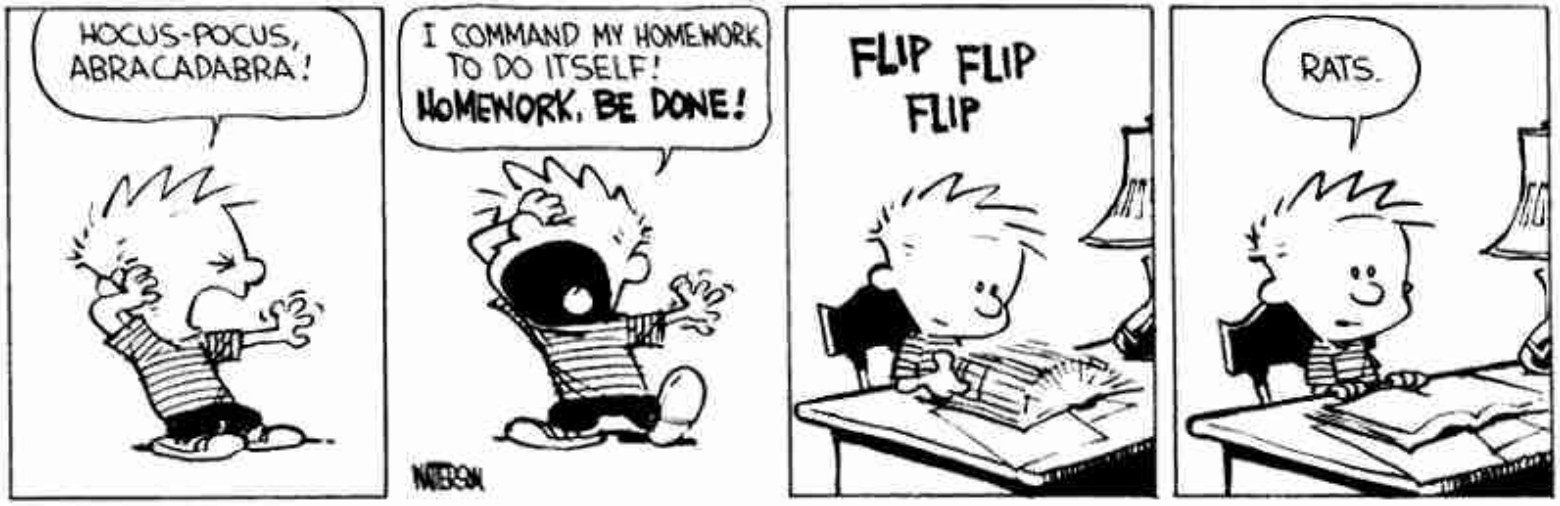
CME 213

SPRING 2019

Eric Darve

Homework instructions

- Homework focusses on C++, standard library and lambda functions.
- Submit a PDF on gradescope; your code is submitted on cardinal using a script.
- Deadline is: Wednesday, 11pm
- 10% penalty for late days; 2 late-day max
- Excuse has to be pre-arranged with instructor



Why Parallel Computing?

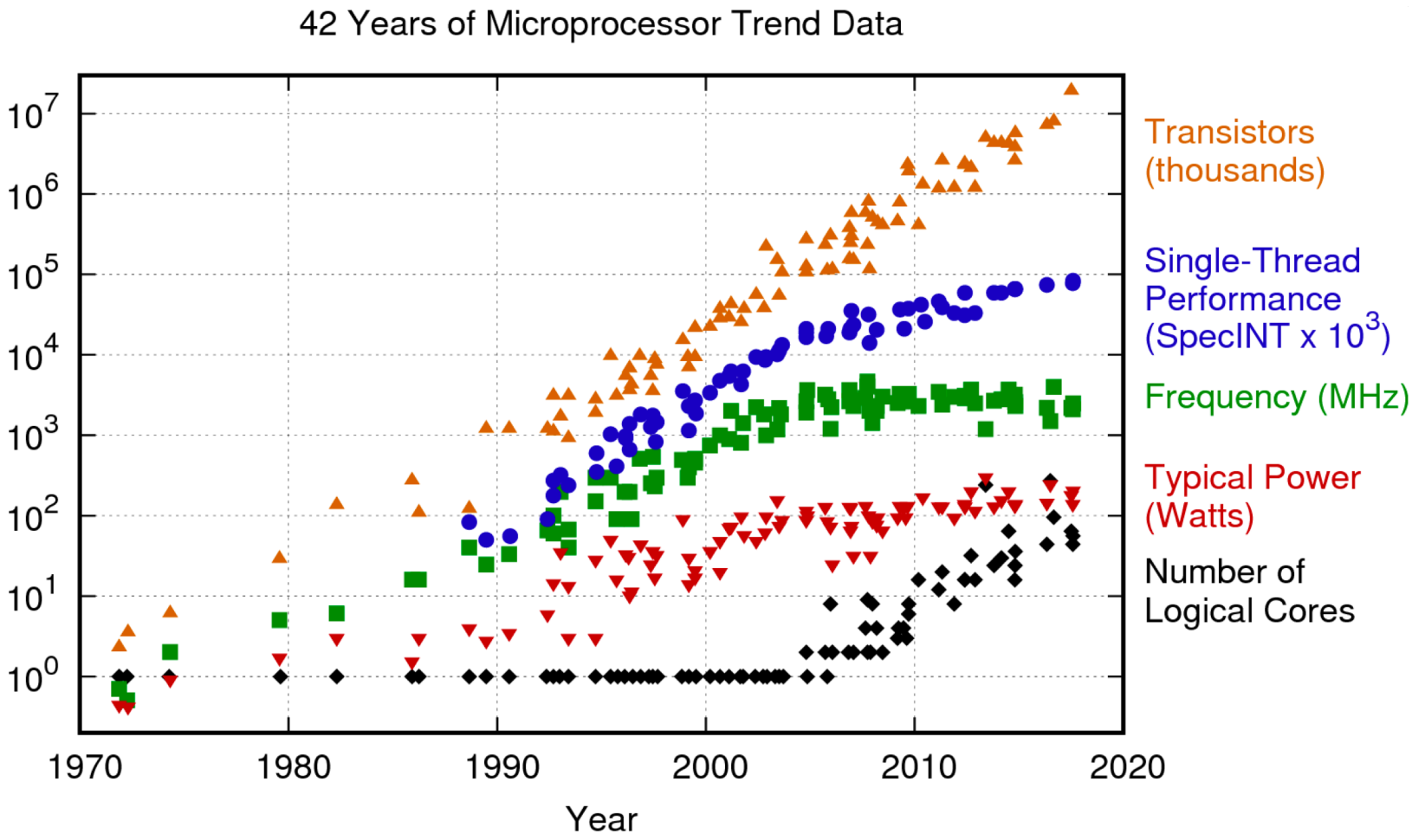
Why parallel computing?

- Parallel computing has existed for a long time but until recently it was a specialized area that concerned only a small fraction of engineers.
- Nowadays, parallel computing is a dominant player in scientific and large scale computing.
- What happened?

Why parallel computing?

- Gordon Moore 1965: the number of transistors on a chip shall double every 18–24 months.
- This has been valid for more than 40 years.
- This increase in the number of transistors has been accompanied by an increase in clock speed.

Intel microprocessor trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Performance increase

- Increase in transistor density is limited by:
  - Leakage current increases
  - Power consumption increases
  - Heat generated increases
- In addition, the memory access time has not been reduced at a rate comparable to the processing speed (processor clock period).
- New ways need to be found.
- The most promising approach is to have multiple cores on a single processor.

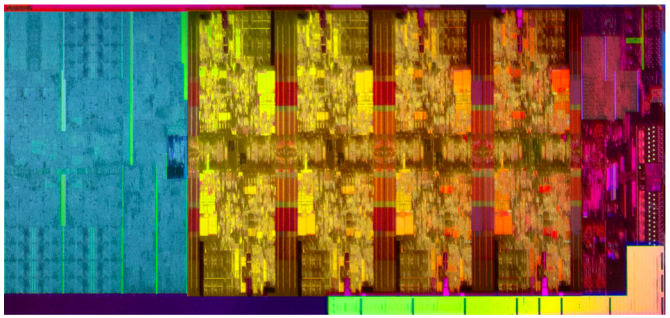
Sequential vs parallel

The Sequential Giant
Build bigger, meaner, faster processor

The Parallel Ants
Lots of smaller, slower cores

Parallel computing everywhere

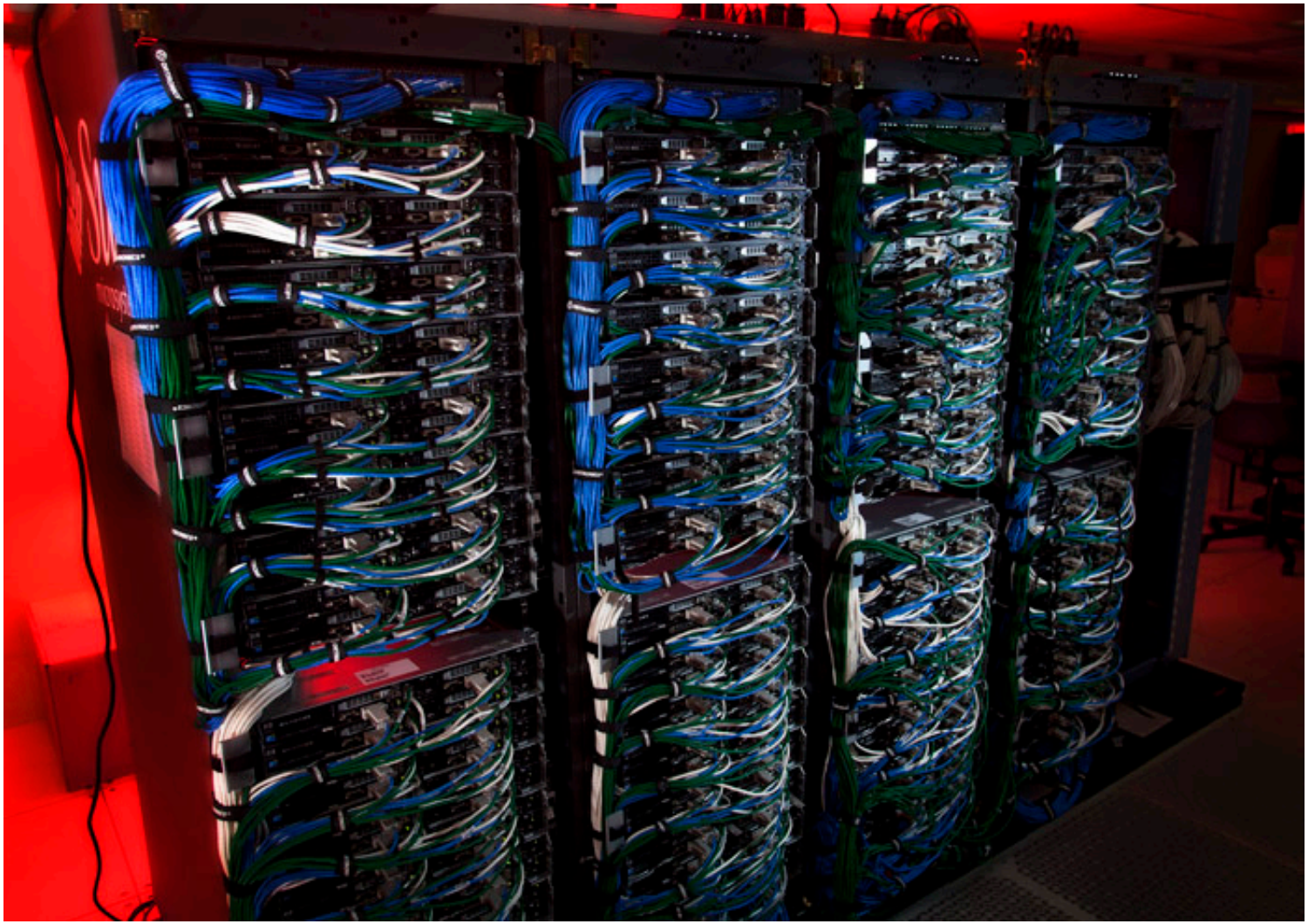
Multi and many core processors



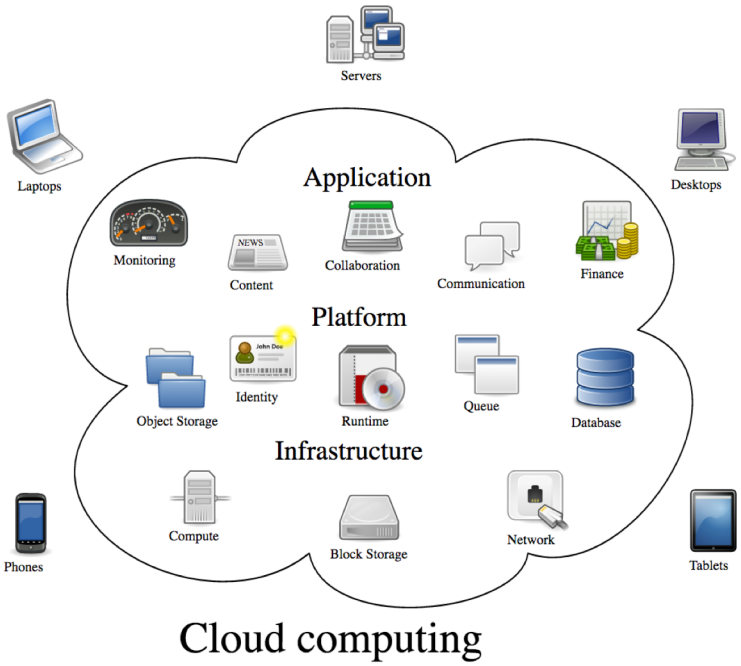
Core i9-9900K, 64-bit octa-core, Coffee Lake microarchitecture, 14nm++ process

NVIDIA Volta, V100: 15 teraflops of FP32, 30 teraflops of FP16, 7.5 teraflops of FP64, 120 teraflops for dedicated tensor operations; 84 SMs with 64 CUDA cores

Clusters



Cloud computing



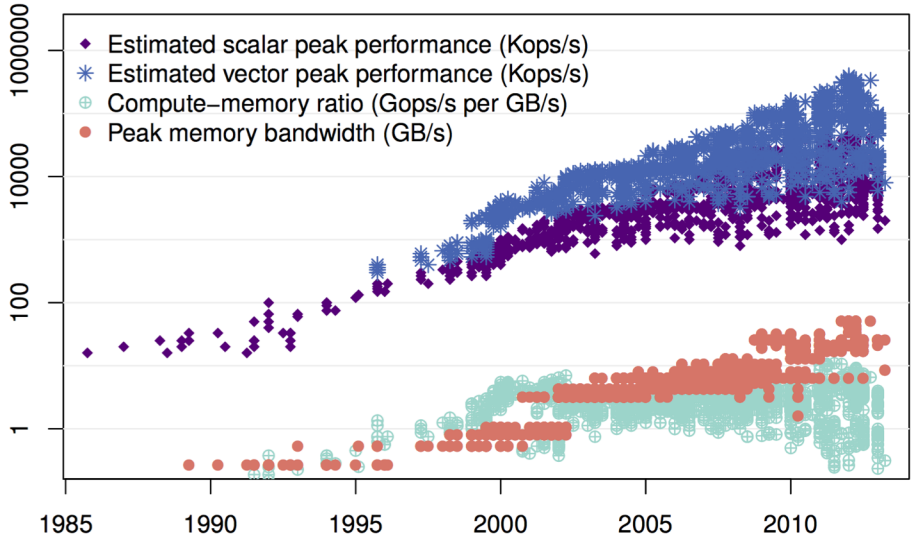
Cloud computing: internet-based computing that provides shared computer processing resources and data to computers and other devices on demand

Supercomputers

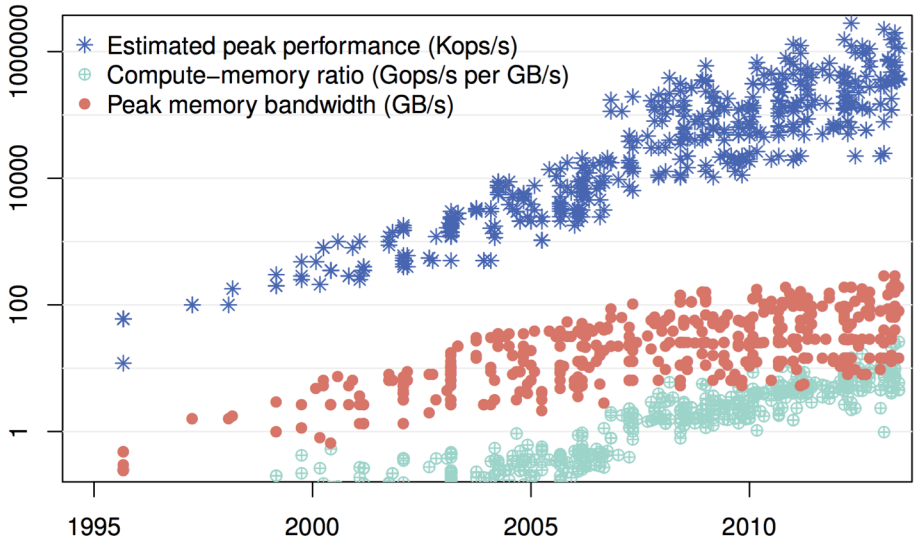
Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband

Site:	DOE/SC/Oak Ridge National Laboratory
System URL:	http://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/
Manufacturer:	IBM
Cores:	2,397,824
Memory:	2,801,664 GB
Processor:	IBM POWER9 22C 3.07GHz
Interconnect:	Dual-rail Mellanox EDR Infiniband
Performance	
Linpack Performance (Rmax)	143,500 TFlop/s
Theoretical Peak (Rpeak)	200,795 TFlop/s
Nmax	16,693,248
HPCG [TFlop/s]	2,925.75
Power Consumption	
Power:	9,783.00 kW (Submitted)
Power Measurement Level:	3
Measured Cores:	2,397,824

Performance through parallel processing



Historical data on 1403 Intel microprocessors



Historical data on 566 NVIDIA GPUs

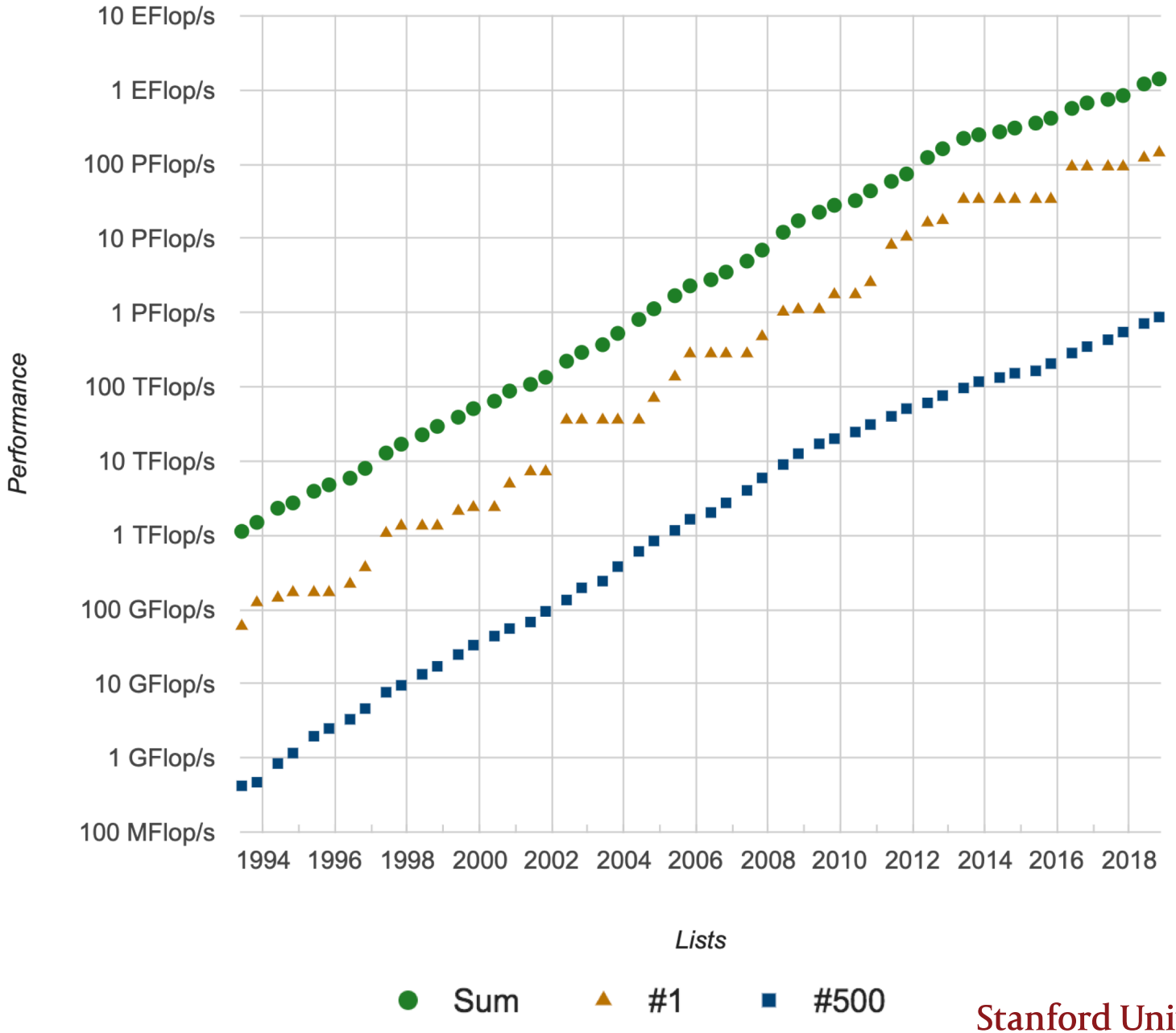
https://pure.tue.nl/ws/portalfiles/portal/3942529/771987.pdf

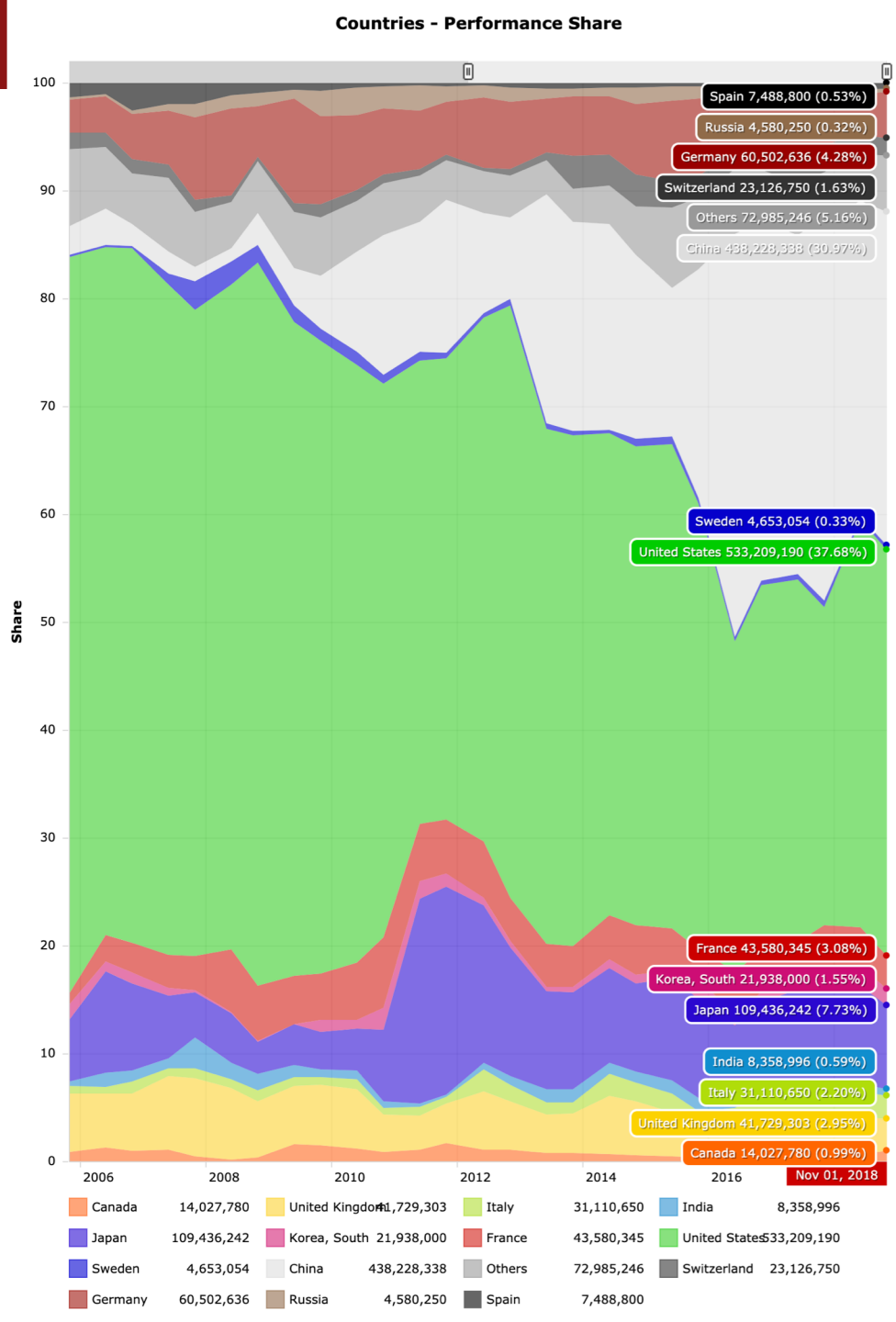
Statistics about the top 500 Supercomputers

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,397,824	143,500.0	200,794.9	9,783
2	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
6	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578
7	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
8	**SuperMUC-NG** - ThinkSystem SD530, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path , Lenovo Leibniz Rechenzentrum Germany	305,856	19,476.6	26,873.9	

ersity

Performance Development



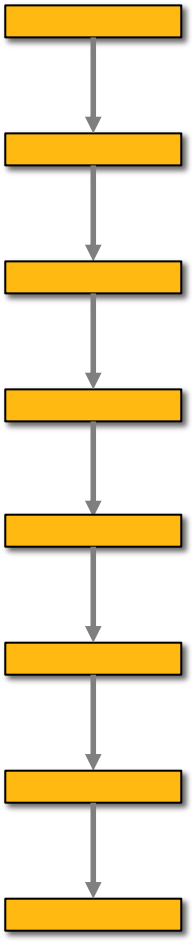


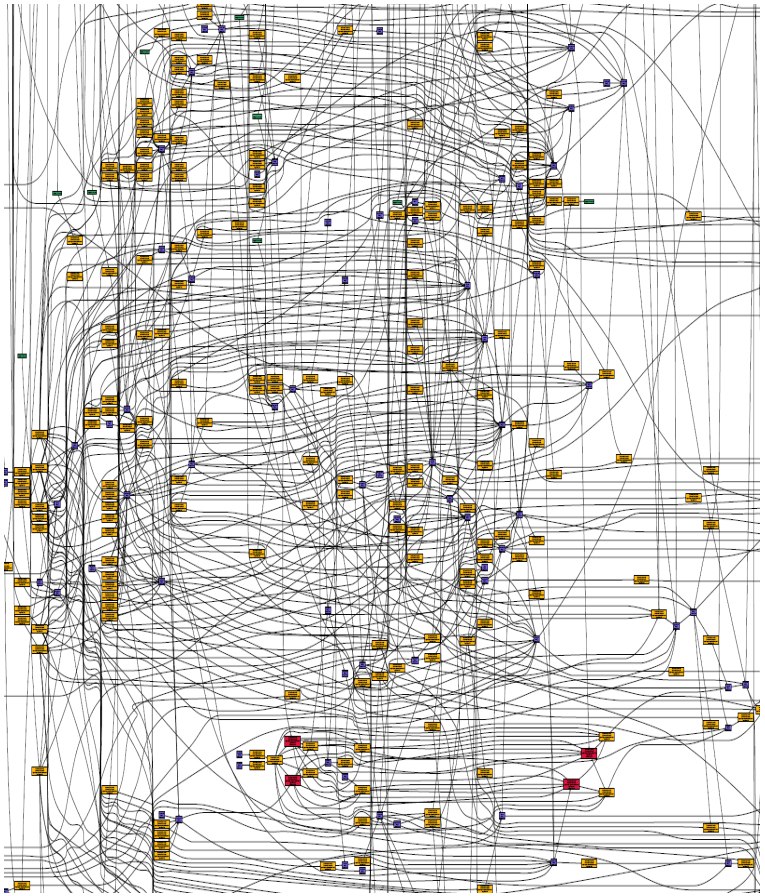
Stanford University

Example of Parallel Computation

Why we need to write parallel programs

- Most programs you have written so far are (probably) sequential.
- Unfortunately parallel programs often look very different…





Sequential program

Parallel program

- An efficient parallel implementation of a serial program may not be obtained by simply parallelizing each step.
- Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm.

Let's calculate the sum of n numbers

```
for (int i = 0; i < n; i++)
{
    x = ComputeNextValue();
    sum += x;
}
```

Our first parallel program

- Assume we have p cores that can compute and exchange data.
- Then we could accelerate the previous calculation by splitting the work among all these cores.

```
int r; /*thread number*/
int b; /*number of entries to process*/
int my_first_i = r*b;
int my_last_i = (r+1)*b;
for (int my_i = my_first_i; my_i < my_last_i; my_i++)
{
    my_x = ComputeNextValue();
    my_sum += my_x;
}
```

But it's not that simple

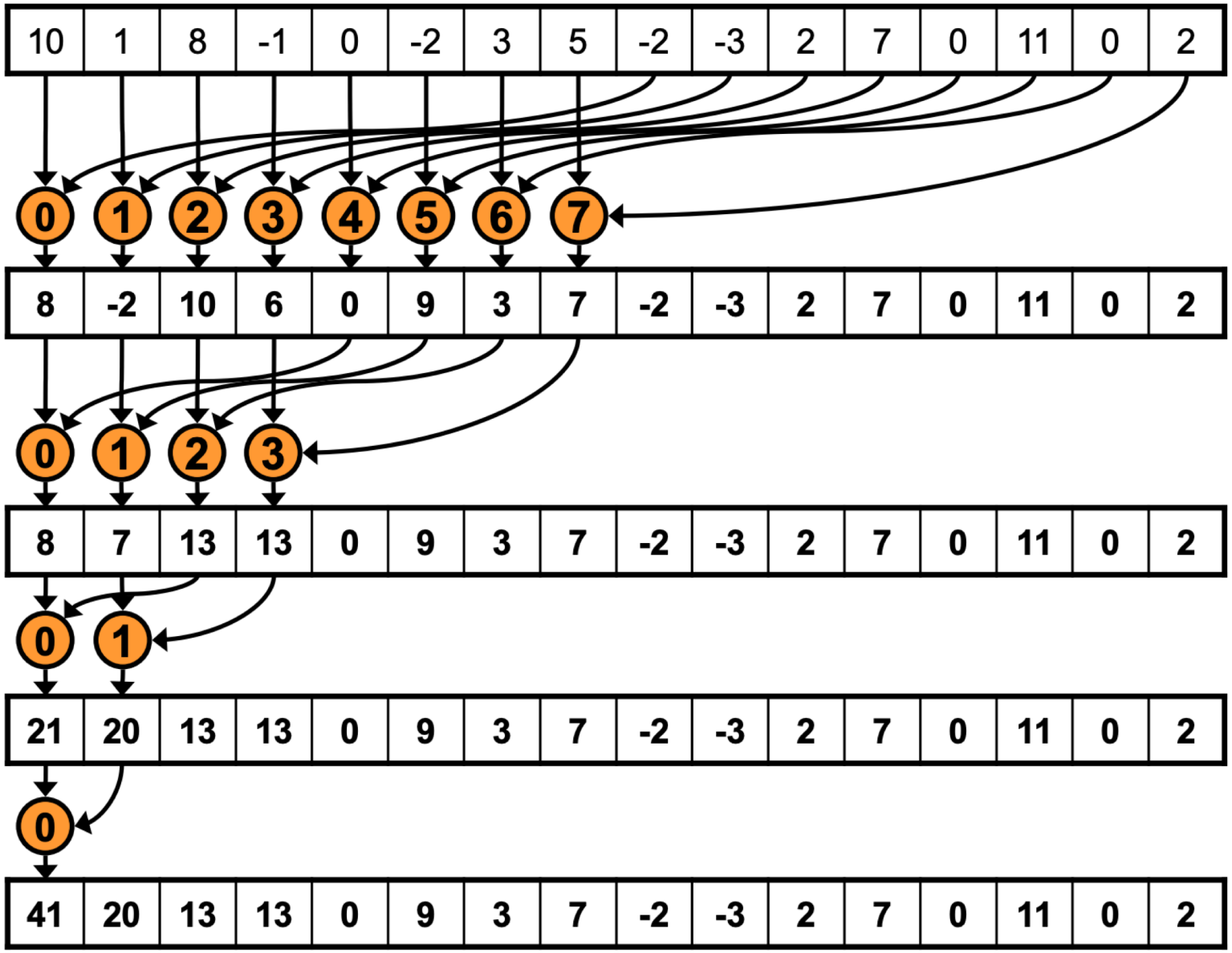
- Each core has computed a partial sum.
- All these partial sums need to summed up together.
- The simplest approach is to have one "master" thread do all the work.

```
if (r == 0) /*master thread*/
{
    int sum = my_sum;
    for (int ro = 1; ro < p; ++ro)
    {
        int sum_ro;
        ReceiveFrom(&sum_ro, ro);
        sum += sum_ro;
    }
}
else /*worker thread*/
{
    SendTo(&my_sum, 0);
}
```

That may not be enough

- If we have many cores, this final sum may in fact take a lot of time.

- How would you design a better implementation?


10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
0 1 2 3 4 5 6 7
8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
0 1 2 3
8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
0 1
21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
0
41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

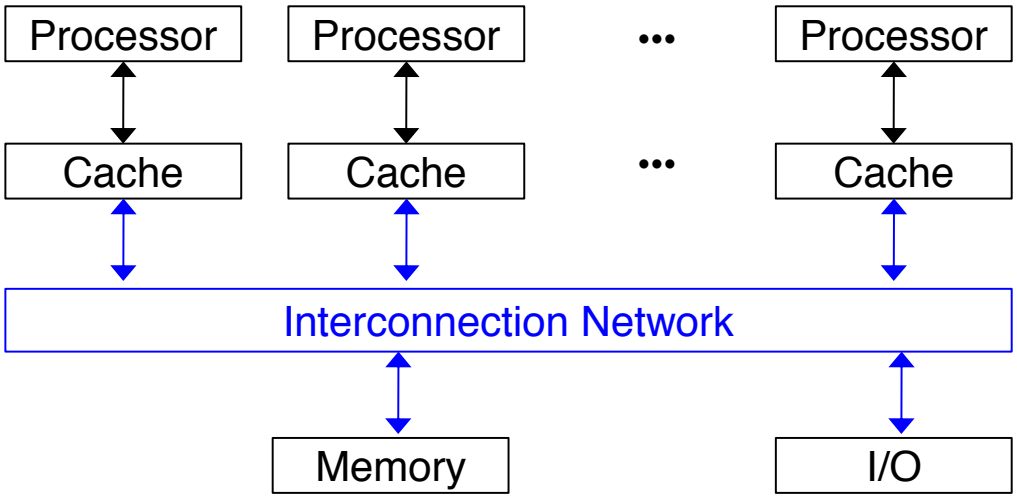
Automatic parallelization

- This simple example illustrates the fact that it is difficult for a compiler to parallelize a program.
- Instead the programmer must often re-write his code having in mind that multiple cores will be computing in parallel.
- The purpose of this class is to teach you the most common parallel languages used in science and engineering.

Shared Memory Processor

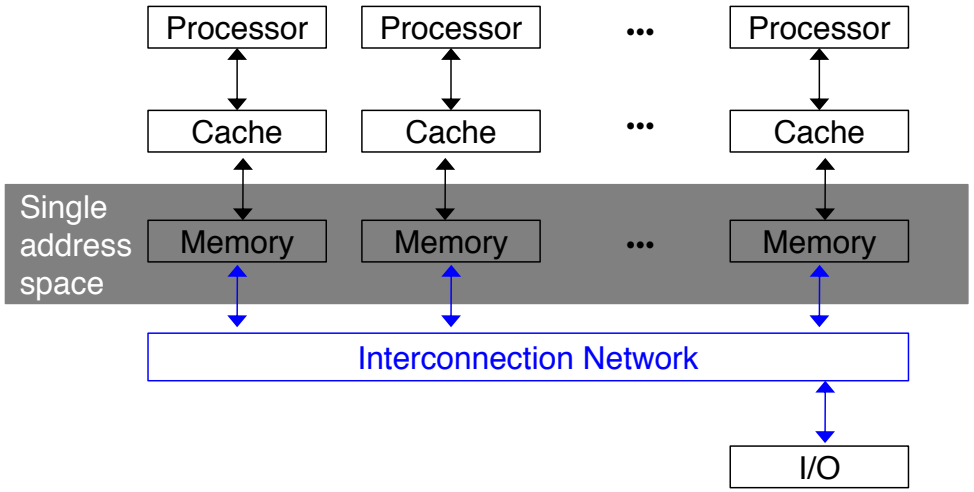
Schematic of a multicore processor

- Model for shared memory machines
- Comprised of:
  - A number of processors or cores
  - A shared physical memory (global memory)
  - An interconnection network to connect the processors with the memory.

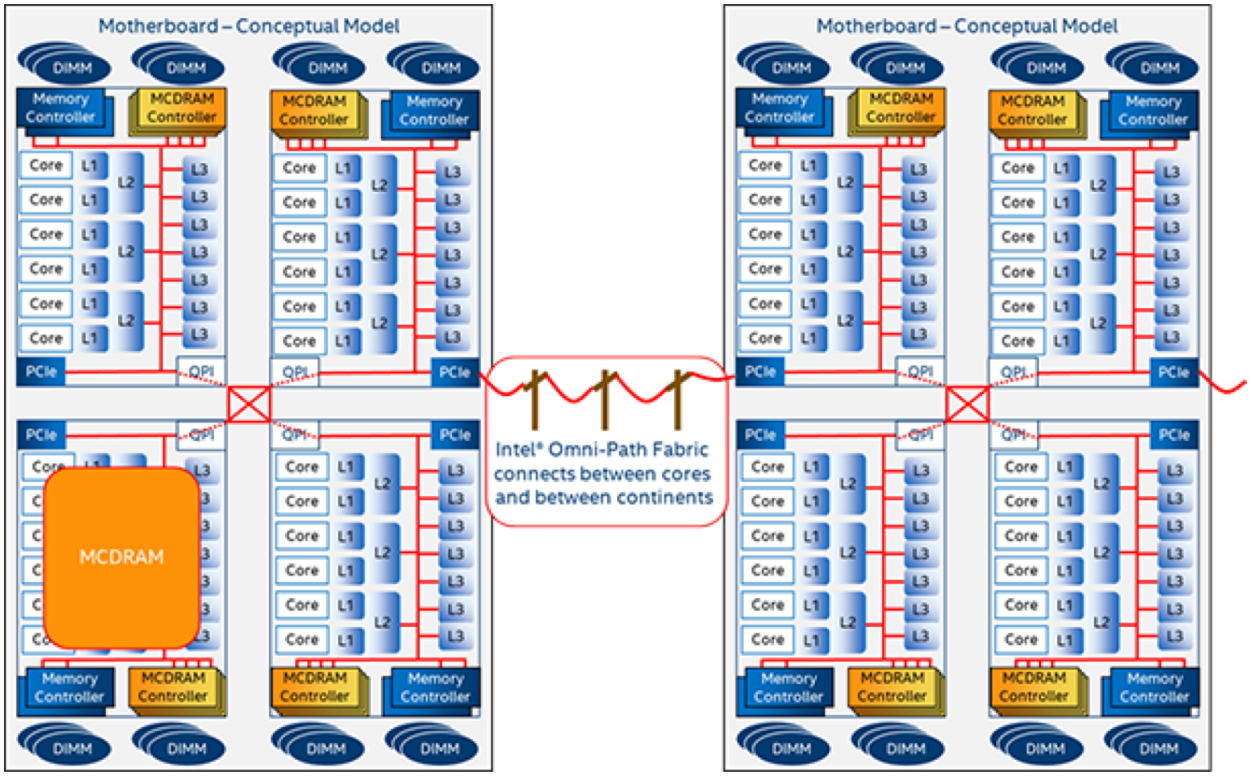


Shared memory NUMA

- In many cases, the program views the memory as a single addressable space, but in reality the memory is physically distributed.
- NUMA: non-uniform memory access.
- Faster access to memory, but special hardware required to move data between memory banks, e.g., Intel Omni-Path Fabric.



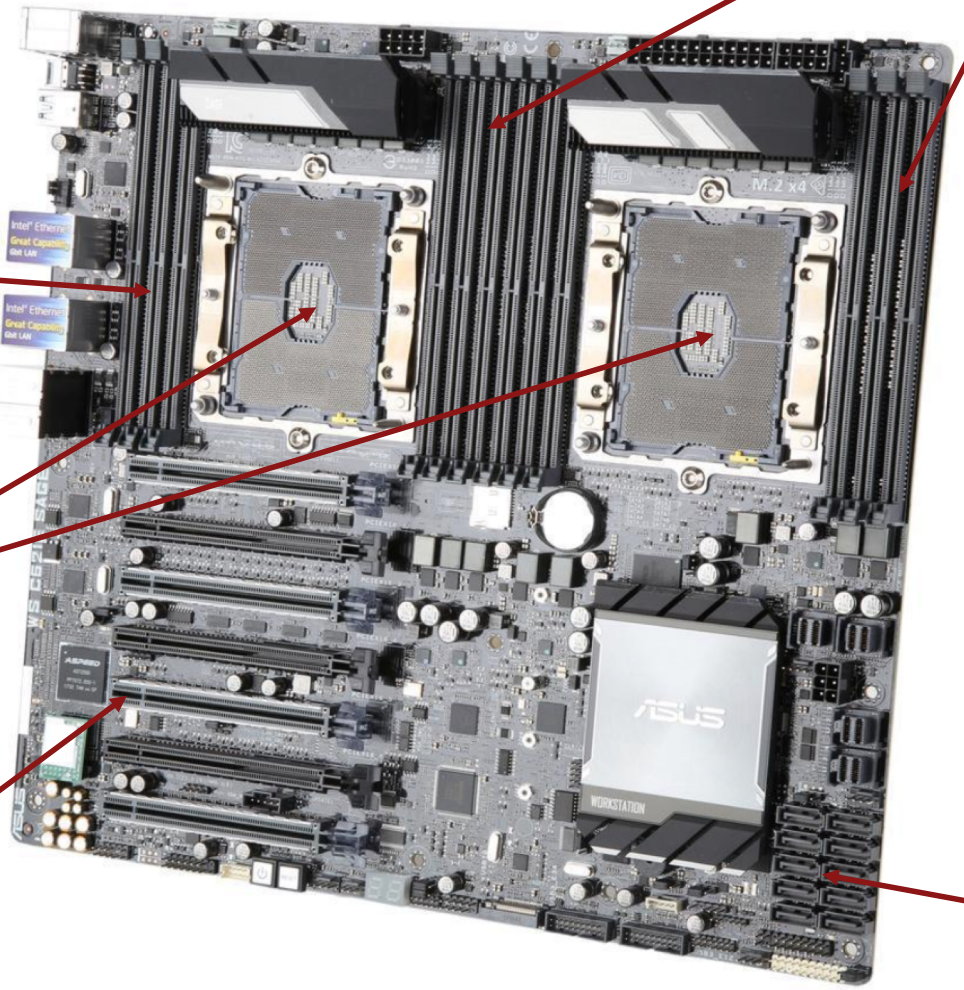
Case study: modern Intel processors with NUMA



Xeon Phi series
Knights Ferry
Knights Corner
Knights Landing
~~Knights Hill~~
Knights Mill

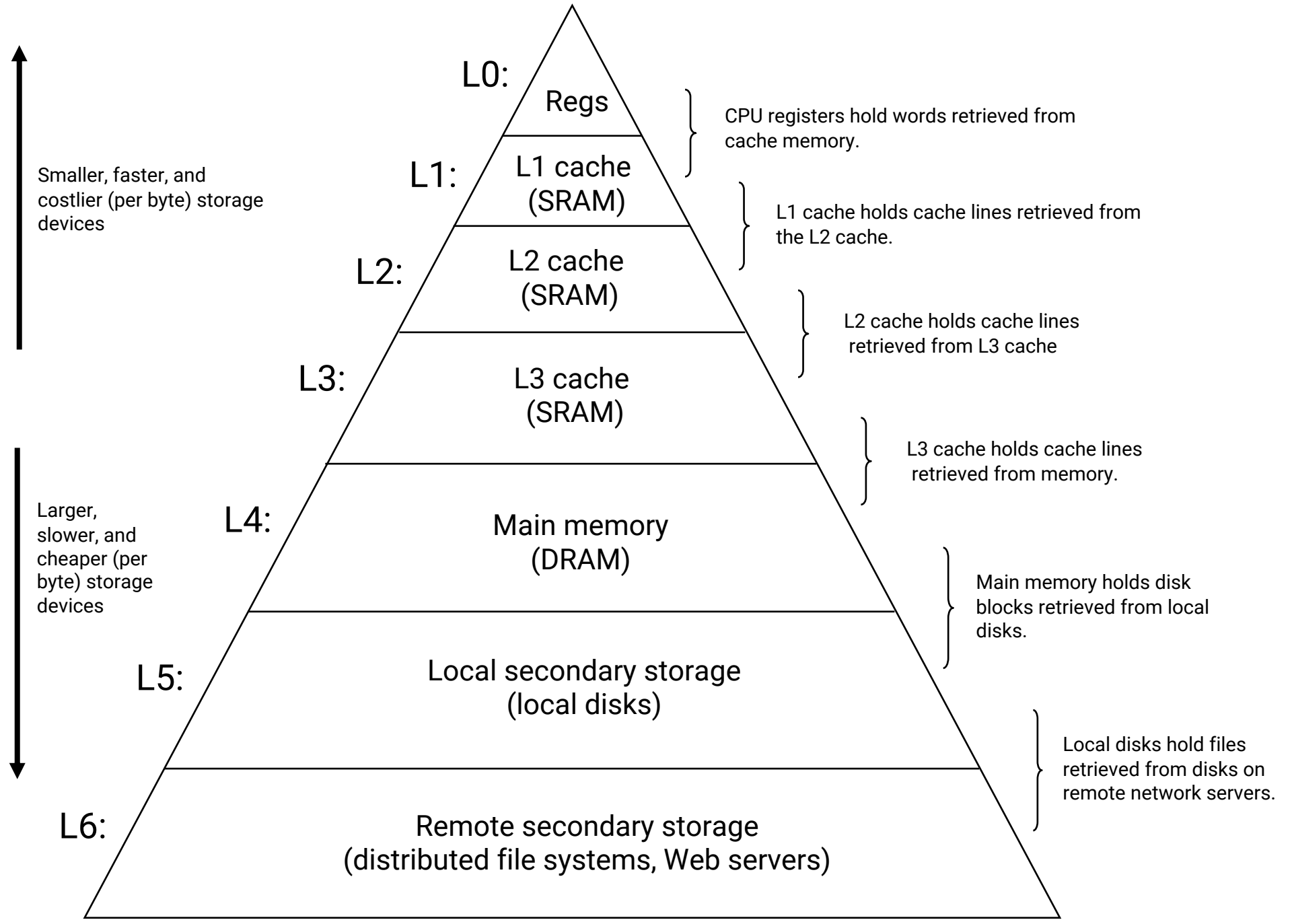
- MCDRAM (Multi-Channel DRAM): proprietary, high-bandwidth memory that sits atop Xeon Phi processors (Knights Landing). HBM (High-Bandwidth Memory): Xeon Phi Knights Hill.
- Omni-Path connects a core to memory sitting next to other cores.

Motherboard with 2 CPU sockets



Memory is hierarchical

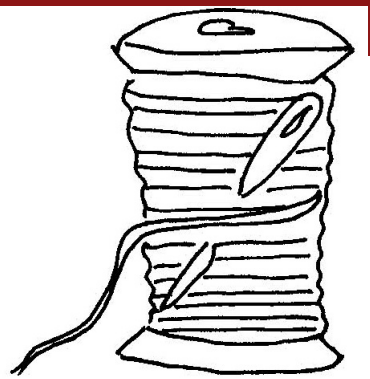
- In this class we will only briefly discuss performance for multicore processors.
- Things to keep in mind for performance:
  - memory is key to developing high-performance multicore applications
  - memory is hierarchical and complex.


L0:
Regs
CPU registers hold words retrieved from cache memory.
L1:
L1 cache (SRAM)
L1 cache holds cache lines retrieved from the L2 cache.
L2:
L2 cache (SRAM)
L2 cache holds cache lines retrieved from L3 cache
L3:
L3 cache (SRAM)
L3 cache holds cache lines retrieved from memory.
L4:
Main memory (DRAM)
Main memory holds disk blocks retrieved from local disks.
L5:
Local secondary storage (local disks)
Local disks hold files retrieved from disks on remote network servers.
L6:
Remote secondary storage (distributed file systems, Web servers)
Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices

Size, latency, Bandwidth of memory subsystem

Memory	Size	Latency	Bandwidth
L1 cache	32 KB	1 nanosecond	1 TB/second
L2 cache	256 KB	4 nanoseconds	1 TB/second Sometimes shared by two cores
L3 cache	8 MB or more	10x slower than L2	>400 GB/second
MCDRAM		2x slower than L3	400 GB/second
Main memory on DDR DIMMs	4 GB-1 TB	Similar to MCDRAM	100 GB/second
Main memory on Intel Omni-Path Fabric	Limited only by cost	Depends on distance	Depends on distance and hardware
I/O devices on memory bus	6 TB	100x-1000x slower than memory	25 GB/second
I/O devices on PCIe bus	Limited only by cost	From less than milliseconds to minutes	GB-TB/hour Depends on distance and hardware

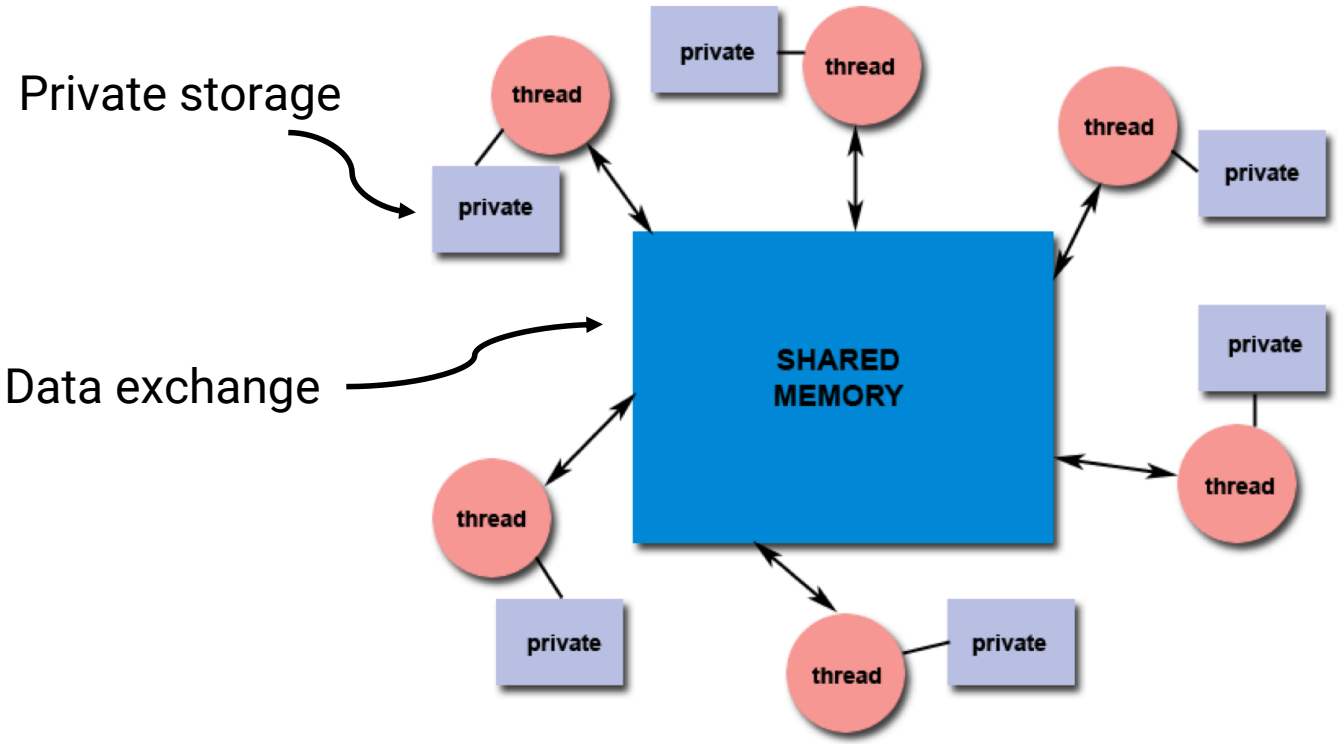
Processes and threads



- Definition:
- Process:
  - Program in execution.
  - Comprises: the executable program along with all information that is necessary for the execution of the program.
- Thread: an extension of the process model. Can be viewed as a "lightweight" process.
- In this model, each process may consist of multiple independent control flows that are called threads.
- A thread may be described as a "procedure" that runs independently from the main program.
- Imagine a program that contains a number of procedures. Then imagine these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. This describes a "multi-threaded" program.

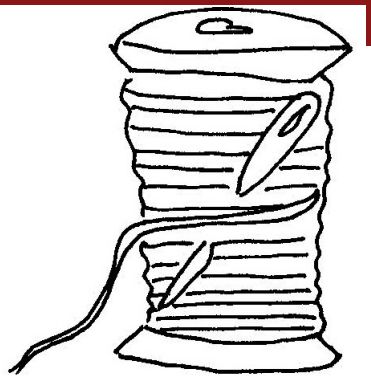
Shared address space

- All the threads of one process share the address space of the process, i.e., they have a common address space.
- When a thread stores a value in the shared address space, another thread of the same process can access this value.



Threads

Threads are everywhere



- C++ threads (11):
  ```
  std::thread
  ```
- C threads: Pthreads
- Java threads:
  ```
  Thread thread = new Thread();
  ```
- Python threads:
  ```
  t = threading.Thread(target=worker)
  ```
- Cilk:
  ```
  x = spawn fib (n-1);
  ```
- Julia:
- ```
  r = remotecall(rand, 2, 2, 2)
  ```
- OpenMP
  ```

C++ threads exercise

- Open the file `cpp_thread.cpp`
- Type `make` to compile

Pthreads

- This is the most “low-level” approach for programming in parallel.
- Pthreads: POSIX threads. This is a standard to implement threads on UNIX systems. It is based on the C programming language.
- Pthreads will serve as an introduction to the most important concepts in multicore programming.
- The other approach we will cover is OpenMP.
- OpenMP is the standard for multicore programming in scientific programs.
- Pthreads will help you understand OpenMP.

The basics

- Include the header file:

```
include <pthread.h>
```

- Compile using:

```
gcc -o hello_pthread hello_pthread.c -lpthread
```

- See `hello_pthread.c`

```
for (t = 0; t < n_thread; t++)
{
    printf("In main: creating thread %ld\n", t);
    pthread_create(&thread[t], NULL, PrintHello, (void *)t);
}

for (t = 0; t < n_thread; t++)
{
    pthread_join(thread[t], (void **)(&thread_result) /*NULL is common*/);
    printf("Thread #%ld just finished; its value is %ld\n", t,
           thread_result);
    assert(ComplexCalculation(t) == thread_result); /* Testing output */
}
```

Thread creation

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

- `thread` thread identifier
- `routine` function that will be executed by the thread
- `arg` pointer to the argument value with which the thread function `routine()` will be executed
- `attr` use `NULL` for the time being

Thread termination

A thread terminates when:

1. Thread reaches the end of its thread function, i.e., returns.
2. Thread calls

```
void pthread_exit(void *valuep)
```

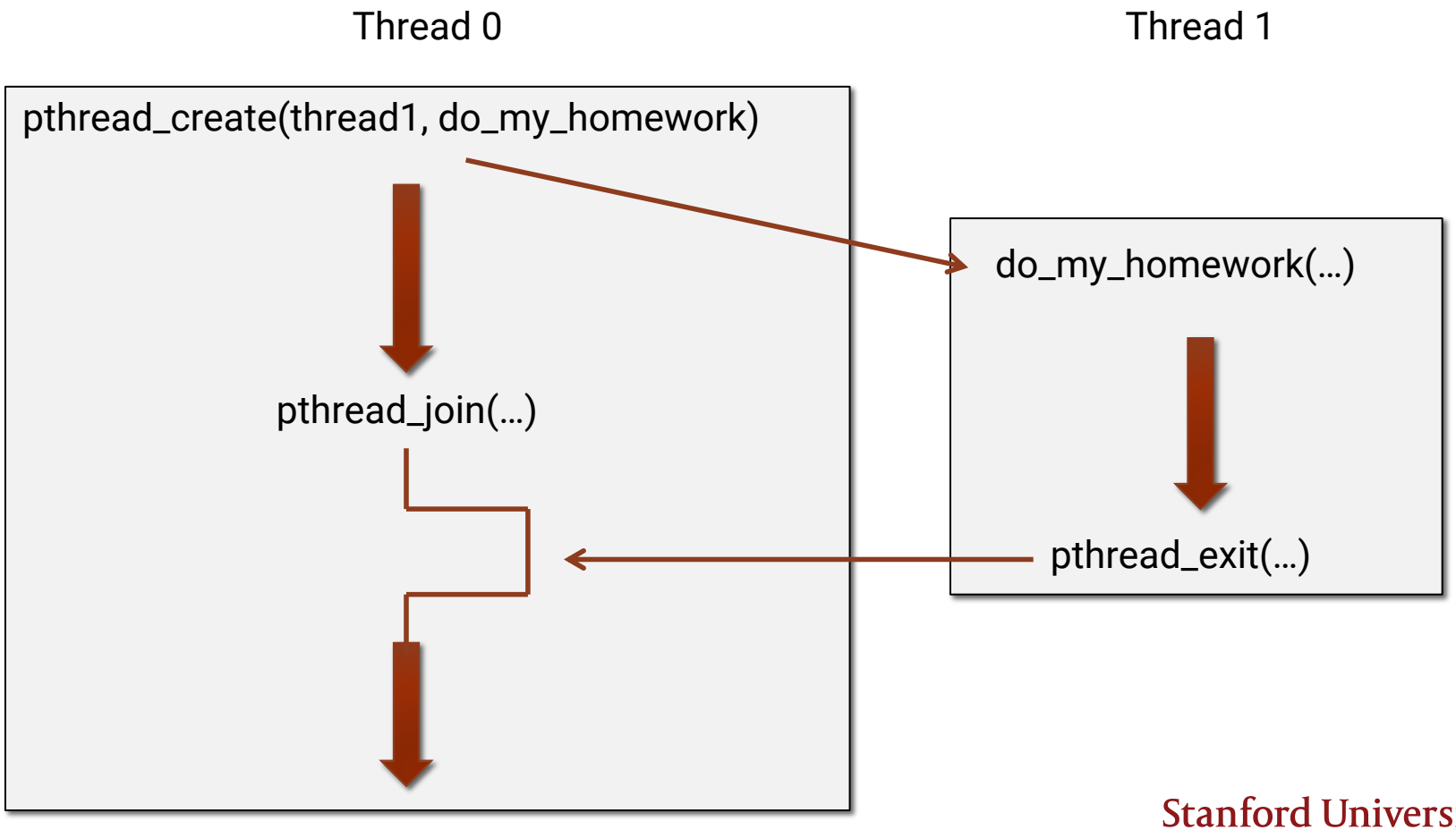
Note:

- Upon termination, a thread releases its runtime stack.
- Therefore, the pointer should point to: 1) a global variable, or 2) a dynamically allocated variable.

```c
void *PrintHello(void *threadid)
{
    long tid = (long)threadid;
    long result;
    result = ComplexCalculation(tid); /* Simulates some useful calculation */
    printf("Hello World! It's me, thread #%ld. My value is %ld!\n", tid,
           result);
    pthread_exit((void *)result /*NULL is common*/);
}
```

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

- Calling thread waits for `thread` to terminate.
- `pthread_join` is used to synchronize threads.
- `valuep` memory address where the return value of thread will be stored.



See

`hello_pthread_bug_1.c`

`hello_pthread_bug_2.c`