

# **CS4403 - CS9535: An Overview of Parallel Computing**

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

January 10, 2017

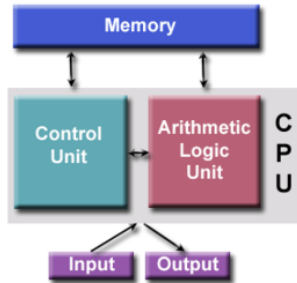
# Plan

- 1 Hardware
- 2 Types of Parallelism
- 3 Concurrency Platforms: Three Examples
  - Julia
  - Cilk
  - CUDA
  - MPI

# Plan

- 1 Hardware
- 2 Types of Parallelism
- 3 Concurrency Platforms: Three Examples
  - Julia
  - Cilk
  - CUDA
  - MPI

## von Neumann Architecture



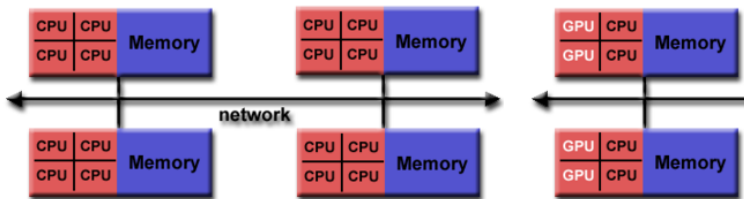
- In 1945, the Hungarian mathematician John von Neumann proposed the above organization for hardware computers.
- The **Control Unit** fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.
- The **Arithmetic Unit** performs basic arithmetic operation, while **Input/Output** is the interface to the human operator.



# The Pentium Family

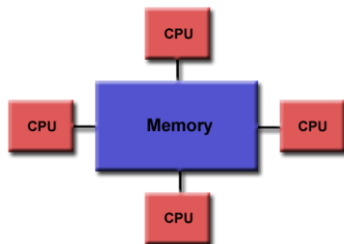


## Parallel computer hardware



- Most computers today (including tablets, smartphones, etc.) are equipped with several processing units (control+arithmetic units).
- Various characteristics determine the types of computations: **shared memory** vs **distributed memory**, **single-core processors** vs **multicore processors**, **data-centric parallelism** vs **task-centric parallelism**.
- Historically, shared memory machines have been classified as UMA and NUMA, based upon memory access times.

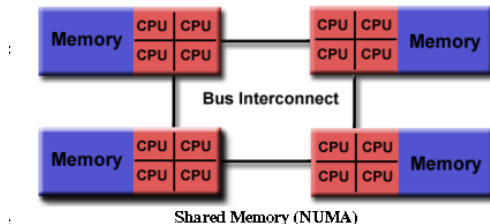
## Uniform memory access (UMA)



Shared Memory (UMA)

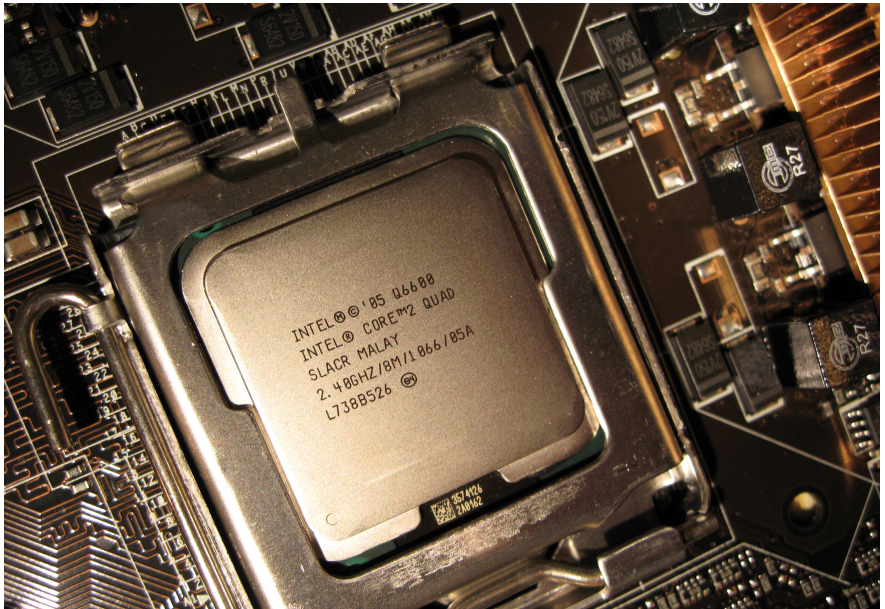
- Identical processors, equal access and access times to memory.
- In the presence of cache memories, [cache coherency](#) is accomplished at the hardware level: if one processor updates a location in shared memory, then all the other processors know about the update.
- UMA architectures were first represented by [Symmetric Multiprocessor \(SMP\) machines](#).
- Multicore processors follow the same architecture and, in addition, integrate the cores onto a single circuit die.

## Non-uniform memory access (NUMA)

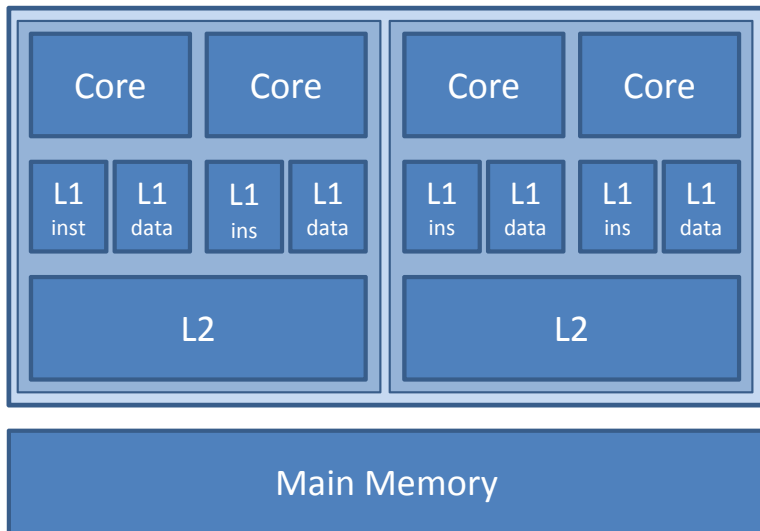


- Often made by physically linking two or more SMPs (or multicore processors).
- Global address space provides a user-friendly programming perspective to memory, that is, it feels like there is a single large memory where all data reside.
- However, not all processors have equal access time to all memories, since memory access across link is slower.
- In fact, memory contention (that is, traffic jam) often limits the ability to scale of these architectures.

## Multicore processors

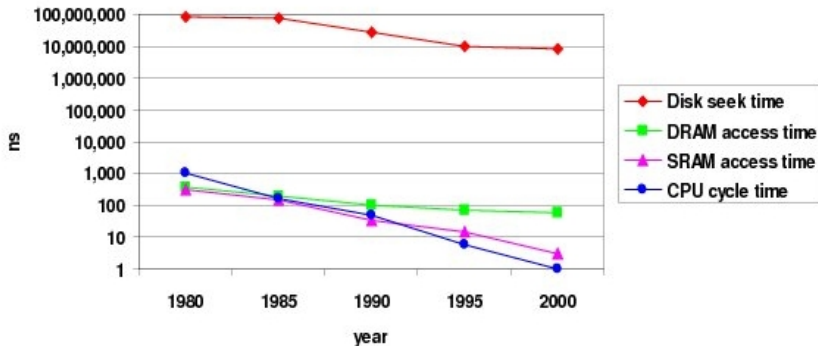


## Multicore processors



# The CPU-Memory Gap

**The increasing gap between DRAM, disk, and CPU speeds.**



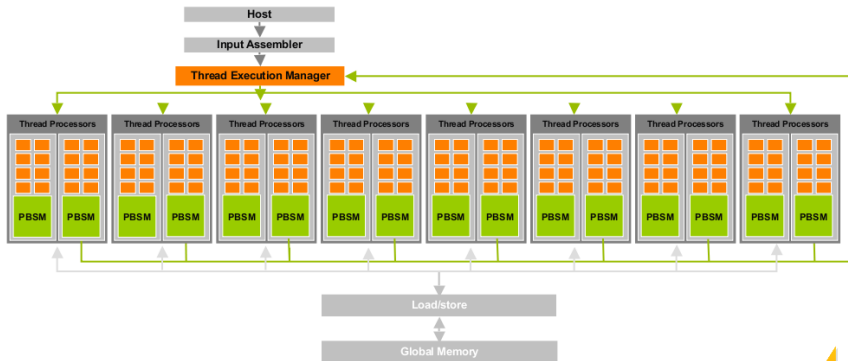
Once upon a time, every thing was slow in a computer . . .

## Graphics processing units (GPUs)



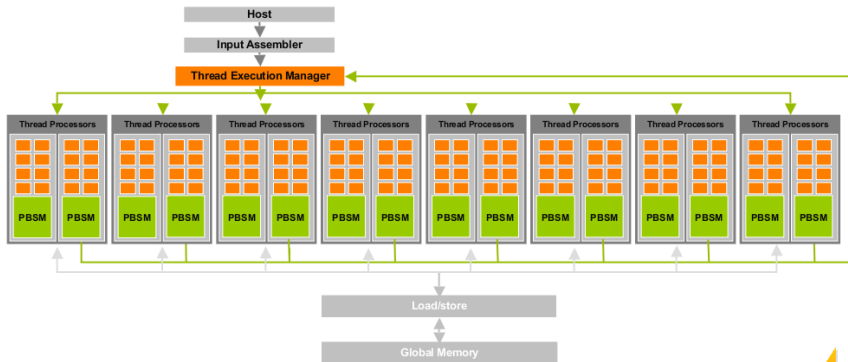


# Graphics processing units (GPUs)



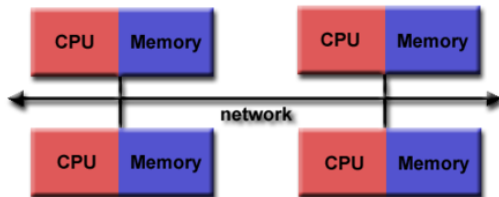
- A GPU consists of several **streaming multiprocessors (SMs)** with a large shared memory. In addition, each SM has a local (and private) and small memory. Thus, GPUs cannot be classified as UMA or NUMA.

# Graphics processing units (GPUs)



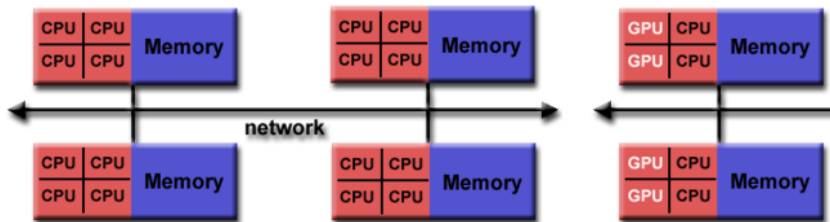
- In a GPU, the small local memories have much smaller access time than the large shared memory.
- Thus, as much as possible, cores access data in the local memories while the shared memory should essentially be used for data exchange between SMs.

## Distributed Memory



- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory and operate independently.
- Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Data exchange between processors is managed by the programmer, not by the hardware.

## Hybrid Distributed-Shared Memory



- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- Current trends seem to indicate that this type of memory architecture will continue to prevail.
- While this model allows for applications to scale, it increases the complexity of writing computer programs.

# Plan

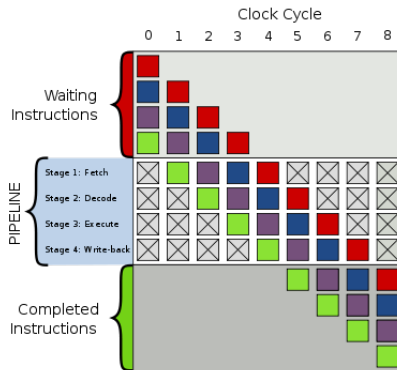
- 1 Hardware
- 2 Types of Parallelism
- 3 Concurrency Platforms: Three Examples
  - Julia
  - Cilk
  - CUDA
  - MPI

## Pipelining



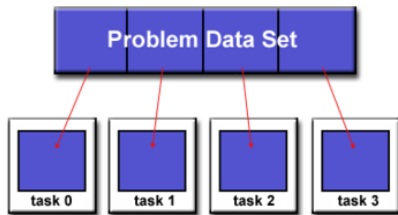
- Pipelining is a common way to organize work with the objective of optimizing throughput.
- It turns out that this is also a way to execute concurrently several **tasks** (that is, work units) processable by the same pipeline.

# Instruction pipeline



- Above is a generic pipeline with four stages: Fetch, Decode, Execute, Write-back.
- The top gray box is the list of instructions waiting to be executed; the bottom gray box is the list of instructions that have been completed; and the middle white box is the pipeline.

## Data parallelism



- The data set is typically organized into a common structure, such as an array.
- A set of tasks work collectively on that structure, however, each task works on a different region.
- Tasks perform the same operation on their region of work, for example, "multiply every array element by some value".



## Data parallelism (2/2)

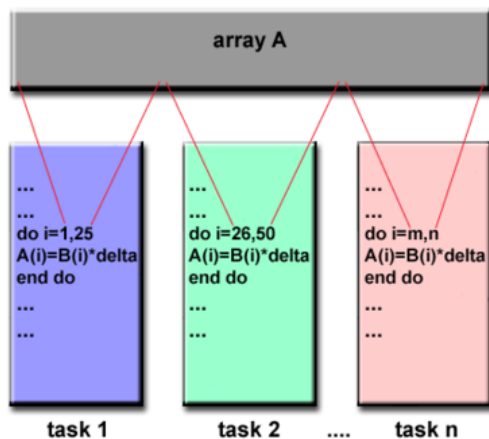


Illustration of a data-centric parallel program.

## Task parallelism (1/4)

```
program:  
  ...  
  if CPU="a" then  
    do task "A"  
  else if CPU="b" then  
    do task "B"  
  end if  
  ...  
end program
```

- Task parallelism is achieved when each processor executes a different thread (or process) on the same or different data.
- The threads may execute the same or different code.
- 
-

## Task parallelism (2/4)

Code executed by CPU "a":

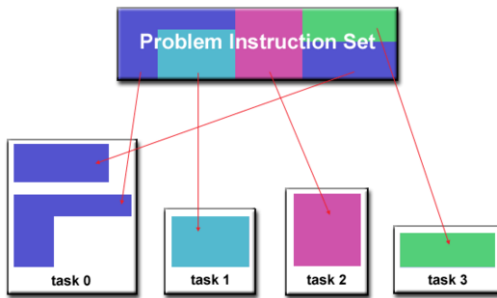
```
program:  
...  
do task "A"  
...  
end program
```

Code executed by CPU "b":

```
program:  
...  
do task "B"  
...  
end program
```

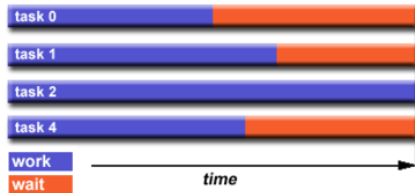
- In the general case, different execution threads communicate with one another as they work.
- Communication usually takes place by passing data from one thread to the next as part of a work-flow.

## Task parallelism (3/4)



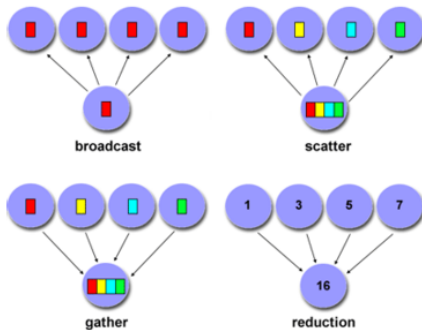
- Task parallelism can be regarded as a more general scheme than data parallelism.
- It applies to situations where the work can be decomposed evenly or where the decomposition of the work is not predictable.

## Task parallelism (4/4)



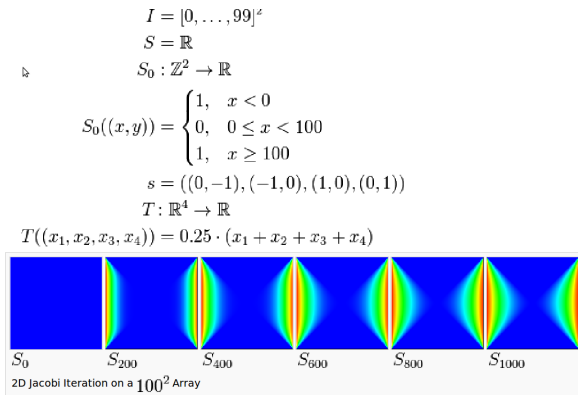
- In some situations, one may feel that work can be decomposed evenly. However, as time progresses, some tasks may finish before others
- Then, some processors may become idle and should be used, if other tasks can be launched. This mapping of tasks onto hardware resources is called [scheduling](#).
- In data-centric parallel applications, scheduling can be done off-line (that is, by the programmer) or by the hardware (like GPUs).
- For task-centric parallel applications, it is desirable that scheduling is done on-line (that is, dynamically) so as to cover cases where tasks consume unpredictable amounts of resources.

## Patterns in task or data distribution



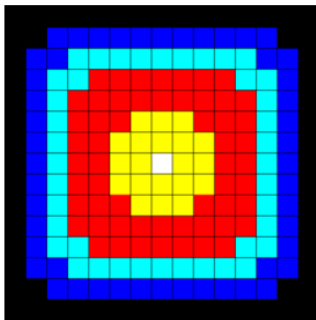
- Exchanging data among processors in a parallel fashion provides fundamental examples of concurrent programs.
- Above, a master processor **broadcasts** or **scatters** data or tasks to slave processors.
- The same master processor **gathers** or **reduces** data from slave processors.

## Stencil computations



- In scientific computing, stencil computations are very common.
- Typically, a procedure updates array elements according to some fixed pattern, called [stencil](#).
- In the above, a 2D array of  $100 \times 100$  elements is updated by the stencil  $T$ .

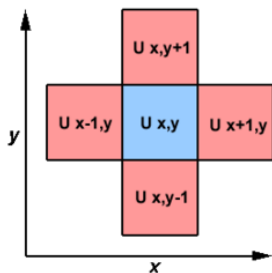
## Stencil computations (2/3)



- The above picture illustrates dissipation of heat into a 2D grid.
- A differential equation rules this phenomenon.
- Once this discretized, through the finite element method, this leads a stencil computation.



## Stencil computations (3/3)



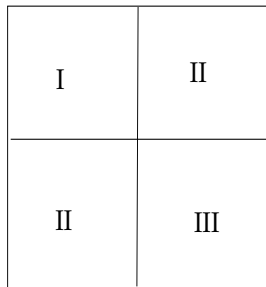
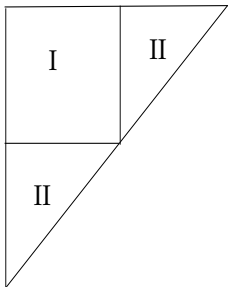
- The above picture illustrates dissipation of heat into a 2D grid.
- A differential equation rules this phenomenon.
- Once this discretized, through the finite element method, this leads a stencil computation.

## Pascal triangle construction: another stencil computation

	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	
1	3	6	10	15	21	28		
1	4	10	20	35	56			
1	5	15	35	70				
1	6	21	56					
1	7	28						
1	8							

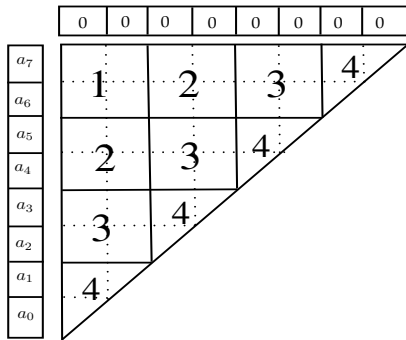
Construction of the Pascal Triangle: nearly [the simplest stencil computation!](#)

## Divide and conquer: principle



- Each triangle region can be computed as a square region followed by two (concurrent) triangle regions.
- Each square region can also be computed in a divide and conquer manner.

## Blocking strategy: principle



- Let  $B$  be the order of a block and  $n$  be the number of elements.
- Each block is processed serially (as a task) and the set of all blocks is computed concurrently.

# Plan

- 1 Hardware
- 2 Types of Parallelism
- 3 Concurrency Platforms: Three Examples
  - Julia
  - Cilk
  - CUDA
  - MPI

## Distributed arrays and parallel reduction (1/4)

```
[moreno@compute-0-3 ~]$ julia -p 5
```

```

      _
    _  _(_) _ | A fresh approach to technical computing
  ( _ ) | ( _ ) ( _ ) | Documentation: http://docs.julialang.org
    _ _ _ | | _ _ _ _ | Type "help()" to list help topics
  | | | | | | | / _ ' | |
  | | | _ | | | | ( _ | | | Version 0.2.0-prerelease+3622
 _ / | \ _ _ ' _ | _ | \ _ _ ' _ | | Commit c9bb96c 2013-09-04 15:34:41 UTC
| _ _ / | | | | | | | | | x86_64-redhat-linux

```

```
julia> da = @parallel [2i for i = 1:10]
10-element DArray{Int64,1,Array{Int64,1}}:
```

```

 2
 4
 6
 8
10
12
14
16
18
20

```

## Distributed arrays and parallel reduction (2/4)

```
julia> procs(da)
4-element Array{Int64,1}:
 2
 3
 4
 5
```

```
julia> da.chunks
4-element Array{RemoteRef,1}:
 RemoteRef(2,1,1)
 RemoteRef(3,1,2)
 RemoteRef(4,1,3)
 RemoteRef(5,1,4)
```

```
julia>
```

```
julia> da.indexes
4-element Array{(Range{Int64},),1}:
 (1:3,)
 (4:5,)
 (6:8,)
 (9:10,)
```

```
julia> da[3]
6
```

```
julia> da[3:5]
3-element SubArray{Int64,1,DArray{Int64,1,Array{Int64,1}},(Range{Int64},)}:
 6
 8
10
```

## Distributed arrays and parallel reduction (3/4)

```
julia> fetch(@spawnat 2 da[3])  
6
```

```
julia>
```

```
julia> { (@spawnat p sum(localpart(da))) for p=procs(da) }  
4-element Array{Any,1}:  
  RemoteRef(2,1,71)  
  RemoteRef(3,1,72)  
  RemoteRef(4,1,73)  
  RemoteRef(5,1,74)
```

```
julia>
```

```
julia> map(fetch, { (@spawnat p sum(localpart(da))) for p=procs(da) })  
4-element Array{Any,1}:  
 12  
 18  
 42  
 38
```

```
julia>
```

```
julia> sum(da)  
110
```



## Distributed arrays and parallel reduction (4/4)

```
julia> reduce(+, map(fetch,  
                    { (@spawnat p sum(localpart(da))) for p=procs(da) })))
```

```
110
```

```
julia>
```

```
julia> preduce(f,d) = reduce(f,  
                           map(fetch,  
                               { (@spawnat p f(localpart(d))) for p=procs(d) })))
```

```
# methods for generic function preduce
```

```
preduce(f,d) at none:1
```

```
julia> function Base.minimum(x::Int64, y::Int64)  
    min(x,y)  
end
```

```
minimum (generic function with 10 methods)
```

```
julia> preduce(minimum, da)
```

```
2
```

## From Cilk to Cilk++ and Cilk Plus

- Cilk has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.
- Besides being used for research and teaching, Cilk was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of Cilk have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to Cilk++, developed by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009 and became Cilk Plus, see <http://www.cilk.com/>
- Cilk++ can be freely downloaded at <http://software.intel.com/en-us/articles/download-intel-cilk-plus>
- Cilk is still developed at MIT <http://supertech.csail.mit.edu/cilk/>

## CilkPlus (and Cilk Plus)

- CilkPlus (resp. Cilk) is a [small set of linguistic extensions to C++](#) (resp. C) supporting [task parallelism](#), using fork & join constructs.
- Both Cilk and CilkPlus feature a [provably efficient work-stealing scheduler](#).
- CilkPlus provides a [hyperobject library](#) for performing reduction for data aggregation.
- CilkPlus includes the [Cilkscreen](#) race detector and the [Cilkview](#) performance analyzer.

## Task Parallelism in CilkPlus

```
int fib(int n)
{
    if (n < 2) return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x+y;
}
```

- The named **child** function `cilk_spawn fib(n-1)` may execute in parallel with its **parent**
- CilkPlus keywords `cilk_spawn` and `cilk_sync` grant **permissions for parallel execution**. They do not command parallel execution.

# The fork-join parallelism model

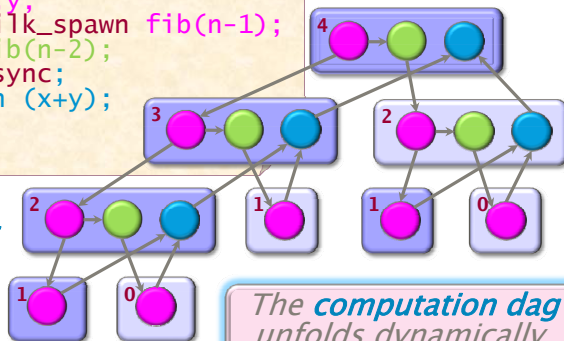
```

int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x+y);
  }
}

```

**Example:**  
**fib(4)**

*“Processor oblivious”*



At run time, the task DAG unfolds dynamically.

## Loop Parallelism in CilkPlus

$$\begin{matrix} \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} & \xrightarrow{\hspace{1cm}} & \begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{pmatrix} \\ A & & A^T \end{matrix}$$

```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

The iterations of a `cilk_for` loop may execute in parallel.

## Serial Semantics (1/2)

- Cilk (resp. CilkPlus) is a multithreaded language for parallel programming that generalizes the semantics of C (resp. C++) by introducing linguistic constructs for parallel control.
- Cilk (resp. CilkPlus) is a **faithful extension** of C (resp. C++):
  - The C (resp. C++) elision of a Cilk (resp. CilkPlus) is a correct implementation of the semantics of the program.
  - Moreover, on one processor, a parallel Cilk (resp. CilkPlus) program scales down to run nearly as fast as its C (resp. C++) elision.
- To obtain the serialization of a CilkPlus program

```
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```

## Serial Semantics (2/2)

```
int fib (int n) {  
    if (n<2) return (n);  
    else {  
        int x,y;  
        x = cilk_spawn fib(n-1);  
        y = fib(n-2);  
        cilk_sync;  
        return (x+y);  
    }  
}
```

Cilk++ source

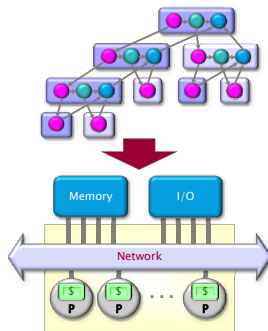


```
int fib (int n) {  
    if (n<2) return (n);  
    else {  
        int x,y;  
        x = fib(n-1);  
        y = fib(n-2);  
        return (x+y);  
    }  
}
```

Serialization



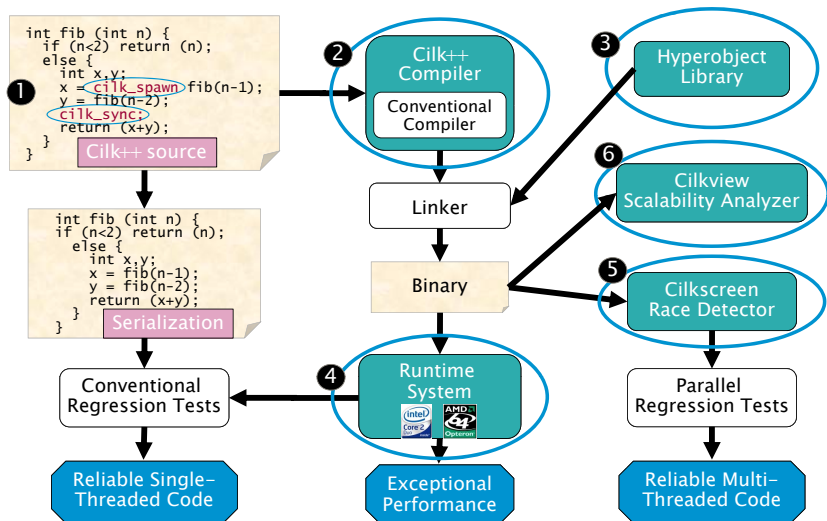
# Scheduling



A **scheduler**'s job is to map a computation to particular processors. Such a mapping is called a **schedule**.

- If decisions are made at runtime, the scheduler is *online*, otherwise, it is *offline*
- CilkPlus's scheduler maps strands onto processors dynamically at runtime.

# The CilkPlus Platform



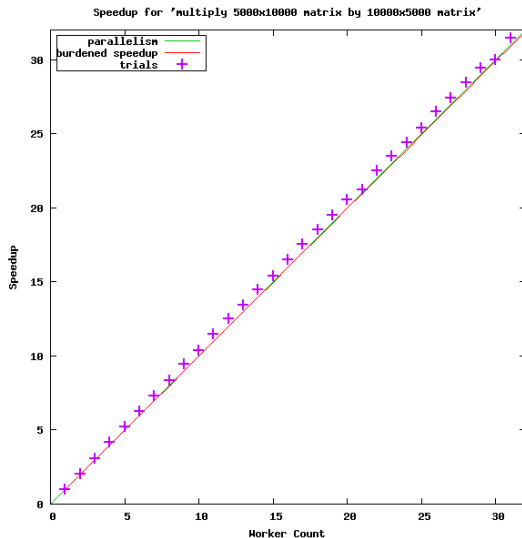
## Benchmarks for parallel divide-and-conquer matrix multiplication

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets x 4 cores (Quad Core AMD Opteron 8354) per socket.
- The 32 cores share a L3 32-way set-associative cache of 2 Mbytes.

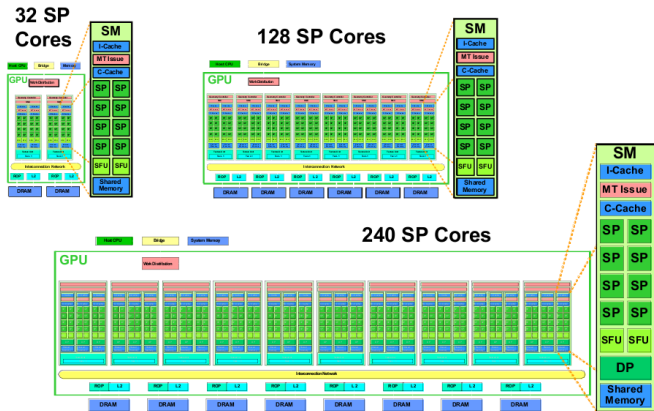
#core	Elision (s)	Parallel (s)	speedup
8	420.906	51.365	8.19
16	432.419	25.845	16.73
24	413.681	17.361	23.83
32	389.300	13.051	29.83

# Using Cilkview



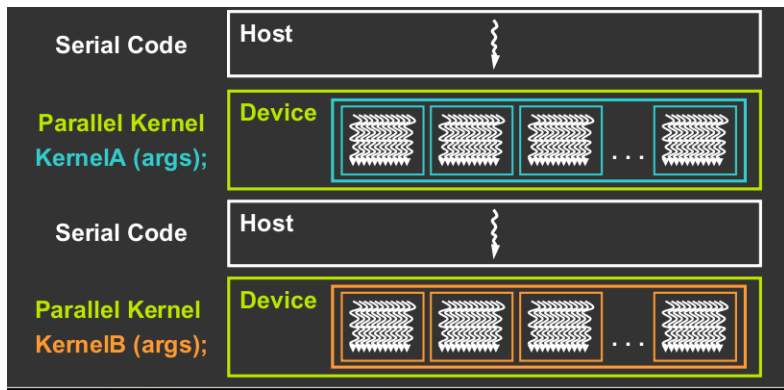
## CUDA design goals

- Enable heterogeneous systems (i.e., CPU+GPU)
- Scale to 100's of cores, 1000's of parallel threads
- Use C/C++ with minimal extensions
- Let programmers focus on parallel algorithms (as much as possible).



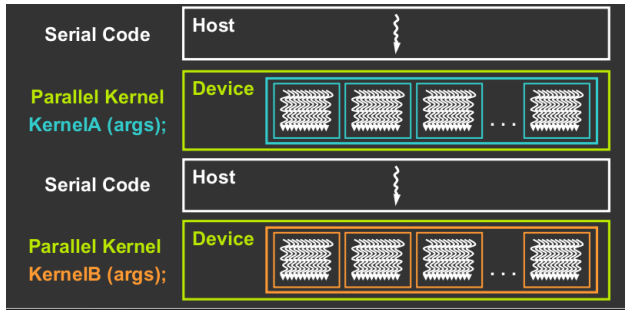
## Heterogeneous programming (1/3)

- A CUDA program is a serial program with parallel kernels, all in C.
- The serial C code executes in a **host** (= CPU) thread
- The parallel kernel C code executes in many **device** threads across multiple GPU processing elements, called **streaming processors** (SP).



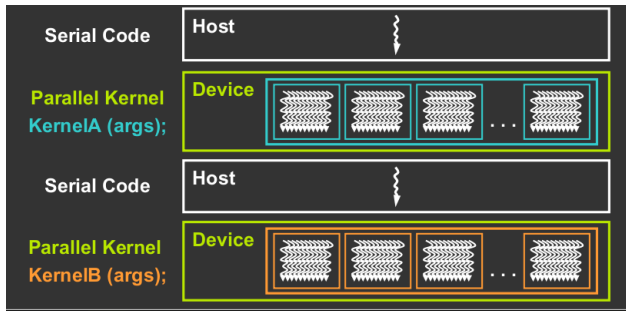
## Heterogeneous programming (2/3)

- Thus, the parallel code (kernel) is launched and executed on a device by many threads.
- Threads are grouped into thread blocks.
- One kernel is executed at a time on the device.
- Many threads execute each kernel.



## Heterogeneous programming (3/3)

- The parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in **thread and block ID variables** are used to map each thread to a specific data tile (see next slide).
- Thus, each thread executes the same code on different data based on its thread and block ID.





## Example: increment array elements (1/2)

Increment N-element vector a by scalar b



Let's assume  $N=16$ ,  $\text{blockDim}=4 \rightarrow 4$  blocks

```
int idx = blockDim.x * blockIdx.x + threadIdx.x;
```



$\text{blockIdx.x}=0$   
 $\text{blockDim.x}=4$   
 $\text{threadIdx.x}=0,1,2,3$   
 $\text{idx}=0,1,2,3$



$\text{blockIdx.x}=1$   
 $\text{blockDim.x}=4$   
 $\text{threadIdx.x}=0,1,2,3$   
 $\text{idx}=4,5,6,7$



$\text{blockIdx.x}=2$   
 $\text{blockDim.x}=4$   
 $\text{threadIdx.x}=0,1,2,3$   
 $\text{idx}=8,9,10,11$



$\text{blockIdx.x}=3$   
 $\text{blockDim.x}=4$   
 $\text{threadIdx.x}=0,1,2,3$   
 $\text{idx}=12,13,14,15$

See our example number 4 in `/usr/local/cs4402/examples/4`

## Example: increment array elements (2/2)

### CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}
```

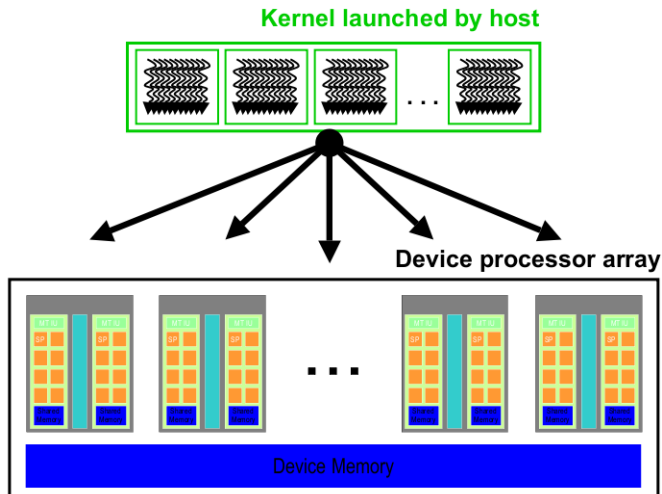
```
void main()
{
    ....
    increment_cpu(a, b, N);
}
```

### CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}
```

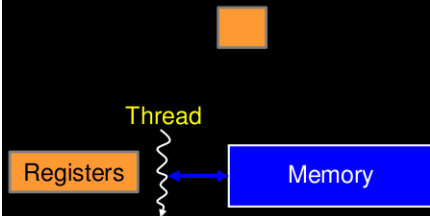
```
void main()
{
    ....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

## Blocks run on multiprocessors

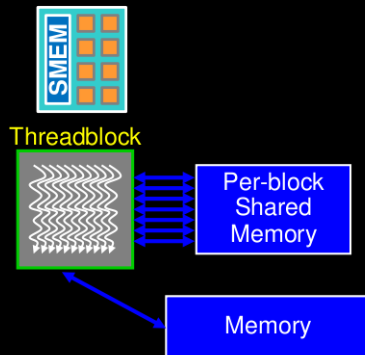


## Streaming processors and multiprocessors

### Streaming Processor



### Streaming Multiprocessor



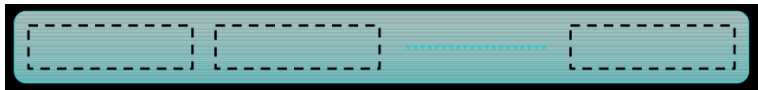
## Hardware multithreading

- **Hardware allocates resources to blocks:**
  - blocks need: thread slots, registers, shared memory
  - blocks don't run until resources are available
- **Hardware schedules threads:**
  - threads have their own registers
  - any thread not waiting for something can run
  - context switching is free every cycle
- **Hardware relies on threads to hide latency:**
  - thus high parallelism is necessary for performance.



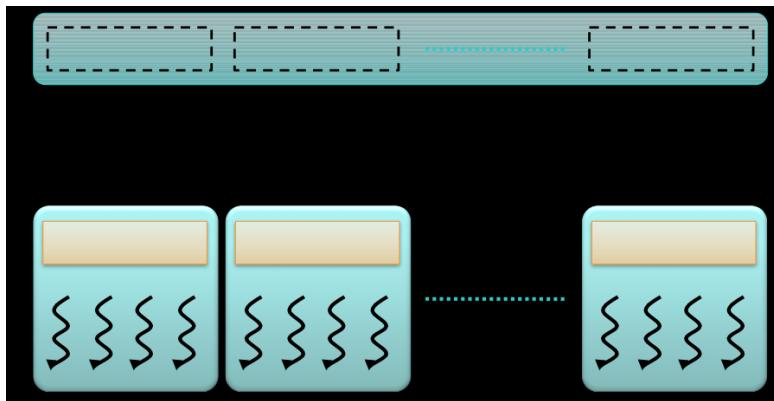
## A Common programming strategy

Partition data into subsets that fit into shared memory



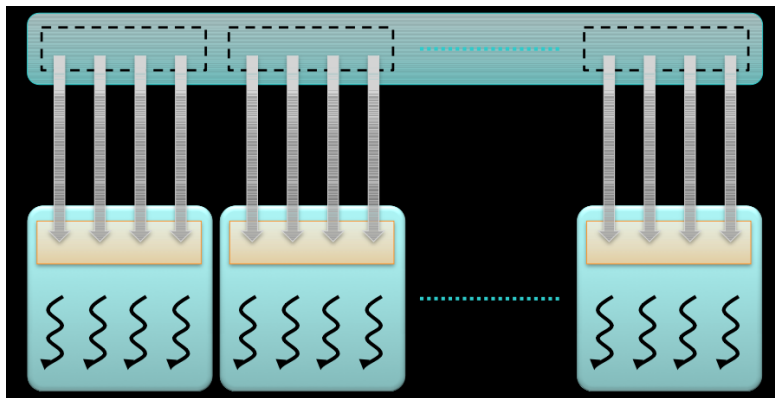
## A Common Programming Strategy

Handle each data subset with one thread block



## A Common programming strategy

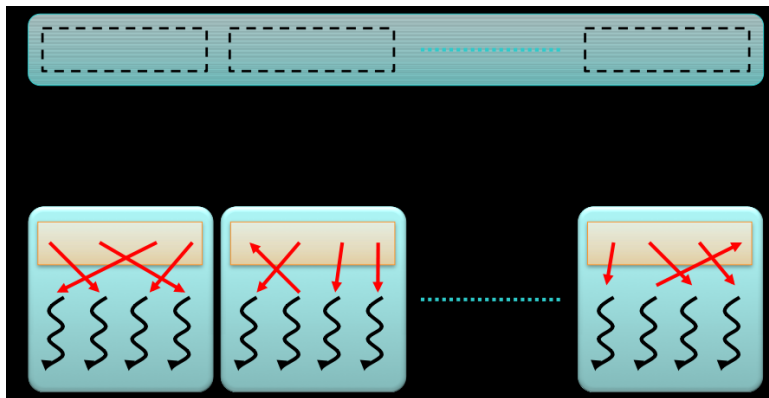
Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.





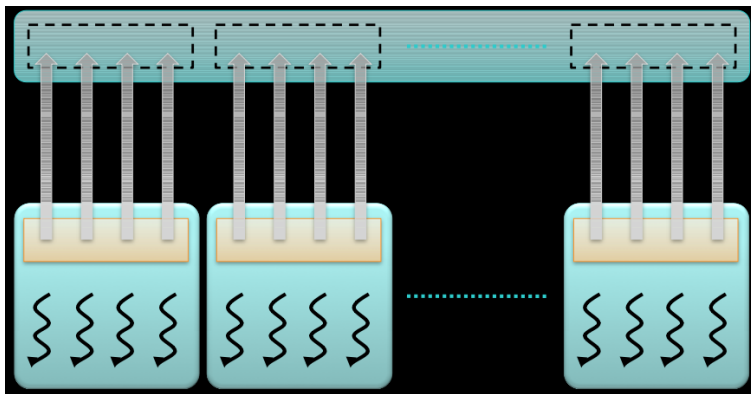
## A Common programming strategy

Perform the computation on the subset from shared memory.



## A Common programming strategy

Copy the result from shared memory back to global memory.



## What is the Messaging Passing Interface (MPI)?

A language-independent communication protocol for parallel computers

- Run the same code on a number of nodes (different hardware threads, servers)
- Explicit message passing
- Dominant model for high performance computing

## High Level Presentation of MPI

- MPI is a type of SPMD (single process, multiple data)
- Idea: to have multiple instances of the same program all working on different data
- The program could be running on the same machine, or cluster of machines
- Allow simple communication of data between processes

## MPI Functions

```
// Initialize MPI
int MPI_Init(int *argc, char **argv)

// Determine number of processes within a communicator
int MPI_Comm_size(MPI_Comm comm, int *size)

// Determine processor rank within a communicator
int MPI_Comm_rank(MPI_Comm comm, int *rank)

// Exit MPI (must be called last by all processors)
int MPI_Finalize()

// Send a message
int MPI_Send (void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)

// Receive a message
int MPI_Recv (void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

## MPI Function Notes

- `MPI_Datatype` is just an enum, `MPI_Comm` is commonly `MPI_COMM_WORLD` for the global communication channel
- `dest/source` are the rank of the process to send the message to/receive the message from:
  - You may use `MPI_ANY_SOURCE` in `MPI_Recv`
- Both `MPI_Send` and `MPI_Recv` are blocking calls
- You can use `man MPI_Send` or `man MPI_Recv` for good documentation
- The tag allows you to organize your messages, so you can receive only a specific tag

## Example

Here's a common example:

- Have the master (rank 0) process create some strings and send them to the worker processes
- The worker processes modify the string and send it back to the master

## Example Code (1)

```
/*  
 "Hello World" MPI Test Program  
*/  
#include <mpi.h>  
#include <stdio.h>  
#include <string.h>  
  
#define BUFSIZE 128  
#define TAG 0  
  
int main(int argc, char *argv[])  
{  
    char idstr[32];  
    char buff[BUFSIZE];  
    int numprocs;  
    int myid;  
    int i;  
    MPI_Status stat;
```



## Example Code (2)

```
/* all MPI programs start with MPI_Init; all 'N'
 * processes exist thereafter
 */
MPI_Init(&argc,&argv);

/* find out how big the SPMD world is */
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

/* and this processes' rank is */
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

/* At this point, all programs are running equivalently,
 * the rank distinguishes the roles of the programs in
 * the SPMD model, with rank 0 often used specially...
 */
```

## Example Code (3)

```
if (myid == 0)
{
    printf("%d: We have %d processors\n", myid, numprocs);
    for (i=1; i<numprocs; i++)
    {
        sprintf(buff, "Hello %d! ", i);
        MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG,
                  MPI_COMM_WORLD);
    }
    for (i=1; i<numprocs; i++)
    {
        MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG,
                  MPI_COMM_WORLD, &stat);
        printf("%d: %s\n", myid, buff);
    }
}
```

## Example Code (4)

```
else
{
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG,
             MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strncat(buff, idstr, BUFSIZE-1);
    strncat(buff, "reporting for duty", BUFSIZE-1);
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG,
             MPI_COMM_WORLD);
}

/* MPI Programs end with MPI Finalize; this is a weak
 * synchronization point
 */
MPI_Finalize();
return 0;
}
```

## Compiling

```
// Wrappers for gcc (C/C++)
mpicc
mpicxx

// Compiler Flags
OMPI_MPICC_CFLAGS
OMPI_MPICXX_CXXFLAGS

// Linker Flags
OMPI_MPICC_LDFLAGS
OMPI_MPICXX_LDFLAGS
```

OpenMPI does not recommend you to set the flags yourself, to see them try:

```
# Show the flags necessary to compile MPI C applications
shell$ mpicc --showme:compile

# Show the flags necessary to link MPI C applications
shell$ mpicc --showme:link
```

## Compiling and Running

```
mpirun -np <num_processors> <program>  
mpiexec -np <num_processors> <program>
```

- Starts `num_processors` instances of the program using MPI

```
jon@riker examples master % mpicc hello_mpi.c  
jon@riker examples master % mpirun -np 8 a.out  
0: We have 8 processors  
0: Hello 1! Processor 1 reporting for duty  
0: Hello 2! Processor 2 reporting for duty  
0: Hello 3! Processor 3 reporting for duty  
0: Hello 4! Processor 4 reporting for duty  
0: Hello 5! Processor 5 reporting for duty  
0: Hello 6! Processor 6 reporting for duty  
0: Hello 7! Processor 7 reporting for duty
```

- By default, MPI uses the lowest-latency resource available (shared memory in this case)

## Other Things MPI Can Do

- We can use nodes on a network (by using a hostfile)
- We can even use MPMD, for multiple processes, multiple data.

```
mpi run np 2 a.out : np 2 b.out
```

- All in the same MPI\_COMM\_WORLD
- Ranks 0 and 1 are instances of a.out
- Ranks 2 and 3 are instances of b.out
- You could also use the app flag with an appfile instead of typing out everything

## Performance Considerations and concluding remarks

- Your bottleneck for performance here is messages
- Keep the communication to a minimum
- The more machines, the slower the communication in general
- MPI is a powerful tool for highly parallel computing across multiple machines
- Programming is similar to a more powerful version of `fork/join`