

# Parallelization programming

Jan Thorbecke

# Contents

- Introduction
- Programming
  - OpenMP
  - MPI
- Issues with parallel computing
- Running programs
- Examples

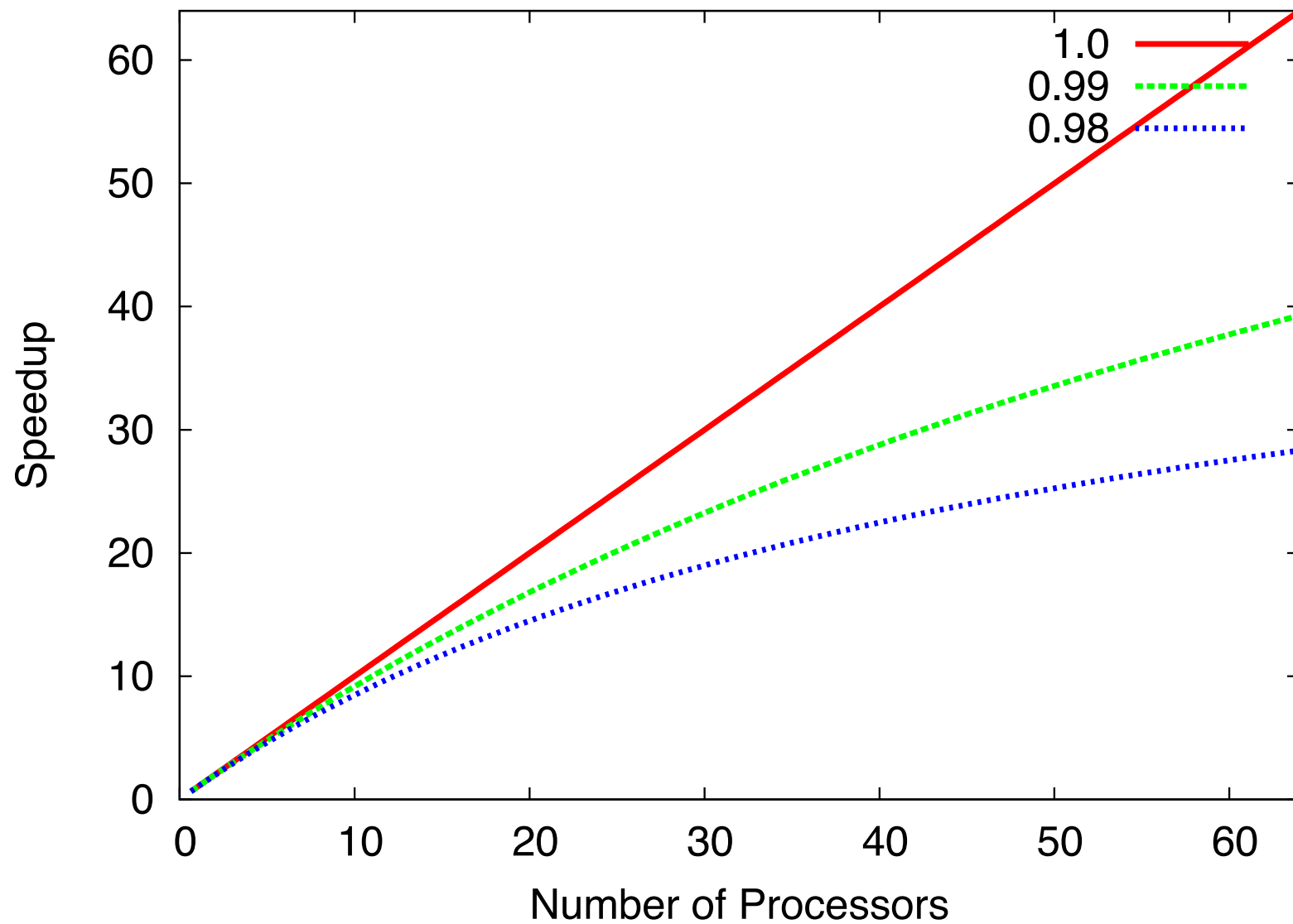
# Amdahl's Law

- Describes the relation between the parallel portion of your code and the expected speedup

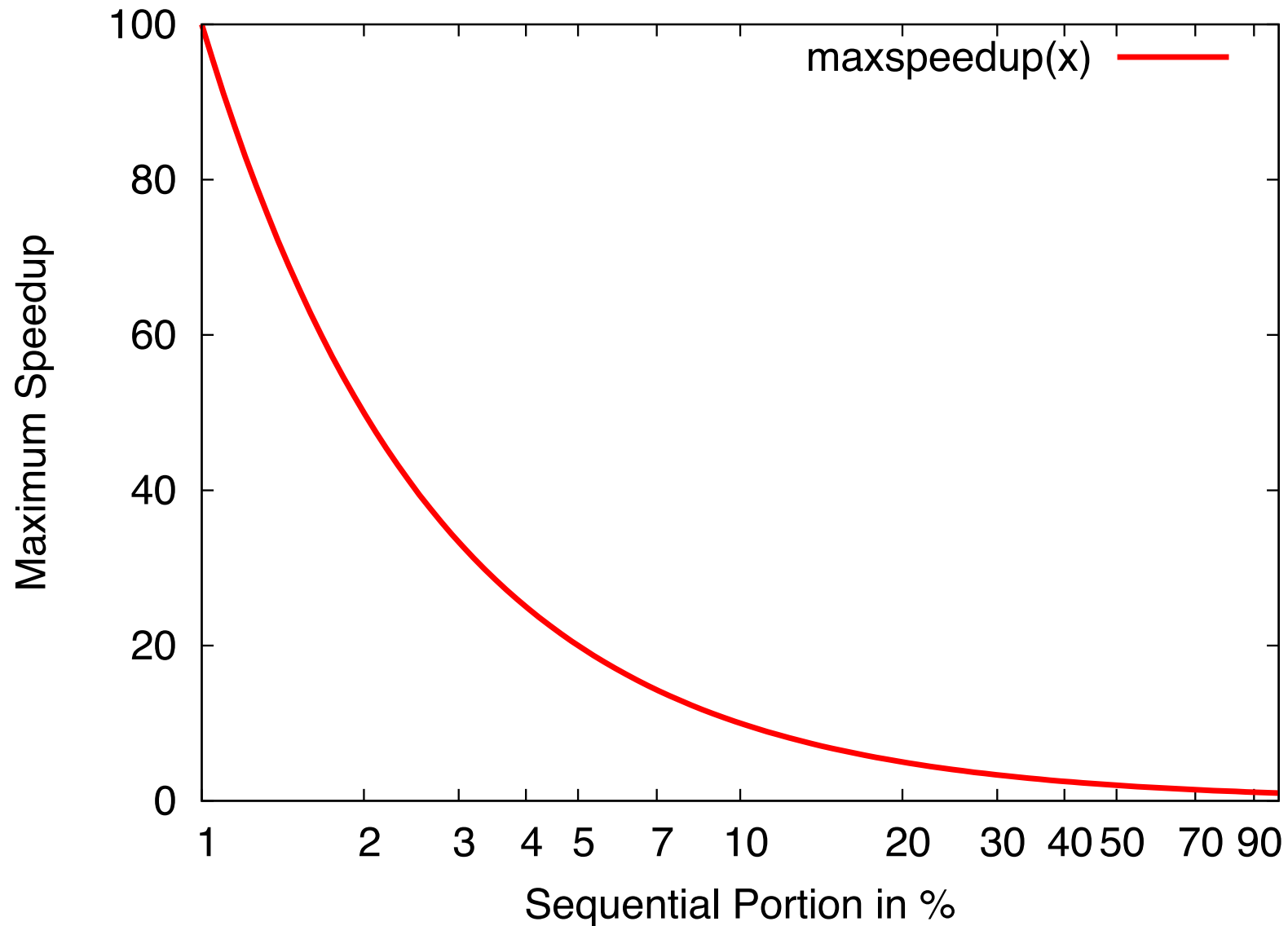
$$speedup = \frac{1}{(1 - P) + \frac{P}{N}}$$

- $P$  = parallel portion
- $N$  = number of processors used in parallel part
- $P/N$  is the ideal parallel speed-up, it will always be less

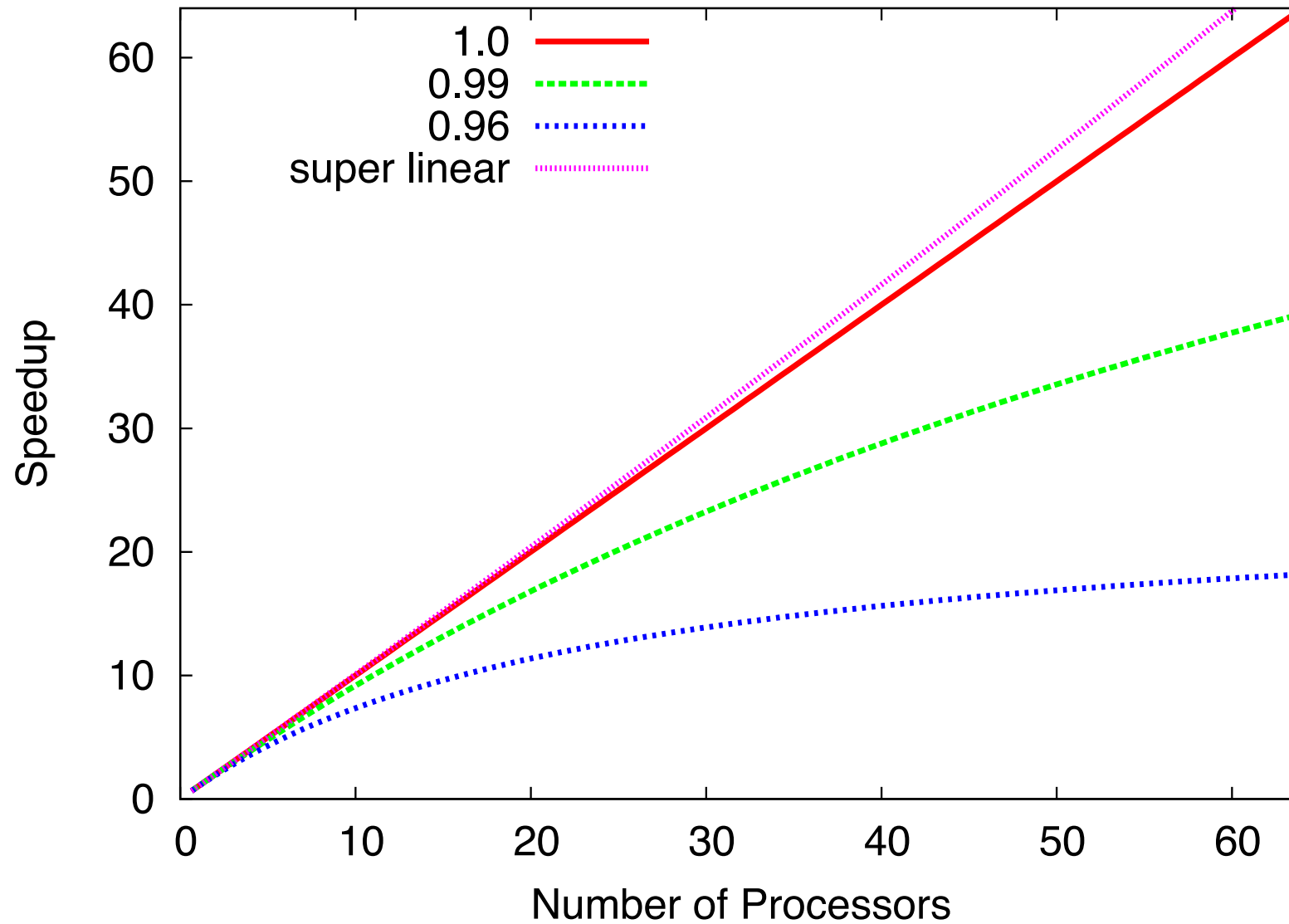
# Amdahl's Law



# Amdahl's Law



# Super Linear speed up



# Concurrency and Parallelism

- Concurrency and parallelism are often used synonymously.

**Concurrency:** The independence of parts of an algorithm  
(= independent of each other).

**Parallelism** (also parallel execution): Two or more parts of a program are executed at the same moment in time.

Concurrency is a necessary prerequisite for parallel execution  
for parallel execution

but

Parallel execution is only one possible consequence of  
concurrency.

# Concurrency vs. Parallelism

- Concurrency: two or more threads are in progress at the same time:



- Parallelism: two or more threads are executing at the same time



Multiple cores needed

# Learning

Classical CPU's are sequential

There is an enormous **sequential** programming knowledge build into compilers and know by most programmers.

**Parallel** Programming is requiring new skills and new tools.

Start to parallelise simple problems and keep on learning along the way to complex real-world problems.

# Recognizing Sequential Processes

- Time is inherently sequential

Dynamics and real-time, event driven applications are often difficult to parallelize effectively

time stepping modeling code

Many games fall into this category

- Iterative processes

The results of an iteration depend on the preceding iteration

conjugate gradient methods

Audio encoders

# Parallel Programming Models

- Parallel programming models exist as an abstraction above hardware and memory architectures.
- Which model to use is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.

# Parallel Programming Models

- Shared Memory
  - tasks share a common address space, which they read and write asynchronously.
- Threads (functional)
  - a single process can have multiple, concurrent execution paths.  
Example implementations: POSIX threads & OpenMP
- Message Passing
  - tasks exchange data through communications by sending and receiving messages. Example: MPI-2 specification.
- Data Parallel languages
  - tasks perform the same operation on their partition of work.  
Example: Co-array Fortran (CAF), Unified Parallel C (UPC), Chapel
- Hybrid

# Programming Models

User

New to parallel programming  
Experienced programmer

Programming Model

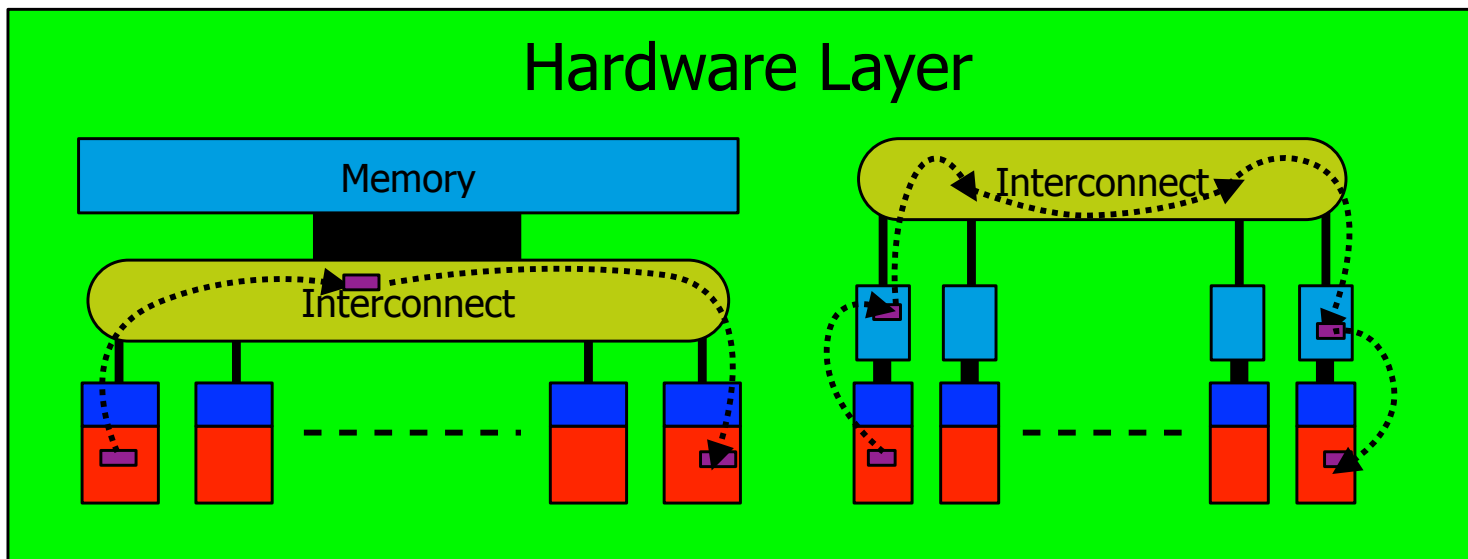
Shared Memory  
Message passing  
Hybrid

System Software

Operating system  
compilers

Hardware Layer

Distributed Memory  
Shared Memory



# Parallel Programming Concepts

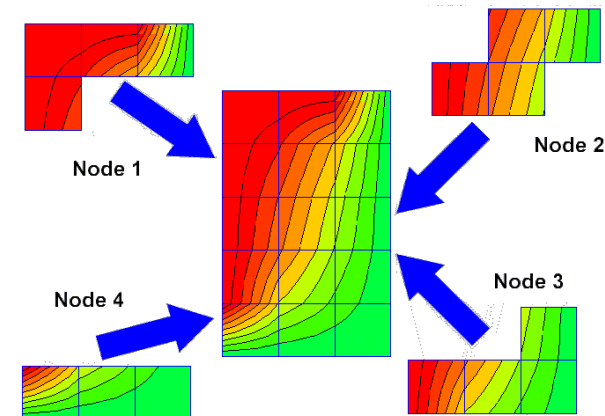
- Work distribution
  - Which parallel task is doing what?
  - Which data is used by which task?
- Synchronization
  - Do the different parallel tasks meet?
- Communication
  - Is communication between parallel parts needed?
- Load Balance
  - Does every task has the same amount of work?
  - Are all processors of the same speed?

# Distributing Work and/or Data

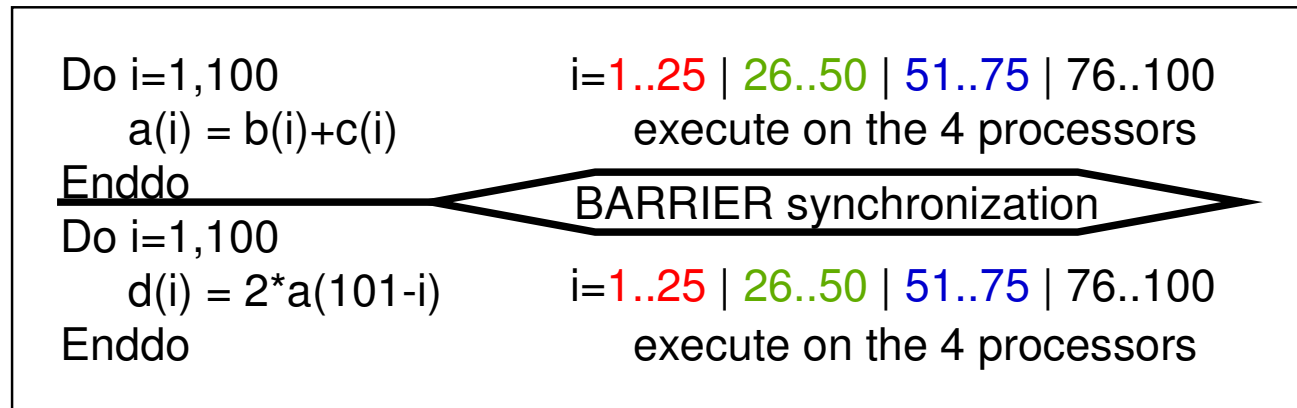
- Work decomposition
  - based on loop counter
- Data decomposition
  - all work for a local portion of the data is done by the local processor
- Domain decomposition
  - decomposition of work and data

```
do i=1,100  
  1: i=1,25  
  2: i=26,50  
  3: i=51,75  
  4: i=76,100
```

```
A(1:10,1:25)  
A(1:10,26:50)  
A(11:20,1:25)  
A(11:20,25:50)
```



# Synchronization



- Synchronization
  - causes overhead
  - idle time, when not all tasks are finished at the same time

# Communication

- communication is necessary on the boundaries

```
do i=2,99
  b(i) = b(i) + h*(a(i-1)-2*a(i)+a(i+1))
end do
```

e.g.  $b(26) = b(26) + h*(a(25) - 2*a(26) + a(27))$

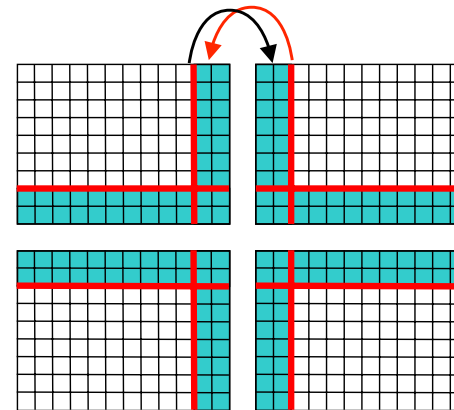
$A(1:25)$

$A(26:50)$

$A(51:75)$

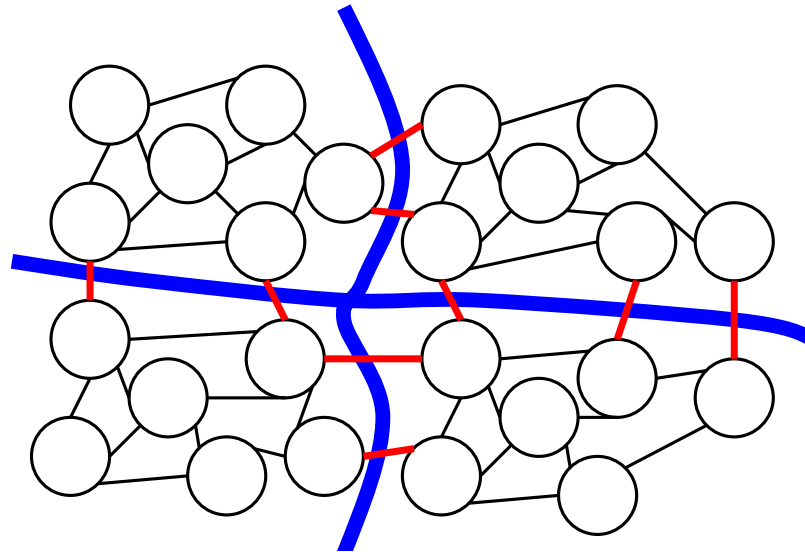
$A(76:100)$

- domain decomposition



# Load Imbalance

- Load imbalance is the time that some processors in the system are idle due to:
  - less parallelism than processors
  - unequal sized tasks together with too little parallelism
  - unequal processors

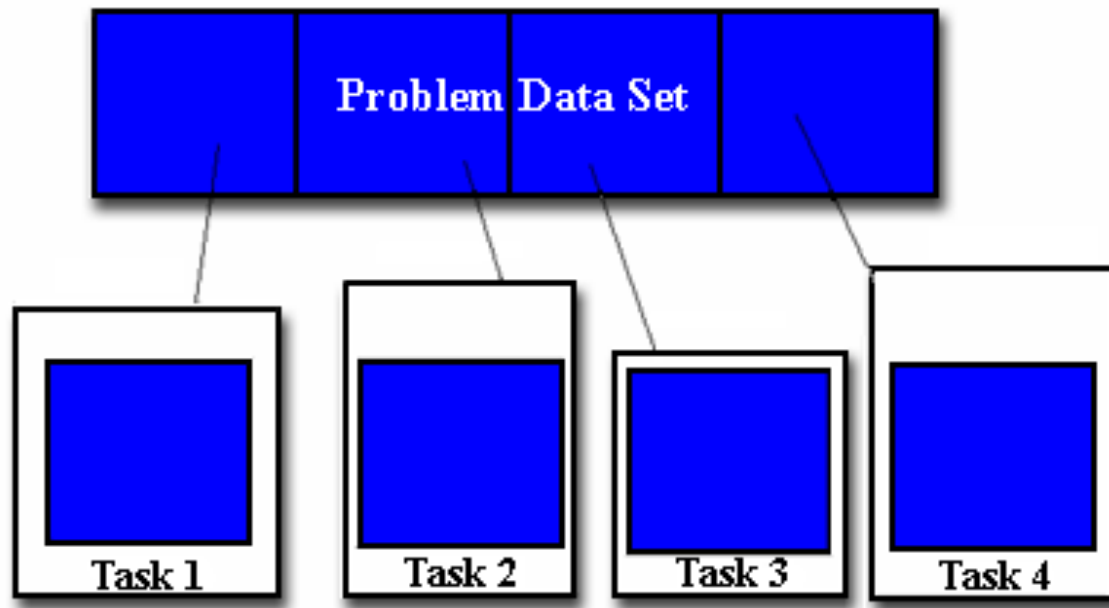


# Examples of work distribution

- Domain Decomposition
- Master Worker
- Task Decomposition

# Domain Decomposition

- First, decide how data elements should be divided among processors
- Second, decide which tasks each processor should be doing



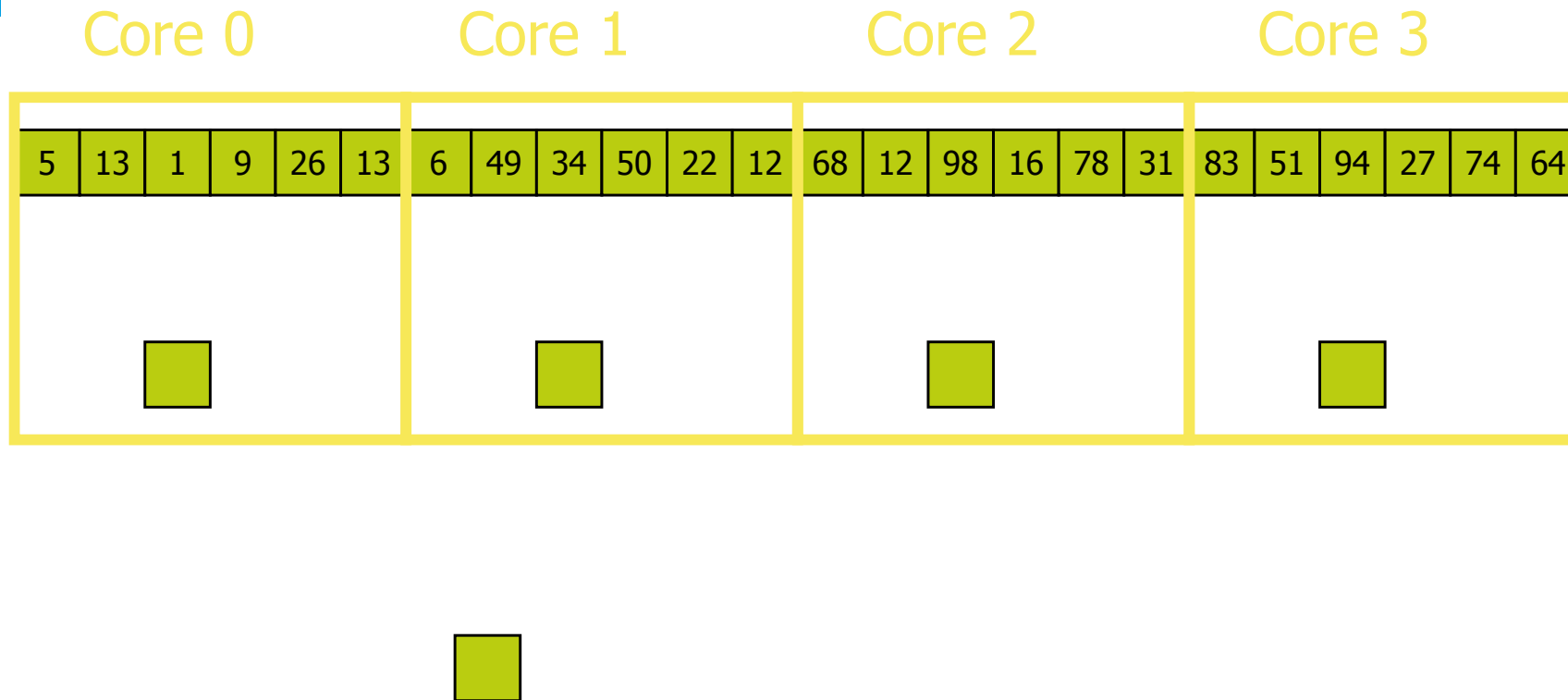
# Domain Decomposition

Find the largest element of an array

|   |    |   |   |    |    |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|---|---|----|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 5 | 13 | 1 | 9 | 26 | 13 | 6 | 49 | 34 | 50 | 22 | 12 | 68 | 12 | 98 | 16 | 78 | 31 | 83 | 51 | 94 | 27 | 74 | 64 |
|---|----|---|---|----|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

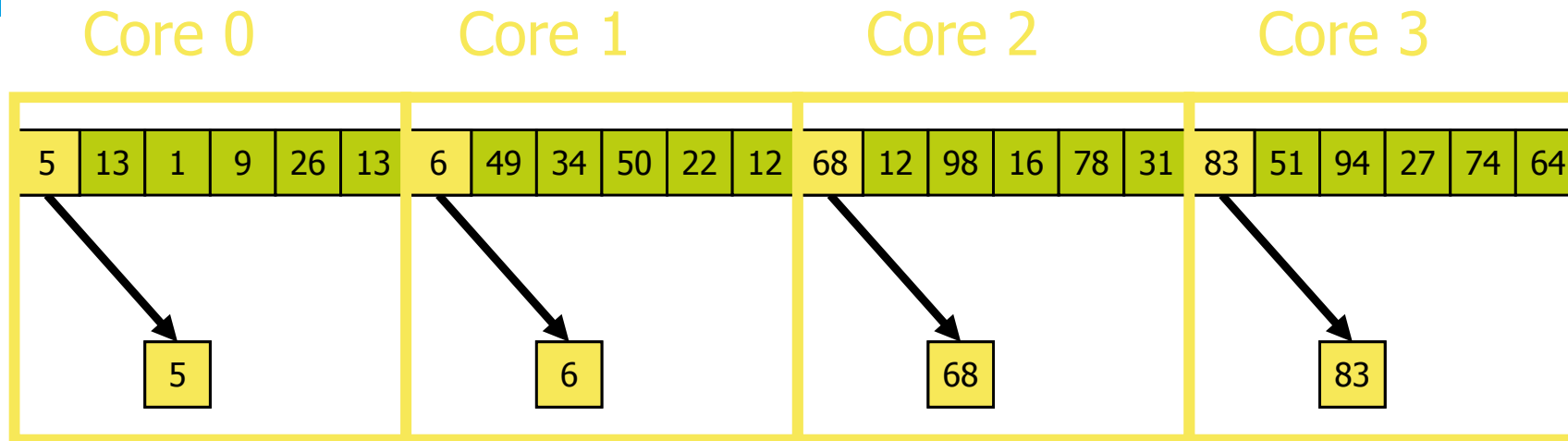
# Domain Decomposition

Find the largest element of an array



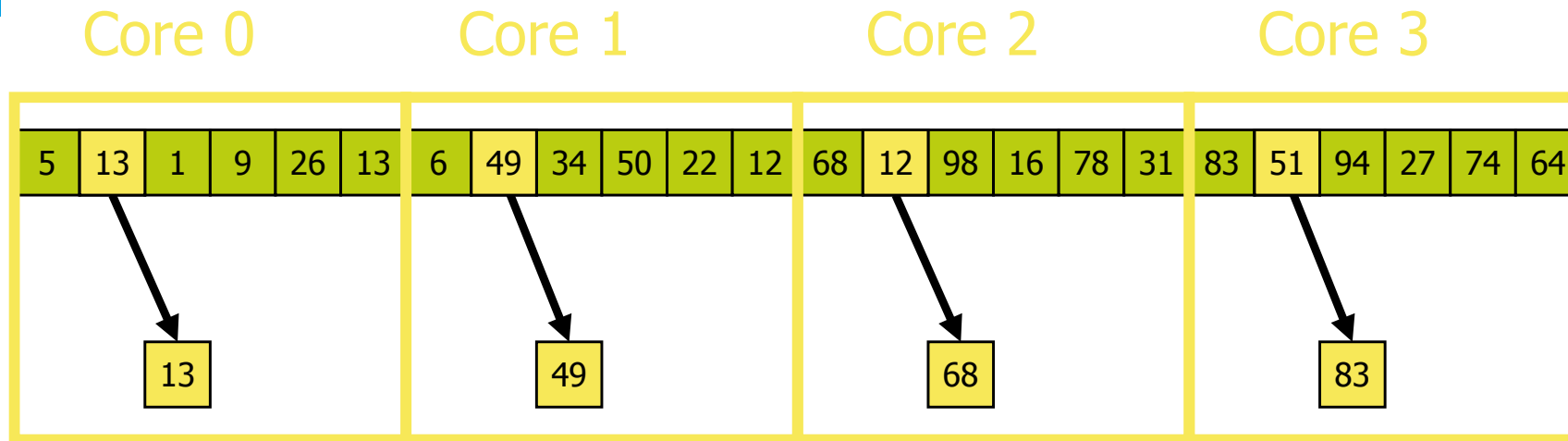
# Domain Decomposition

Find the largest element of an array



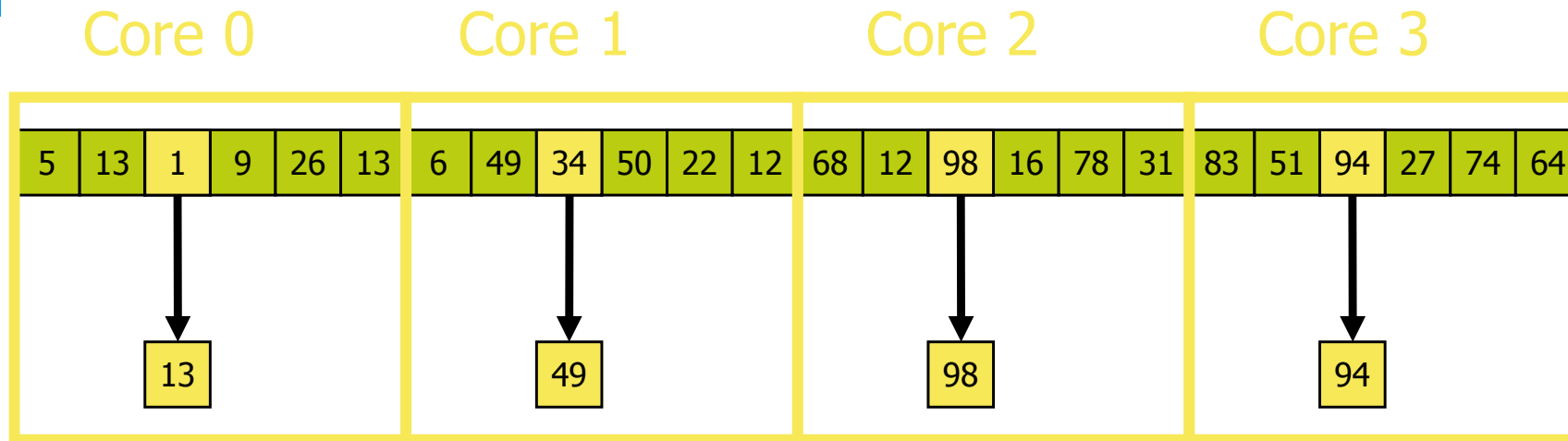
# Domain Decomposition

Find the largest element of an array



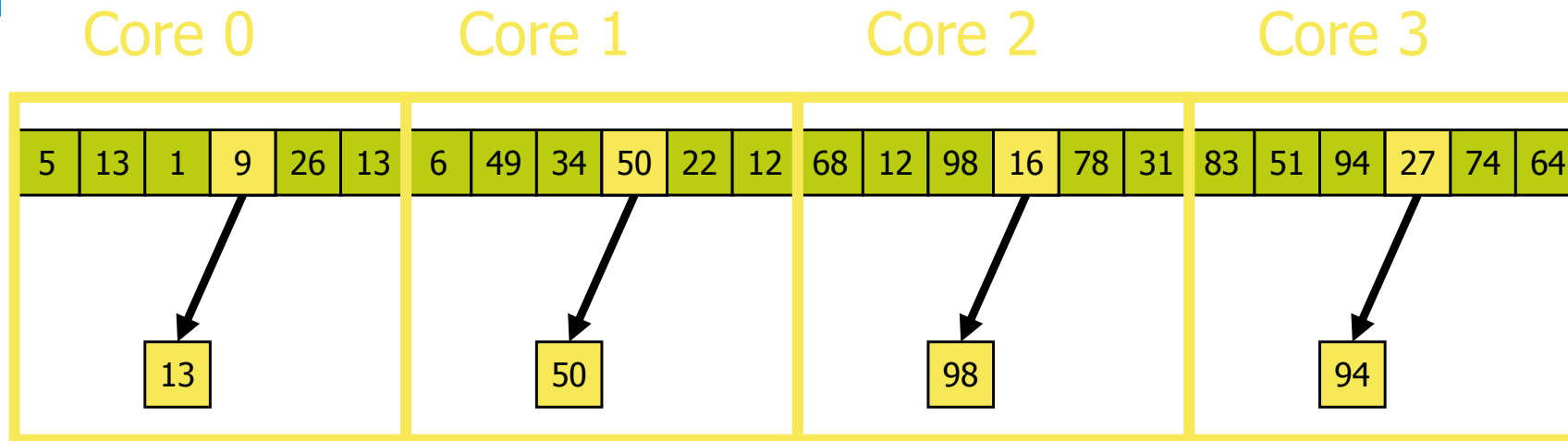
# Domain Decomposition

Find the largest element of an array



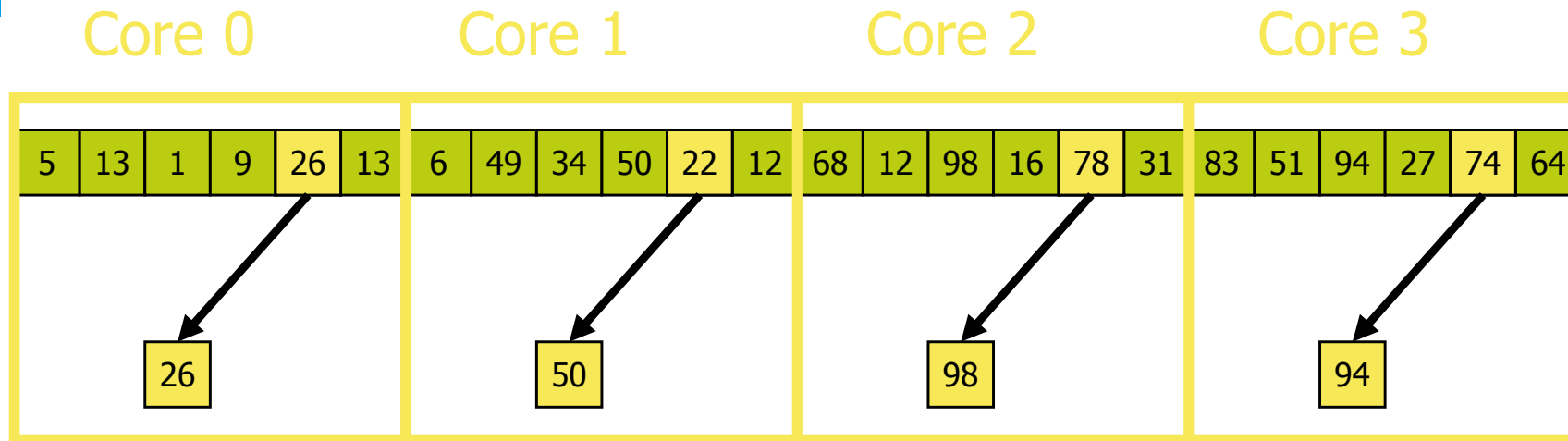
# Domain Decomposition

Find the largest element of an array



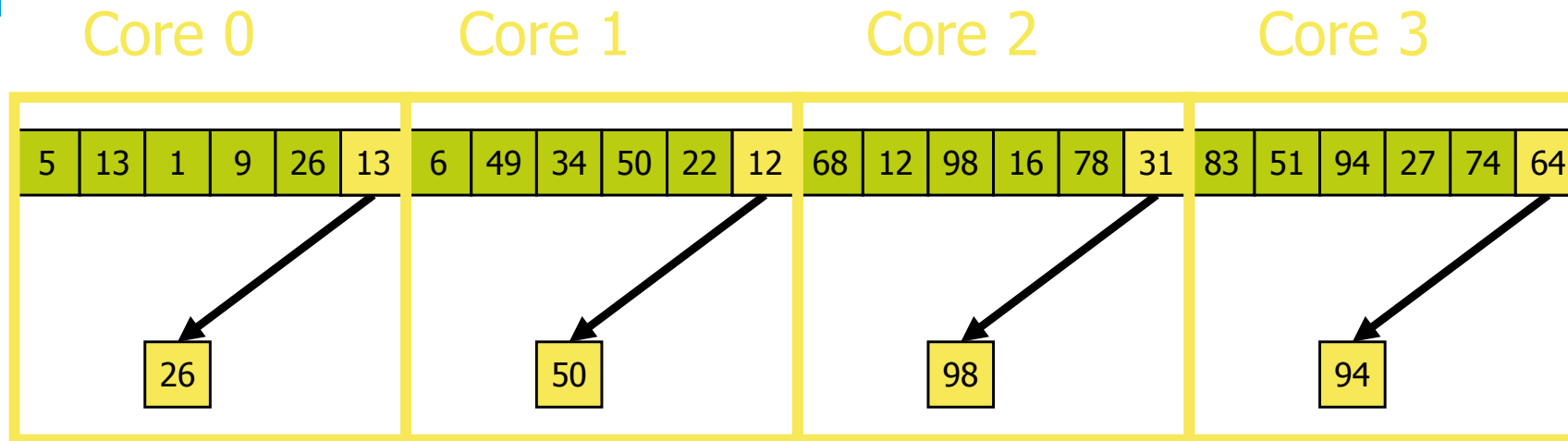
# Domain Decomposition

Find the largest element of an array



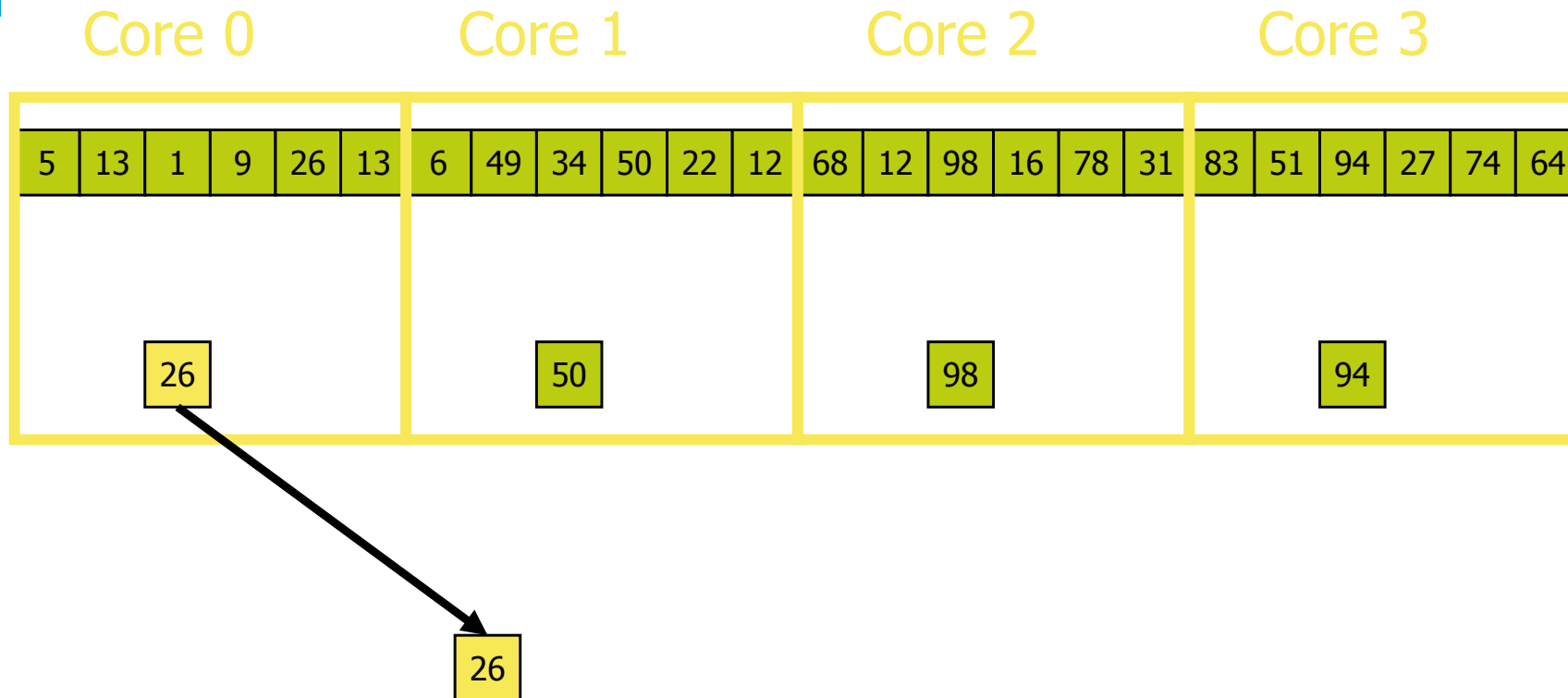
# Domain Decomposition

Find the largest element of an array



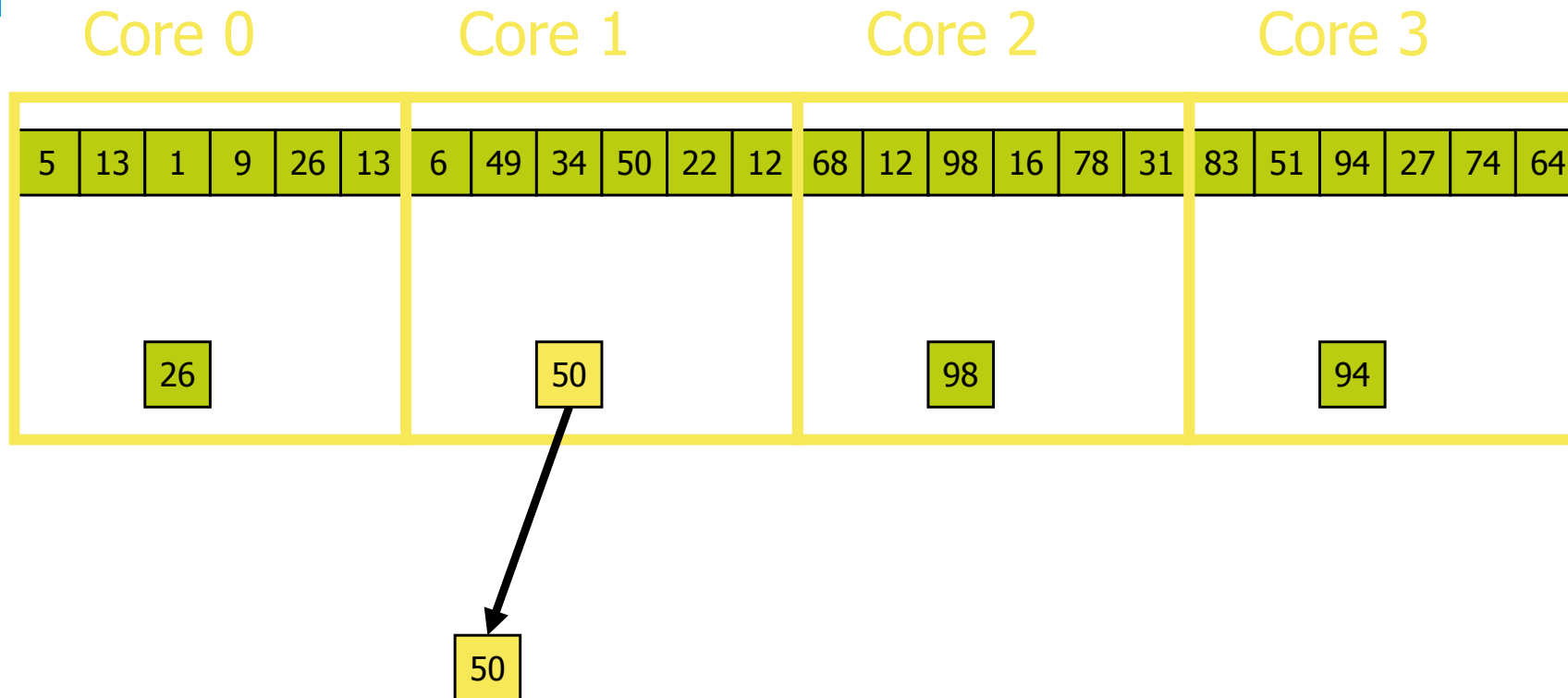
# Domain Decomposition

Find the largest element of an array



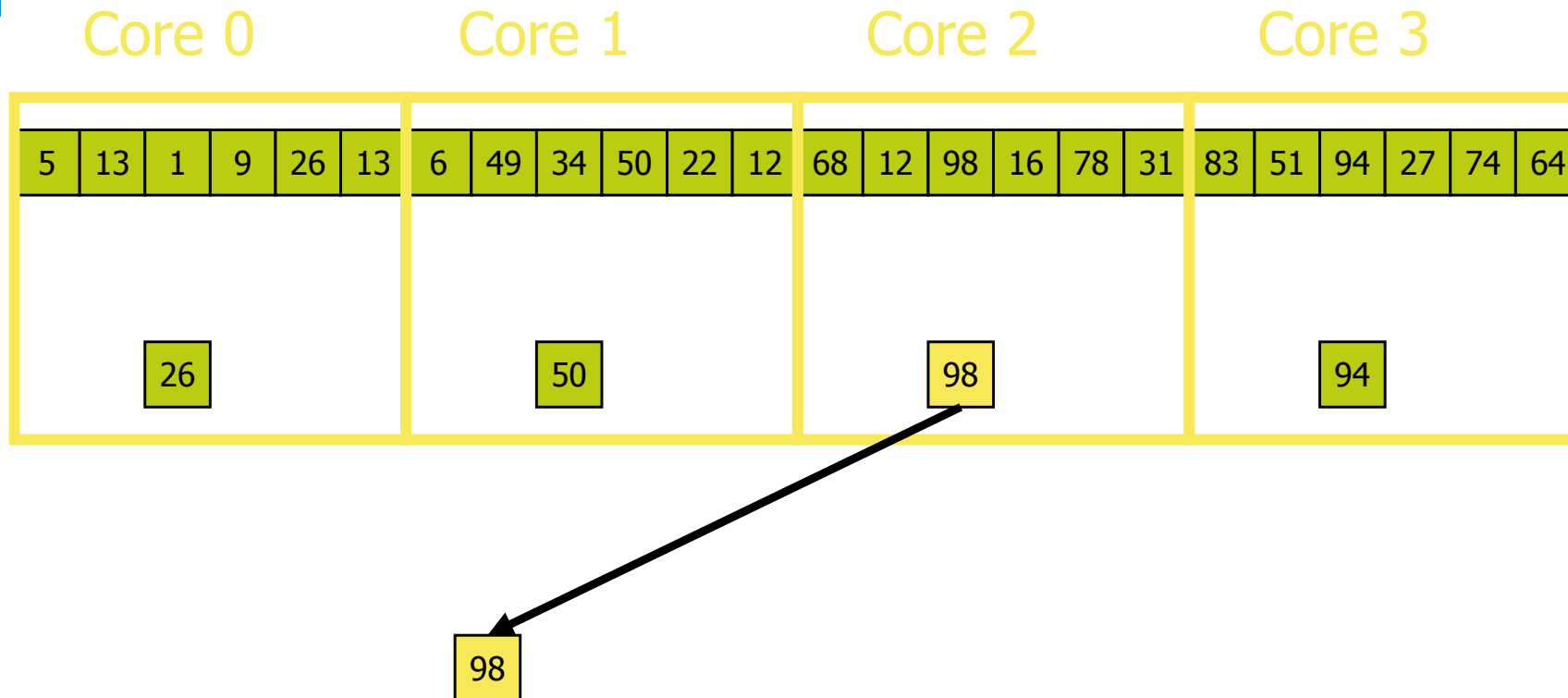
# Domain Decomposition

Find the largest element of an array



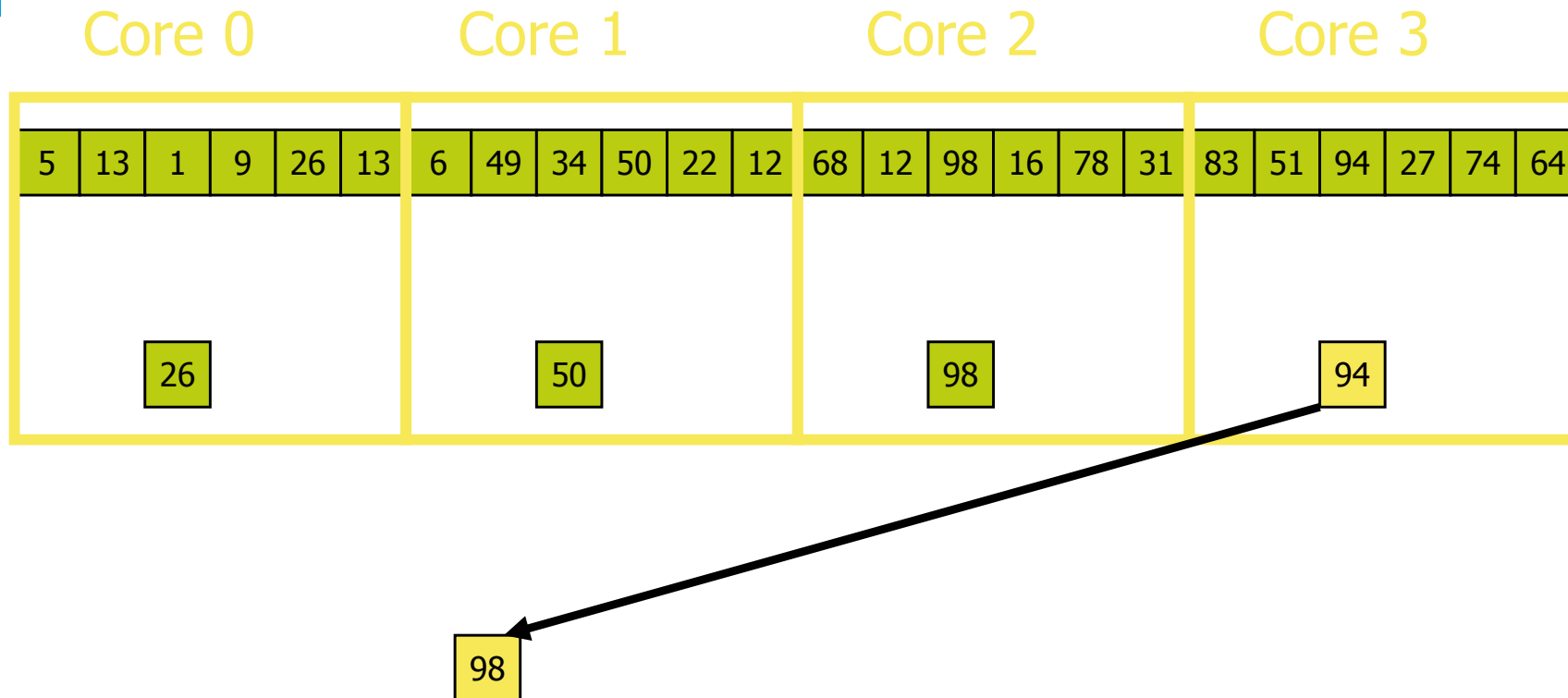
# Domain Decomposition

Find the largest element of an array



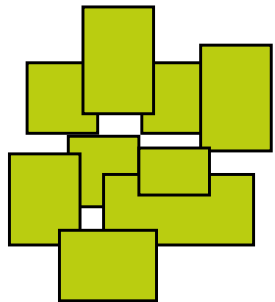
# Domain Decomposition

Find the largest element of an array

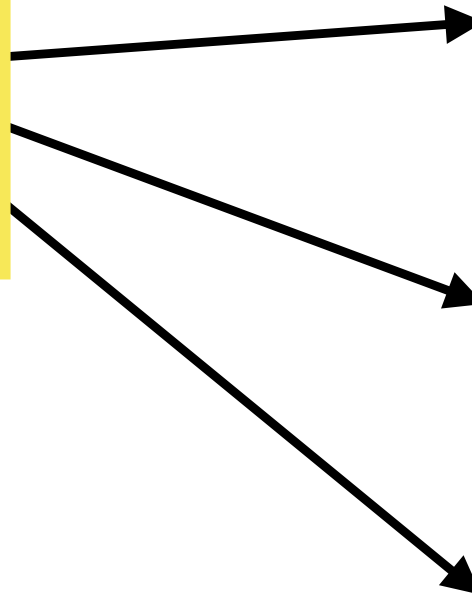


# Master Worker

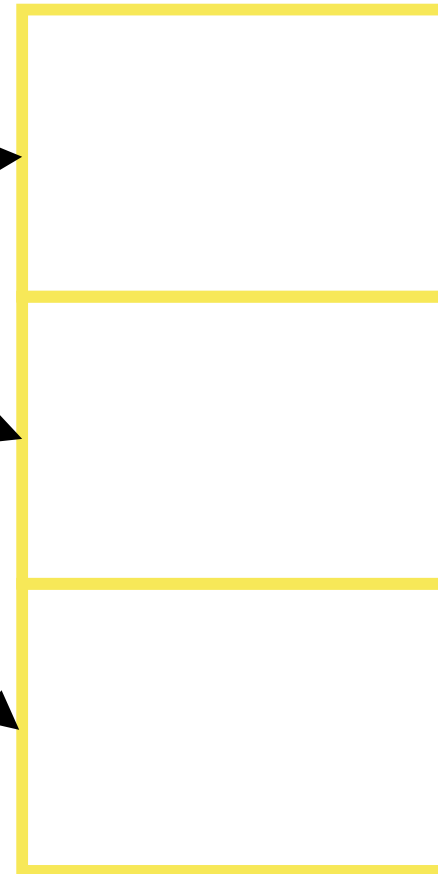
Get a heap of work done



Core 0 (master)



Core 1

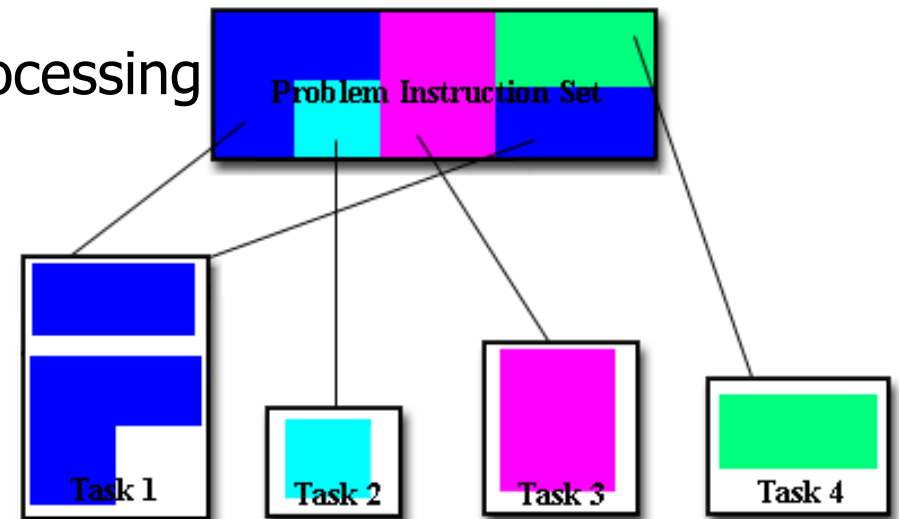


Core 2

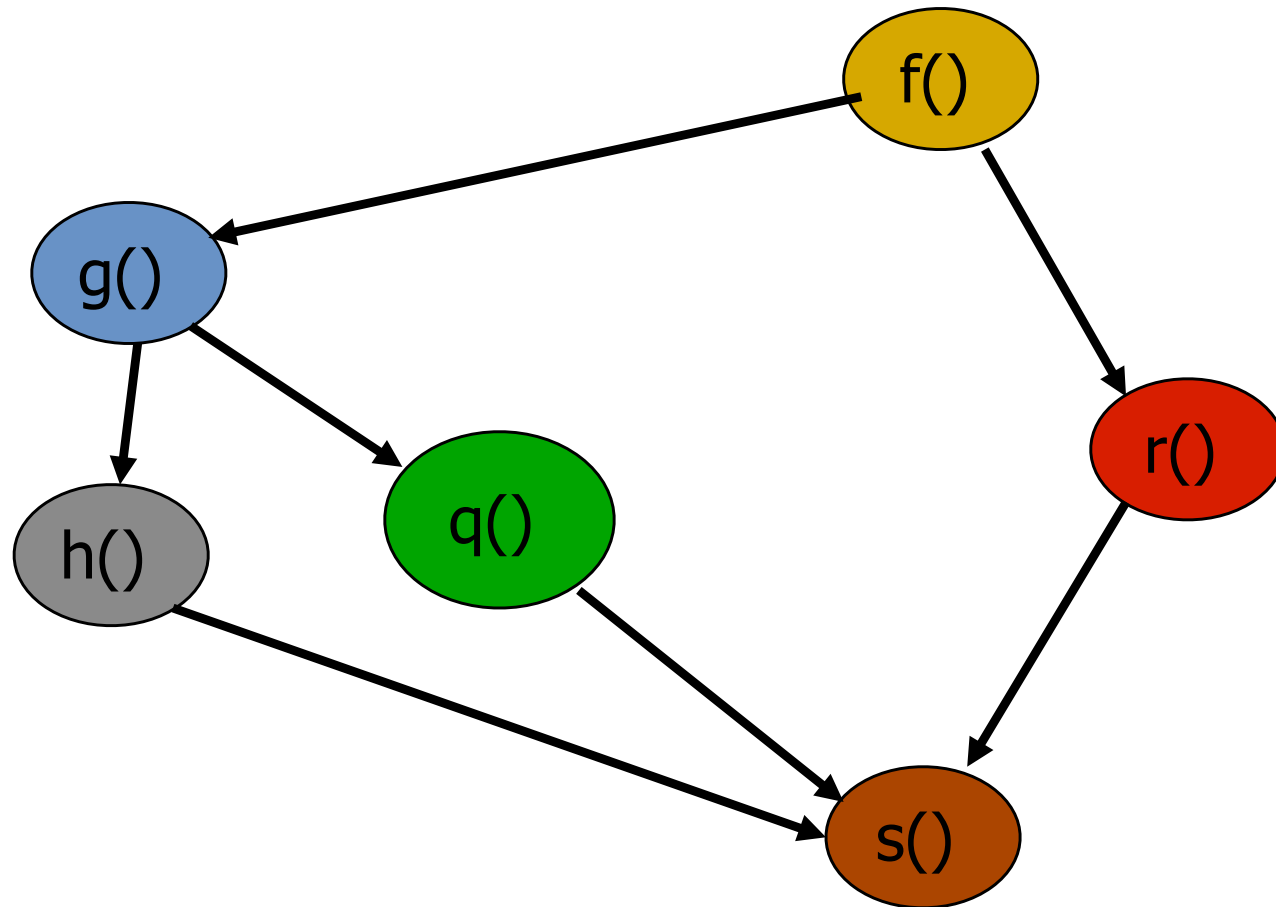
Core 3

# Functional Decomposition

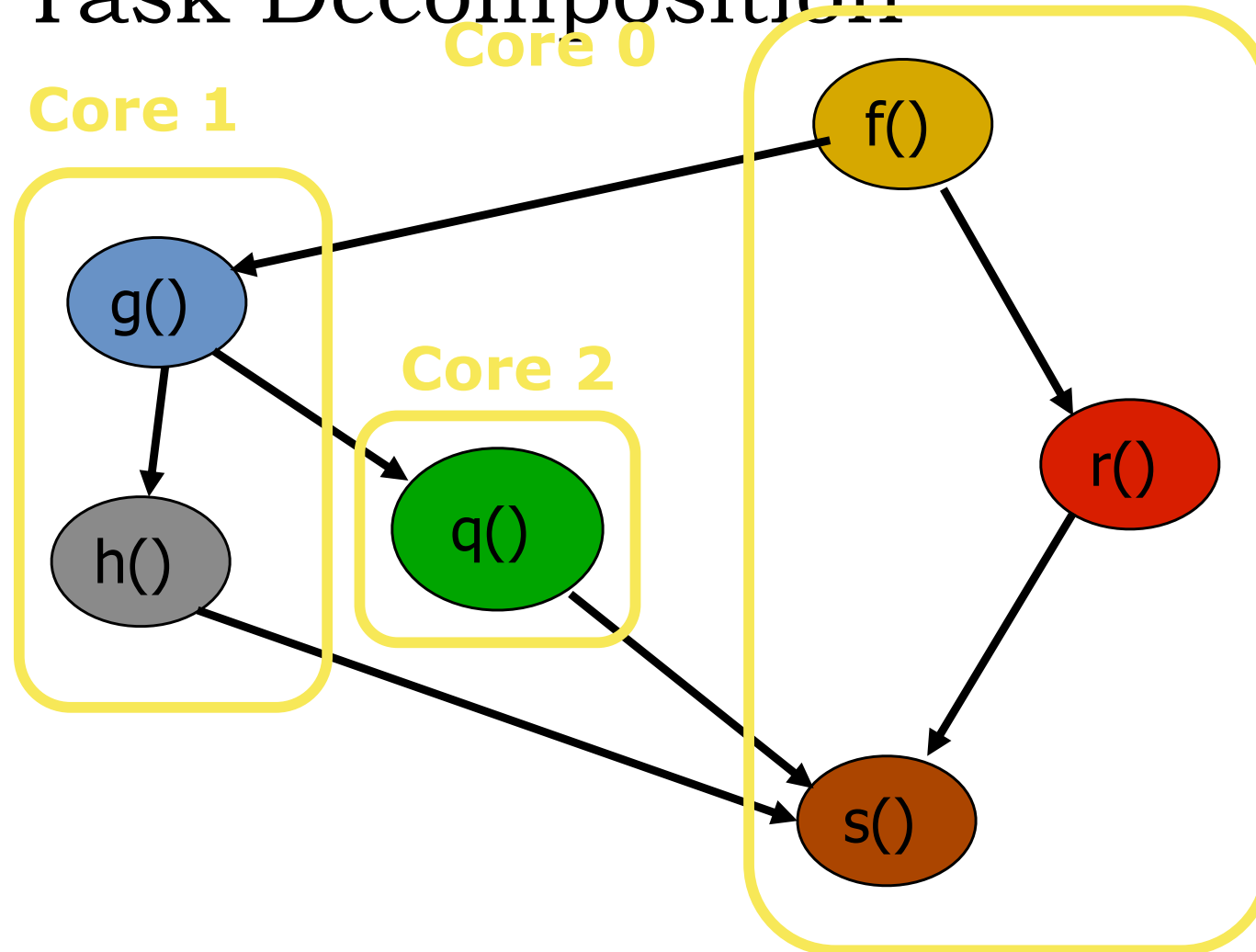
- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
- Divide computation based on natural set of independent tasks
  - ▶ Assign data for each task as needed
- Example: pipeline seismic data pre-processing
  - static-correction
  - deconvolution
  - nmo correction
  - stacking
  - ....



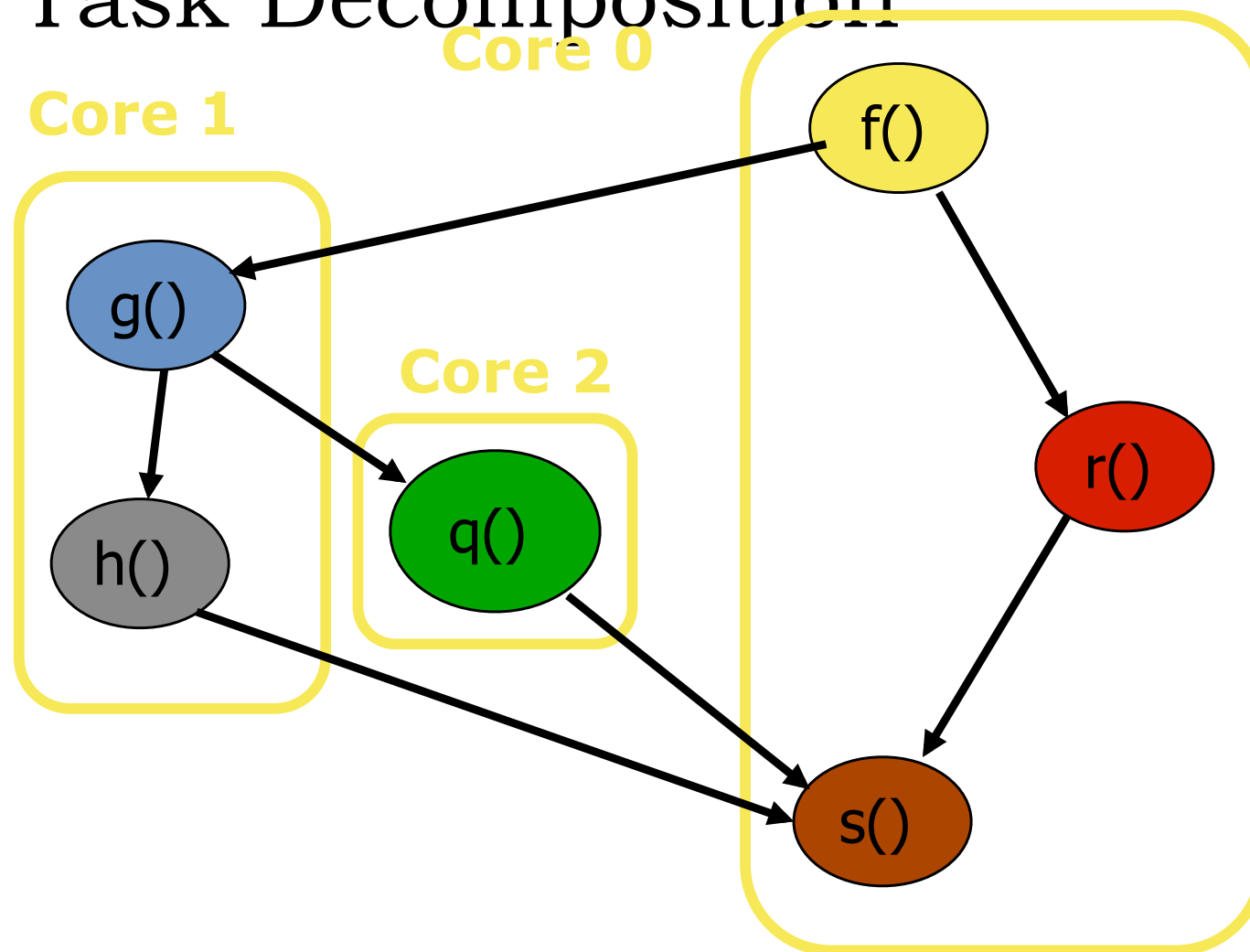
# Task/Functional Decomposition



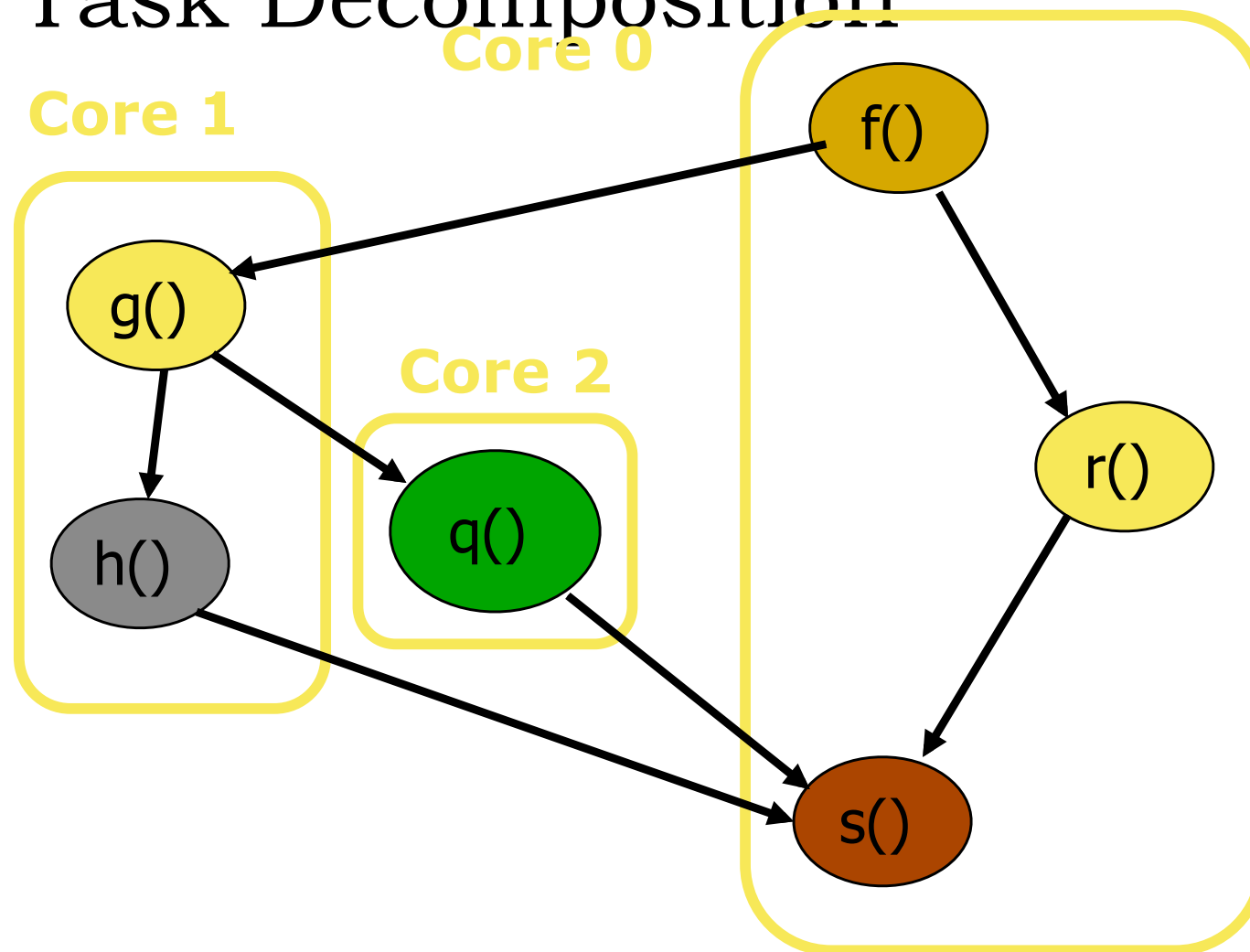
# Task Decomposition



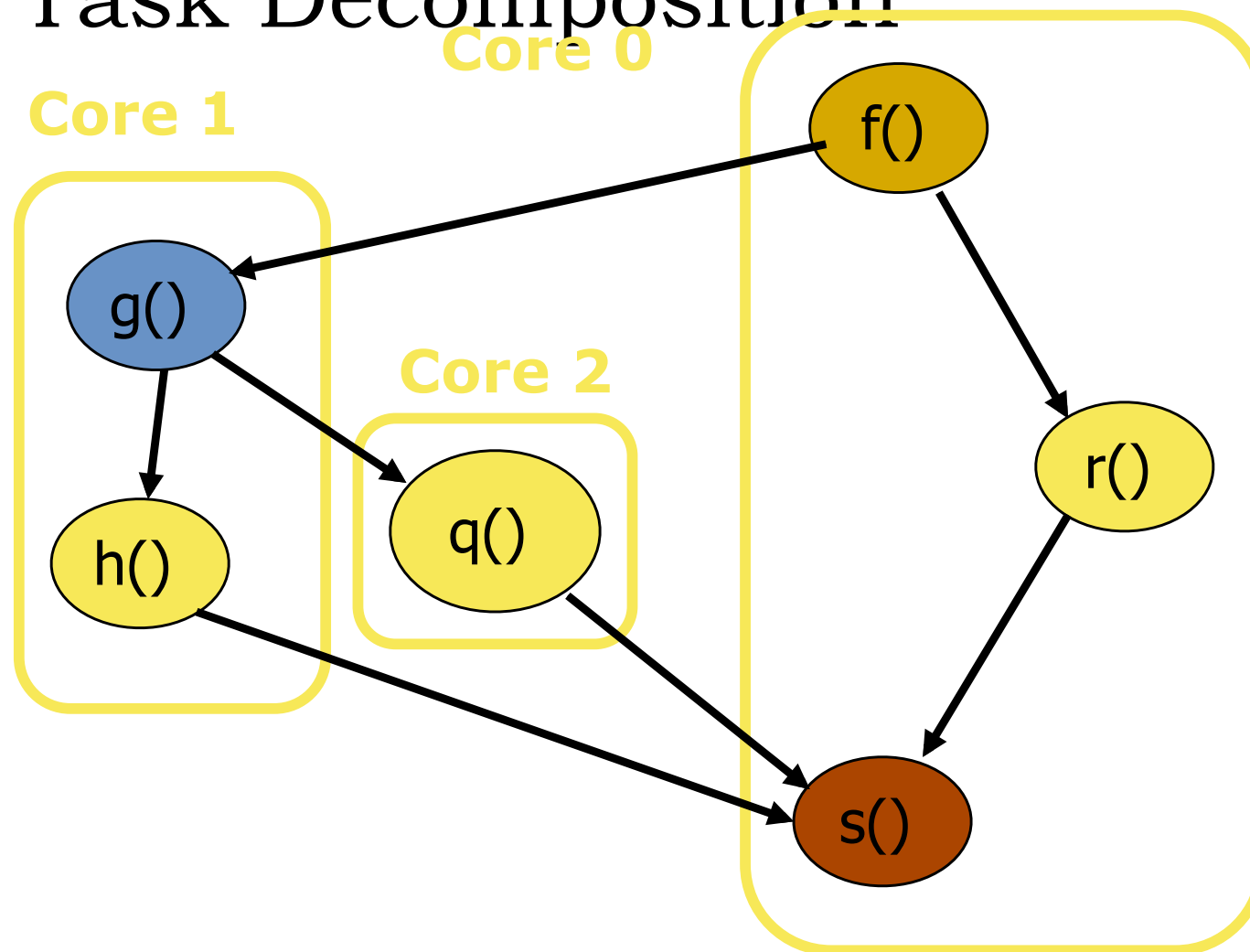
# Task Decomposition



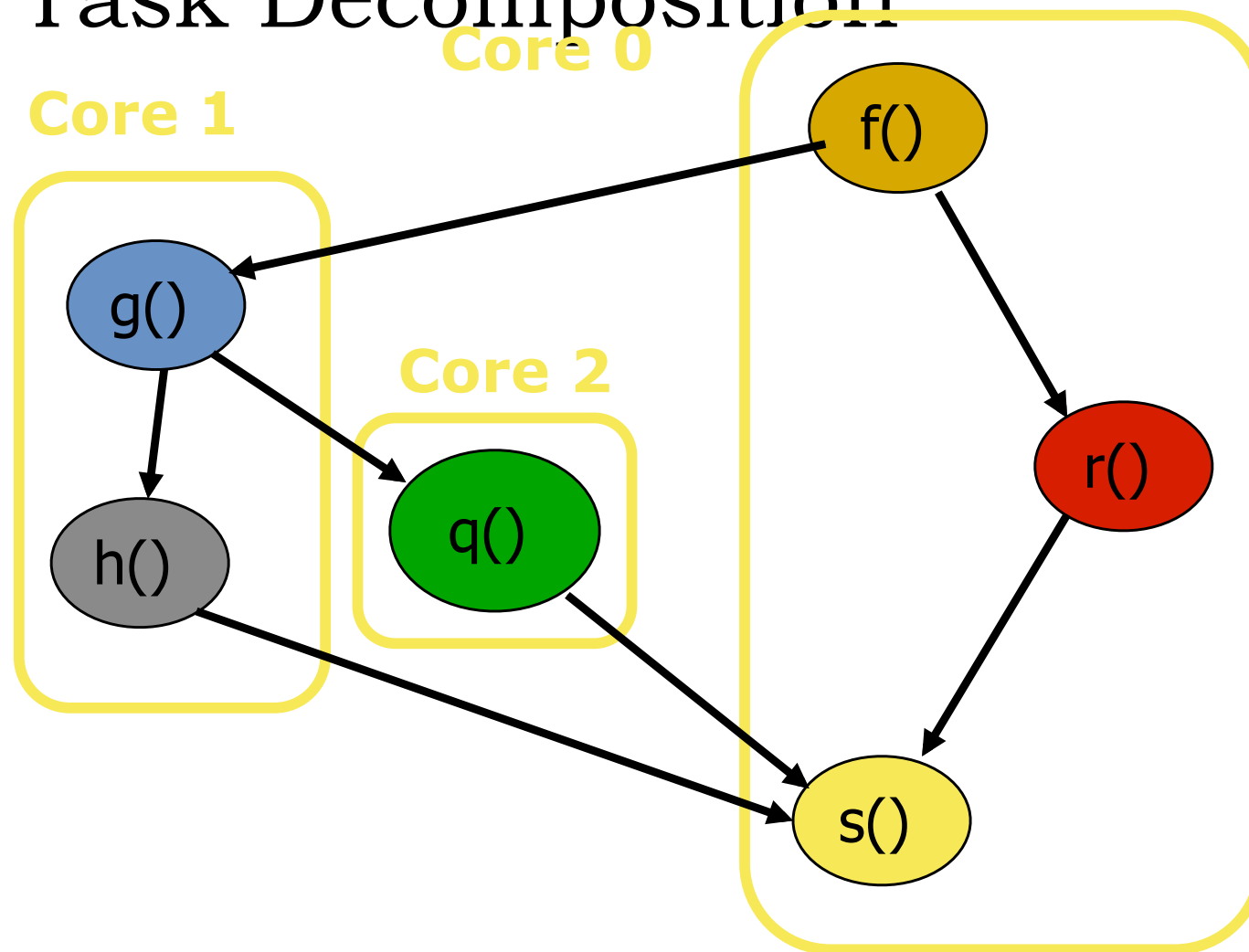
# Task Decomposition



# Task Decomposition



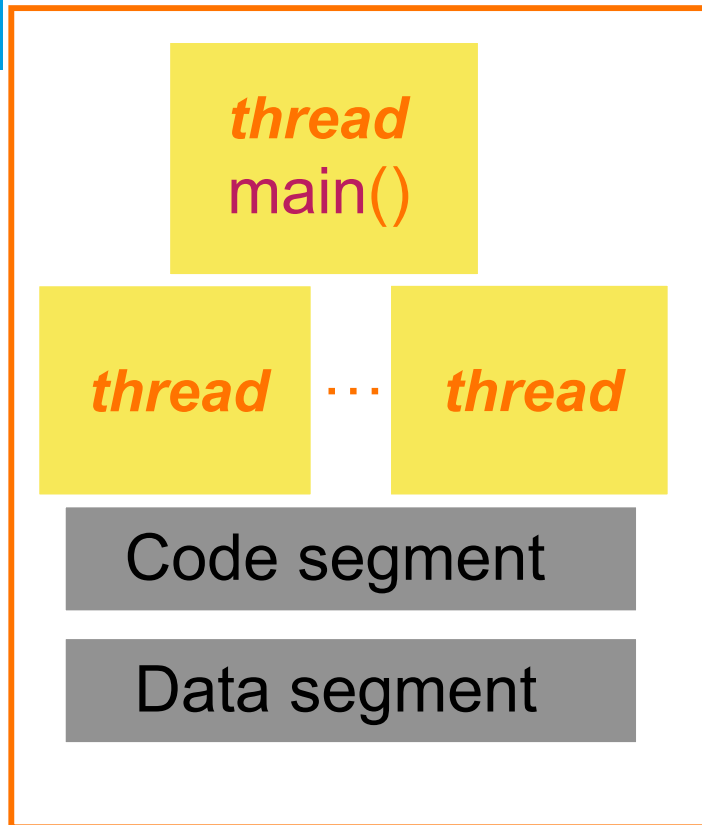
# Task Decomposition



# Shared Memory Parallelism

- Introduction to Threads
  - Exercise: Racecondition
- OpenMP Programming Model
  - Scope of Variables: Exercise 1
  - Synchronisation: Exercise 2
- Scheduling
  - Exercise: OpenMP scheduling
- Reduction
  - Exercise: Pi
- Shared variables
  - Exercise: CacheTrash
- Tasks
- Future of OpenMP

# Processes and Threads



Modern operating systems load programs as processes

Resource holder

Execution

A process starts executing at its entry point as a thread

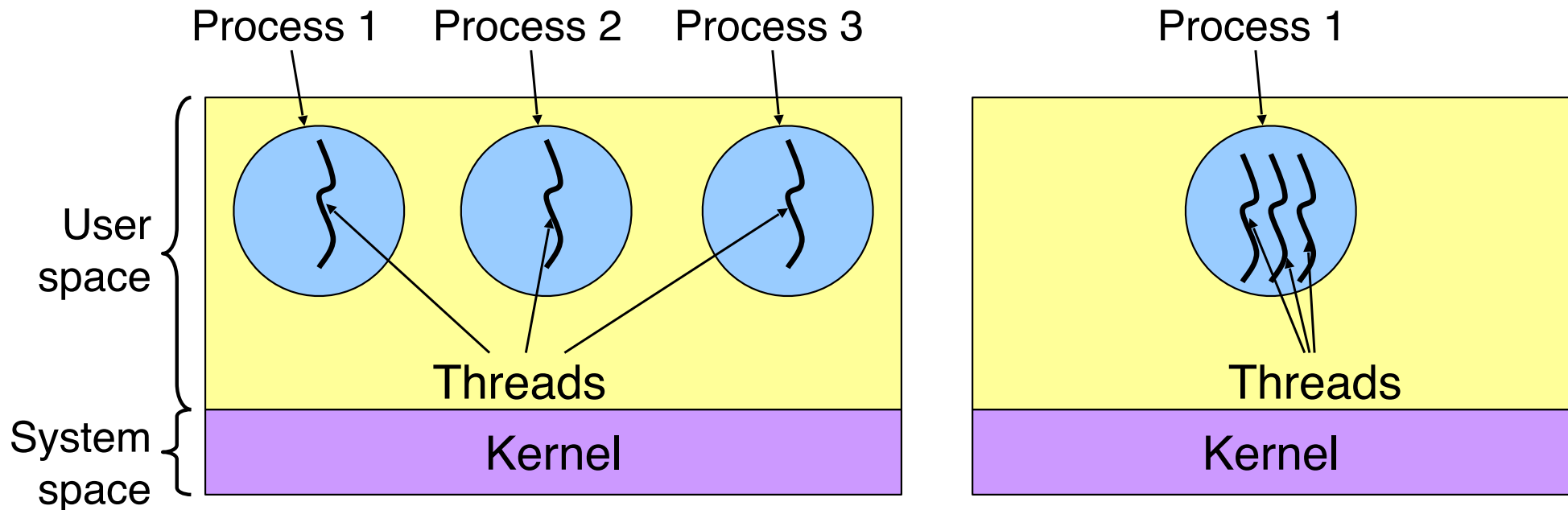
Threads can create other threads within the process

All threads within a process share code & data segments

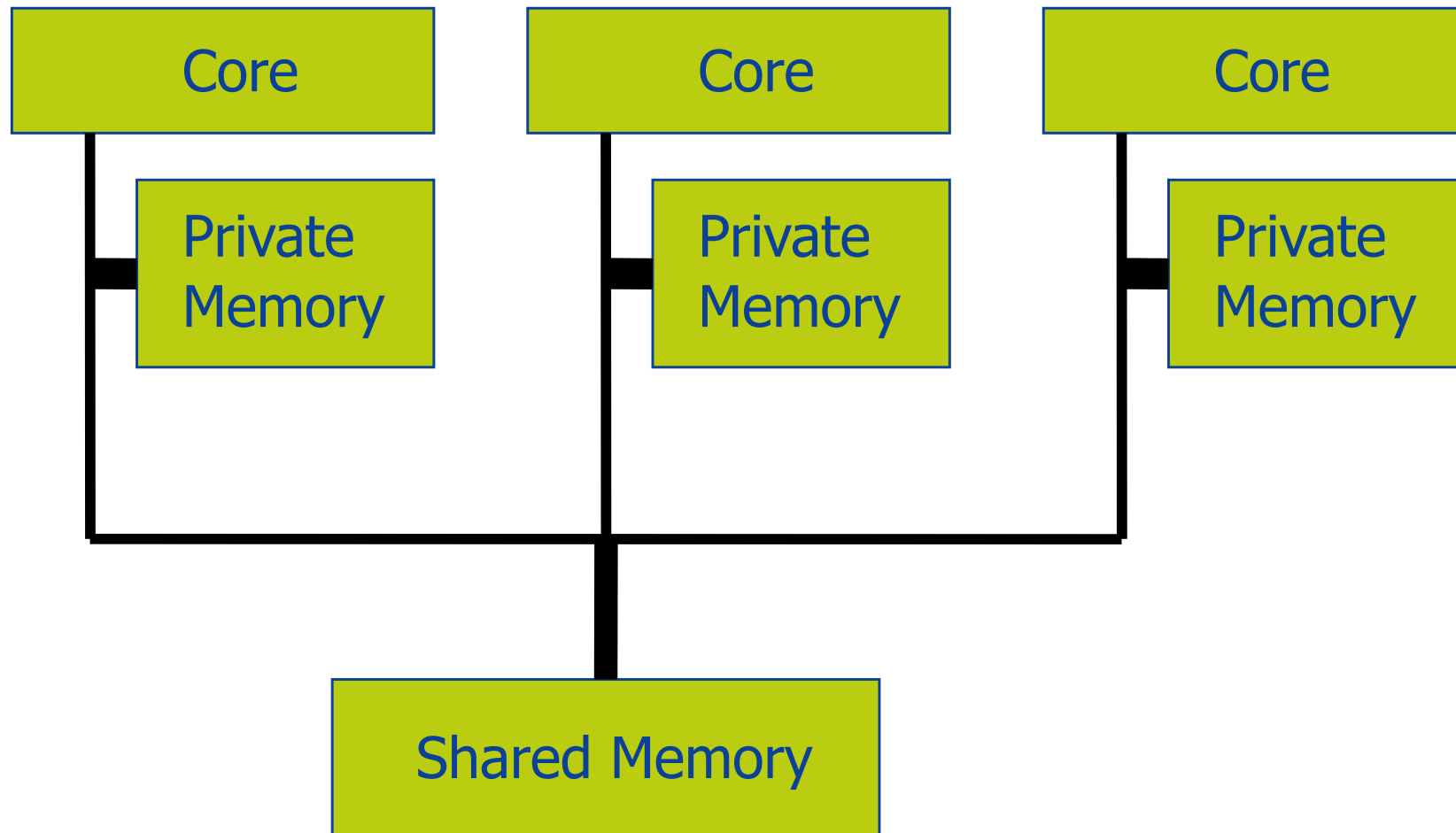
Threads have lower overhead than processes

# Threads: “processes” sharing memory

- Process == address space
- Thread == program counter / stream of instructions
- Two examples
  - Three processes, each with one thread
  - One process with three threads



# The Shared-Memory Model



# What Are Threads Good For?

Making programs easier to understand

Overlapping computation and I/O

Improving responsiveness of GUIs

Improving performance through parallel execution

- with the help of OpenMP

# Fork/Join Programming Model

- When program begins execution, only master thread active
- Master thread executes sequential portions of program
- For parallel portions of program, master thread ***forks*** (creates or awakens) additional threads
- At ***join*** (end of parallel section of code), extra threads are suspended or die

# Relating Fork/Join to Code (OpenMP)

```
██████████  
██████████  
██████████
```

```
for {
```

```
    ████████████████████  
    ████████████████  
    ████████████████  
    ████████████████
```

```
}
```

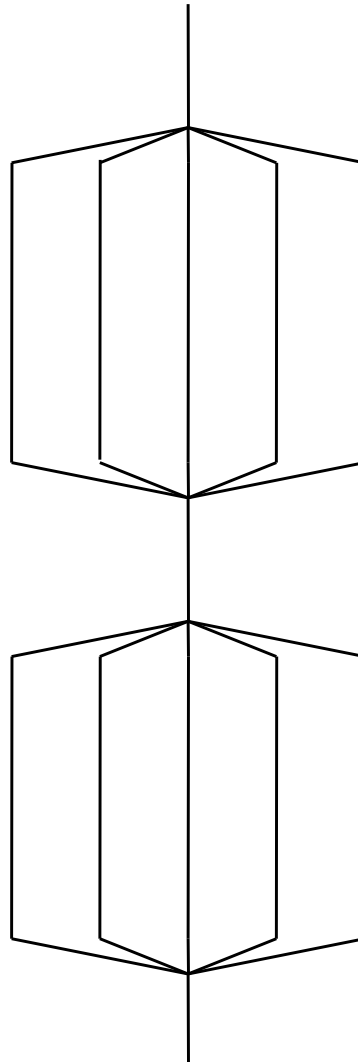
```
██████████  
██████████  
██████████  
██████████
```

```
for {
```

```
    ████████████████████  
    ████████████████  
    ████████████████  
    ████████████████
```

```
}
```

```
██████████  
██████████  
██████████
```



Sequential code

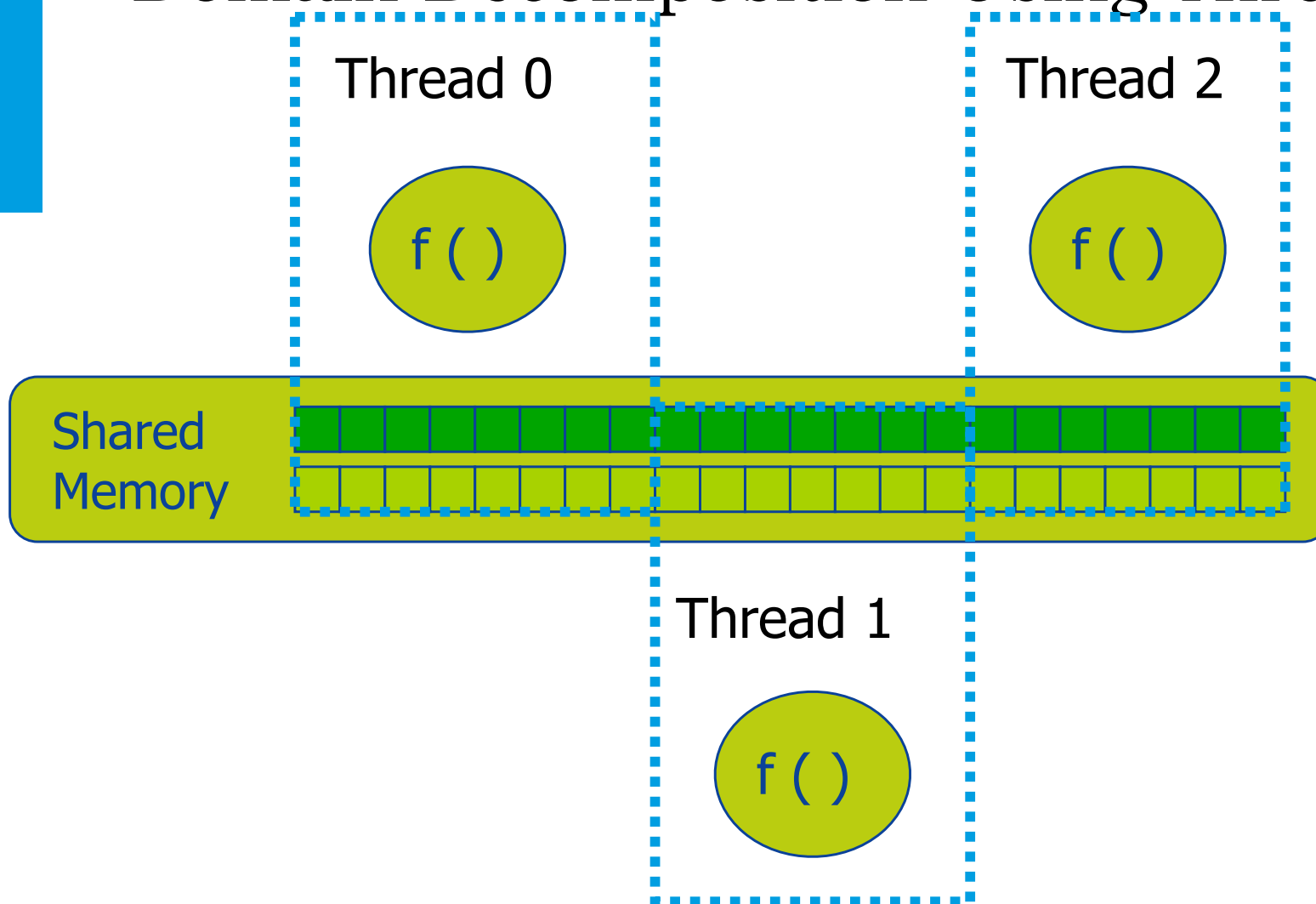
Parallel code

Sequential code

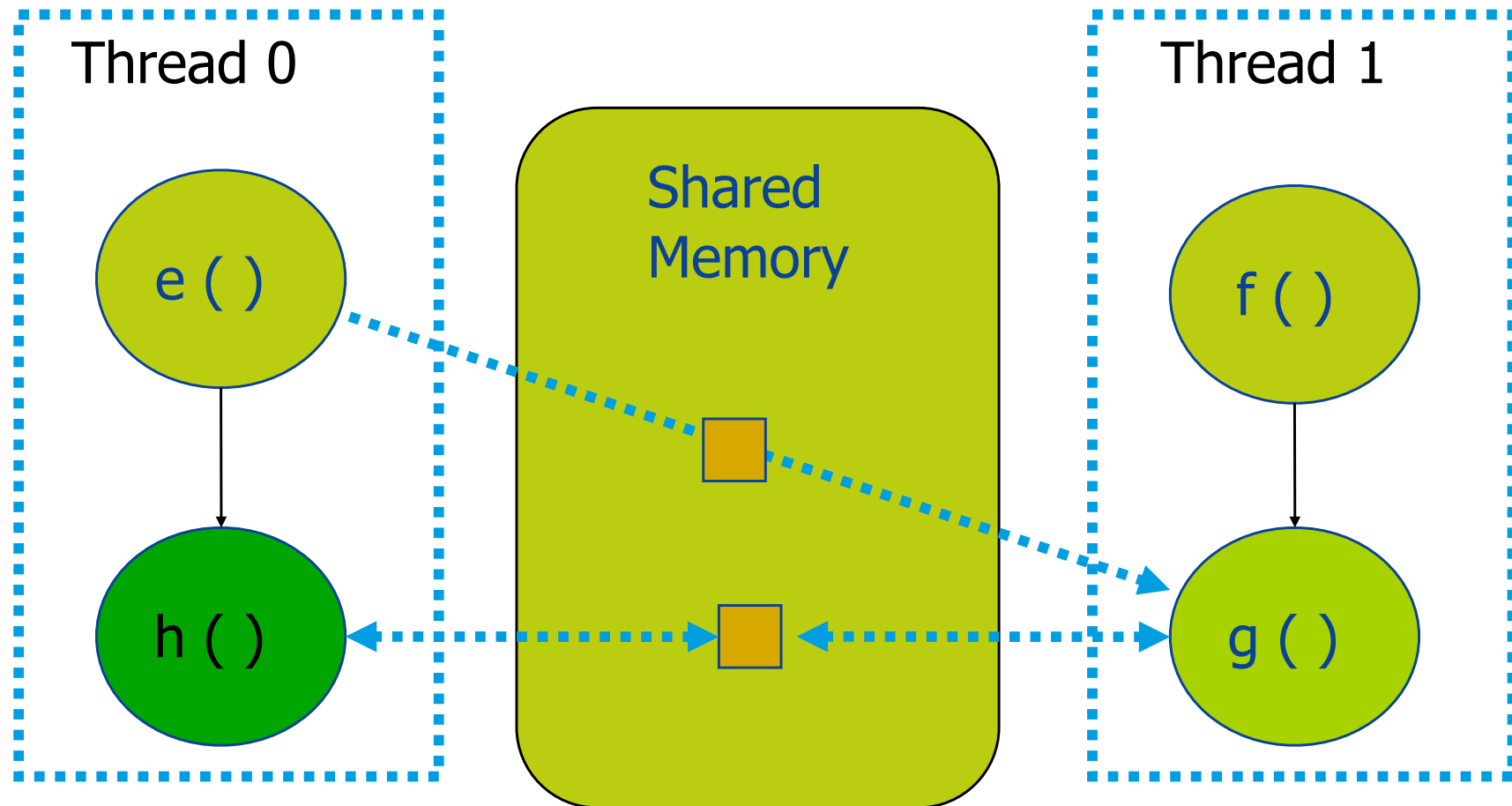
Parallel code

Sequential code

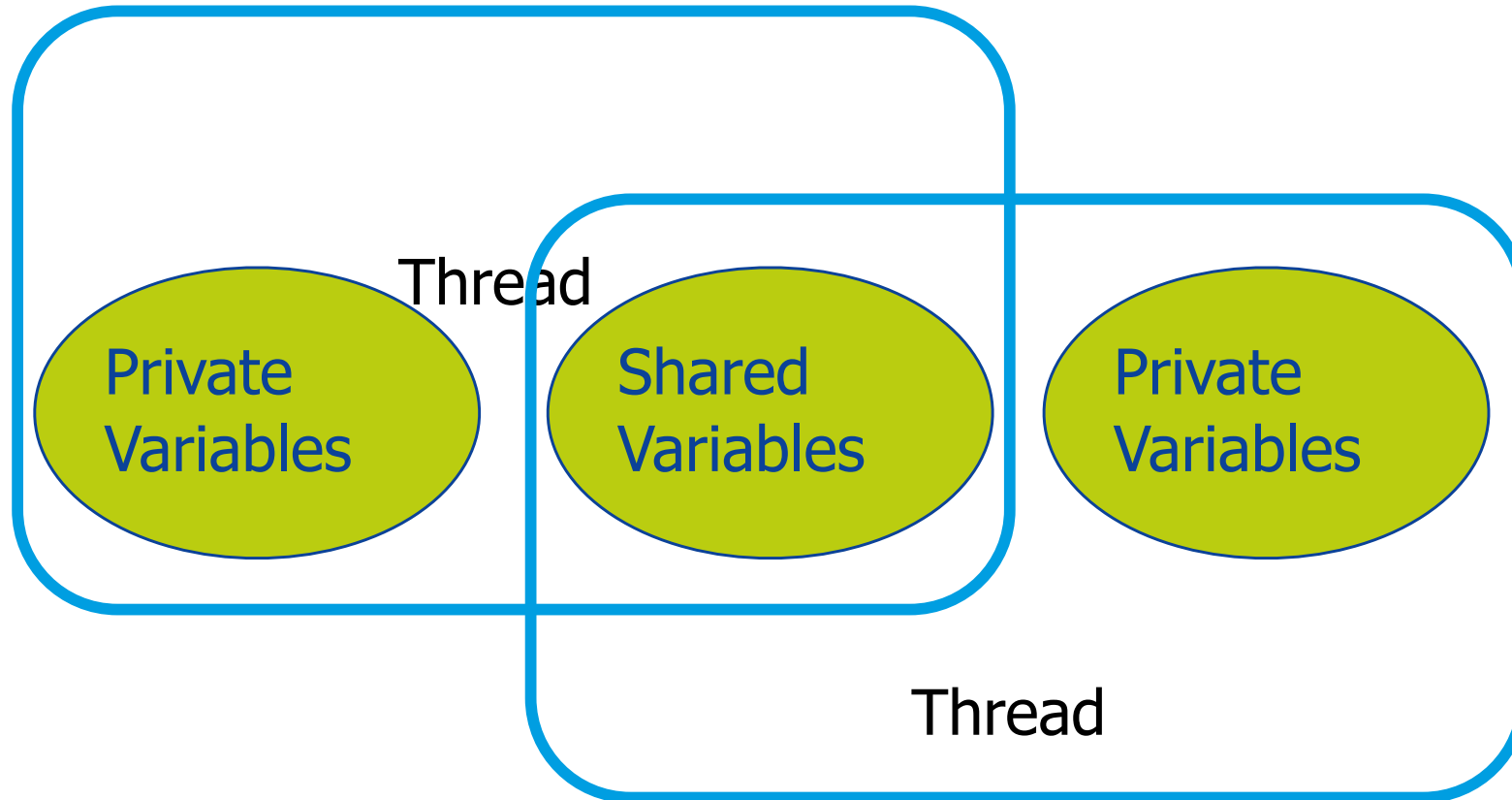
# Domain Decomposition Using Threads



# Task Decomposition Using Threads



# Shared versus Private Variables



# Race Conditions

Parallel threads can “race” against each other to update resources

Race conditions occur when execution order is assumed but not guaranteed

*Example: un-synchronised access to bank account*



# Race Conditions



Withdraws \$100  
from account



Deposits \$100  
into account

Initial balance = \$1000  
Final balance = ?

| Time  | Withdrawal              | Deposit                  |
|-------|-------------------------|--------------------------|
| $T_0$ | Load (balance = \$1000) |                          |
| $T_1$ | Subtract \$100          | Load (balance = \$1000)  |
| $T_2$ | Store (balance = \$900) | Add \$100                |
| $T_3$ |                         | Store (balance = \$1100) |

# Code Example in OpenMP

## exercise: HPCourse/RaceCondition

```
for (i=0; i<NMAX; i++) {  
    a[i] = 1;  
    b[i] = 2;  
}  
#pragma omp parallel for shared(a,b)  
for (i=0; i<12; i++) {  
    a[i+1] = a[i]+b[i];  
}
```

1: a= 1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0, 17.0, 19.0, 21.0, 23.0

4: a= 1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 3.0, 5.0, 7.0, 9.0, 11.0

4: a= 1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0, 17.0, 19.0, 21.0, 23.0

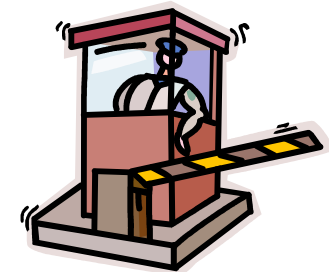
4: a= 1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0, 17.0, 19.0, 21.0, 23.0

# Code Example in OpenMP

| thread | computation                      |
|--------|----------------------------------|
| 0      | a[1] = a[0] + b[0]               |
| 0      | a[2] = a[1] + b[1]               |
| 0      | a[3] = a[2] + b[2] <--  Problem  |
| 1      | a[4] = a[3] + b[3] <--  Problem  |
| 1      | a[5] = a[4] + b[4]               |
| 1      | a[6] = a[5] + b[5] <--  Problem  |
| 2      | a[7] = a[6] + b[6] <--  Problem  |
| 2      | a[8] = a[7] + b[7]               |
| 2      | a[9] = a[8] + b[8] <--  Problem  |
| 3      | a[10] = a[9] + b[9] <--  Problem |
| 3      | a[11] = a[10] + b[10]            |

# How to Avoid Data Races

- Scope variables to be local to threads
  - Variables declared within threaded functions
  - Allocate on thread's stack
  - TLS (Thread Local Storage)
- Control shared access with critical regions
  - Mutual exclusion and synchronization
  - Lock, semaphore, event, critical section, mutex...



# Examples variables

# Domain Decomposition

## Sequential Code:

```
int a[1000], i;  
for (i = 0; i < 1000; i++) a[i] = foo(i);
```

# Domain Decomposition

## Sequential Code:

```
int a[1000], i;  
for (i = 0; i < 1000; i++) a[i] = foo(i);
```

## Thread 0:

```
for (i = 0; i < 500; i++) a[i] = foo(i);
```

## Thread 1:

```
for (i = 500; i < 1000; i++) a[i] = foo(i);
```

# Domain Decomposition

## Sequential Code:

```
int a[1000], i;  
for (i = 0; i < 1000; i++) a[i] = foo(i);
```

## Thread 0:

```
for (i = 0; i < 500; i++) a[i] = foo(i);
```

## Thread 1:

```
for (i = 500; i < 1000; i++) a[i] = foo(i);
```

Private

Shared

# Task Decomposition

```
int e;
```

```
main () {  
    int x[10], j, k, m;    j = f(k);    m = g(k); ...  
}
```

```
int f(int *x, int k)  
{  
    int a;    a = e * x[k] * x[k];    return a;  
}
```

```
int g(int *x, int k)  
{  
    int a;    k = k-1;    a = e / x[k];    return a;  
}
```

# Task Decomposition

```
int e;
```

```
main () {  
    int x[10], j, k, m;    j = f(k);    m = g(k);  
}
```

```
int f(int *x, int k)  
{  
    int a;    a = e * x[k] * x[k];    return a;  
}
```

Thread 0

```
int g(int *x, int k)  
{  
    int a;    k = k-1;    a = e / x[k];    return a;  
}
```

Thread 1

# Task Decomposition

`int e;` Static variable: Shared

```
main () {  
    int x[10], j, k, m;    j = f(k);    m = g(k);  
}
```

```
int f(int *x, int k)      Thread 0  
{  
    int a;    a = e * x[k] * x[k];    return a;  
}
```

```
int g(int *x, int k)      Thread 1  
{  
    int a;    k = k-1;    a = e / x[k];    return a;  
}
```

# Task Decomposition

```
int e;
```

Heap variable: Shared

```
main () {
```

```
    int x[10], j, k, m;    j = f(x, k);    m = g(x, k);  
}
```

```
int f(int *x, int k)
```

Thread 0

```
{
```

```
    int a;    a = e * x[k] * x[k];    return a;
```

```
}
```

```
int g(int *x, int k)
```

Thread 1

```
{
```

```
    int a;    k = k-1;    a = e / x[k];    return a;
```

```
}
```

# Task Decomposition

```
int e;
```

```
main () {  
    int x[10], j, k, m;    j = f(k);    m = g(k);  
}
```

Function's local variables: Private

```
int f(int *x, int k)  
{  
    int a;    a = e * x[k] * x[k];    return a;  
}
```

Thread 0

```
int g(int *x, int k)  
{  
    int a;    k = k-1;    a = e / x[k];    return a;  
}
```

Thread 1

# Shared and Private Variables

- Shared variables
  - Static variables
  - Heap variables
  - Contents of run-time stack at time of call
- Private variables
  - Loop index variables
  - Run-time stack of functions invoked by thread



# What Is OpenMP?

- Compiler directives for multithreaded programming
- Easy to create threaded Fortran and C/C++ codes
- Supports data parallelism model
- Portable and Standard
- Incremental parallelism
  - ➔ Combines serial and parallel code in single source

# OpenMP is not ...

## **Not** Automatic parallelization

- User explicitly specifies parallel execution
- Compiler does **not** ignore user directives even if wrong

## **Not** just loop level parallelism

- Functionality to enable general parallel parallelism

## **Not** a new language

- Structured as extensions to the base
- Minimal functionality with opportunities for extension

# Directive based

- Directives are special comments in the language
  - Fortran fixed form: !\$OMP, C\$OMP, \*\$OMP
  - Fortran free form: !\$OMP

Special comments are interpreted by OpenMP compilers

```
w = 1.0/n
sum = 0.0
!$OMP PARALLEL DO PRIVATE(x) REDUCTION(+:sum)
do I=1,n
    x = w*(I-0.5)
    sum = sum + f(x)
end do
pi = w*sum
print *,pi
end
```

Comment in  
Fortran  
but interpreted by  
OpenMP compilers

# C example

## #pragma omp directives in C

- Ignored by non-OpenMP compilers

```
w = 1.0/n;  
sum = 0.0;  
#pragma omp parallel for private(x) reduction(+:sum)  
  for(i=0, i<n, i++) {  
    x = w*((double)i+0.5);  
    sum += f(x);  
  }  
pi = w*sum;  
printf("pi=%g\n", pi);  
}
```

# Architecture of OpenMP

## Directives, Pragmas

- Control structures
- Work sharing
- Synchronization
- Data scope attributes
  - private
  - shared
  - reduction
- Orphaning

## Runtime library routines

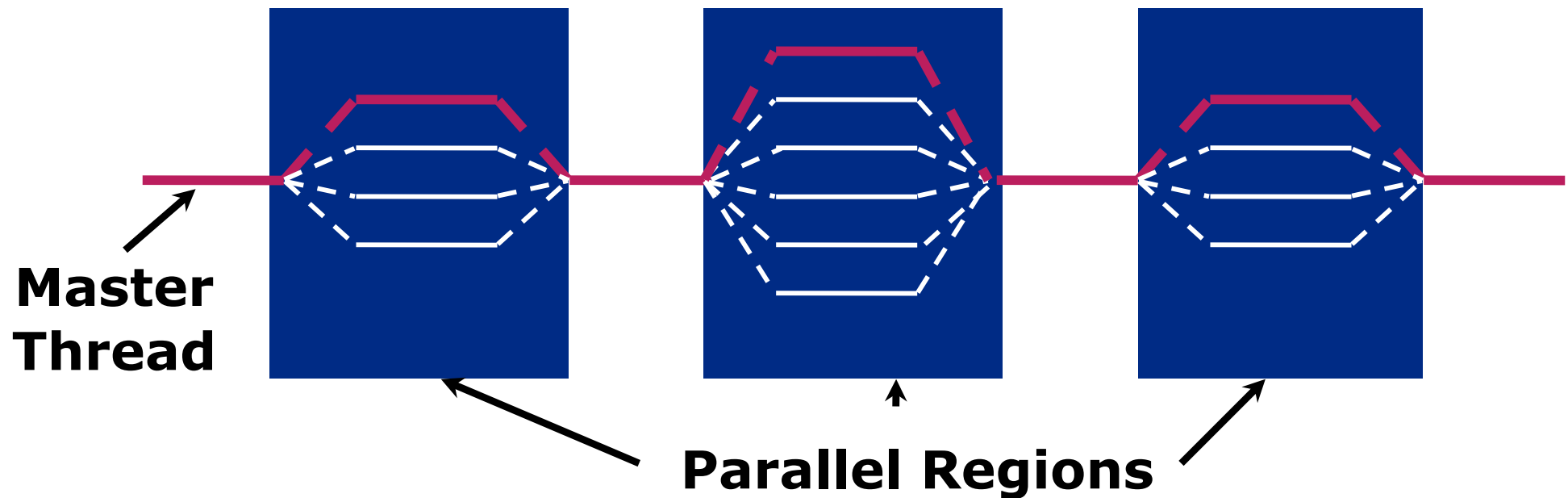
- Control & query routines
  - number of threads
  - throughput mode
  - nested parallelism
- Lock API

## Environment variables

- Control runtime
  - schedule type
  - max threads
  - nested parallelism
  - throughput mode

# Programming Model

- Fork-join parallelism:
  - ▶ Master thread spawns a team of threads as needed
  - ▶ Parallelism is added incrementally: the sequential program evolves into a parallel program

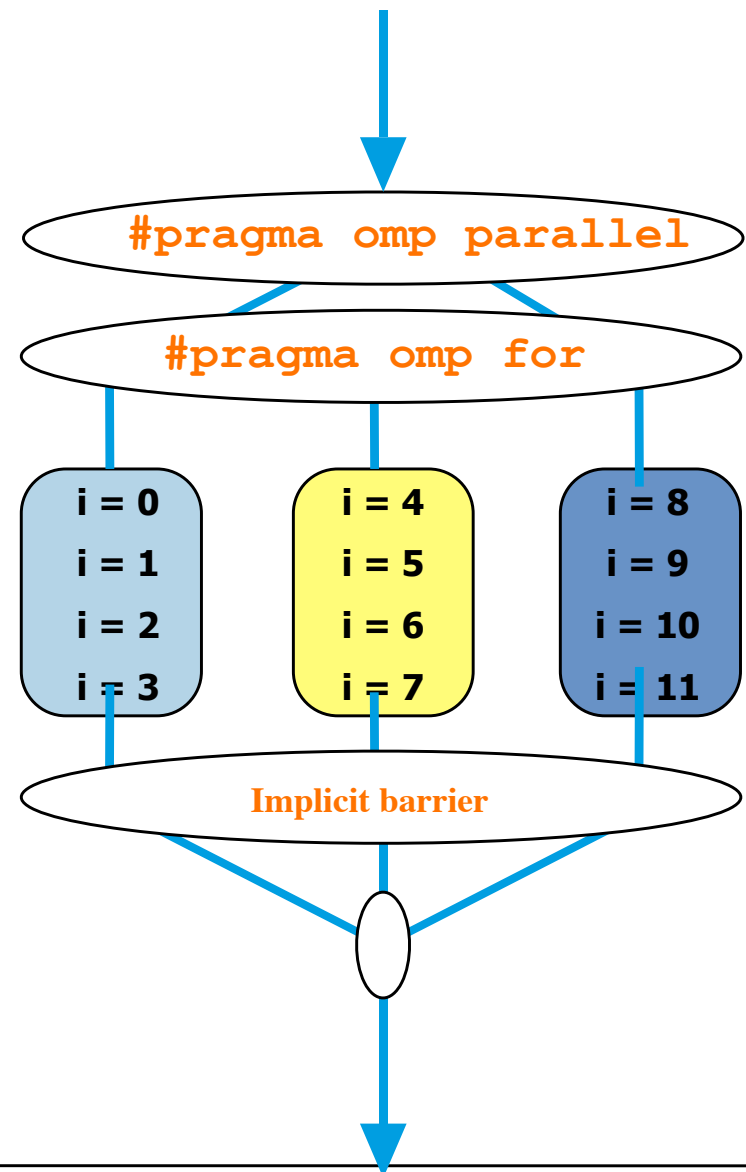


# Work-sharing Construct

```
#pragma omp parallel
#pragma omp for
  for(i = 0; i < 12; i++)
    c[i] = a[i] + b[i]
```

Threads are assigned an independent set of iterations

Threads must wait at the end of work-sharing construct



# Combining pragmas

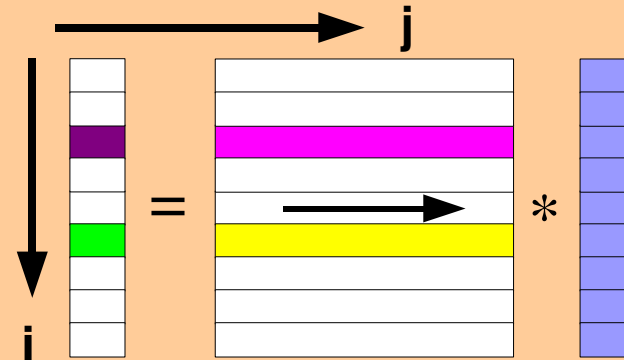
These two code segments are equivalent

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0; i< MAX; i++) {
        res[i] = huge();
    }
```

# Matrix-vector example

```
#pragma omp parallel for default(none) \
    private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
    sum = 0.0;
    for (j=0; j<n; j++)
        sum += b[i][j]*c[j];
    a[i] = sum;
}
```



**TID = 0**

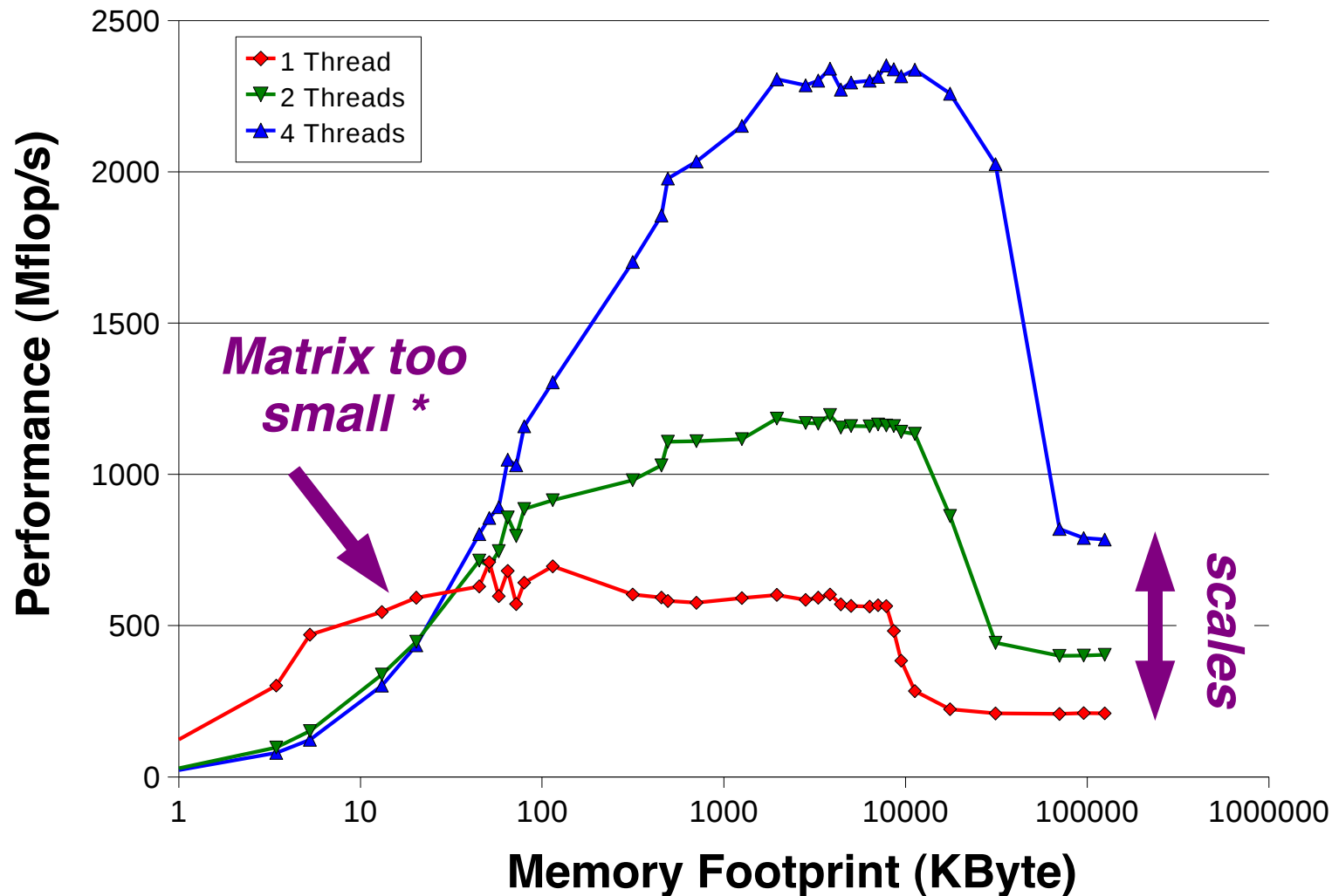
```
for (i=0,1,2,3,4)
    i = 0
    sum =  $\sum$  b[i=0][j]*c[j]
    a[0] = sum
    i = 1
    sum =  $\sum$  b[i=1][j]*c[j]
    a[1] = sum
```

**TID = 1**

```
for (i=5,6,7,8,9)
    i = 5
    sum =  $\sum$  b[i=5][j]*c[j]
    a[5] = sum
    i = 6
    sum =  $\sum$  b[i=6][j]*c[j]
    a[6] = sum
```

*etc*

# Performance is matrix size dependent



# OpenMP parallelization

- OpenMP Team := Master + Workers
- A Parallel Region is a block of code executed by all threads simultaneously
  - The master thread always has thread ID 0
  - Thread adjustment (if enabled) is only done before entering a parallel region
  - Parallel regions can be nested, but support for this is implementation dependent
  - An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially
- A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work

# Data Environment

- OpenMP uses a shared-memory programming model
  - Most variables are shared by default.
  - Global variables are shared among threads  
C/C++: File scope variables, static
- Not everything is shared, there is often a need for “local” data as well

# Data Environment

... not everything is shared...

- Stack variables in functions called from parallel regions are PRIVATE
- Automatic variables within a statement block are PRIVATE
- Loop index variables are private (with exceptions)  
C/C+: The **first** loop index variable in nested loops following a `#pragma omp for`

# About Variables in SMP

- Shared variables

Can be accessed by every thread. Independent read/write operations can take place.

- Private variables

Every thread has its own copy of the variables that are created/destroyed upon entering/leaving the procedure. They are not visible to other threads.

| <b>serial code</b> | <b>parallel code</b> |
|--------------------|----------------------|
| global             | shared               |
| auto local         | local                |
| static             | use with care        |
| dynamic            | use with care        |

# Data Scope clauses

attribute clauses

`default (shared)`

`shared (varname, ...)`

`private (varname, ...)`

# The Private Clause

Reproduces the variable for each thread

- Variables are un-initialised; C++ object is default constructed
- Any value external to the parallel region is undefined

```
void* work(float* c, float *a, float
*x, int N)
{
    float x, y; int i;
    #pragma omp parallel for private(x,y)
        for(i=0; i<N; i++) {
            x = a[i]; y = b[i];
            c[i] = x + y;
        }
}
```

# Synchronization

- Barriers `#pragma omp barrier`
- Critical sections `#pragma omp critical()`
- Lock library routines

`omp_set_lock(omp_lock_t *lock)`

`omp_unset_lock(omp_lock_t *lock)`

`...`

# OpenMP Critical Construct

```
#pragma omp critical [(lock_name)]
```

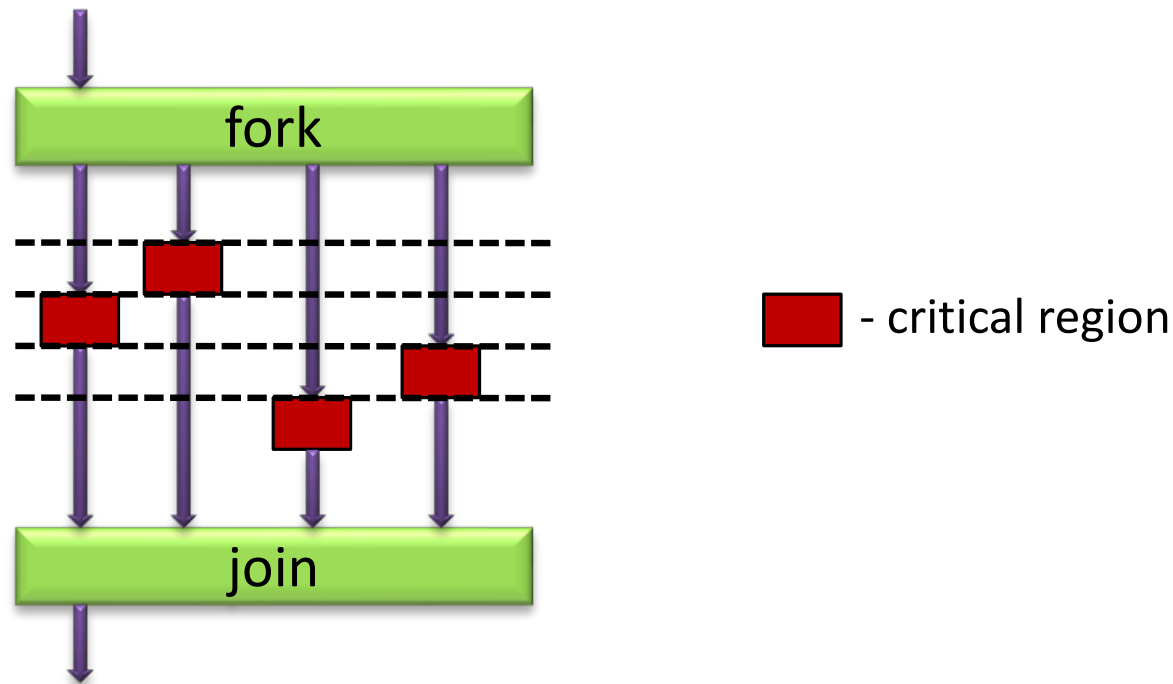
Defines a critical region on a structured block

All threads execute the code, but only one at a time. Only one calls `consum()` thereby protecting R1 and R2 from race conditions.

Naming the critical constructs is optional, but may increase performance.

```
float R1, R2;
#pragma omp parallel
{ float A, B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
    #pragma omp critical(R1_lock)
      consum (B, &R1);
    A = bigger_job(i);
    #pragma omp critical(R2_lock)
      consum (A, &R2);
  }
}
```

# OpenMP Critical



All threads execute the code, but only one at a time. Other threads in the group must wait until the current thread exits the critical region. Thus only one thread can manipulate values in the critical region.

# Critical Example 1

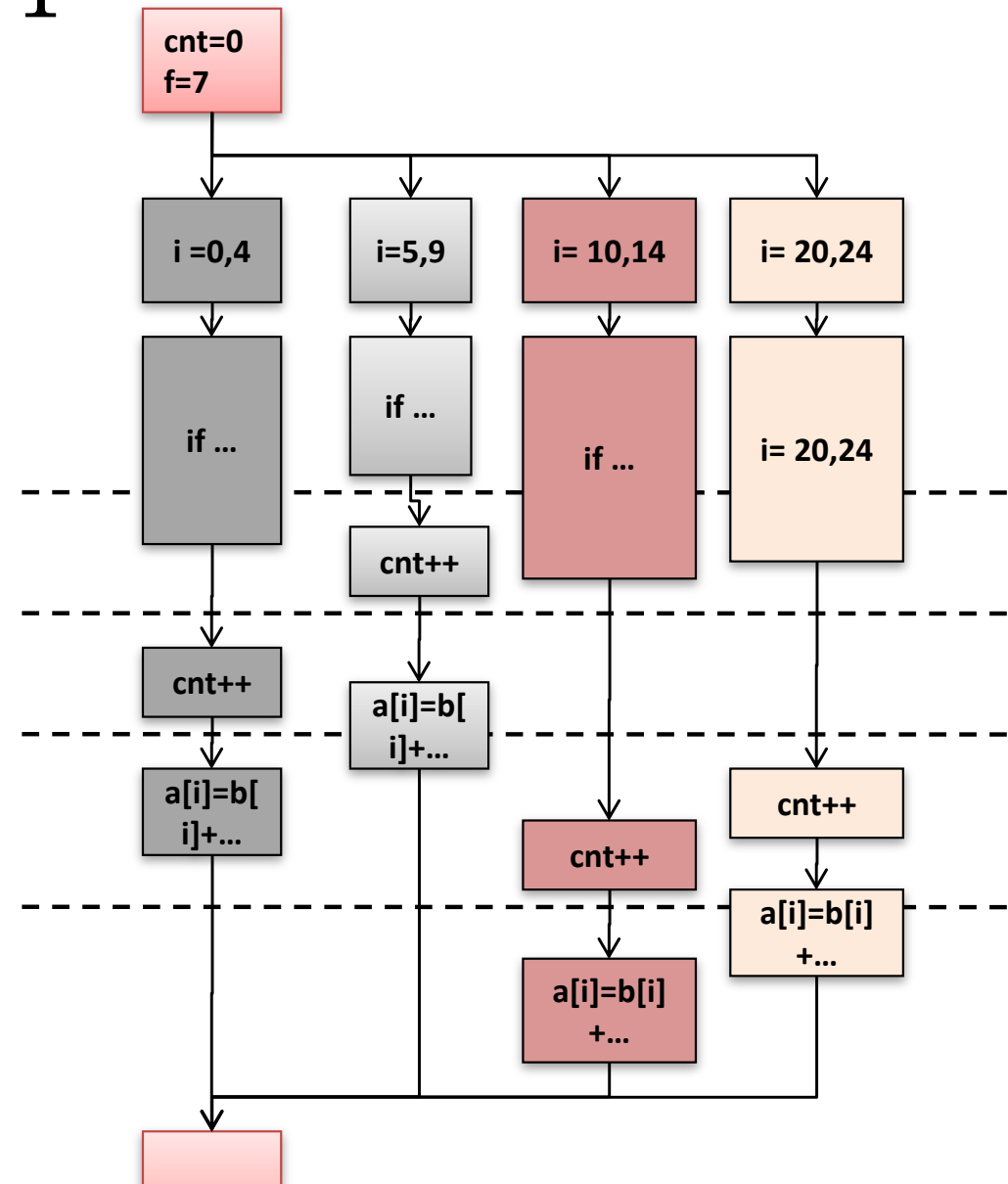
```

cnt = 0;
f = 7;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i<20;i++){
        if(b[i] == 0){

            #pragma omp critical
            cnt ++;

        } /* end if */
        a[i]=b[i]+f*(i+1);
    } /* end for */
} /* omp end parallel */

```



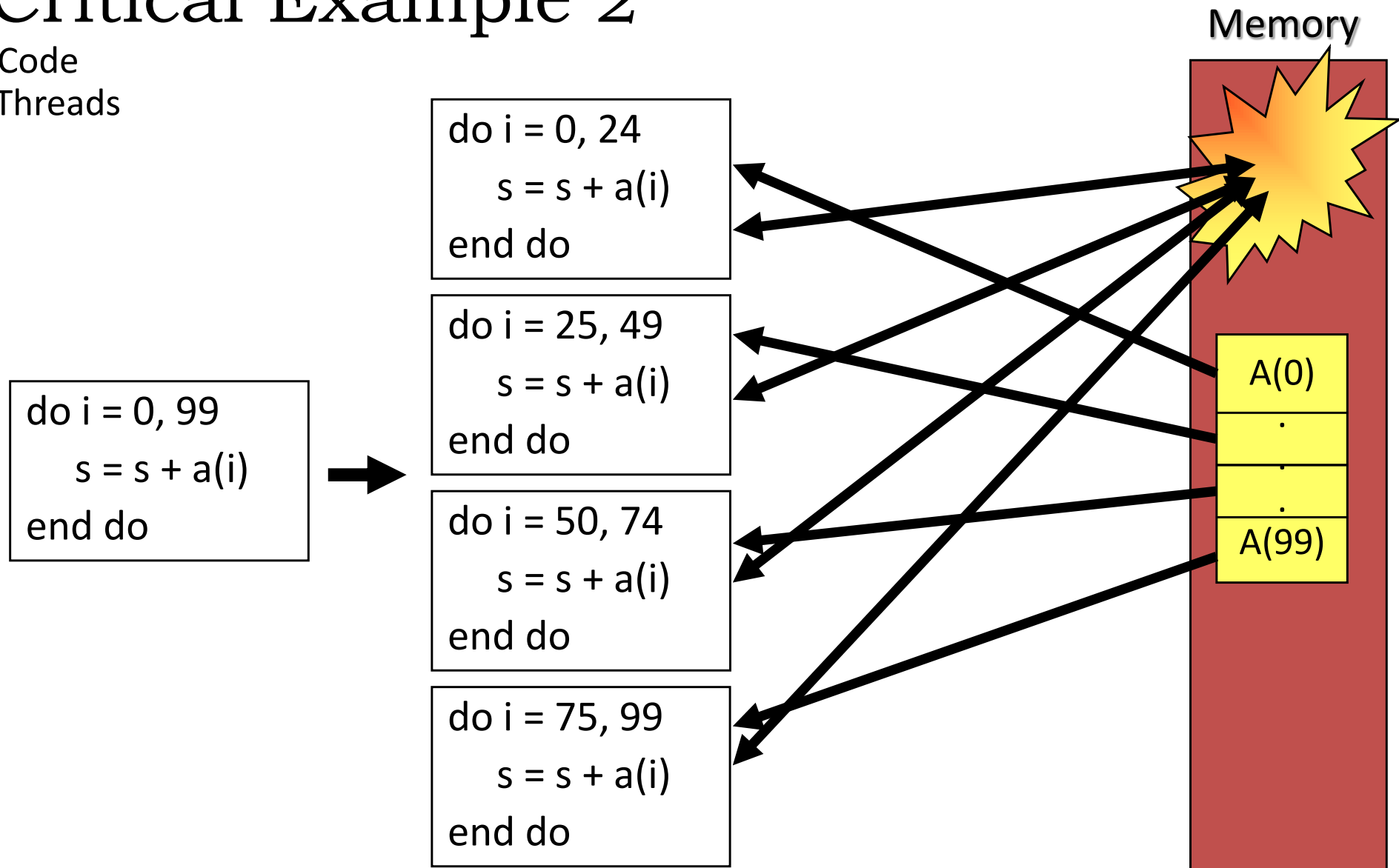
# Critical Example 2

```
int i;  
  
#pragma omp parallel for  
for (i = 0; i < 100; i++) {  
    s = s + a[i]; }  
}
```

# Critical Example 2

Pseudo-Code

Here: 4 Threads



# OpenMP Single Construct

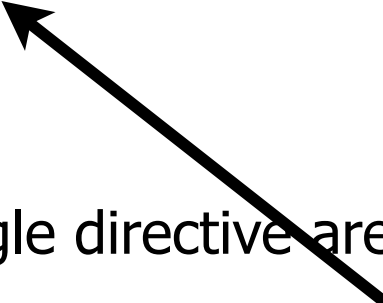
- Only one thread in the team executes the enclosed code
- The Format is:

```
#pragma omp single [nowait][clause, ..]{  
    "block"  
}
```

- The supported clauses on the single directive are:

```
private (list)  
firstprivate (list)
```

NOWAIT:  
the other threads  
will not wait at the  
end single directive



# OpenMP Master directive

```
#pragma omp master {  
    "code"  
}
```

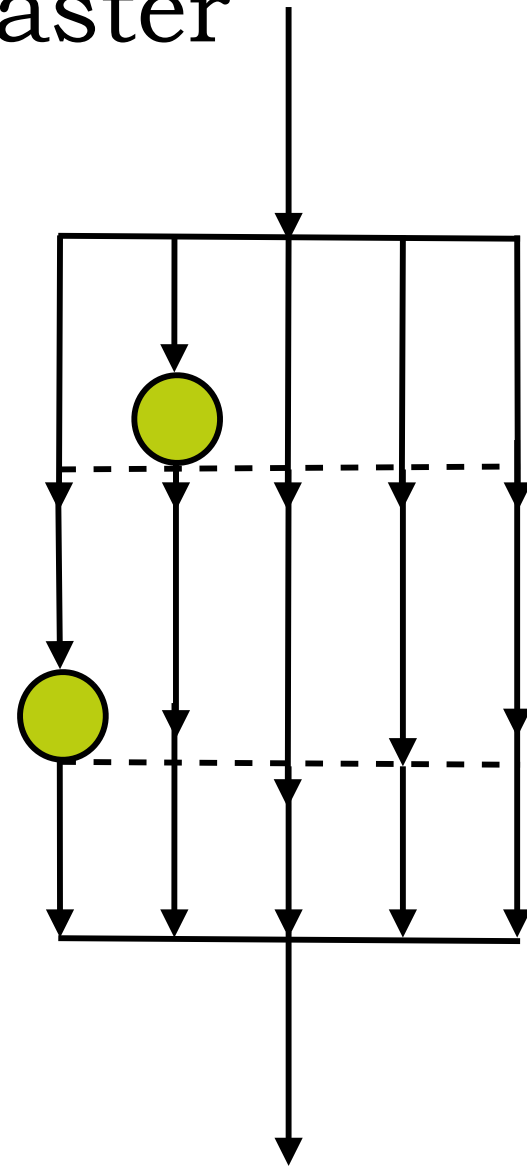
- All threads but the master, skip the enclosed section of code and continue
- There is no implicit barrier on entry or exit !

```
#pragma omp barrier
```

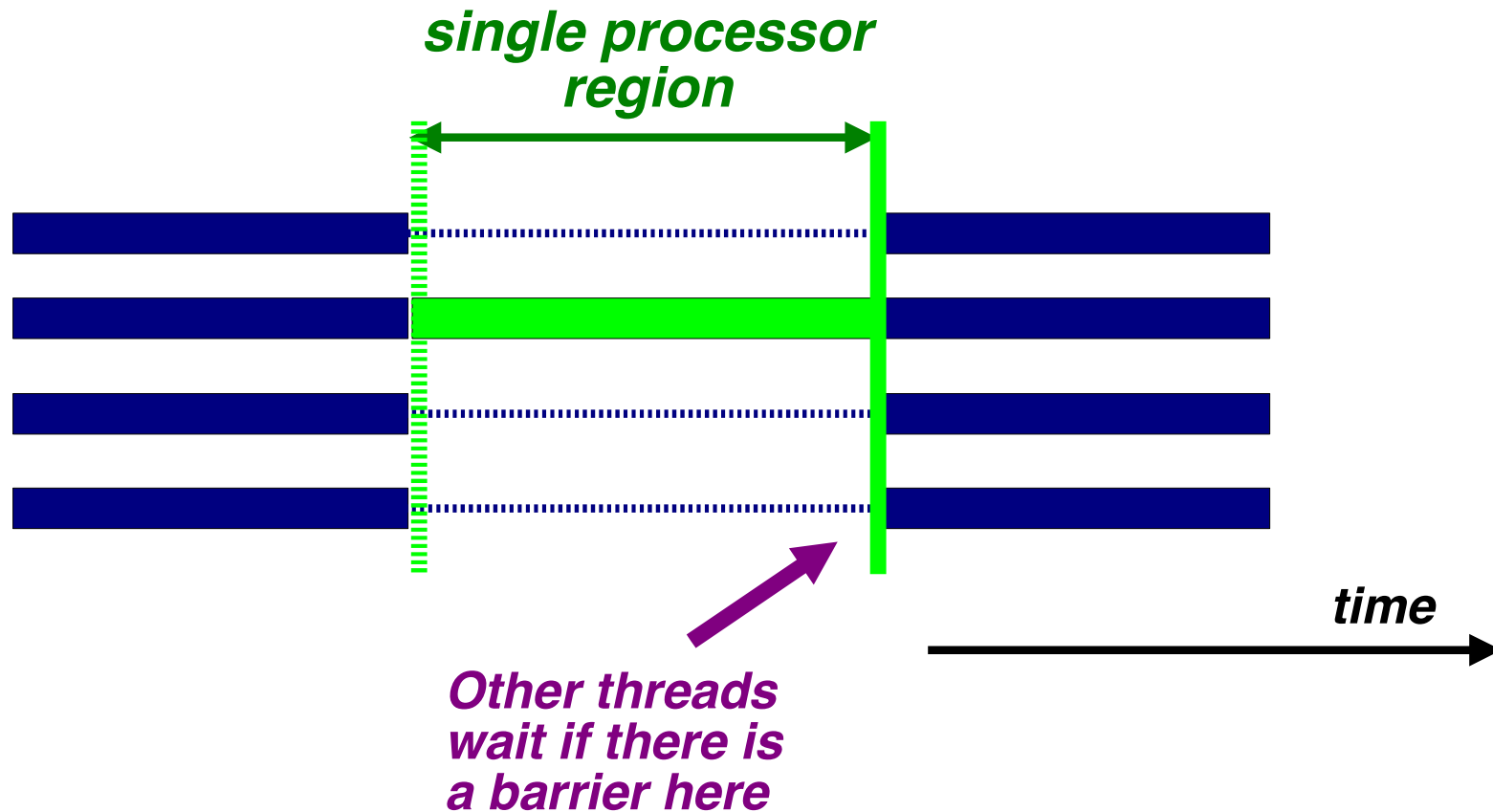
- Each thread waits until all others in the team have reached this point.

# Work Sharing: Single Master

```
#pragma omp parallel
{
    ....
    #pragma omp single [nowait]
    {
        ....
    }
    #pragma omp master
    {
        ....
    }
    ....
    #pragma omp barrier
}
```



# Single processor



# Work Sharing: Orphaning

- Worksharing constructs may be outside lexical scope of the parallel region

```
#pragma omp parallel
{
    ....
    dowork( )
    ....
}
....
```

```
void dowork( )
{
    #pragma omp for
    for (i=0; i<n; i++) {
        ....
    }
}
```

# Scheduling the work

- schedule ( static | dynamic | guided | auto [, chunk] ) schedule (runtime)

## **static [, chunk]**

- Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion
- In absence of "chunk", each thread executes approx.  $N/P$  chunks for a loop of length  $N$  and  $P$  threads

| Thread           | 0               | 1                | 2                | 3                |
|------------------|-----------------|------------------|------------------|------------------|
| <i>no chunk*</i> | 1 - 4           | 5 - 8            | 9 - 12           | 13 - 16          |
| <i>chunk = 2</i> | 1 - 2<br>9 - 10 | 3 - 4<br>11 - 12 | 5 - 6<br>13 - 14 | 7 - 8<br>15 - 16 |

- **dynamic [, chunk]**

- Fixed portions of work; size is controlled by the value of chunk
- When a thread finishes, it starts on the next portion of work

- **guided [, chunk]**

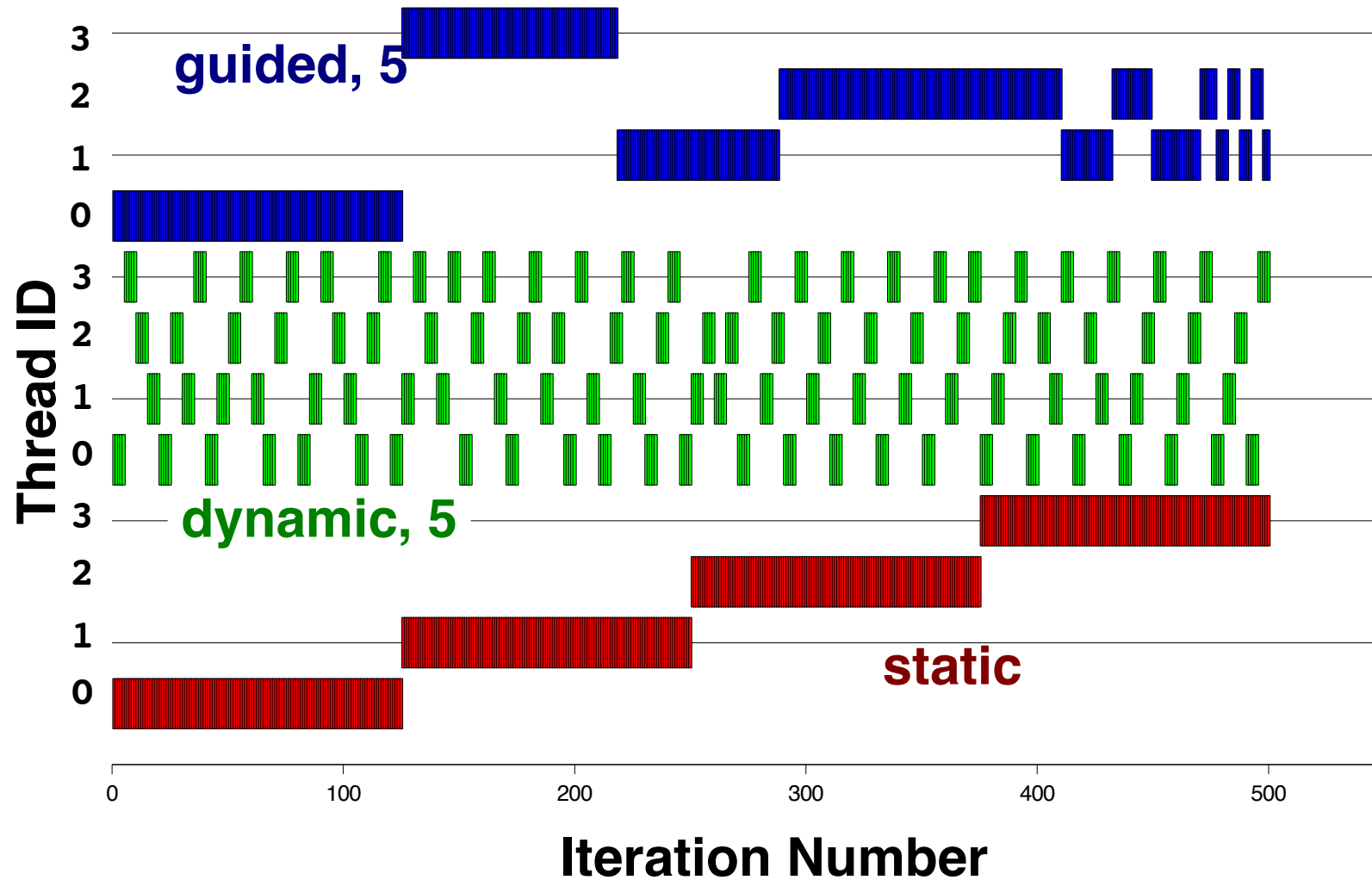
- Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially

## **runtime**

- Iteration scheduling scheme is set at runtime through environment variable OMP\_SCHEDULE

# Example scheduling

*500 iterations on 4 threads*



# Environment Variables

- The names of the OpenMP environment variables must be UPPERCASE
- The values assigned to them are case insensitive

`OMP_NUM_THREADS`

`OMP_SCHEDULE "schedule [chunk]"`

`OMP_NESTED { TRUE | FALSE }`

# Exercise: OpenMP scheduling

- How OpenMP can help parallelize a loop, but as a user you can help in making it better ;-)
- On the git clone: `cd HPCourse/OMP_schedule`
- The README has instructions:  
This works best with an Intel compiler that uses an auto-parallelizer to generate threaded code
- This exercise requires already some knowledge about OpenMP.  
The OpenMP website at <https://www.openmp.org/> can be helpful to get started.

# OpenMP Reduction Clause

```
reduction (op : list)
```

The variables in “*list*” must be shared in the enclosing parallel region

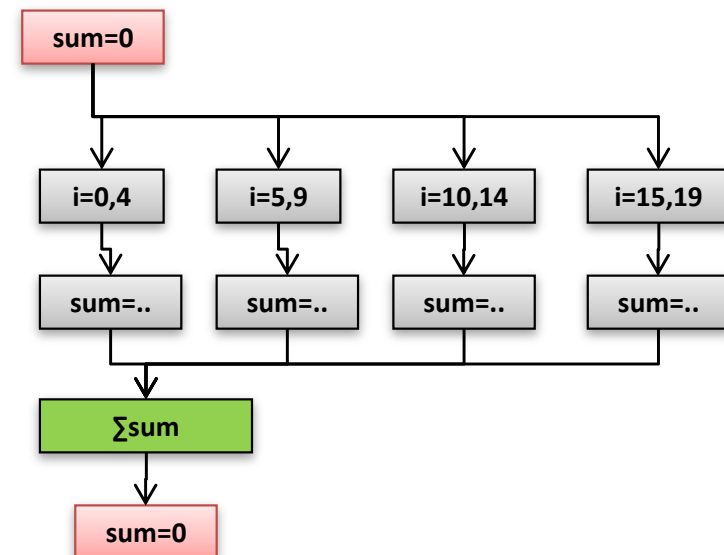
Inside parallel or work-sharing construct:

- ▶ A PRIVATE copy of each list variable is created and initialized depending on the “op”
- ▶ These copies are updated locally by threads
- ▶ At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable

# Reduction Example

```
#pragma omp parallel for  
reduction(+:sum)  
    for(i=0; i<N; i++) {  
        sum += a[i] * b[i];  
    }
```

Local copy of *sum* for each thread  
All local copies of *sum* added together  
and stored in "global" variable



# C/C++ Reduction Operations

A range of associative and commutative operators can be used with reduction

Initial values are the ones that make sense

| Operator | Initial Value |
|----------|---------------|
| +        | 0             |
| *        | 1             |
| -        | 0             |
| ^        | 0             |

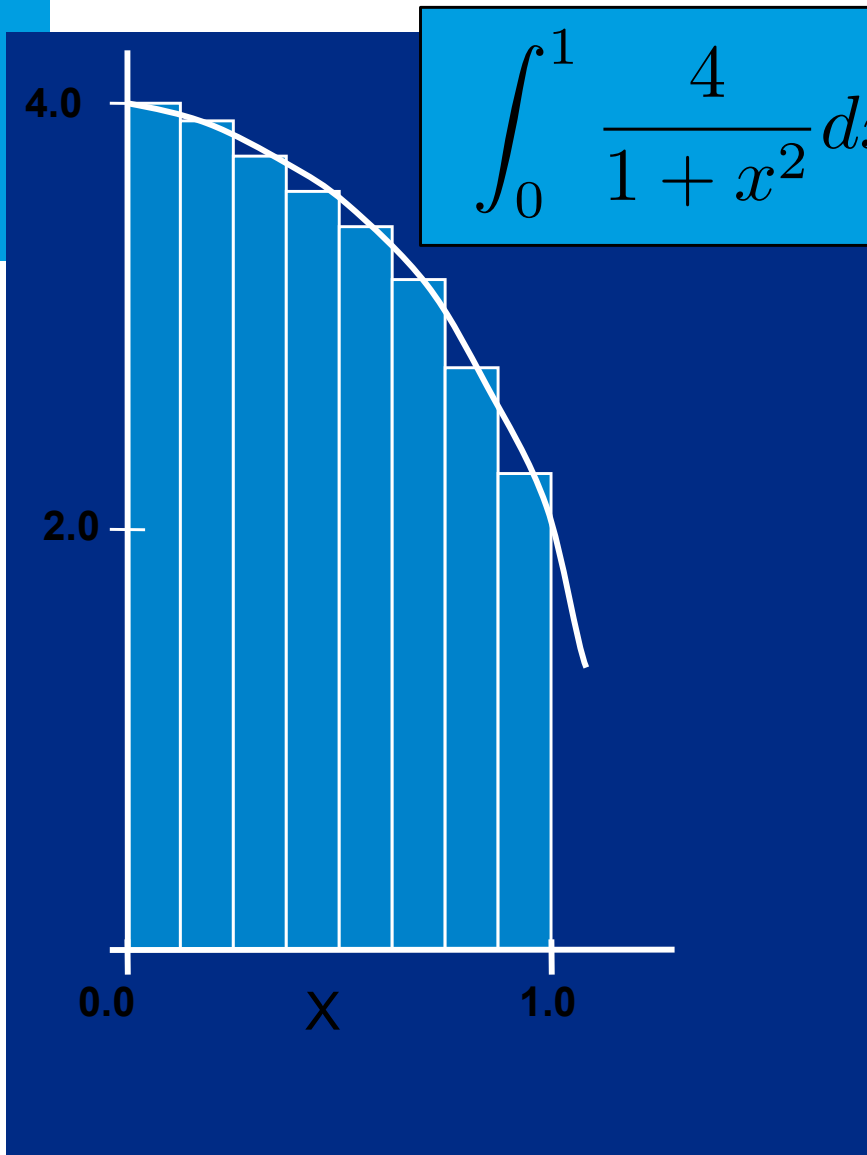
| Operator | Initial Value |
|----------|---------------|
| &        | ~0            |
|          | 0             |
| &&       | 1             |
|          | 0             |

***FORTRAN:***

*intrinsic* is one of MAX, MIN, IAND, IOR, IEOR

*operator* is one of +, \*, -, .AND., .OR., .EQV., .NEQV.

# Numerical Integration Example



$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

```
static long num_steps=100000;  
double step, pi;
```

```
void main()  
{ int i;  
  double x, sum = 0.0;  
  
  step = 1.0/(double) num_steps;  
  for (i=0; i<num_steps; i++){  
    x = (i+0.5)*step;  
    sum = sum + 4.0/(1.0 + x*x);  
  }  
  pi = step * sum;  
  printf("Pi = %f\n",pi);  
}
```

# Numerical Integration to Compute Pi

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
   for (i=0; i<num_steps; i++){
       x = (i+0.5)*step;
       sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

Parallelize the numerical integration code using OpenMP

What variables can be shared?

**step, num\_steps**

What variables need to be private?

**x, i**

What variables should be set up for reductions?

**sum**

# Solution to Computing Pi

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;
   step = 1.0/(double) num_steps;
   #pragma omp parallel for private(x) reduction(+:sum)
   for (i=0; i<num_steps; i++){
       x = (i+0.5)*step;
       sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

# Let's try it out

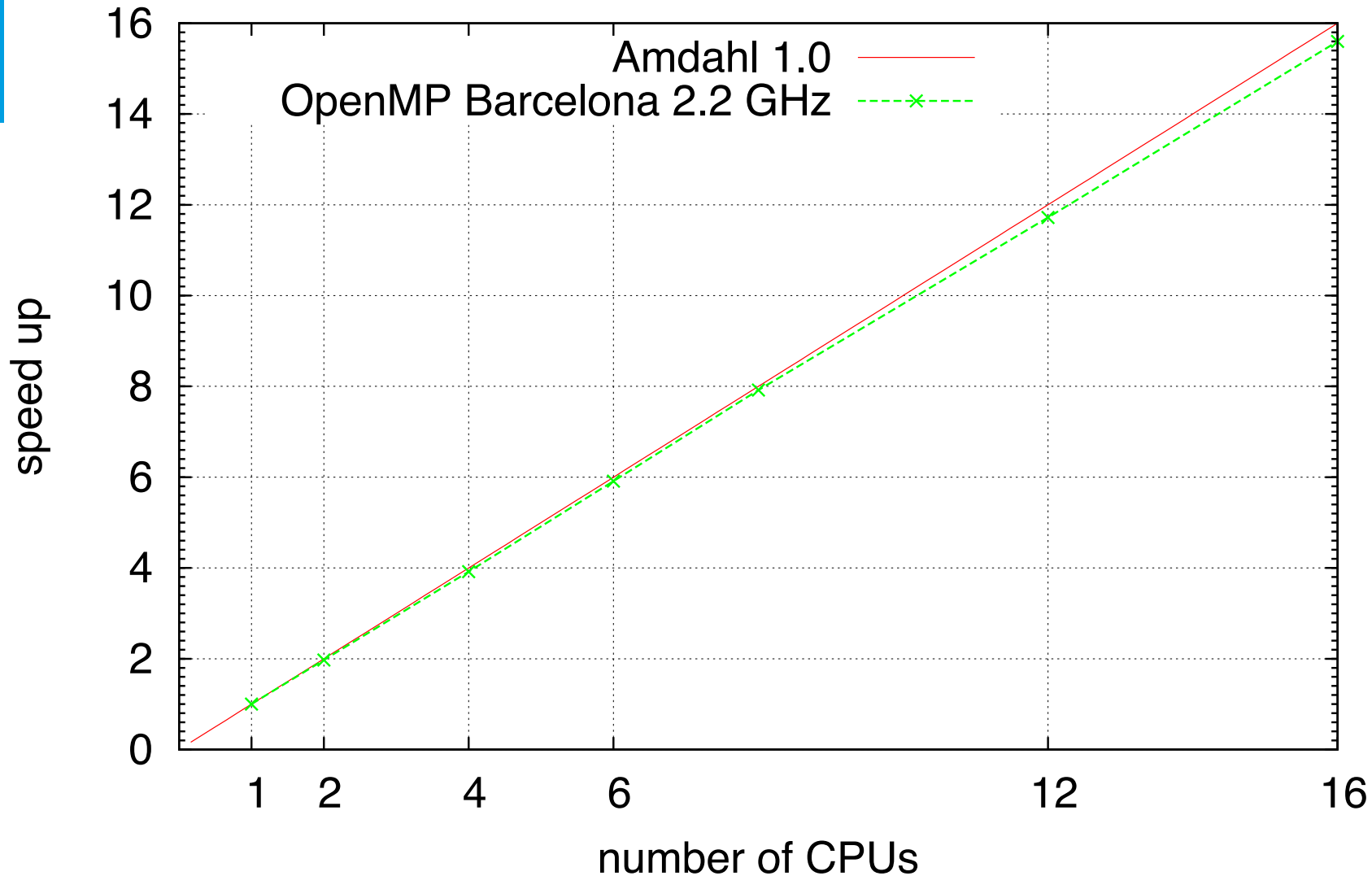
- Go to example MPI\_pi and will work with openmp\_pi2.c

# Exercise: PI with MPI and OpenMP

| cores | OpenMP   |
|-------|----------|
| 1     | 9.617728 |
| 2     | 4.874539 |
| 4     | 2.455036 |
| 6     | 1.627149 |
| 8     | 1.214713 |
| 12    | 0.820746 |
| 16    | 0.616482 |

# Exercise: PI with MPI and OpenMP

Pi Scaling



# Cuda Computing PI

```
float step = 1.0f / (float)NSET;
float sum = 0.0f;
```

```
PiSimple2<<<GRIDDIM, BLOCKDIM>>>
    (d_partials, step, NSET);
CUT_CHECK_ERROR("***PiSimple2
    execution failed!!!***");
```

```
CUDA_SAFE_CALL(cudaMemcpy(h_partia
    ls, d_partials,
    fSmallArraySize,
    cudaMemcpyDeviceToHost) );
```

```
for (j = 0; j < GRIDDIM; j++)
{
    sum += h_partials[j];
}
Pi = step * sum;
```

```
__global__ void
PiSimple2( float* g_partialOut, float step,
    int NSamples)
{
    const int tid = blockDim.x * blockIdx.x +
        threadIdx.x;
    const int blocksize = blockDim.x;
    const int THREAD_N = blockDim.x * gridDim.x;
    float x, partialsum = 0.0f;
```

```
for(int i = tid; i < NSamples; i += THREAD_N){
    x = (i * 0.5f)*step;
    partialsum = partialsum + 4.0f / (1.0f
        + x*x);
}
```

```
__shared__ float threadsum[BLOCKDIM];
threadsum[threadIdx.x] = partialsum;
```

```
__syncthreads();
float blocksum = 0;
if (threadIdx.x == 0) {
    const int blockindex = blockIdx.x;
    for (int i = 0; i < blocksize; i++)
        blocksum += threadsum[i];
    g_partialOut[blockindex] = blocksum;
}
```

```
}
__device__ float p_blockindex = blockIdx.x;
p_blocksum += p_threadsum[i];
for (int i = 0; i < p_blocksize; i++)
```

# Exercise: Shared Cache Trashing

- Let's do the exercise: CacheTrash

# About local and shared data

- Consider the following example:

```
for (i=0; i<10; i++){  
    a[i] = b[i] + c[i];  
}
```

- Let's assume we run this on 2 processors:
  - processor 1 for i=0,2,4,6,8
  - processor 2 for i=1,3,5,7,9

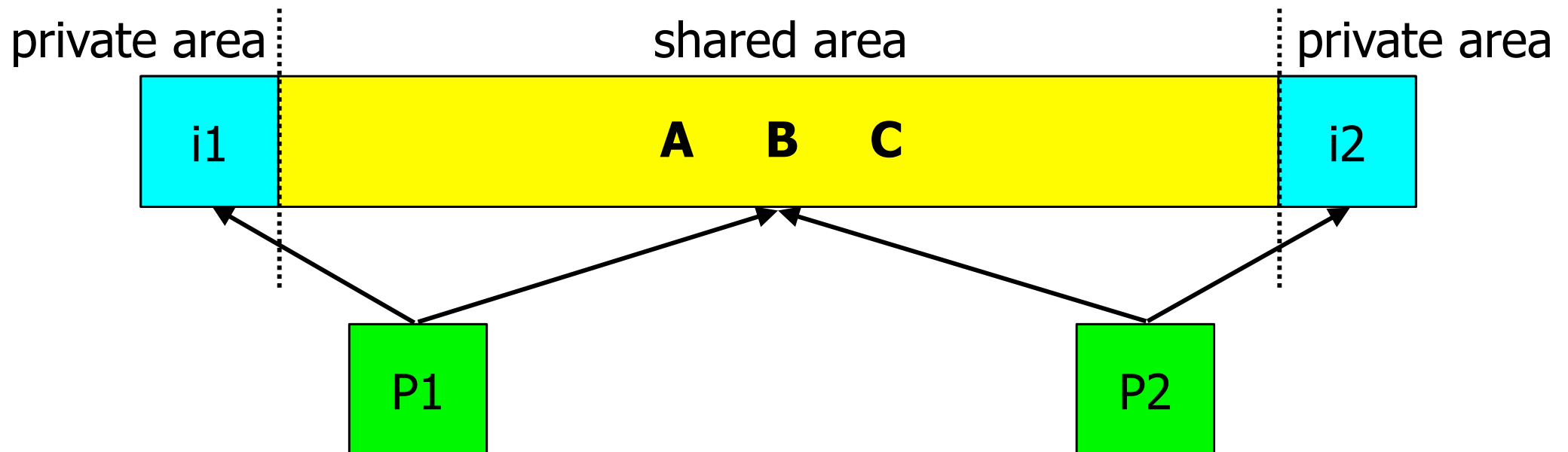
# About local and shared data

Processor 1

```
for (i1=0,2,4,6,8){  
    a[i1] = b[i1] + c[i1];  
}
```

Processor 2

```
for (i2=1,3,5,7,9){  
    a[i2] = b[i2] + c[i2];  
}
```



# About local and shared data

processor 1 for  $i=0,2,4,6,8$

processor 2 for  $i=1,3,5,7,9$

- This is not an efficient way to do this!

## Why?

# Doing it the bad way

- Because of cache line usage

```
for (i=0; i<10; i++){  
    a[i] = b[i] + c[i];  
}
```

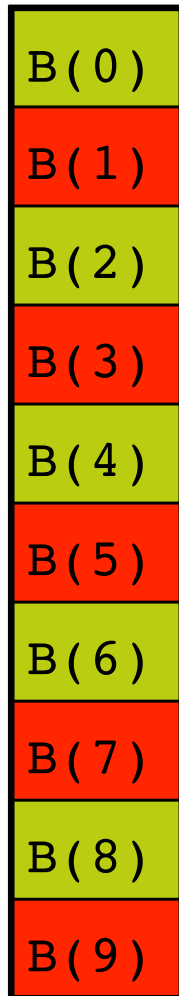
- b[] and c[]: we use half of the data
- a[]: false sharing

# False sharing and scalability

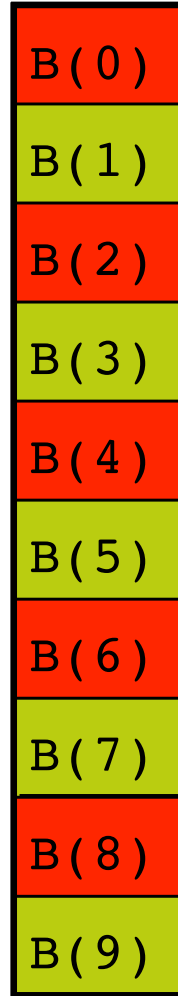
- The Cause:  
Updates on independent data elements that happen to be part of the same cache line.
- The Impact:  
Non-scalable parallel applications
- The Remedy:  
False sharing is often quite simple to solve

# Poor cache line utilization

Processor 1



Processor 2



cache line



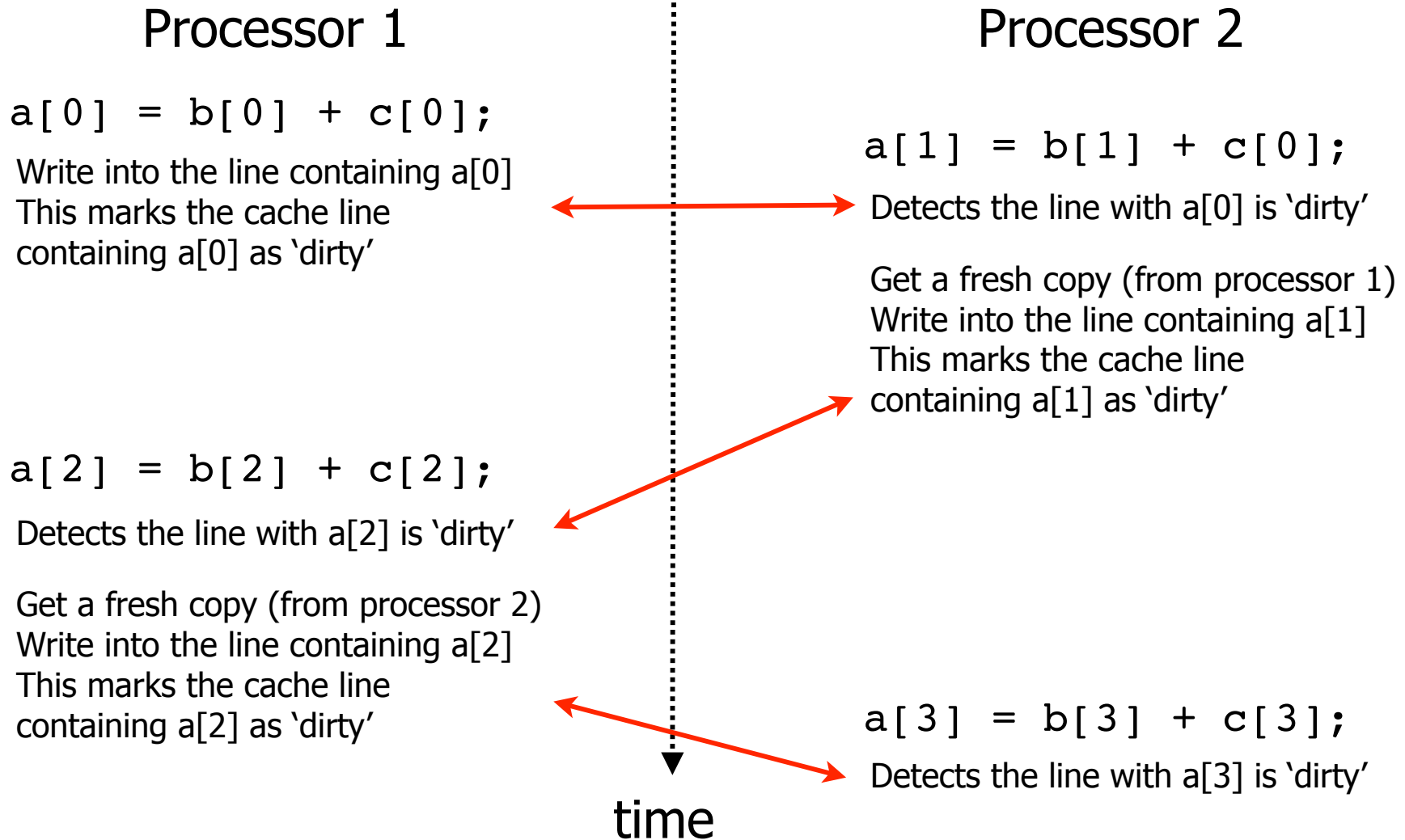
used data

not used data

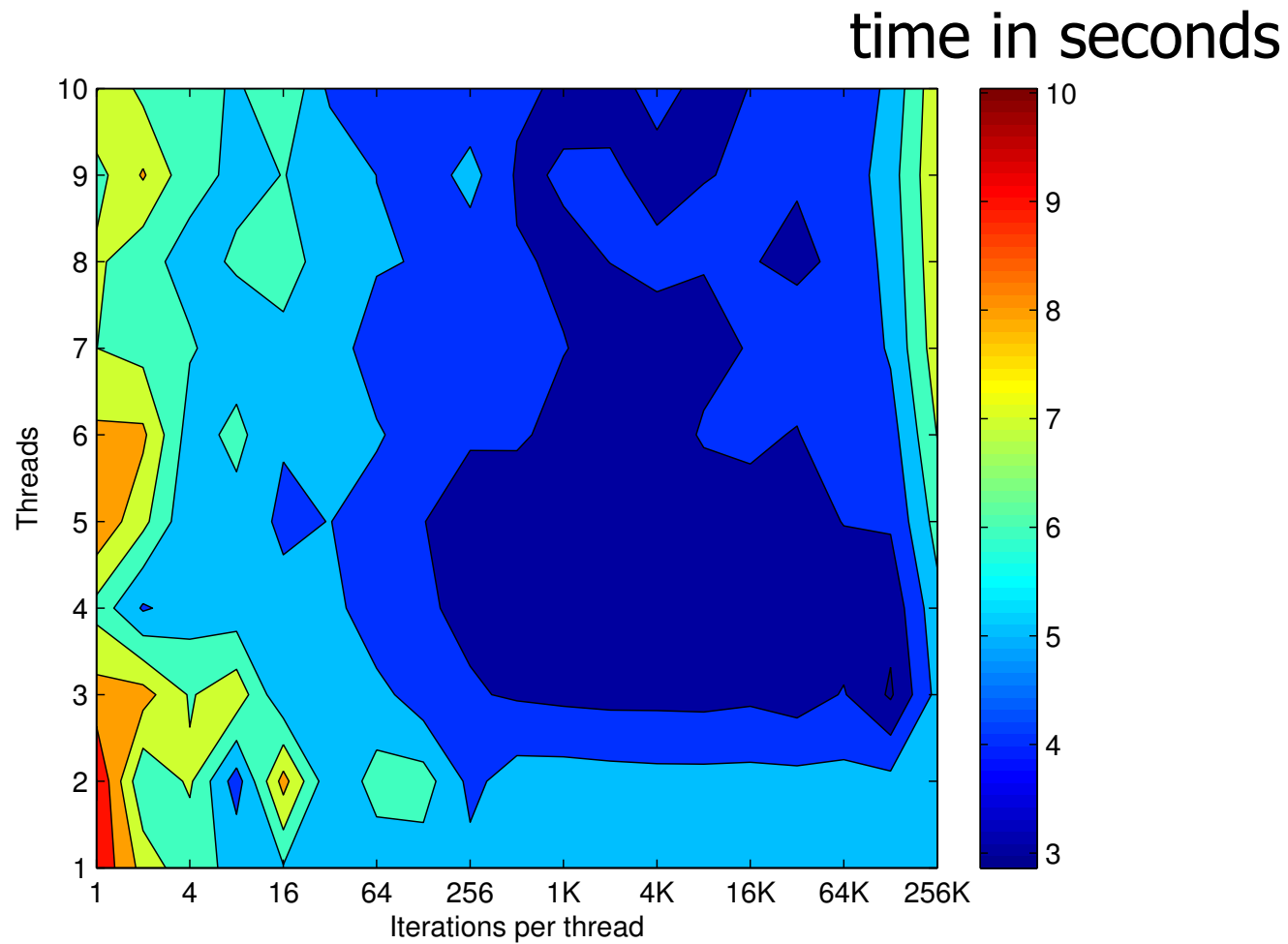
Both processors read  
the same cache lines

The same holds for array C

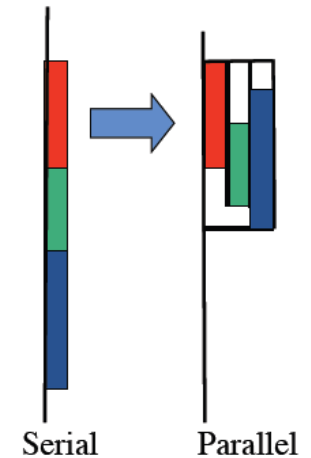
# False Sharing



# False Sharing results



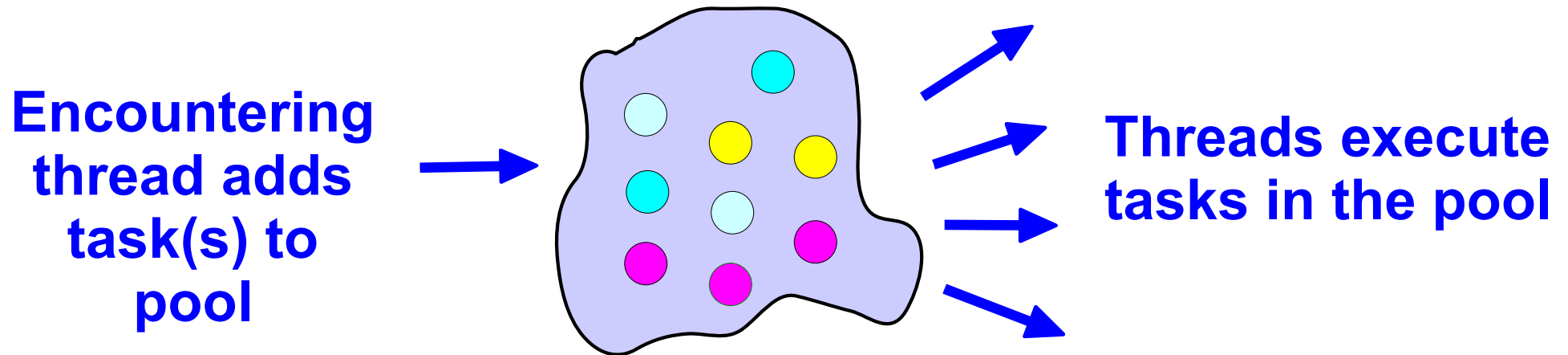
# OpenMP tasks



- What are tasks
  - Tasks are independent units of work
  - Threads are assigned to perform the work of each task.
    - Tasks may be deferred
    - Tasks may be executed immediately
    - The runtime system decides which of the above
- Why tasks?
  - The basic idea is to set up a task queue: when a thread encounters a task directive, it arranges for some thread to execute the associated block at some time. The first thread can continue.

# The Tasking Example

122



***Developer specifies tasks in application  
Run-time system executes tasks***

# OpenMP tasks

Tasks allow to parallelize irregular problems

- Unbounded loops
- Recursive algorithms
- Manager/work schemes

A task has

- Code to execute
- Data environment (It owns its data)
- Internal control variables
- An assigned thread that executes the code and the data

OpenMP has always had tasks, but they were not called “task”.

- A thread encountering a parallel construct, e.g., “for”, packages up a set of implicit tasks, one per thread.
- A team of threads is created.
- Each thread is assigned to one of the tasks.
- Barrier holds master thread till all implicit tasks are finished.

# OpenMP tasks

`#pragma omp parallel` -> A parallel region creates a team of threads;

`#pragma omp single`

{

...

`#pragma omp task`

{ ... }

...

`#pragma omp taskwait`

}

-> One thread enters the execution

-> pick up threads „from the work queue“

-> the other threads wait at the end of the single

# Summary

- First tune single-processor performance
- Tuning parallel programs
  - Has the program been properly parallelized?
    - Is enough of the program parallelized (Amdahl's law)?
    - Is the load well-balanced?
  - location of memory
    - Cache friendly programs: no special placement needed
    - Non-cache friendly programs
      - False sharing?
  - Use of OpenMP
    - try to avoid synchronization (`barrier`, `critical`, `single`, `ordered`)

# Plenty of Other OpenMP Stuff

Scheduling clauses

Atomic

Barrier

Master & Single

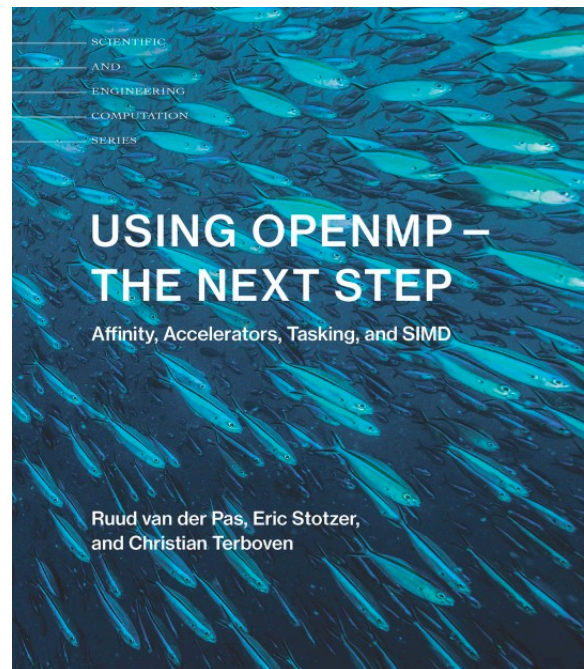
Sections

Tasks (OpenMP 3.0)

API routines

# OpenMP references

<https://mitpress.mit.edu/books/using-openmp-next-step>



## Paperback

\$50.00 S | £40.00

ISBN: 9780262534789

392 pp. | 8 in x 9 in

250 b&w illus.

October 2017

# Compiling and running OpenMP

- Compile with -openmp flag (intel compiler) or -fopenmp (GNU)
- Run program with variable:

```
export OMP_NUM_THREADS=4
```



# OpenACC

- Set of directives to support accelerators
  - Developed by PGI and HPE-Cray:
  - Used to support all vendors: PGI now part of Nvidia
  - Intel's MIC's not supported anymore
  - AMD Fusions processors: only support by HPE-Cray
- OpenMP 5.0 also aims at GPU's and real open standards
  - might replace OpenACC in the future.
  - not yet same compiler support of all vendors: Intel? AMD?

# OpenACC example

```
void convolution_SM_N(typeToUse A[M][N], typeToUse B[M][N]) {
    int i, j, k;
    int m=M, n=N;
    // OpenACC kernel region
    // Define a region of the program to be compiled into a sequence of kernels
    // for execution on the accelerator device
    #pragma acc kernels pcopyin(A[0:m]) pcopy(B[0:m])
    {
        typeToUse c11, c12, c13, c21, c22, c23, c31, c32, c33;

        c11 = +2.0f;  c21 = +5.0f;  c31 = -8.0f;
        c12 = -3.0f;  c22 = +6.0f;  c32 = -9.0f;
        c13 = +4.0f;  c23 = +7.0f;  c33 = +10.0f;

        // The OpenACC loop gang clause tells the compiler that the iterations of the loops
        // are to be executed in parallel across the gangs.
        // The argument specifies how many gangs to use to execute the iterations of this loop.
        #pragma acc loop gang(64)
        for (int i = 1; i < M - 1; ++i) {

            // The OpenACC loop worker clause specifies that the iteration of the associated loop are
            // to be
            // executed in parallel across the workers within the gangs created.
            // The argument specifies how many workers to use to execute the iterations of this loop.
            #pragma acc loop worker(128)
            for (int j = 1; j < N - 1; ++j) {
                B[i][j] = c11 * A[i - 1][j - 1] + c12 * A[i + 0][j - 1] + c13 * A[i + 1][j - 1]
                    + c21 * A[i - 1][j + 0] + c22 * A[i + 0][j + 0] + c23 * A[i + 1][j + 0]
                    + c31 * A[i - 1][j + 1] + c32 * A[i + 0][j + 1] + c33 * A[i + 1][j + 1];
            }
        }
    }
}
```

# MPI

# Message Passing

- Point-to-Point
- Requires explicit commands in program
  - Send, Receive
- Must be synchronized among different processors
  - Sends and Receives must match
  - Avoid Deadlock -- all processors waiting, none able to communicate
- Multi-processor communications
  - e.g. broadcast, reduce

# MPI advantages

- Mature and well understood
  - Backed by widely-supported formal standard (1992)
  - Porting is “easy”
- Efficiently matches the hardware
  - Vendor and public implementations available
- User interface:
  - Efficient and simple
  - Buffer handling
  - Allow high-level abstractions
- Performance

# MPI disadvantages

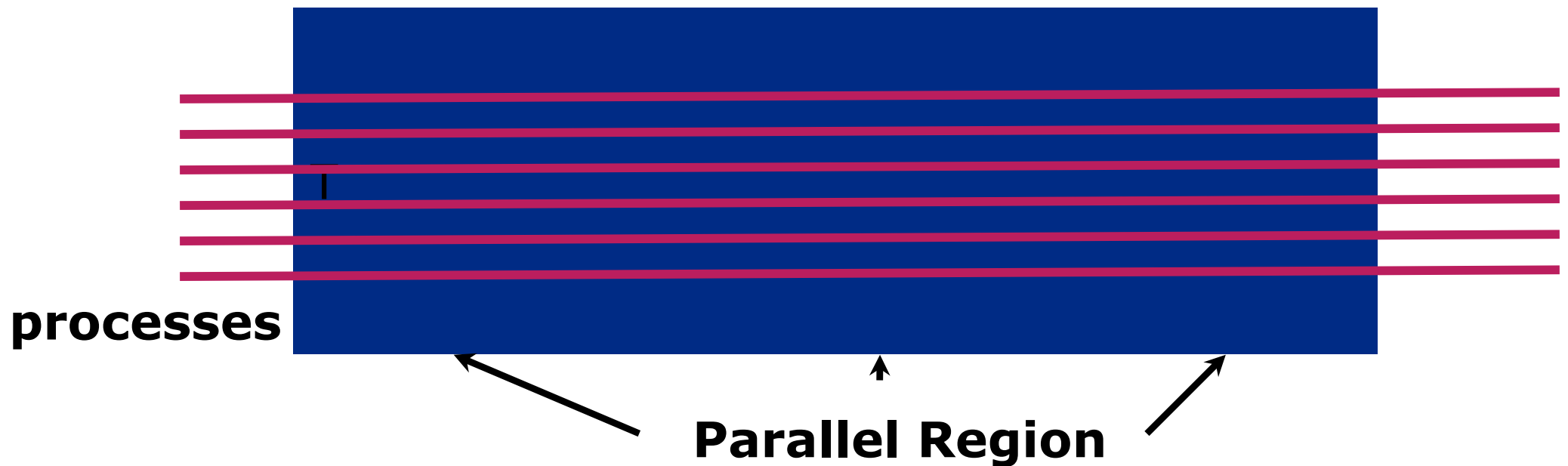
- MPI 2.0 includes many features beyond message passing



- Execution control environment depends on implementation

# Programming Model

- Explicit parallelism:
  - ▶ All processes starts at the same time at the same point in the code
  - ▶ Full parallelism: there is no sequential part in the program



# Work Distribution

- All processors run the same executable.
- Parallel work distribution must be explicitly done by the programmer:
  - domain decomposition
  - master worker

# A Minimal MPI Program (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

# A Minimal MPI Program (Fortran 90)

```
program main
use MPI
integer ierr

call MPI_INIT( ierr )
print *, 'Hello, world!'
call MPI_FINALIZE( ierr )
end
```

# Starting the MPI Environment

- **MPI\_INIT ( )**

Initializes MPI environment. This function must be called and must be the first MPI function called in a program (exception: **MPI\_INITIALIZED**)

## Syntax

```
int MPI_Init ( int *argc, char ***argv )
```

```
MPI_INIT ( IERROR )
```

```
INTEGER IERROR
```

NOTE: Both C and Fortran return error codes for all calls.

# Exiting the MPI Environment

- **MPI\_FINALIZE ( )**

Cleans up all MPI state. Once this routine has been called, no MPI routine ( even **MPI\_INIT** ) may be called

## **Syntax**

```
int MPI_Finalize ( );
```

```
MPI_FINALIZE ( IERROR )
```

```
INTEGER IERROR
```

**MUST** call MPI\_FINALIZE when you exit from an MPI program

# C and Fortran Language Considerations

- Bindings
  - C
    - All MPI names have an **MPI\_** prefix
    - Defined constants are in all capital letters
    - Defined types and functions have one capital letter after the prefix; the remaining letters are lowercase
  - Fortran
    - All MPI names have an **MPI\_** prefix
    - No capitalization rules apply
    - last argument is an returned error value

# Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **MPI\_Comm\_size** reports the number of processes.
  - **MPI\_Comm\_rank** reports the *rank*, a number between 0 and size-1, identifying the calling process

# Better Hello (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

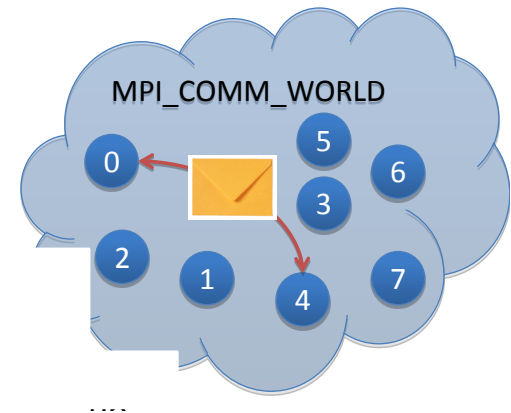
# Better Hello (Fortran)

```
program main
use MPI
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

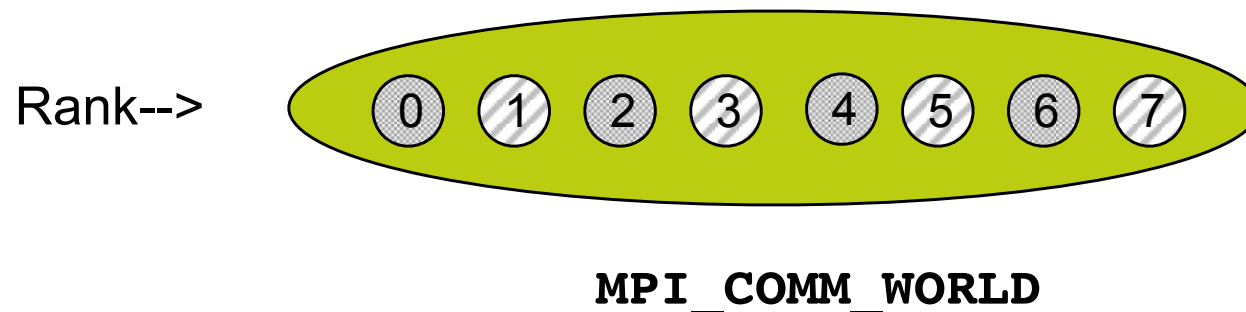
# Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **MPI\_COMM\_WORLD**.



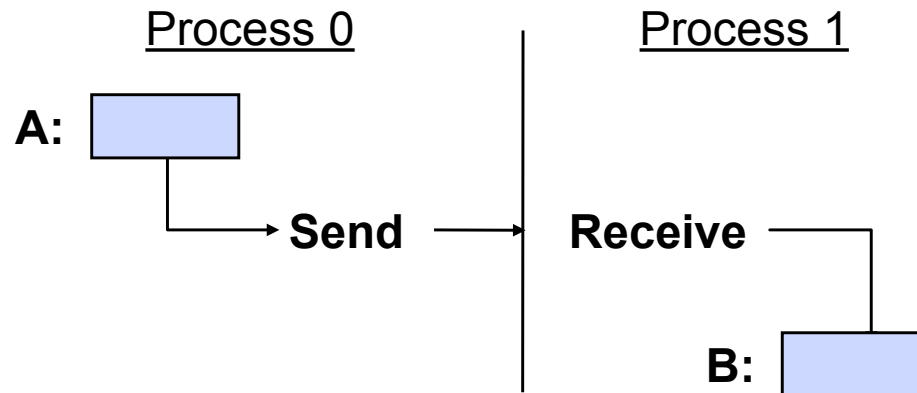
# Communicator

- Communication in **MPI** takes place with respect to communicators
- **MPI\_COMM\_WORLD** is one such predefined communicator (something of type "**MPI\_COMM**") and contains group and context information
- **MPI\_COMM\_RANK** and **MPI\_COMM\_SIZE** return information based on the communicator passed in as the first argument
- Processes may belong to many different communicators



# MPI Basic Send/Receive

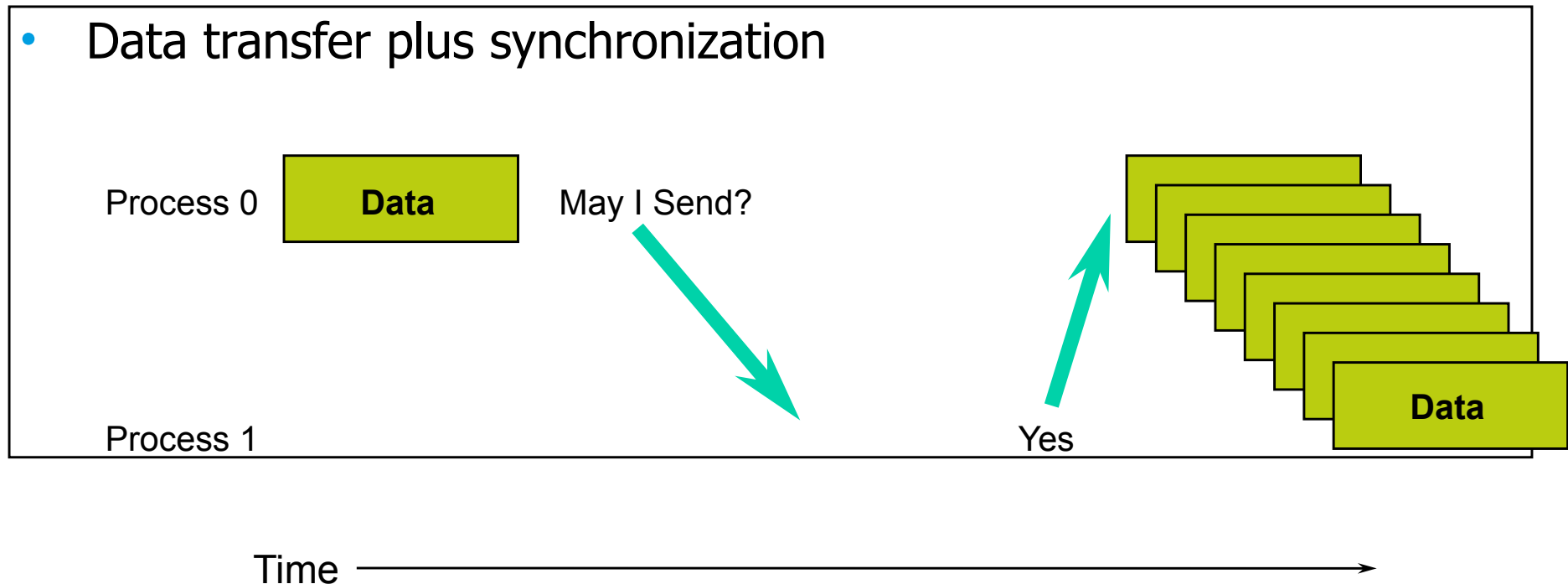
- Basic message passing process. Send data from one process to another



- Questions
  - To whom is data sent?
  - Where is the data?
  - What type of data is sent?
  - How much of data is sent?
  - How does the receiver identify it?

# MPI Basic Send/Receive

- Data transfer plus synchronization



- Requires co-operation of sender and receiver
- Co-operation not always apparent in code
- Communication and synchronization are combined

# Message Organization in MPI

- Message is divided into data and envelope

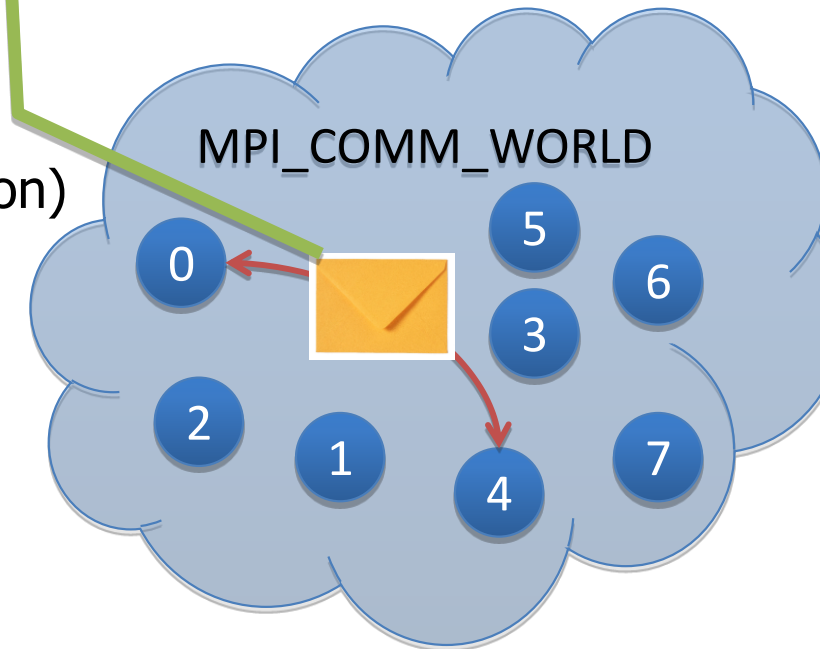
- data

- buffer
- count
- datatype

- envelope

- process identifier (source/destination)
- message tag
- communicator

Source : process0  
Destination : process1  
Tag : 1234  
Communicator : MPI\_COMM\_WORLD



# MPI Basic Send/Receive

- Thus the basic (blocking) send has become:

```
MPI_Send ( start, count, datatype, dest, tag,  
comm )
```

- Blocking means the function does not return until it is safe to reuse the data in buffer. The message may not have been received by the target process.

- And the receive has become:

```
MPI_Recv( start, count, datatype, source, tag,  
comm, status )
```

- The source, tag, and the count of the message actually received can be retrieved from `status`

# MPI C Datatypes

| MPI datatype                    | C datatype                      |
|---------------------------------|---------------------------------|
| <code>MPI_CHAR</code>           | <code>signed char</code>        |
| <code>MPI_SHORT</code>          | <code>signed short int</code>   |
| <code>MPI_INT</code>            | <code>signed int</code>         |
| <code>MPI_LONG</code>           | <code>signed long int</code>    |
| <code>MPI_UNSIGNED_CHAR</code>  | <code>unsigned char</code>      |
| <code>MPI_UNSIGNED_SHORT</code> | <code>unsigned short int</code> |
| <code>MPI_UNSIGNED_LONG</code>  | <code>unsigned long_int</code>  |
| <code>MPI_UNSIGNED</code>       | <code>unsigned int</code>       |
| <code>MPI_FLOAT</code>          | <code>float</code>              |
| <code>MPI_DOUBLE</code>         | <code>double</code>             |
| <code>MPI_LONG_DOUBLE</code>    | <code>long double</code>        |
| <code>MPI_BYTE</code>           |                                 |
| <code>MPI_PACKED</code>         |                                 |

# MPI Fortran Datatypes

| <b>MPI FORTRAN</b>          | <b>FORTRAN datatypes</b> |
|-----------------------------|--------------------------|
| <b>MPI_INTEGER</b>          | <b>INTEGER</b>           |
| <b>MPI_REAL</b>             | <b>REAL</b>              |
| <b>MPI_REAL8</b>            | <b>REAL*8</b>            |
| <b>MPI_DOUBLE_PRECISION</b> | <b>DOUBLE PRECISION</b>  |
| <b>MPI_COMPLEX</b>          | <b>COMPLEX</b>           |
| <b>MPI_LOGICAL</b>          | <b>LOGICAL</b>           |
| <b>MPI_CHARACTER</b>        | <b>CHARACTER</b>         |
| <b>MPI_BYTE</b>             |                          |
| <b>MPI_PACKED</b>           |                          |

# Is MPI Large or Small?

- Is MPI large (128 functions) or small (6 functions)?
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
  - Many programs can be written with just 6 basic functions

**MPI\_INIT**

**MPI\_COMM\_SIZE**

**MPI\_SEND**

**MPI\_FINALIZE**

**MPI\_COMM\_RANK**

**MPI\_RECV**

- MPI is just right
  - A small number of concepts
  - Large number of functions provides flexibility, robustness, efficiency, modularity, and convenience
  - One need not master all parts of MPI to use it

# Example: PI in Fortran 90 and C

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
        h = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += 4.0 / (1.0 + x*x);
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if (myid == 0)
            printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    }
    MPI_Finalize();
    return 0;
}
```

work distribution

```

program main
use MPI
double precision PI25DT
parameter (PI25DT = 3.141592653589793238462643d0)
double precision mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, ierr

c                                     function to integrate

f(a) = 4.d0 / (1.d0 + a*a)
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
10  if ( myid .eq. 0 ) then
    write(6,98)
98   format('Enter the number of intervals: (0 quits)')
    read(5,'(i10)') n
endif
call MPI_BCAST( n, 1, MPI_INTEGER, 0,MPI_COMM_WORLD, ierr)
if ( n .le. 0 ) goto 30
h = 1.0d0/n
sum = 0.0d0
do 20 i = myid+1, n, numprocs
    x = h * (dble(i) - 0.5d0)
    sum = sum + f(x)
20  continue
mypi = h * sum
call MPI_REDUCE( mypi, pi, 1, MPI_DOUBLE_PRECISION,
+               MPI_SUM, 0, MPI_COMM_WORLD,ierr)
if (myid .eq. 0) then
    write(6, 97) pi, abs(pi - PI25DT)
97   format(' pi is approximately: ', F18.16,
+          ' Error is: ', F18.16)
endif
goto 10
30  call MPI_FINALIZE(ierr)
end

```

work distribution

# Exercise: PI with MPI and OpenMP

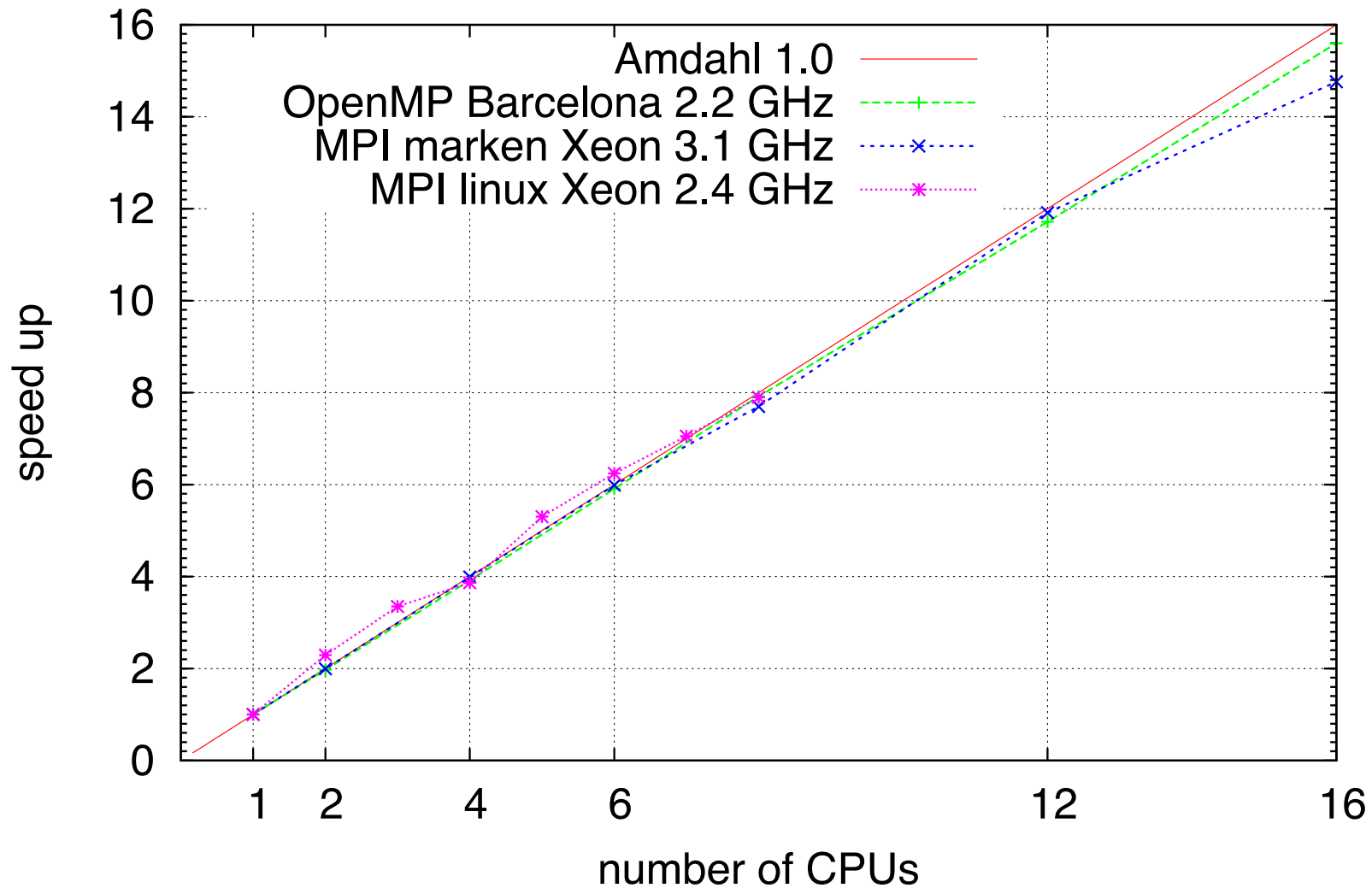
- Compile and run for computing PI parallel
- From git `HPCourse/MPi_pi`
- There is a README with instructions.
- It is assumed that you use `mpich1` and has PBS installed as a job scheduler. If you have `mpich2` let me know.
- Use `qsub` to submit the mpi job (an example script is provided) to a queue.

# Exercise: PI with MPI and OpenMP

| cores | OpenMP   | marken   | linux    |
|-------|----------|----------|----------|
| 1     | 9.617728 | 14.10798 | 22.15252 |
| 2     | 4.874539 | 7.071287 | 9.661745 |
| 4     | 2.455036 | 3.532871 | 5.730912 |
| 6     | 1.627149 | 2.356928 | 3.547961 |
| 8     | 1.214713 | 1.832055 | 2.804715 |
| 12    | 0.820746 | 1.184123 |          |
| 16    | 0.616482 | 0.955704 |          |

# Exercise: PI with MPI and OpenMP

Pi Scaling





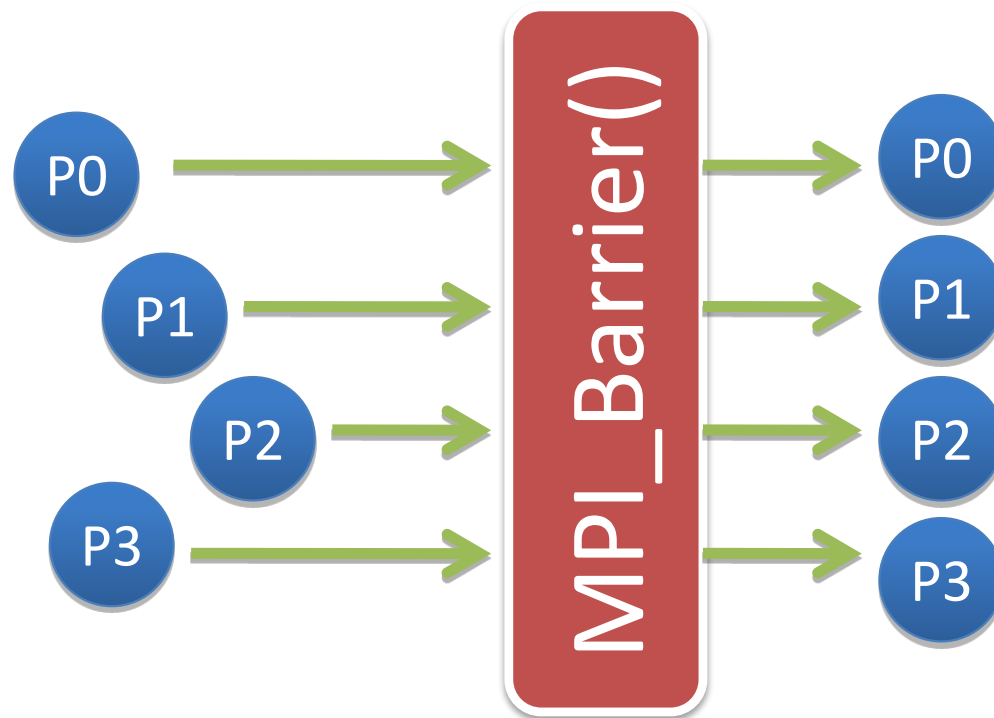
# Collective Communications in MPI

- Communication is co-ordinated among a group of processes, as specified by communicator, not on all processes
- All collective operations are blocking and no message tags are used (in MPI-1)
- All processes in the communicator group must call the collective operation
- Collective and point-to-point messaging are separated by different “contexts”
- Three classes of collective operations
  - Data movement
  - Collective computation
  - Synchronization

# Collective calls

- MPI has several collective communication calls, the most frequently used are:
  - Synchronization
    - Barrier
  - Communication
    - Broadcast
    - Gather Scatter
    - All Gather
  - Reduction
    - Reduce
    - All Reduce

# MPI\_Barrier()



Creates barrier synchronization in a communicator group comm. Each process, when reaching the MPI\_Barrier call, blocks until all the processes in the group reach the same MPI\_Barrier call.

# MPI Basic Collective Operations

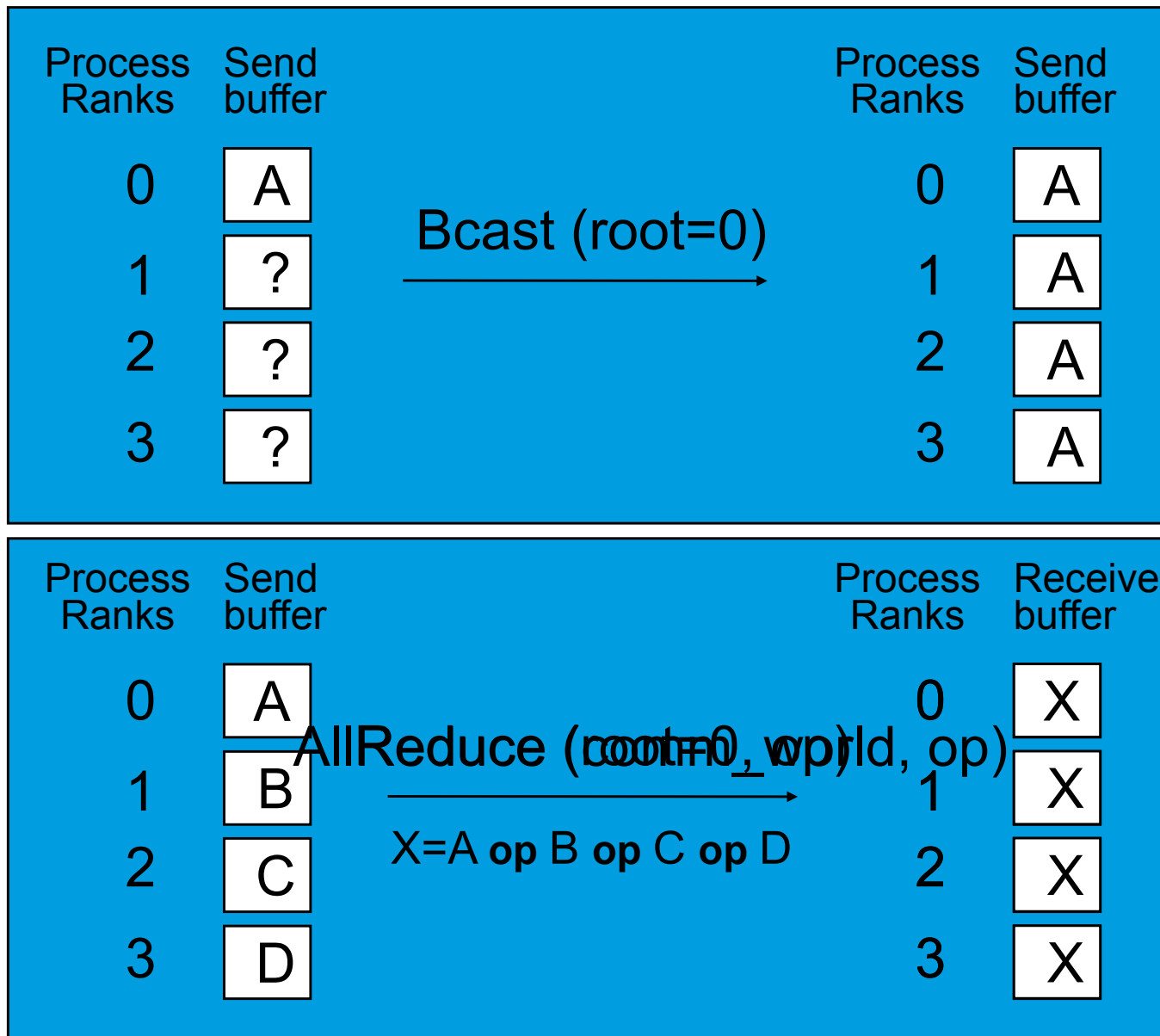
- Two simple collective operations

```
MPI_BCAST( start, count, datatype, root, comm )
```

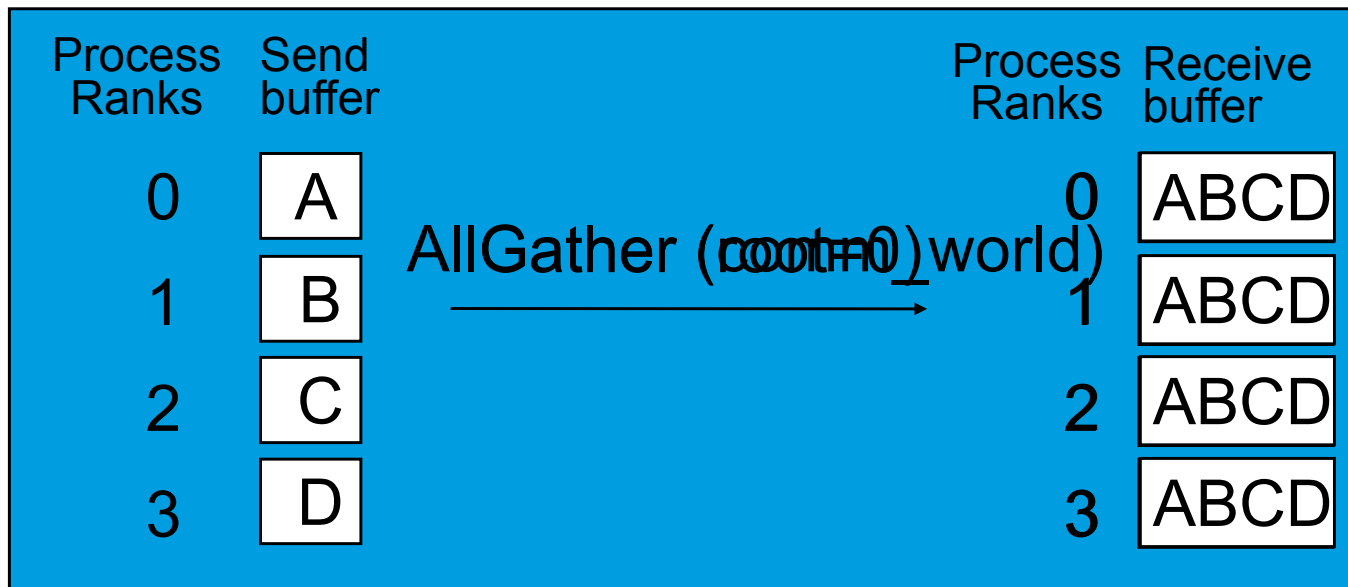
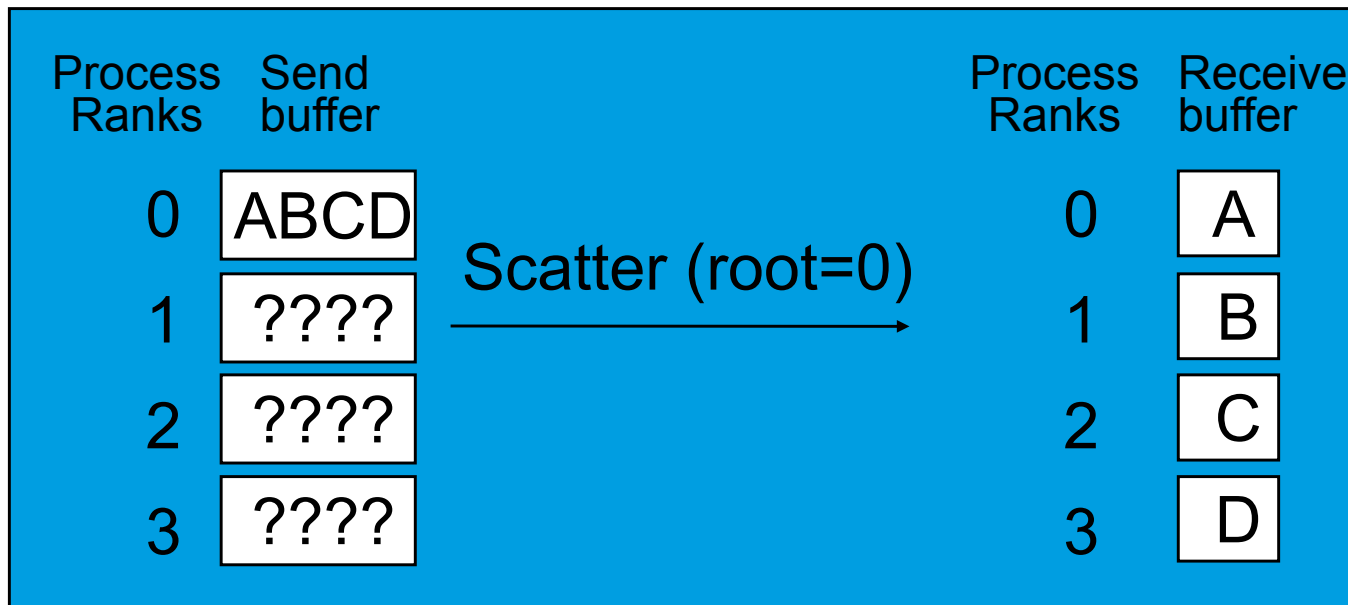
```
MPI_REDUCE( start, result, count, datatype, operation,  
             root, comm )
```

- The routine **MPI\_BCAST** sends data from one process to all others
- The routine **MPI\_REDUCE** combines data from all processes, using a specified operation, and returns the result to a single process
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.

# Broadcast and Reduce



# Scatter and Gather



# MPI Collective Routines

- Several routines:

|                           |                       |                      |
|---------------------------|-----------------------|----------------------|
| <b>MPI_ALLGATHER</b>      | <b>MPI_ALLGATHERV</b> | <b>MPI_BCAST</b>     |
| <b>MPI_ALLTOALL</b>       | <b>MPI_ALLTOALLV</b>  |                      |
| <b>MPI_GATHER</b>         | <b>MPI_GATHERV</b>    |                      |
| <b>MPI_REDUCE_SCATTER</b> | <b>MPI_REDUCE</b>     | <b>MPI_ALLREDUCE</b> |
| <b>MPI_SCATTERV</b>       | <b>MPI_SCATTER</b>    |                      |

- All** versions deliver results to all participating processes
- "v"** versions allow the chunks to have different sizes
- MPI\_ALLREDUCE, MPI\_REDUCE, MPI\_REDUCE\_SCATTER,** and take both built-in and user-defined combination functions

# Built-In Collective Computation Operations

| <b>MPI Name</b>   | <b>Operation</b>                    |
|-------------------|-------------------------------------|
| <b>MPI_MAX</b>    | <b>Maximum</b>                      |
| <b>MPI_MIN</b>    | <b>Minimum</b>                      |
| <b>MPI_PROD</b>   | <b>Product</b>                      |
| <b>MPI_SUM</b>    | <b>Sum</b>                          |
| <b>MPI_LAND</b>   | <b>Logical and</b>                  |
| <b>MPI_LOR</b>    | <b>Logical or</b>                   |
| <b>MPI_LXOR</b>   | <b>Logical exclusive or ( xor )</b> |
| <b>MPI_BAND</b>   | <b>Bitwise and</b>                  |
| <b>MPI_BOR</b>    | <b>Bitwise or</b>                   |
| <b>MPI_BXOR</b>   | <b>Bitwise xor</b>                  |
| <b>MPI_MAXLOC</b> | <b>Maximum value and location</b>   |
| <b>MPI_MINLOC</b> | <b>Minimum value and location</b>   |

# Extending the Message-Passing Interface

- Dynamic Process Management
  - Dynamic process startup
  - Dynamic establishment of connections
- One-sided communication
  - Put/get
  - Other operations
- Parallel I/O
- Other MPI-2 features
  - Generalized requests
  - Bindings for C++/ Fortran-90; inter-language issues

# When to use MPI

- Portability and Performance
- Irregular Data Structures
- Building Tools for Others
  - Libraries
- Need to Manage memory on a per processor basis

# When *not* to use MPI

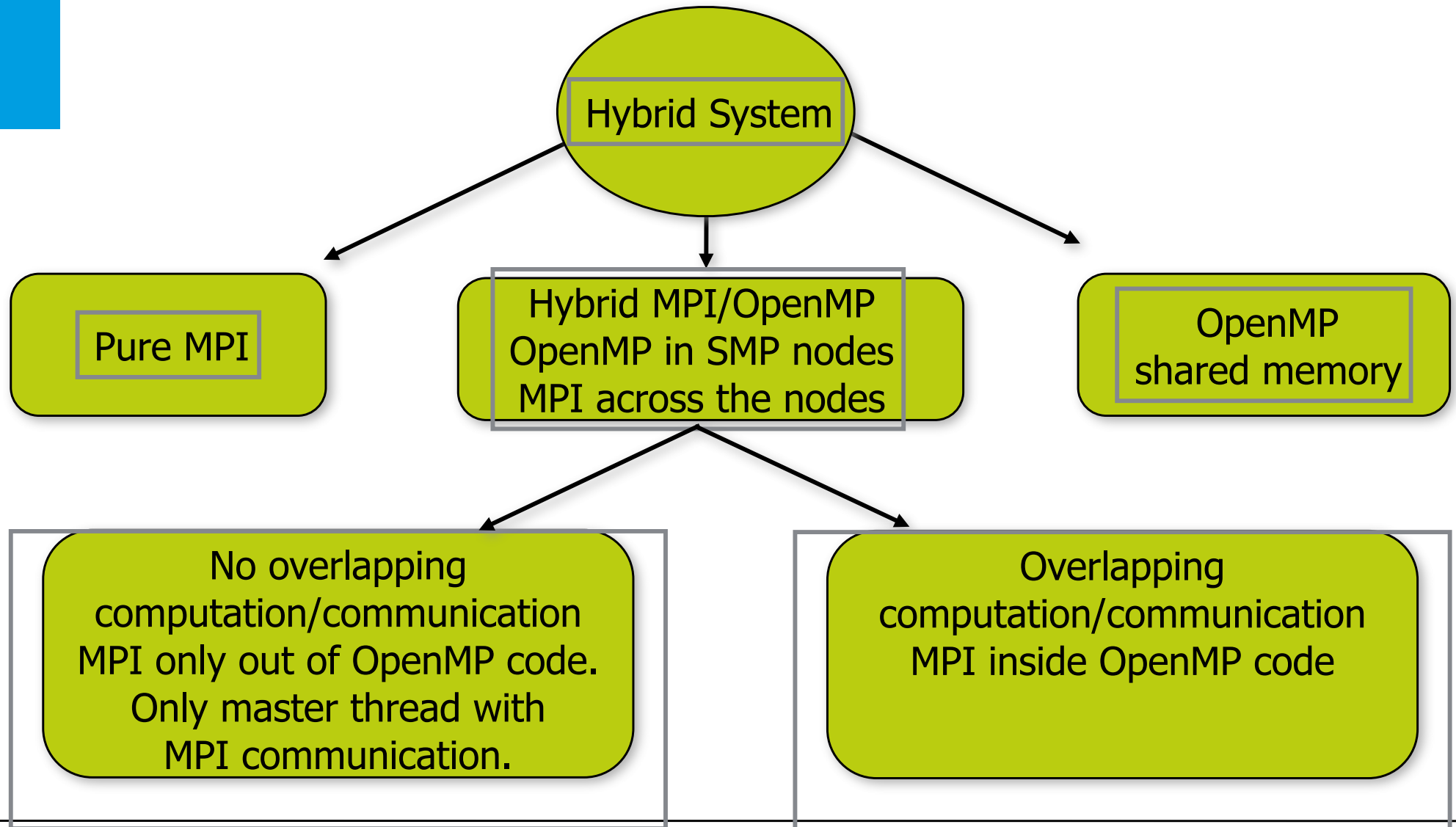
- Number of cores is limited and OpenMP is doing well on that number of cores
  - typically 16-32 cores in SMP
- Course: Introduction to MPI course
  - June 2022: ...

# Summary

# MPI Summary

- MPI Standard widely accepted by vendors and programmers
  - MPI implementations available on most modern platforms
  - Several MPI applications deployed
  - Several tools exist to trace and tune MPI applications
- **Simple** applications use point-to-point and collective communication operations
- **Advanced** applications use point-to-point, collective, communicators, datatypes, one-sided, and topology operations

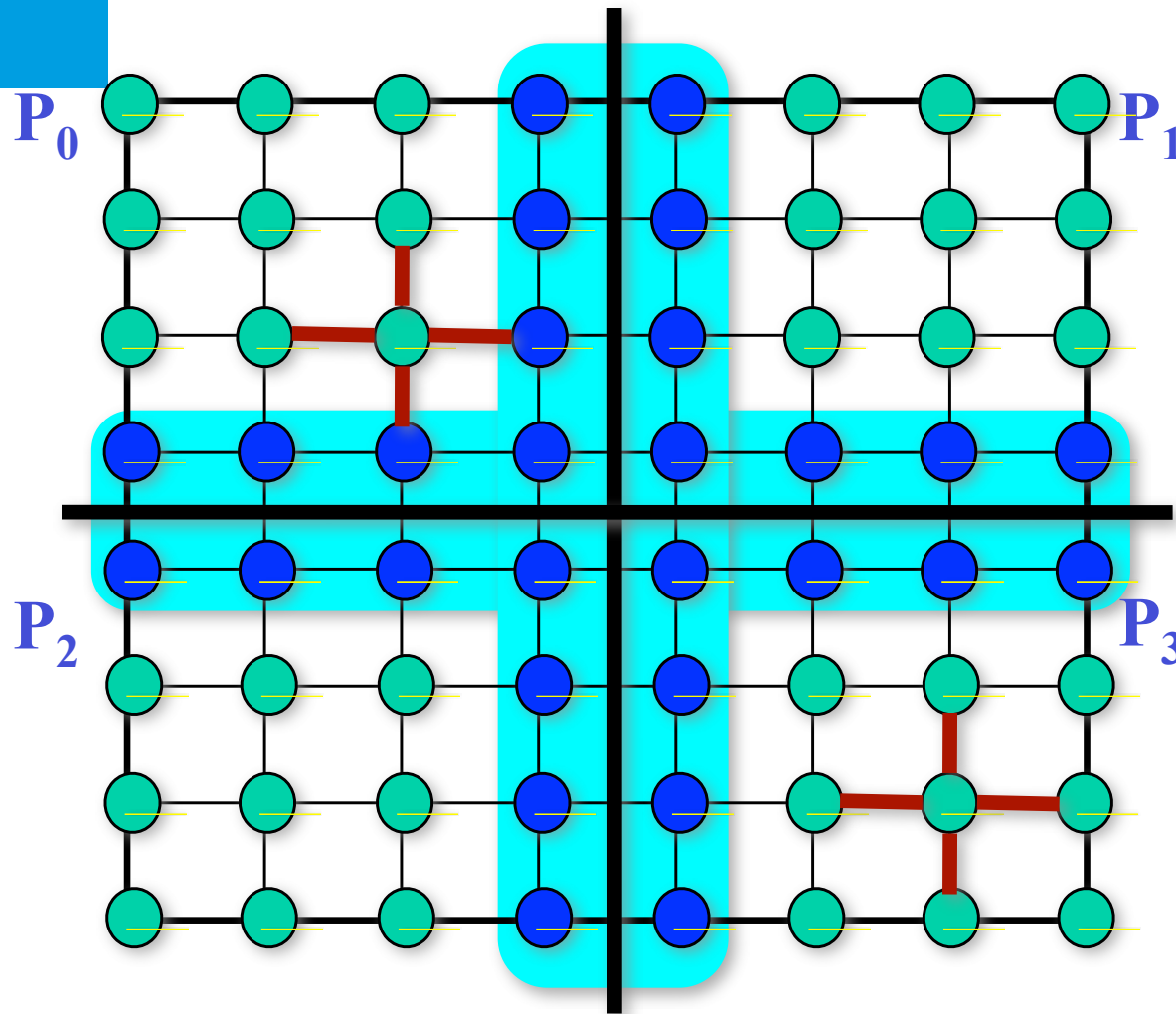
# Hybrid systems programming hierarchy



# Hybrid OpenMP/MPI

- Natural paradigm for clusters of SMP's
- May offer considerable advantages when application mapping and load balancing is tough
- Benefits with slower interconnection networks (overlapping computation/communication)
- ▶ Requires work and code analysis to change pure MPI codes
- ▶ Start with auto parallelization?
- ▶ Link shared memory libraries...check various thread/MPI processes combinations
- ▶ Study carefully the underlying architecture
- What is the future of this model? Could it be consolidated in new languages?
- Connection with many-core?

# Overlapping computation/communication: Example



Suppose we wish to solve the PDE

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x, y)$$

Using the Jacobi method: the value of  $u$  at each discretization point is given by a certain average among its neighbors, until convergence.

Distributing the mesh to SMP clusters by Domain Decomposition, it is clear that the **GREEN** nodes can proceed without any comm., while the **Blue** nodes have to communicate first and calculate later.

# MPI/OpenMPI: Overlapping computation/communication

Not only the master but other threads communicate. Call MPI primitives in OpenMP code regions.

```
if (my_thread_id < # ) {  
    MPI_... (communicate needed data)  
} else  
    /* Perform computations that to not need  
    communication */  
    .  
    .  
}  
/* All threads execute code that requires  
communication */  
.  
.
```

```
for (k=0; k < MAXITER; k++){
    /* Start parallel region here */
    #pragma omp parallel private(){
        my_id = omp_get_thread_num();

        if (my_id is given "halo points")
            MPI_SendRecv("From neighboring MPI process");
        else{
            for (i=0; i < # allocated points; i++)
                newval[i] = avg(oldval[i]);
        }

        if (there are still points I need to do) /* Thi
            for (i=0; i< # remaining points; i++)
                newval[i] = avg(oldval[i]);
        }
        for (i=0; i<(all_my_points); i++)
            oldval[i] = newval[i];
    }
    MPI_Barrier(); /* Synchronize all MPI processes here */
}
```

# Hiding IO with IO-Server

Use more nodes to act as IO-Servers (pseudo code)

Text

## Compute Node

```
do i=1,time_steps
  compute(j)
  checkpoint(data)
end do

subroutine checkpoint(data)
  MPI_Wait(send_req)
  buffer = data
  MPI_Isend(IO_SERVER, buffer)
end subroutine
```

## I/O Server

```
do i=1,time_steps
  do j=1,compute_nodes
    MPI_Recv(j, buffer)
    write(buffer)
  end do
end do
```

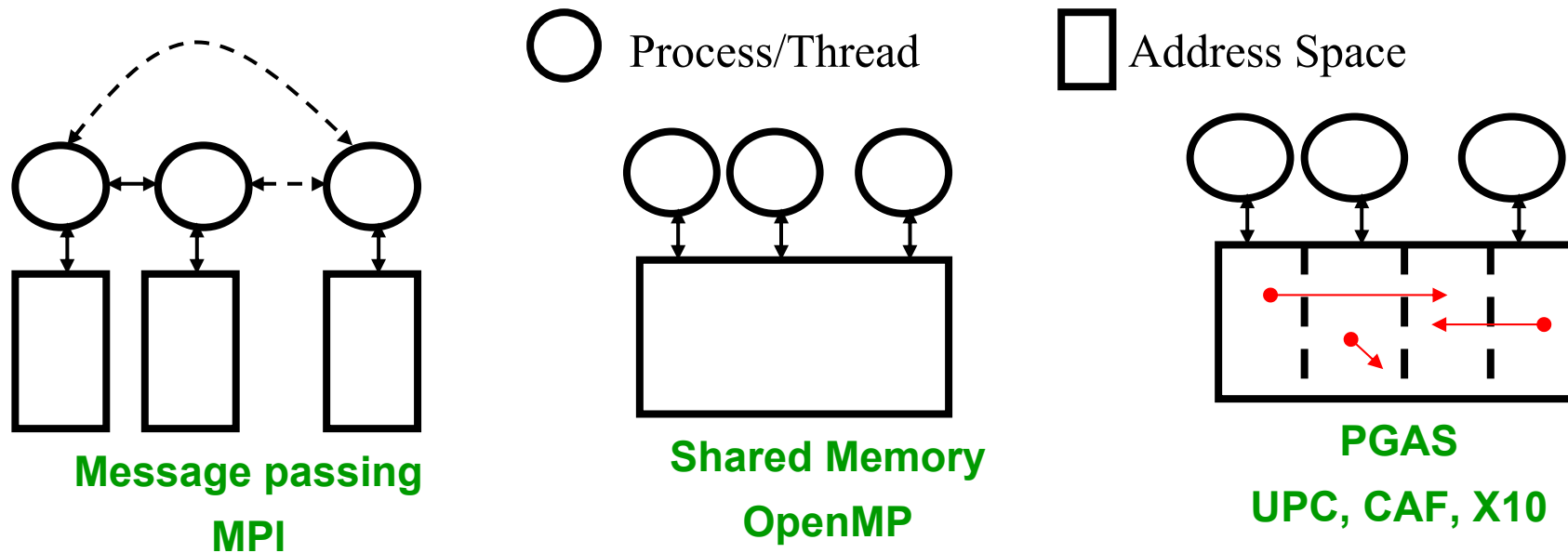
# PGAS

- What is PGAS?
- How to make use of PGAS as a programmer?

# Partitioned Global Address Space

- **Explicitly parallel, shared-memory like programming model**
- **Global addressable space**
  - Allows programmers to declare and “directly” access data distributed across the machine
- **Partitioned address space**
  - Memory is logically partitioned between *local* and *remote* (a two-level hierarchy)
  - Forces the programmer to pay attention to data locality, by exposing the inherent NUMA-ness of current architectures
- **Single Processor Multiple Data (SPMD) execution model**
  - All threads of control execute the **same** program
  - Number of threads fixed at startup
  - Newer languages such as X10 escape this model, allowing fine-grain threading
- **Different language implementations:**
  - **UPC** (C-based), CoArray Fortran (Fortran-based), Titanium and X10 (Java-based)

# Partitioned Global Address Space



- Computation is performed in multiple **places**.
- A place contains data that can be operated on remotely.
- Data lives in the place it was created, for its lifetime.

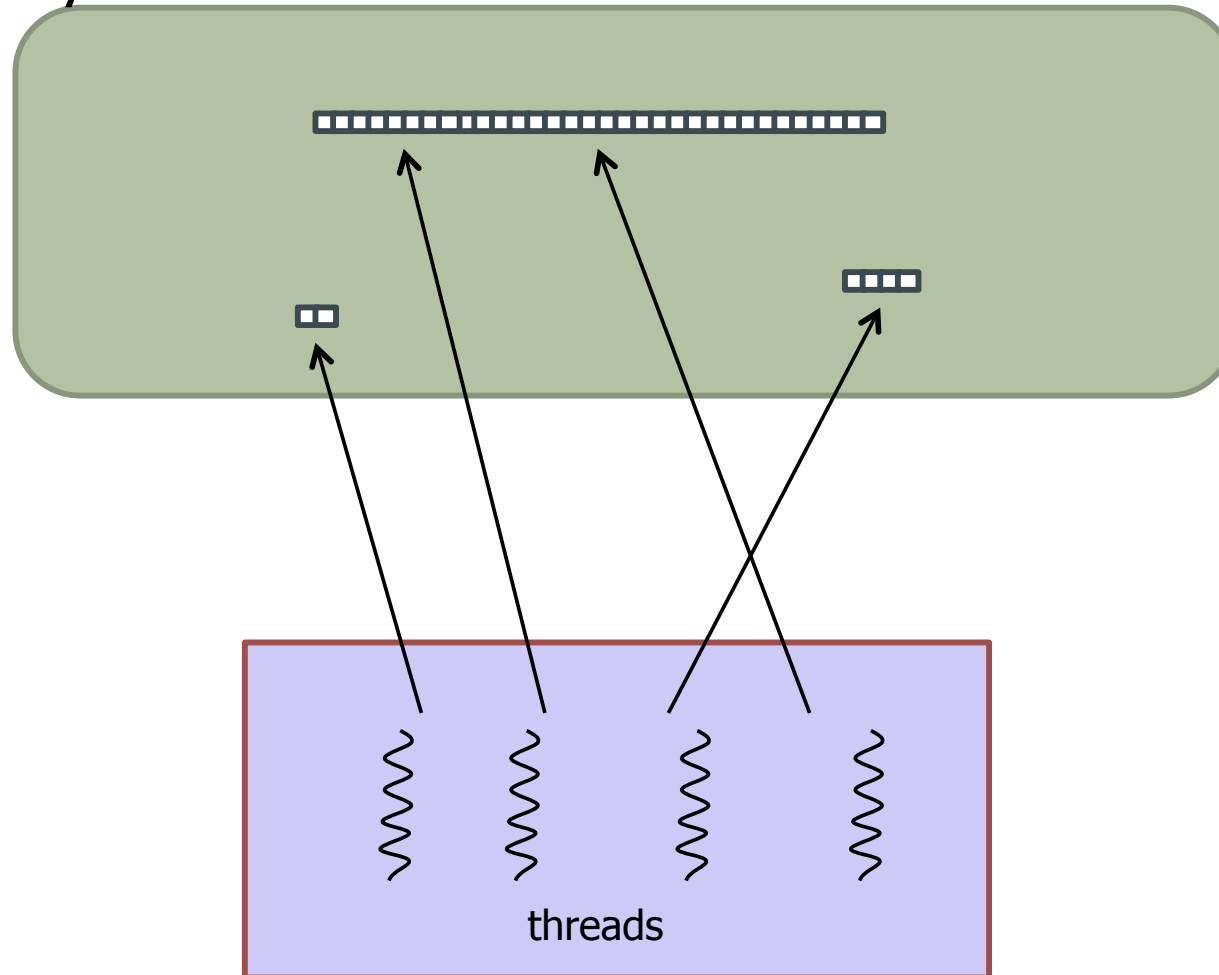
- A datum in one place may point to a datum in another place.
- Data-structures (e.g. arrays) may be distributed across many places.

# Shared Memory (OpenMP)

- Multiple threads share global memory
  - Most common variant: OpenMP
- Program loop iterations distributed to threads, more recent task features
  - Each thread has a means to refer to private objects within a parallel context
- Terminology
  - Thread, thread team
- Implementation
  - Threads map to user threads running on one SMP node
  - Extensions to multiple servers not so successful

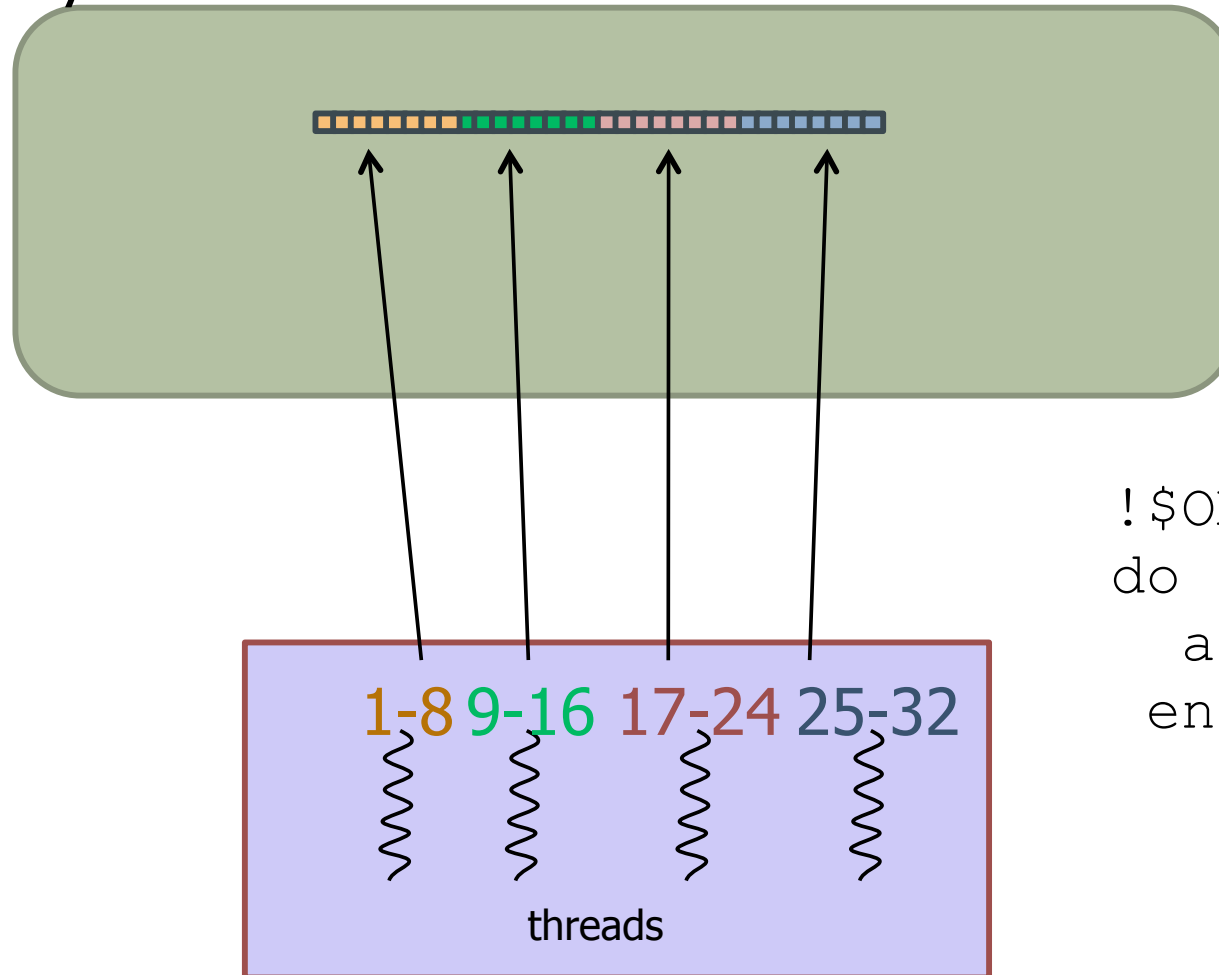
# OpenMP

memory



# OpenMP: work distribution

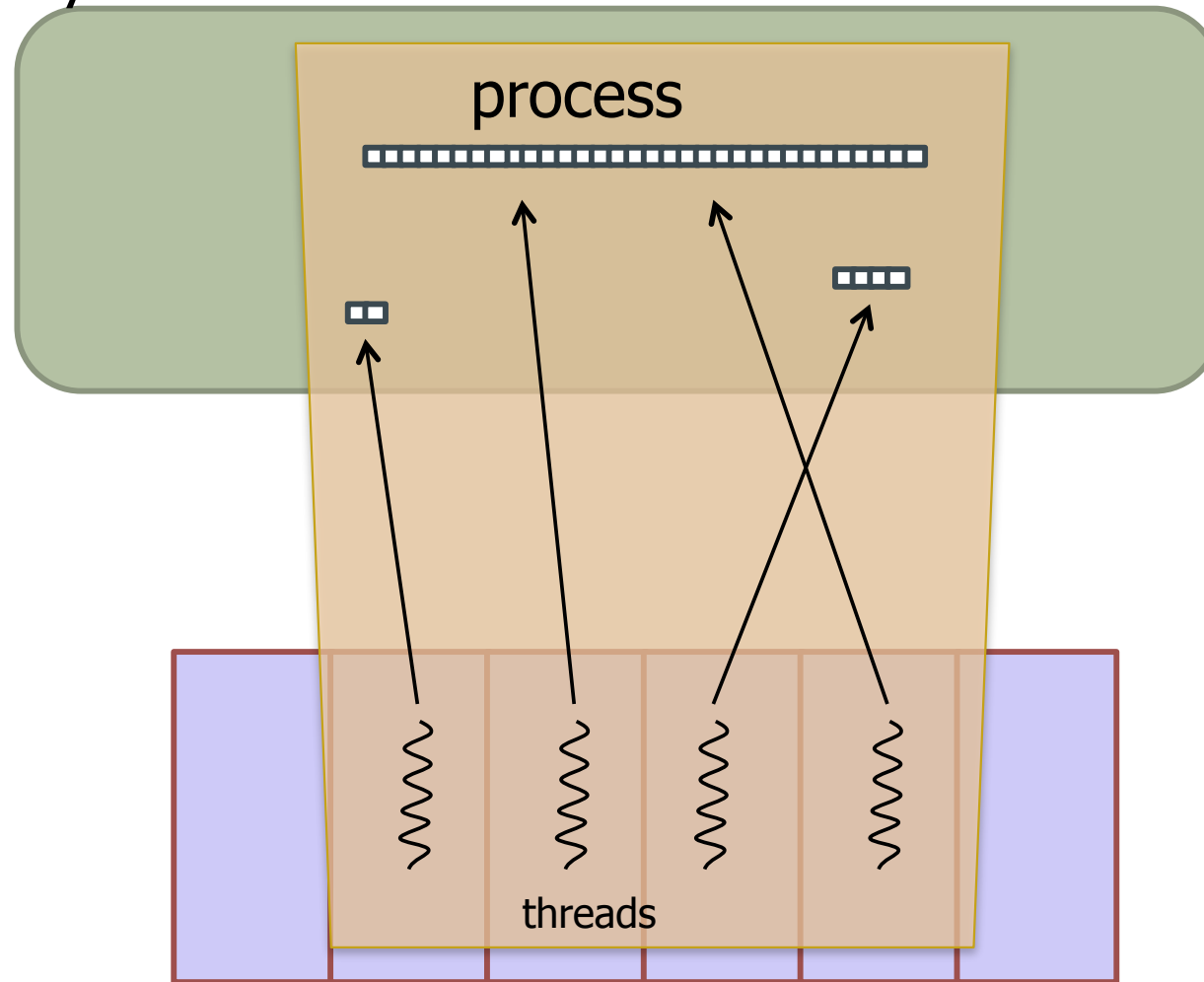
memory



```
!$OMP PARALLEL  
do i=1,32  
  a(i)=a(i)*2  
end do
```

# OpenMP: implementation

memory

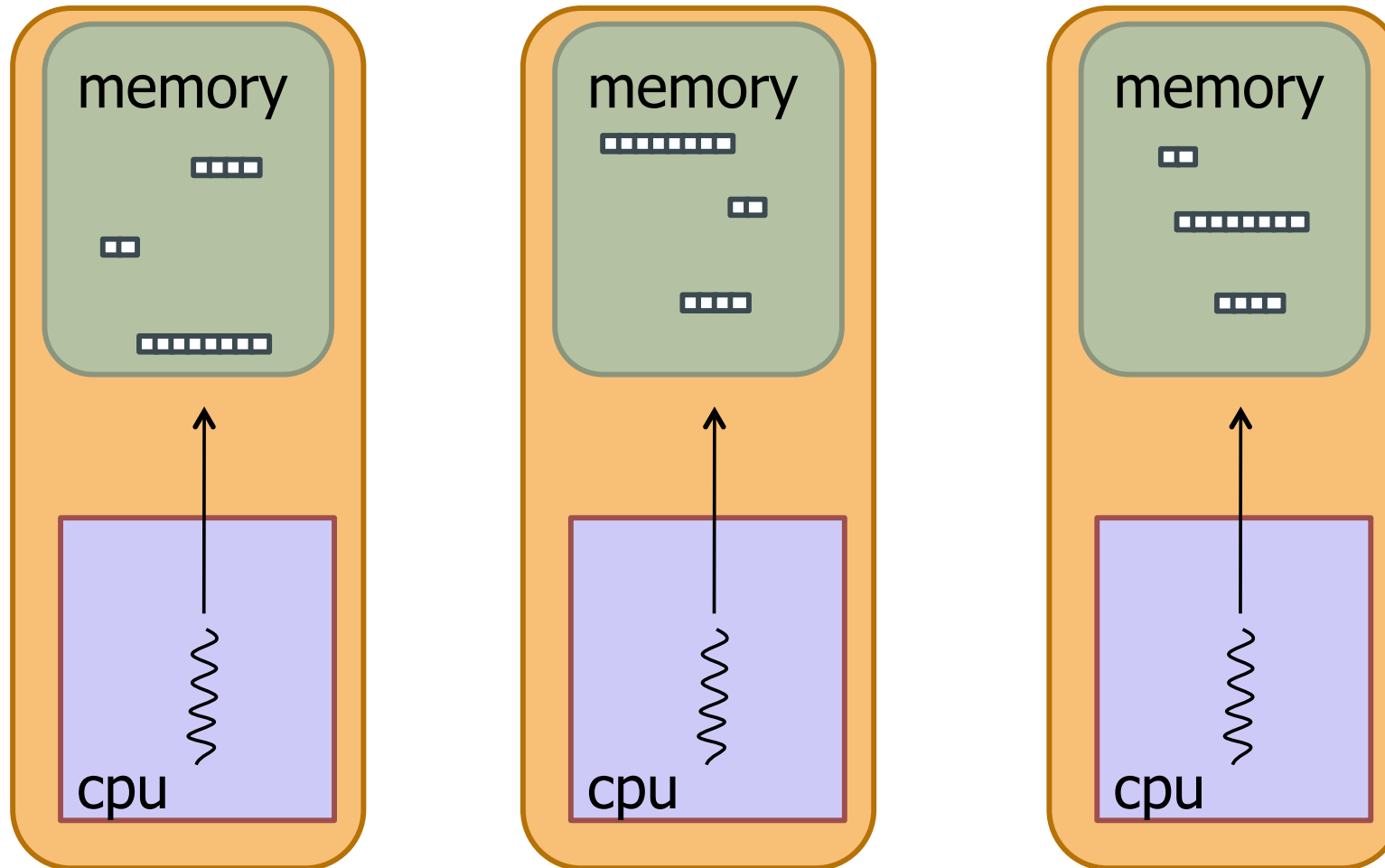


# Message Passing (MPI)

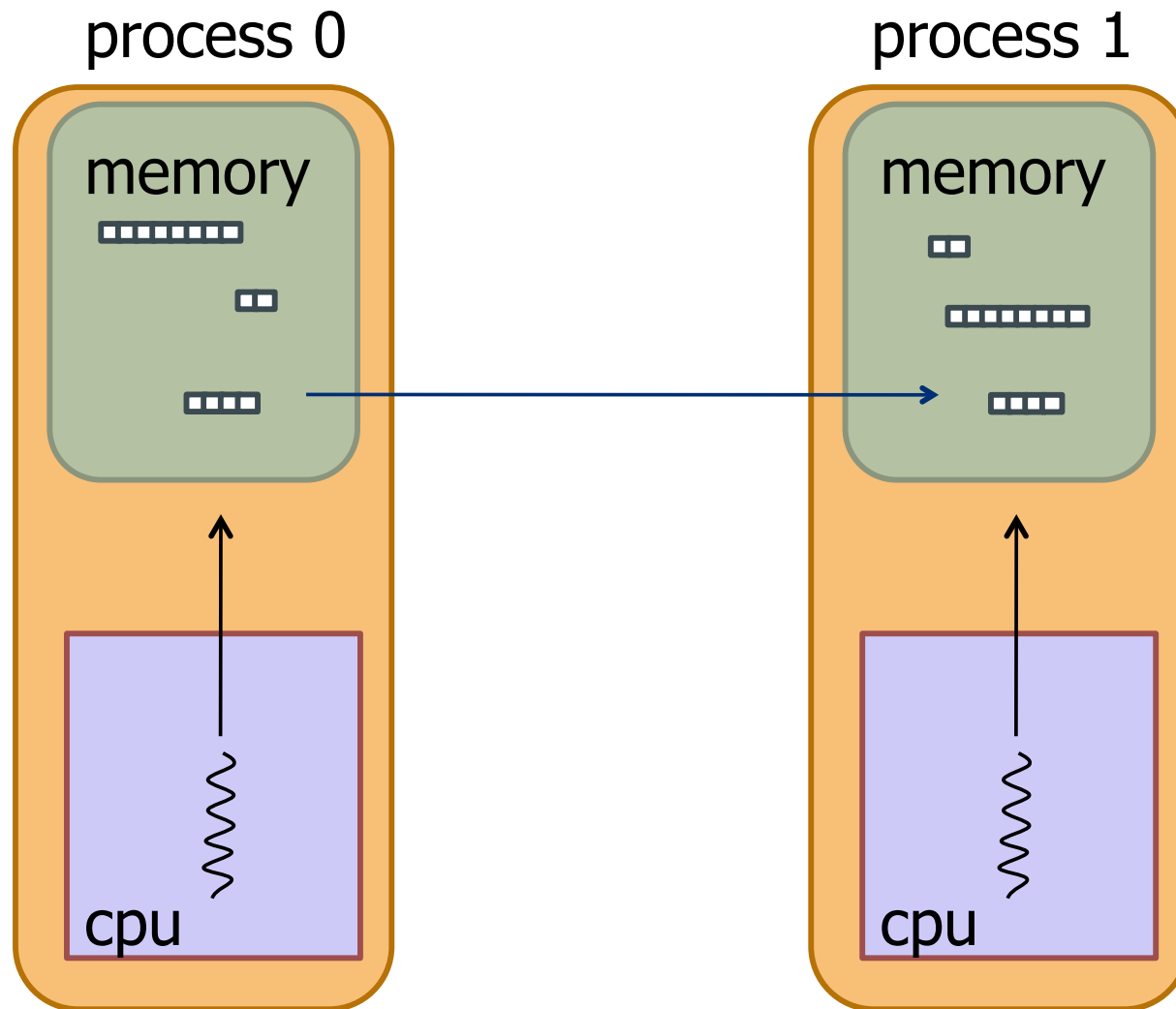
- Participating processes communicate using a message-passing API
- Remote data can only be communicated (sent or received) via the API.
- MPI (the Message Passing Interface) is the standard
- Implementation:
  - MPI processes map to processes within one SMP node or across multiple networked nodes
  - API provides process numbering, point-to-point and collective messaging operations

# MPI

processes



# MPI

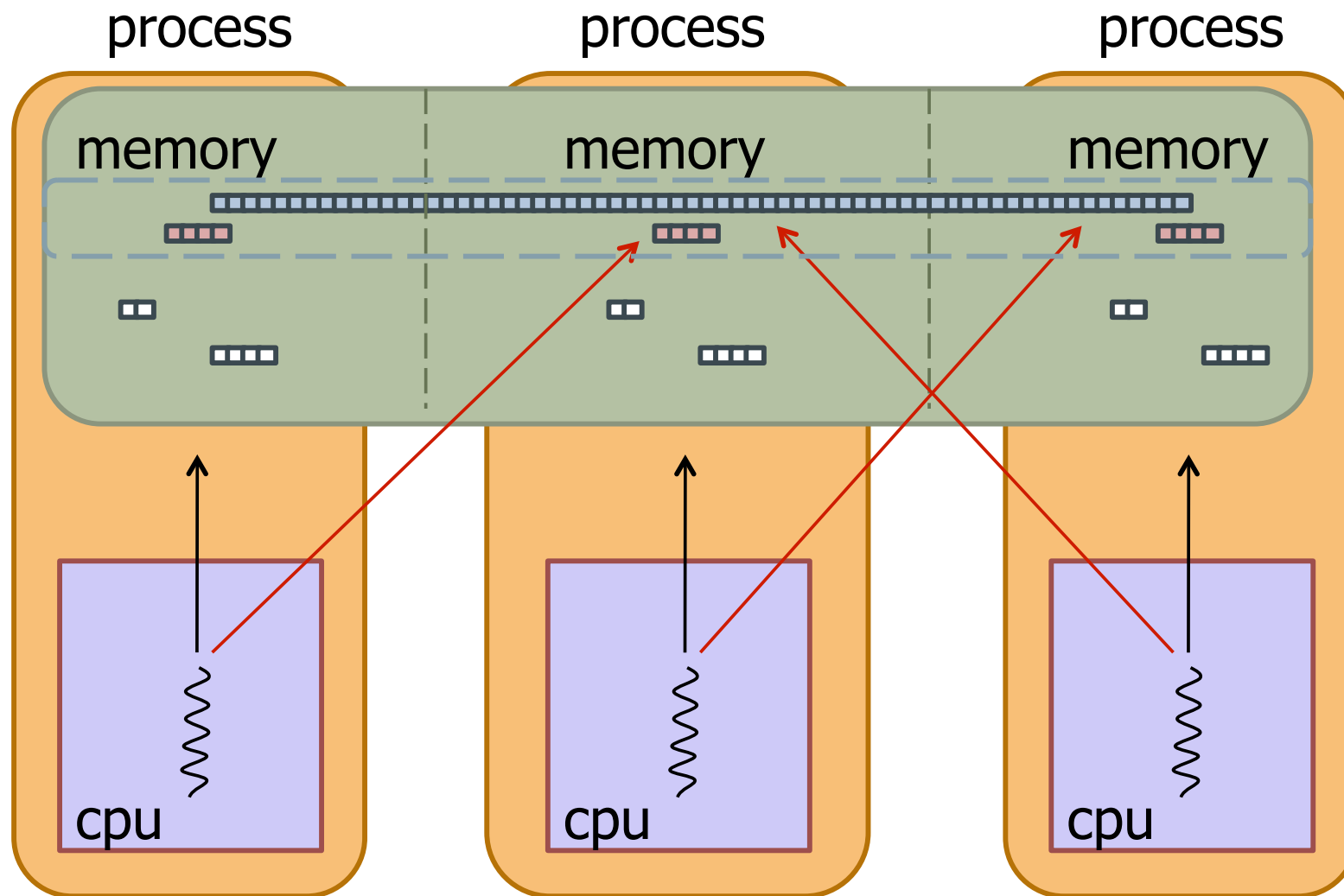


`MPI_Send(a, ..., 1, ...)`    `MPI_Recv(a, ..., 0, ...)`

# Partitioned Global Address Space

- Shortened to PGAS
- Participating processes/threads have access to local memory via standard program mechanisms
- Access to remote memory is directly supported by the PGAS language

# PGAS



# PGAS languages

Partitioned Global Address Space (PGAS):

- Global address space – any process can address memory on any processor
- Partitioned GAS – retain information about locality
- Core idea– hardest part of writing parallel code is managing data distribution and communication; make that simple and explicit
- PGAS Languages try to simplify parallel programming (increase programmer productivity).

# Data Parallel Languages

- **Unified Parallel C** (UPC) is an extension of the C programming language designed for high performance computing on large-scale parallel machines.  
<http://upc.lbl.gov/>
- **Co-array Fortran** (CAF) is part of Fortran 2008 standard. It is a simple, explicit notation for data decomposition, such as that often used in message-passing models, expressed in a natural Fortran-like syntax.  
<http://www.co-array.org>
- both need a global address space (which is not equal to SMP)

# UPC

- Unified Parallel C:
  - An extension of C99
  - An evolution of AC, PCP, and Split-C
- Features
  - SPMD parallelism via replication of *threads*
  - Shared and private address spaces
  - Multiple memory consistency models
- Benefits
  - Global view of data
  - One-sided communication

# Co-Array Fortran

- Co-Array Fortran:
  - An extension of Fortran 95 and part of “Fortran 2008”
  - The language formerly known as F--
- Features
  - SPMD parallelism via replication of *images*
  - *Co-arrays* for distributed shared data
- Benefits
  - Syntactically transparent communication
  - One-sided communication
  - Multi-dimensional arrays
  - Array operations

# Basic execution model co-array F--

- Program executes as if replicated to multiple copies with each copy executing asynchronously (SPMD)
- Each copy (called an image) executes as a normal Fortran application
- New object indexing with `[]` can be used to access objects on other images.
- New features to inquire about image index, number of images and to synchronize

# CAF (F--)



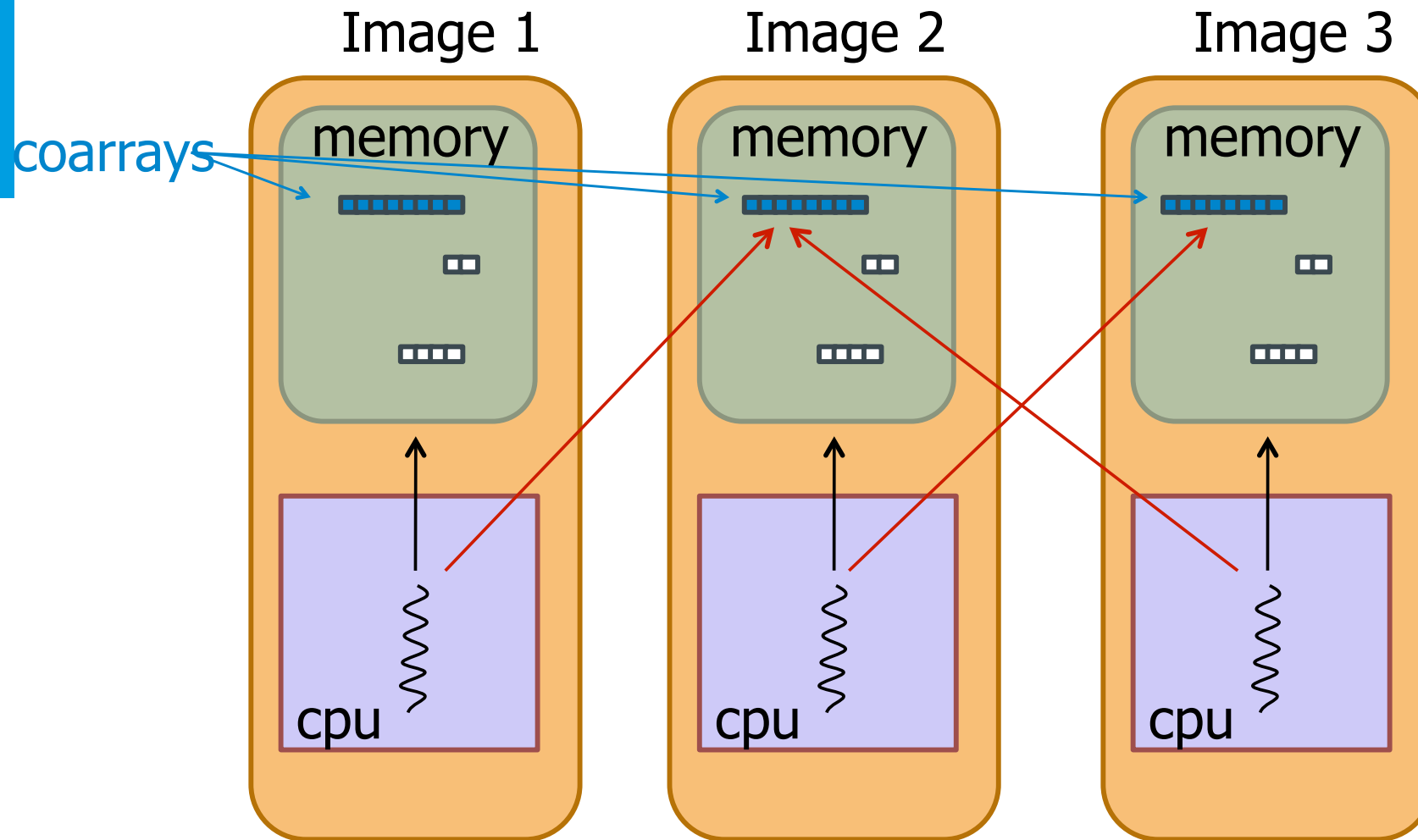
parallel

```
REAL, DIMENSION(N) [*] :: X,Y  
X(:) = Y(:)[Q]
```

Array indices in parentheses follow the normal Fortran rules within one memory image.

Array indices in square brackets provide an equally convenient notation for accessing **objects across images** and follow similar rules.

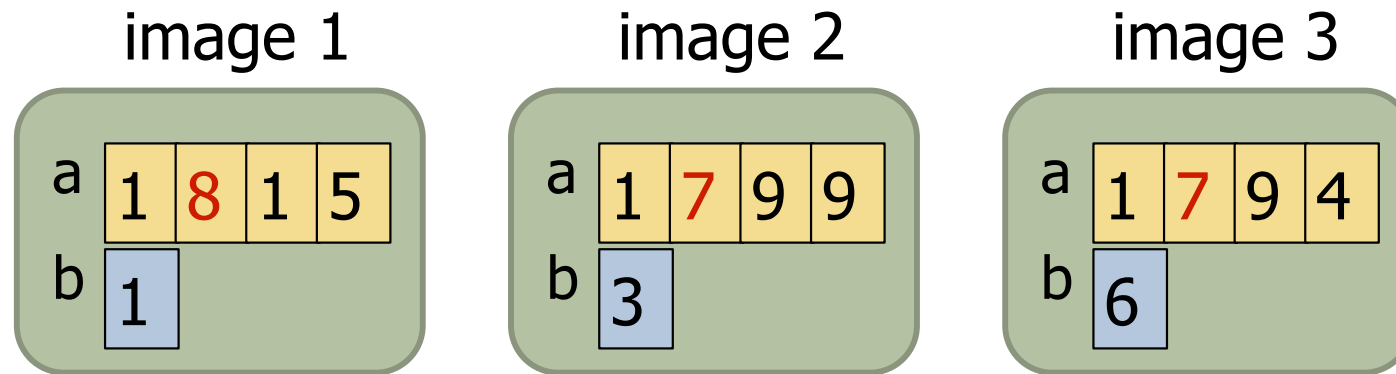
# Coarray execution model



Remote access with square bracket indexing: `a(:)[2]`

# Basic coarray declaration and usage

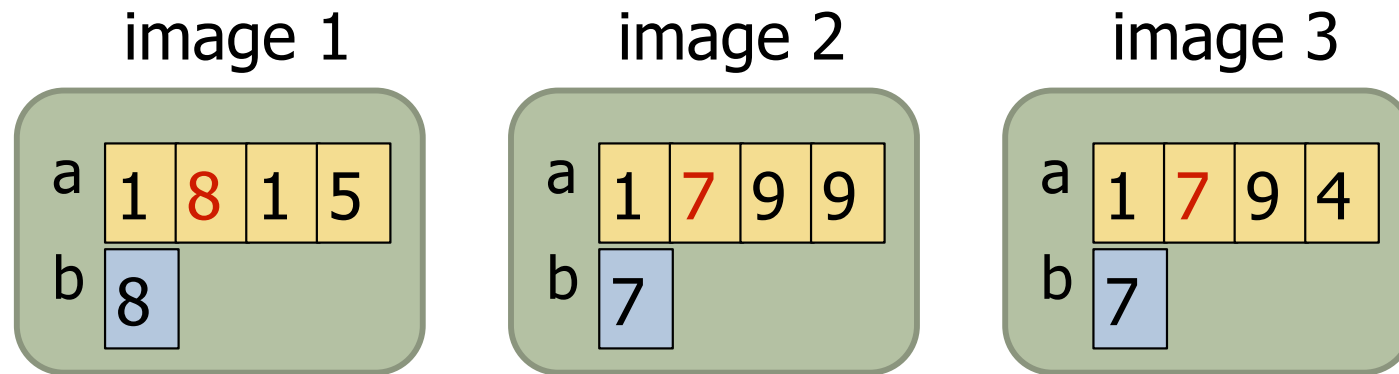
```
integer :: b  
integer :: a(4) [*] !coarray
```



```
b=a(2)
```

# Basic coarray declaration and usage

```
integer :: b  
integer :: a(4) [*] !coarray
```



```
b=a(2)
```

# Basic coarray declaration and usage

```
integer :: b  
integer :: a(4) [*] !coarray
```

Text

image 1

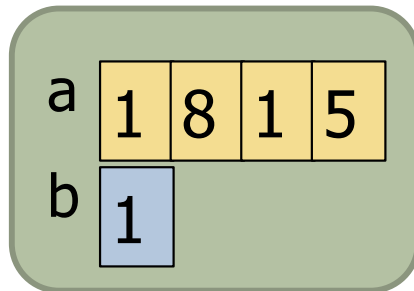


image 2

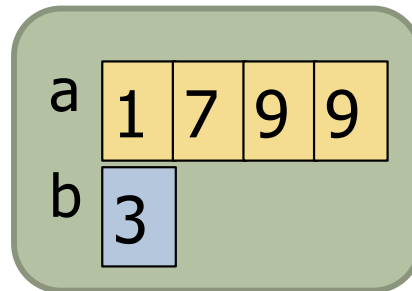
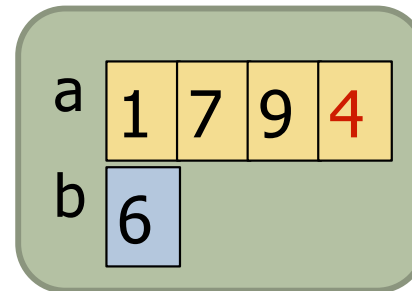


image 3



```
b=a(4)[3]
```

# Basic coarray declaration and usage

```
integer :: b  
integer :: a(4) [*] !coarray
```

Text

image 1

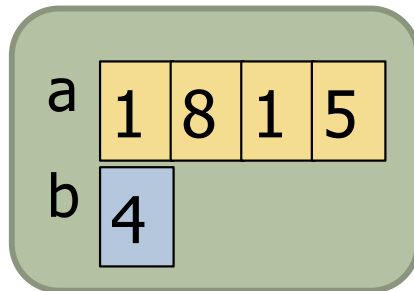


image 2

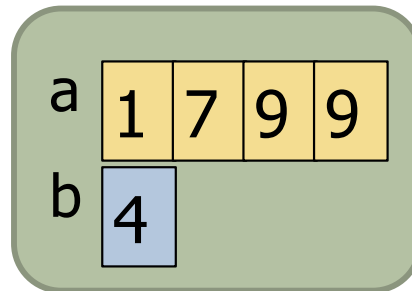
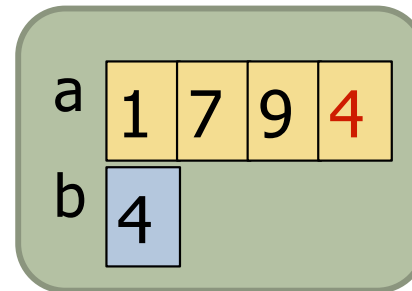
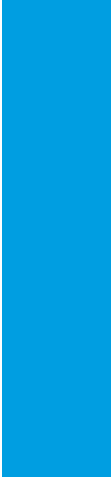


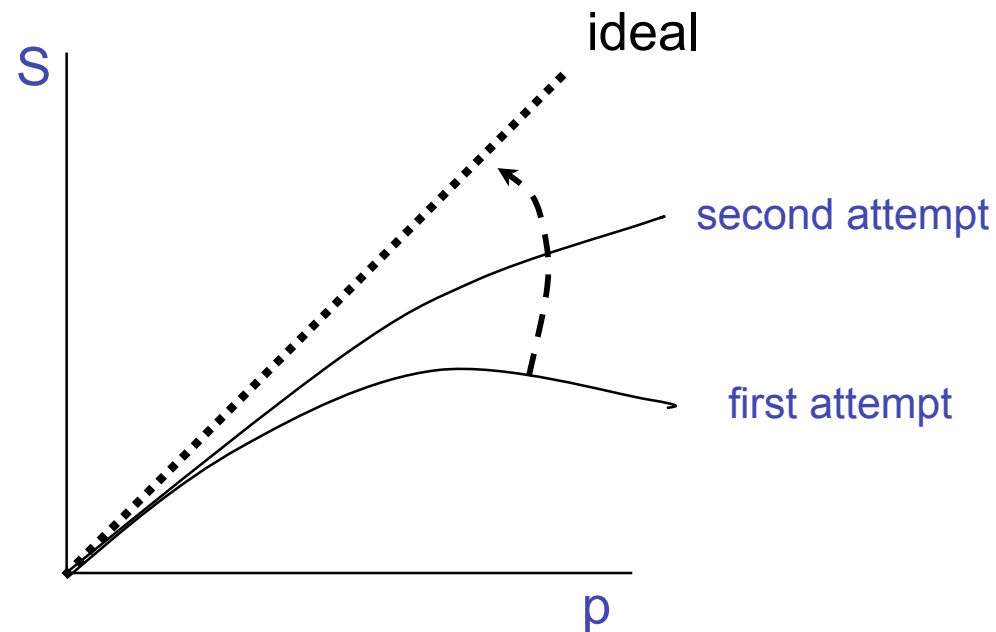
image 3



```
b=a(4)[3]
```



# Performance Tuning



Performance tuning of parallel applications is an iterative process

# Compiling MPI Programs

# Compiling and Starting MPI Jobs

- Compiling:
  - Need to link with appropriate MPI and communication subsystem libraries and set path to MPI Include files
  - Most vendors provide scripts or wrappers for this (mpxlf, mpif77, mpicc, etc)
- Starting jobs:
  - Most implementations use a special loader named mpirun
    - **mpirun -np <no\_of\_processors> <prog\_name>**
  - In MPI-2 it is recommended to use
    - **mpiexec -n <no\_of\_processors> <prog\_name>**

# MPICH: a Portable MPI Environment

- MPICH is a high-performance portable implementation of MPI (both 1 and 2).
- It runs on MPP's, clusters, and heterogeneous networks of workstations.
- The CH comes from **Chameleon**, the portability layer used in the original MPICH to provide portability to the existing message-passing systems.

- In a wide variety of environments, one can do:

```
mpicc -mpitrace myprog.c  
mpirun -np 10 myprog  
upshot myprog.log
```

to build, compile, run, and analyze performance.

# MPICH2

MPICH2 is an all-new implementation of the MPI Standard, designed to implement all of the MPI-2 additions to MPI.

- ▶ separation between process management and communications
  - ▶ use daemons (mpd) on nodes
  - ▶ dynamic process management,
  - ▶ one-sided operations,
  - ▶ parallel I/O, and others
- 
- <http://www.mcs.anl.gov/research/projects/mpich2/>

# Compiling MPI programs

- From a command line:  
`mpicc -o prog prog.c`
- Use profiling options (specific to mpich)
  - **-mpilog** Generate log files of MPI calls
  - **-mpitrace** Trace execution of MPI calls
  - **-mpianim** Real-time animation of MPI (not available on all systems)
  - **--help** Find list of available options
- The environment variables `MPICH_CC`, `MPICH_CXX`, `MPICH_F77`, and `MPICH_F90` may be used to specify alternate C, C++, Fortran 77, and Fortran 90 compilers, respectively.



# example hello.c

```
#include "mpi.h"
#include <stdio.h>
int main(int argc ,char *argv[])
{
    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    fprintf(stdout, "Hello World, I am process
%d\n", myrank);
    MPI_Finalize();
    return 0;
}
```

# Example: using MPICH1

```
-bash-3.1$ mpicc -o hello hello.c  
-bash-3.1$ mpirun -np 4 hello  
Hello World, I am process 0  
Hello World, I am process 2  
Hello World, I am process 3  
Hello World, I am process 1
```

# Example: details

- If using frontend and compute nodes in machines file use  
`mpirun -np 2 -machinefile machines hello`
- If using only compute nodes in machine file use  
`mpirun -nolocal -np 2 -machinefile machines hello`
  - `-nolocal` - don't start job on frontend
  - `-np 2` - start job on 2 nodes
  - `-machinefile machines` - nodes are specified in machines file
  - `hello` - start program hello

# Notes on clusters

- Make sure you have access to the compute node (ssh keys are generated ssh-keygen) and ask your system administrator.
- Which mpicc are you using
  - \$ which mpicc
- command line arguments are not always passed to mpirun/mpiexec (depending on your version). In that case make a script which calls your program with all its arguments

# MPICH2 daemons

- `mpdtrace`: output a list of nodes on which you can run MPI programs (runs mpd daemons).
  - ▶ The `-l` option lists full hostnames and the port where the mpd is listening.
- `mpd` starts an mpd daemon.
- `mpdboot` starts a set of mpd's on a list of machines.
- `mpdlistjobs` lists the jobs that the mpd's are running.
- `mpdkilljob` kills a job specified by the name returned by `mpdlistjobs`
- `mpdsigjob` delivers a signal to the named job.

# MPI - Message Passing Interface

- MPI or MPI-1 is a library specification for message-passing.
- MPI-2: Adds in Parallel I/O, Dynamic Process management, Remote Memory Operation, C++ & F90 extension ...
- MPI Standard:  
<http://www-unix.mcs.anl.gov/mpi/standard.html>
- MPI Standard 1.1 Index:  
<http://www.mpi-forum.org/docs/mpi-11-html/node182.html>
- MPI-2 Standard Index:  
<http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/node306.htm>
- MPI Forum Home Page:  
<http://www.mpi-forum.org/index.html>

# MPI tutorials

- [http://www.nccs.nasa.gov/tutorials/mpi\\_tutorial2/mpi\\_II\\_tutorial.html](http://www.nccs.nasa.gov/tutorials/mpi_tutorial2/mpi_II_tutorial.html)
- [https://fs.hlrs.de/projects/par/par\\_prog\\_ws/](https://fs.hlrs.de/projects/par/par_prog_ws/)
- Course: Introduction to MPI
  - 50/50 lecture and exercises

# MPI Sources

- The Standard (3.0) itself:
  - at <http://www.mpi-forum.org>
  - All MPI official releases, in both PDF and HTML
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
- Other information on Web:
  - at <http://www.mcs.anl.gov/mpi>
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages
  - [http://mpi.deino.net/mpi\\_functions/index.htm](http://mpi.deino.net/mpi_functions/index.htm)

# Job Management and queuing

# Job Management and queuing

- On a large system many users are running simultaneously.
- What to do when:
  - The system is full and you want to run your 512 CPU job?
  - You want to run 16 jobs, should others wait on that?
  - Have bigger jobs priority over smaller jobs?
  - Have longer jobs lower/higher priority?
- The job manager and queue system takes care of it.

# MPICH1 in PBS

```
#!/bin/bash
#PBS -N Flank
#PBS -V
#PBS -l nodes=11:ppn=2
#PBS -j eo

cd $PBS_O_WORKDIR
export nb=`wc -w < $PBS_NODEFILE`
echo $nb

mpirun -np $nb -machinefile $PBS_NODEFILE ~/bin/
migr_mpi \
    file_vel=$PBS_O_WORKDIR/grad_salt_rot.su \
    file_shot=$PBS_O_WORKDIR/cfp2_sx.su
```

# Job Management and queuing

- PBS (maui):
    - <http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php>
    - Torque resource manager <http://www.clusterresources.com/pages/products/torque-resource-manager.php>
  - Sun Grid Engine
    - <http://gridengine.sunsource.net/>
  - SLURM
    - <http://slurm.schedmd.com>
- Luckily their interface is very similar (qsub, qstat, ...)

# queuing commands

- qsub
- qstat
- qdel
- xpbsmon

# qsub

```
#PBS -l nodes=10:ppn=1
#PBS -l mem=20mb
#PBS -l walltime=1:00:00
#PBS -j eo
```

submit:

```
qsub -q normal job.scr
```

output:

```
jobname.ejobid
jobname.ojobid
```

# qstat

- available queue and resources

```
qstat -q
```

- queued and running jobs

```
qstat (-a)
```

# qdel

- deletes job from queue and stops all running executables

```
qdel jobid
```

# Submitting many jobs in one script

```
#!/bin/bash -f
#
export xsrc1=93
export xsrc2=1599
export dxsrc=3
xsrc=$xsrc1

while (( xsrc <= xsrc2 ))
do
echo ' modeling shot at x=' $xsrc

cat << EOF > jobs/pbs_${xsrc}.job
#!/bin/bash
#
#PBS -N fahud_${xsrc}
#PBS -q verylong
#PBS -l nodes=1:ppn=1
#PBS -V

program with arguments

EOF

qsub jobs/pbs_${xsrc}.job
(( xsrc = $xsrc + $dxsrc))
done
```

Be careful !

# submitting job-arrays with slurm

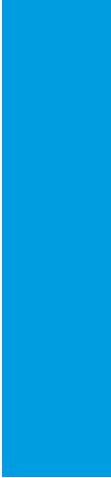
```
#!/bin/bash
#SBATCH --job-name=mega_array    # Job name
#SBATCH --nodes=1                # Use one node
#SBATCH --ntasks=1              # Run a single task
#SBATCH --time=00:10:00         # Time limit hrs:min:sec
#SBATCH --output=array_%A-%a.out # Standard output and error log
#SBATCH --array=1-5             # Array range
pwd; hostname; date

#Set the number of runs that each SLURM task should do
PER_TASK=1000

# Calculate the starting and ending values for this task based
# on the SLURM task and the number of runs per task.
START_NUM=$(( ($SLURM_ARRAY_TASK_ID - 1) * $PER_TASK + 1 ))
END_NUM=$(( $SLURM_ARRAY_TASK_ID * $PER_TASK ))

echo This is task $SLURM_ARRAY_TASK_ID, which will do runs $START_NUM to
$END_NUM

# Run the loop of runs for this task.
for (( run=$START_NUM; run<=END_NUM; run++ )); do
    echo This is SLURM task $SLURM_ARRAY_TASK_ID, run number $run
    #Do your stuff here
done
```



# Exercise: PI with MPI and OpenMP

- Compile and run for computing PI parallel
- MPI\_pi directory
- Check the README for instructions.
- It is assumed that you use mpich1 and has PBS installed as a job scheduler. If you have mpich2 let me know.
- Use qsub to submit the mpi job (an example script is provided) to a queue.

# Exercise: PI with MPI and OpenMP

| cores | OpenMP   | marken (MPI) | linux (MPI) |
|-------|----------|--------------|-------------|
| 1     | 9.617728 | 14.10798     | 22.15252    |
| 2     | 4.874539 | 7.071287     | 9.661745    |
| 4     | 2.455036 | 3.532871     | 5.730912    |
| 6     | 1.627149 | 2.356928     | 3.547961    |
| 8     | 1.214713 | 1.832055     | 2.804715    |
| 12    | 0.820746 | 1.184123     |             |
| 16    | 0.616482 | 0.955704     |             |

# Exercise: OpenMP Max

- Find the maximum number in a random generated array.
- on github [HPCource/OMP\\_MAX](#)
- There is a README for instructions.
- The exercise focus on using the available number of cores in an efficient way.
- Also inspect the code and see how the reductions are done, is there another way of doing the reductions?

# Exercise: OpenMP details

- More details is using OpenMp and shared memory parallelisation
- collection of code is in HPCource/PowerGroup
- Unpack tar file and check the README for instructions.
- These are 'old' exercises from SGI and give insight in problems you can encounter using OpenMP.
- It requires already some knowledge about OpenMP. The OpenMP F-Summary.pdf from the website can be helpful.

END