CS 541

Parallel Programming Concepts

What is concurrency?

- What is a sequential program?
  - A single thread of control that executes one instruction and when it is finished execute the next logical instruction
- What is a concurrent program?
  - A collection of autonomous sequential threads, executing (logically) in parallel
- The implementation (i.e. execution) of a collection of threads can be:

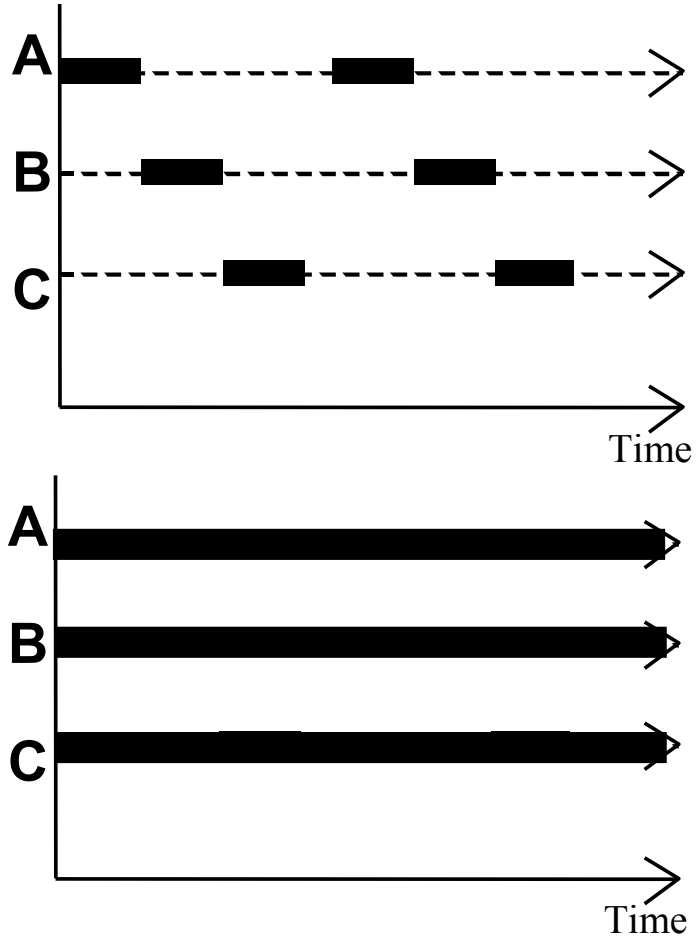
  Multiprogramming
  - Threads multiplex their executions on a single processor.

  Multiprocessing
  - Threads multiplex their executions on a multiprocessor or a multicore system

  Distributed Processing
  - Processes multiplex their executions on several different machines

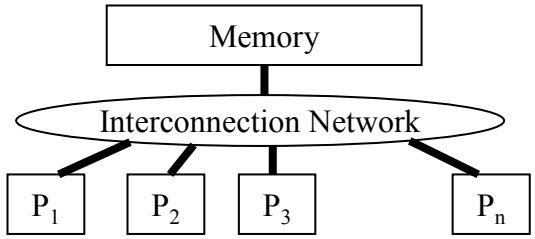
Concurrency and Parallelism

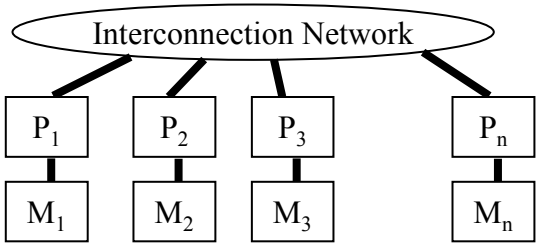
- Concurrency is not (only) parallelism
- Interleaved Concurrency
  - Logically simultaneous processing
  - Interleaved execution on a single processor
- Parallelism
  - Physically simultaneous processing
  - Requires a multiprocessor or a multicore system



Recap

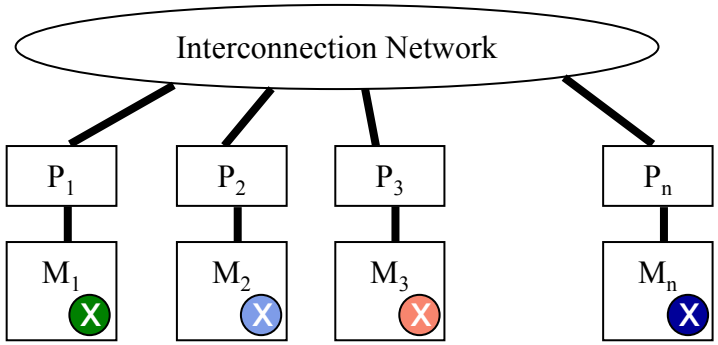
- Two primary patterns of multicore architecture design
  - Shared memory
    - Ex: Intel Core 2 Duo/Quad
    - One copy of data shared among many cores
    - Atomicity, locking and synchronization essential for correctness
    - Many scalability issues
  - Distributed memory
    - Ex: Cell
    - Cores primarily access local memory
    - Explicit data exchange between cores
    - Data distribution and communication orchestration is essential for performance





Programming Distributed Memory Processors

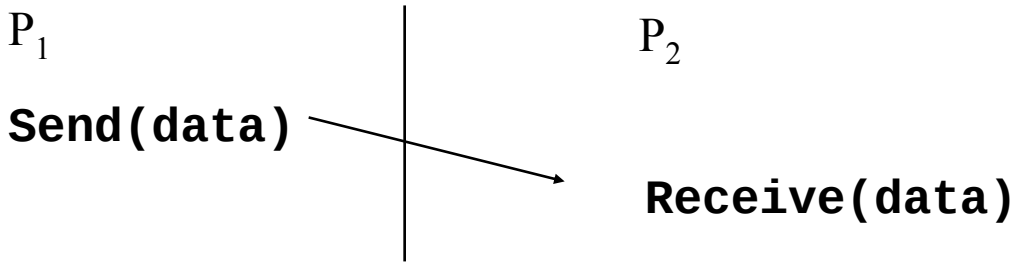
- Processors 1…n ask for X
- There are n places to look
  - Each processor's memory has its own X
  - Xs may vary



- For Processor 1 to look at Processors 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

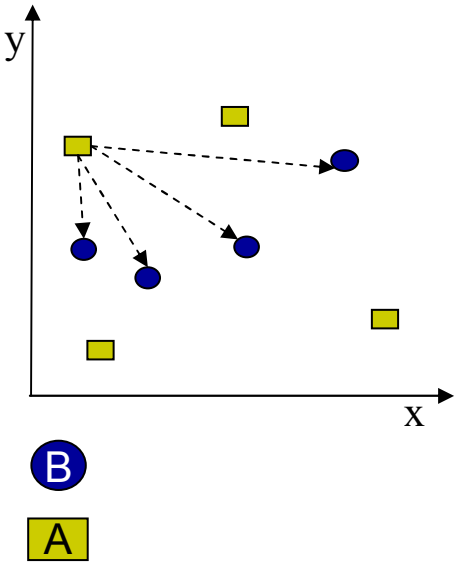
- Architectures with distributed memories use explicit communication to exchange data
  - Data exchange requires synchronization (cooperation) between senders and receivers



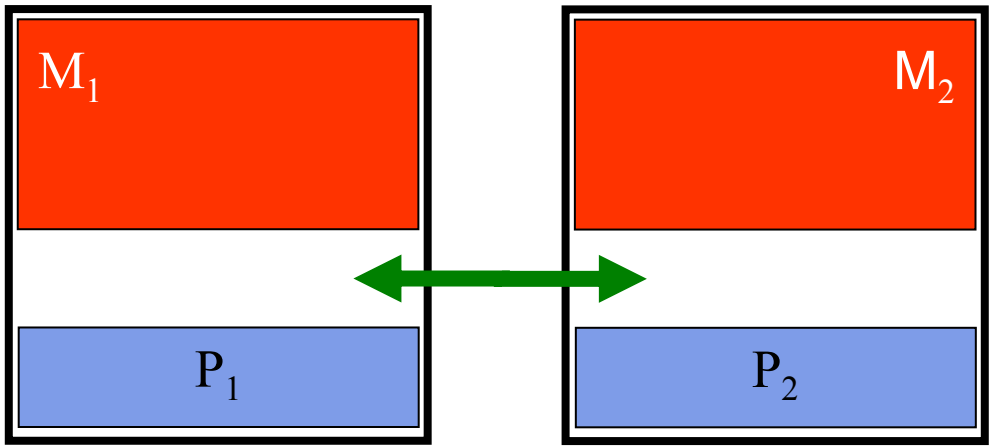
  - How is “data” described
  - How are processes identified
  - Will receiver recognize or screen messages
  - What does it mean for a send or receive to complete

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]**

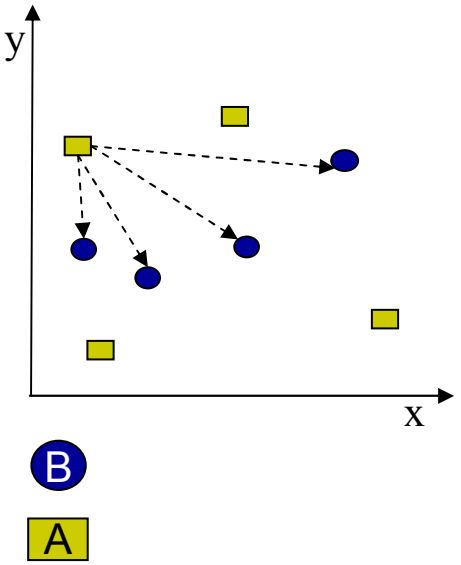



```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

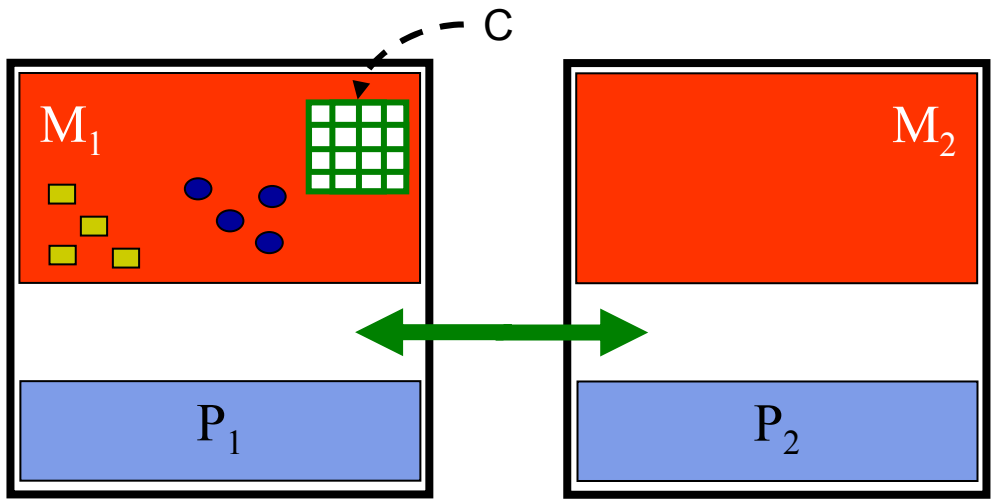


Example Message Passing Program

- 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])
```

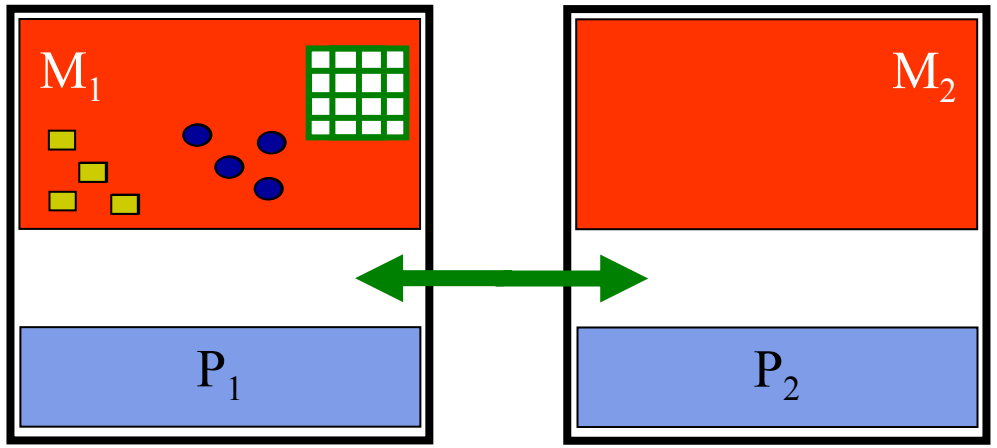


Example Message Passing Program

- 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

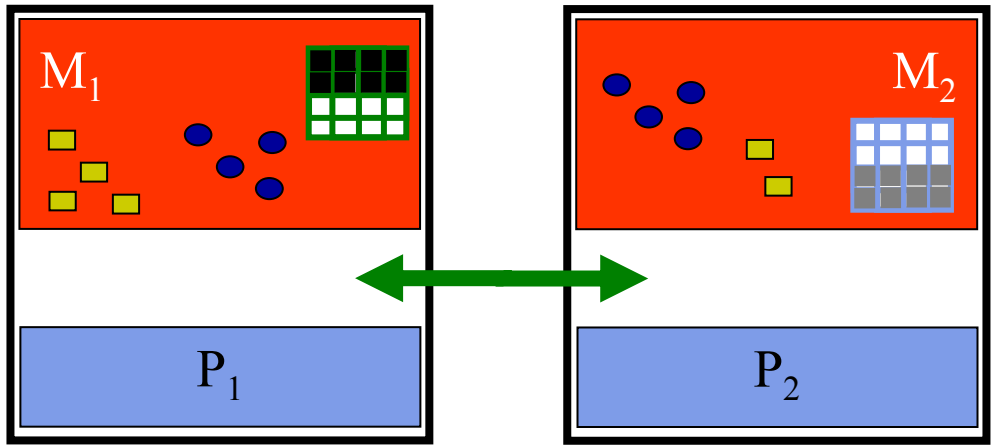


Example Message Passing Program

- 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]**

```
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
  - P_1 and P_2 compute

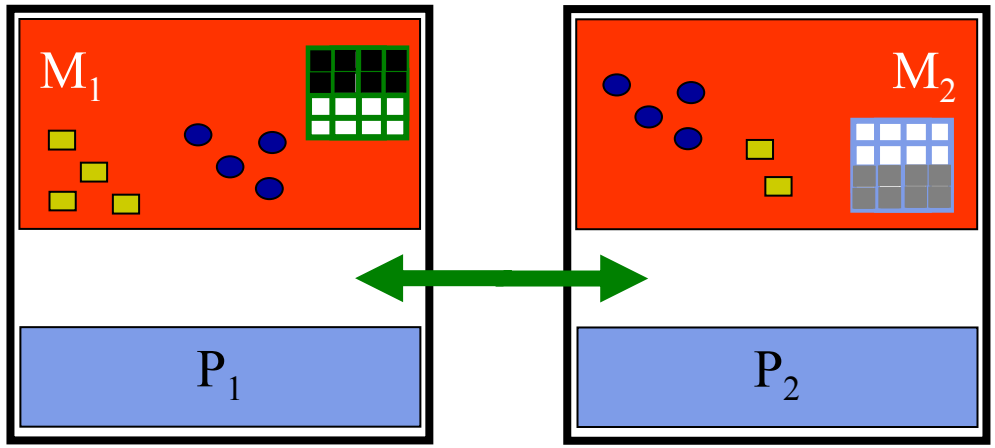


Example Message Passing Program

- 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
  - P_1 and P_2 compute
  - P_2 sends output to P_1



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

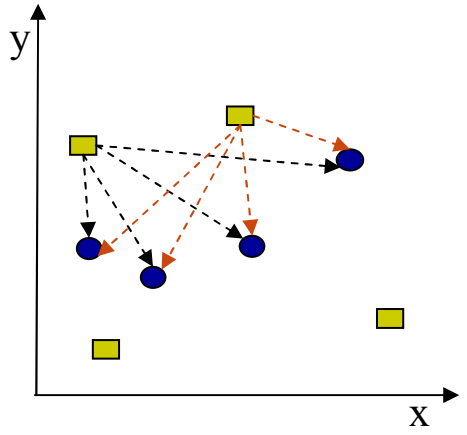
```
A[n] = {…}
B[n] = {…}

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 → 2x speedup
  - Dividing the work between four processors → 4x speedup

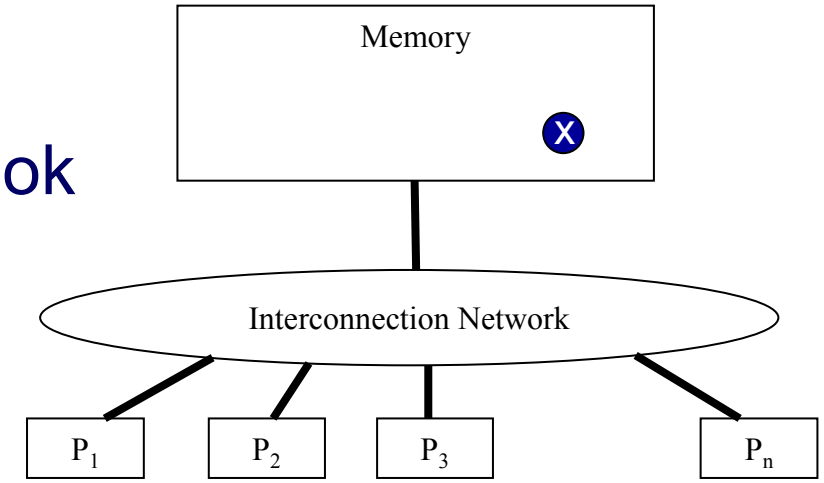


- Communication
  - 1 copy of **`B[]`** sent to each processor
  - 1 copy of **subset** of **`A[]`** to each processor

- Granularity of **`A[]`** subsets directly impact communication costs
  - Communication is not free

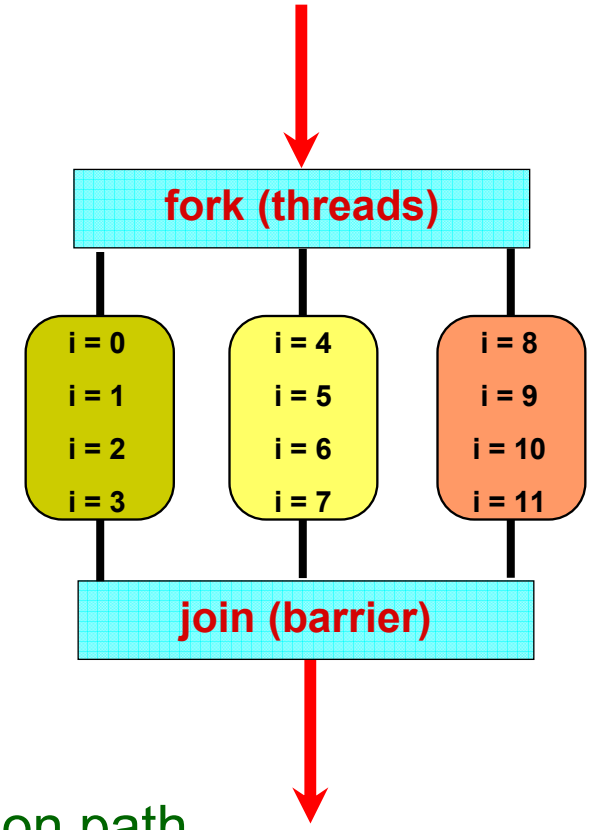
Programming Shared Memory Processors

- 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];
```

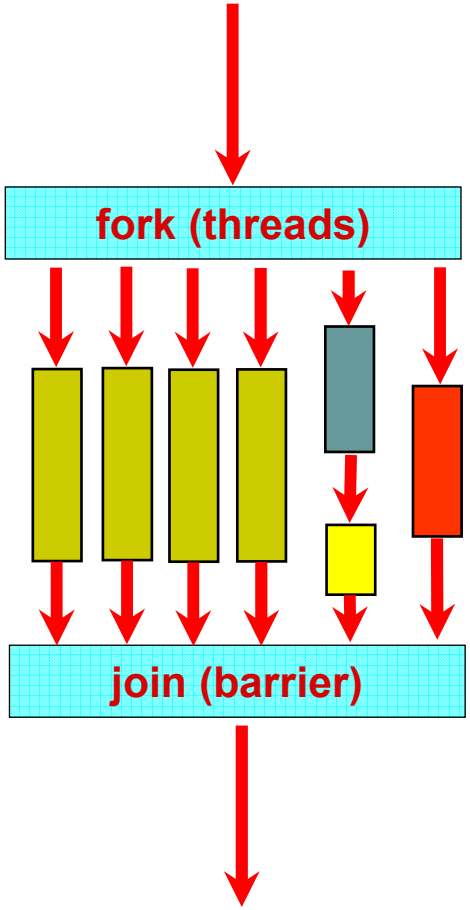


- Data parallel
  - Perform same computation but operate on different data
- A single process can fork multiple concurrent threads
  - Each thread encapsulates its own execution path
  - Each thread has local state and shared resources
  - Threads communicate through shared resources such as global memory

Types of Parallelism

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

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

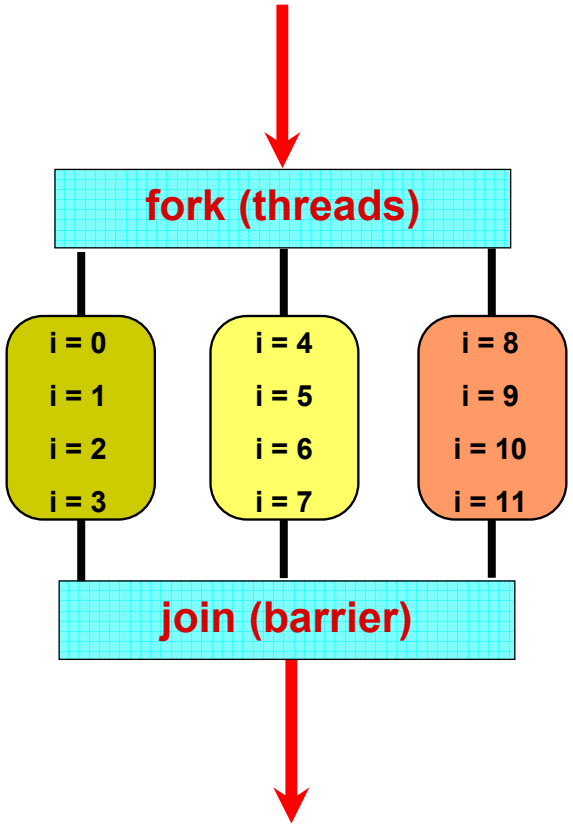


Parallel Programming with OpenMP

- Start with a parallelizable algorithm
  - SPMD model (same program, multiple data)
- Annotate the code with parallelization and synchronization directives (pragmas)
  - Assumes programmers know 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

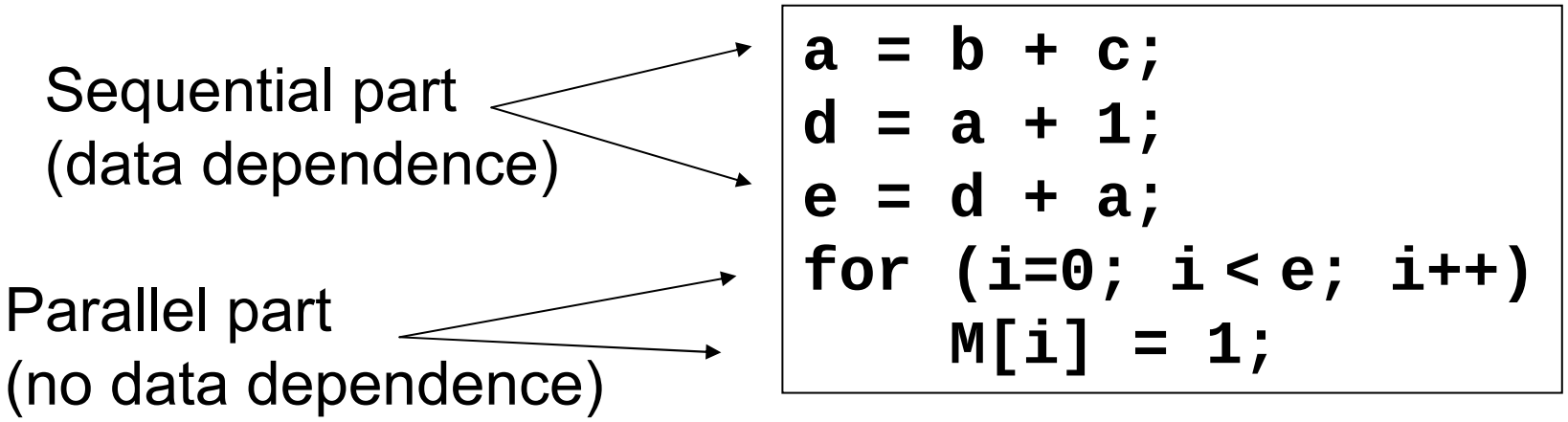
  loop is parallel, can divide work (work-sharing)

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

Limits to Performance Scalability

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



Coverage

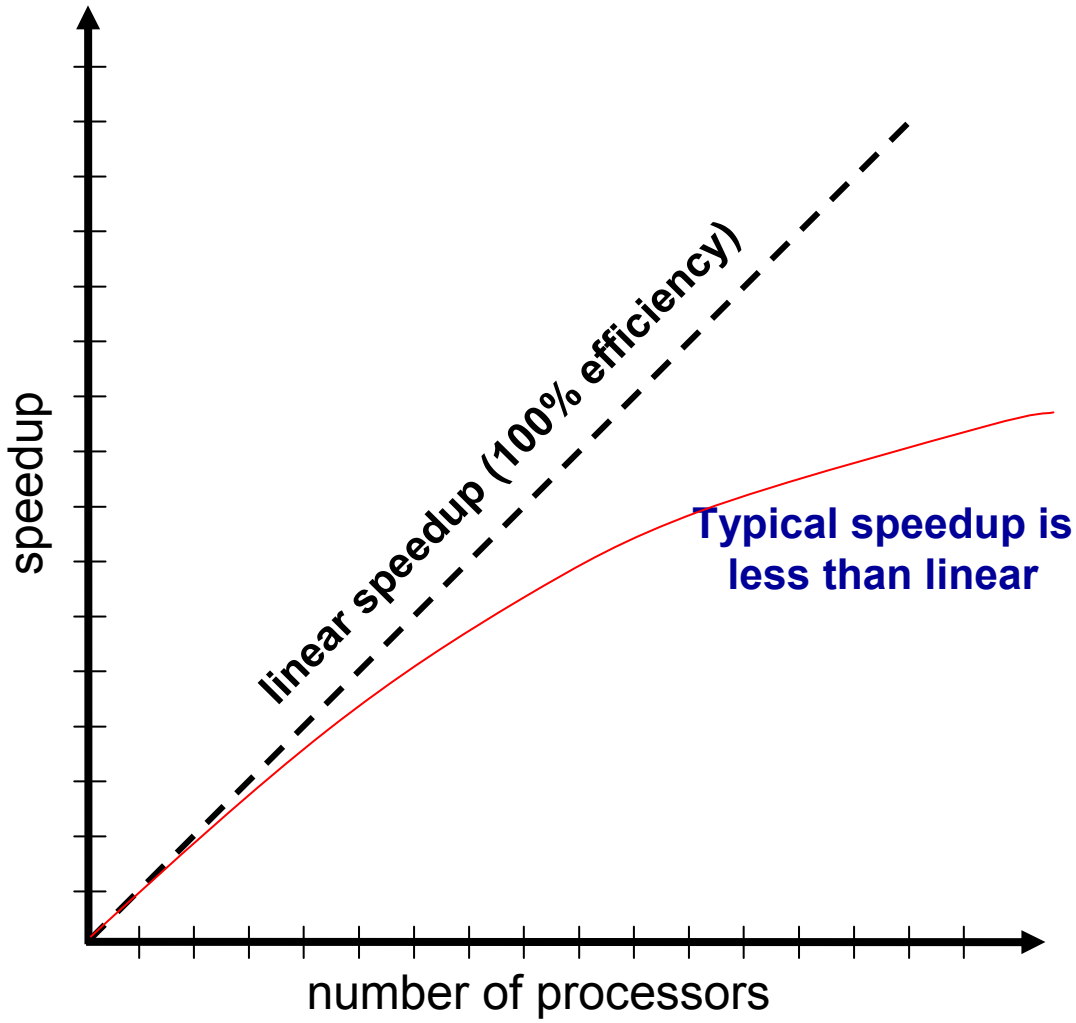
- **Amdahl's Law**: *The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.*
  - Demonstration of the law of diminishing returns

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

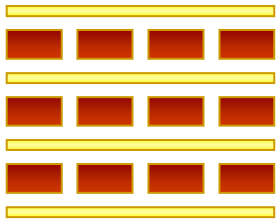
Granularity

- Granularity is a qualitative measure of the ratio of computation to communication

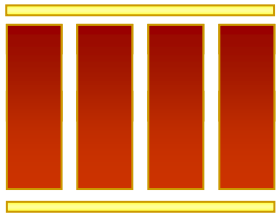
- Computation stages are typically separated from periods of communication by synchronization events

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



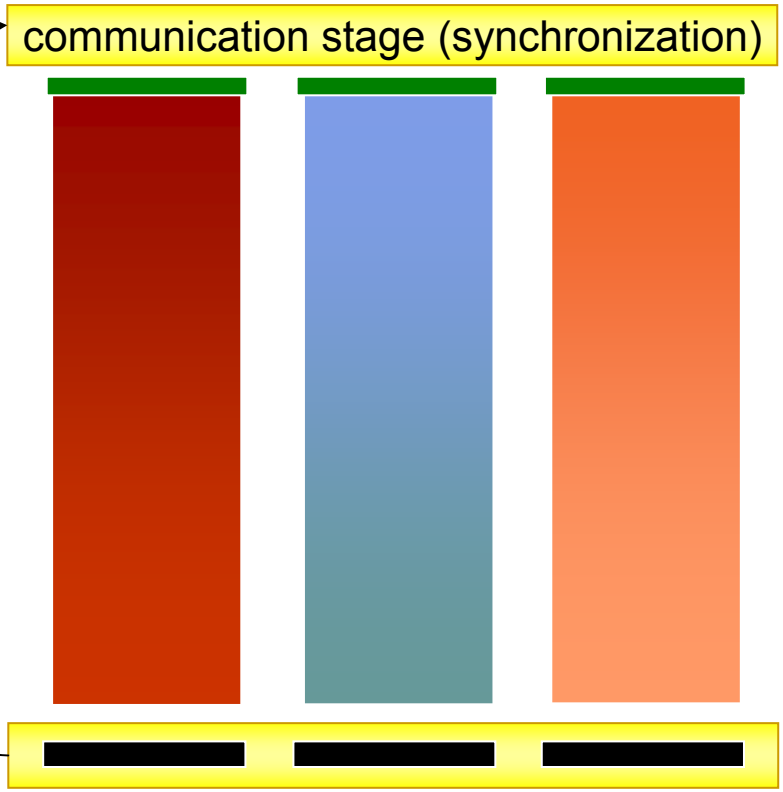
- Coarse-grain Parallelism
  - High computation to communication ratio
  - Large amounts of computational work between communication events
  - More opportunity for performance increase
  - Harder to load balance efficiently



The Load Balancing Problem

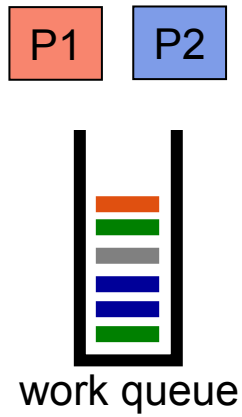
- Processors that finish early have to wait for the processor with the largest amount of work to complete
  - Leads to idle time, lowers utilization

```
// 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]);
}
```



Static Load Balancing

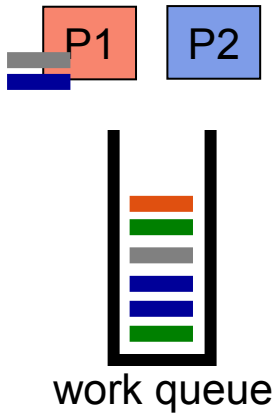
- Programmer make decisions and assigns a fixed amount of work to each processing core a priori

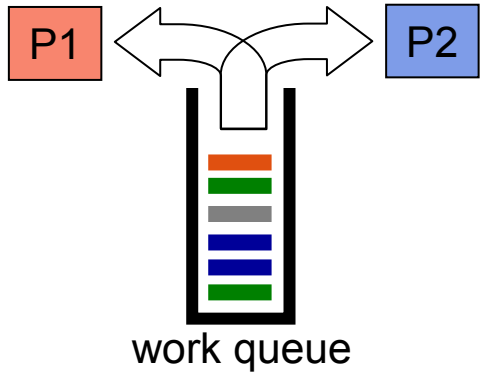


- Works well for homogeneous multicores
  - All core are the same
  - Each core has an equal amount of work
- Not so well for heterogeneous multicores
  - Some cores may be faster than others
  - Work distribution is uneven

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



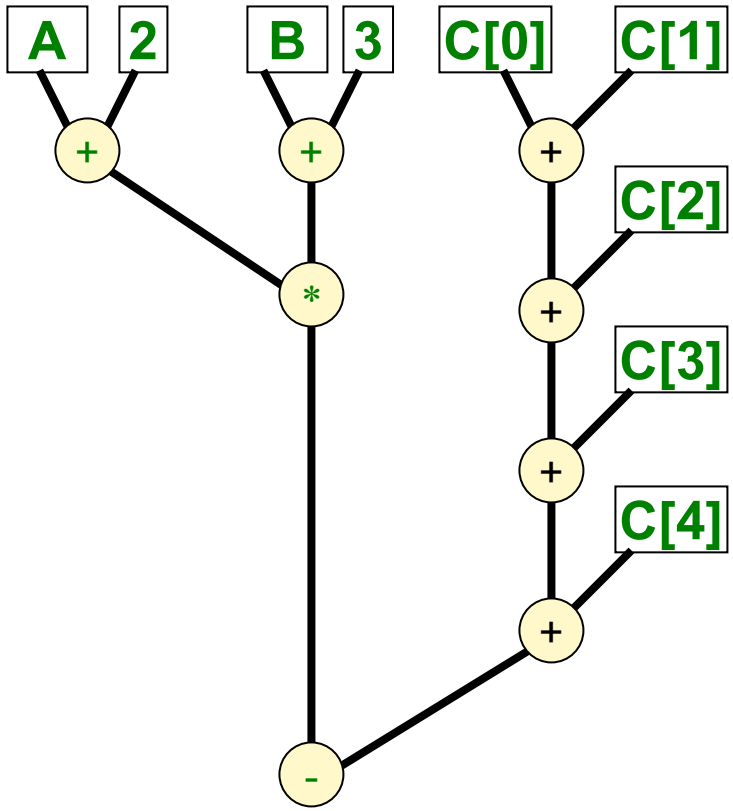


Granularity and Performance Tradeoffs

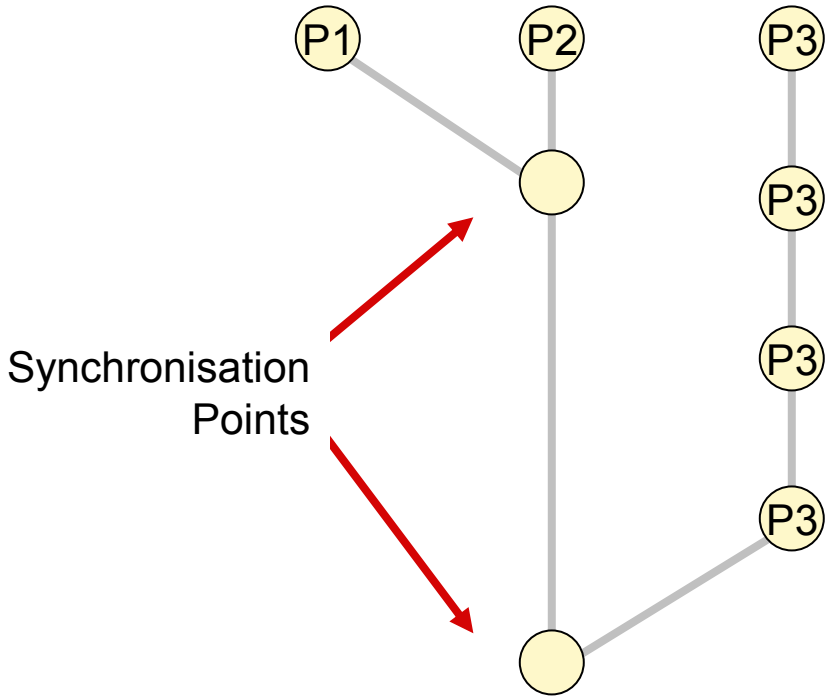
1. Load balancing
   - How well is work distributed among cores?

2. Synchronization
   - Are there ordering constraints on execution?

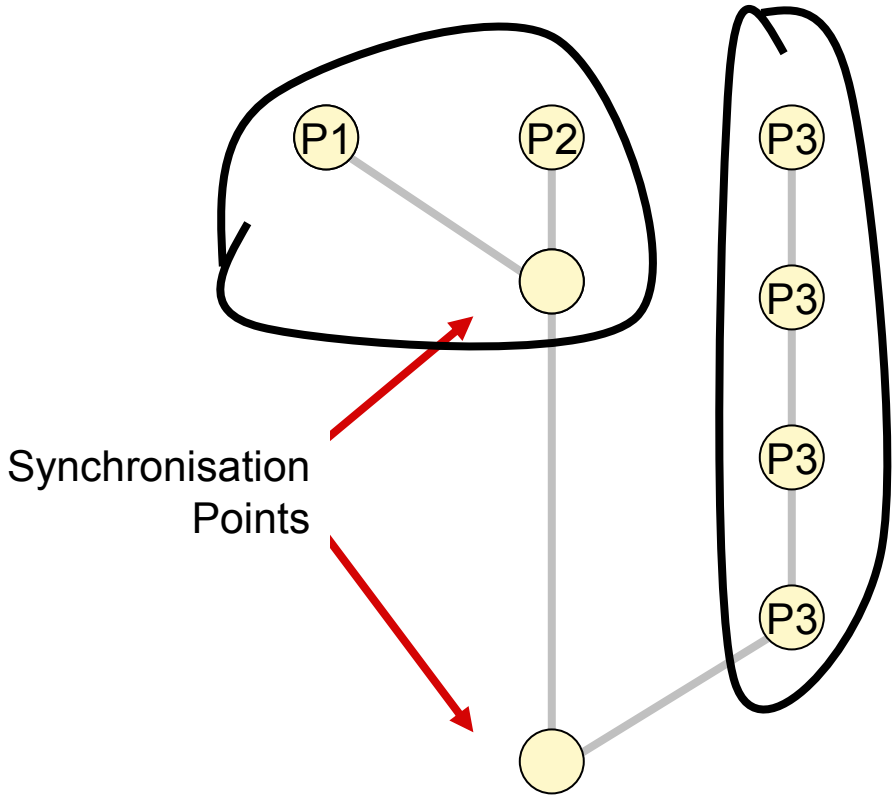
Data Dependence Graph



Dependence and Synchronization



Synchronization Removal



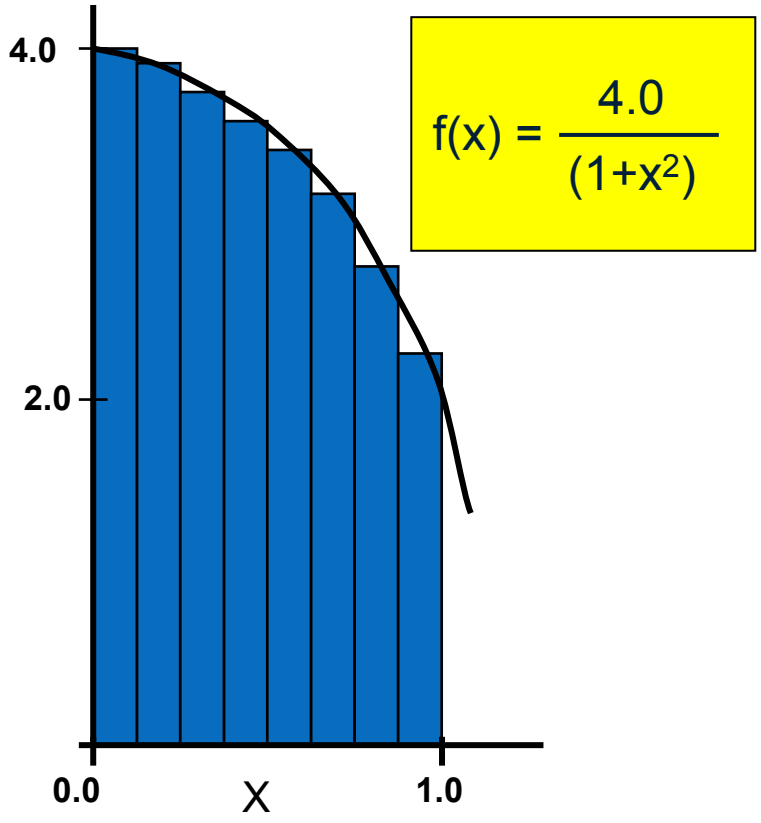
Granularity and Performance Tradeoffs

1. Load balancing
   - How well is work distributed among cores?
2. Synchronization
   - Are there ordering constraints on execution?
3. Communication
   - Communication is not cheap!

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

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 (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();
}
```

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);
}
```

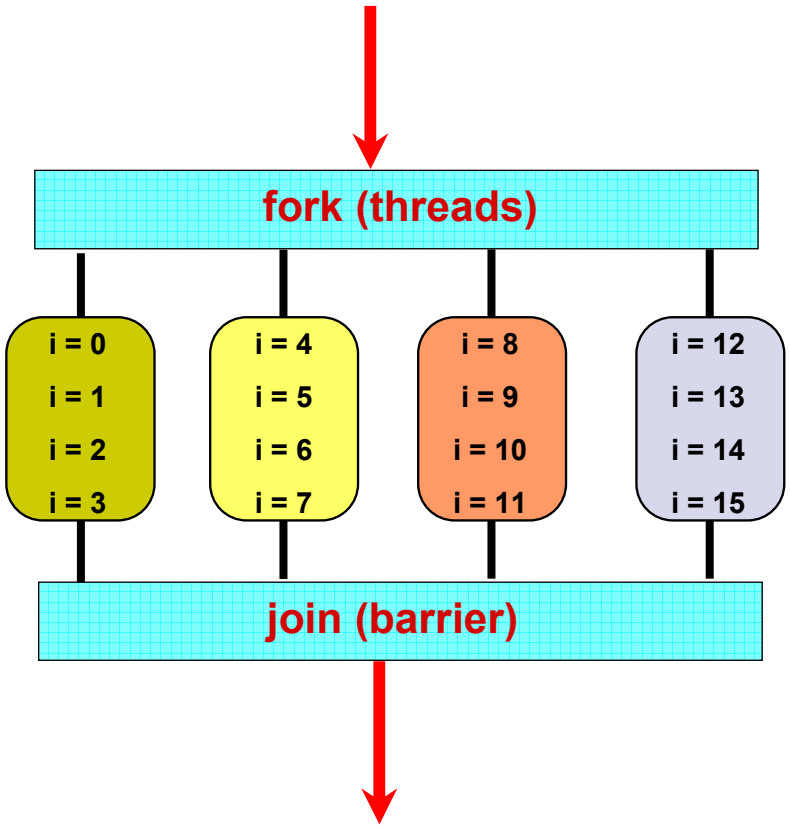
- Which variables are shared?
  - `step`
- Which variables are private?
  - `x`
- Which variables does reduction apply to?
  - `sum`

Understanding Performance

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

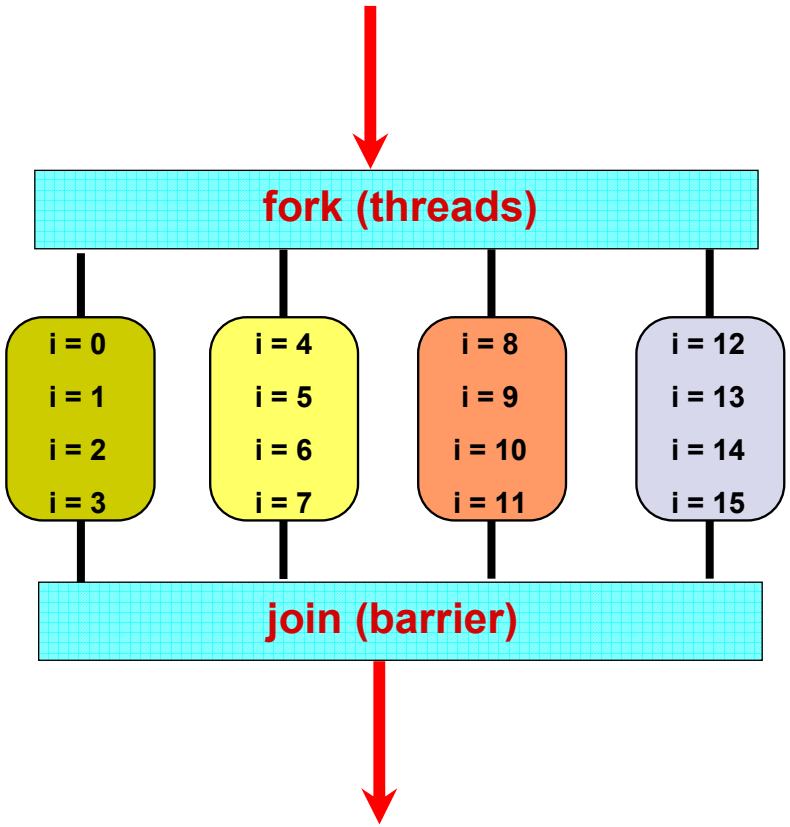
Locality of Memory Accesses (Shared Memory)

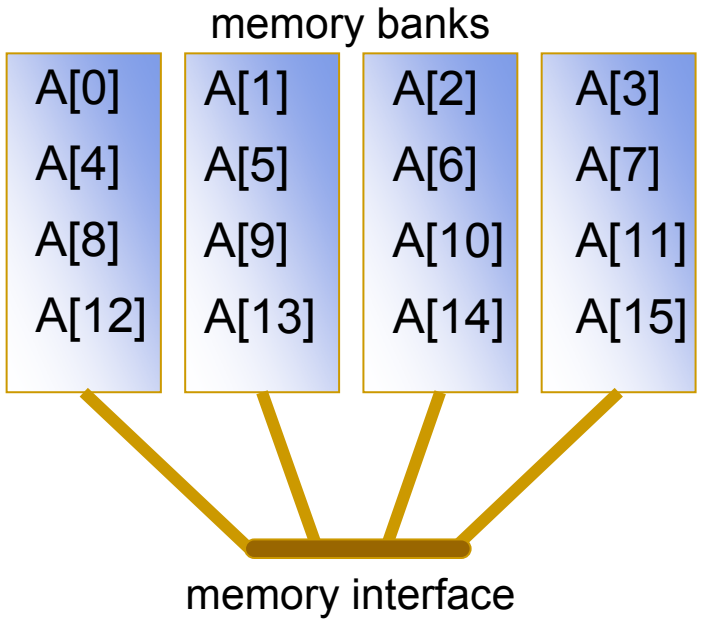
```
for (i = 0; i < 16; i++)
      C[i] = A[i] + ...;
```



Locality of Memory Accesses (Shared Memory)

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





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?