CS4403 - CS9535: An Overview of Parallel Computing

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

January 10, 2017

Plan

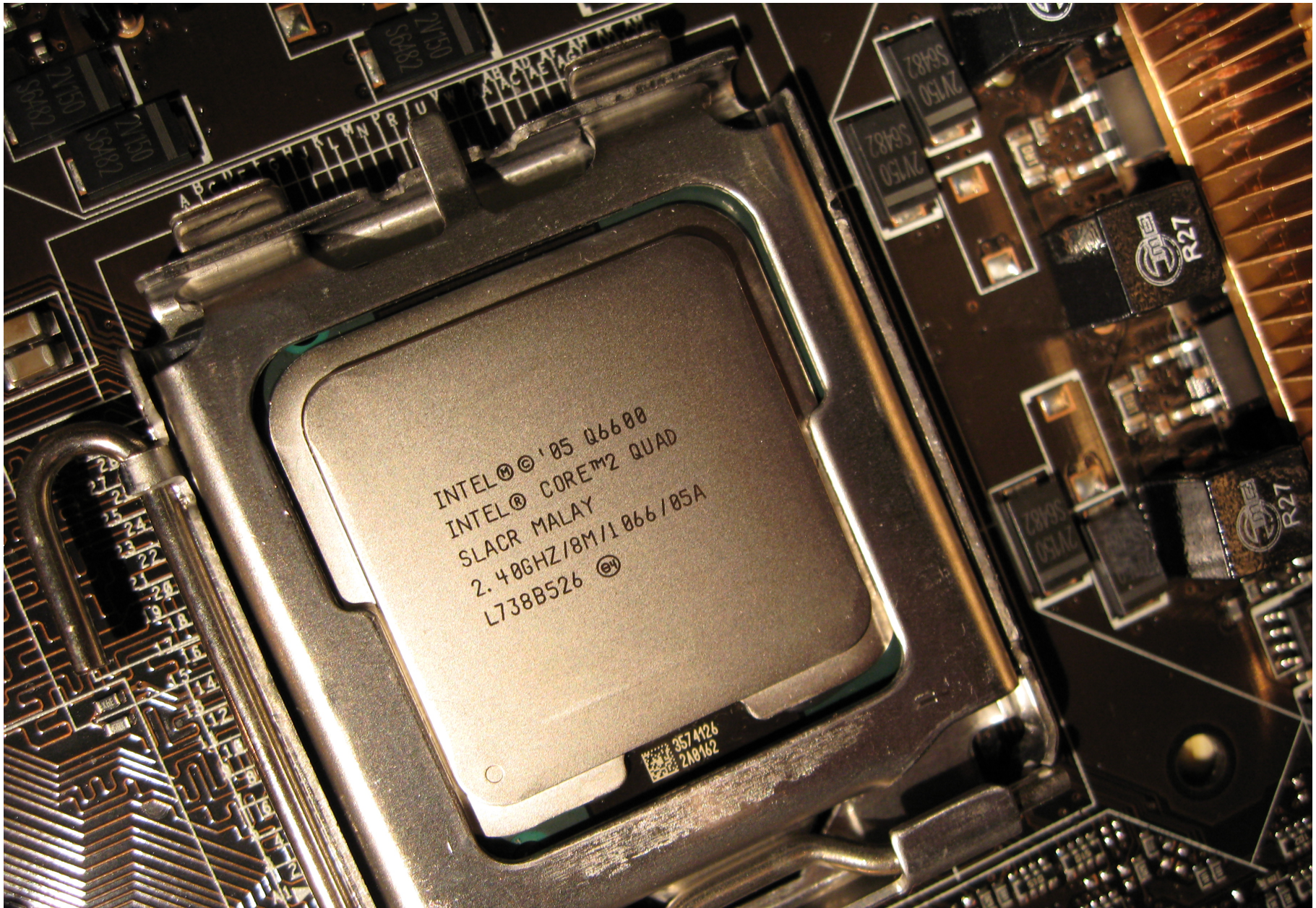
1. Hardware
2. Types of Parallelism
3. Concurrency Platforms: Three Examples
   - Julia
   - Cilk
   - CUDA
   - MPI

Plan

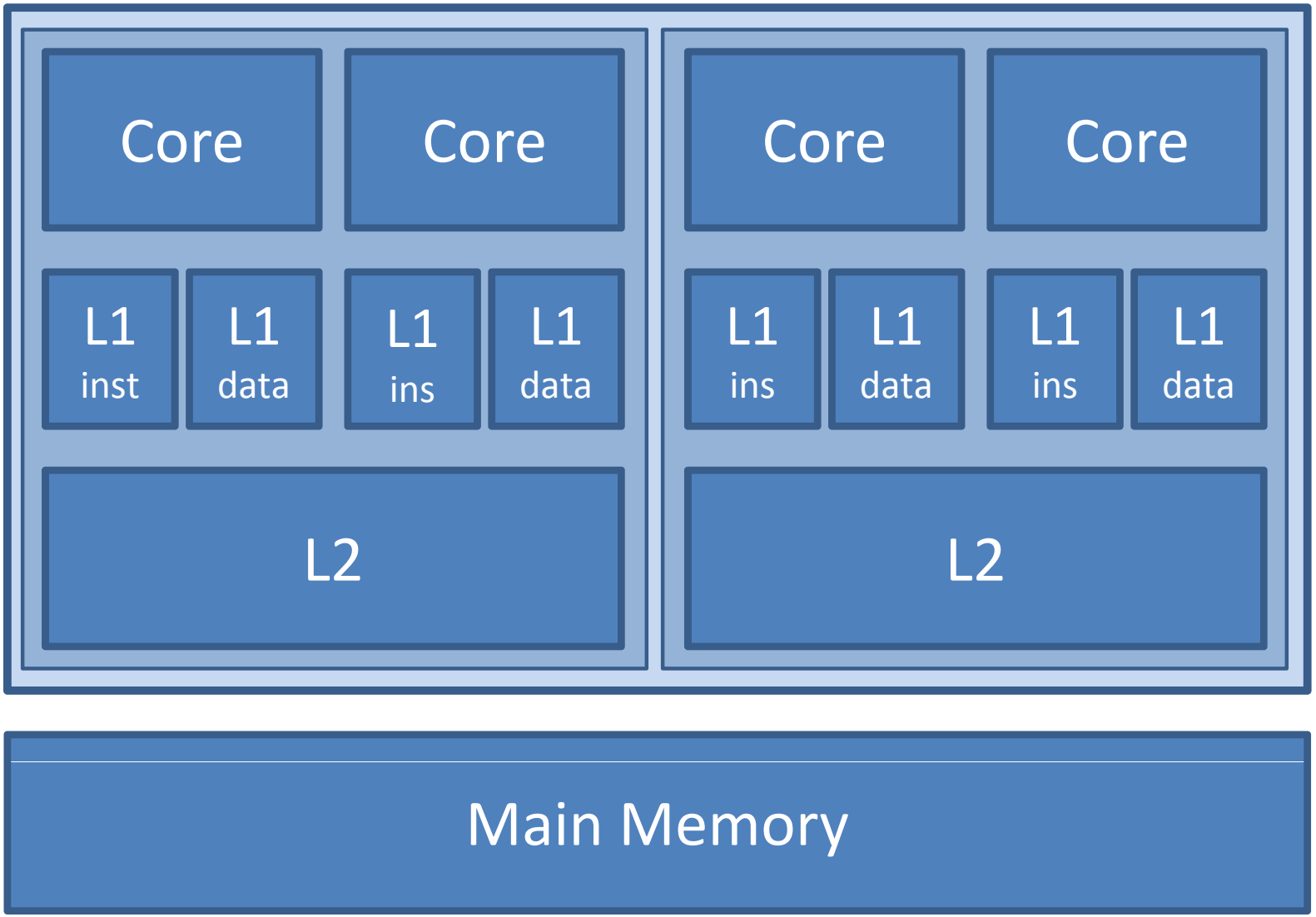
1. Hardware
2. Types of Parallelism
3. Concurrency Platforms: Three Examples
   - Julia
   - Cilk
   - CUDA
   - MPI

The Pentium Family

Multicore processors

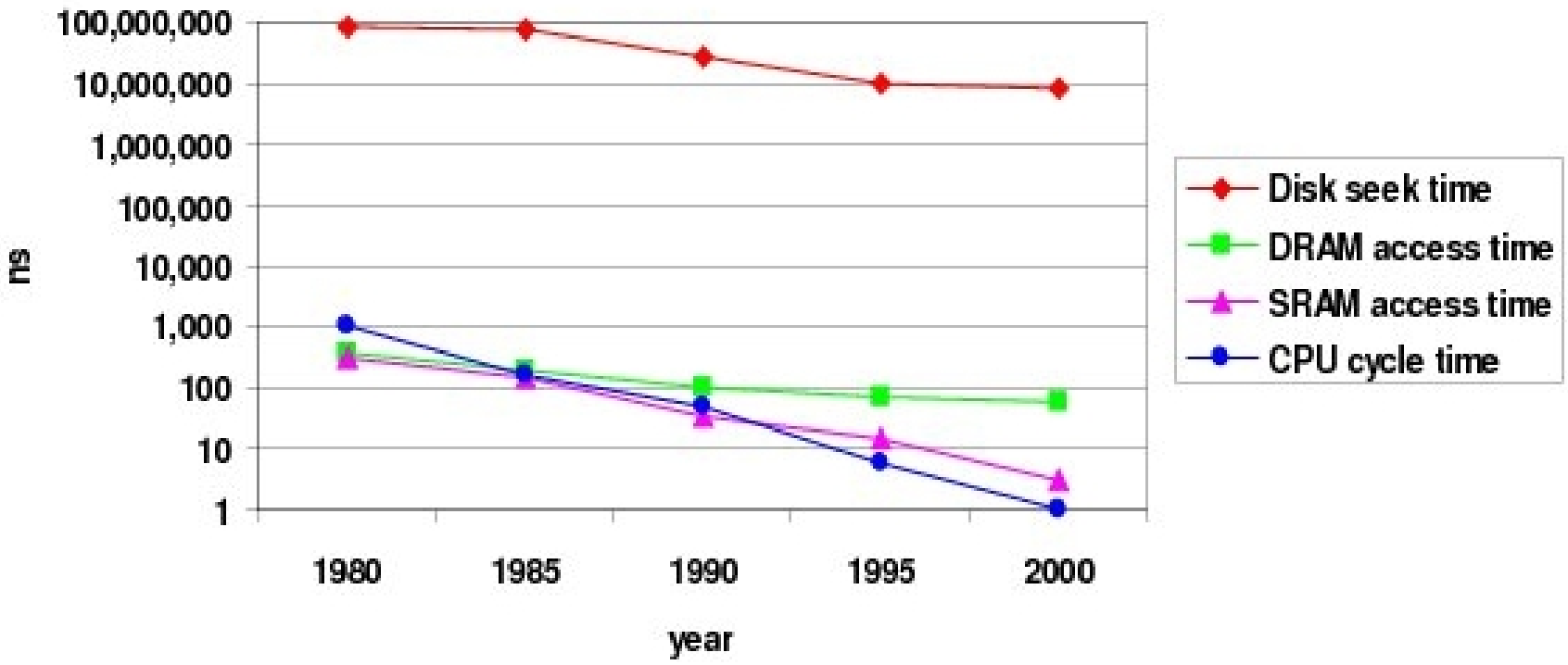


Multicore processors



The CPU-Memory Gap

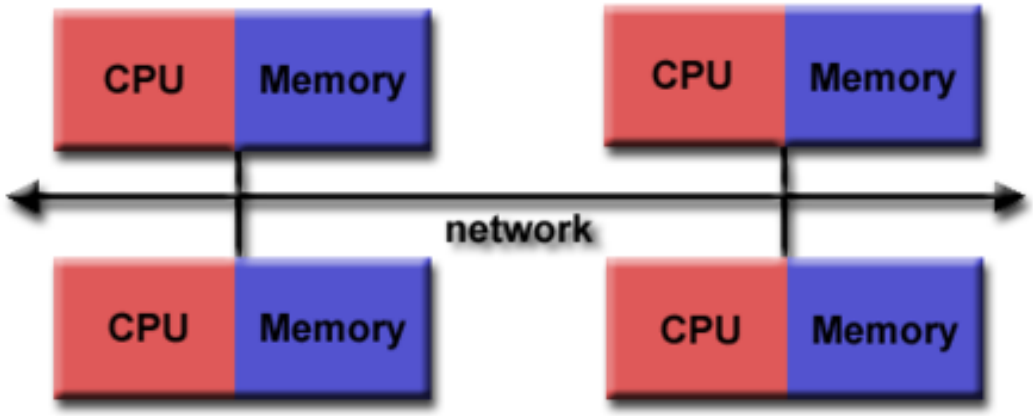
The increasing gap between DRAM, disk, and CPU speeds.



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

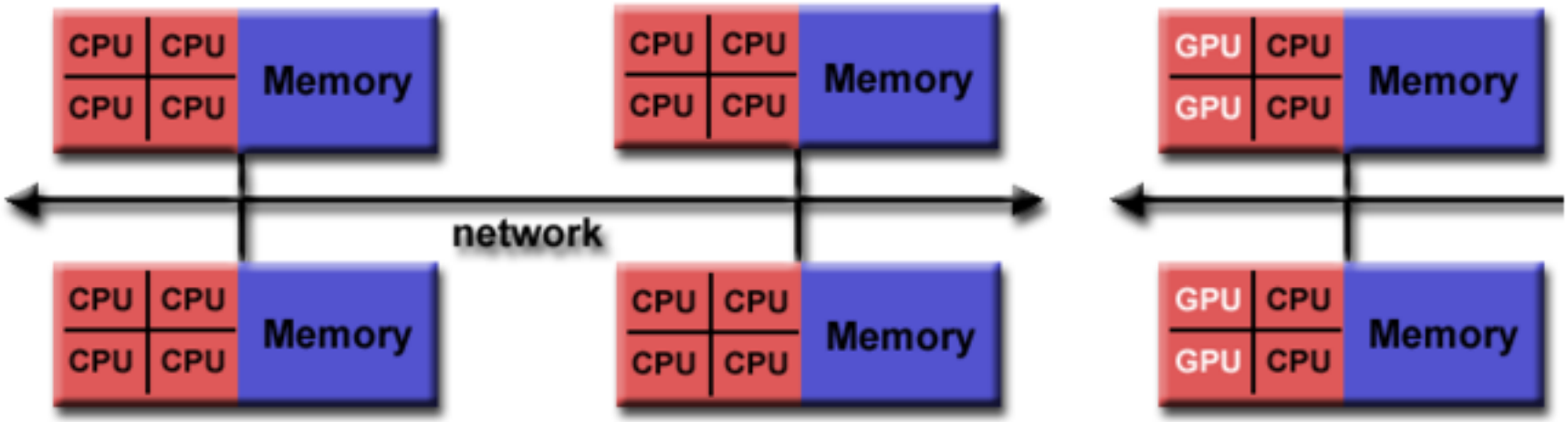
Graphics processing units (GPUs)

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.

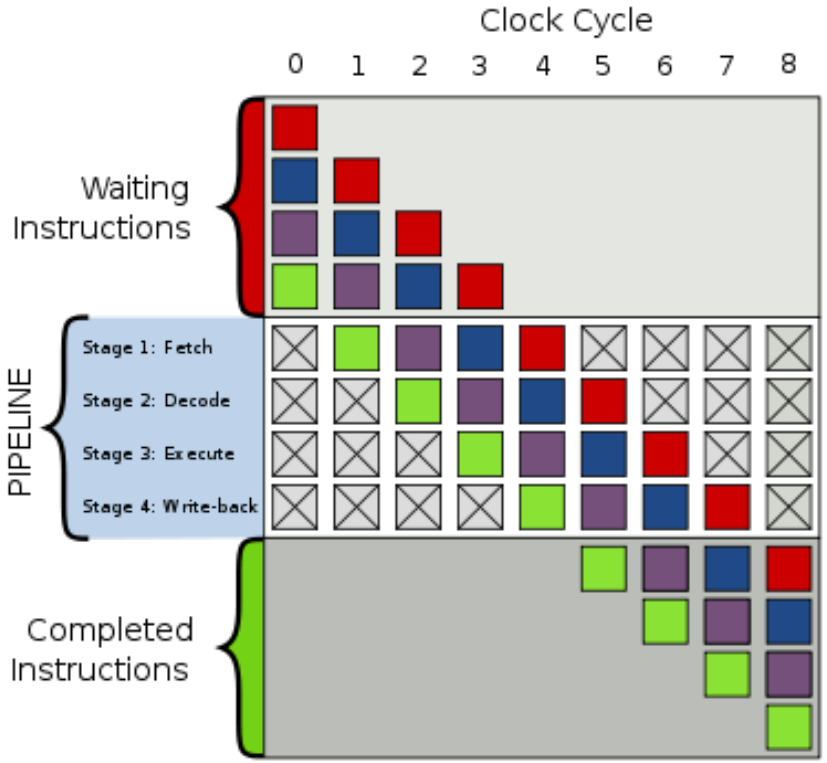
Plan

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

Pipelining

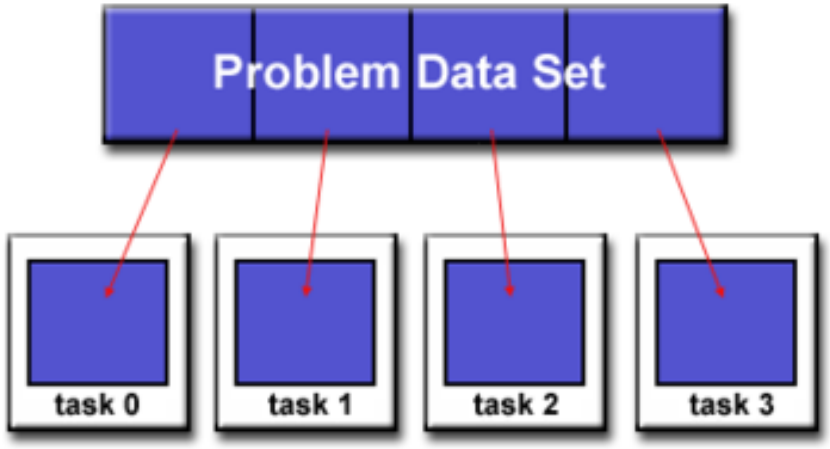
- Pipelining is a common way to organize work with the objective of optimizing throughput.
- It turns out that this is also a way to execute concurrently several tasks (that is, work units) processable by the same pipeline.

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".

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.
-
-

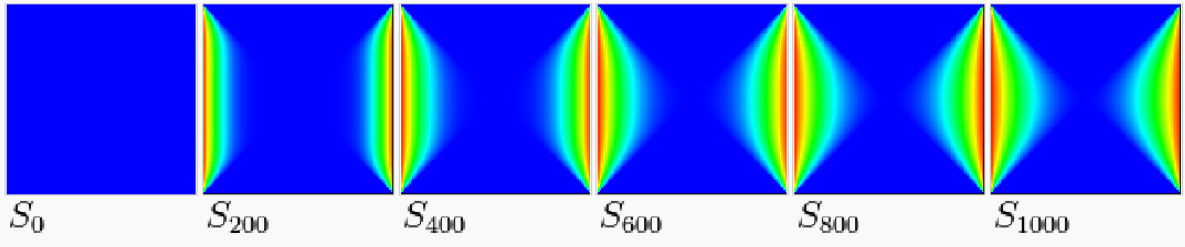
Task parallelism (2/4)

```
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.

Stencil computations

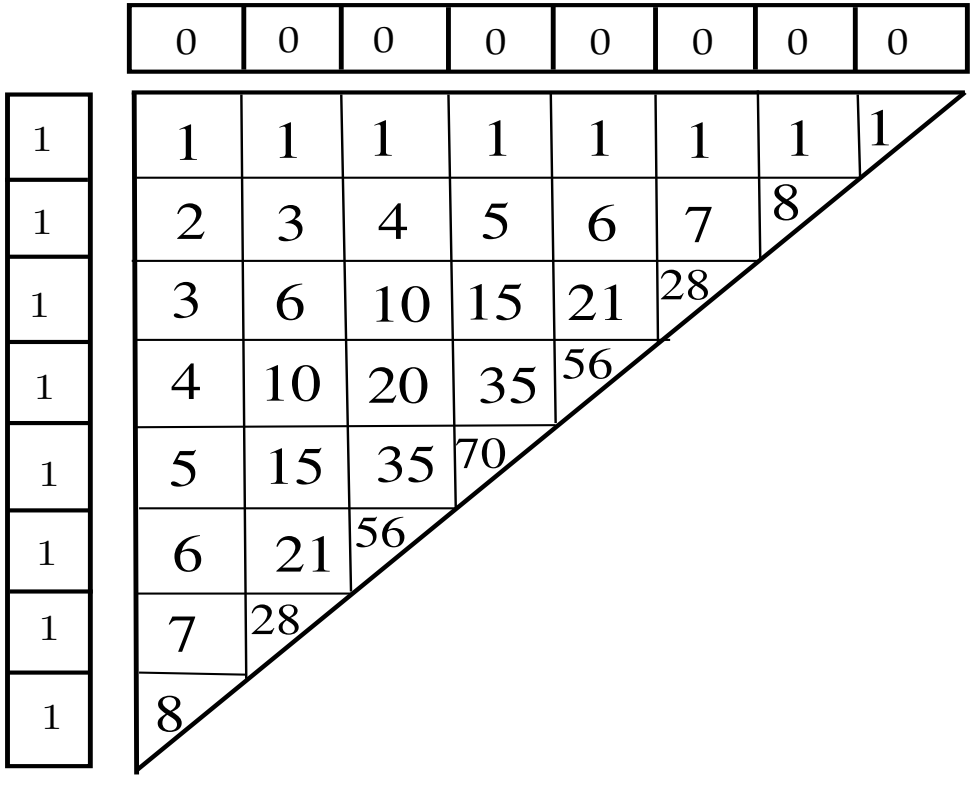
$$I = [0, \ldots, 99]^2$$
$$S = \mathbb{R}$$
$$S_0 : \mathbb{Z}^2 \to \mathbb{R}$$
$$S_0((x,y)) = \begin{cases} 1, & x < 0 \\ 0, & 0 \leq x < 100 \\ 1, & x \geq 100 \end{cases}$$
$$s = ((0,-1),(-1,0),(1,0),(0,1))$$
$$T : \mathbb{R}^4 \to \mathbb{R}$$
$$T((x_1,x_2,x_3,x_4)) = 0.25 \cdot (x_1 + x_2 + x_3 + x_4)$$



2D Jacobi Iteration on a 100^2 Array

- 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 100×100 elements is updated by the stencil T.

Pascal triangle construction: another stencil computation



Construction of the Pascal Triangle: nearly the simplest stencil computation!

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{(Range1{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}},(Range1{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)

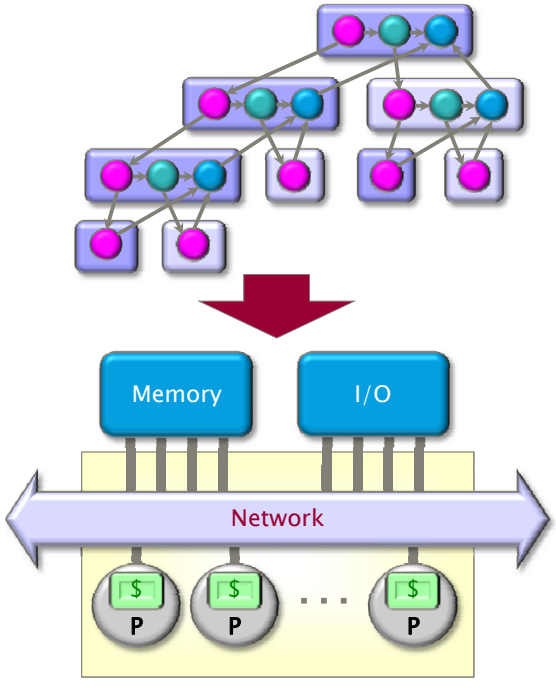
```
julia> reduce(+, map(fetch,
                { (@spawnat p sum(localpart(da))) for p=procs(da) }))
110

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
```

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.

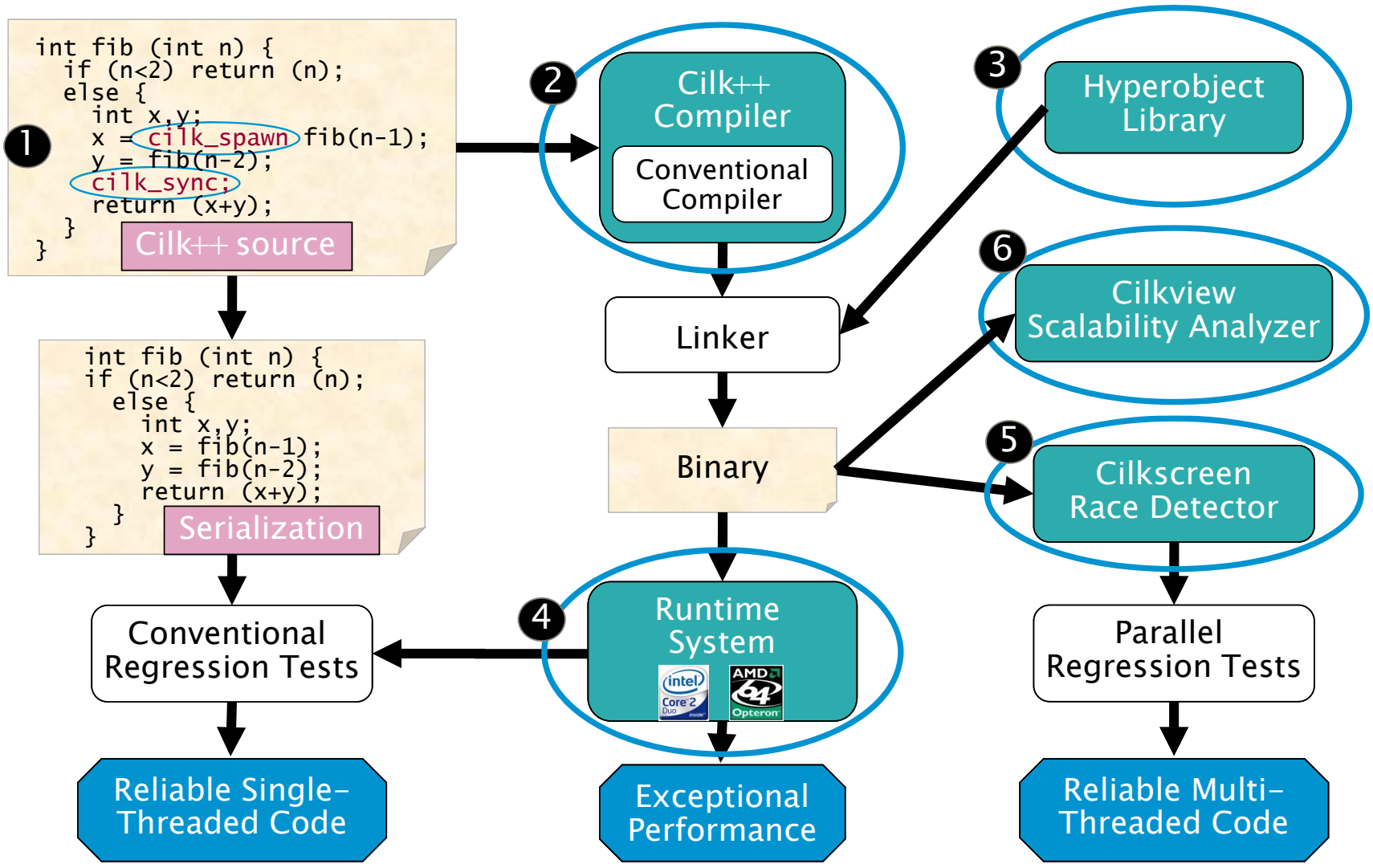
Scheduling



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



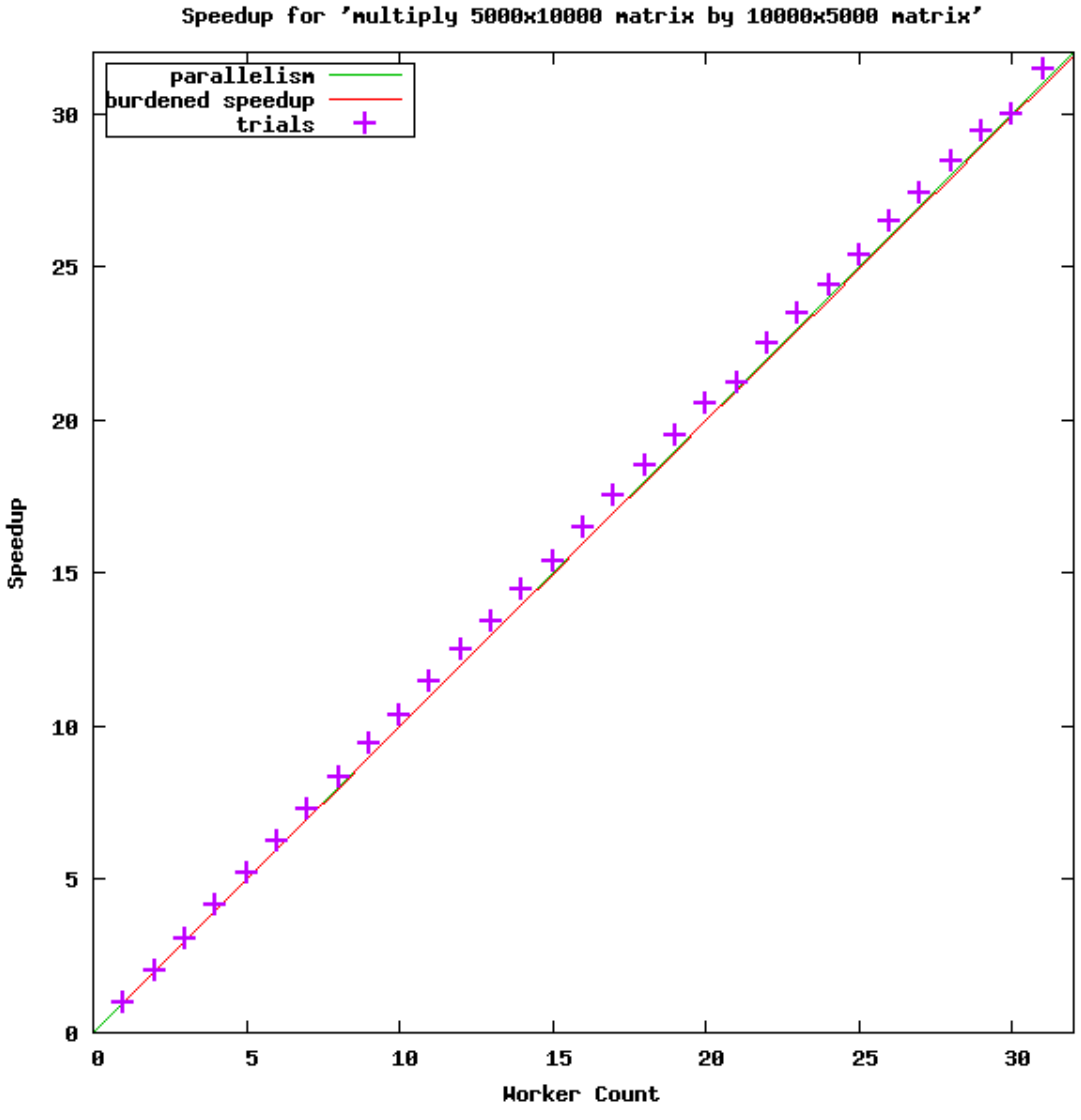
Benchmarks for parallel divide-and-conquer matrix multiplication

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.

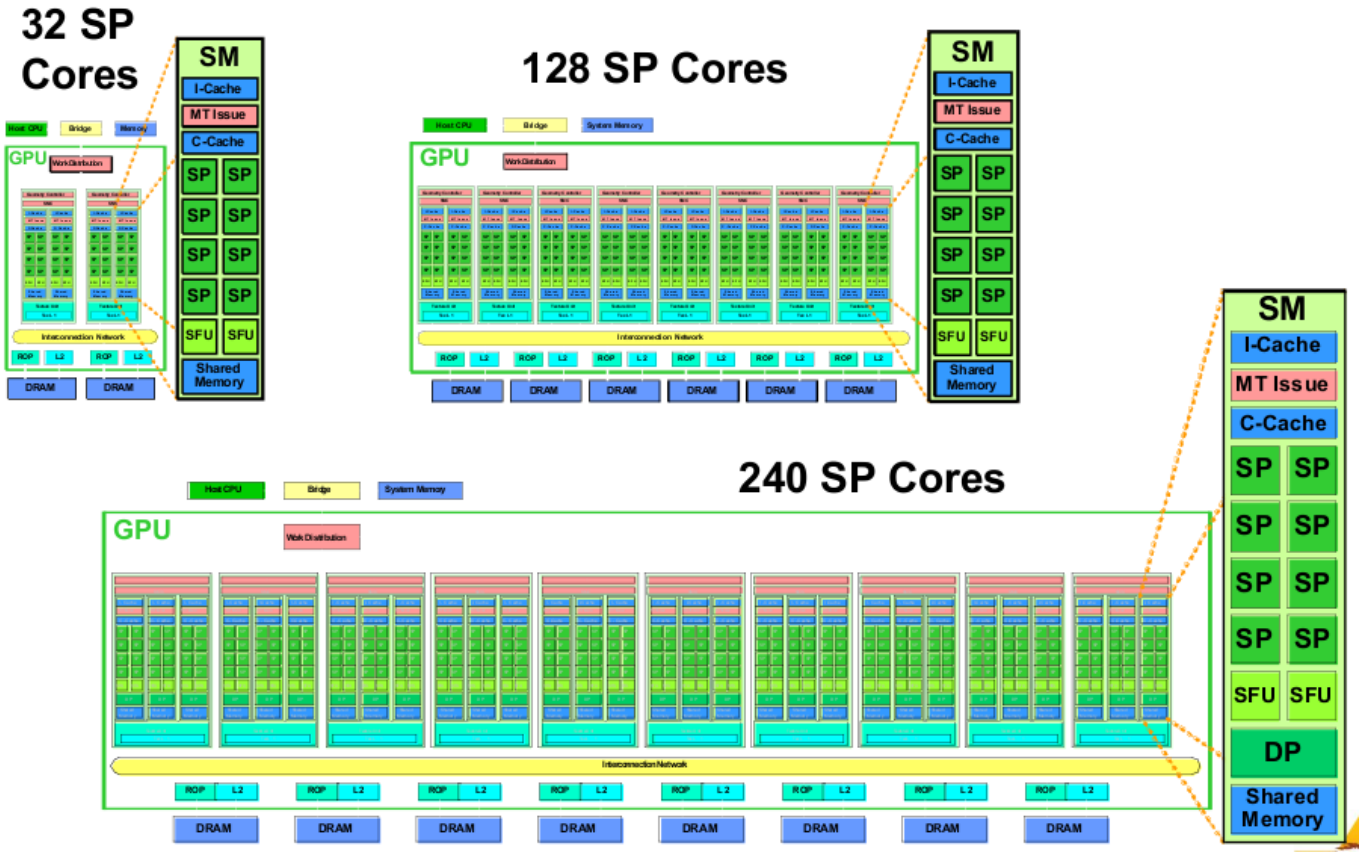
#core	Elision (s)	Parallel (s)	speedup
8	420.906	51.365	8.19
16	432.419	25.845	16.73
24	413.681	17.361	23.83
32	389.300	13.051	29.83

Uisng `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).



Example: increment array elements (1/2)

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

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

blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

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

Example: increment array elements (2/2)

CPU program

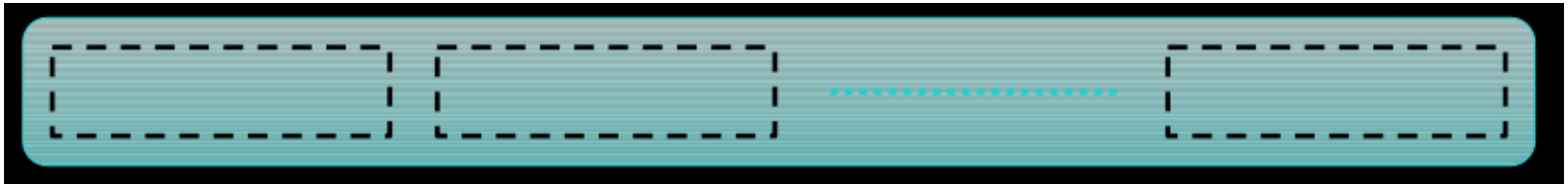
```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}

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);
}
```

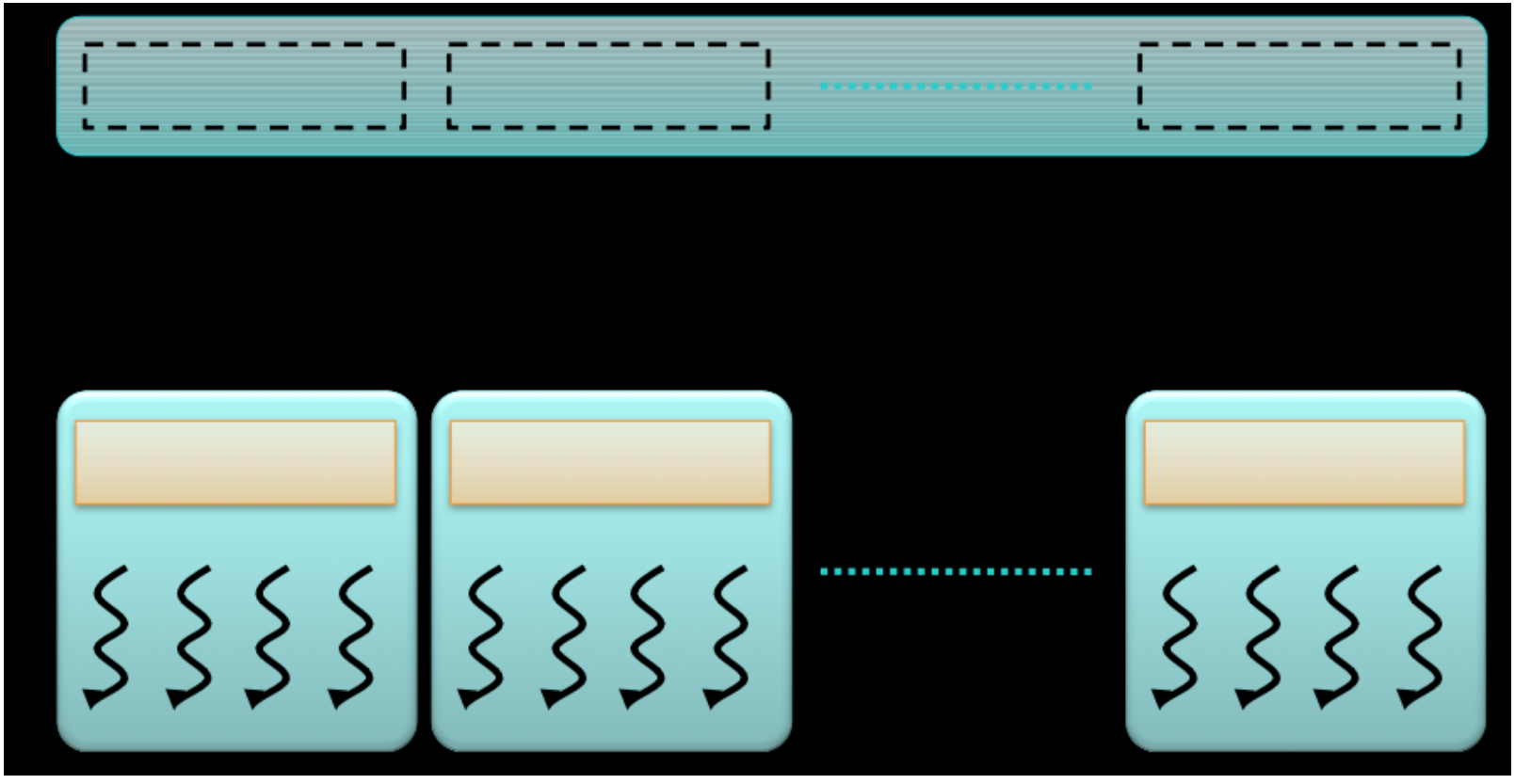
A Common programming strategy

Partition data into subsets that fit into shared memory



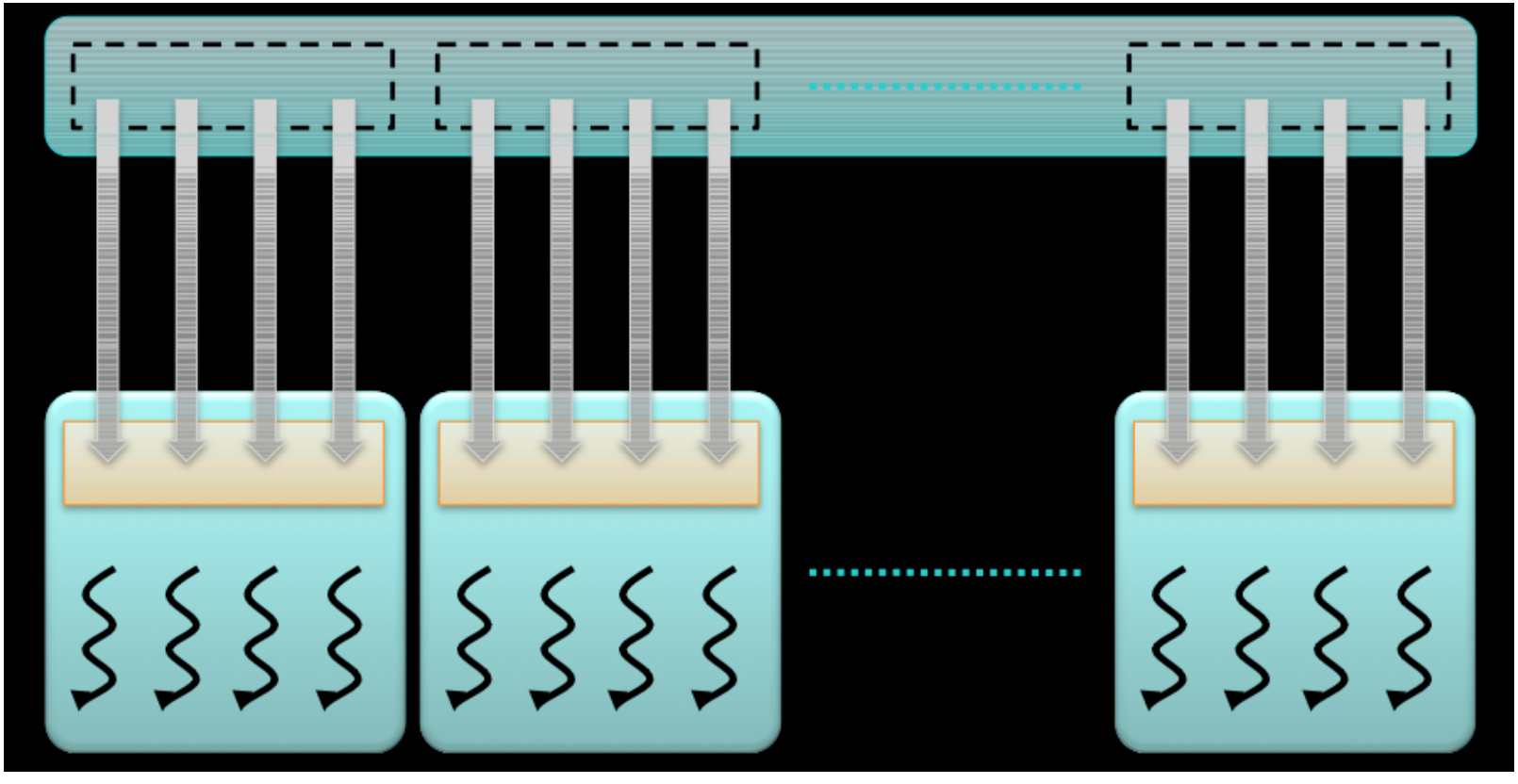
A Common Programming Strategy

Handle each data subset with one thread block



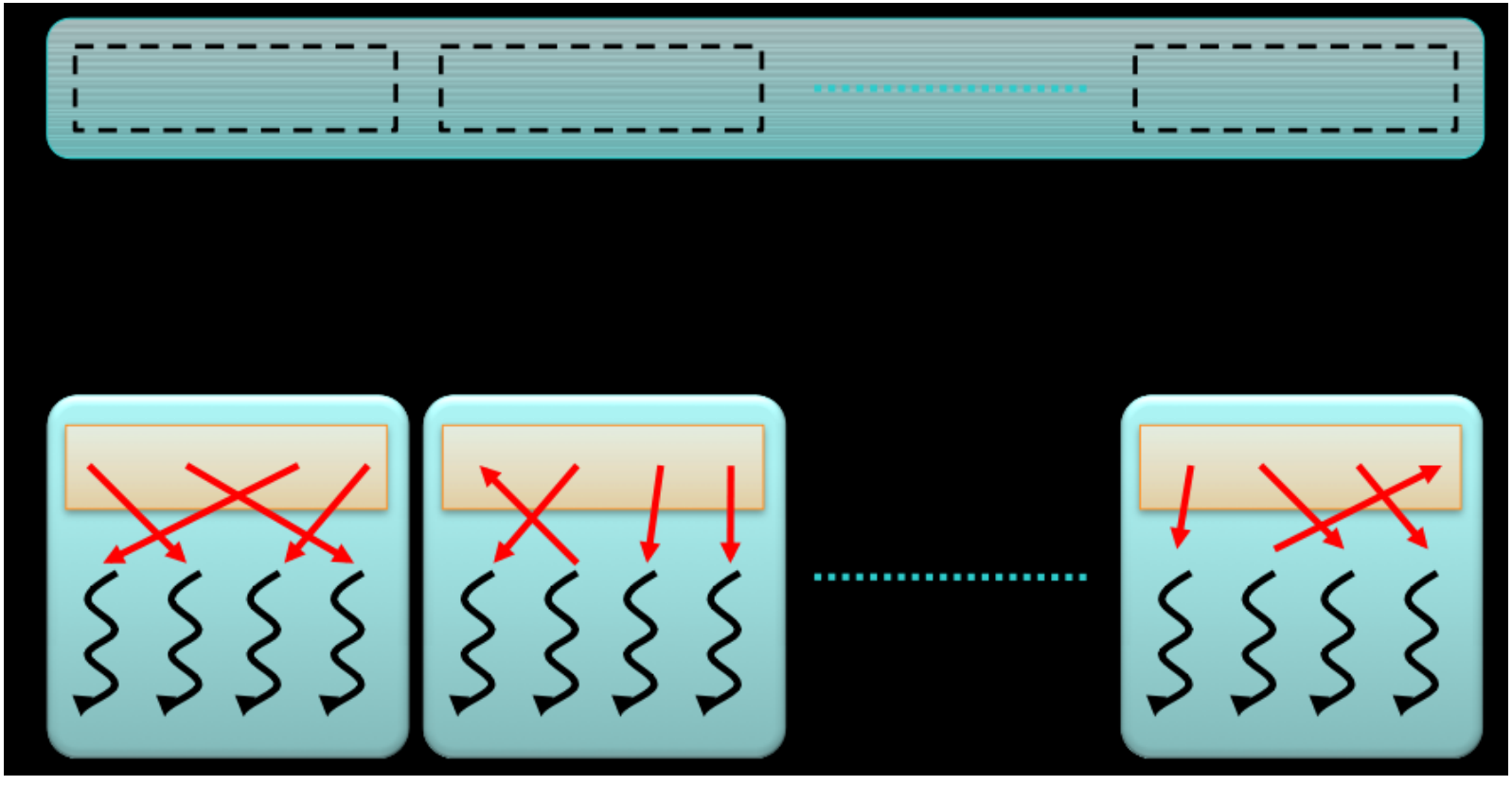
A Common programming strategy

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



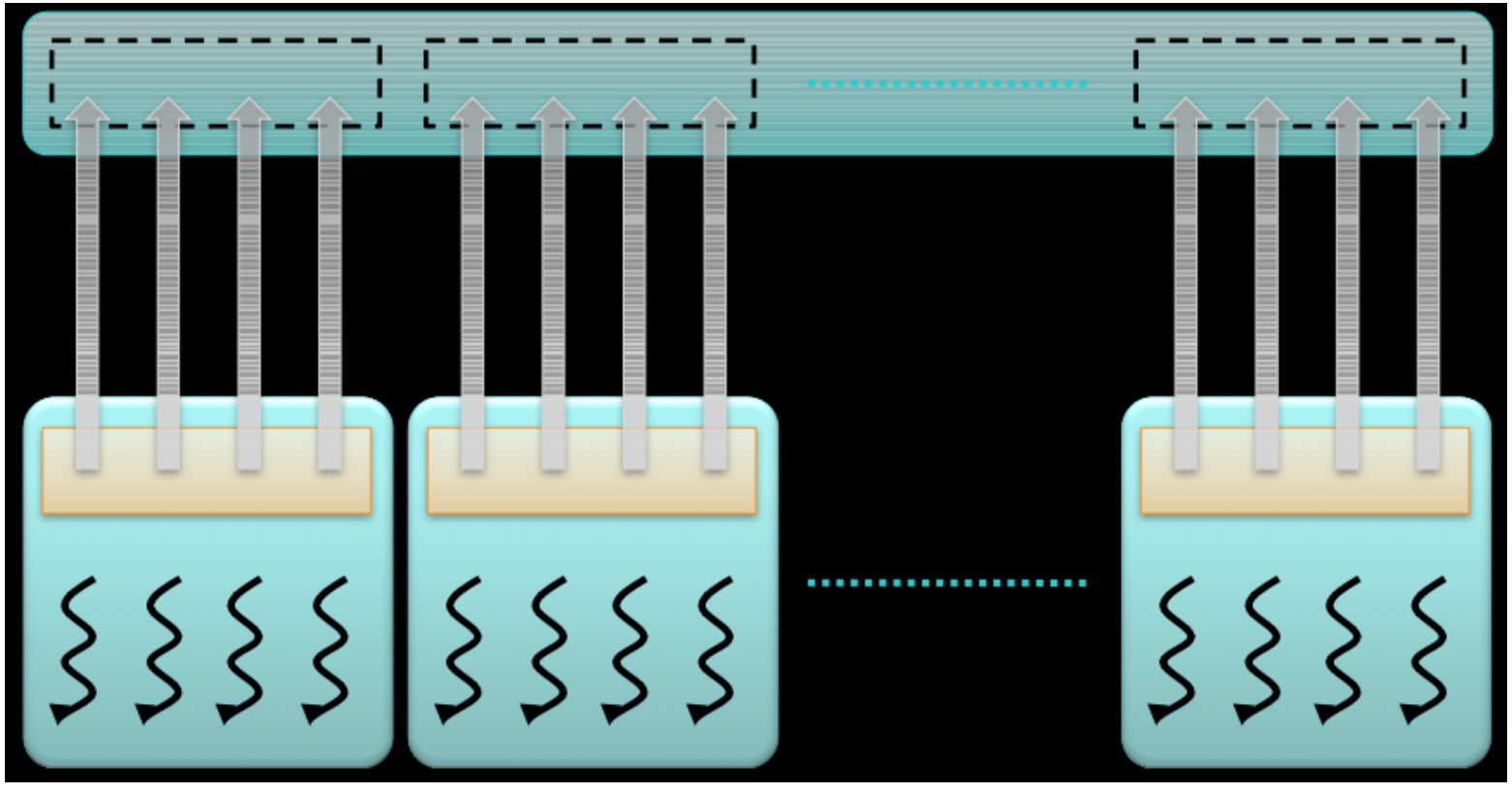
A Common programming strategy

Perform the computation on the subset from shared memory.



A Common programming strategy

Copy the result from shared memory back to global memory.



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;
}
```