;
}
```

Computes the number of iterations for the given complex point (cr, ci)

Mandelbrot Set: sequential code

```
y = startDoubleY;
for (int j = 0; j < height; j++) {
    double x = startDoubleX;
    int offset = j * width;
    for (int i = 0; i < width; i++) {
        int iter = calculate(x, y);
        pixels[i + offset] = colors[iter];
        x += dx;
    }
    y -= dy;
}
```

Mandelbrot Set: parallelization

- Static task assignment:
 - Cut the image into p areas, and assign them to each p processor
 - Problem: some areas are easier to compute (less iterations)
 - Assign n^2/p pixels per processor in a round robin or random manner
 - On average, the number of iterations per processor should be approximately the same
 - Left as an exercise

Mandelbrot set: load balancing

- Number of iterations per pixel varies
 - Performance of processors may also vary
- Dynamic Load Balancing
 - We want each CPU to be 100% busy ideally
- Approach: *work-pool*, a collection of tasks
 - Sometimes, processes can add new tasks into the work-pool
 - Problems:
 - How to define a task to minimize the communication overhead?
 - When a task should be given to processes to increase the computation/communication ratio?

Mandelbrot set: dynamic load balancing

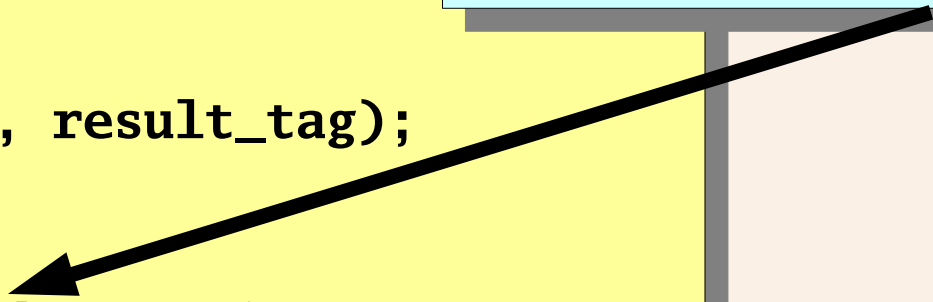
- Task == pixel
 - Lot of communications, Good load balancing
- Task == row
 - Fewer communications, less load balanced
- Send a task on request
 - Simple, low communication/computation ratio
- Send a task in advance
 - Complex to implement, good communication/computation ratio
- Problem: Termination Detection?

Mandelbrot Set: parallel code

Master Code

```
count = row = 0;
for(k = 0, k < num_procs; k++) {
    send(row, Pk, data_tag);
    count++, row++;
}
do {
    recv(r, pixels, PANY, result_tag);
    count--;
    if (row < HEIGHT) {
        send(row, Pslave, data_tag);
        row++, count++;
    } else send(row, Pslave, terminator_tag);
    display(r, pixels);
} while(count > 0);
```

Pslave is the process which sends the last received message.



Mandelbrot Set: parallel code

Slave Code

```
recv(row, PMASTER, ANYTAG, source_tag);
while(source_tag == data_tag) {
    double y = startDoubleYFrom(row);
    double x = startDoubleX;
    for (int i = 0; i < WIDTH; i++) {
        int iter = calculate(x, y);
        pixels[i] = colors[iter];
        x += dx;
    }
    send(row, pixels, PMASTER, result_tag);
    recv(row, PMASTER, ANYTAG, source_tag);
};
```

Mandelbrot Set: parallel code Analysis

$$T_s \leq \text{Max}_{iter} \cdot n^2$$

$$T_p = T_{comm} + T_{comp}$$

$$T_{comm} = n(t_{startup} + 2t_{data}) + (p-1)(t_{startup} + t_{data}) + n(t_{startup} + nt_{data})$$

$$T_{comp} \leq \frac{n^2}{p} \text{Max}_{iter}$$

$$\frac{T_s}{T_p} = \frac{\text{Max}_{iter} \cdot n^2}{p \cdot \text{Max}_{iter} \cdot n^2 + n^2 t_{data} + (2n + p - 1)(t_{startup} + t_{data})}$$

$$\frac{T_{comp}}{T_{comm}} = \frac{\text{Max}_{iter} \cdot n^2}{n^2 t_{data} + (2n + p - 1)(t_{startup} + t_{data})}$$

Only valid when
all pixels are black!
(Inequality of
 T_s and T_p)

- Speedup approaches p when Max_{iter} is high
- Computation/Communication is in $O(\text{Max}_{iter})$

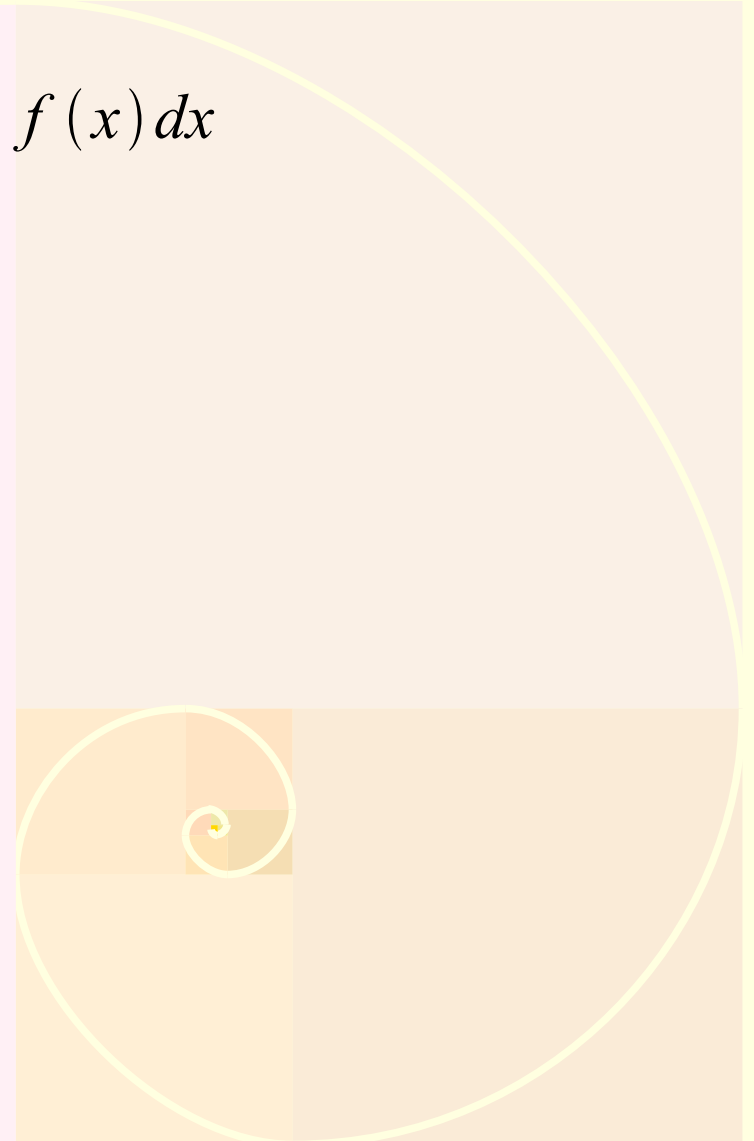
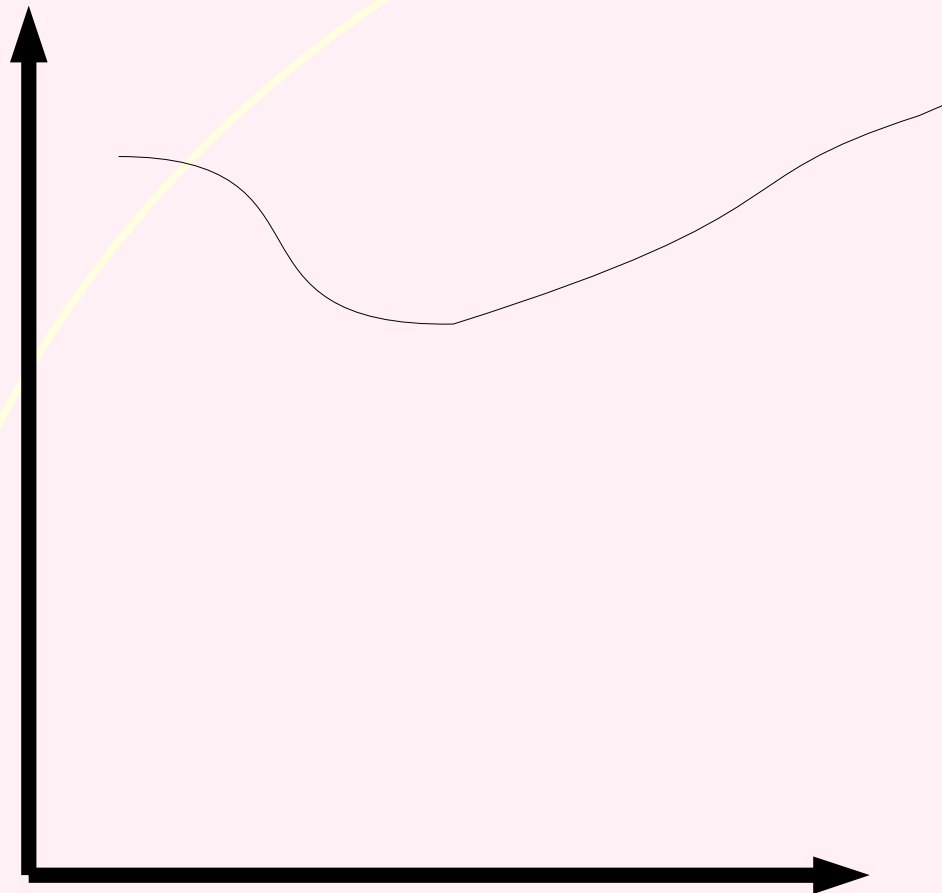
Partitioning

- Basis of parallel programming
- Steps:
 - cut the initial problem into smaller part
 - solve the smaller part in parallel
 - combine the results into one
- Two ways
 - Data partitioning
 - Functional partitioning
- Special case: divide and conquer
 - subproblems are of the same form as the larger problem

Partitioning Example: numerical integration

- We want to compute:

$$\int_a^b f(x) dx$$



Quizz



Readings

- Describe and compare the following mechanisms for initiating concurrency:
 - Fork/Join
 - Asynchronous Method Invocation with futures
 - Cobegin
 - Forall
 - Aggregate
 - Life Routine (*aka* Active Objects)