

# Parallel Programming Overview

Introduction to High Performance Computing

Dr. Tim Cramer

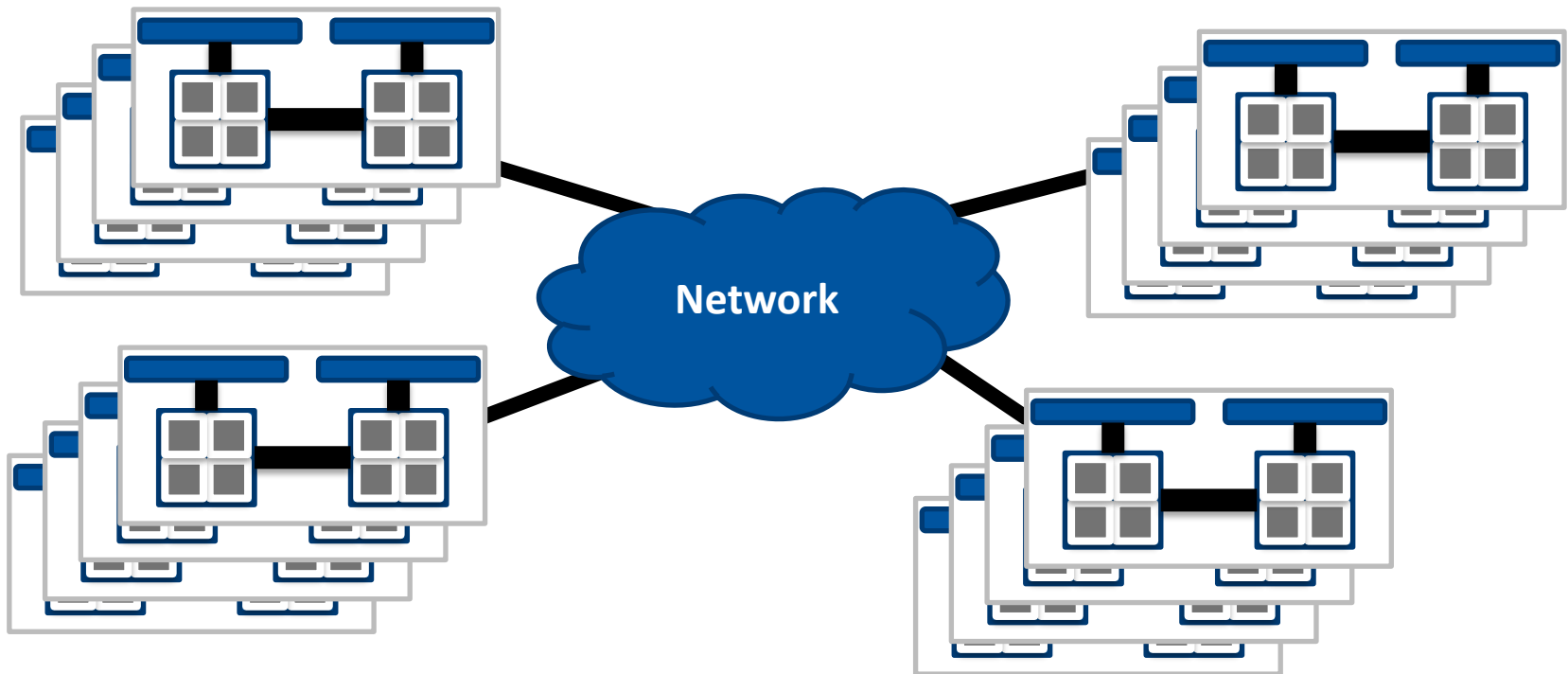
- **Programming concepts and models for**



# Programming for Clusters

## ■ Clusters

- HPC market is dominated by distributed memory multicomputers (clusters)
- Many nodes with no direct access to other nodes' memory



## ■ How to do useful work in parallel if source code is the same?

→ Each process receives a unique identifier

→ Multiple code paths based on the ID

```
int my_id = get_my_id();

if (my_id == id_1) {
    // Code for process id_1
}
else if (my_id == id_2) {
    // Code for process id_2
}
else {
    // Code for other processes
}
```

## Serial program

```
a[0..9] = a[100..109];
```

## SPMD program

→ process 0: **aa** holds **a[0..9]**

```
array aa[10];  
  
if (my_id == 0) {  
    recv(aa, 10);  
}  
else if (my_id == 10) {  
    send(aa, 0);  
}
```

process 10: **aa** holds **a[100..109]**

```
array aa[10];  
  
if (my_id == 0) {  
    recv(aa, 10);  
}  
else if (my_id == 10) {  
    send(aa, 0);  
}
```

## ■ C

```
#include <stdio.h>
1 #include <mpi.h>
int main(int argc, char **argv) {
    int rank, nprocs;

2 MPI_Init(&argc, &argv);

3 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

4 printf("Hello, MPI! I am %d of %d\n",
        rank, nprocs);

5 MPI_Finalize();
  return 0;
}
```

- 1 Header file inclusion – makes available prototypes of all MPI functions
- 2 MPI library initialisation – must be called before other MPI operations are called
- 3 MPI operations – more on that later
- 4 Text output – MPI programs also can print to the standard output
- 5 MPI library clean-up – no other MPI calls after this one allowed

## ■ How many processes are there in a given communicator?

```
MPI_Comm_size (MPI_Comm comm, int *size)
```

- Returns the total number of MPI processes when called on `MPI_COMM_WORLD`
- Returns 1 when called on `MPI_COMM_SELF`

## ■ What is the rank of the calling process in a given communicator?

```
MPI_Comm_rank (MPI_Comm comm, int *rank)
```

- Returned rank will differ in each calling process given the same communicator
- Ranks values are in `[0, size-1]` (always `0` for `MPI_COMM_SELF`)



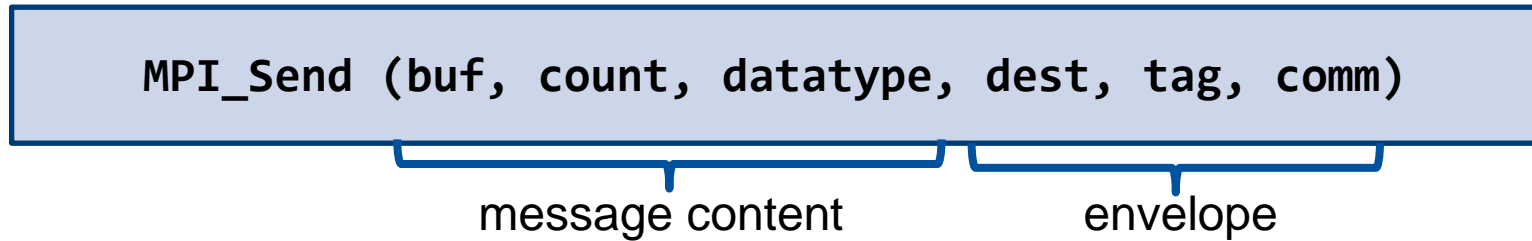
- **MPI passes data around in the form of messages**
- **Two components**
  - Message content (user data)
  - Envelope



Field	Meaning
Sender rank	Who sent the message
Receiver rank	To whom the message is addressed to
Tag	Additional message identifier
Communicator	Communication context

- **MPI retains the logical order in which messages between any two ranks are sent (FIFO)**
  - But the receiver have the option to peek further down the queue

- Messages are sent using the `MPI_Send` family of operations

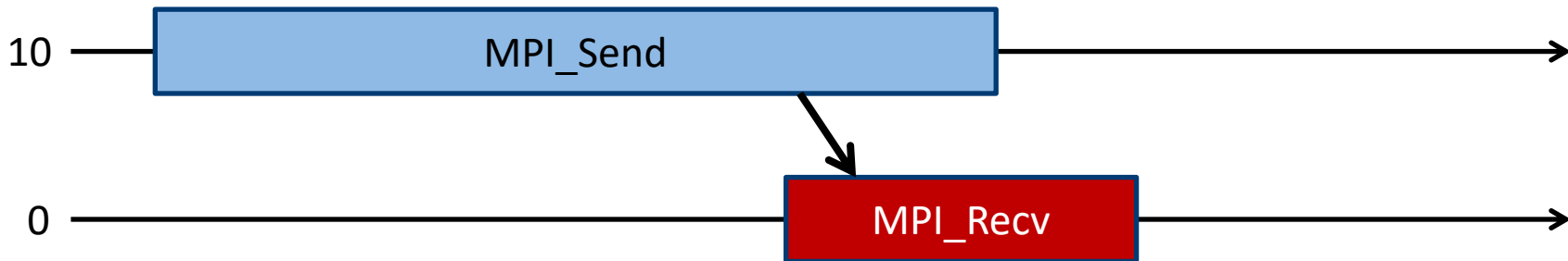


Parameter	Meaning
<code>buf</code>	Location of data in memory
<code>count</code>	Number of consecutive data elements to send
<code>datatype</code>	MPI data type handle
<code>dest</code>	Rank of the receiver
<code>tag</code>	Message tag
<code>comm</code>	Communicator handle

- The MPI API is built on the idea that data structures are array-like
  - No fancy C++ objects supported

## ■ Our earlier SPMD example written in MPI

```
int aa[10];  
MPI_Status status;  
  
if (rank == 0) {  
    MPI_Recv(aa, 10, MPI_INT, 10, 0, MPI_COMM_WORLD, &status);  
}  
else if (rank == 10) {  
    MPI_Send(aa, 10, MPI_INT, 0, 0, MPI_COMM_WORLD);  
}
```



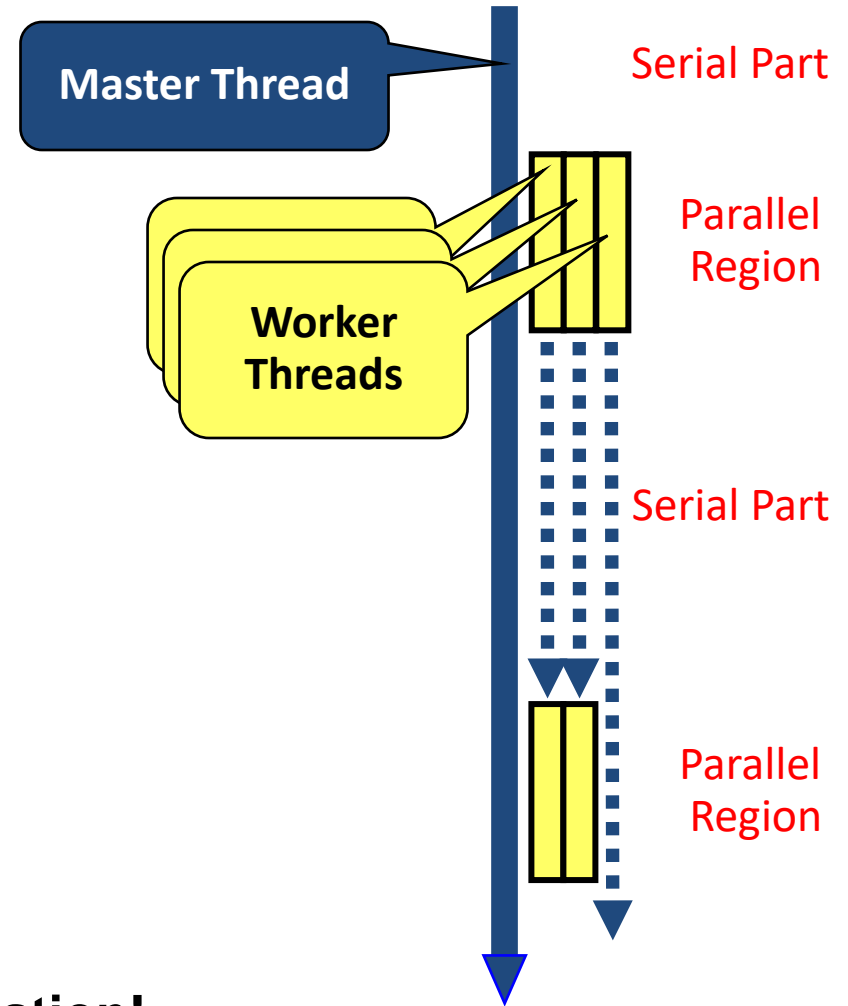
- 1. Type the code on slide 7 into a text file named `hello.c`
- 2. Compile:
  - C: `mpicc -o hello.exe hello.c`
  - Fortran: `mpif90 -o hello.exe hello.f90`
- 3. Run:
  - `mpiexec -n 4 hello.exe`
- On CLAIX we recommend the RWTH specific environment variables:
  - `$MPICC -o hello.exe hello.c`
  - `$MPIEXEC $FLAGS_MPI_BATCH hello.exe`



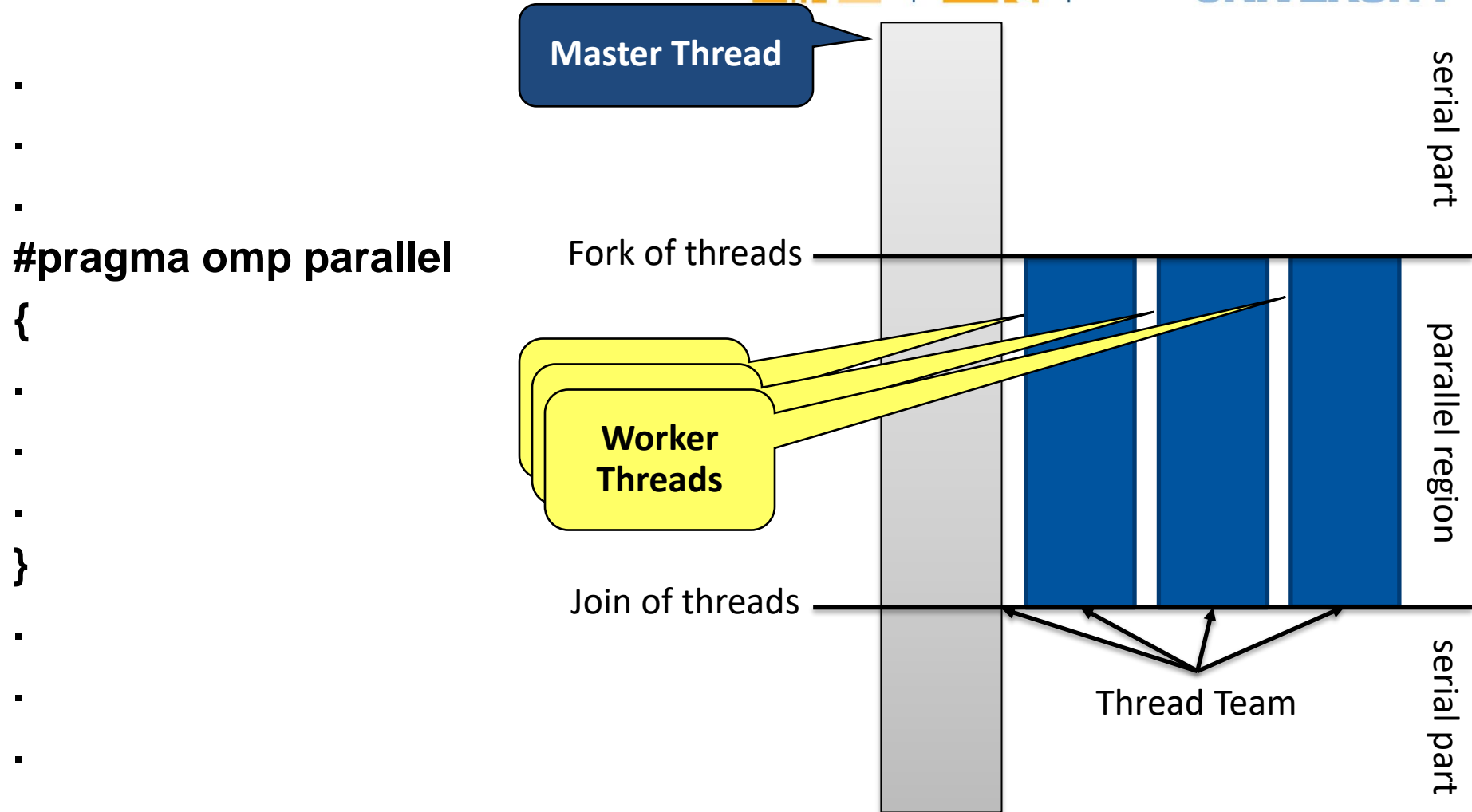
# Programming for Multi-Core Nodes

# OpenMP Execution Model

- OpenMP programs start with just one thread: The *Master*.
- *Worker* threads are spawned at *Parallel Regions*, together with the Master they form the *Team* of threads.
- In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.
- Concept: *Fork-Join*.
- Allows for an incremental parallelization!

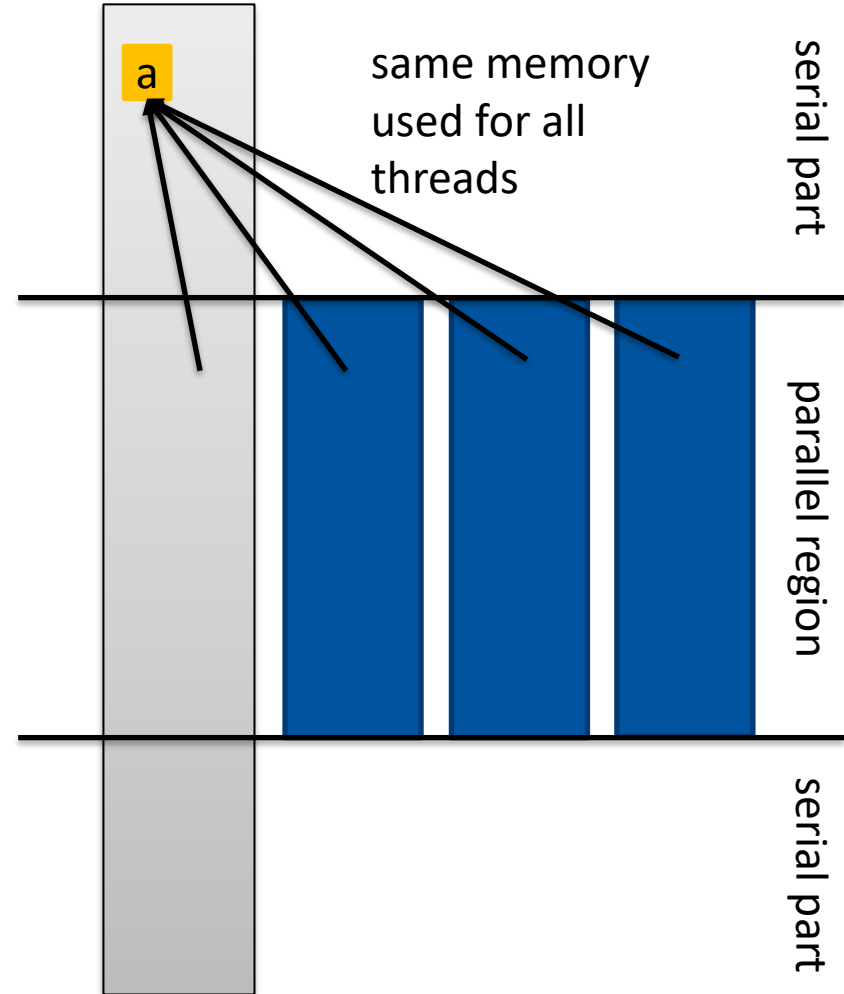


# Fork-Join Execution Model



# Data Sharing Attributes (1/3)

```
int a;  
.  
.  
#pragma omp parallel shared(a)  
{  
.  
.  
.  
}  
.  
.  
.
```

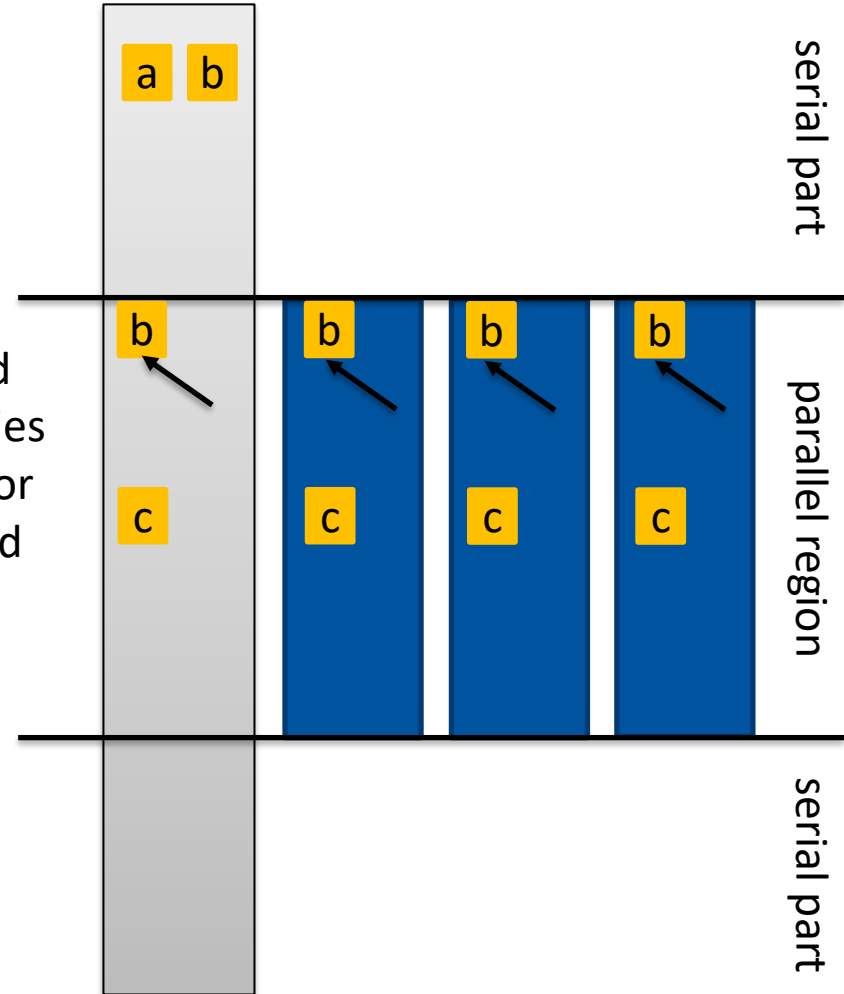




# Data Sharing Attributes (2/3)

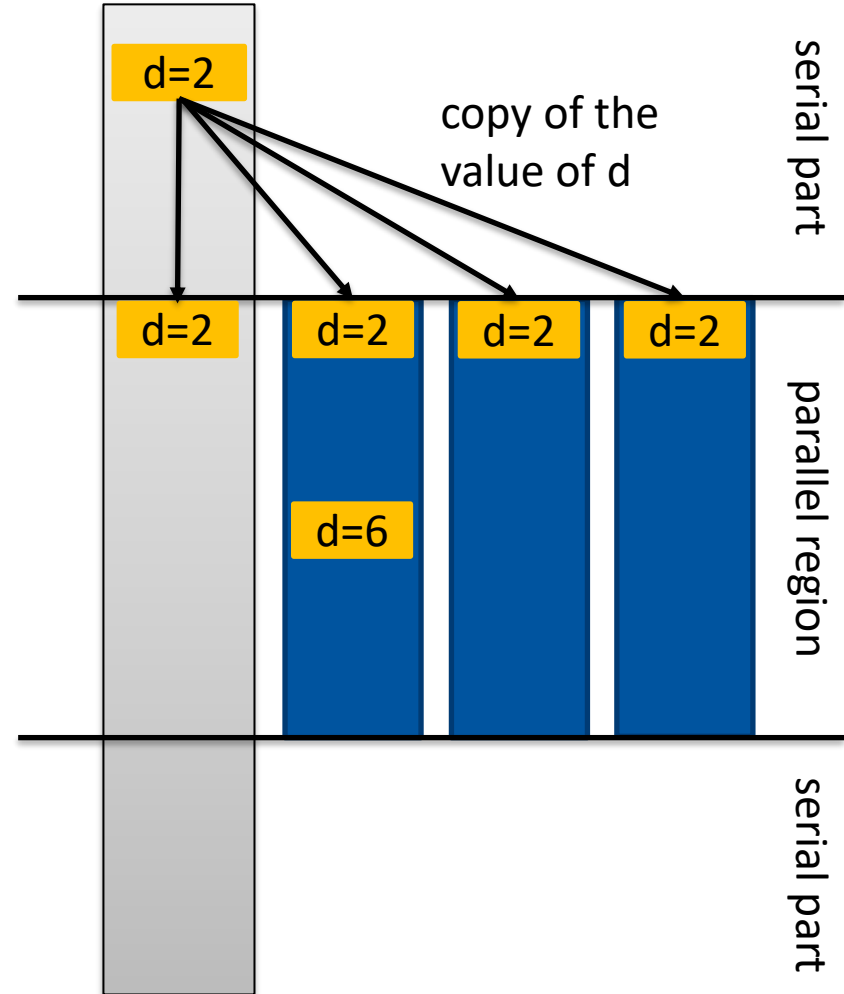
```
int a,b;  
.  
#pragma omp parallel shared(a) //  
  private(b)  
{  
.  
int c;  
.  
}  
.  
.  
.
```

uninitialized  
private copies  
of b and c for  
every thread



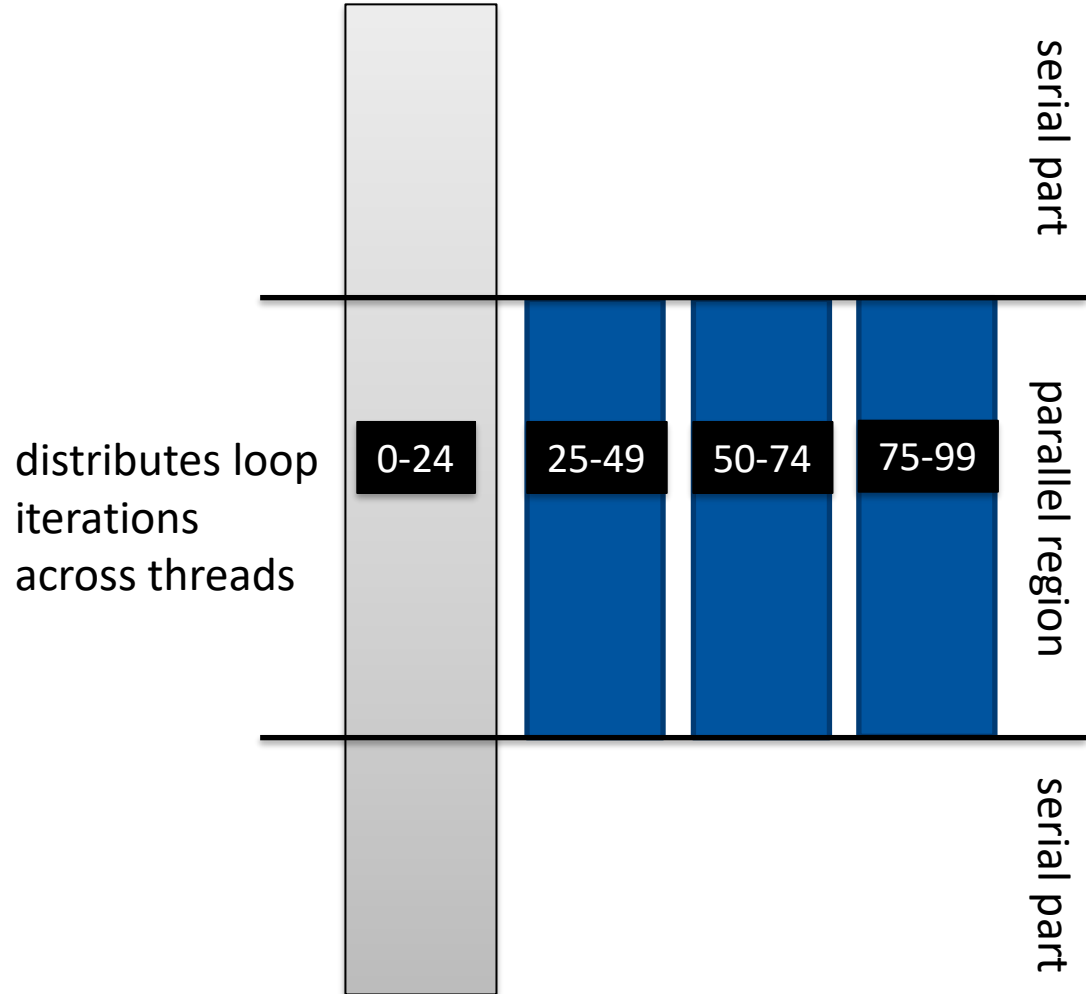
# Data Sharing Attributes (3/3)

```
int d=2;  
.  
.  
#pragma omp parallel firstprivate(d)  
{  
#pragma omp single  
{d=6;}  
.  
}  
.  
.  
.
```



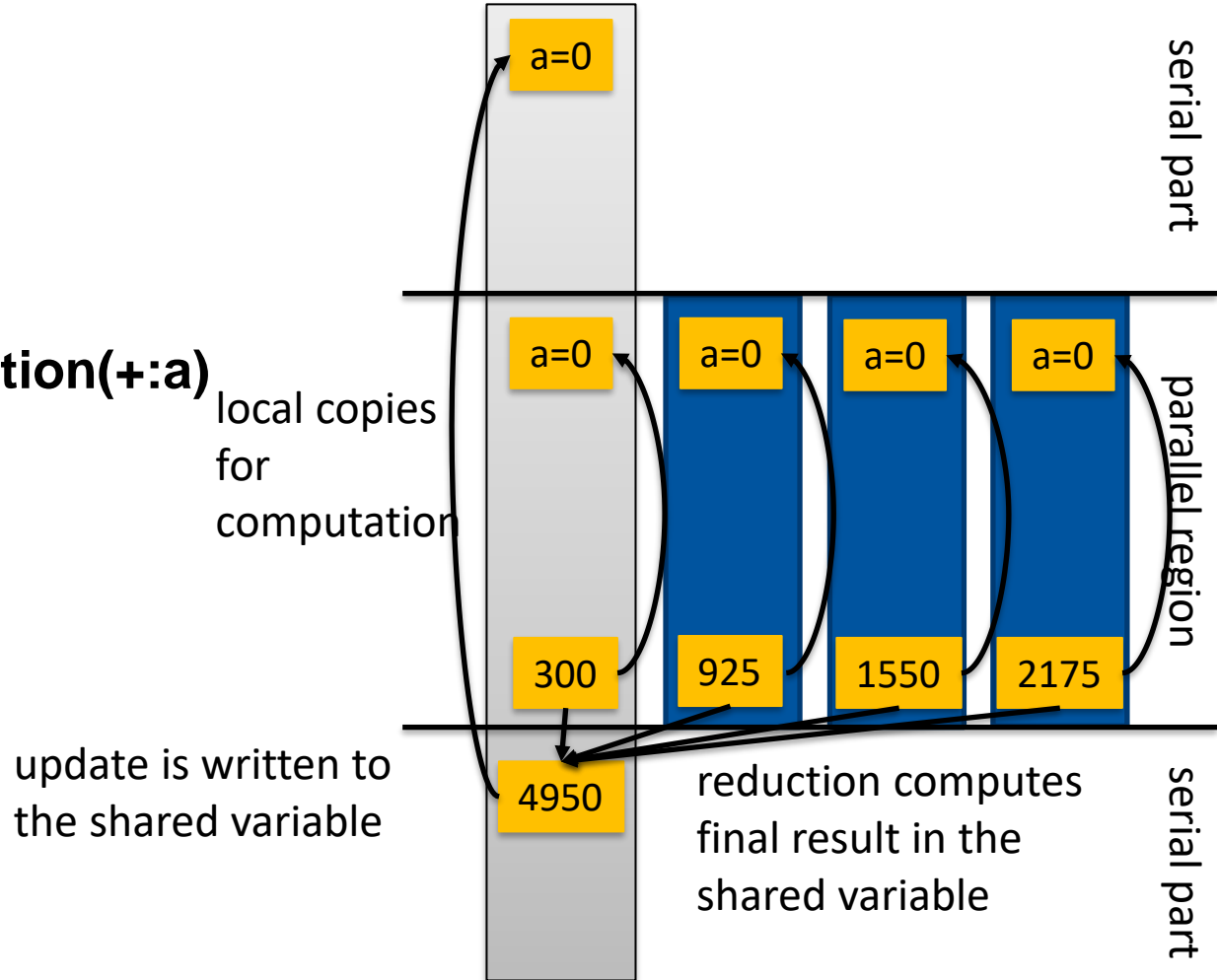
# For Worksharing

```
.....  
#pragma omp parallel  
#pragma omp for  
for (int i=0; i<100; i++){  
.....  
.....  
.....  
}  
.....  
.....  
.....
```



# Reduction Operations

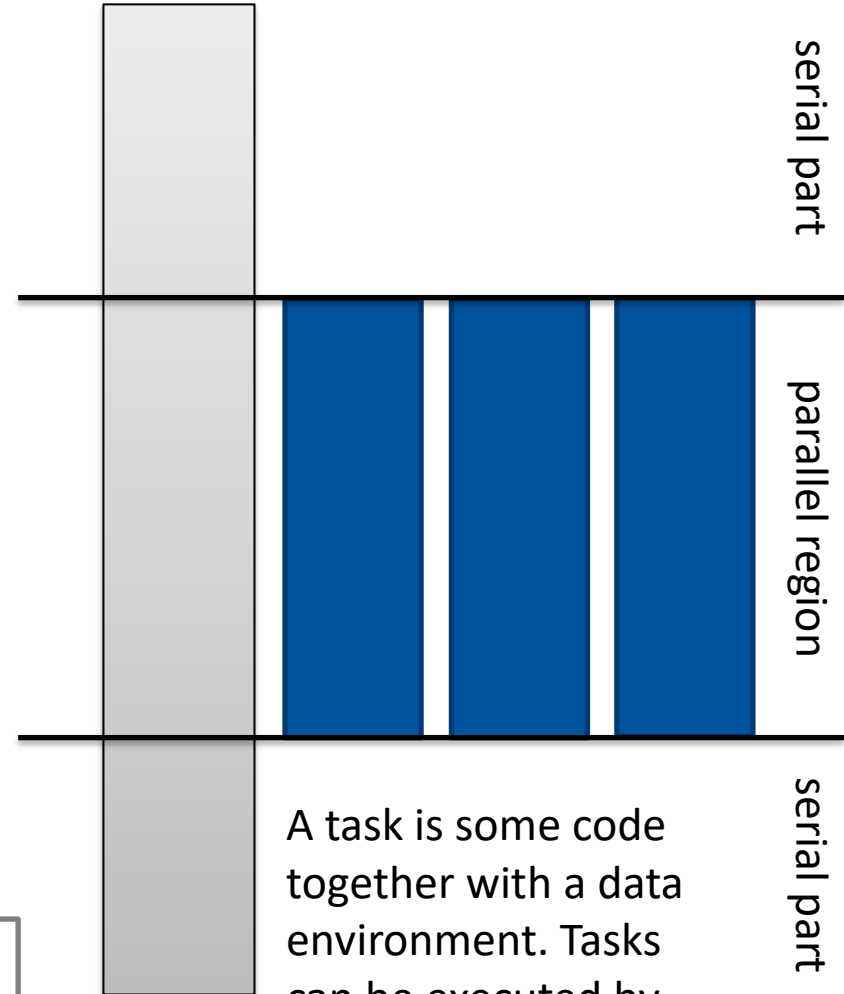
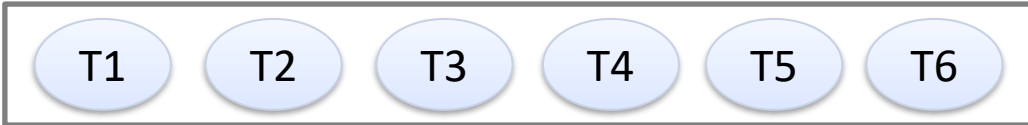
```
int a=0;  
.  
.  
#pragma omp parallel  
#pragma omp for reduction(+:a)  
for (int i=0; i<100; i++)  
{  
    a+=i;  
}
```



# Tasks

```
#pragma omp parallel
#pragma omp single
while (work()){
    #pragma omp task
    {
        ...
    }
} // implicit barrier here
```

Taskqueue



A task is some code together with a data environment. Tasks can be executed by any thread in any order.

- **Compilation: add `-fopenmp` flag to compiler (and linker)**

→ In our environment better use `$FLAGS_OPENMP`

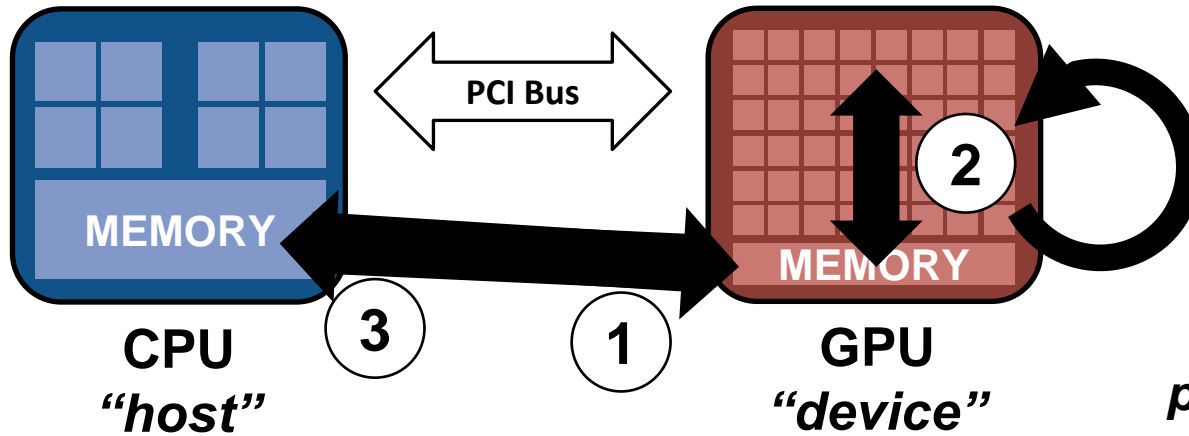
- **From within a shell, global setting of the number of threads:**

```
export OMP_NUM_THREADS=4
./program
```

- **From within a shell, one-time setting of the number of threads:**

```
OMP_NUM_THREADS=4 ./program
```

# Programming for Accelerators



We refer to “discrete GPUs” here.

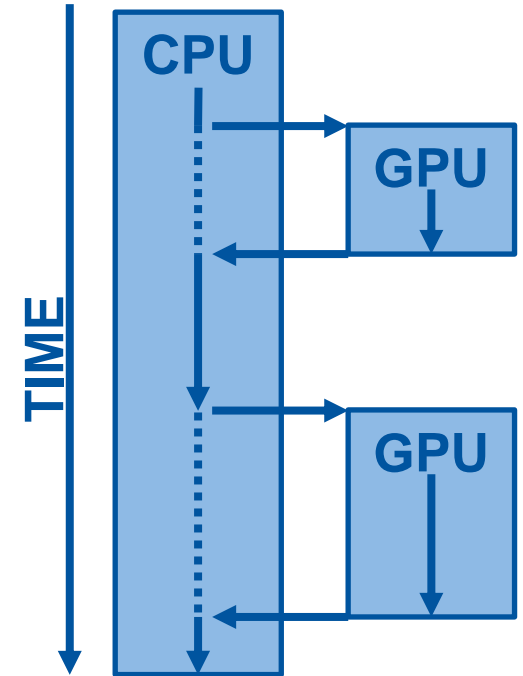
## Weak memory model

- Host + device memory = separate entities
- No coherence between host + device
- **Data transfers** needed

## Host-directed execution model

- Copy input data from CPU mem. to device mem.
- Execute the device program
- Copy results from device mem. to CPU mem.

*processing flow (simplified)*





## ■ **CUDA (Compute Unified Device Architecture)**

→ C/C++ (NVIDIA): architecture + programming language, NVIDIA GPUs

→ Fortran (PGI): NVIDIA's CUDA for Fortran, NVIDIA GPUs

## ■ **OpenCL**

→ C (Khronos Group): open standard, portable, CPU/GPU/...

## ■ **OpenACC**

→ C/Fortran (PGI): Directive-based accelerator programming, industry standard published in Nov. 2011 (NVIDIA GPUs)

## ■ **OpenMP**

→ C/C++, Fortran: Directive-based programming for hosts and accelerators, standard, portable, published in July 2013

■ ...

# Example SAXPY – CPU



```
void saxpyCPU(int n, float a, float *x, float *y) {  
    for (int i = 0; i < n; ++i)  
        y[i] = a*x[i] + y[i];  
}
```

SAXPY = Single-precision real Alpha X Plus Y

$$\vec{y} = \alpha \cdot \vec{x} + \vec{y}$$

```
int main(int argc, const char* argv[]) {  
    int n = 10240; float a = 2.0f;  
    float *x = (float*) malloc(n * sizeof(float));  
    float *y = (float*) malloc(n * sizeof(float));  
  
    // Initialize x, y  
    for(int i=0; i<n; ++i){  
        x[i]=i;  
        y[i]=5.0*i-1.0;  
    }  
  
    // Invoke serial SAXPY kernel  
    saxpyCPU(n, a, x, y);  
  
    free(x); free(y);  
    return 0;  
}
```

```
void saxpyOpenACC(int n, float a, float *x, float *y) {  
#pragma acc parallel loop copy(y[0:n]) copyin(x[0:n])  
    for (int i = 0; i < n; ++i)  
        y[i] = a*x[i] + y[i];  
}  
  
int main(int argc, const char* argv[]) {  
    int n = 10240; float a = 2.0f;  
    float *x = (float*) malloc(n * sizeof(float));  
    float *y = (float*) malloc(n * sizeof(float));  
  
    // Initialize x, y  
    for(int i=0; i<n; ++i){  
        x[i]=i;  
        y[i]=5.0*i-1.0;  
    }  
  
    // Invoke serial SAXPY kernel  
    saxpyOpenACC(n, a, x, y);  
  
    free(x); free(y);  
    return 0;  
}
```

# Example SAXPY – CUDA



```
__global__ void saxpy_parallel(int n,
float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x +
threadIdx.x;
    if (i < n){
        y[i] = a*x[i] + y[i];
    }
}

int main(int argc, char* argv[]) {
    int n = 10240; float a = 2.0f;
    float* h_x,*h_y; // Pointer to CPU memory
    h_x = (float*) malloc(n* sizeof(float));
    h_y = (float*) malloc(n* sizeof(float));
    // Initialize h_x, h_y
    for(int i=0; i<n; ++i){
        h_x[i]=i;
        h_y[i]=5.0*i-1.0;
    }
}
```

**1. Allocate data on GPU + transfer data**

```
cudaMemcpy(d_x, h_x, n * sizeof(float),
cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, n * sizeof(float),
cudaMemcpyHostToDevice);
```

```
// Invoke parallel SAXPY kernel
dim3 threadsPerBlock(128);
dim3 b...
saxpy_parallel<<<blocksPerGrid,
threadsPerBlock>>>(n, 2.0, d_x, d_y);
```

**2. Launch kernel**

```
cudaMemcpy(h_y, d_y, n * sizeof(float),
cudaMemcpyDeviceToHost);
cudaFree(d_x); cudaFree(d_y);
```

**3. Transfer data to CPU + free data on GPU**

```
free(h_x); free(h_y);
return 0;
}
```

# Putting it all together

- (Hierarchical) mixing of different programming paradigms

