HPC Performance & Architectures

2024 NSF CyberTraining Workshop

Jan. 8, 2024 – Jan. 19, 2024

Clarkson University

Topics of this Lecture

- Performance Evaluation
- HPC Architectures
- Process vs. Thread

Performance

- **Determine which computer is best suited for a given (set of) application(s)?**
  - Gaming PC or MacBook Pro?
  - Cluster or fat server? Fast CPU? Intel or AMD or GPU???
  - Which applications? Which input/data sets?
- **Validate impact of new optimization / implementation / parallelization strategy and present to others**
  - Results need to be interpreted and potentially reproduced by external people
  - Compare with other / previous work
  - Justify efficient usage of expensive resources
- **Determine capabilities for individual parts of the computer**
  - Data transfer / IO / computational capabilities
  - Often required to guide optimization strategies

Performance - Metrics

- Performance = **WORK** / **TIME**

- "Pure" metrics – basic choices for "**WORK**"
  - **MFlop/s**: Millions of Floating Point Operations per Second
  - **MFlop/s** = Number of Floating Point Operations executed / 10^6 * **TIME**

  - **MIPS**: Millions of Instructions per Second
  - **MIPS** = Number of Instructions Executed / 10^6 * **TIME**

- How to determine **WORK**, e.g. "Floating Point Operations"
  - Count them manually (high level code / algorithm)
  - Use CPUs event counter → e.g., LIKWID Toolkit (like likwid-perfscope on Ubuntu)

Performance Optimization

- "My vector update code runs at 2,000 MFlop/s on a 2GHz processor!
- Great – isn't it?

```
for(i=0; i<n; i++)
{
   a[i]= 3.0*c0+c1*c2 +c3*c4*a[i] -d0 *a[i];
}
```

→ #FLOP = 8 * n

#FLOP has been reduce to only ¼!

```
d0 = 3.0*c0+c1*c2; d1 = c3*c4-d0;

for(i=0; i<n; i++)
{
    a[i]= d0 + d1*a[i];
}
```

→ #FLOP = 2* n + 5

→ Define **WORK** carefully – independent of implementation issues

Performance – Metric Choice

- **Iterations:** Total number of loop iterations performed: WORK = n iterations
→ Performance metric: **Iterations / s**

- **Lattice Site/ Cell / Particle Updates:** Often used for stencil codes or Lattice Boltzmann fluid solvers: WORK = number of sites/cells/particles to be updated/computed
→ Performance metric: **Cell updates / s**

- **Physical simulation time:** Often used in molecular dynamics codes: WORK = Physical time (e.g. nanosenconds) a system is propagated
→ Performance metric: **nanoseconds / day**

- **Complete problem solution:** WORK: ”1” well defined problem
→ Performance metric: **1 / s**

Performance – Time

- Simplest performance metric ("Bestseller"): 1 / TIME
  - Measures time to solution
  - Carefully specify the "problem" you solved!
  - Best metric thinkable, but not intuitive in all situations

- Problem: Which TIME?
- LINUX / UNIX command `time`:

```
yuliu@yuliu-server:~$ time sleep 30
real 0m30.003s
user 0m0.002s
sys 0m0.001s
```

real refers to actual elapsed time; **user** and **sys** refer to CPU time used only by the process.

Performance – Time (Cont.)

- **Stay away from CPU time – it's evil!**
- Elapsed time (walltime) is the time you wait for your result!
- Measuring `walltime` within code on UNIX (-like) systems
  - Use `gettimeofday()` to measure timestamps:

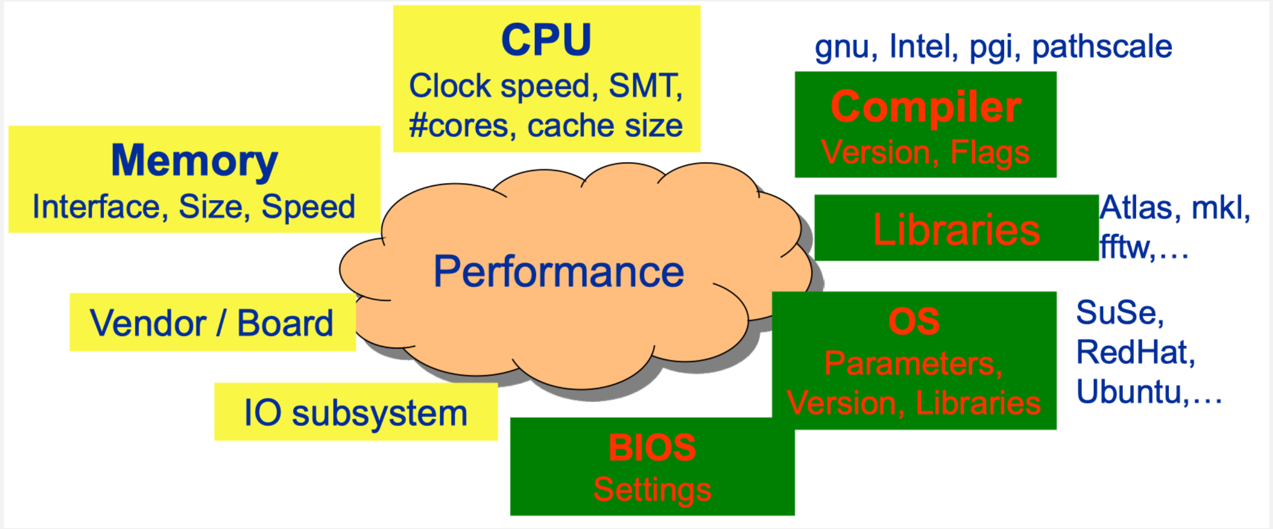
```
#include <sys/time.h>

double timestamp(void){
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return((double)(tp.tv_sec + tp.tv_usec/1000000.0)); }
```

- WALL**TIME**:= Difference of two timestamps!

Performance – Impact Factors

- For a given code/problem performance may be influenced by many factors



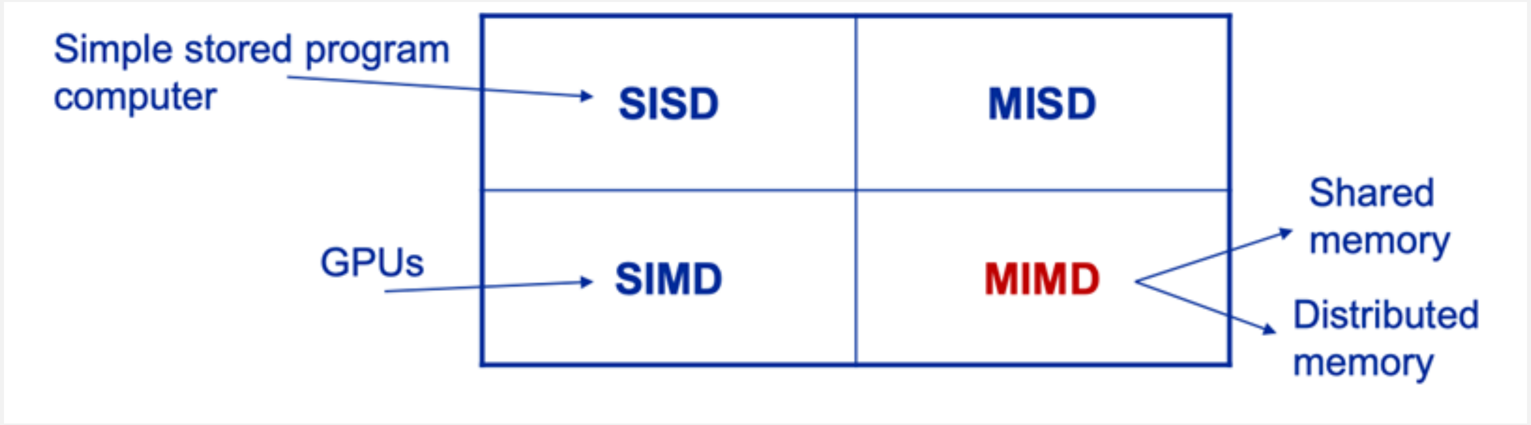
- For reproducibility of performance results all critical factors need to be reported!
- Sensibility and stability analysis!
- Statistics - fluctuations between runs

Topics of this Lecture

- Performance Evaluation
- HPC Architectures
- Process vs. Thread

Parallel Computers

- **Parallel Computing**: A number of compute elements solve a problem in a cooperative way
- **Parallel Computer**: A number of compute elements connected such way to do parallel computing for a large set of applications
- Classification according to Flynn: SISD, MISD, SIMD, Multiple Instruction Multiple Data (MIMD)



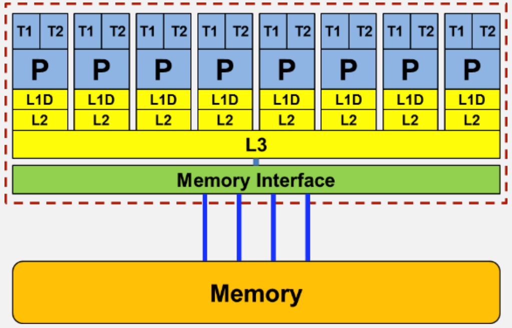
Parallel Computers - Classifications

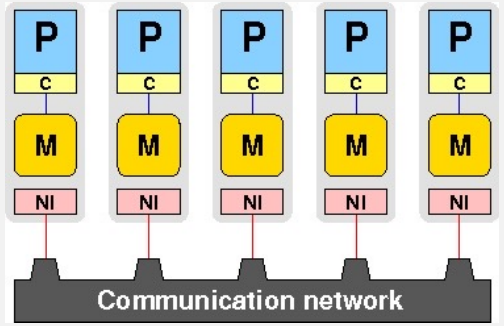
Classification according to address space organization:

- Shared-memory Architectures:
Cache-Coherent Single Address Space

- Distributed-memory Architectures
No (Cache-Coherent) Single Address Space

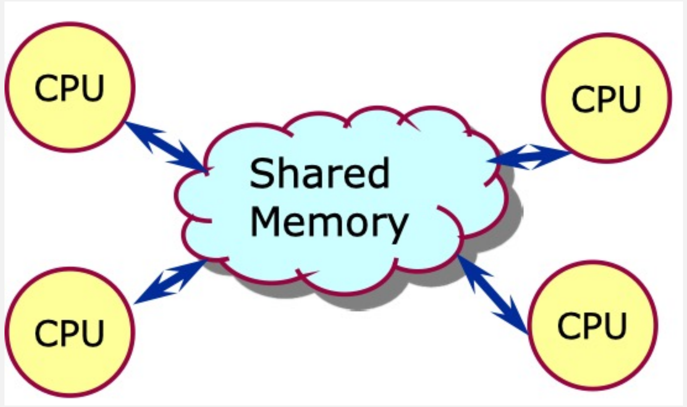
- Hybrid architectures containing both concepts are state-of-the art





Shared-Memory Arch.

- **Shared memory computers provide**
  - **Single shared address space for all processors**
  - All processors share the same view of the address space!



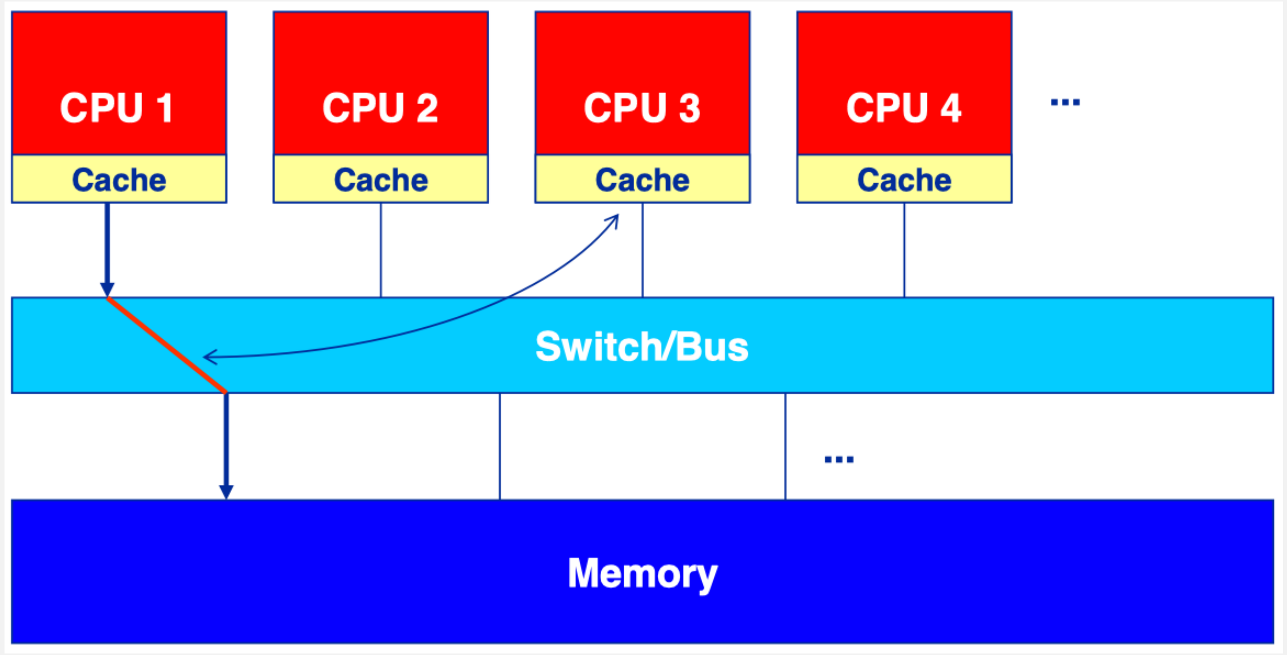
- **Two basic categories of shared memory systems**
  - **Uniform Memory Access (UMA):**

    Memory is equally accessible to all processors with the same performance (Bandwidth & Latency)

  - **Cache-coherent Non Uniform Memory Access (ccNUMA)**: Memory is physically distributed: Performance (Bandwidth & Latency) is different for local and remote memory access

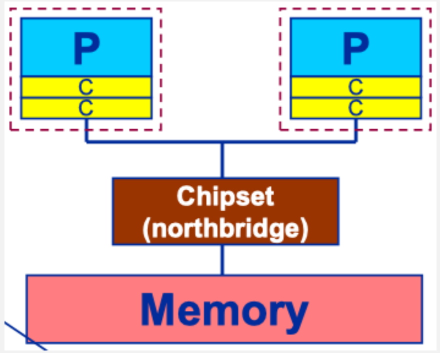
Shared-memory: UMA

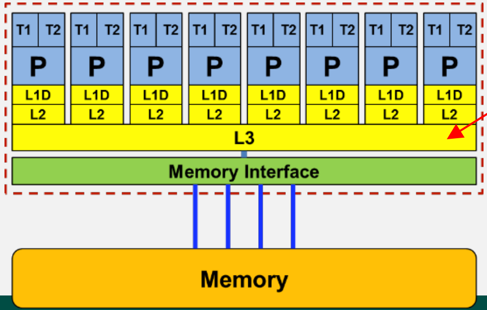
- **UMA Architecture: switch/bus arbitrates memory access**
  - Special protocol ensures cross-CPU cache data consistency
  - Flat memory – also known as “Symmetric Multi-Processor” (SMP)

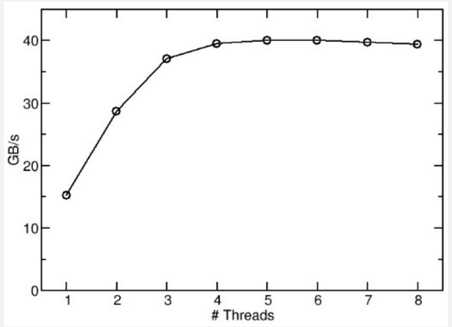


Shared-memory: UMA/Bus

- **Worst case: bus system provides single bandwidth to multiple processors**
- Only "**one consumer**" at a time can use the bus and access memory at any one time – No need to provide for faster memory
- Collisions occur frequently, causing one or more CPUs to wait for "bus ready" (contention) → Saturation
- Multi-core architectures: "consumer" is the L3 cache (due to L3 cache misses)



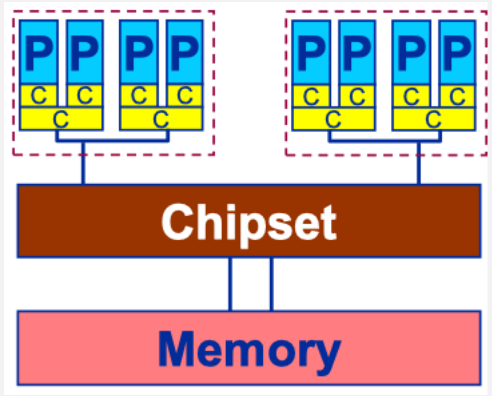




Shared Memory: UMA/Crossbar

- **Best case: memory crossbar switch provides separate data path to memory for each CPU**
  - Can saturate full memory bandwidth of every CPU concurrently

    (→ Bandwidth is "parallel" resource)
  - Contention only if same memory module/bank is accessed by multiple CPUs

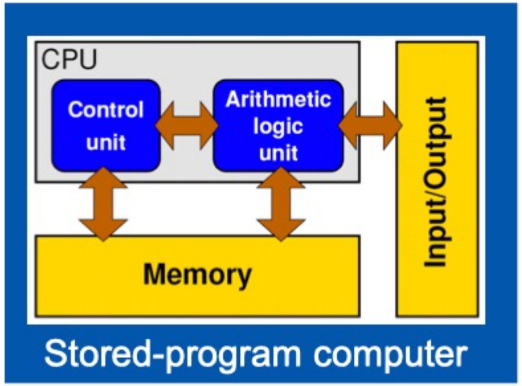


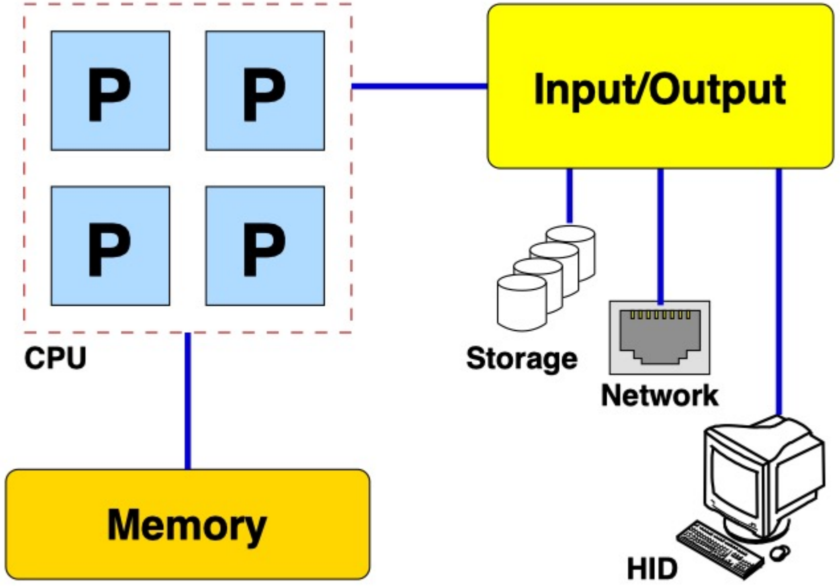
Shared-memory: UMA Nodes

- **Examples:**
  - Intel/AMD Dual-/quad-/hexa-/octo-/…/22-core laptop/desktop/server processor
  - IBM BlueGene series
  - NEC vector systems
  - NVIDIA GPUs
  - Intel Xeon Phi (KNC, KNL,…)
- **Advantages**
  - Cache Coherence is "easy" to implement
  - Easy to optimize memory access
  - Incremental parallelization
- **Disadvantages**
  - Memory bandwidth and **price** (!) often limit scalability (2 – 20 cores per UMA node)

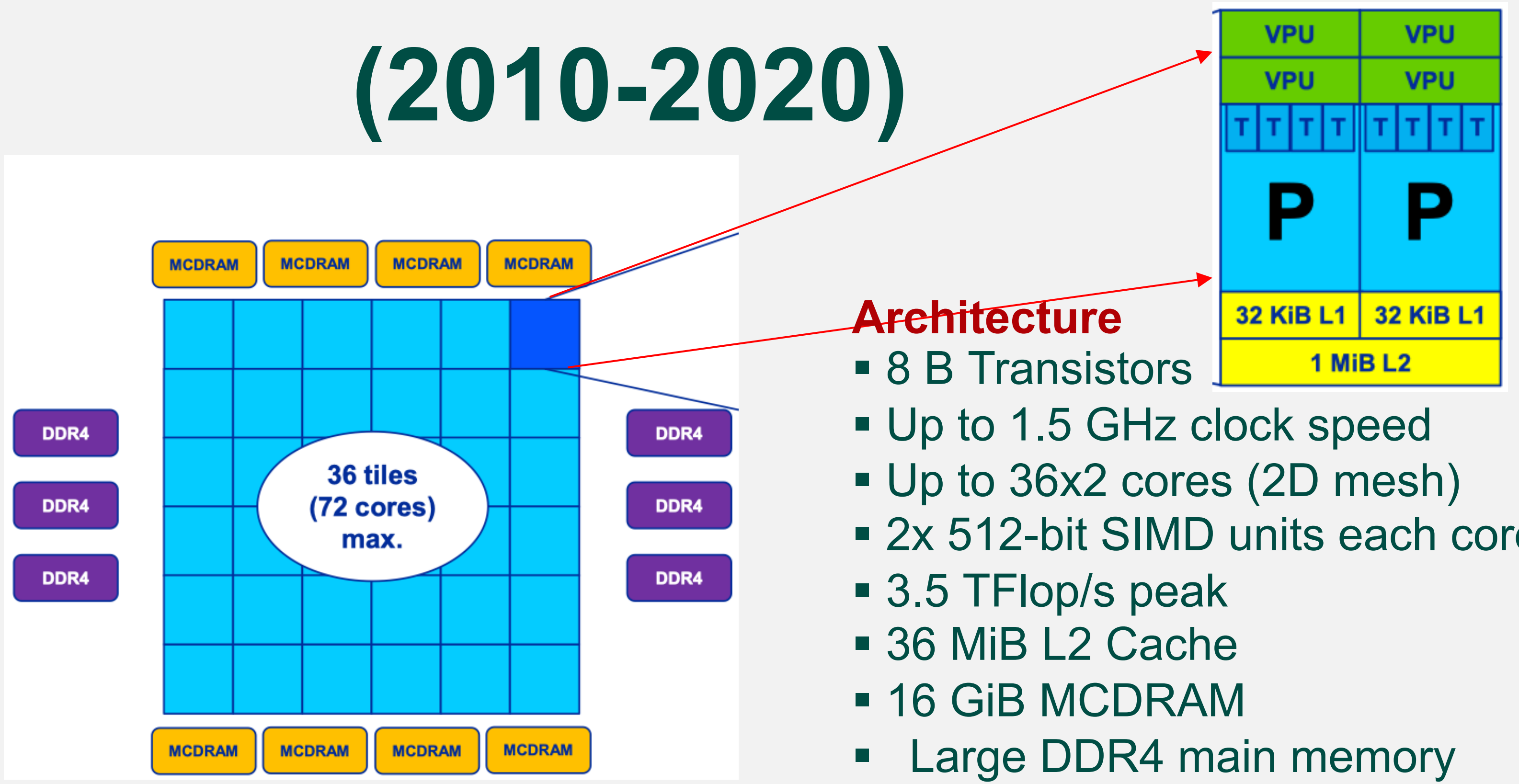
Basic Computer Concept

- Stored Program Computer” concept (Turing 1936)
- Similar designs on all modern systems





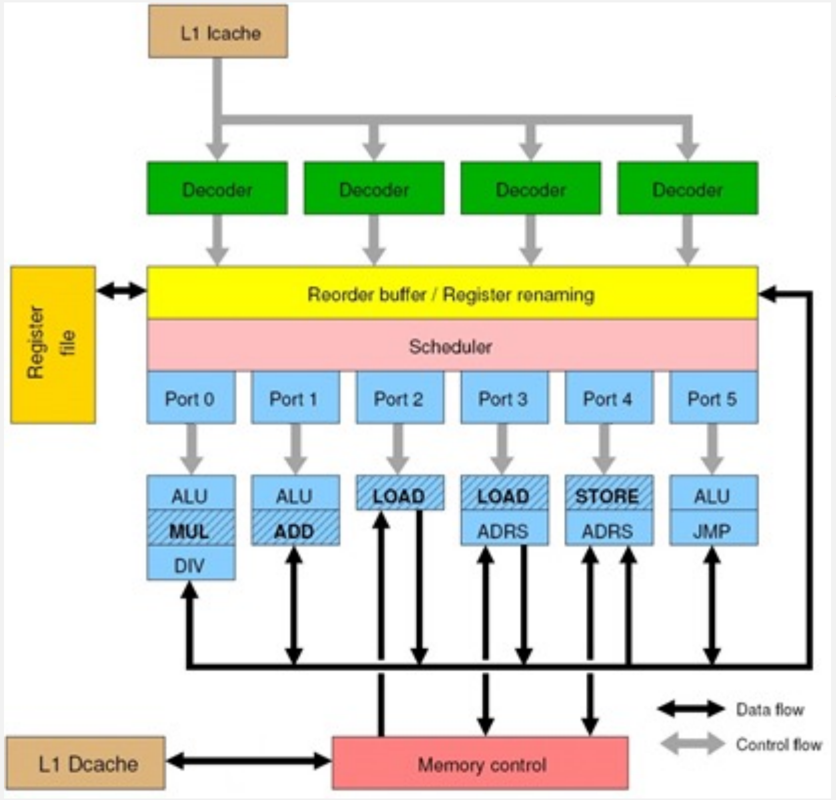
Intel Xeon Phi CPU (2010-2020)



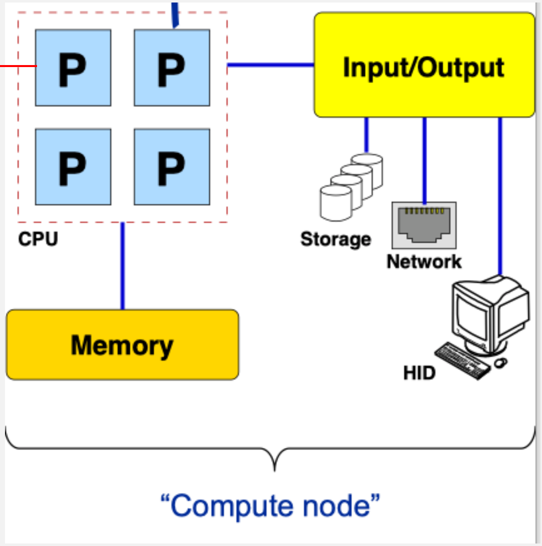
Architecture

- 8 B Transistors
- Up to 1.5 GHz clock speed
- Up to 36x2 cores (2D mesh)
- 2x 512-bit SIMD units each core
- 3.5 TFlop/s peak
- 36 MiB L2 Cache
- 16 GiB MCDRAM
- Large DDR4 main memory
- Built Tianhe-2 supercomputer

CPU Core

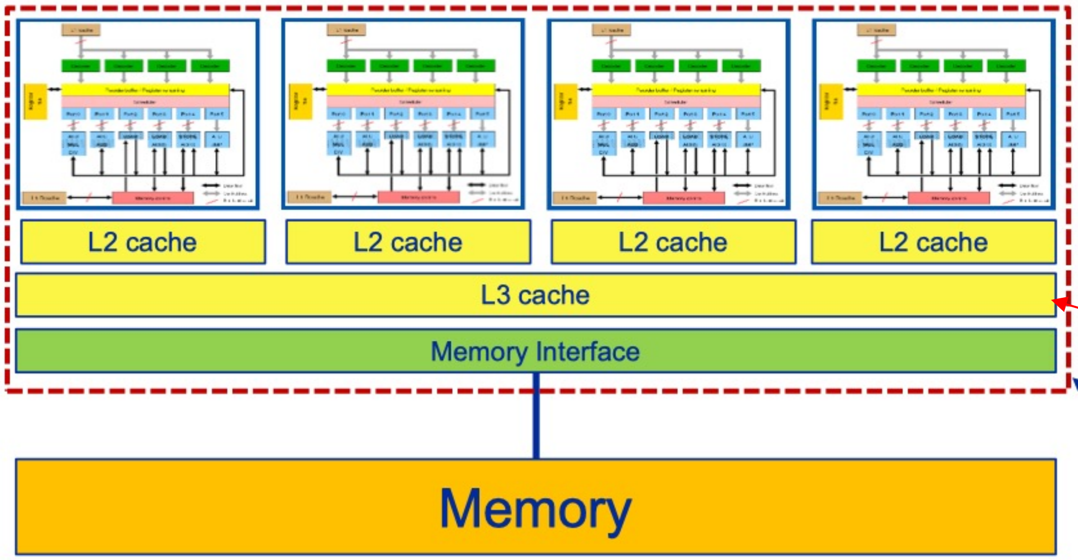


L1 Icache
Decoder
Decoder
Decoder
Decoder
Register file
Reorder buffer / Register renaming
Scheduler
Port 0
Port 1
Port 2
Port 3
Port 4
Port 5
ALU
MUL
DIV
ALU
ADD
LOAD
LOAD
ADRS
STORE
ADRS
ALU
JMP
Data flow
Control flow
L1 Dcache
Memory control



P
P
P
P
CPU
Input/Output
Storage
Network
Memory
HID
"Compute node"

Multicore CPU



“Socket”

Shared L3 Cache

Instruction Execution

This is the primary resource of the processor. All efforts in hardware design are targeted towards increasing the instruction throughput.

Instructions are the concept of “work” as seen by processor designers.
Not all instructions count as “work” as seen by application developers!

Example: Adding two arrays **`A(:)`** and **`B(:)`**

```
do i=1,N
  A(i)=A(i)+B(i)
enddo
```

User work:
N Flops (ADDs)

Processor work:

```
LOAD r1 = A(i)
LOAD r2 = B(i)
ADD r1 = r1 + r2
STORE A(i) = r1
INCREMENT i
BRANCH → top if i<N
```

Data Transfer

Data transfers are a consequence of instruction execution and therefore a secondary resource. Maximum bandwidth is determined by the request rate of executed instructions and technical limitations (bus width, speed).

Example: Adding two arrays `A(:)` and `B(:)`

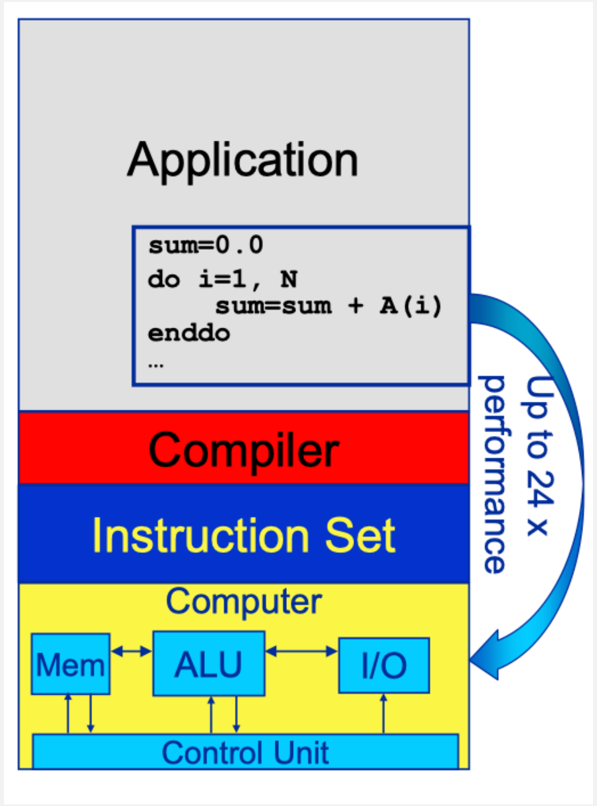
```
do i=1,N
  A(i)=A(i)+B(i)
enddo
```

Data transfers:
8 byte: `LOAD r1 = A(i)`
8 byte: `LOAD r2 = B(i)`
8 byte: `STORE A(i) = r2`
Sum: **24 byte**

Crucial question: What determines the runtime?

- Data transfer?
- Code execution?
- Something else?

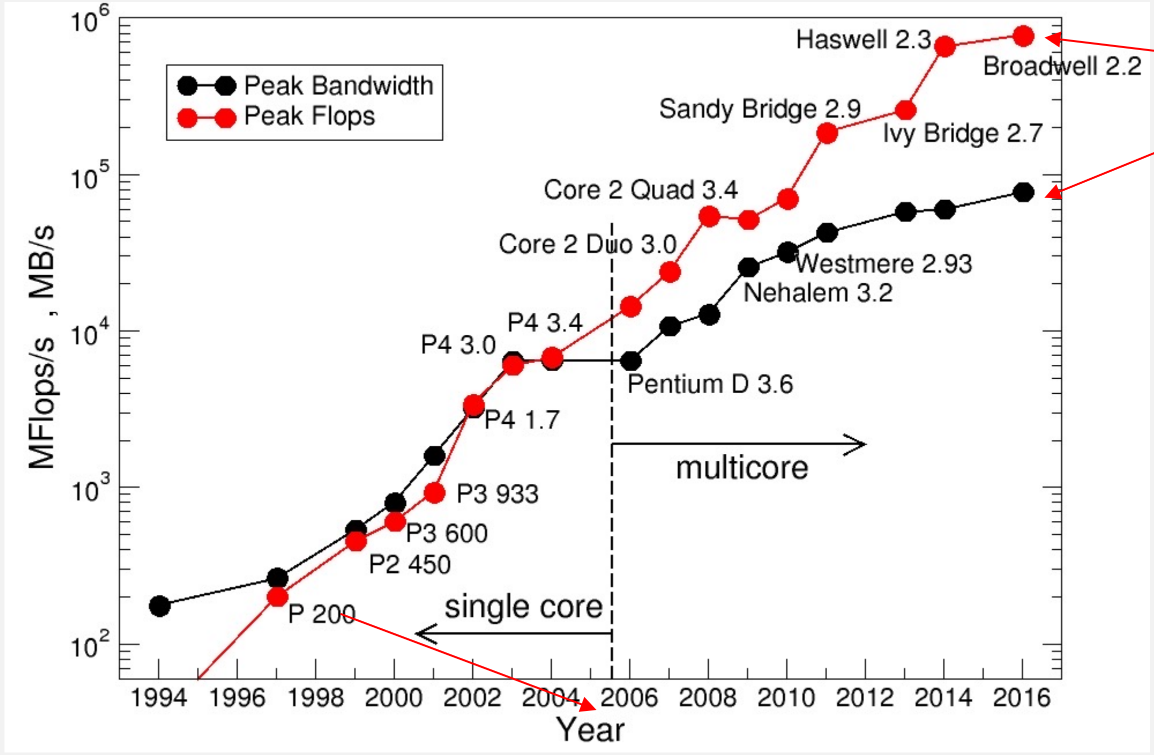
From Application to CPU



```
sum=0.0
do i=1, N
     sum=sum + A(i)
enddo
…
```

- Application: High Level Programming Language (e.g. C / C++ / Fortran) – portable
- Compiler translates program to Instruction set (architecture) (IA32, Intel 64, AMD64 a.k.a. x86, x86_64)
- Hardware specific execution of Instruction Set Architecture (ISA)

DRAM Gap



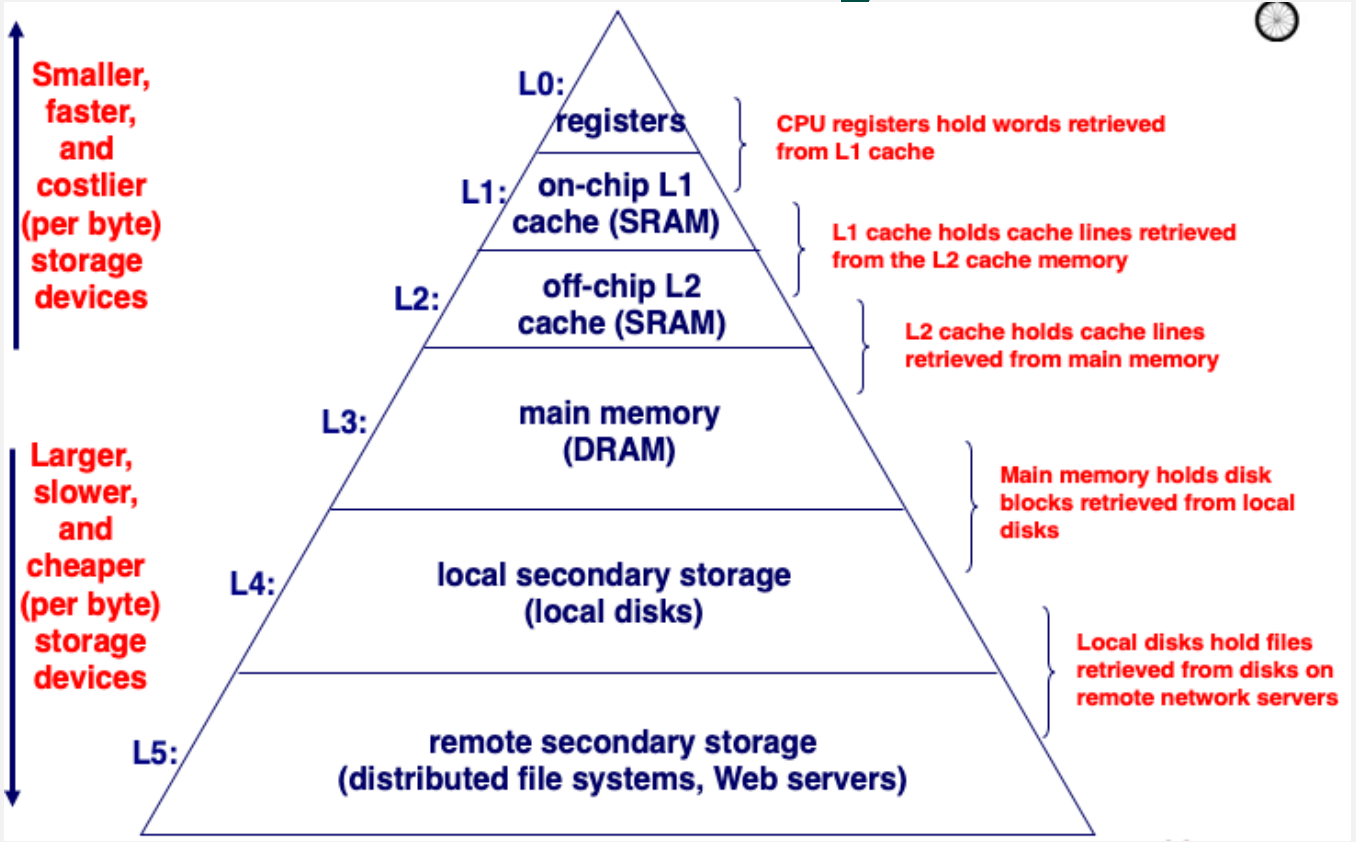
Approx. 10 F/B

Main memory access speed not sufficient to keep CPU busy…

→ Introduce fast on-chip caches, holding copies of recently used data items

Clarkson University defy convention

Solution - Memory Hierarchy

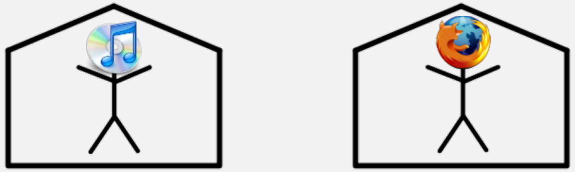

Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0: registers
L1: on-chip L1 cache (SRAM)
L2: off-chip L2 cache (SRAM)
L3: main memory (DRAM)
L4: local secondary storage (local disks)
L5: remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from L1 cache
L1 cache holds cache lines retrieved from the L2 cache memory
L2 cache holds cache lines retrieved from main memory
Main memory holds disk blocks retrieved from local disks
Local disks hold files retrieved from disks on remote network servers

Topics of this Lecture

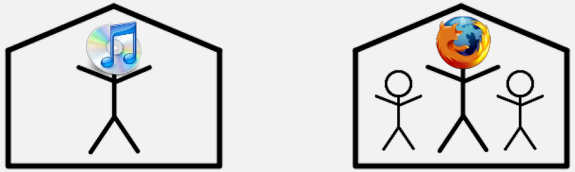
- Performance Evaluation
- HPC Architectures
- Process vs. Thread

Definitions: threads vs. processes

- A *process* is a "program" with its own address space.
  - A process has at least one thread!



- A *thread of execution* is an independent sequential computational task with its own control flow, stack, registers, etc.
  - There can be many threads in the same process sharing the same address space

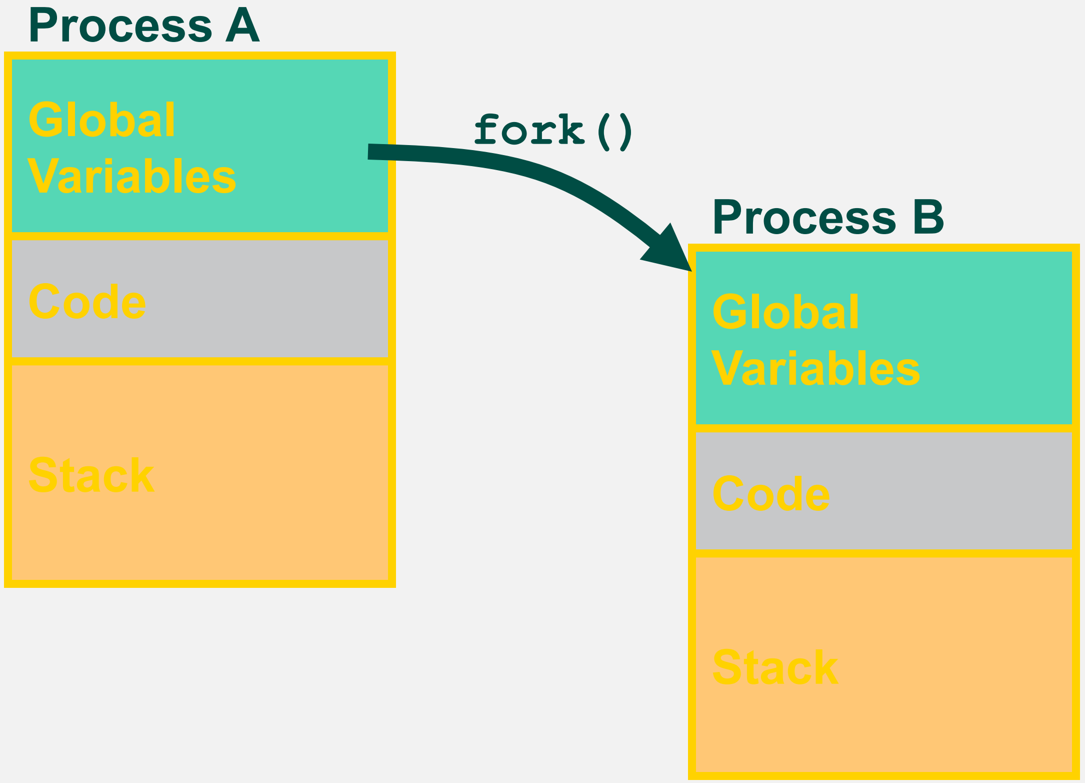


Threads vs. Processes

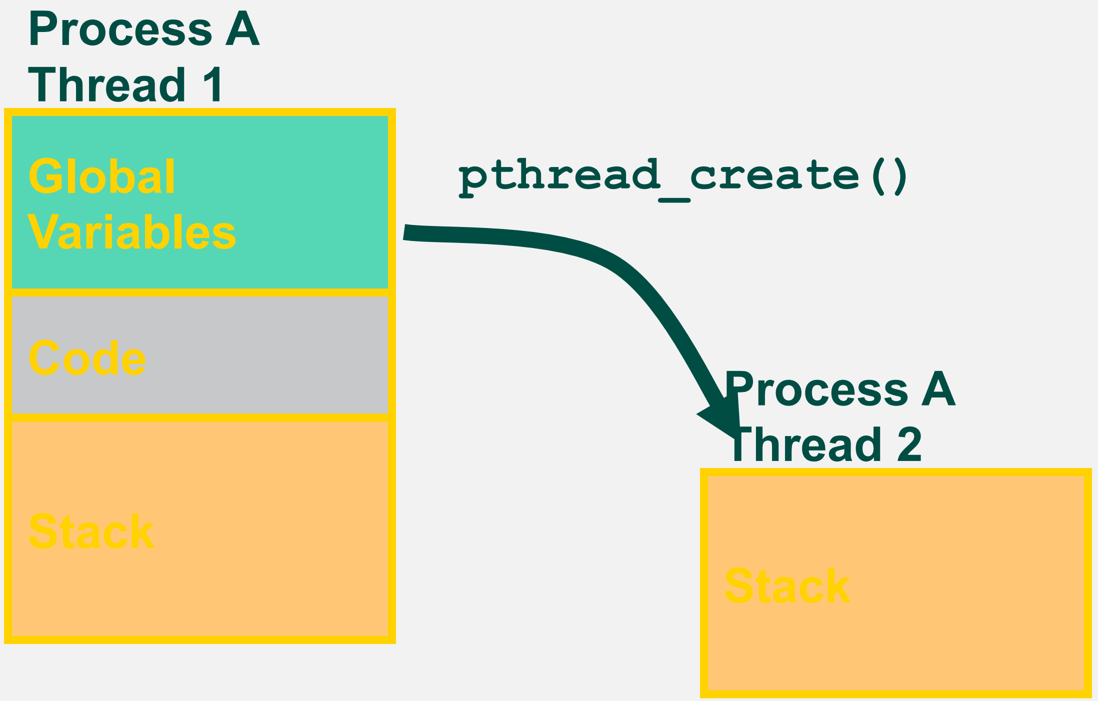
Creation of a new process using fork is expensive (time & memory).

A thread (sometimes called a lightweight process) does not require lots of memory or startup time.

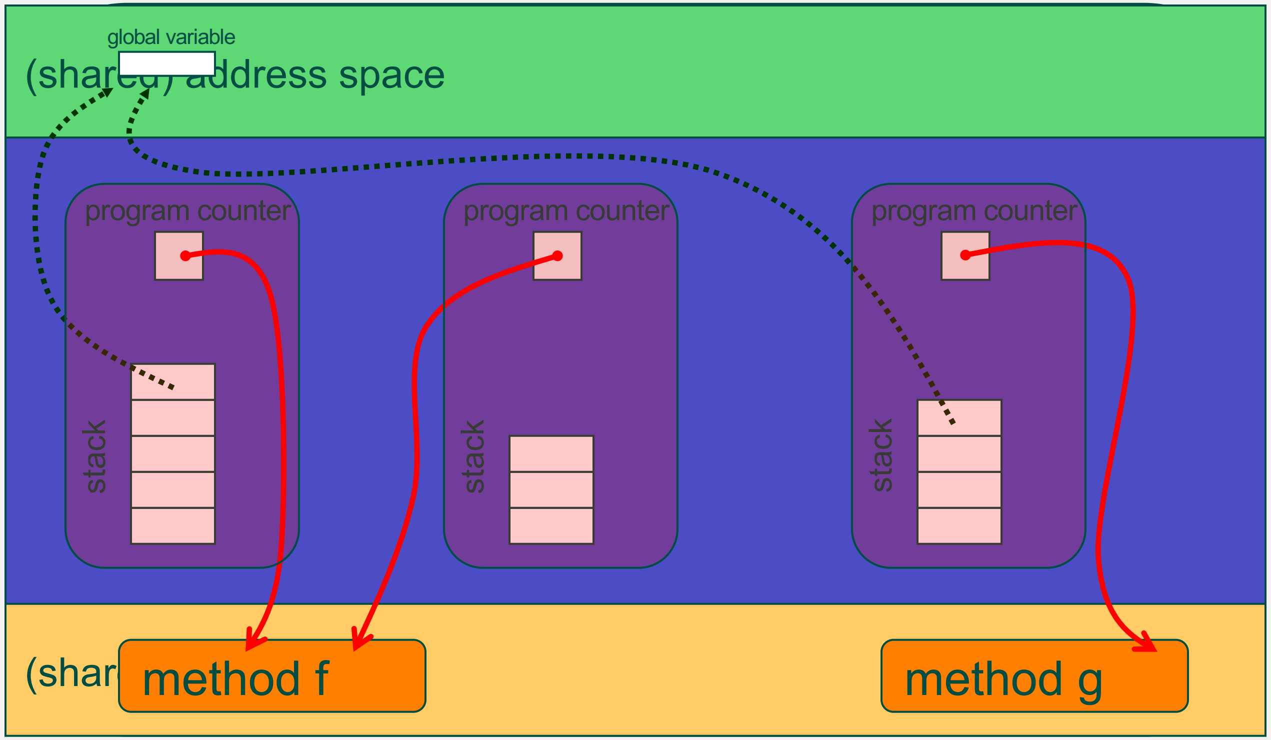
`fork()`


Process A
Global Variables
Code
Stack
fork()
Process B
Global Variables
Code
Stack

`pthread_create()`



Threads in a Process



process