

MIT OpenCourseWare
<http://ocw.mit.edu>

6.189 Multicore Programming Primer, January (IAP) 2007

Please use the following citation format:

Rodric Rabbah, *6.189 Multicore Programming Primer, January (IAP) 2007*. (Massachusetts Institute of Technology: MIT OpenCourseWare).
<http://ocw.mit.edu> (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit:
<http://ocw.mit.edu/terms>

6.189 IAP 2007

Lecture 5

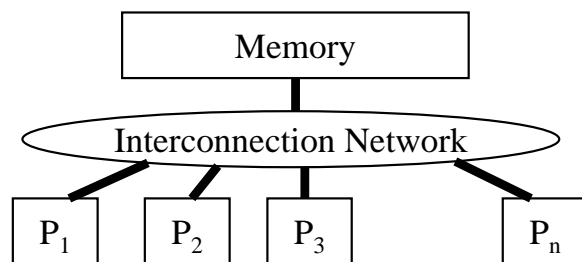
Parallel Programming Concepts

Recap

- Two primary patterns of multicore architecture design

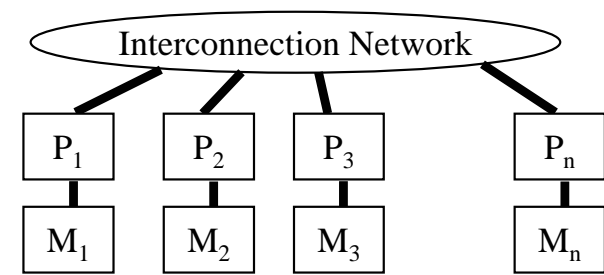
- Shared memory

- Ex: Intel Core 2 Duo/Quad
- One copy of data shared among many cores
- Atomicity, locking and synchronization essential for correctness
- Many scalability issues



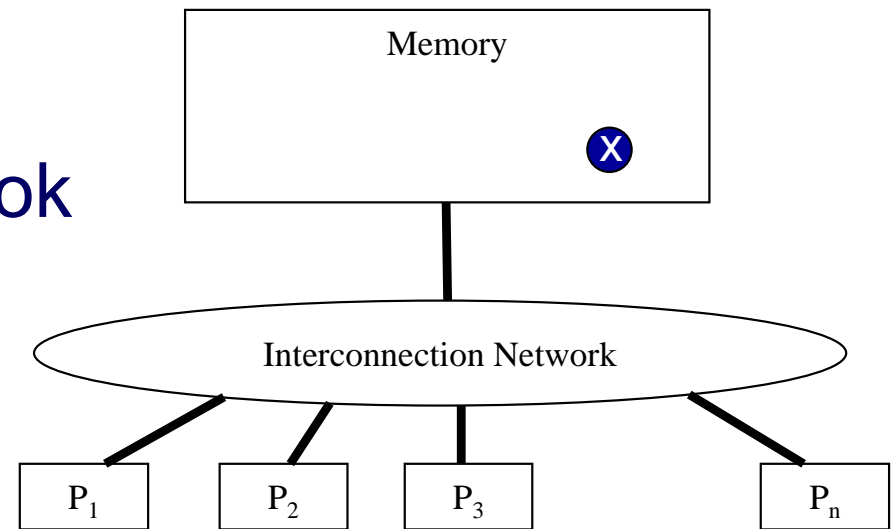
- Distributed memory

- Ex: Cell
- Cores primarily access local memory
- Explicit data exchange between cores
- Data distribution and communication orchestration is essential for performance



Programming Shared Memory Processors

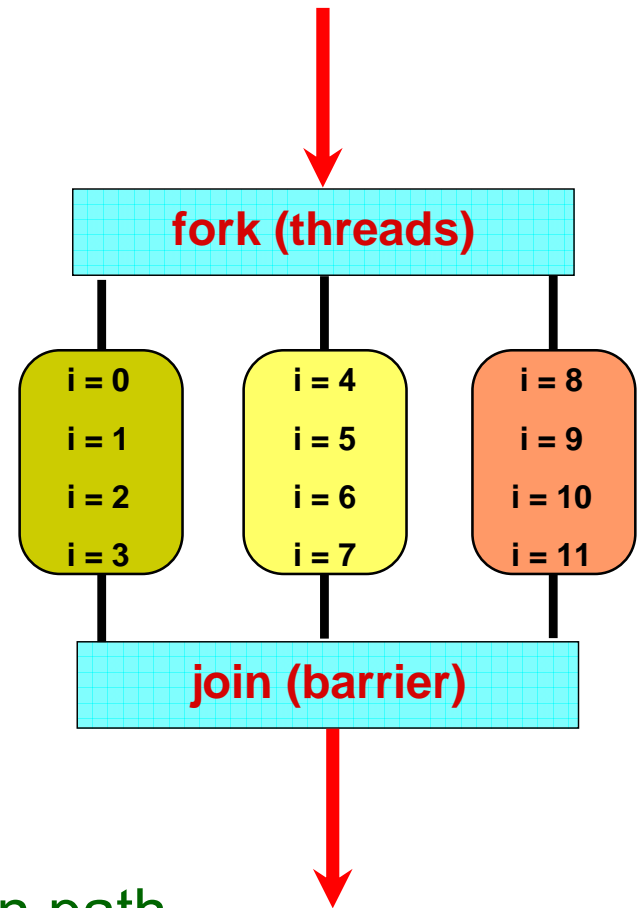
- Processor 1...n ask for X
- There is only one place to look
- Communication through shared variables
- Race conditions possible
 - Use synchronization to protect from conflicts
 - Change how data is stored to minimize synchronization



Example Parallelization

```
for (i = 0; i < 12; i++)  
    C[i] = A[i] + B[i];
```

- Data parallel
 - Perform same computation but operate on different data
- A single process can fork multiple concurrent threads
 - Each thread encapsulate its own execution path
 - Each thread has local state and shared resources
 - Threads communicate through shared resources such as global memory

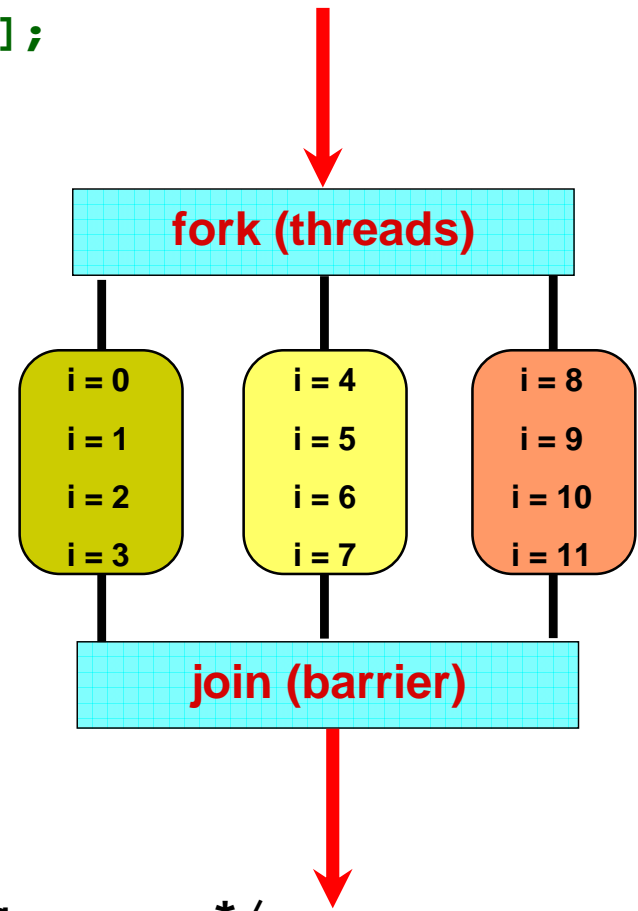


Example Parallelization With Threads

```
int A[12] = {...}; int B[12] = {...}; int C[12];
```

```
void add_arrays(int start)
{
    int i;
    for (i = start; i < start + 4; i++)
        C[i] = A[i] + B[i];
}
```

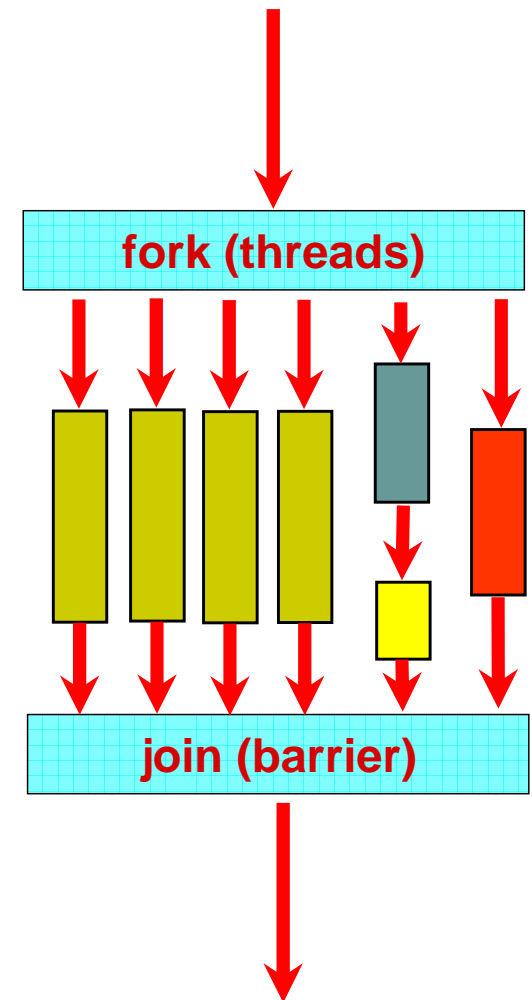
```
int main (int argc, char *argv[])
{
    pthread_t threads_ids[3];
    int rc, t;
    for(t = 0; t < 4; t++) {
        rc = pthread_create(&thread_ids[t],
                           NULL /* attributes */,
                           add_arrays /* function */,
                           t * 4 /* args to function */);
    }
    pthread_exit(NULL);
}
```



Types of Parallelism

- Data parallelism
 - Perform same computation but operate on different data
- Control parallelism
 - Perform different functions

```
pthread_create(/* thread id      */,  
              /* attributes     */,  
              /* any function   */,  
              /* args to function */);
```



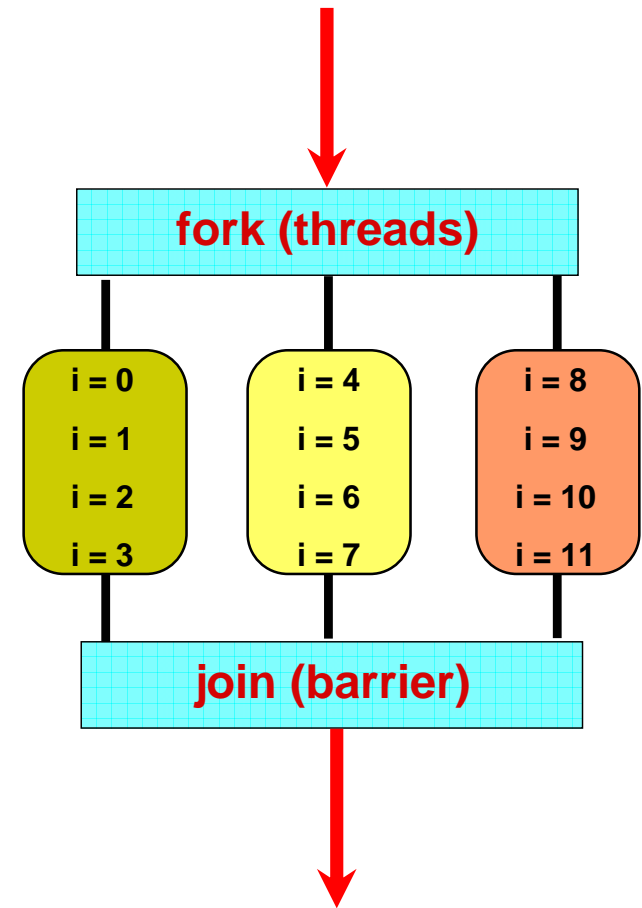
Parallel Programming with OpenMP

- Start with a parallelizable algorithm
 - SPMD model (same program, multiple data)
- Annotate the code with parallelization and synchronization directives (pragmas)
 - Assumes programmers knows what they are doing
 - Code regions marked parallel are considered independent
 - Programmer is responsibility for protection against races
- Test and Debug

Simple OpenMP Example

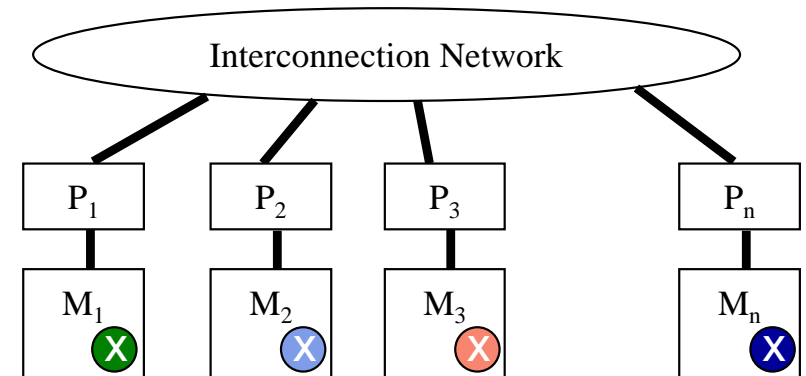
```
#pragma omp parallel
#pragma omp for
  for(i = 0; i < 12; i++)
    C[i] = A[i] + B[i];
```

- (data) **parallel pragma**
execute as many as there are processors (threads)
- **for pragma**
loop is parallel, can divide work (work-sharing)



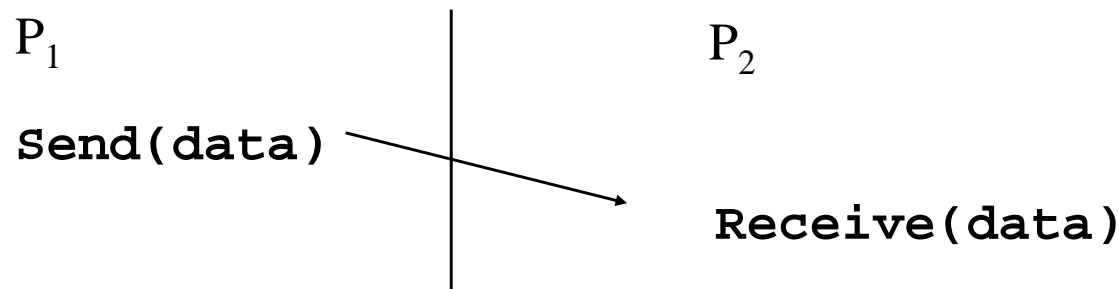
Programming Distributed Memory Processors

- Processors $1 \dots n$ ask for X
- There are n places to look
 - Each processor's memory has its own X
 - X s may vary
- For Processor 1 to look at Processor 2's X
 - Processor 1 has to request X from Processor 2
 - Processor 2 sends a copy of its own X to Processor 1
 - Processor 1 receives the copy
 - Processor 1 stores the copy in its own memory



Message Passing

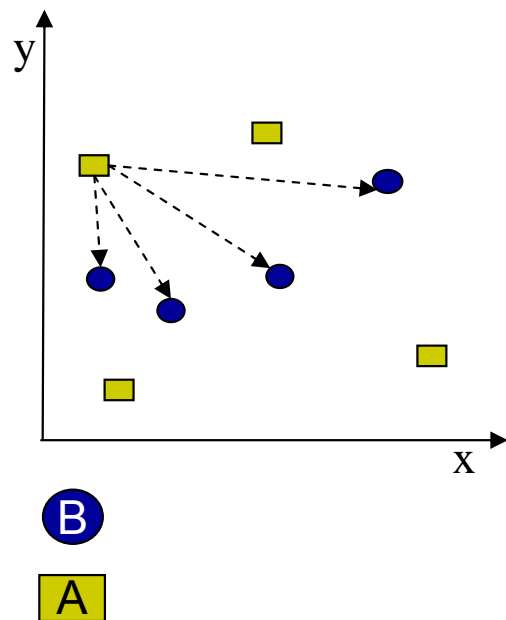
- Architectures with distributed memories use explicit communication to exchange data
 - Data exchange requires synchronization (cooperation) between senders and receivers



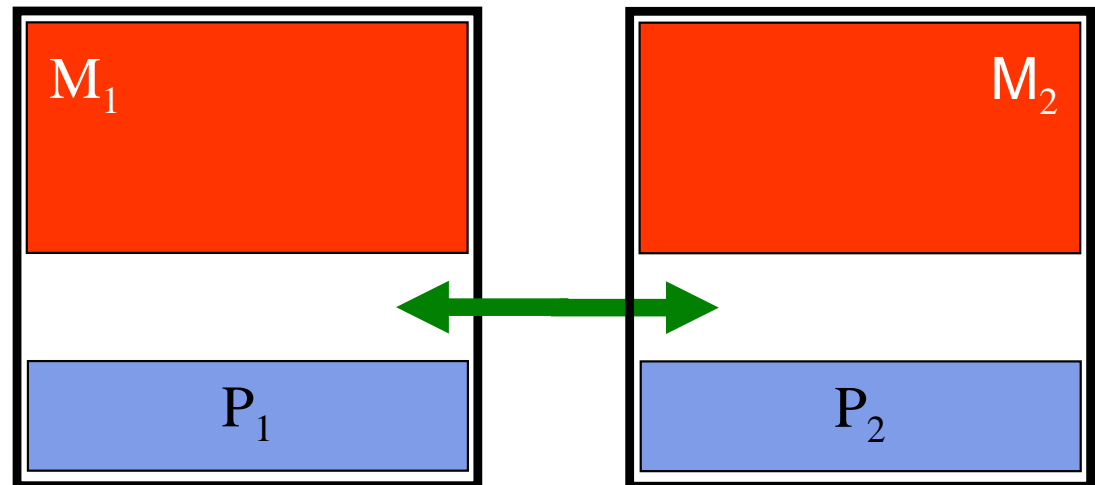
- How is “data” described
- How are processes identified
- Will receiver recognize or screen messages
- What does it mean for a send or receive to complete

Example Message Passing Program

- Calculate the distance from each point in $A[1..4]$ to every other point in $B[1..4]$ and store results to $C[1..4][1..4]$

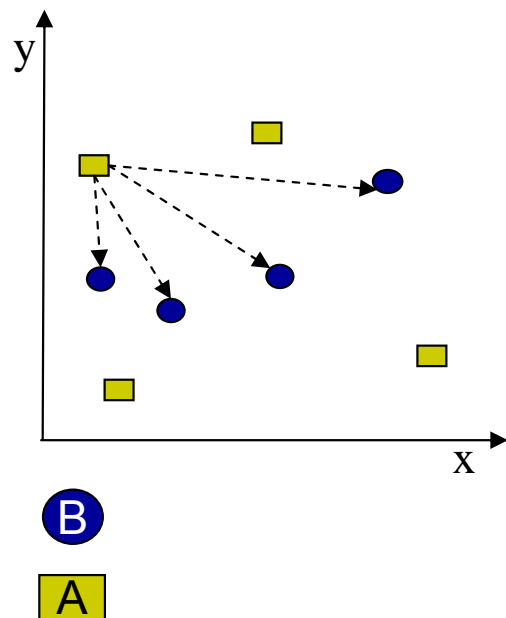


```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

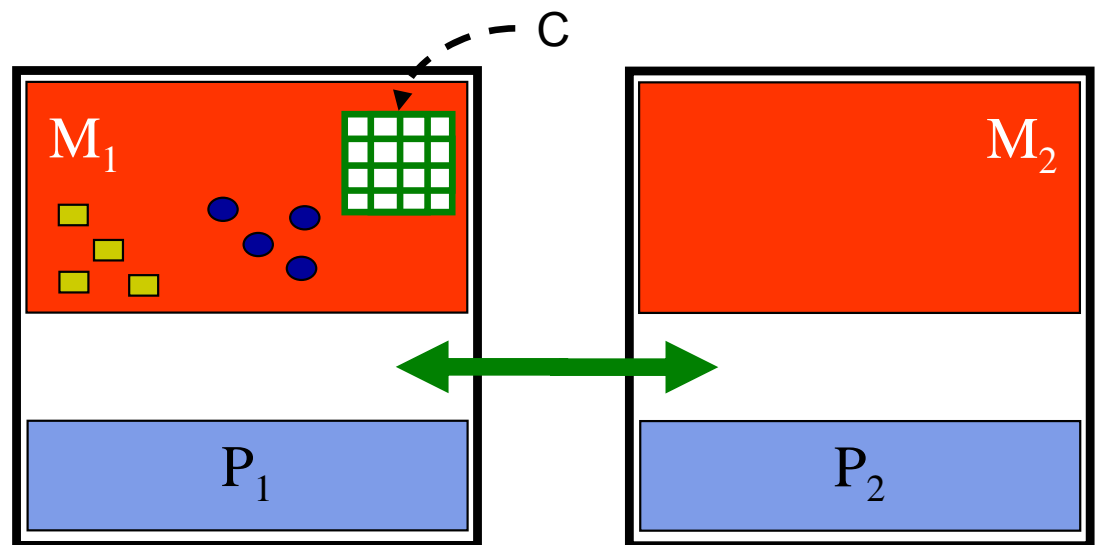


Example Message Passing Program

- Calculate the distance from each point in $A[1..4]$ to every other point in $B[1..4]$ and store results to $C[1..4][1..4]$



```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```



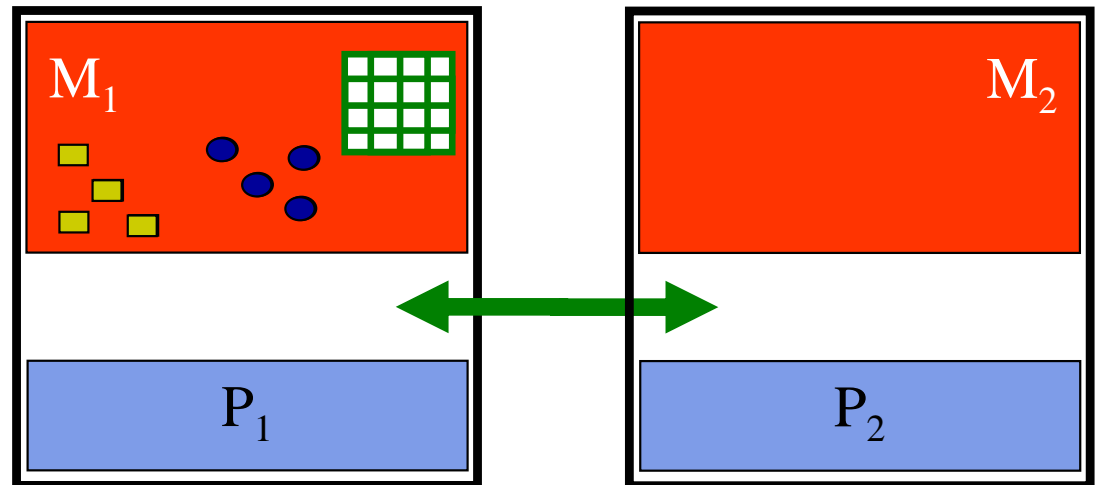
Example Message Passing Program

- Calculate the distance from each point in $A[1..4]$ to every other point in $B[1..4]$ and store results to $C[1..4][1..4]$

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

- Can break up work between the two processors

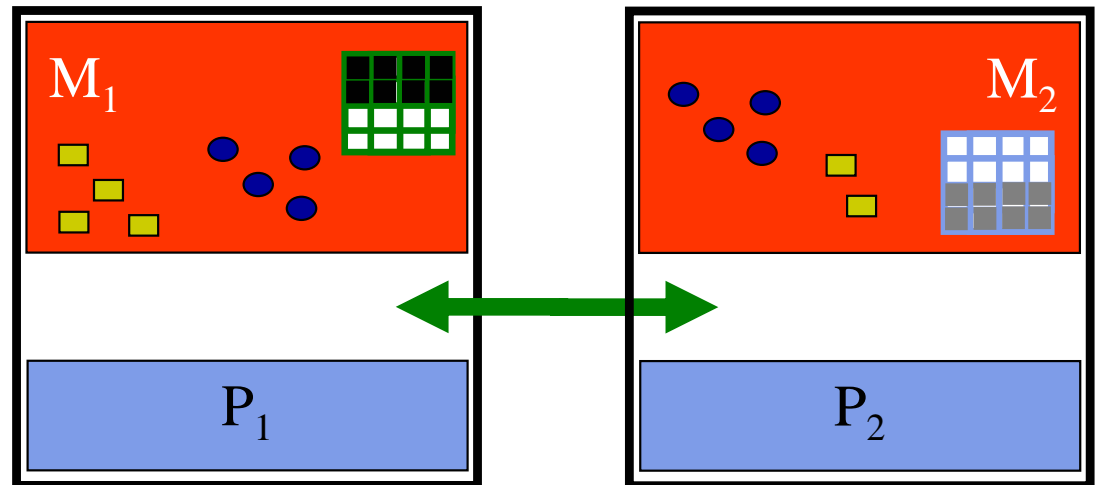
- P_1 sends data to P_2



Example Message Passing Program

- Calculate the distance from each point in $A[1..4]$ to every other point in $B[1..4]$ and store results to $C[1..4][1..4]$
- Can break up work between the two processors
 - P_1 sends data to P_2
 - P_1 and P_2 compute

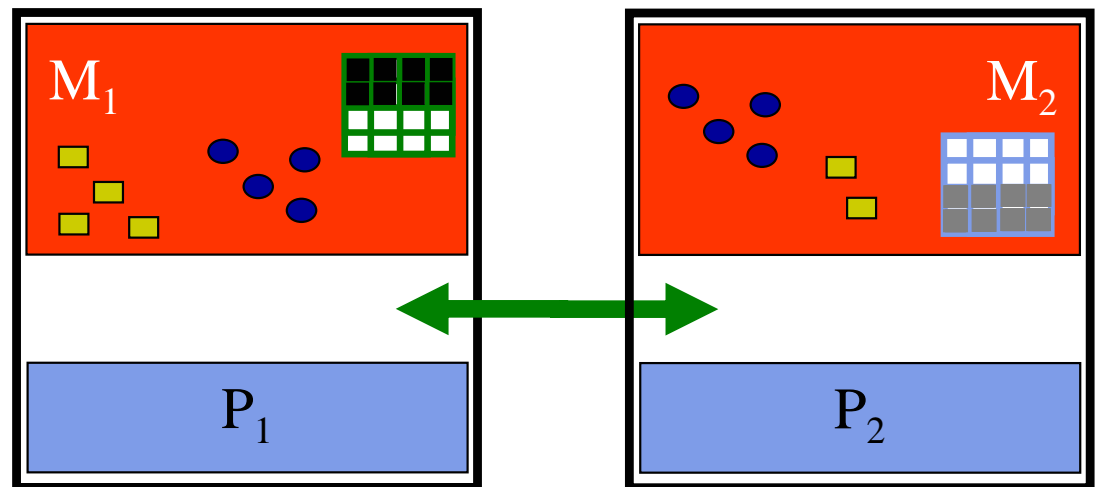
```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```



Example Message Passing Program

- Calculate the distance from each point in $A[1..4]$ to every other point in $B[1..4]$ and store results to $C[1..4][1..4]$
- Can break up work between the two processors
 - P_1 sends data to P_2
 - P_1 and P_2 compute
 - P_2 sends output to P_1

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```



Example Message Passing Program

processor 1

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

sequential

parallel with messages

processor 1

```
A[n] = {...}
B[n] = {...}

Send (A[n/2+1..n], B[1..n])

for (i = 1 to n/2)
  for (j = 1 to n)
    C[i][j] = distance(A[i], B[j])

Receive(C[n/2+1..n][1..n])
```

processor 2

```
A[n] = {...}
B[n] = {...}

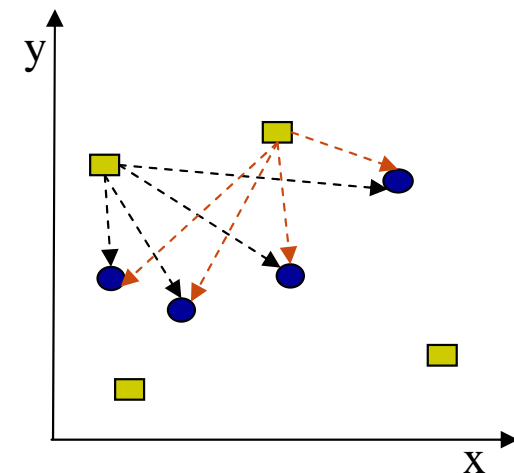
Receive(A[n/2+1..n], B[1..n])

for (i = n/2+1 to n)
  for (j = 1 to n)
    C[i][j] = distance(A[i], B[j])

Send (C[n/2+1..n][1..n])
```

Performance Analysis

- Distance calculations between points are independent of each other
 - Dividing the work between two processors \rightarrow 2x speedup
 - Dividing the work between four processors \rightarrow 4x speedup
- Communication
 - 1 copy of $\mathbf{B}[\]$ sent to each processor
 - 1 copy of **subset** of $\mathbf{A}[\]$ to each processor
- Granularity of $\mathbf{A}[\]$ subsets directly impact communication costs
 - Communication is not free

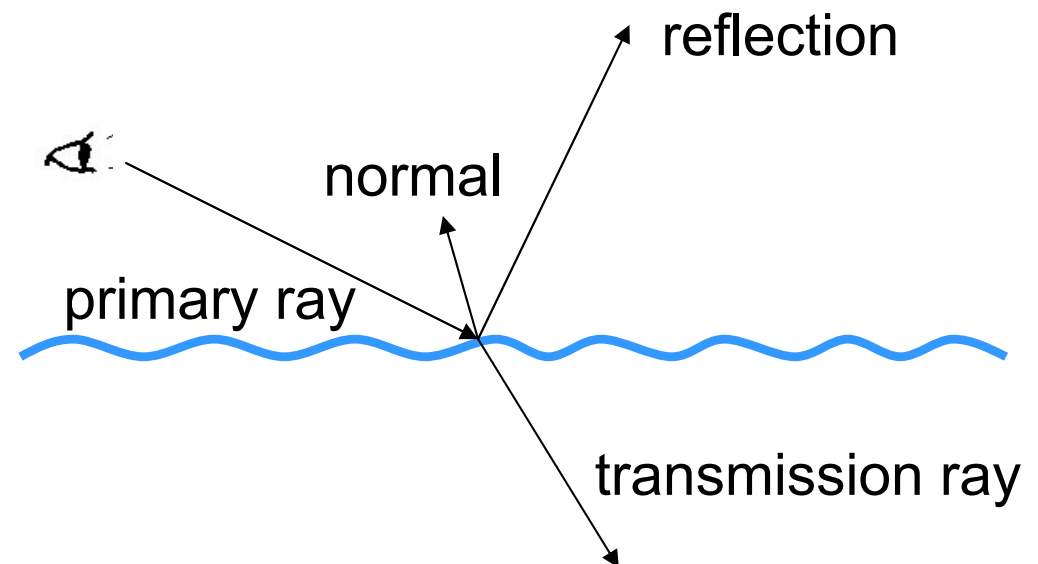
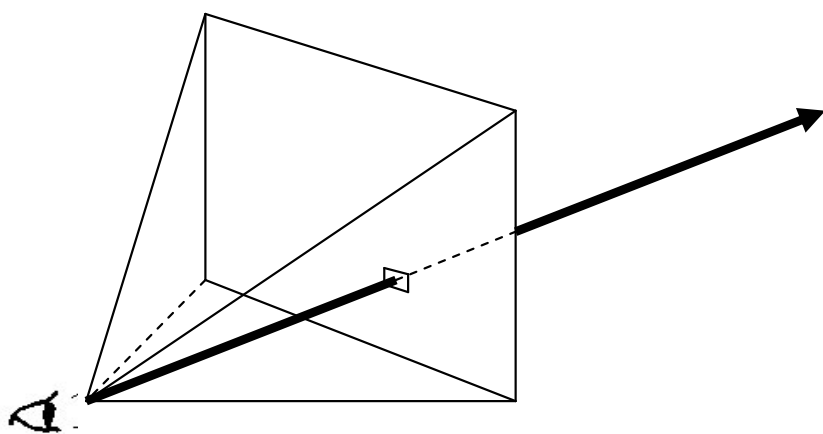


Understanding Performance

- What factors affect performance of parallel programs?
- **Coverage** or extent of parallelism in algorithm
- **Granularity** of partitioning among processors
- **Locality** of computation and communication

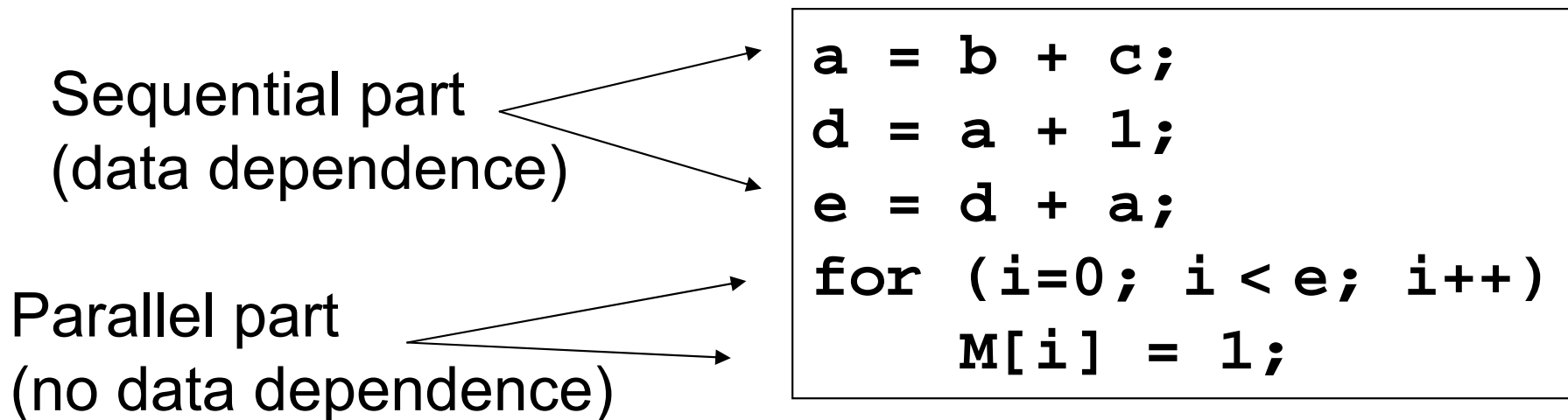
Rendering Scenes by Ray Tracing

- Shoot rays into scene through pixels in image plane
- Follow their paths
 - Rays bounce around as they strike objects
 - Rays generate new rays
- Result is color and opacity for that pixel
- Parallelism across rays



Limits to Performance Scalability

- Not all programs are “embarrassingly” parallel
- Programs have sequential parts and parallel parts

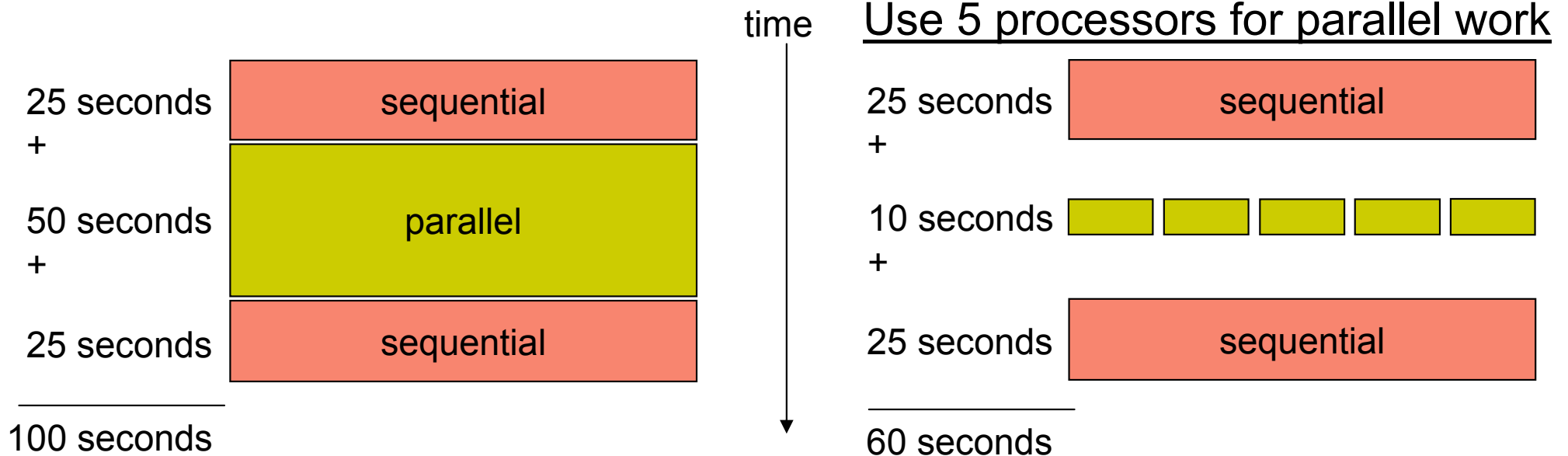


Coverage

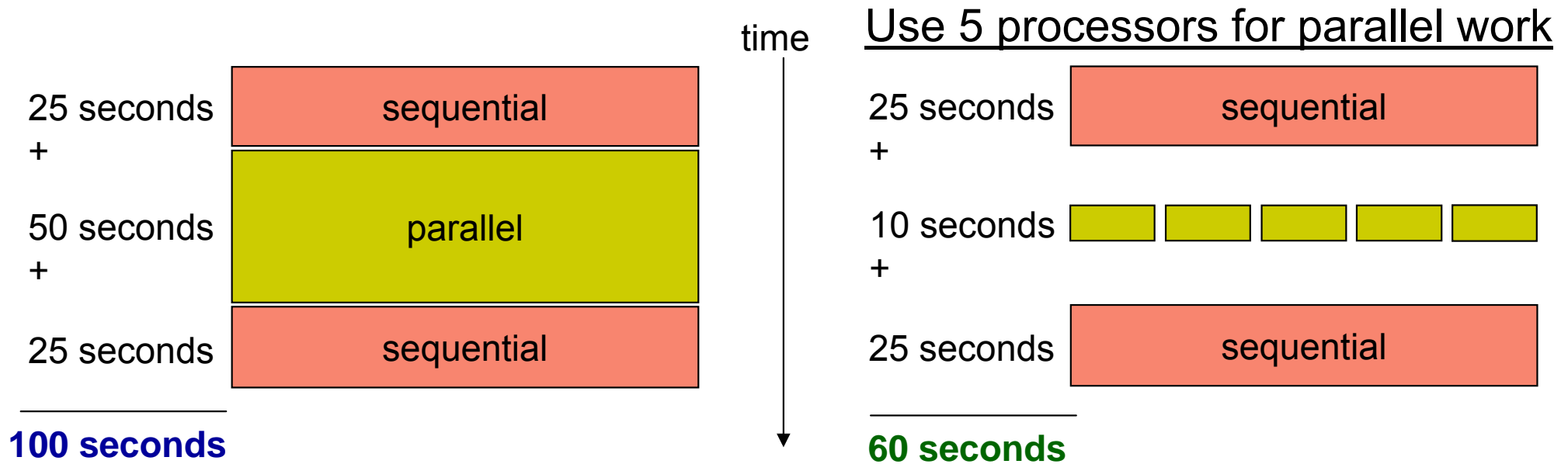
- **Amdahl's Law:** *The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.*
 - Demonstration of the law of diminishing returns

Amdahl's Law

- Potential program speedup is defined by the fraction of code that can be parallelized



Amdahl's Law



- Speedup = old running time / new running time
= 100 seconds / 60 seconds
= 1.67
(parallel version is 1.67 times faster)

Amdahl's Law

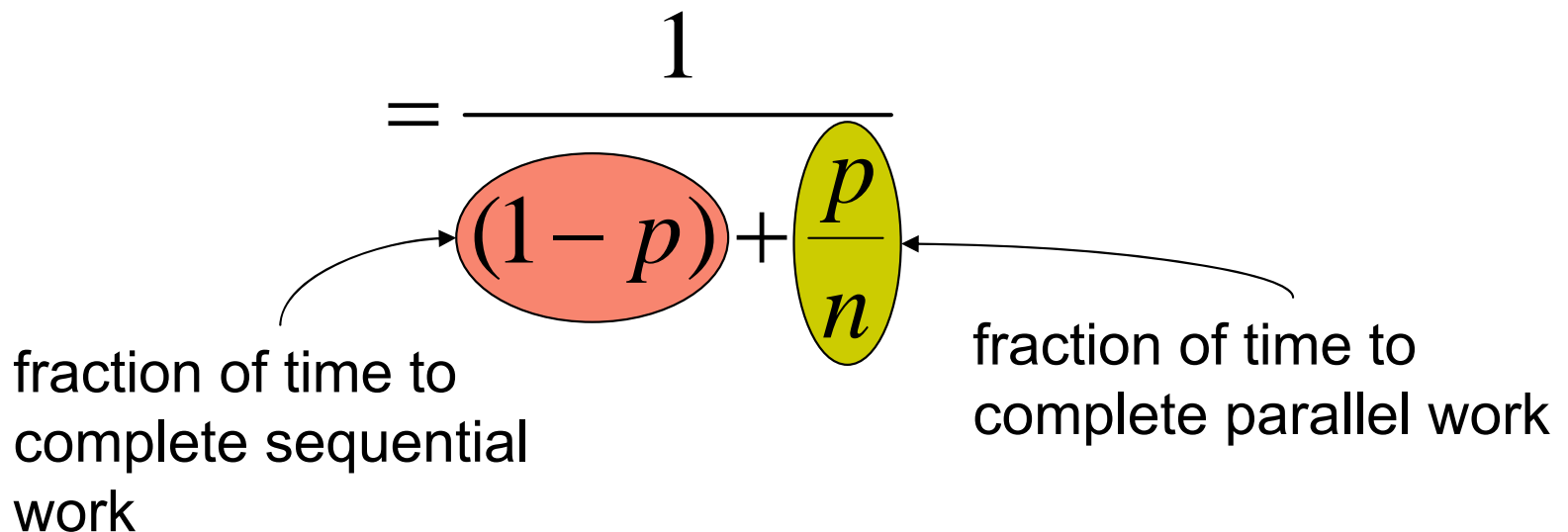
- p = fraction of work that can be parallelized
- n = the number of processor

$$\textit{speedup} = \frac{\text{old running time}}{\text{new running time}}$$

$$= \frac{1}{(1-p) + \frac{p}{n}}$$

fraction of time to complete sequential work

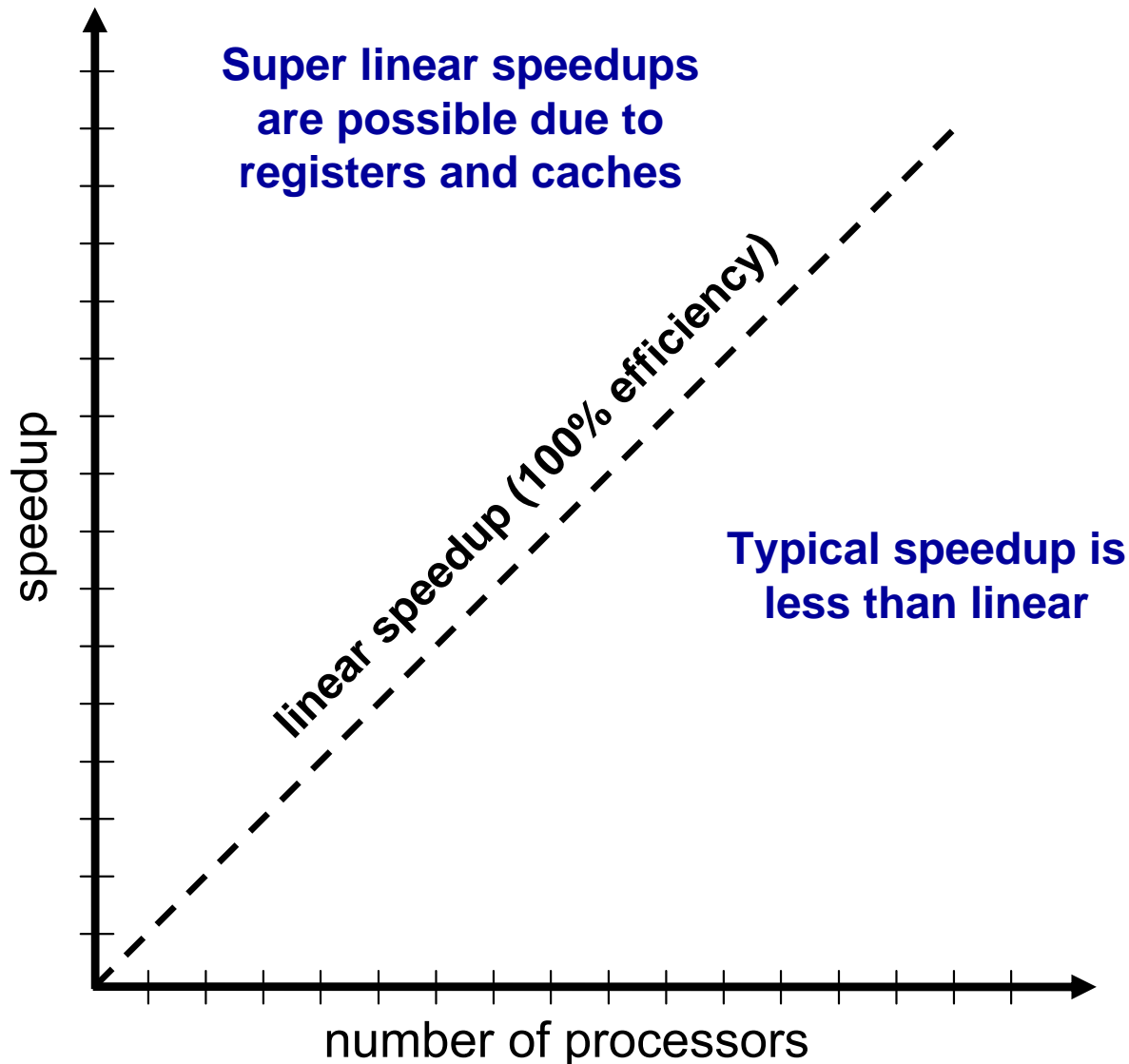
fraction of time to complete parallel work



Implications of Amdahl's Law

- Speedup tends to $\frac{1}{1-p}$ as number of processors tends to infinity
- Parallel programming is worthwhile when programs have a lot of work that is parallel in nature

Performance Scalability



Understanding Performance

- **Coverage** or extent of parallelism in algorithm
- **Granularity** of partitioning among processors
- **Locality** of computation and communication

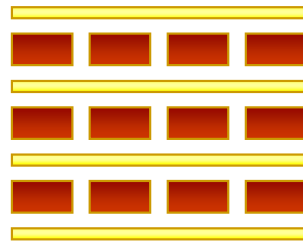
Granularity

- Granularity is a qualitative measure of the ratio of computation to communication
- Computation stages are typically separated from periods of communication by synchronization events

Fine vs. Coarse Granularity

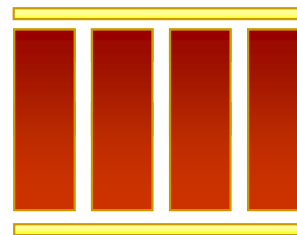
- Fine-grain Parallelism

- Low computation to communication ratio
- Small amounts of computational work between communication stages
- Less opportunity for performance enhancement
- High communication overhead



- Coarse-grain Parallelism

- High computation to communication ratio
- Large amounts of computational work between communication events
- More opportunity for performance increase
- Harder to load balance efficiently



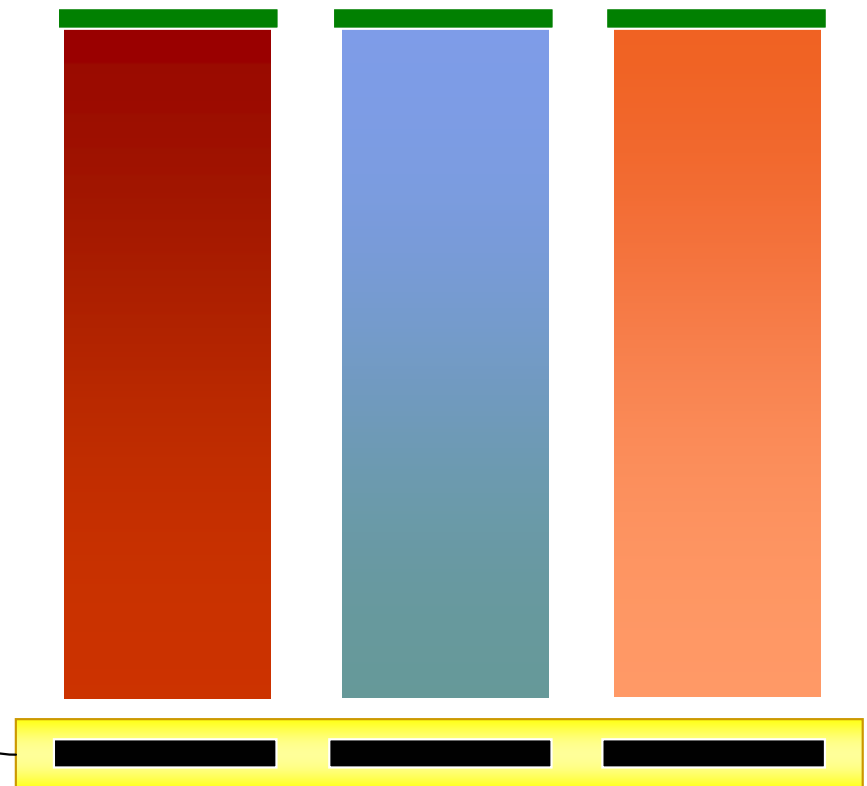
The Load Balancing Problem

- Processors that finish early have to wait for the processor with the largest amount of work to complete
 - Leads to idle time, lowers utilization

```
// PPU tells all SPEs to start
for (int i = 0; i < n; i++) {
    spe_write_in_mbox(id[i], <message>);
}
```

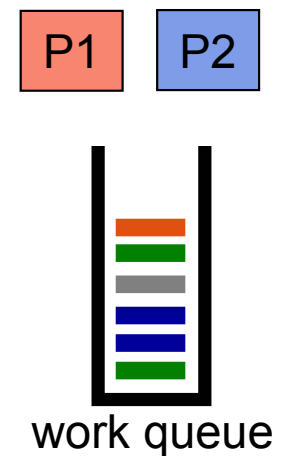
```
// PPU waits for SPEs to send completion message
for (int i = 0; i < n; i++) {
    while (spe_stat_out_mbox(id[i]) == 0);
    spe_read_out_mbox(id[i]);
}
```

communication stage (synchronization)



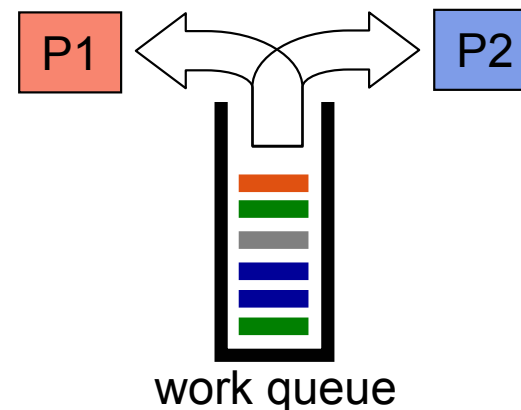
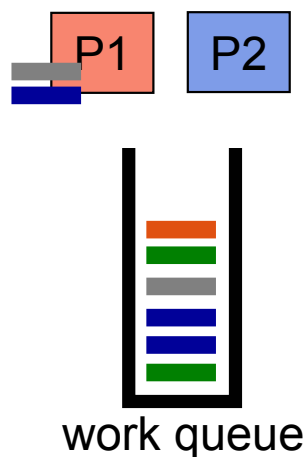
Static Load Balancing

- Programmer make decisions and assigns a fixed amount of work to each processing core a priori
- Works well for homogeneous multicores
 - All core are the same
 - Each core has an equal amount of work
- Not so well for heterogeneous multicores
 - Some cores may be faster than others
 - Work distribution is uneven



Dynamic Load Balancing

- When one core finishes its allocated work, it takes on work from core with the heaviest workload
- Ideal for codes where work is uneven, and in heterogeneous multicore



Granularity and Performance Tradeoffs

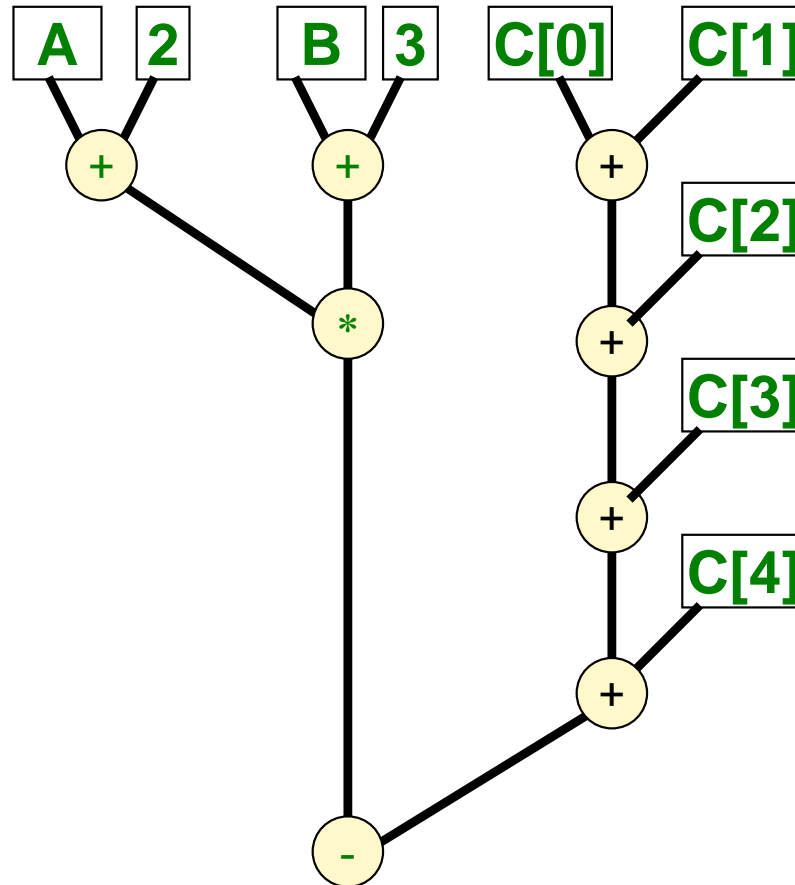
1. Load balancing

- How well is work distributed among cores?

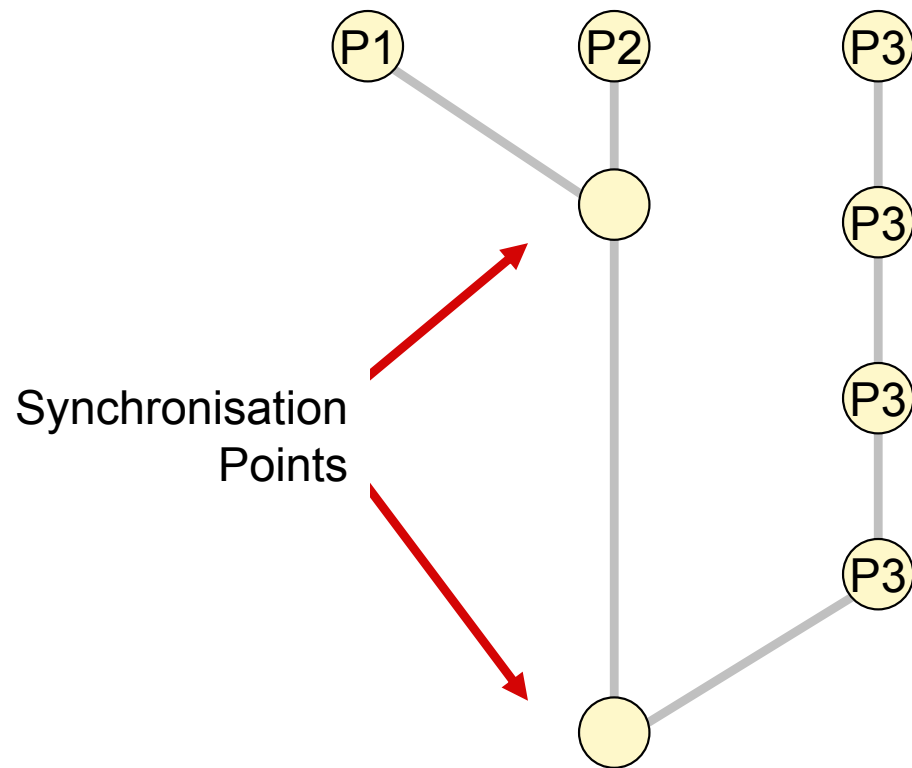
2. Synchronization

- Are there ordering constraints on execution?

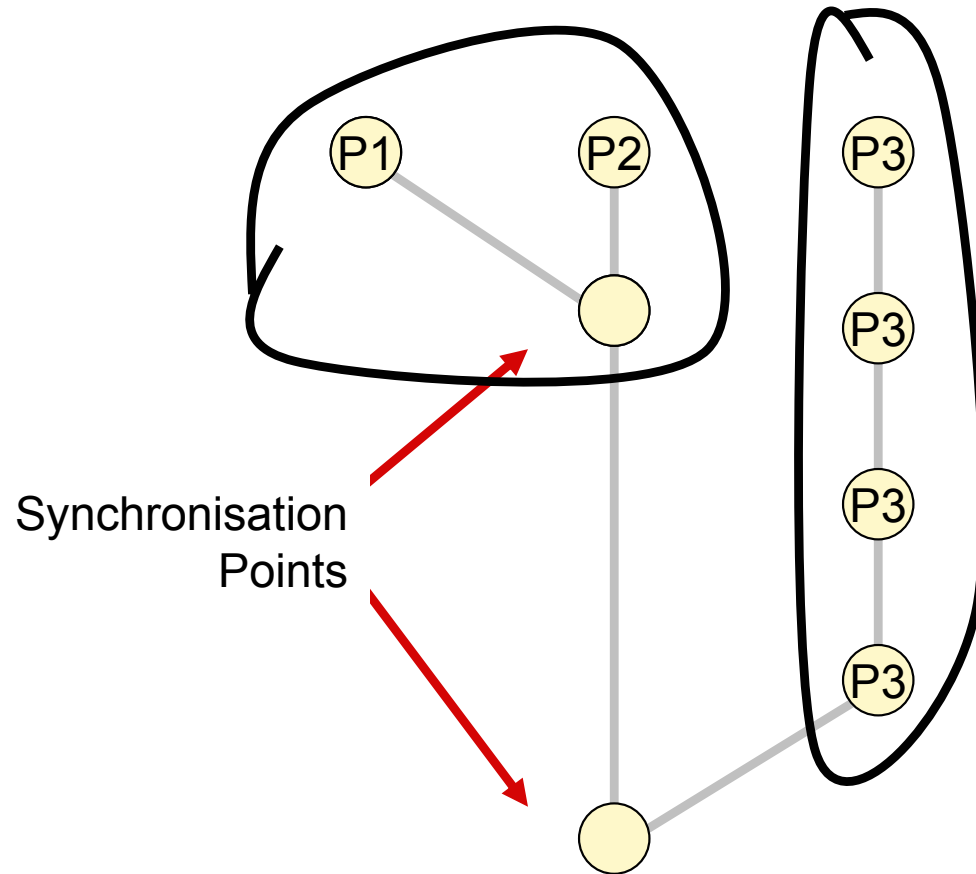
Data Dependence Graph



Dependence and Synchronization



Synchronization Removal



Granularity and Performance Tradeoffs

1. Load balancing

- How well is work distributed among cores?

2. Synchronization

- Are there ordering constraints on execution?

3. Communication

- Communication is not cheap!

Communication Cost Model

$$C = f * (o + l + \frac{n/m}{B} + t - \text{overlap})$$

The diagram illustrates the Communication Cost Model equation $C = f * (o + l + \frac{n/m}{B} + t - \text{overlap})$. Each term in the equation is annotated with a description:

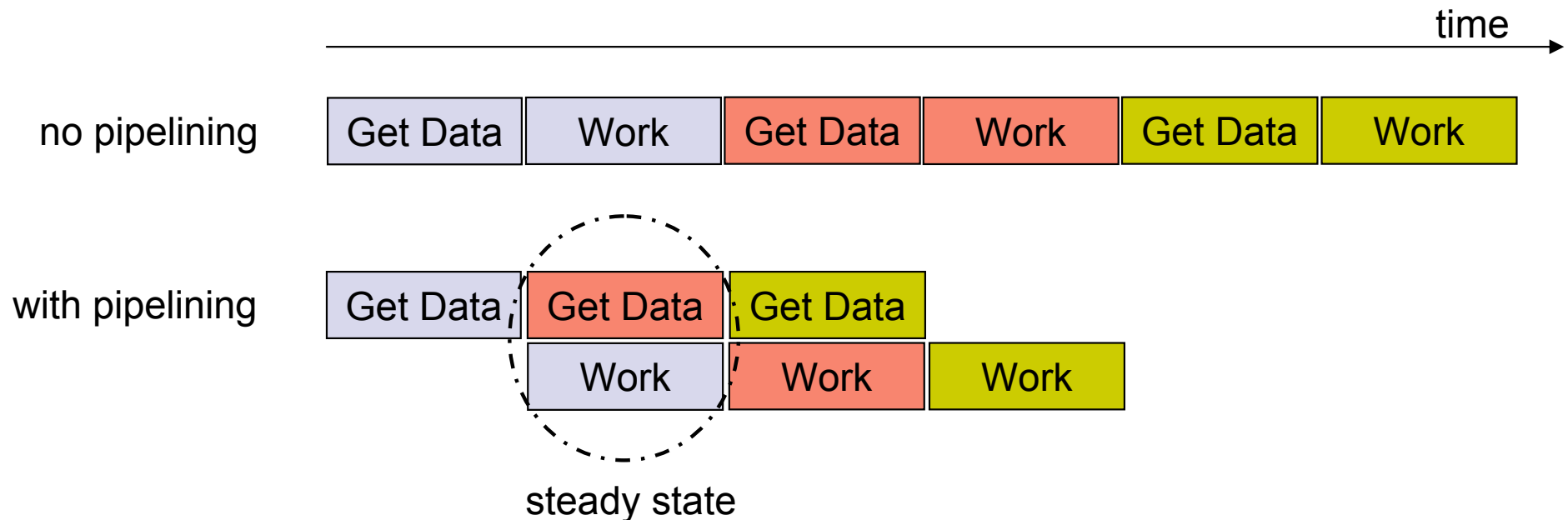
- f : frequency of messages
- o : overhead per message (at both ends)
- l : network delay per message
- n/m : total data sent (numerator) and number of messages (denominator)
- B : bandwidth along path (determined by network)
- t : cost induced by contention per message
- overlap : amount of latency hidden by concurrency with computation

Types of Communication

- Cores exchange data or control messages
 - Cell examples: DMA vs. Mailbox
- Control messages are often short
- Data messages are relatively much larger

Overlapping Messages and Computation

- Computation and communication concurrency can be achieved with pipelining
 - Think instruction pipelining in superscalars



Overlapping Messages and Computation

- Computation and communication concurrency can be achieved with pipelining
 - Think instruction pipelining in superscalars
- Essential for performance on Cell and similar distributed memory multicores

Cell buffer pipelining example

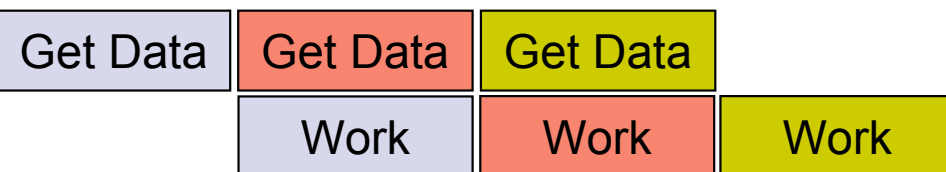
```
// Start transfer for first buffer
id = 0;
mfc_get(buf[id], addr, BUFFER_SIZE, id, 0, 0);
id ^= 1;

while (!done) {
  // Start transfer for next buffer
  addr += BUFFER_SIZE;
  mfc_get(buf[id], addr, BUFFER_SIZE, id, 0, 0);

  // Wait until previous DMA request finishes
  id ^= 1;
  mfc_write_tag_mask(1 << id);
  mfc_read_tag_status_all();

  // Process buffer from previous iteration
  process_data(buf[id]);
}
```

time →



Communication Patterns

- With message passing, programmer has to understand the computation and orchestrate the communication accordingly
 - Point to Point
 - Broadcast (one to all) and Reduce (all to one)
 - All to All (each processor sends its data to all others)
 - Scatter (one to several) and Gather (several to one)

A Message Passing Library Specification

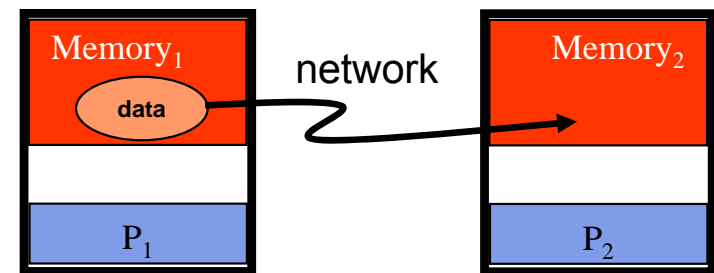
- MPI: specification
 - Not a language or compiler specification
 - Not a specific implementation or product
 - SPMD model (same program, multiple data)
- For parallel computers, clusters, and heterogeneous networks, multicores
- Full-featured
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication

Where Did MPI Come From?

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
 - Did not address the full spectrum of issues
 - Lacked vendor support
 - Were not implemented at the most efficient level
- The MPI Forum organized in 1992 with broad participation
 - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
 - Portability library writers: PVM, p4
 - Users: application scientists and library writers
 - Finished in 18 months

Point-to-Point

- Basic method of communication between two processors
 - Originating processor "sends" message to destination processor
 - Destination processor then "receives" the message
- The message commonly includes
 - Data or other information
 - Length of the message
 - Destination address and possibly a tag



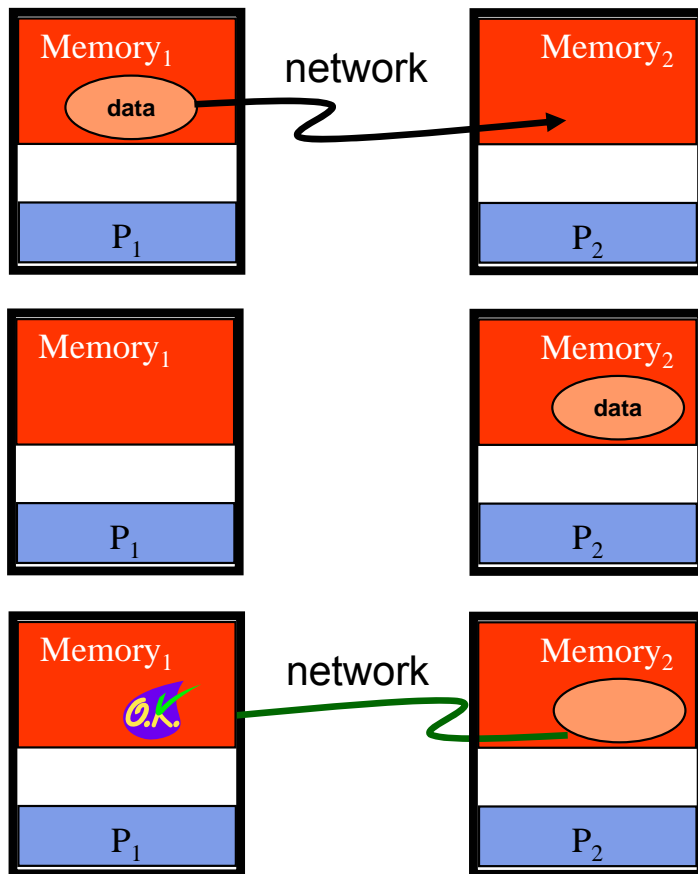
Cell "send" and "receive" commands

```
mfc_get(destination LS addr,  
        source memory addr,  
        # bytes,  
        tag,  
        <...>)
```

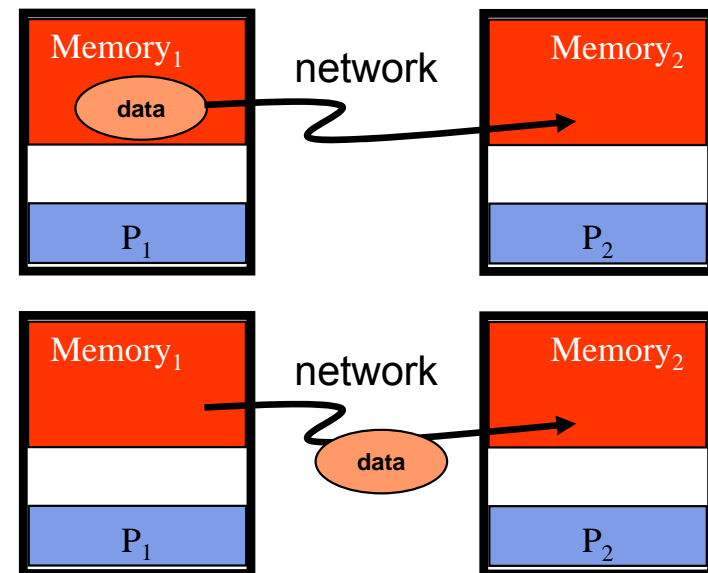
```
mfc_put(source LS addr,  
        destination memory addr,  
        # bytes,  
        tag,  
        <...>)
```

Synchronous vs. Asynchronous Messages

- Synchronous send
 - Sender notified when message is received



- Asynchronous send
 - Sender only knows that message is sent




Blocking vs. Non-Blocking Messages

- Blocking messages
 - Sender waits until message is transmitted: **buffer is empty**
 - Receiver waits until message is received: **buffer is full**
 - Potential for deadlock
- Non-blocking
 - Processing continues even if message hasn't been transmitted
 - Avoid idle time and deadlocks

Cell blocking mailbox “send”

```
// SPE does some work
...
// SPE notifies PPU that task has completed
spu_write_out_mbox(<message>);

// SPE does some more work
...
// SPE notifies PPU that task has completed
spu_write_out_mbox(<message>); 
```

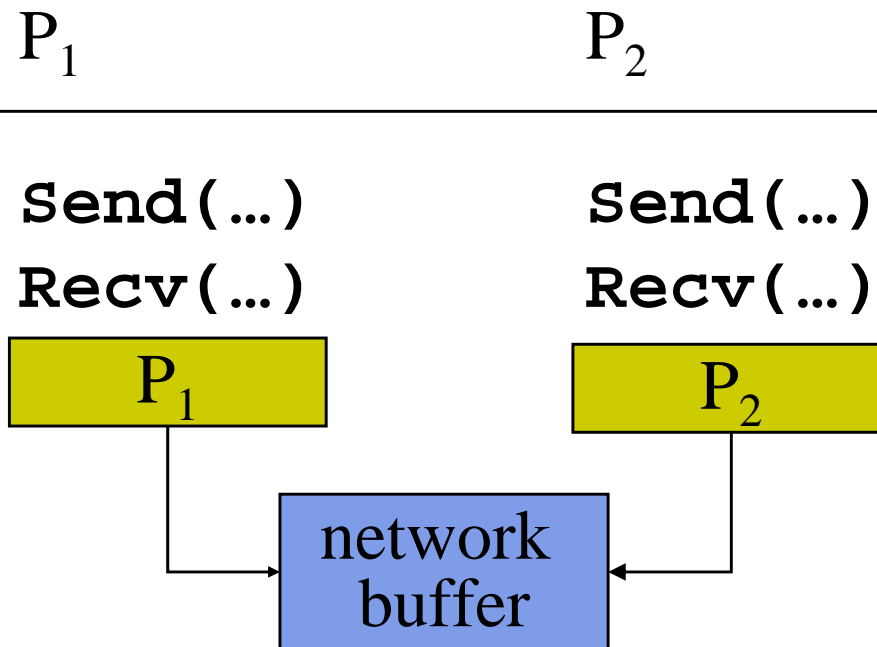
Cell non-blocking data “send” and “wait”

```
// DMA back results
mfc_put(data, cb.data_addr, data_size, ...);

// Wait for DMA completion
mfc_read_tag_status_all();
```


Sources of Deadlocks

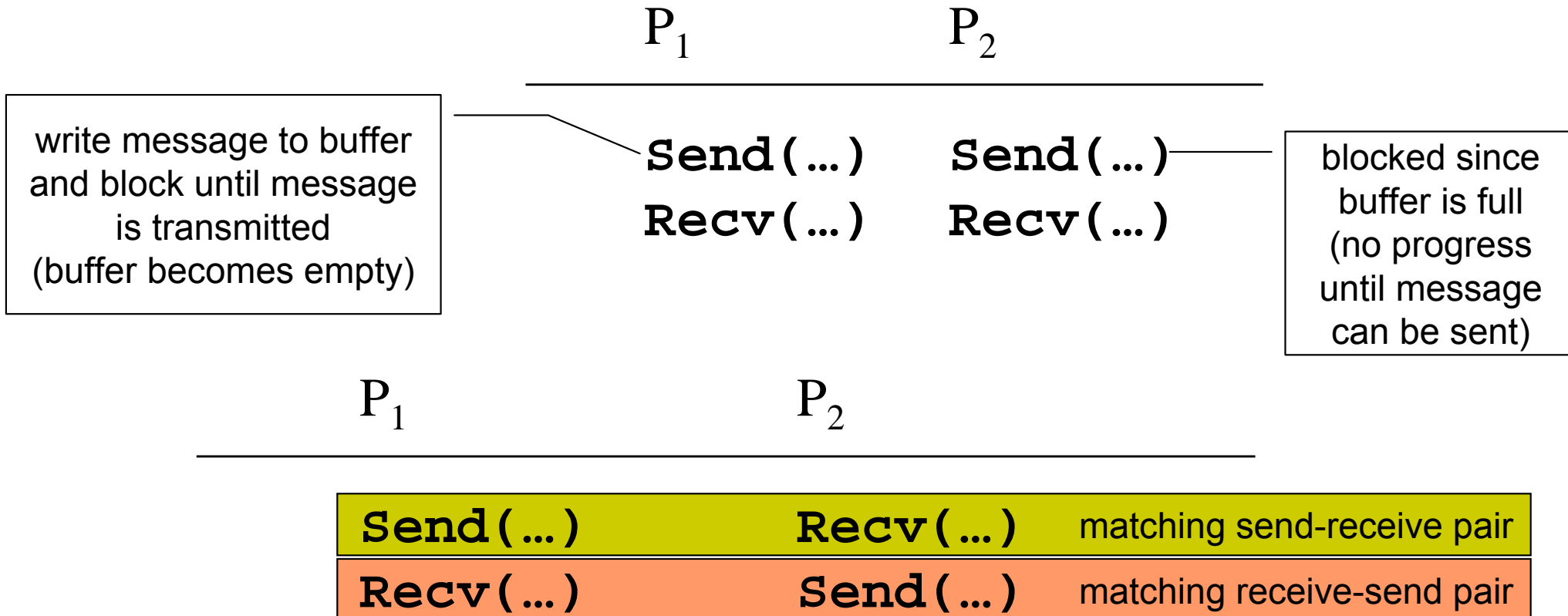
- If there is insufficient buffer capacity, sender waits until additional storage is available
- What happens with this code?



- Depends on length of message and available buffer

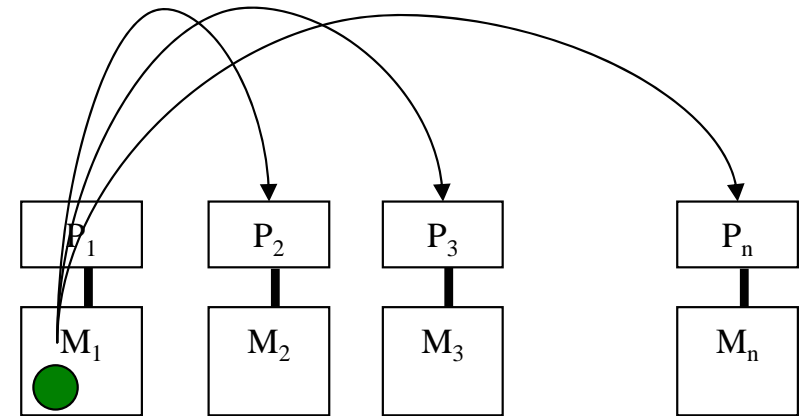
Solutions

- Increasing local or network buffering
- Order the sends and receives more carefully



Broadcast

- One processor sends the same information to many other processors
 - `MPI_BCAST`



```
for (i = 1 to n)
  for (j = 1 to n)
    C[i][j] = distance(A[i], B[j])
```

```
A[n] = {...}
```

```
B[n] = {...}
```

```
Broadcast(B[1..n])
```

```
for (i = 1 to n)
```

```
  // round robin distribute B
```

```
  // to m processors
```

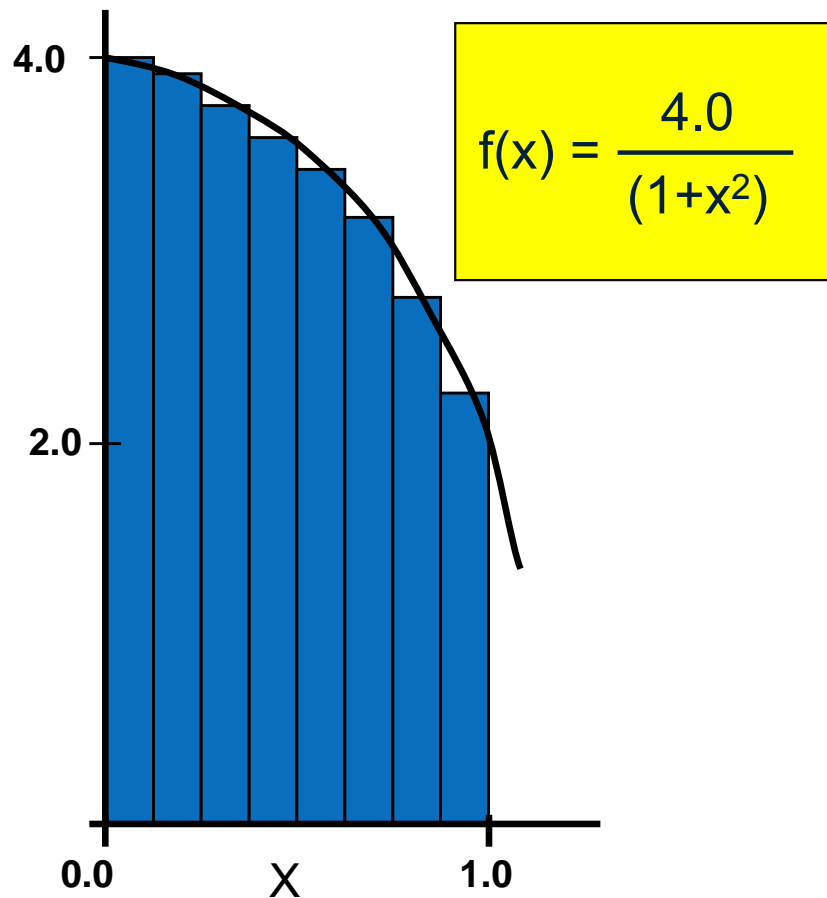
```
  Send(A[i % m])
```

```
...
```

Reduction

- Example: every processor starts with a value and needs to know the sum of values stored on all processors
- A reduction combines data from all processors and returns it to a single process
 - **MPI_REDUCE**
 - Can apply any associative operation on gathered data
 - ADD, OR, AND, MAX, MIN, etc.
 - No processor can finish reduction before each processor has contributed a value
- **BCAST/REDUCE** can reduce programming complexity and may be more efficient in some programs

Example: Parallel Numerical Integration



```
static long num_steps = 100000;

void main()
{
    int i;
    double pi, x, step, 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);
}
```

Computing Pi With Integration (OpenMP)

```
static long num_steps = 100000;

void main()
{
    int i;
    double pi, x, step, 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);
}
```

- Which variables are shared?
 - `step`
- Which variables are private?
 - `x`
- Which variables does reduction apply to?
 - `sum`

Computing Pi With Integration (MPI)

```
static long num_steps = 100000;

void main(int argc, char* argv[])
{
    int i_start, i_end, i, myid, numprocs;
    double pi, mypi, x, step, sum = 0.0;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    MPI_BCAST(&num_steps, 1, MPI_INT, 0, MPI_COMM_WORLD);

    i_start = my_id * (num_steps/numprocs)
    i_end = i_start + (num_steps/numprocs)

    step = 1.0 / (double) num_steps;
    for (i = i_start; i < i_end; i++) {
        x = (i + 0.5) * step
        sum = sum + 4.0 / (1.0 + x*x);
    }
    mypi = step * sum;

    MPI_REDUCE(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

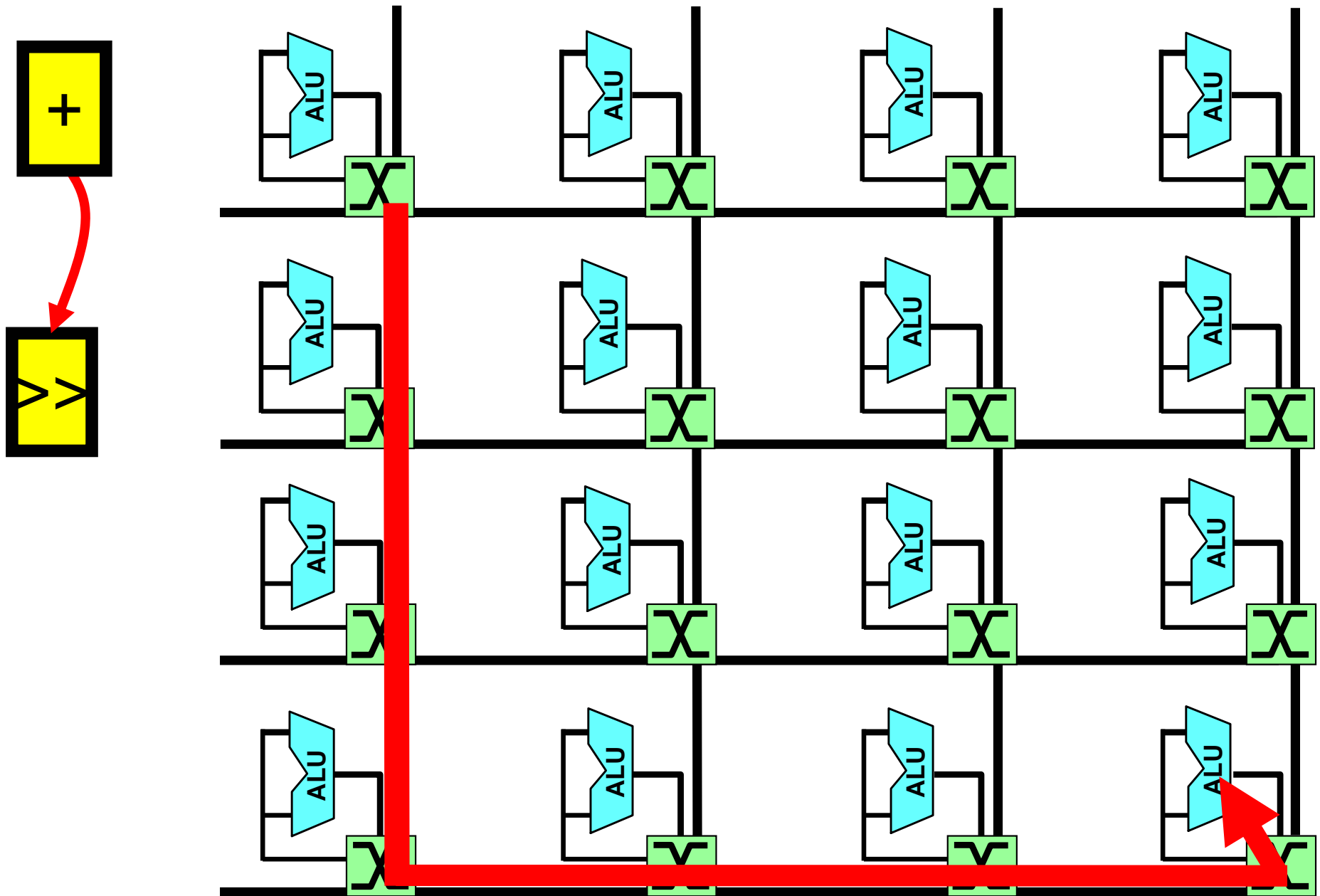
    if (myid == 0)
        printf("Pi = %f\n", pi);

    MPI_Finalize();
}
```

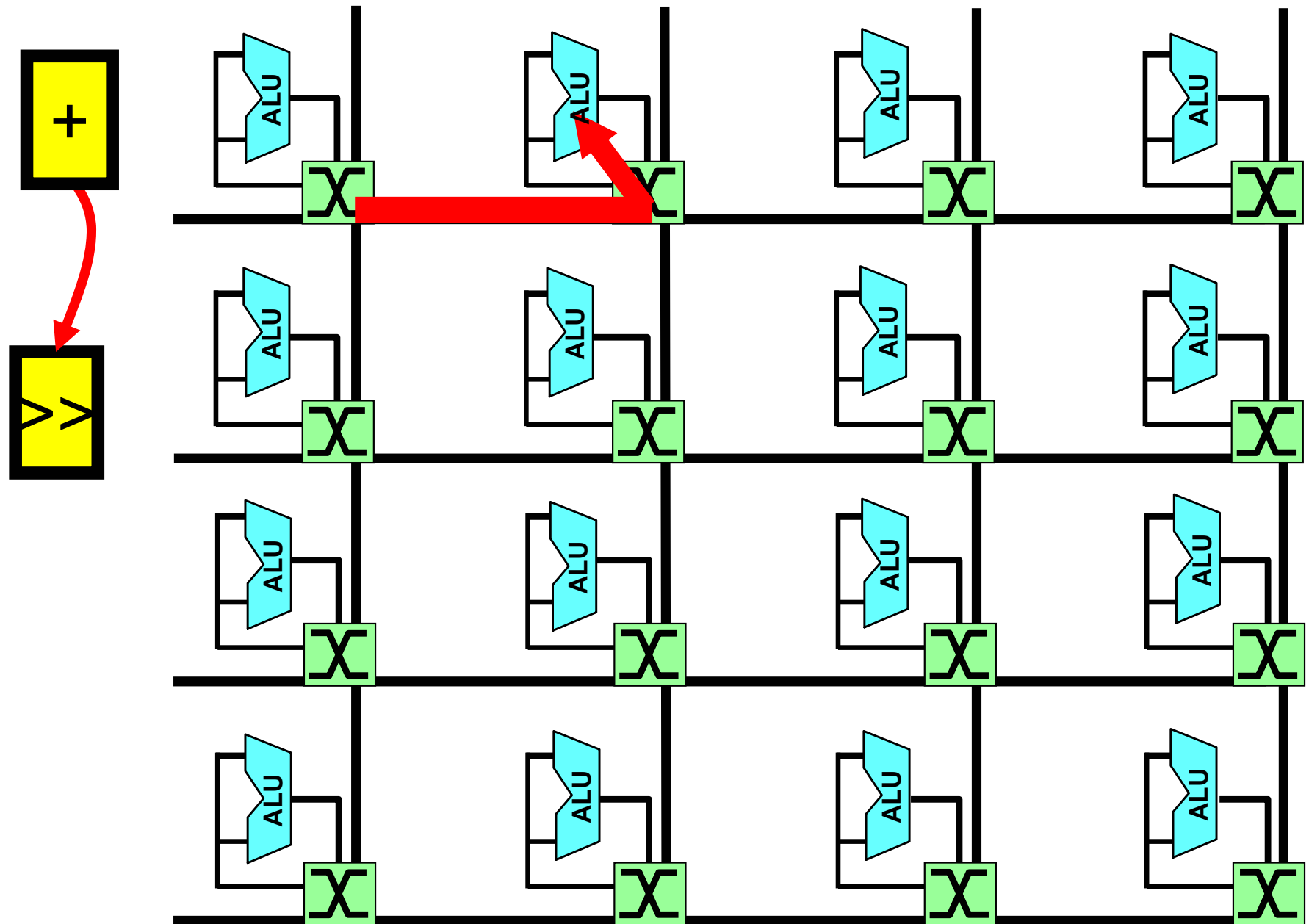
Understanding Performance

- **Coverage** or extent of parallelism in algorithm
- **Granularity** of data partitioning among processors
- **Locality** of computation and communication

Locality in Communication (Message Passing)

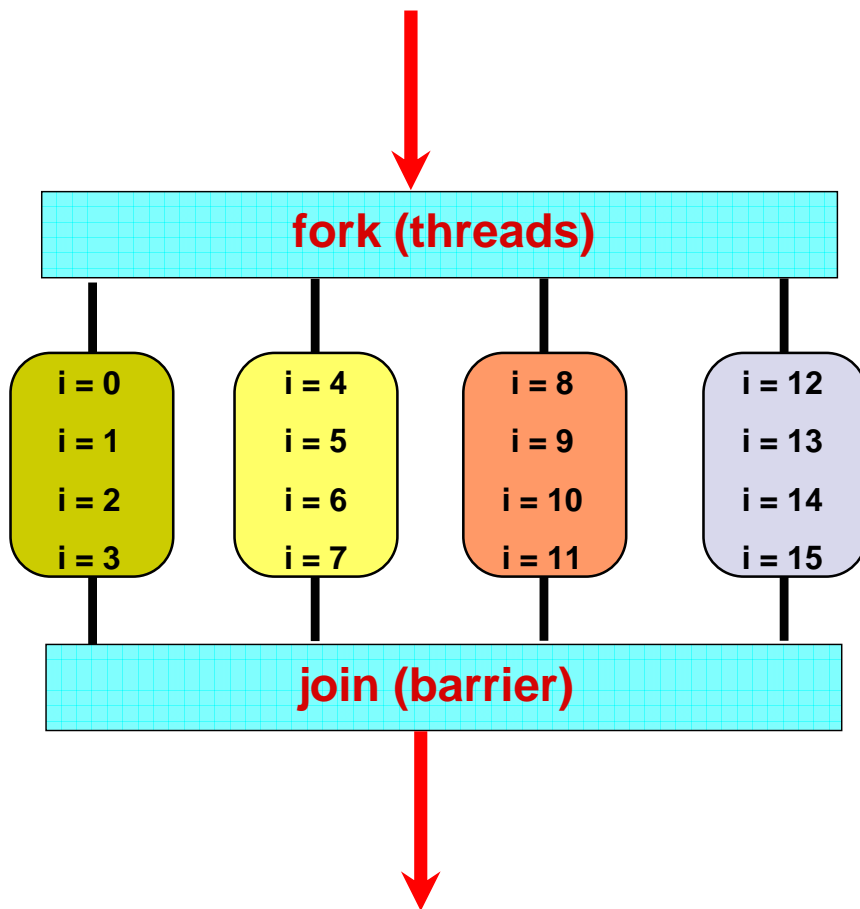


Exploiting Communication Locality



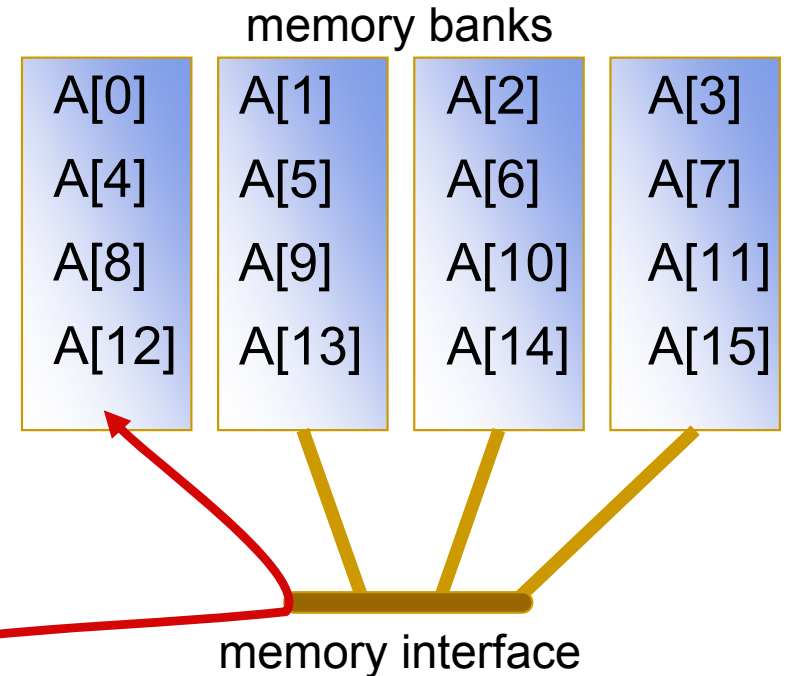
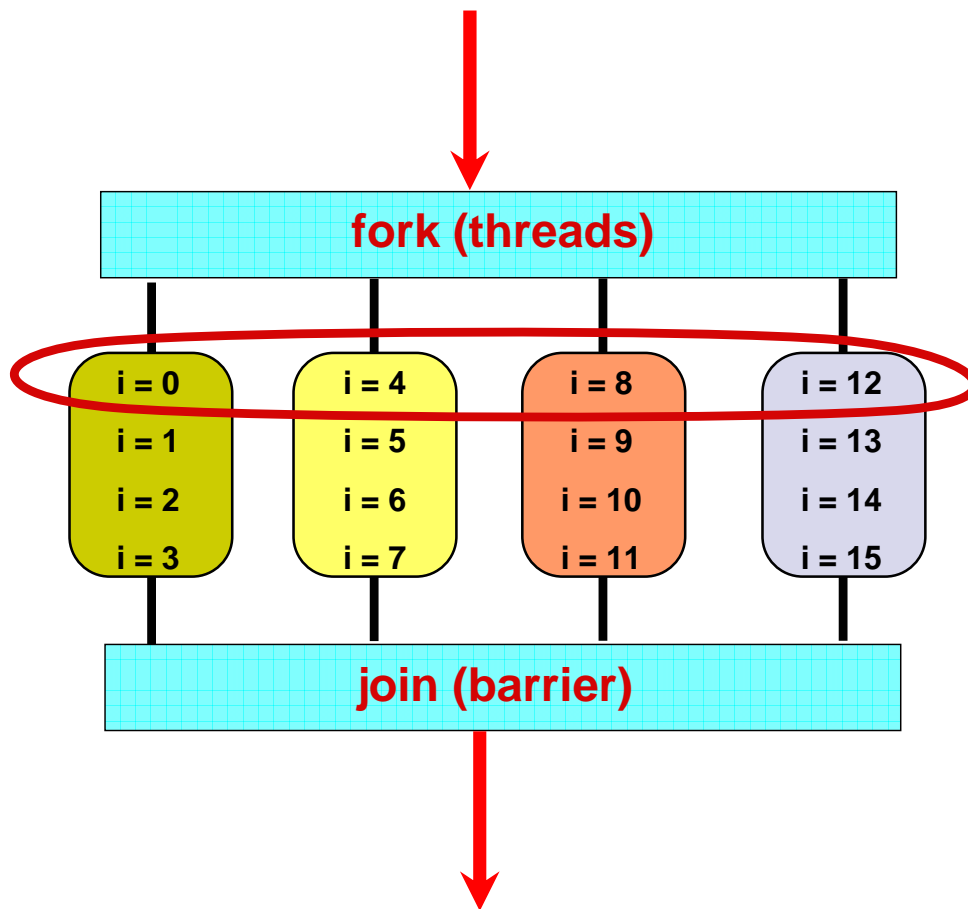
Locality of Memory Accesses (Shared Memory)

```
for (i = 0; i < 16; i++)  
    C[i] = A[i] + ...;
```



Locality of Memory Accesses (Shared Memory)

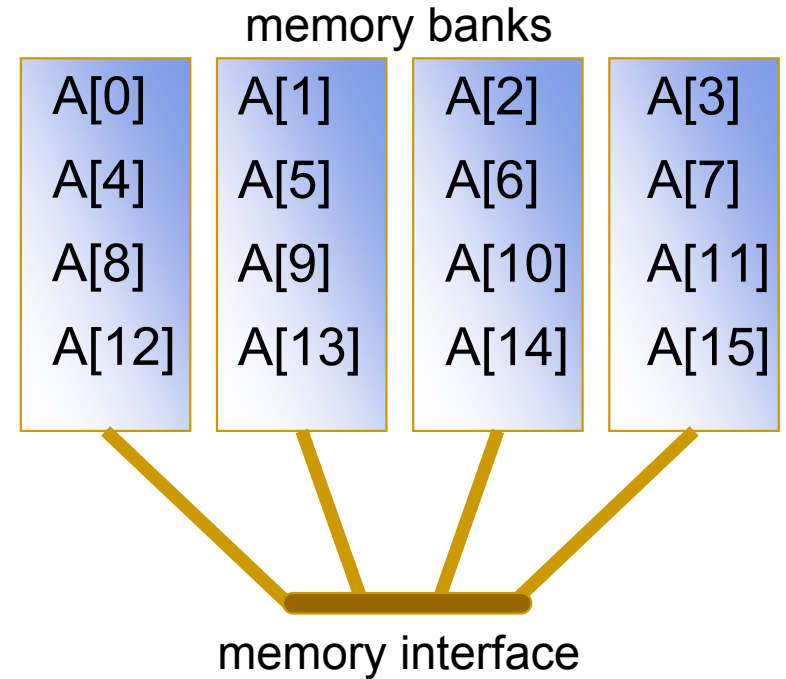
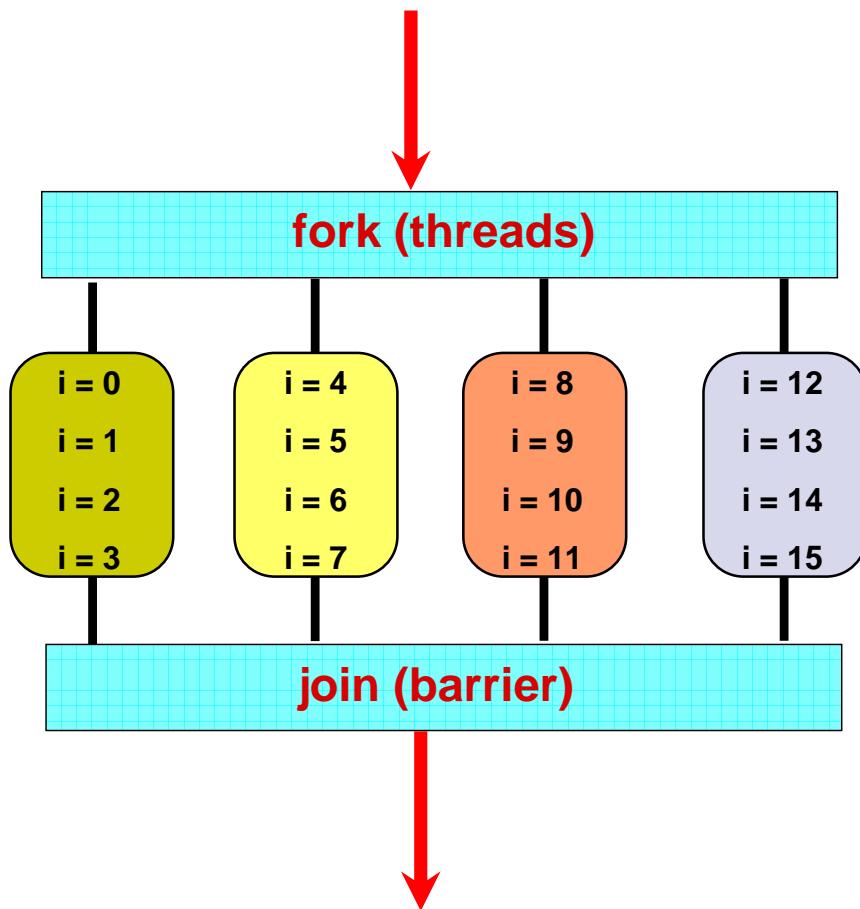
```
for (i = 0; i < 16; i++)  
    C[i] = A[i] + ...;
```



- Parallel computation is serialized due to memory contention and lack of bandwidth

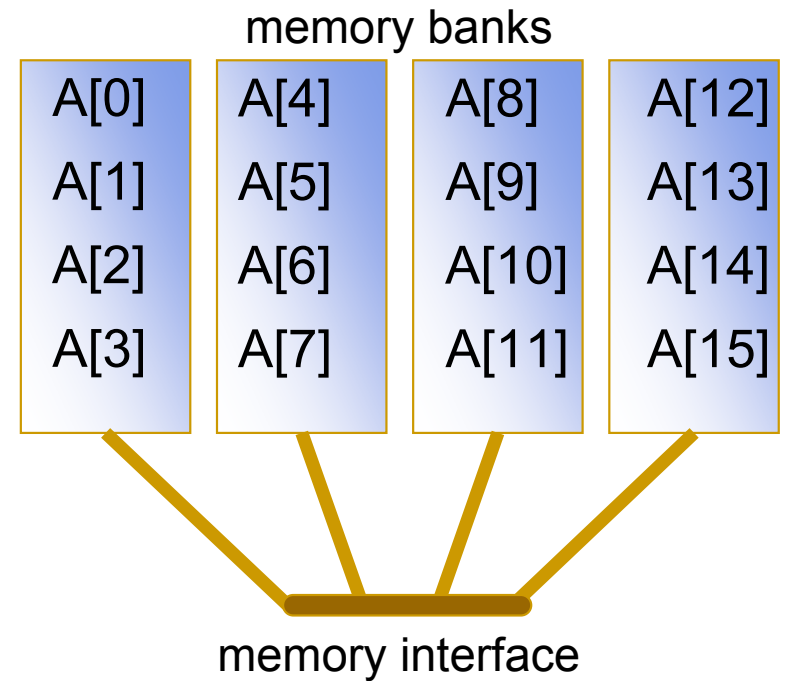
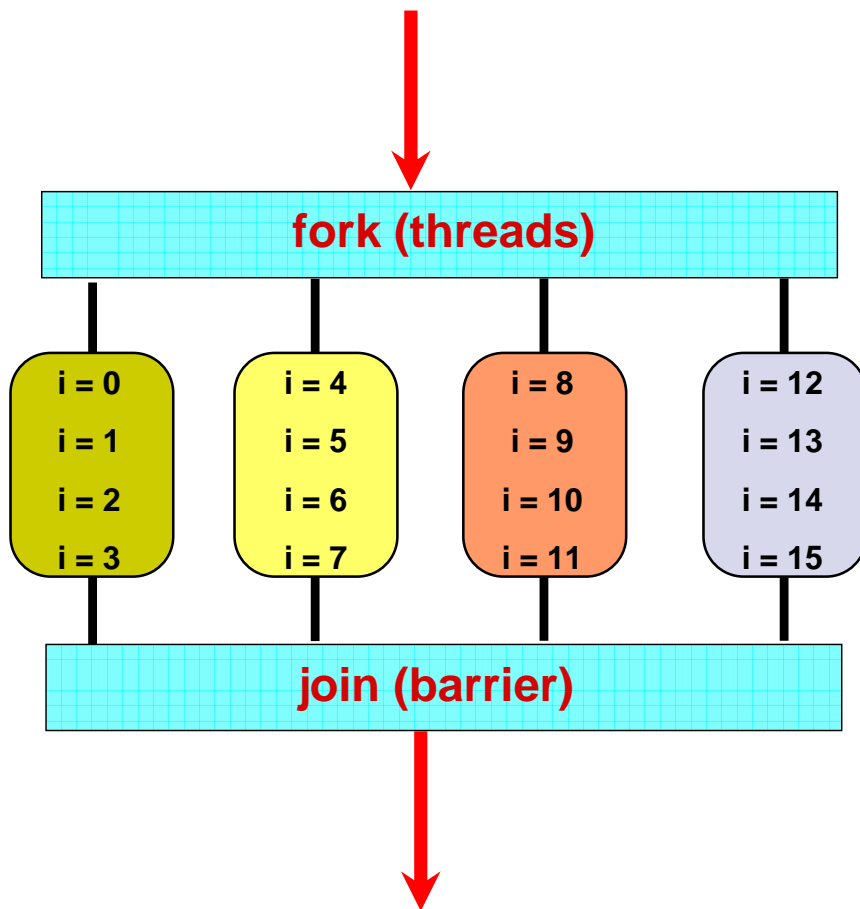
Locality of Memory Accesses (Shared Memory)

```
for (i = 0; i < 16; i++)  
    C[i] = A[i] + ...;
```



Locality of Memory Accesses (Shared Memory)

```
for (i = 0; i < 16; i++)  
    C[i] = A[i] + ...;
```



- Distribute data to relieve contention and increase effective bandwidth

Memory Access Latency in Shared Memory Architectures

- Uniform Memory Access (UMA)
 - Centrally located memory
 - All processors are equidistant (access times)
- Non-Uniform Access (NUMA)
 - Physically partitioned but accessible by all
 - Processors have the same address space
 - Placement of data affects performance

Summary of Parallel Performance Factors

- Coverage or extent of parallelism in algorithm
- Granularity of data partitioning among processors
- Locality of computation and communication
- ... so how do I parallelize my program?