POLITECNICO DI MILANO

Parallel Programming Models
POSIX Threads

Parallel Computing

Serena Curzel
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
serena.curzel@polimi.it

Parallel Algorithms and Parallel Programs

- Designing parallel *algorithms* is not an easy task
- There is not an algorithm to design parallel algorithms
- These are some rules which can help in the design
- Design of parallel *programs* have more specific rules but they depend on the particular chosen language *and architecture*

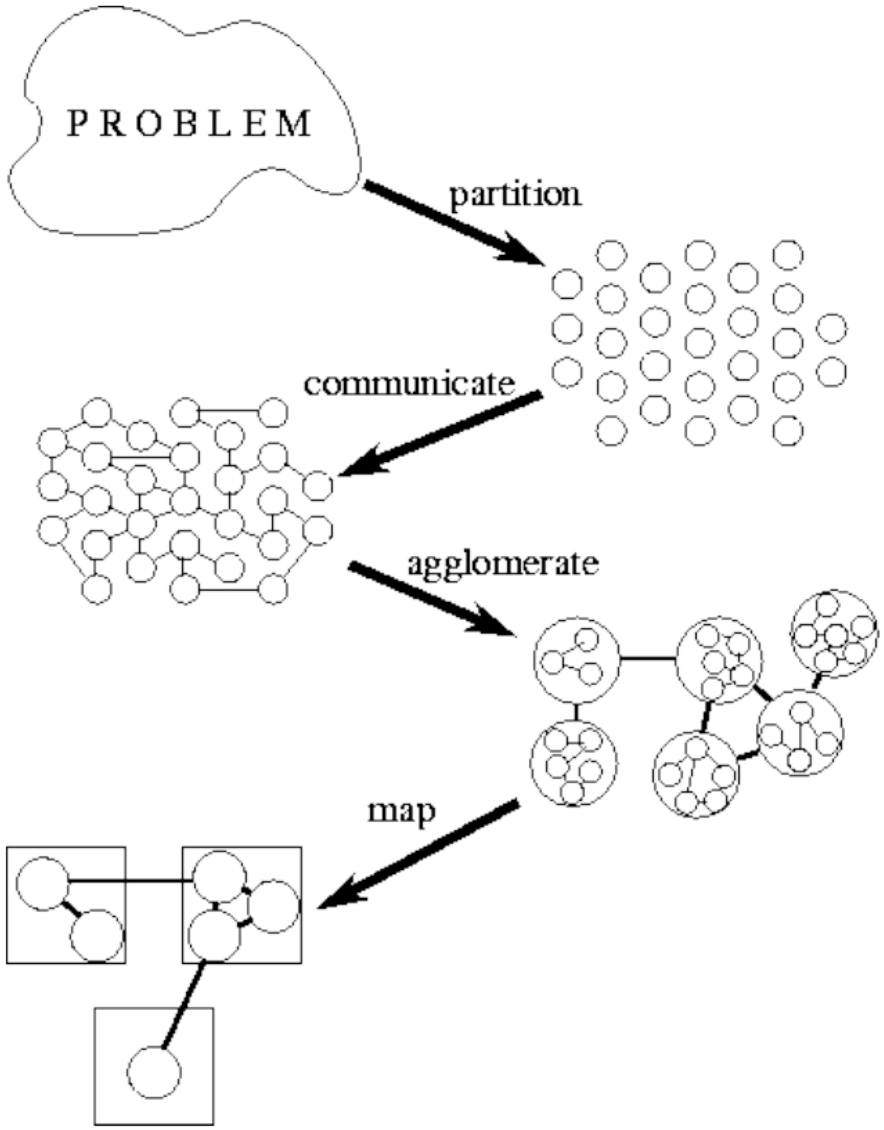
Parallel Algorithms and Parallel Programs

- Most problems have several possible parallel solutions
- A good approach is to start from machine-independent issues (concurrency) and delay target-specific aspects as much as possible

Parallel Algorithms and Parallel Programs

1. Design a parallel algorithm:
   - Understand the problem to be solved
   - Analyze data dependences
   - Partition the solution
2. Design a parallel program:
   - Analyze the target architecture(s)
   - Select best parallel programming model and language
   - Analyze the communications (cost, latency, bandwidth, visibility, synchronization, etc.)

PCAM Design Methodology



- Partitioning => task/data decomposition
- Communication => task execution coordination
- Agglomeration => evaluation of the structure
- Mapping => resource assignment

- In practice, these steps are often overlapping

Design of a parallel algorithm

- Sequential complexity
  - evaluation of the limiting behavior of the algorithm
  - *(time) complexity*: quantifies the amount of time required to produce a solution
  - *memory complexity*: quantifies the amount of memory required to run the algorithm

Design of a parallel algorithm

- Which metrics need to be taken into account?

TIME COMPLEXITY: quantifies the amount of time required to produce a solution

Design of a parallel algorithm

- Which metrics need to be taken into account?

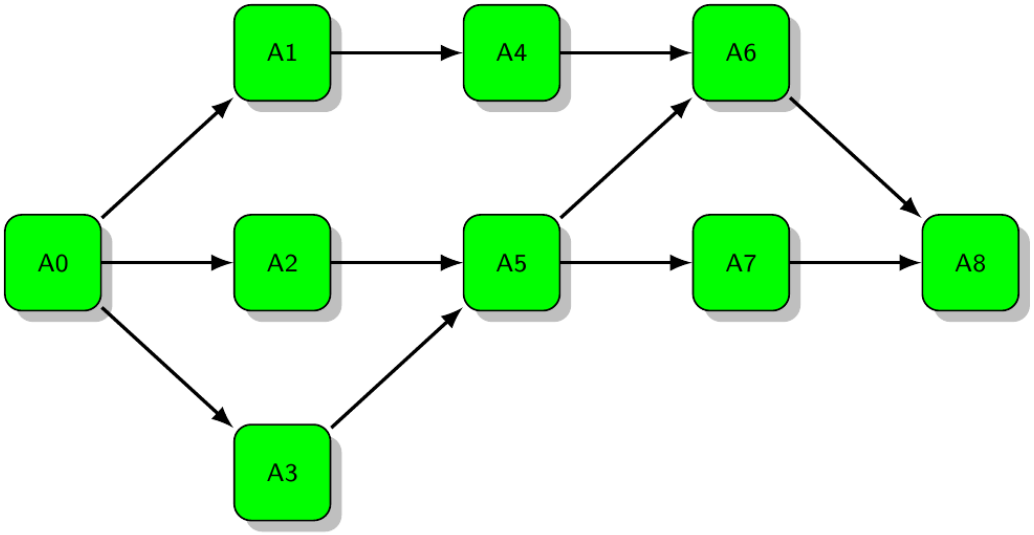
TIME COMPLEXITY: quantifies the amount of time required to produce a solution

AND

RESOURCE COMPLEXITY: quantifies how many resources are needed to produce the solution in that time

Qualitative Evaluation

- To analyze a parallel algorithm we can consider its structure (DAG):



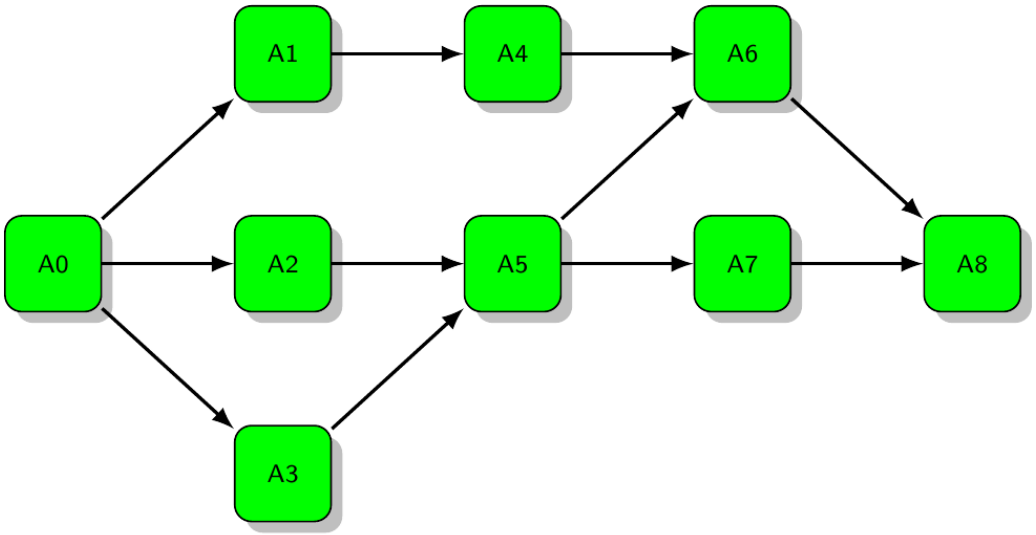
- Node = task
- Edge = data dependence

Qualitative Evaluation

- Note:
  - Concurrent tasks = they can be executed **independently** from each other
  - Parallel tasks = they are executed **at the same time** (because multiple computing resources are available)

Qualitative Evaluation

- Work *W*: number of operations performed
  - It can be higher than the sequential version of the algorithm due to communication overhead etc.
- Span *S*: longest chain of dependences (i.e., critical path) which determines the minimal time required to execute the algorithm



W = 9
S = 5

Qualitative Evaluation

- Composition rules:

W(op) = 1
W(e1, e2) = W(e1) + W(e2)
W(e1 || e2) = W(e1) + W(e2)

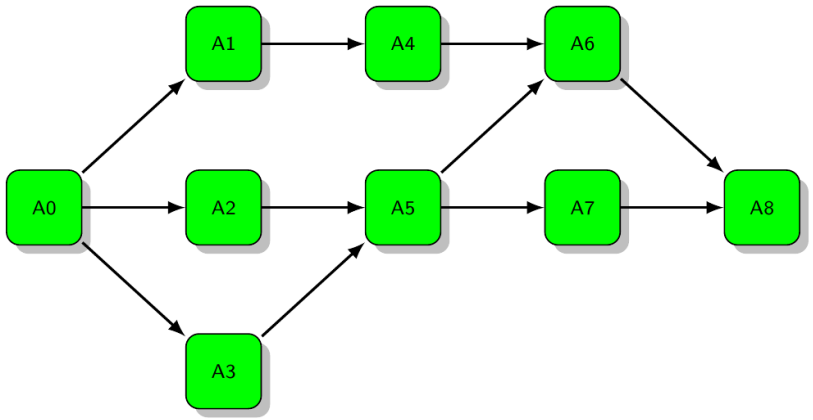
S(op) = 1
S(e1, e2) = W(e1) + W(e2)
S(e1 || e2) = max(W(e1), W(e2))

Qualitative Evaluation

- Parallelism *P*: measures the efficiency in the utilization of resources

$$P = \frac{W}{S}$$

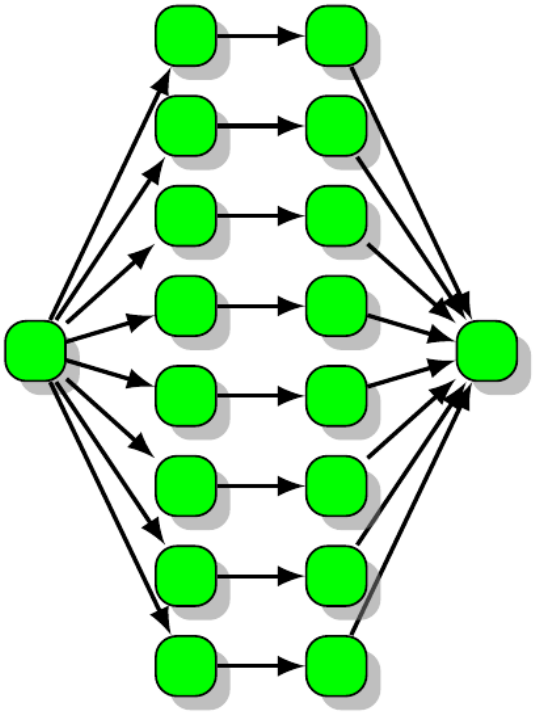
- Average number of processors that are not idle
- The actual number of processors needed to run the algorithm at full speed will be larger



W = 9
S = 5
P = 1.8

Qualitative Evaluation

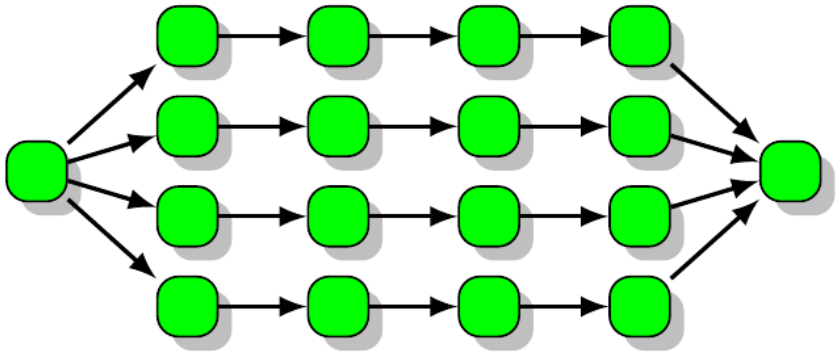
- Comparison of two different algorithms:



$W = 18$

$S = 4$

$P = 4.5$



$W = 18$

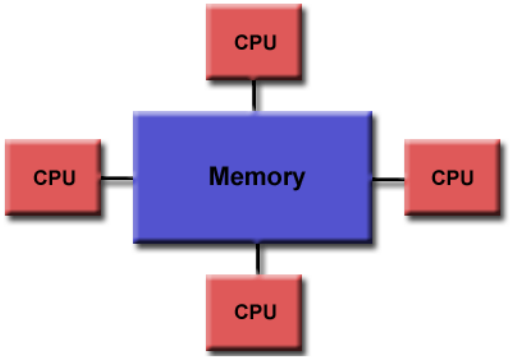
$S = 6$

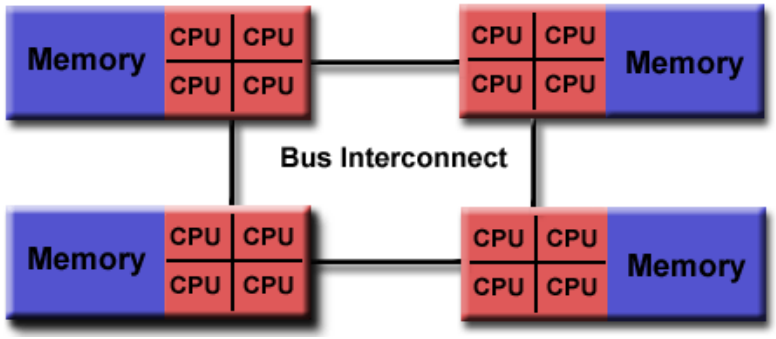
$P = 3$

Qualitative Evaluation

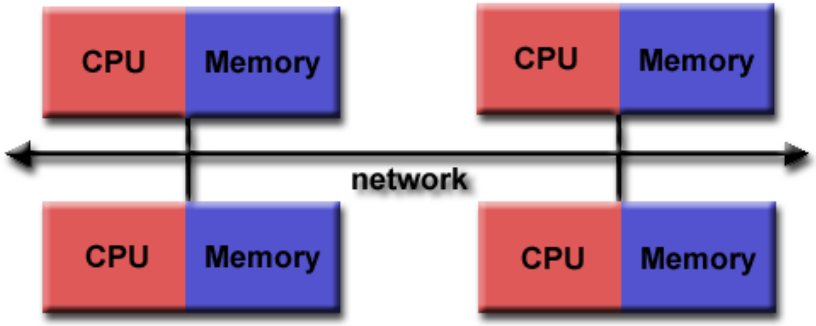
- Good parallel algorithms are designed to have
  - Lowest possible work (to reduce resource usage)
  - Highest possible parallelism (decreasing span)
- There is a trade-off between the two quantities
  - For example, reducing the span could mean adding communication/synchronization overhead
- Qualitative evaluation may not be enough

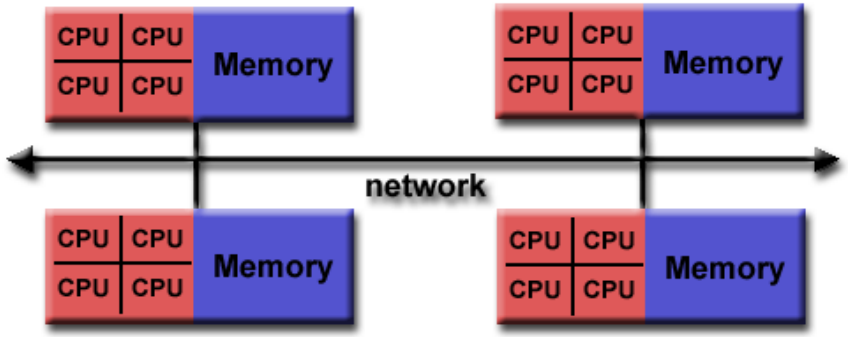
From algorithm to program: Memory architectures





- Shared memory



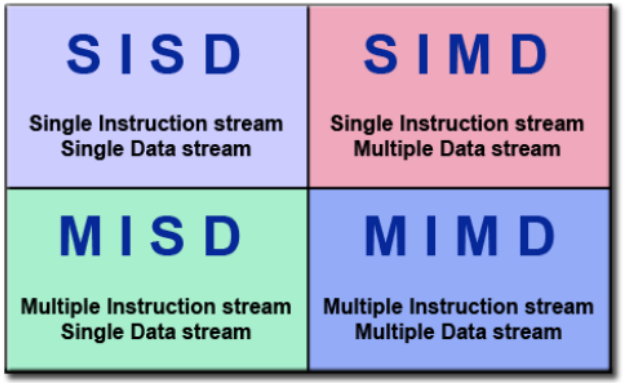


- Distributed memory

From algorithm to program: Parallel Programming Models

- Shared memory (without threads)
- Threads
- Message Passing / Distributed Memory
- Data Parallel

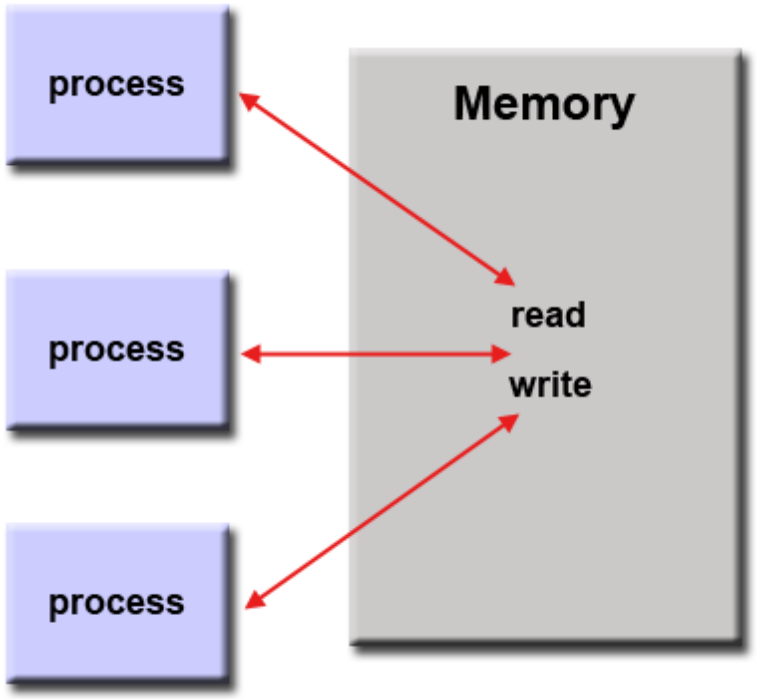
- Flyn's taxonomy: Single Program Multiple Data, Multiple Program Multiple Data etc.



- Programming models *may or may not* correspond to the underlying hardware architecture

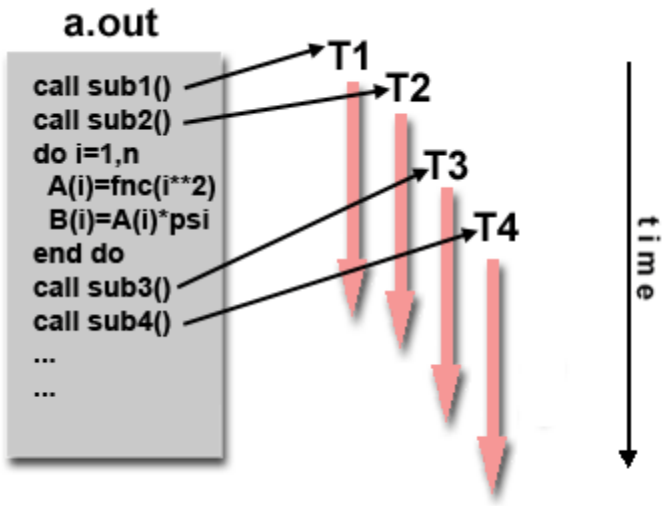
Shared memory (without threads)

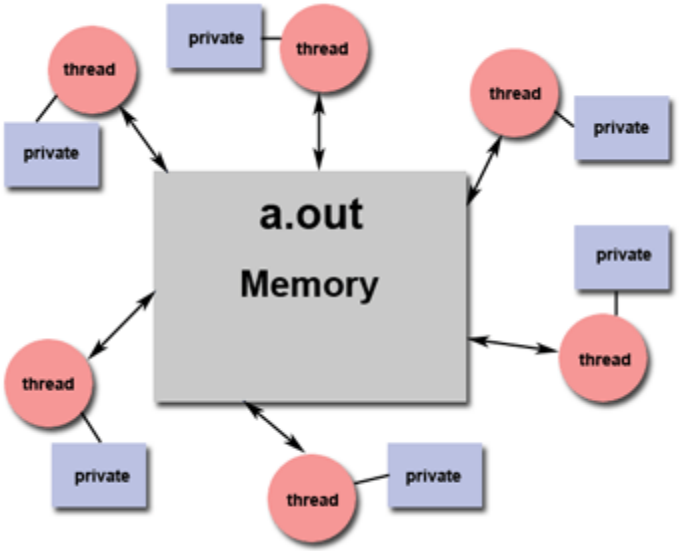
- Simplest model
- Processes share a common address space
- No explicit communication
- Difficult to maintain locality



Threads model

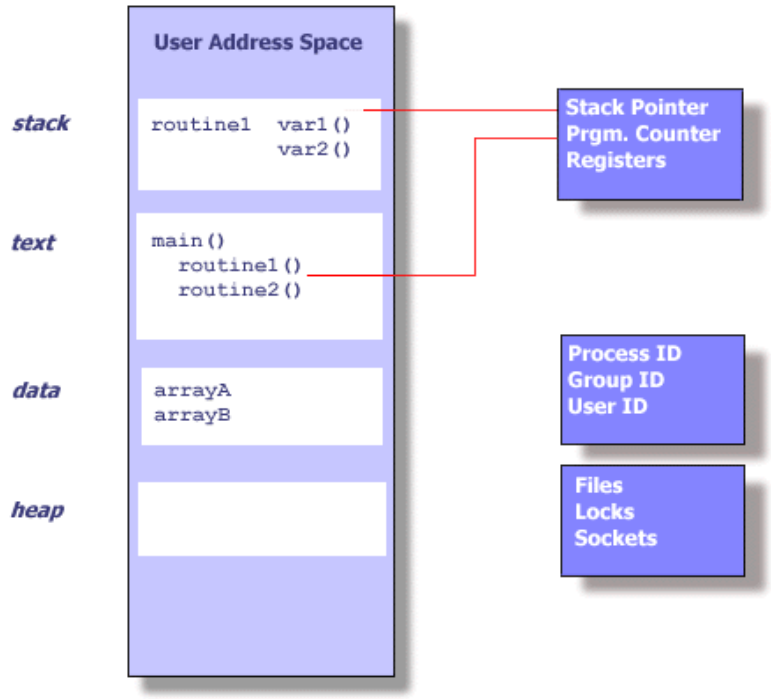
- Single process with multiple threads that can execute concurrently
- Each thread has local data, but it can access all resources acquired by the main process





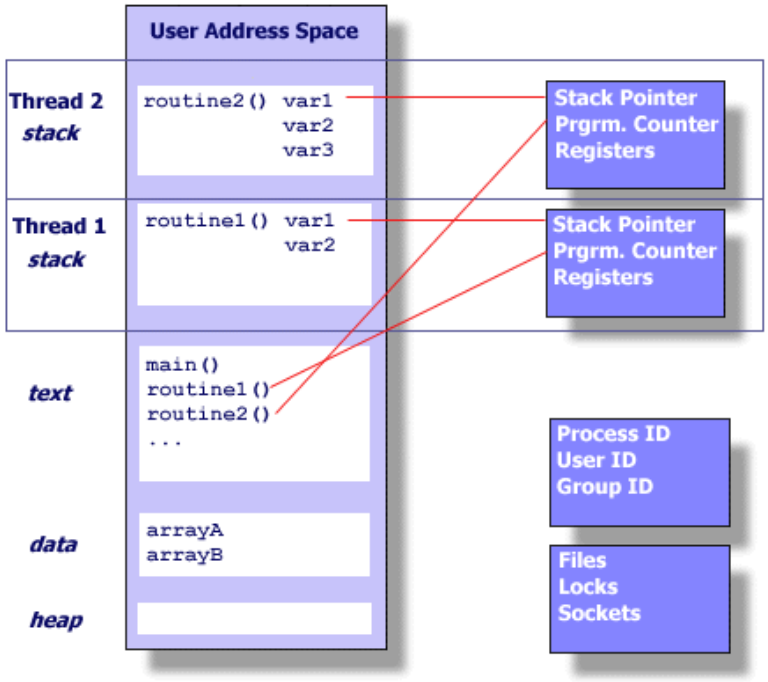
Processes and threads

- A UNIX process is created by the operating system
- Processes contain information about program resources and program execution state
  - Process ID, process group ID, user ID, and group ID,
  - Environment
  - Working directory
  - Program instructions
  - etc.



Processes and threads

- A thread is an independent stream of instructions within a process
- Threads can be scheduled to run by the operating system
- Any thread can execute at the same time as other threads
- Threads have local resources and can access the shared process resources



Threads

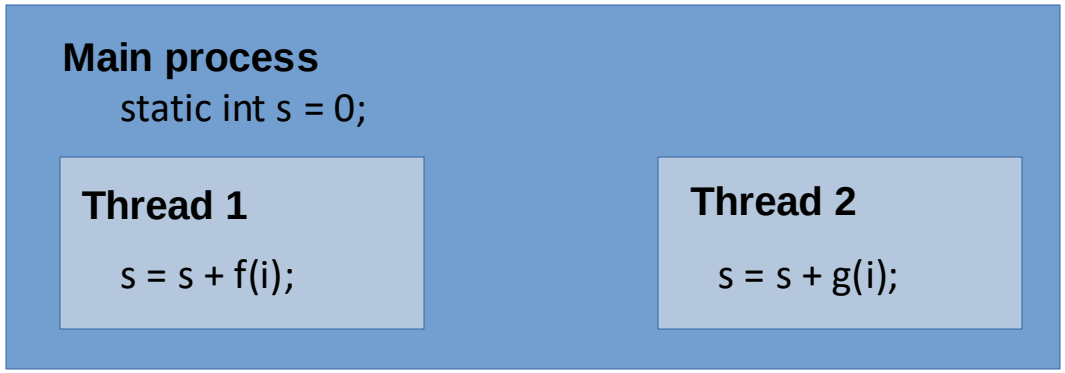
- A thread can be seen as any "procedure" that runs independently from its main program
- Example of multi-thread program:
  - `main` calls some procedures
  - all the procedures are run **simultaneously and/or independently** by the operating system
- Threads can be created *dynamically* during execution
- Duplicate only the *bare essential* resources
  - Multi-thread is “lighter” than multi-processes

Threads

- Threads exists within a process and share most of the process resources
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads
  - Two pointers having the same value point to the same data
  - Implicit communication by reading and writing shared variables
  - Reading and writing to the same memory locations requires explicit synchronization by the programmer

Data Race

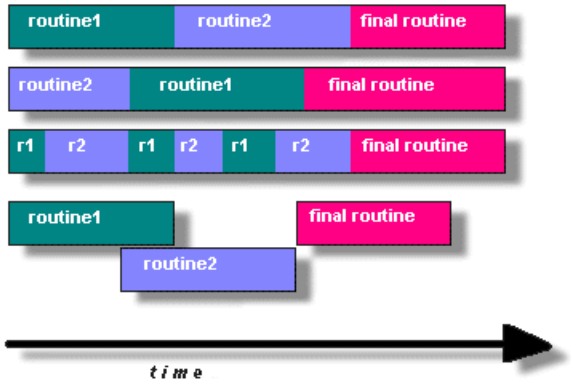
- Data race or race condition occurs when
  - Two threads (or processes) access the same variables, and at least one does a write
  - Accesses are concurrent, so they could happen at the same time



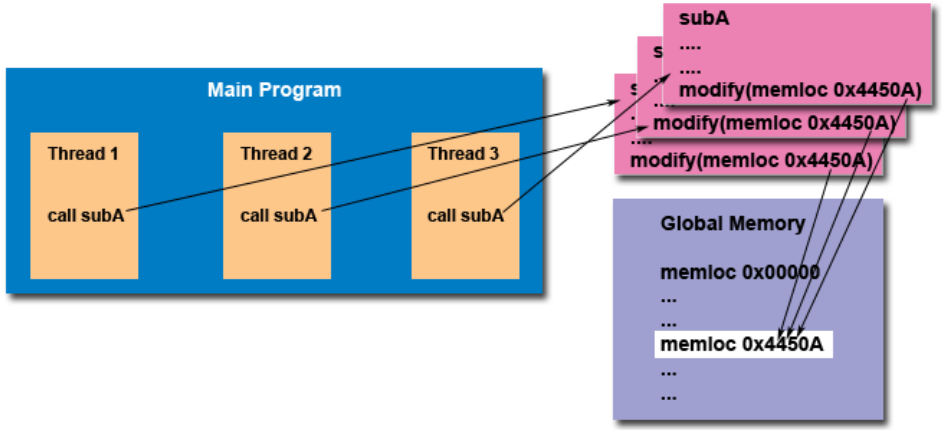
- Race condition on the global variable **s** → synchronization is needed

Designing threaded programs

- Can my program be implemented with threads?
  - Can it be organized in discrete, *concurrent* tasks?

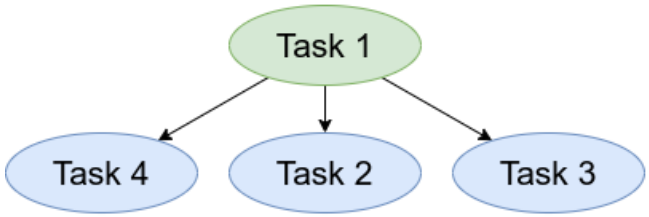


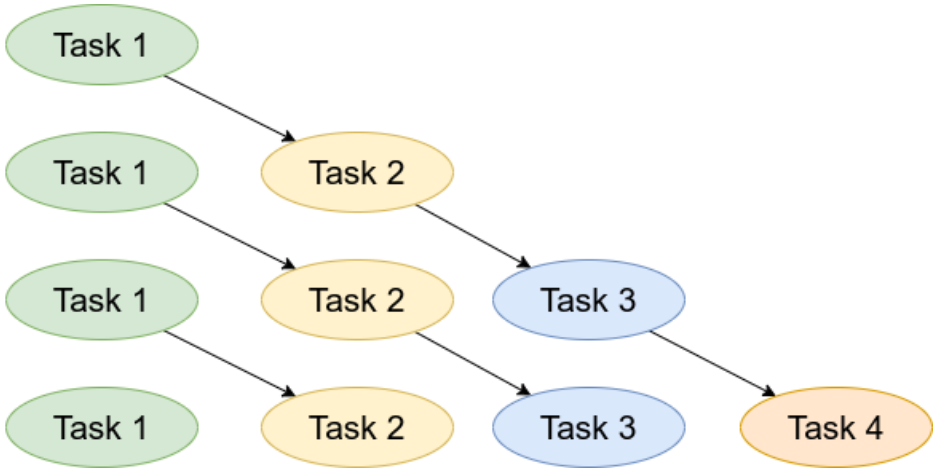
  - Is it *thread-safe*?



Designing threaded programs

- Common models for threaded programs:
  - Manager/worker
  - Pipeline





Evolution of Parallel Programming

TECHNOLOGY	TYPE	YEAR
Verilog/VHDL	Languages	1984/1987
MPI	Library	1994
PThread	Library	1995
OpenMP	C/Fortran Extensions	1997
CUDA	C Extensions	2007
OpenCL	C/C++ Extensions + API	2008
Apache Spark	API	2014

- Message passing and threads are technologies older than standards that defined them
- New technologies directly introduced as standard

Authors and Developers

TECHNOLOGY	AUTHORS	DEVELOPERS
Verilog/VHDL	Phil Moorby – Prabhu Goel / United States Department of Defence	Accellera Systems Initiative / IEEE VHDL Analysis and Stan-dardization Group
MPI	MPI Forum	MPI Forum
PThread	IEEE Technical Committee on Operating System	Austin Group
OpenMP	OpenMP Architecture Review Board	OpenMP Architecture Review Board
CUDA	NVIDIA	NVIDIA
OpenCL	Apple	Khronos Group
Apache Spark	Berkeley	Apache Foundation Databricks

All technologies except CUDA are developed by consortiums/Foundation

Target architectures

TECHNOLOGY	Processors	Memory
Verilog/VHDL	ASIC – FPGA	
MPI	Multi CPUs	(Mainly) Distributed Memory
PThread	Multi-core CPU	(Mainly) Shared Memory
OpenMP	Multi-core CPU	(Mainly) Shared Memory
CUDA	CPU + GPU(s)	(Distributed) Shared Memory
OpenCL	Heterogeneous Architecture	Both distributed and shared memory
Apache Spark	Multi CPUs	Distributed Memory

CUDA memory:

- Distributed between CPU and GPU
- Shared on the GPU

Parallelization and Communication

TECHNOLOGY	Parallelism	Communication
Verilog/VHDL	Explicit	Explicit
MPI	Implicit	Explicit
PThread	Explicit	Implicit
OpenMP	Explicit	Implicit
CUDA	Implicit/Explicit	Implicit/Explicit
OpenCL	Explicit/Implicit	Explicit/Implicit
Apache Spark	Implicit	Implicit

- CUDA and OpenCL:
  - Different kernels-CPU code: explicit parallelism and communication
  - Threads of the same kernel: implicit parallelism and communication
- In OpenCL explicit parallelism can be used also in a single kernel

Pros and Cons: Verilog/VHDL

- Pros:
  - Complete control on computation and memory
  - No overhead introduced in the computation
  - Provides access to potentially large computational power
- Cons:
  - Requires specific hardware (e.g., ASIC or FPGA) to implement functionality
  - Difficult to learn: completely different programming language and programming paradigm
  - Depends on the chosen target architecture

Pros and Cons: MPI

- Pros:
  - Can be adopted on different types of architectures
  - Scalable solutions
  - Synchronization and data communication are explicitly managed
- Cons:
  - Communication can introduce significant overhead
  - Programming paradigm more difficult than shared memory-based ones
  - Standard does not reflect immediately advances in architecture characteristics

Pros and Cons: PThread

- Pros:
  - Can be adopted on different architectures
  - Explicit parallelism and full control over application
- Cons:
  - Task management overhead can be significant
  - Not easily scalable solutions
  - Low level API

Pros and Cons: OpenMP

- Pros:
  - Easy to learn
  - Scalable solution
  - Parallel applications can also be executed sequentially
- Cons:
  - Mainly focused on shared memory homogeneous systems
  - Requires small interaction between tasks

Pros and Cons: CUDA

- Pros:
  - Provides access to the computational power of GPUs
  - Writing a CUDA kernel is quite easy
  - Already optimized libraries
- Cons:
  - Targets only NVIDIA GPUs
  - Difficult to extract massive parallelism from application
  - Difficult to optimize CUDA kernel

Pros and Cons: OpenCL

- Pros:
  - Target-independent standard
  - Hides architecture details
  - Same programming infrastructure for very heterogeneous architecture: CPU + GPU (+FPGA)
- Cons:
  - Difficult programming paradigm for its heterogeneity
  - Hiding of architecture details makes difficult to obtain best performances
  - Gradually abandoned

Pros and Cons: Apache Spark

- Pros:
  - API for different languages
  - Explicit parallelization and communication are not required
  - Preinstalled on cloud provider VMs
- Cons:
  - Suitable only for big data applications
  - Does not (yet) fully support GPUs

Mix of parallelism technologies: OpenMP + CUDA

- Allows to exploit multi-core CPU and GPU
- CUDA is used to parallelize GPU code
- OpenMP is used to parallelize CPU code

Mix of parallelism technologies: MPI + OpenMP

- First scenario:
  - MPI used to express coarser parallelism (Multi CPU)
  - OpenMP used to express finer parallelism (Multi core)
- Second scenario:
  - MPI used to implement communications
  - OpenMP used to parallelize computation

Mix of parallelism technologies: OpenCL + Verilog/VHDL

- In principle, hardware kernels (implemented for example on FPGA) can be used as accelerators
- OpenCL used to describe parallelism among different processing elements
- Verilog/VHDL used to describe hardware kernel
- Example of target: Intel Xeon Scalable (Skylake + Arria 10 FPGA)

From design to implementation

- Parallel patterns describe an *algorithm*
- Parallel architectures describe *computing resources, memories and interconnections*
- Parallel programming models are an *abstraction* above the hardware
- There are different *implementations* for each parallel programming model
- POSIX threads and OpenMP are two implementations of a **shared memory** *parallel programming model* with **threads**

Threads implementations

- The programmer is responsible for handling parallelism and synchronization, usually through
  - A library of subroutines
  - A set of compiler directives
- Historically, hardware vendors have implemented their own proprietary versions of threads
- We will see two different standards:
  - POSIX Threads (Pthreads)
  - OpenMP

Pthreads

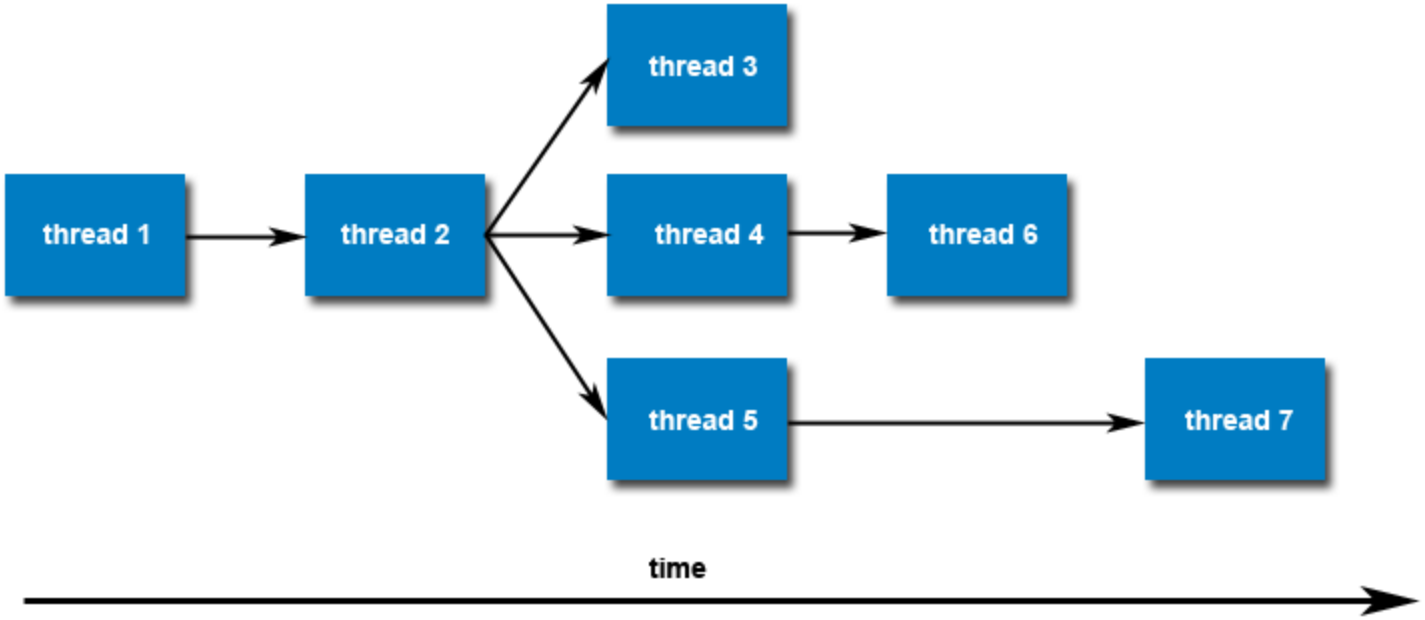
- IEEE POSIX 1003.1c standard (1995) specifies the API to *explicitly* manage threads
- Standard is (slightly) evolving:
  - IEEE Std 1003.1-2017/Open Group Base Specification Issue 7
- API is a set of *C language* programming types and procedure calls
  - Header file: `pthread.h`
  - Library (stand-alone or part of another library)
  - Add `-pthread` to the gcc options on Linux

The Pthreads API

- Thread management
  - Creation
  - Joining
  - Detaching
  - Set/query attributes
- Synchronization
  - Mutexes: Control access to mutual exclusion sections
  - Condition variables: Control conditional inter-threads communications

Thread Management: Creation

- Once created, threads are peers, and may create other threads: there is no implied hierarchy or dependency between threads.
- The maximum number of threads depends on the implementation



Thread Management: Creation


```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr,
        void * (* start_routine) (void *), void *arg)
```

- `thread`: identifier for the new thread returned by the subroutine
- `attr`: used to set thread attributes
  - Joinable/detached
  - Scheduling
  - Stack size
- `start_routine`: the C routine that the thread will execute once it is created.
- `arg`: argument passed to start_routine. It must be passed by address as a pointer cast of type void

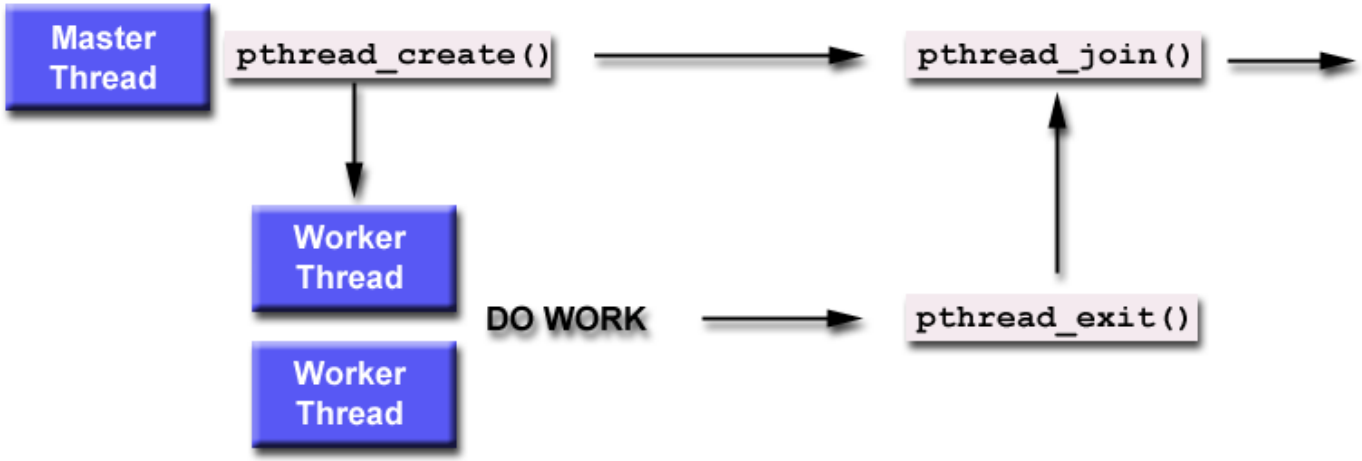
Thread Management: Termination

- The thread returns from its starting routine
- The thread makes a call to the `pthread_exit` subroutine
- The thread is canceled by another thread via the `pthread_cancel` routine
- The entire process is terminated
  - (Created threads survive if main calls `pthread_exit`)

Thread Management: Joining


```
int pthread_join(pthread_t thread, void ** retval)
```

- `pthread_join` blocks the calling thread until the specified `thread` terminates
- `retval` will contain the copy of the argument of `pthread_exit`



Thread Management: Joining vs. Detaching

- Threads can be created as joinable or detached
- A joinable thread can become detached (not vice versa)
- Explicitly set joinable or detached attribute
  - Default attribute should be joinable
  - Not all implementations adopt this default
  - Detached threads may consume less resources

Thread Management: Exercise


```
int D1,D2,D3;
void *return_value;
pthread_t th1, th2, th3;
pthread_attr_t attr;

void task1(){
  D1 = 7;
  D2 = 5;
  D3 = 3; }

void* task2(void*){
  D3 = D1 + 5;
  pthread_exit((void *) 0); }

void* task3(void *){
  D1 = D1 + 2;
  task4(); }

void task4(){
  D1 = D1 + 4;
  pthread_exit((void *) 0); }

void task5(){
  D2 = D2 + 3;
  D3 = D3 + 3; }
```

```
void* task6(void*){
  D2 = D2 + 7;
  D1 = 5;
  pthread_exit((void *) 0); }

void task7(){
  Temp = D1;
  D2 = D2 + Temp;
  D1 = D1 + 2; }

int main(int argc, char ** argv) {
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
     PTHREAD_CREATE_JOINABLE);
  task1();
  pthread_create(&th2, &attr, task2, NULL);
  pthread_create(&th3, &attr, task3, NULL);
  task5();
  pthread_join(th2, &return_value);
  pthread_join(th3, &return_value);
  pthread_create(&th1, &attr, task6, NULL);
  task7();
  pthread_join(th1, &return_value);
  printf("%d %d %d\n", D1, D2, D3); }
```

Thread Management: Exercise

- Draw a diagram with the threads, the tasks they execute, and synchronization points.
- Assume that all tasks are atomic (i.e., threads do not interfere with each other). What are the possible values printed at the end?
- What happens if tasks are not atomic?
- Does Task 4 always execute before Task 6?

Thread Management: Joining through Barriers


```
int pthread_barrier_init(pthread_barrier_t * barrier,
      pthread_barrierattr_t * attr, unsigned int count)
```

```
int pthread_barrier_wait(pthread_barrier_t * barrier)
```

- `pthread_join` requires to wait for a specific thread exit
- Several threads can be synchronized among them through barriers
  - all the threads of the group need to wait for all the other threads, then they can resume working
- `count` is the number of threads to be waited
- Not all implementations include barriers!

Synchronization: Mutexes

- Mutex (*mutual exclusion*) variables are the basic method to protect shared data when multiple writes occur
- Only one thread can lock a mutex variable at any given time
- If several threads try to lock a mutex only one thread will be successful

```
pthread_mutex_lock(&my_lock);
        /* critical section */
pthread_mutex_unlock(&my_lock);
```

- Threads that could not acquire the mutex are blocked

Synchronization: Mutexes

- Less blocking overhead: `trylock`
  - The thread can keep on working while the lock is not available
- Multiple mutexes can lead to **deadlocks**

Thread 1

```
lock(a);
lock(b);
```

Thread 2

```
lock(b);
lock(a);
```

- Three types of mutexes:
  - Normal
  - Recursive
  - Error check

Synchronization: Condition Variables

- Mutexes implement synchronization by serializing data accesses
- Condition variables allow threads to synchronize *explicitly* by signaling the meeting of a condition
- Without condition variables, the programmer would need to *poll* to check if the condition is met

Thread 1

```
lock(a);
wait on(cond, a);
do work;
unlock(a);
```

Thread 2

```
lock(a);
do work;
signal(cond);
unlock(a);
```

Examples Link

https://colab.research.google.com/drive/19e-2QBftsBi2CNJCPE54Bcy5SLMfQGDL?usp=sharing

Hello World - Pthreads


```
void *PrintHello(void *threadid)
{
   // some long computation
   printf("%ld: Hello World!\n", (long) threadid);
   pthread_exit(NULL);
}

int main()
{
   pthread_t threads[5];
   int rc;
   long t;
   for(t=0; t<5; t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
     }
   }
}
```

- There is an error in here. What will happen at runtime?
- Are the printed messages going to be in order?

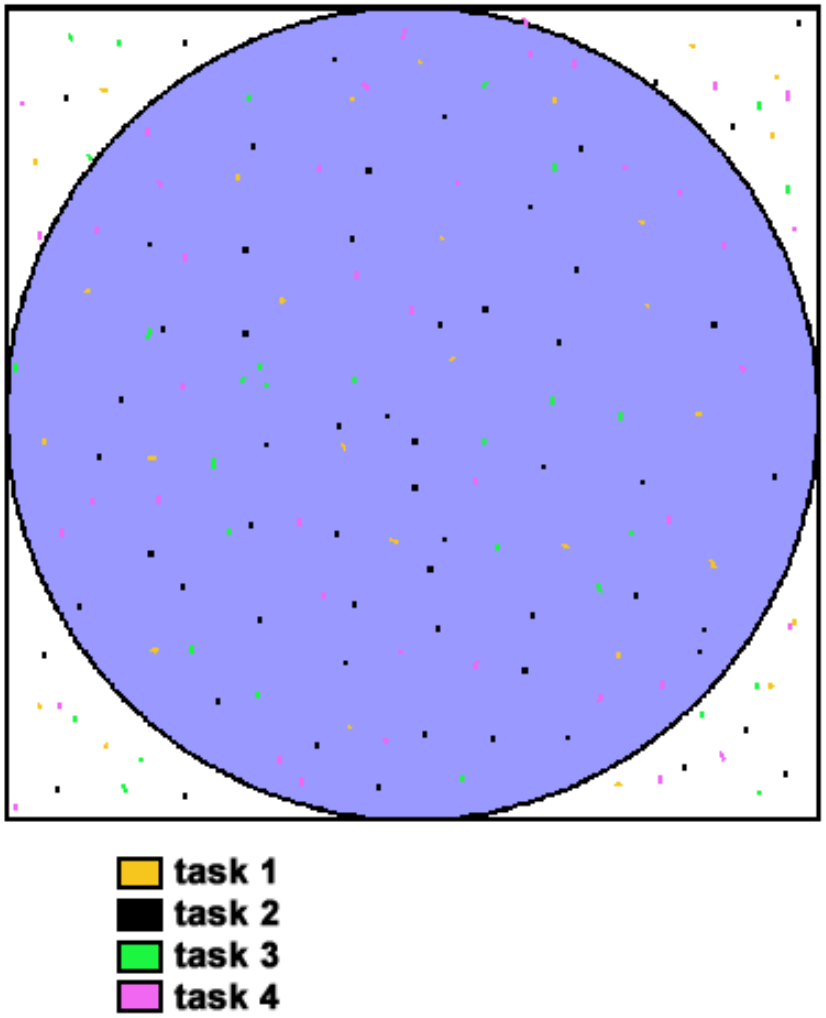
Mutexes and conditional variables


```
pthread_mutex_t mux;
pthread_cond_t cond;
int actual_condition = 0;

void *message(void * tID) {
  if (*((unsigned int *) tID) == 0)
  {
    pthread_mutex_lock(&mux);
    printf("Hello, ");
    actual_condition = 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mux);
  }
  else
  {
    pthread_mutex_lock(&mux);
    while (actual_condition == 0)
    {
      pthread_cond_wait(&cond, &mux);
    }
    printf("World!\n");
    pthread_mutex_unlock(&mux);
  }
  return 0;
}
```

```
int main() {
 unsigned int idx[2];
 pthread_t threads[2];
 pthread_attr_t attr;

 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr,
          PTHREAD_CREATE_JOINABLE);

 pthread_mutex_init(&mux, NULL);
 pthread_cond_init(&cond, NULL);

 for (unsigned int ii = 0; ii < 2; ii++) {
   idx[ii] = ii;
   pthread_create(&threads[ii], &attr, message,
          (void *) &idx[ii]);
 }
 pthread_join(threads[1], NULL);
 pthread_mutex_destroy(&mux);
 pthread_cond_destroy(&cond);
 pthread_attr_destroy(&attr);

 return 0;
}
```

- Synchronization constructs enforce execution order

Calculation of pi - Montecarlo/1



$$A_S = 4r^2$$
$$A_C = \pi r^2$$
$$\pi = 4\frac{A_C}{A_S}$$

- Ratio between areas can be approximated by ratio between internal and external points
- Main computation: check if a point is internal or not
- Each point can be checked in parallel

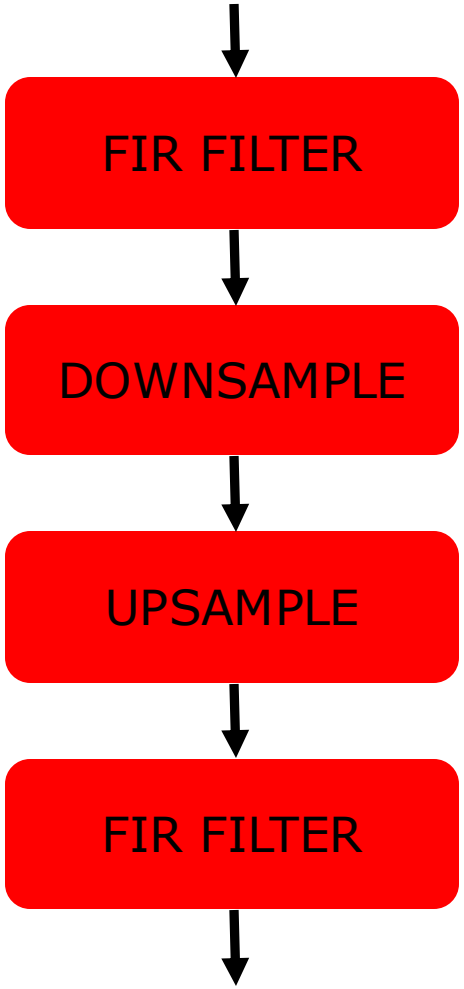
Calculation of pi - Montecarlo/2


```
for(index = 0; index < num_threads; index++)
{
   indexes[index] = index;
   pthread_create(&threads[index], &attr, check_points, (void *) &indexes[index]);
}

/* Wait the end of all the threads */
pthread_barrier_wait(&barrier);
double pi = (4.0 * inside_points)/num_points;
```

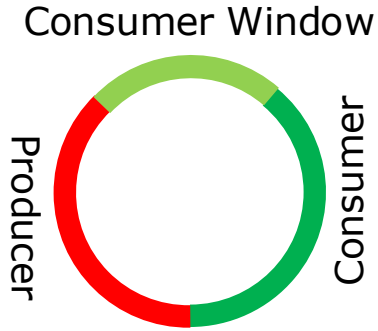
```
for(i = 0; i < num_points_per_thread; i++)
{
   double x = (rand_r(&seed)) / (double) RAND_MAX;
   double y = (double) rand_r(&seed) / (double) RAND_MAX;
   if(x*x + y*y <= 1)
      local_count++;
}

   pthread_mutex_lock(&mutex);
   inside_points += local_count;
   pthread_mutex_unlock(&mutex);
   pthread_barrier_wait(&barrier);
```

- How is access to the shared variable solved?
- Why is the index argument passed like that?
- Why do we need the barrier?

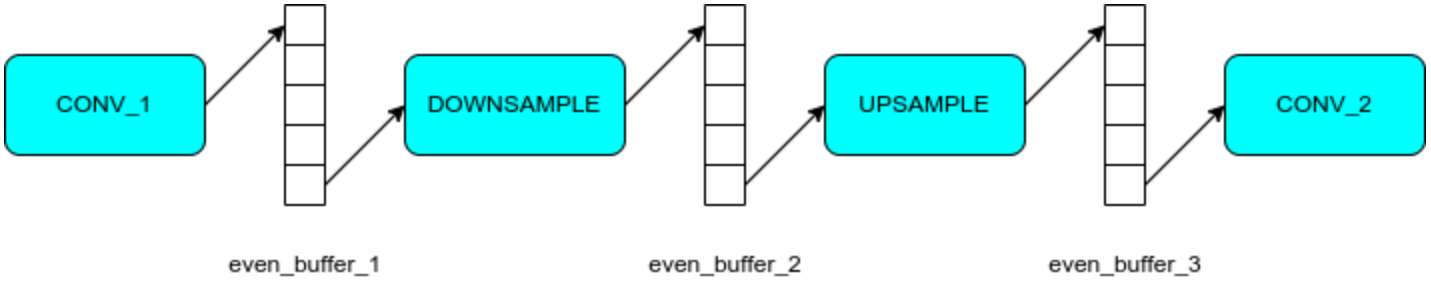
Multi rate band pass filter/1



- Pipeline implementation with four threads
- Stages work on "windows" of input, not on single samples: y[i] = f(x[i], x[i-1], x[i-2], …)
- Buffer between stages:
  - Producer cannot overwrite data not yet read by Consumer
  - Buffer size should be minimized



Multi rate band pass filter/2



main

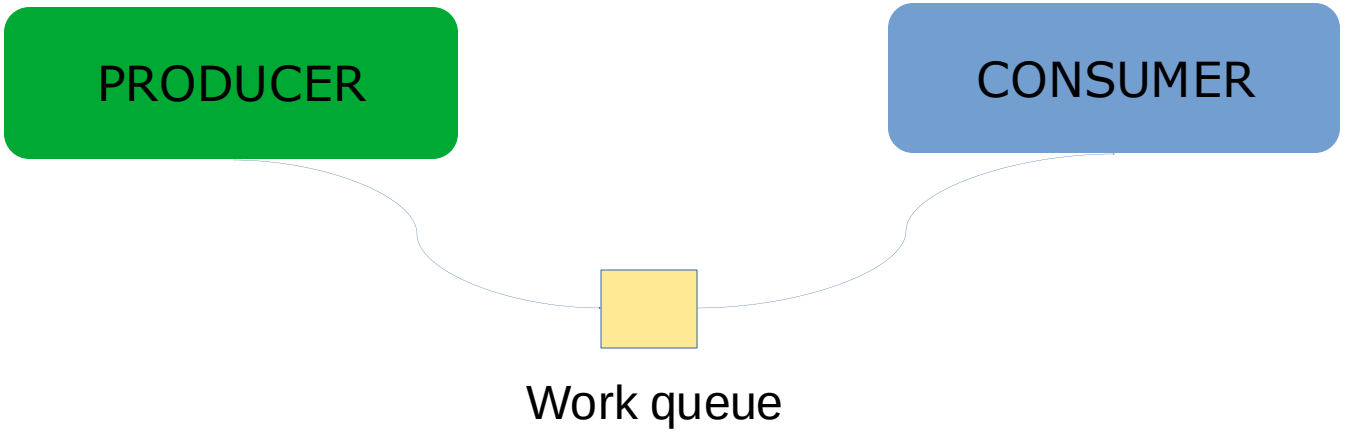
```
pthread_create(&threads[0], &attr, convolution_1, NULL);
pthread_create(&threads[1], &attr, downsample, NULL);
pthread_create(&threads[2], &attr, upsample, NULL);
pthread_create(&threads[3], &attr, convolution_2, NULL);
```

convolution

```
for(index = 0; index < WINDOW_SIZE; index++)
{
   unsigned int coefficient_index = 0;
   for(coefficient_index = 0; coefficient_index < FILTER_SIZE; coefficient_index++)
   {
      buffer[out_buffer_position] += filter[coefficient_index] * input[in_position];
   }
   out_buffer_position++;
   if(out_buffer_position == BUFFER_SIZE)
   {
      out_buffer_position = 0;
   }
}
pthread_barrier_wait(&barrier);
```

Producer – consumer work queue/1

- Producer places a task in the work queue, *if it is empty*
- Consumer picks a task from the work queue, *if it is full*, and executes it
- Single producer, single consumer, single task (but easily scalable)



Producer – consumer work queue/2


```
void *producer(void
*producer_thread_data) {
  int inserted;
  struct task my_task;
  while (!done()) {
    inserted = 0;
    create_task(&my_task);
    while (inserted == 0) {
      pthread_mutex_lock(&queue_lock);
      if (task_available == 0) {
        insert_into_queue(my_task);
        task_available = 1;
        inserted = 1;
      }
      pthread_mutex_unlock(&queue_lock);
    }
  }
}
```

```
void *consumer(void *consumer_thread_data)
{
  int extracted;
  struct task my_task;
  while (!done()) {
    extracted = 0;
    while (extracted == 0) {
      pthread_mutex_lock(&queue_lock);
      if (task_available == 1) {
        extract_from_queue(&my_task);
        task_available = 0;
        extracted = 1;
      }
      pthread_mutex_unlock(&queue_lock);
    }
    process_task(my_task);
  }
}
```

- How can we reduce the locking overhead?

Credits & References

- *Introduction to Parallel Computing,* and *POSIX Threads Programming* by B. Barney, Lawrence Livermore National Laboratory https://hpc.llnl.gov/training/tutorials
- UC Berkeley CS267: *Applications of Parallel Computers* https://sites.google.com/lbl.gov/cs267-spr2020
- *Introduction to Parallel Computing*, A. Grama, A. Gupta, Gg. Karypis, V. Kumar https://www.cs.purdue.edu/homes/ayg/book/Slides/
- I.Foster, "Designing and Building Parallel Programs", Addison-Wesley, 1995