ID1217 Concurrent Programming
Lecture 1

Introduction.
Parallel Programming Concepts, Models and Paradigms

Vladimir Vlassov
KTH/ICT/ECS
VT 2008

ID1217 Course Info

- Given by the Dept of Electronic, Computer and Software Systems (ECS), ICT School, KTH
- URL: http://www.imit.kth.se/courses/ID1217/
- E-mail addresses:
  - ID1217-teachers@ict.kth.se – teachers
  - ID1217-students@ict.kth.se – students
  - ID1217@ict.kth.se – students and teachers

ID1217 Course Staff

- Vladimir Vlassov, Associate professor, course responsible
  - Phone 08-790 4115
  - E-mail: vladv@kth.se
  - Consultation: Wednesday13:00-14:00, Forum, 8th floor (elevator C), Kista; or by appointment (via email)
- Leif Lindbäck, Assistant professor, teaching assistant
  - Phone: 08-790 4425
  - Email: leifl@kth.se
  - Consultation by appointment (send email)
- Gunnar Johansson, secretary, grades administration
  - Phone 08-790 4102
  - E-mail: gunnarj@imit.kth.se

Course Material and Course Layout

- Material
  - Book: ***Foundations of Multithreaded, Parallel and Distributed Programming***, by G. R. Andrews, Addison Wesley, 2000
  - Lecture notes and Lab PM
  - URL: http://www.imit.kth.se/courses/ID1217/
- Layout
  - Lectures (18)
  - Programming assignments:
    - Programming project (1)
    - Homework (5) – 4 out of 5 is required to pass the course
  - Examination requirements:
    - Written exam – 4,5 hp, 5 hours
    - Assignments reports – 3 hp

Bonus Policy

- A report on a programming project turned in before or on the specified deadline, gives at most **10% bonus credit points** to the ID1217 exams in this academic year if the report is accepted.
- Each homework turned in before or on the date the HW is due, gives at most **3% bonus credit points** to the ID1217 exams in in this academic year if the homework solution is accepted.
- A student can collect up to $(10 + 3 \times 5) = 25\%$ bonus credit points.

Content of the Course

- Introduction. Parallel programming concepts, models, and paradigms
- Shared memory programming
  - Processes and synchronization;
  - Introduction to the formal semantics of concurrent programs with shared variables;
  - Synchronization mechanisms: locks, flags, barriers, condition variables, semaphores, monitors;
  - Implementation of processes and synchronization.
- Distributed memory programming
  - Message passing, RPC and rendezvous, RMI;
  - Paradigms for process interaction.
- Parallelism in scientific computing
- Overview of parallel programming environments

Outline (today). Ch.1 of the text

- Introduction
- Parallel programming concepts: task, process, state, etc.
- Parallel programming models
  - Shared address space (a.k.a. shared memory) programming model
  - Message passing (a.k.a. dist. memory) programming model
- Parallel programming basic paradigms
  - Iterative parallelism
  - Recursive parallelism
  - Producers and consumers (pipelines)
  - Clients and servers
  - Interacting peers

Sequential Program Versus Parallel Program

- Sequential program
  - Represents a sequence of actions that produces a result
  - Contains a single thread of control
  - Is called a ***task***
    - The task is executed by a ***process***
- Parallel (concurrent, distributed) program
  - Contains two or more processes that cooperate in solving a problem
    - Each process executes a sequential program and has its own state
    - Processes communicate via shared memory or/and by message passing
  - Can be executed
    - Concurrently on a single-processor machine or
    - In parallel on a multiprocessor or on several computers.

Programming with Processes

- ***Concurrent Programming***
  - Several concurrent processes (threads) share a single CPU
  - Shared memory programming model
  - Motivations:
    - Improving performance, utilization and scalability
    - Modeling of concurrency in the (real) world
- ***Distributed Programming***
  - Processes distributed among computers communicate over network
  - Message passing programming model
  - For distributed computing: Users, data, machines are physically distributed
  - For high performance (scalable) computing
- ***Parallel Programming***
  - Parallel (concurrent) processes execute on their own processors
  - Both, shared memory and message passing programming models
  - For high performance computing : solve a problem faster or/and solve a larger problem

Speedup

- Major goal of applications in using a parallel machine is ***speedup which is offered*** by the parallel machine:

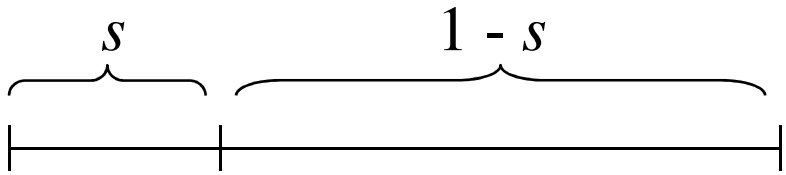
$$\text{Speedup } (n \text{ processors}) = \frac{\text{Performance } (n \text{ processors})}{\text{Performance } (1 \text{ processor})}$$

- Major goal of applications in being executed in parallel is ***speedup which can be achieved*** due to parallel execution:
  - for a fixed problem size (input data set), performance = 1/time:

$$\text{Speedup}_{\text{ fixed problem size}} (n \text{ processors}) = \frac{\text{Execution time } (1 \text{ processor})}{\text{Execution time } (n \text{ processors})}$$

Limited Concurrency

- ***Amdahl's Law***: If fraction *s* of sequential execution is inherently serial, then speedup ≤ 1/*s*, i.e. the speedup is limited by 1/*s* when $n \rightarrow \infty$



$$\lim_{n\to\infty} \frac{T}{\frac{T\cdot(1-s)}{n} + T\cdot s} = \frac{1}{s}$$

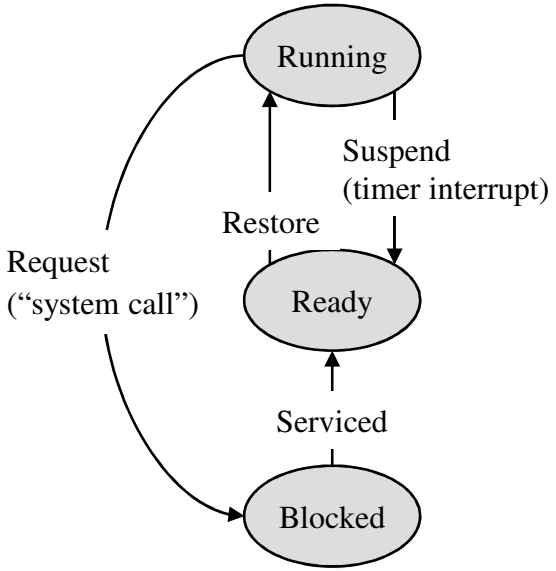
- Impediments to speedup
  - Inherently sequential parts (Amdahl's law) – unavoidable
  - Load imbalance
  - Synchronization and context switch overheads: critical sections, delays, fork/join

Hardware

- For multiprogramming and concurrent programming with shared memory
  - A single-CPU processor
- For concurrent and parallel programming with shared memory
  - Shared-memory multiprocessor
    - Centralized shared memory (UMA)
    - Distributed shared memory (NUMA)
- For distributed and parallel programming with message passing
  - Distributed memory multiprocessor
  - Computer clusters
  - Computer networks
- For distributed and parallel programming with message passing and sh.memory
  - ***Hierarchical multiprocessor***: distributed memory MP with SMP nodes

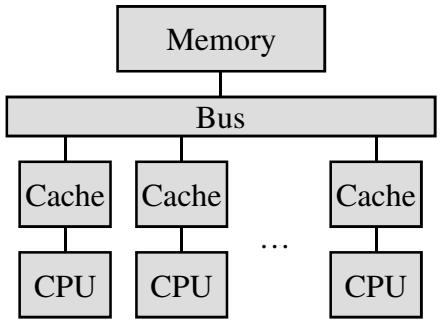
Multiprogramming on a Single Processor

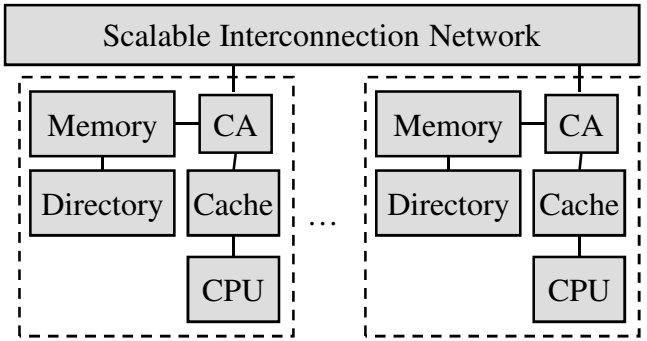
- The CPU time is shared between several programs by ***time-slicing***
- OS controls and schedules processes
  - If a time slice period has expired or the process blocks for some reason (e.g. IO operation, blocking synchronization operation), OS makes a process (context) switch
- ***Process switching***: Suspend the current process and restore a new process:
  1. Save the current process state to the memory;
     Add the process to the tail of the ready queue
     or to a wait queue;
  2. Take a process from the head of the ready queue;
     Restore the proc state and make it running.
- Resume a blocked process:
  - Move a process from the wait queue
    to the ready queue.



Shared-Memory Multiprocessors

- A shared-memory MP offers a shared address space
  - Processes executing on different processors have access to a common (shared) memory
- ***Symmetric multiprocessors*** (SMP, UMA)
  - Centralized shared memory
  - Bus
  - Snoopy cache coherence protocols
- ***Distributed shared memory MP*** (NUMA)
  - Shared memory is distributed among nodes
  - Scalable interconnection network
  - Directory-based cache coherency protocols





Multi-Core Processors

- Combines several (two or more) independent cores into a single package on the same die.
  - Tightly-coupled multiprocessor
  - May share L2 cache or have separate caches
  - Shared the same interconnect to the rest of the system (memory)
- TLP – thread-level-parallelism on the chip
- Important R&D areas (by Erik Hagersten, Uppsala Univerisy)
  - Algorithms
  - Parallelization
  - Verification
  - Modeling/Simulation/Tools
  - Bandwidth optimizations
  - Managing data locality
  - …
- "There are no indications auto parallelization will radically improve any time soon." – Tim Mattson, Intel

Distributed Memory Multiprocessors and Multi-Computers

- There is no HW-supported shared memory
  - Each node has its own local memory
  - Processes on different nodes communicates by message passing
  - Major problem is the communication latency
- ***Massively parallel processors (MPP)*** and ***computer clusters***
  - Tightly coupled computers connected with a high-speed interconnection network
- ***Computer networks***
  - Loosely coupled computers (LAN or WAN)

Parallel Programming Concepts

- ***Task*** – an arbitrary piece of un-decomposed work in parallel or sequential computation
  - **Executed sequentially**; concurrency is only across tasks
  - Fine-grained versus coarse-grained tasks
- ***Process (thread)*** – an abstract entity that performs tasks assigned to it
  - Each process has its state and a unique ID
  - Processes communicate and synchronize to perform their tasks via shared memory or/and by message passing
  - Three types of processes:
    - Heavy-weight processes
    - Light-weight processes
    - Threads

Parallel Programming Concepts (cont'd)

- A ***process state (context)*** includes all information needed to execute the process:
  - In CPU: contents of PC, NPC, PSW, SP, registers
  - In memory: contents of text segment, data segment, heap, stack
    - Can be cached in data and instruction caches
    - A portion (but not stack) can be shared with other processes
- A ***context switch*** – terminate or suspend the current process and start or resume another process
    - Performed by an OS kernel
    - The CPU state of the current process must be saved to the memory
    - "Dirty" cache lines can be written back to memory on replacement

Parallel Programming Concepts (cont'd)

- ***Processor*** – a physical engine on which processes execute
  - Processes virtualize machine to programmer
    - First write program in term of processes, then map to processors
- ***Concurrent program (execution)*** is formed of several processes (threads) that share a (multi)processor
    - Usually more processes than processors
- ***Parallel program (execution)*** is formed of several processes each executes on its own processor
    - Usually more processors than processes
- ***Distributed program (execution)*** is formed of processes that are distributed among processors and communicate over network
    - No shared memory

Heavy-Weight Processes (HWP)

- A ***heavy-weight process*** has its own virtual address space not shared with other HWPs
  - Entire process state is private
  - A HW process can be multithreaded
  - A HWP can create another HWP (a copy of itself or a different process)
- Example: UNIX processes

Example: UNIX Processes

- **`fork`** creates a new process (child) which is an exact copy of the parent
  - The child starts execution at **`fork`**, gets own PID, and has its own state.
  - COW – copy on write policy
  - Processes are "joined" by **`wait`** in parent and **`exit`** in child.
  - On success, **`fork`** returns 0 to child and the child's PID to parent

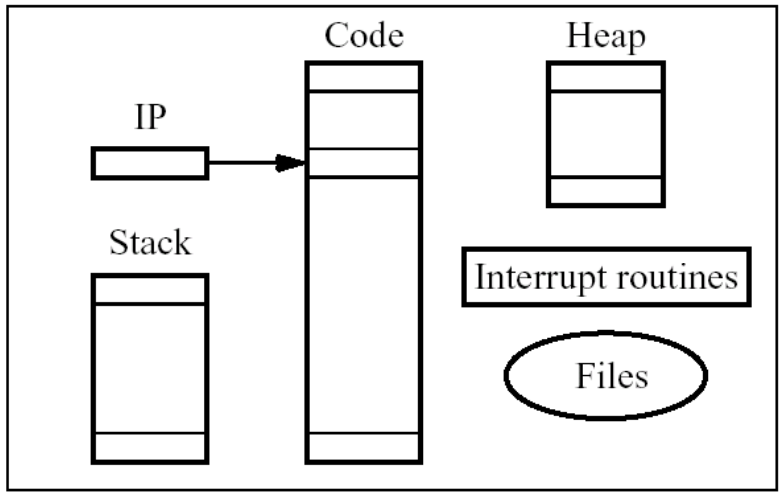
```
pid = fork();
if (pid == 0) {
     Code to be executed by child;
    exit(0); // join
} else {
     Code to be executed by parent;
    wait(0); // join
}
```

- **`fork`** followed by **`exec`** (in child) is used to execute a completely different program in the child process.

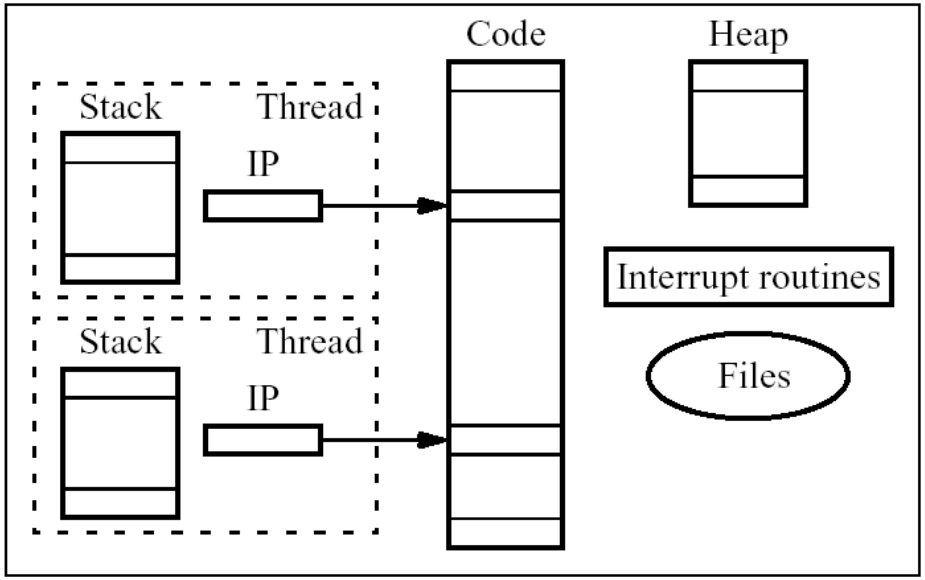
Threads

- A ***thread*** is essentially a program counter, an execution stack, and a set of registers – thread context.
  - All the other data structures and the code belong to a HWP where threads are created, and are shared by the threads.
  - Each thread is assigned a unique thread ID.
- Example: Pthreads – POSIX (IEEE Portable OS Interface) threads
  - **`pthread_create`** creates a new thread
    - The thread executes a given function, has its own stack and registers, but share global variables with other threads.
  - Threads can be "joined" by **`pthread_join`** in parent and **`return`** or **`pthread_exit()`** in child

Process versus Thread



(a) Process



(b) Threads

Parallel Programming Models

- A programming model defines how processes communicate and synchronize
- ***Shared memory programming model***, a.k.a. Shared address space (SAS) model.
    - Portions of virtual address spaces of processes are shared (common)
    - Processes communicate via shared memory (common variables) by conventional reads and writes
    - An imperative language, e.g. C or Java, requires explicit synchronization mechanisms such as locks, barriers, semaphores, monitors.
- ***Distributed memory programming model***, a.k.a. Message passing programming model
    - No shared memory, only communication channels are shared
    - Processes communicate by sending and receiving messages over communication channels.
    - Synchronization is implicit: a message to be received must be sent

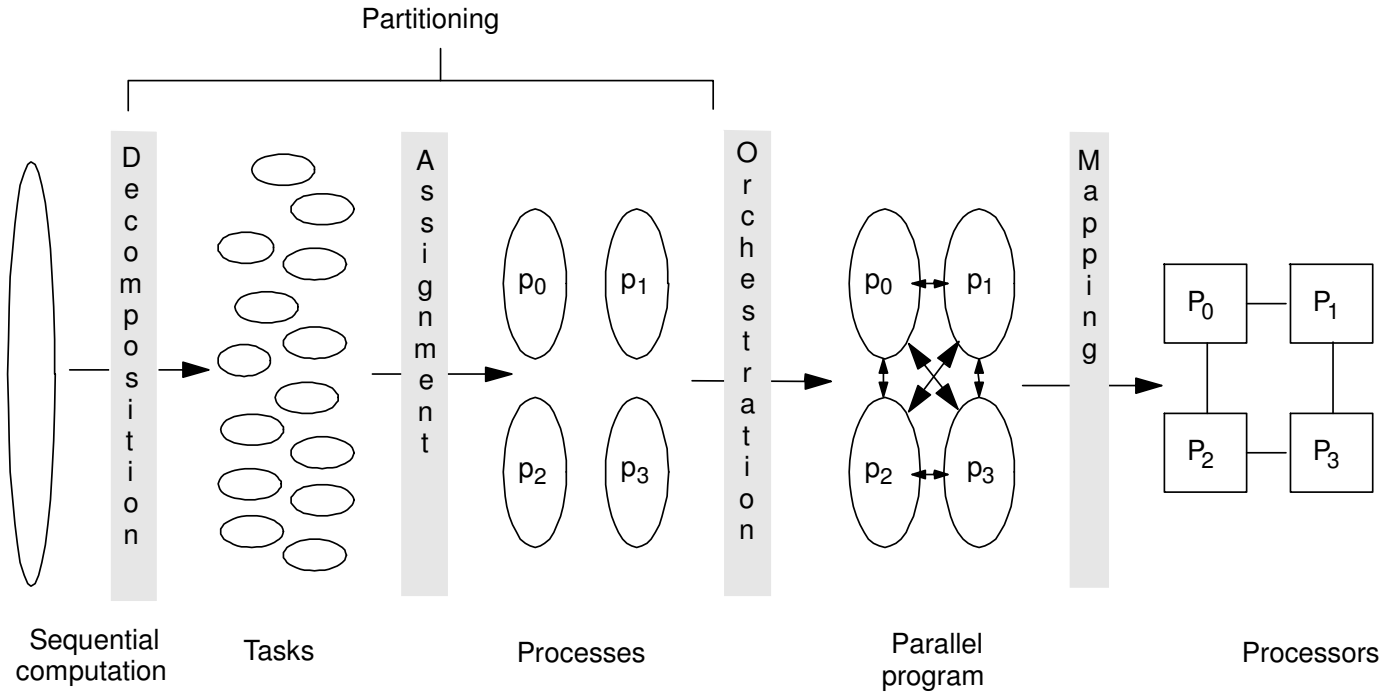
Existing Synchronization and Communication Mechanisms

- Synchronization mechanisms – for shared memory programming
  - Locks
  - Barriers
  - Condition variables
  - Semaphores
  - Monitors
- Communication mechanisms – for distributed memory programming
  - Asynchronous message passing
  - Synchronous message passing (e.g. CSP: Communicating Sequential Processes)
  - Remote procedure call (RPC) and Rendezvous
  - Remote method invocation (RMI)

Four Steps in Creating a Parallel Program

(1) Decomposition; (2) Assignment; (3) Orchestration; (3) Mapping

- Done by programmer or system software (compiler, runtime, ...).
- The programmer does it explicitly



1. Decomposition

- Break up computation into ***tasks*** to be divided among processes
    - Tasks may become available dynamically
    - Number of available tasks may vary with time
    - Number of tasks available at a time is upper bound on achievable speedup
  - i.e. identify concurrency and decide level at which to exploit it
- Goal: Enough tasks to keep processes busy, but not too many
- Simple way to identify concurrency in a sequential program is to look at loop iterations
  - dependence analysis; if not enough concurrency (loops are sequential), then look further: examine fundamental dependences, ignoring loop structure

2. Assignment

- Specifying mechanism to divide work up among ***processes***
  - E.g. which process computes forces on which stars, or which rays
  - Together with decomposition, also called ***partitioning***
  - Balance workload, reduce communication and management cost
- Structured approaches usually work well
  - Code inspection (parallel loops) or understanding of application
  - Well-known heuristics
  - Static assignment (when number of tasks is known)
  - Dynamic assignment
    - "Bag of tasks" for the fixed number of processes ("work farm")
    - Create a new process for a new task

3. Orchestration

- Naming data
- Structuring communication
  - Shared versus private memory
  - Messaging
- Synchronization
- Organizing data structures and scheduling tasks temporally
- Goals
  - Reduce cost of communication and synch. as seen by processors
  - Hide communication and/or synchronization latency by multithreading and/or data prefetching

4. Mapping

- After orchestration, already have parallel program
- Two aspects of mapping:
  - Which processes will run on same processor, if necessary
  - Which process runs on which particular processor
    - mapping to a network topology
- Two extremes:

  (1) Space-sharing: machine divided into subsets, one appl in a subset

  (2) Complete resource management control to OS
  - Real world is in between: user specifies desires, system may ignore
- Usually adopt the view: process $\leftrightarrow$ processor

Programming Notation Used in the Text and on Lectures

- A language similar to C
- Declarations – much like in C and Matlab
  - Primitive types (**`int`**, **`double`**, **`char`**, etc.)
  - Arrays: **`int c[1:n]`**, **`double c[n, n]`**
  - Process declaration **`process`** – start one or several processes in background (detached thread)
- Statements:
  - Assignment statement
  - Control flow statements: **`if`**, **`while`**, **`for`** – much like in C
    - **`for`** with quantifiers:

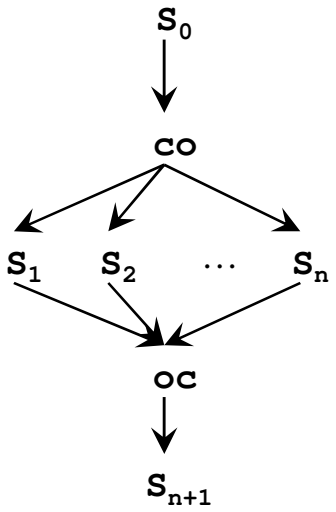
```
for [i=0 to n-1, j = 0 to n-1] …;
for [i=1 to n by 2] …; # odd values from 1 to n
for [i=1 to n st i!=x] …; # every value except i==x, "st" stands for "such that"
```

  - ***Concurrent statement*** (**`co`**-statement) – starts two or more threads in parallel and then waits until all the threads complete
  - ***Await statement*** – used for synchronization – will be introduced later

Two Forms of co-Statement

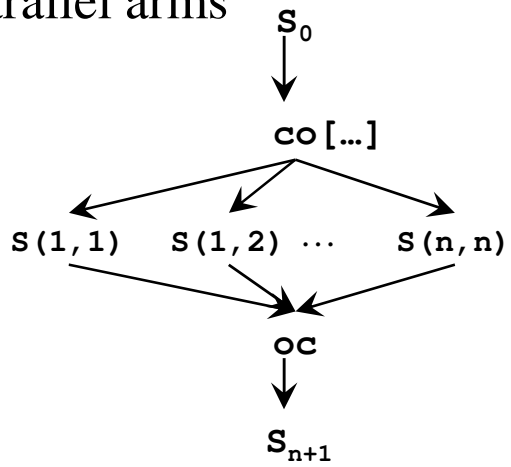
1. Different statements in parallel arms

```
S0;
co S1; # thread 1
|| ...
|| Sn; # thread n
oc;
Sn+1;
```



2. With quantifiers: Same sequence of statements in parallel arms (for every combination of variables in quantifiers)

```
S0;
co [i=1 to n, j=1 to n] { # n x n threads
    S(i,j);
}
oc;
Sn+1;
```



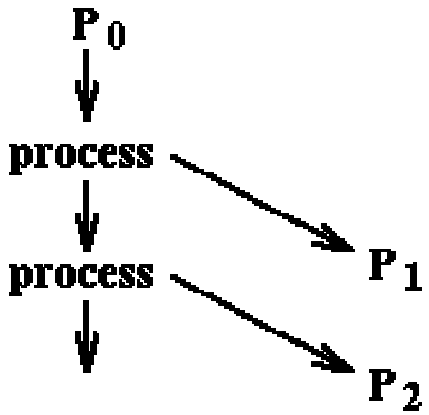
`process` Declaration

- Syntax is similar to **`co`** but with one arm and /or one quantifier
- Start a single process in background

```
process p
    { body }
```

- Start array of processes in background

```
process p[quantifier]
    { body }
```



- May declare local variables and access global variables
- Can appear where procedure declarations can appear
- Forked when its declaration is encountered
- No synchronization upon termination – detached processes

Parallel Programming Paradigms (Basic Application Patterns)

1. Iterative parallelism
2. Recursive parallelism
3. Producers and consumers (pipelines), a.k.a. dataflow
4. Clients and servers
5. Interacting peers

1. Iterative Parallelism

- Parallelism of ***independent iterations.***
- An iterative program uses loops to examine data and compute results. Some loops can be parallelized.
- Example: Matrix multiplication C = A · B
  - Sequential version:

```
Double a[n,n], b[n,n], c[n,n];
for [i=0 to n-1] {
  for [j=0 to n-1 {
    c[i,j]=0.0;
    for [k=0 to n-1]
      c[i,j]= c[i,j]+a[i,k]*b[k,j];
  }
}
```

  - Parallel version (by rows):

```
Double a[n,n], b[n,n], c[n,n];
co  [i=0 to n-1] {
  for [j=0 to n-1 {
    c[i,j]=0.0;
    for [k=0 to n-1]
      c[i,j]= c[i,j]+a[i,k]*b[k,j];
   }
} oc;
```

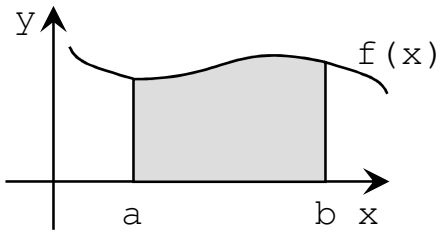
  - Replace **`for`** with **`co`**
  - n threads in **`co`** are executed concurrently for different values of i
  - in parallel by elements: **`co [i=0 to n-1, j=0, n-1]`**

2. Recursive Parallelism

- Parallelism of ***independent recursive calls***
  - Assume, a recursive procedure calls itself ***more than once*** in its body.
  - If the calls are independent, they can be executed in concurrent threads
- Examples: Quick sort, adaptive quadrature
  - "Divide and conquer" (domain decomposition)
    - Split a data region (e.g. list, interval) into several sub-regions to be processed recursively using the same algorithm

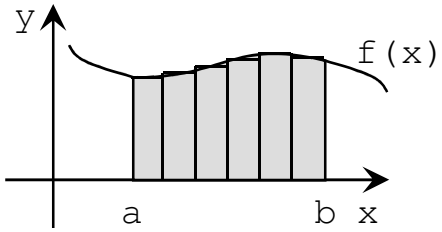
Example: The Quadrature Problem

- Compute an approximation of the integral of a continuous function $f(x)$ on the interval from a to b

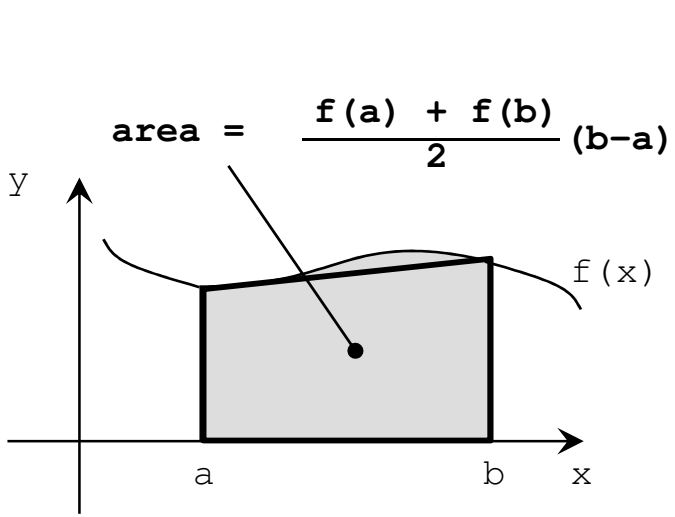


- ***Sequential iterative quadrature program***
  - using the trapezoidal method:

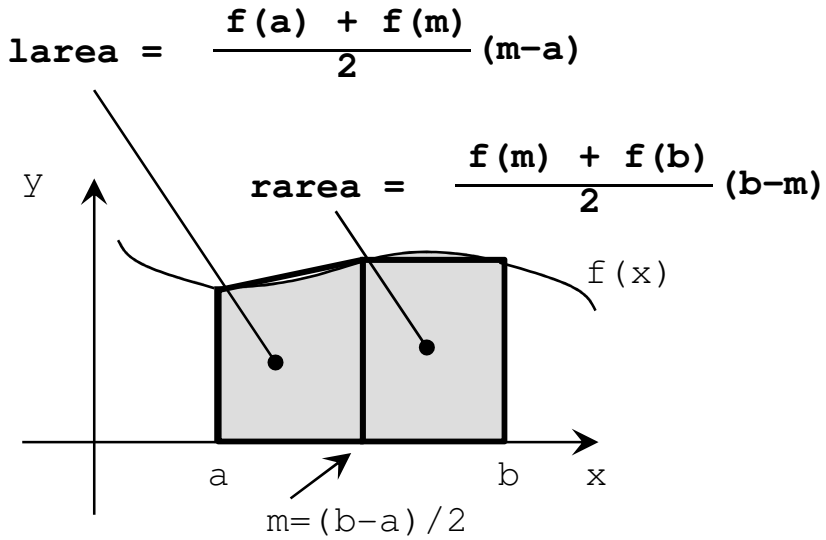
```
double fl = f(a), fr, area = 0.0;
double dx = (b-a)/ni;
for [x = (a + dx) to b by dx] {
  fr = f(x);
  area = area + (fl + fr) * dx / 2;
  fl = fr;
}
```



Recursive Adaptive Quadrature Procedure



(a) First approximation (`area`)



(b) Second approximation (`larea + rarea`)

- If | (larea + rarea) - area | > *e*, repeat computations for each of the intervals [a, m] and [m,b] in a similar way until the difference between consecutive approximations is within a given *e*

Recursive Adaptive Quadrature Procedure

- Sequential procedure:

```
double quad(double l,r,fl,fr,area) {
  double m = (l+r)/2;
  double fm = f(m);
  double larea = (fl+fm)*(m-l)/2;
  double rarea = (fm+fr)*(r-m)/2;
  if (abs((larea+rarea)-area) > e) {
     larea = quad(l,m,fl,fm,larea);
     rarea = quad(m,r,fm,fr,rarea);

  }
  return (larea+rarea);
}
```

- Parallel procedure:

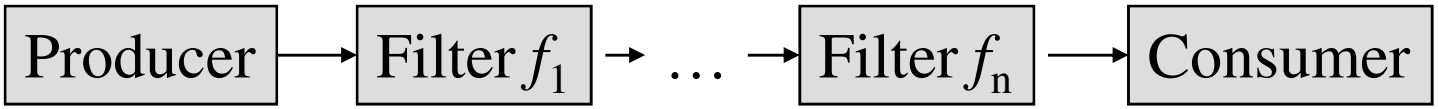
```
double quad(double l,r,fl,fr,area) {
   double m = (l+r)/2;
   double fm = f(m);
   double larea = (fl+fm)*(m-l)/2;
   double rarea = (fm+fr)*(r-m)/2;
   if (abs((larea+rarea)-area) > e) {
      co larea = quad(l,m,fl,fm,larea);
      || rarea = quad(m,r,fm,fr,rarea);
      oc
   }
   return (larea+rarea);
}
```

- Two recursive calls are independent and can be executed in parallel
- Usage:

```
area = quad(a,b,f(a),f(b),(f(a)+f(b))*(b-a)/2)
```

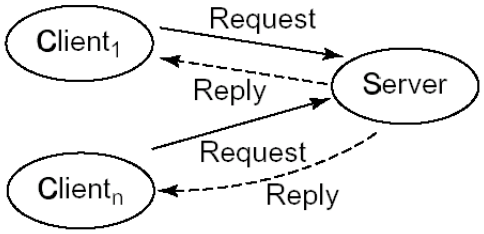
3. Producers and Consumers. Pipes

- Parallelism of production (of next data) and consumption (of previous data)
  - One-way data stream between Producer and Consumer
  - "Filters" can be placed in between
  - Processes can be organized in a pipeline
    - Parallelism of pipeline stages
    - Each consumes the output of predecessor and produces the input for its successor – true data dependence between stages
    - Data buffers (FIFO queues) are placed between processes



4. Clients and Servers

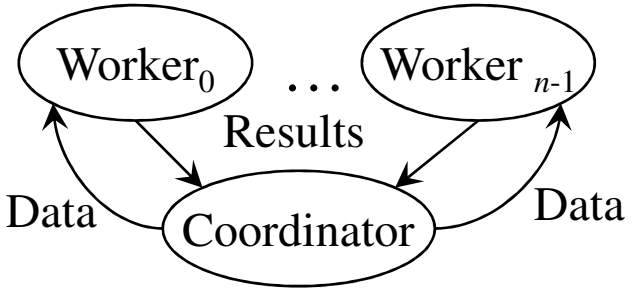
- Parallelism of client and server processes
  - Client requests a service
  - Server provides the service
  - Two-way communication: request – reply pairs



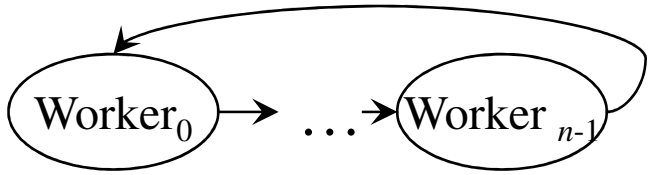
- Parallelism in servicing of multiple clients in separate threads
  - Multithreaded servers. Synchronization might be required
- Implemented
  - Distributed-memory: using message passing, RPC, rendezvous, RMI
  - Shared-memory: using subroutines, monitors etc.
- Example: (Distributed) file systems
  - open, read, write, close – client requests
  - Data, acknowledgements – server replies

5. Interacting Peers

- Parallelism of "equal" peers
  - Each execute the same set of algorithms and communicate with others in order to achieve the goal
- Configurations
  - Grid
    - Master (coordinator) and slaves (workers)
    - Roles may change
  - A circular pipeline
  - Each to each
  - Mesh
  - Arbitrary



(a) Coordinator/worker interaction



(b) A circular pipeline