An Introduction into Parallelization

Part I - Concepts, Terminology & Models

Jascha Schewtschenko

Royal Observatory of Edinburgh, University of Edinburgh

June 27, 2023

Outline

1. Concepts and Terminology
   - Amdahl's Law
   - Granularity
   - Scalability
   - Complexity

2. Parallel Programming Models
   - Recap: Computer Architectures
   - Single-Instruction Multiple-Data (SIMD)
   - Shared memory without threads
   - Shared memory with Multithreading

Concepts and Terminology

Concepts and Terminology

CPU/Processor/Core while technically nowadays each CPU/processor hosts more than one core, we use this terms interchangeably

Node A 'standalone' unit consisting of its own CPUs, memory (& storage).

Process/Task logically discrete section of computational work - typically a program or program-like set of instructions that is executed by a processor

Thread part of the computational work of a process that is executed in parallel (on an additional processor)

Concepts and Terminology

CPU/Processor/Core while technically nowadays each CPU/processor hosts more than one core, we use this terms interchangeably

Node A 'standalone' unit consisting of its own CPUs, memory (& storage).

Process/Task logically discrete section of computational work - typically a program or program-like set of instructions that is executed by a processor

Thread part of the computational work of a process that is executed in parallel (on an additional processor)

Massively Parallel Refers to the hardware that comprises a given parallel system - having many processing elements (the meaning of "many" keeps increasing)

Embarrassingly Parallel Solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks

Concepts and Terminology (cont.)

Wall(-clock) time "physical" time (i.e. time measured on a stopwatch)

Throughput amount of (sub)tasks/data processed per (wall) time unit

Latency (wall time) delay between invoking the operation and getting the response (e.g. finishing a task)

Observed speed-up ratio between wall time of serial and parallelized code

Concepts and Terminology (cont.)

Wall(-clock) time "physical" time (i.e. time measured on a stopwatch)

Throughput amount of (sub)tasks/data processed per (wall) time unit

Latency (wall time) delay between invoking the operation and getting the response (e.g. finishing a task)

Observed speed-up ratio between wall time of serial and parallelized code

CPU time time a process spent running on CPUs (with each used CPU adding to it)

Parallel overhead Additional amount of (CPU/wall) time/resources required to run parallelized code (e.g. start-up time and memory usage of framework, data comm., synchronization)

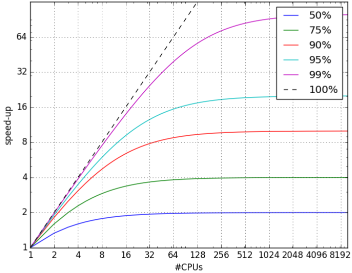
Concepts and Terminology - Amdahl's Law

- theoretical speedup in *latency* S_{latency} of execution of task with fixed workload:

Amdahl's law

$$S_{\text{latency}} = \frac{1}{(1-p) + \frac{p}{s}}$$

p is parallelizable fraction of code
s its speed-up



Concepts and Terminology - Amdahl's Law

- theoretical speedup in *latency* S_{latency} of execution of task with fixed workload:

Amdahl's law

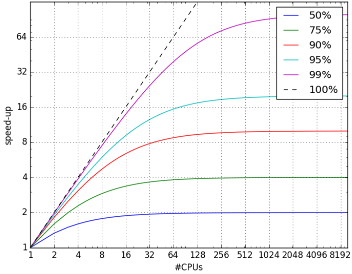
$$S_{\text{latency}} = \frac{1}{(1-p) + \frac{p}{s}}$$

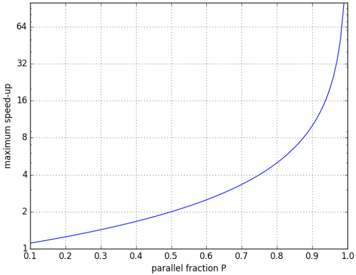
p is parallelizable fraction of code
s its speed-up

- From this follows

$$\lim_{s \to \infty} S_{\text{latency}} = \frac{1}{1-p}$$

i.e. never speeds up more than the inverse serial fraction of code





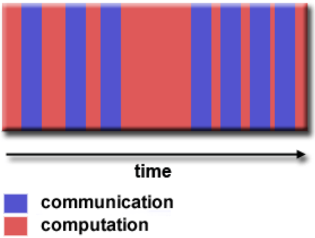
Concepts and Terminology - Granularity

- Computation / Communication Ratio

Concepts and Terminology - Granularity

- Computation / Communication Ratio

  fine-grained: frequent communication; facilitates e.g. load balancing, comes with overhead costs

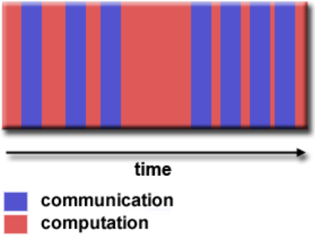


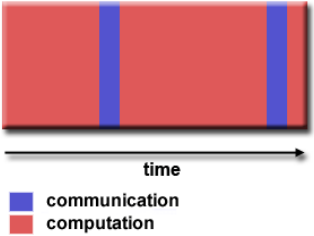
Concepts and Terminology - Granularity

- Computation / Communication Ratio

  fine-grained frequent communication; facilitates e.g. load balancing, comes with overhead costs

  coarse-grained less frequent comm.; lower communication costs, but potentially poorer load balancing





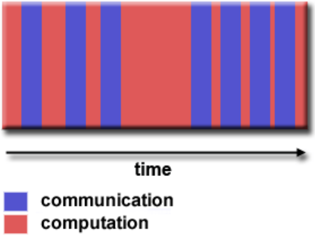
Concepts and Terminology - Granularity

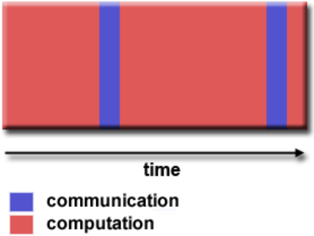
- Computation / Communication Ratio

  fine-grained frequent communication; facilitates e.g. load balancing, comes with overhead costs

  coarse-grained less frequent comm.; lower communication costs, but potentially poorer load balancing

- best choice dependents on circumstances





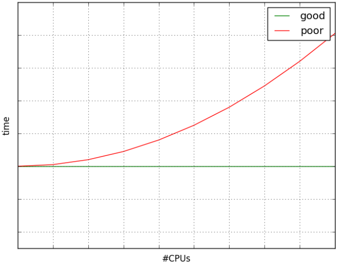
Concepts and Terminology - Scalability

Ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources:

Concepts and Terminology - Scalability

Ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources:

Weak scaling for running larger problem while fixing the problem size per processor

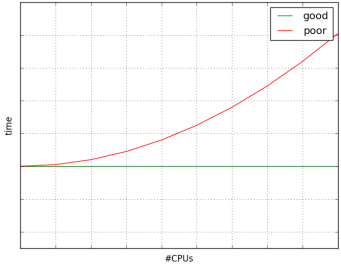


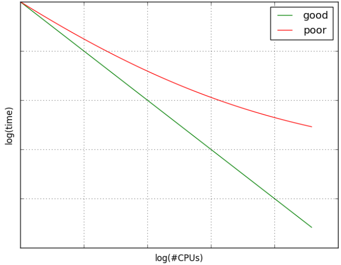
Concepts and Terminology - Scalability

Ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources:

Weak scaling for running larger problem while fixing the problem size per processor

Strong scaling for running the same problem size in less time



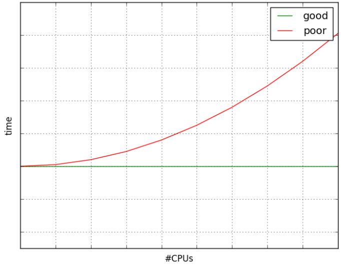


Concepts and Terminology - Scalability

Ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources:

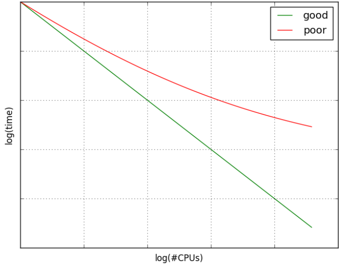
Weak scaling for running larger problem while fixing the problem size per processor

Strong scaling for running the same problem size in less time



Factors affecting scalability:

- I/O bandwidth (for RAM, storage and communication)
- imperfect/impossible load balancing
- overhead on comm. (e.g. exchange of padding around domain)
- limitations of parallel support libraries / parallel overhead



Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but CPU time and memory requirements increase). Additionally, the increased complexity comes with increased development costs for:

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but CPU time and memory requirements increase). Additionally, the increased complexity comes with increased development costs for:

- Design

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but CPU time and memory requirements increase). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but CPU time and memory requirements increase). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding
- Debugging

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but CPU time and memory requirements increase). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding
- Debugging
- Tuning

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but CPU time and memory requirements increase). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding
- Debugging
- Tuning
- Maintenance

Concepts and Terminology - Cost of Complexity

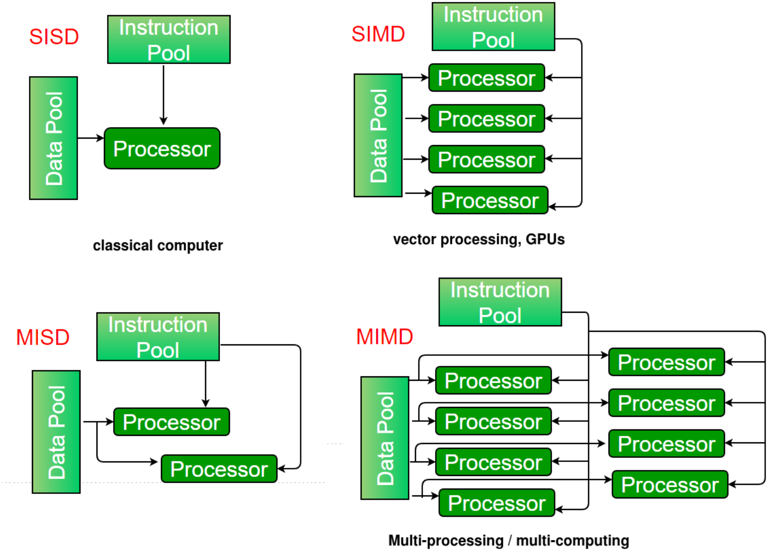
Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but CPU time and memory requirements increase). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding
- Debugging
- Tuning
- Maintenance

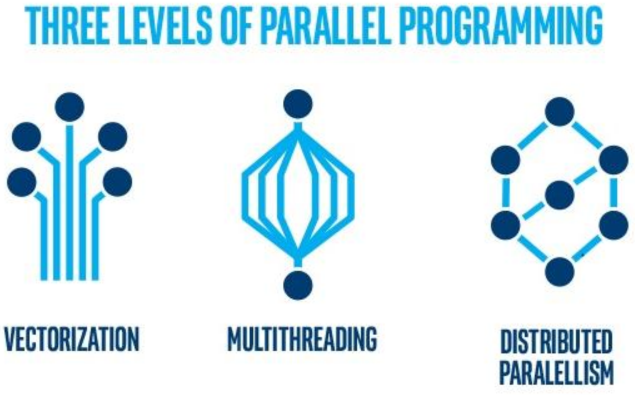
You have to find a trade-off between development time and runtime. Make sure the development of a speed-up does not cost you more time/resources than it saves you in the end!

Parallel Programming Models

Recap: Computer Architectures - Flynn's taxonomy

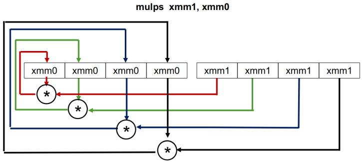


Parallel Programming Models - Overview



Recap: Processor - Supplementary instruction sets (SIMD/Vectorization)

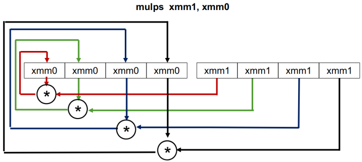
Processors provide special hardware extensions to perform certain operations faster/more efficiently:



SSE2/SSE3/SSE4.x (since Pentium 4 / Xeon) SIMD instruction set for SP/DP floats, long/standard/short integers, chars; additional registers

Recap: Processor - Supplementary instruction sets (SIMD/Vectorization)

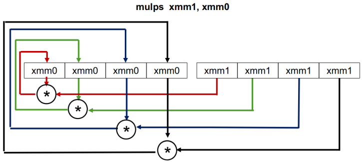
Processors provide special hardware extensions to perform certain operations faster/more efficiently:



SSE2/SSE3/SSE4.x	(since Pentium 4 / Xeon) SIMD instruction set for SP/DP floats, long/standard/short integers, chars; additional registers
AVX	extension to SSE extensions (sciama2.q)
AVX2	extension of SSE/AVX operations to 256 bits (sciama3.q)
AVX-512	extension of SSE/AVX operations to 512 bits (sciama4.q)

Recap: Processor - Supplementary instruction sets (SIMD/Vectorization)

Processors provide special hardware extensions to perform certain operations faster/more efficiently:

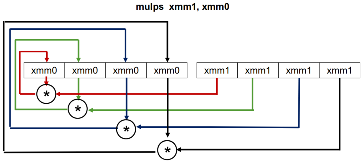


SSE2/SSE3/SSE4.x	(since Pentium 4 / Xeon) SIMD instruction set for SP/DP floats, long/standard/short integers, chars; additional registers
AVX	extension to SSE extensions (sciama2.q)
AVX2	extension of SSE/AVX operations to 256 bits (sciama3.q)
AVX-512	extension of SSE/AVX operations to 512 bits (sciama4.q)

What to consider when using these extensions:

Recap: Processor - Supplementary instruction sets (SIMD/Vectorization)

Processors provide special hardware extensions to perform certain operations faster/more efficiently:



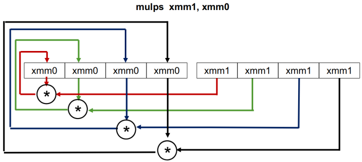
SSE2/SSE3/SSE4.x	(since Pentium 4 / Xeon) SIMD instruction set for SP/DP floats, long/standard/short integers, chars; additional registers
AVX	extension to SSE extensions (sciama2.q)
AVX2	extension of SSE/AVX operations to 256 bits (sciama3.q)
AVX-512	extension of SSE/AVX operations to 512 bits (sciama4.q)

What to consider when using these extensions:

users try to compile code newest instruction set supported by the CPU you are planning to run on

Recap: Processor - Supplementary instruction sets (SIMD/Vectorization)

Processors provide special hardware extensions to perform certain operations faster/more efficiently:



SSE2/SSE3/SSE4.x	(since Pentium 4 / Xeon) SIMD instruction set for SP/DP floats, long/standard/short integers, chars; additional registers
AVX	extension to SSE extensions (sciama2.q)
AVX2	extension of SSE/AVX operations to 256 bits (sciama3.q)
AVX-512	extension of SSE/AVX operations to 512 bits (sciama4.q)

What to consider when using these extensions:

- **users** try to compile code newest instruction set supported by the CPU you are planning to run on
- **coders** try to organize code in a way to make best use of SIMD

Parallel Programming Models - SIMD / Vectorization

Scalar version	Vectorized version

`int A[], B[], C[];`
`...`
`for(i=0; i<n; i++) {`
`  a = A[i];`
`  b = B[i];`
`  c = a+b;`
`  C[i] = c;`
`}`
	`int A[], B[], C[];`
	`...`
	`/* vectorized loop */`
	`for(i=0; i<n; i+=vf) {`
	`  va = A[i..i+vf[;`
	`  vb = B[i..i+vf[;`
	`  vc = padd(va, vb);`
	`  C[i..i+vf[ = vc;`
	`}`
	`/* epilogue */`
	`for( ; i<n; i++) {`
	`  /* remaining iterations */`
	`}`

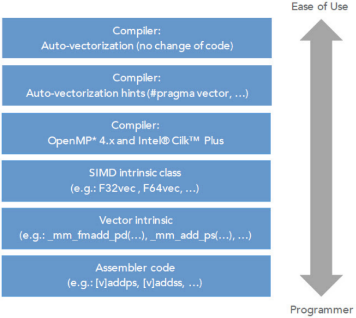
Parallel Programming Models - SIMD / Vectorization

Scalar version

```
int A[], B[], C[];
...
for(i=0; i<n; i++) {
  a = A[i];
  b = B[i];
  c = a+b;
  C[i] = c;
}
```

Vectorized version

```
int A[], B[], C[];
...
/* vectorized loop */
for(i=0; i<n; i+=vf) {
  va = A[i..i+vf[;
  vb = B[i..i+vf[;
  vc = padd(va, vb);
  C[i..i+vf[ = vc;
}
/* epilogue */
for( ; i<n; i++) {
  /* remaining iterations */
}
```



- various ways of using vectorization:
  - automatic optimization by compiler (-O2 and higher)
  - assisted auto-vectorization using special statements
  - explicit vectorization (cf. OpenMP SIMD, ASM)

Parallel Programming Models - SIMD / Vectorization

- data dependencies preventing (auto-)vectorization e.g.

Parallel Programming Models - SIMD / Vectorization

- data dependencies preventing (auto-)vectorization e.g.
  Loop Dependencies :

```
for (i=1; i<end; i++)
    C[i] = C[i-1]+B[i-1]
```

Parallel Programming Models - SIMD / Vectorization

- data dependencies preventing (auto-)vectorization e.g.

  Loop Dependencies :

```
for (i=1; i<end; i++)
    C[i] = C[i-1]+B[i-1]
```

  Indirect Memory Access :

```
for (i=1; i<end; i++)
    C[idxC[i]] = A[i]+b[i]
```

Parallel Programming Models - SIMD / Vectorization

- data dependencies preventing (auto-)vectorization e.g.

  Loop Dependencies :

```
for (i=1; i<end; i++)
    C[i] = C[i-1]+B[i-1]
```

  Indirect Memory Access :

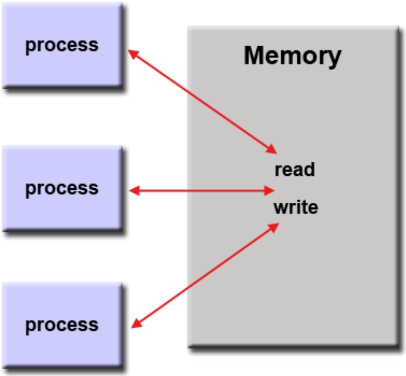
```
for (i=1; i<end; i++)
    C[idxC[i]] = A[i]+b[i]
```

  Non 'Straight line' code :

```
for (i=1; i<CalcEnd(); i++) {
    if (DoJump()) i+= CalcJump();
    C[i] = A[i]+B[i];
}
```

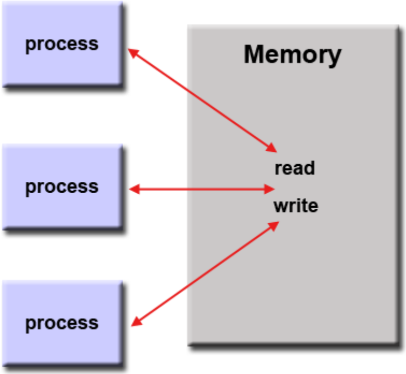
Parallel Programming Models - Shared memory without Threads

- simplest parallel programming model



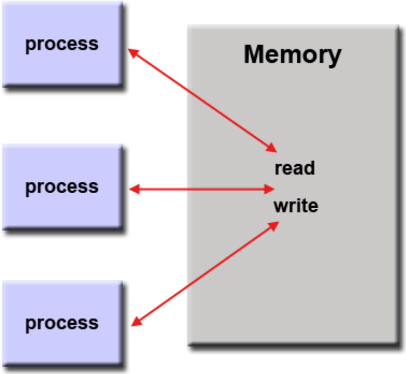
Parallel Programming Models - Shared memory without Threads

- simplest parallel programming model
- processes share common address space



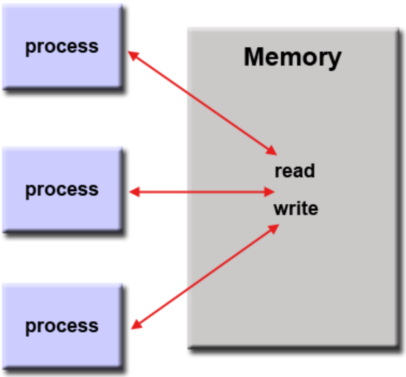
Parallel Programming Models - Shared memory without Threads

- simplest parallel programming model
- processes share common address space
- access to shared memory has to be controlled to prevent race conditions and deadlocks (see later)



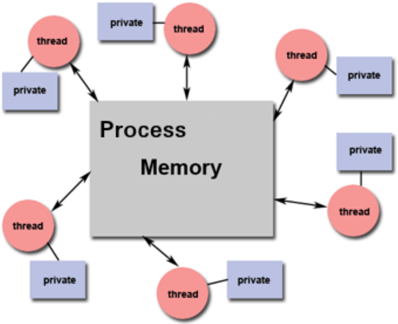
Parallel Programming Models - Shared memory without Threads

- simplest parallel programming model
- processes share common address space
- access to shared memory has to be controlled to prevent race conditions and deadlocks (see later)
- while not very common in use, e.g. POSIX standards provide API, UNIX provides shared memory segments



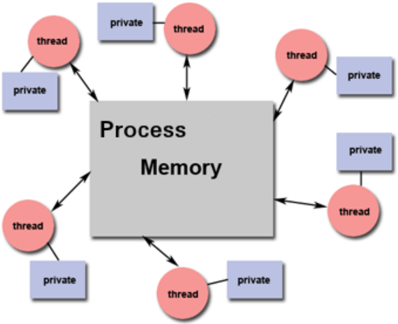
Parallel Programming Models - Shared memory with Multithreading

- In Multithreading, a single "heavy weight" process can have multiple "light weight", concurrent execution paths (threads).



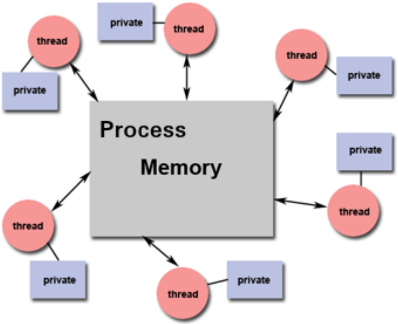
Parallel Programming Models - Shared memory with Multithreading

- In Multithreading, a single "heavy weight" process can have multiple "light weight", concurrent execution paths (threads).
- A thread's work may best be described as a subroutine within the main program.



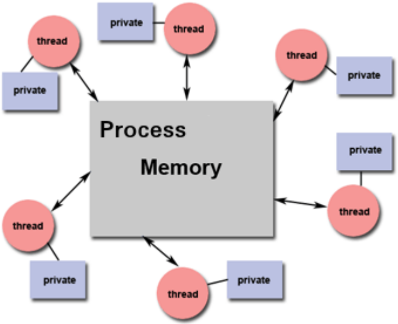
Parallel Programming Models - Shared memory with Multithreading

- In Multithreading, a single "heavy weight" process can have multiple "light weight", concurrent execution paths (threads).
- A thread's work may best be described as a subroutine within the main program.
- Any thread can execute any subroutine at the same time as other threads.



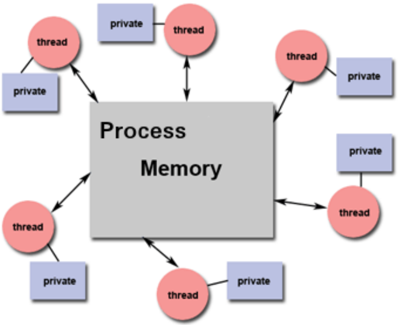
Parallel Programming Models - Shared memory with Multithreading

- In Multithreading, a single "heavy weight" process can have multiple "light weight", concurrent execution paths (threads).
- A thread's work may best be described as a subroutine within the main program.
- Any thread can execute any subroutine at the same time as other threads.
- Each thread has local data, but also, shares the entire resources of its parent process i.e. saves replicating a program's resources for each thread ("light weight").



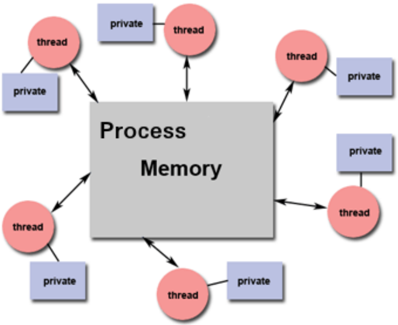
Parallel Programming Models - Shared memory with Multithreading (cont.)

- Each thread also benefits from global memory view; it shares memory space of the host process (requires **access control**!)



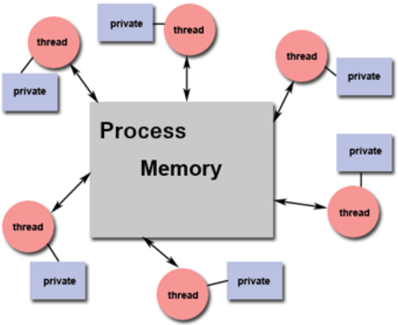
Parallel Programming Models - Shared memory with Multithreading (cont.)

- Each thread also benefits from global memory view; it shares memory space of the host process (requires **access control**!)
- Threads communicate with each other through global memory.



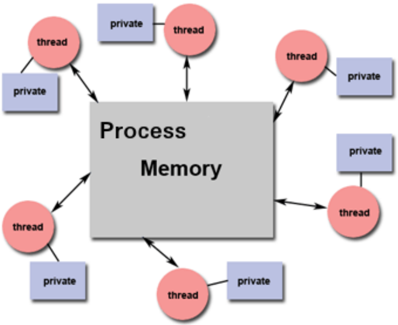
Parallel Programming Models - Shared memory with Multithreading (cont.)

- Each thread also benefits from global memory view; it shares memory space of the host process (requires **access control**!)
- Threads communicate with each other through global memory.
- Threads can be spawned and terminated at any time in the host process, but the host process remains present to provide the necessary shared resources until the application has completed.

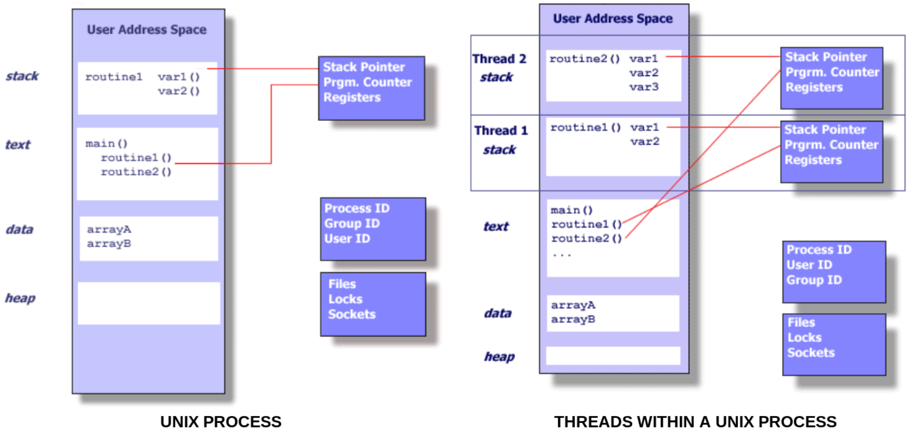


Parallel Programming Models - Shared memory with Multithreading (cont.)

- Each thread also benefits from global memory view; it shares memory space of the host process (requires **access control**!)
- Threads communicate with each other through global memory.
- Threads can be spawned and terminated at any time in the host process, but the host process remains present to provide the necessary shared resources until the application has completed.
- We will focus on two standards: **OpenMP** & **POSIX Threads**



Parallel Programming Models - Shared memory with Multithreading (cont.)



To be continued ...