OpenMP Continued!

Thanks to XSEDE and the APC course consortium

Up Next…

The Schedule clause

Multiple for directives

The omp section clause

Synchronization constructs

False sharing

Scheduling work to threads

- The `schedule` clause – ways to assign iterations to threads

  `schedule(scheduling_class[, parameter])`

- Scheduling classes
  - `static`
  - `dynamic`
  - `guided`
  - `runtime`

- Recall models studied in the early lectures!

Scheduling classes

- Example: matrix-matrix multiplication (assume n x n square)

```
for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        c[i][j] = 0;
        for(k = 0; k < n; k++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Static scheduling

- Split iterations into equal chunks of size S, assign to threads round-robin
  - If no chunk size specified, divide into as many chunks as threads

```
#pragma omp parallel default(none)
        shared(a, b, c, n)
        num_threads(8)
{
  #pragma omp for schedule(static)
  for(int i = 0; i < n; i++) {
     for(int j = 0; j < n; j++) {
       c[i][j] = 0;
       for(int k = 0; k < n; k++) {
          c[i][j] += a[i][k] * b[k][j];
       }
     }
  }
}
```

- Outer loop is split into 8 chunks
  - e.g., for n = 1024, chunk = 128 rows => same as `schedule(static, 128)`

Dynamic scheduling

- Load per iteration may not be balanced => equally partitioned tasks may take different execution time
- Split iterations into chunks, assign new chunk to thread only when idle
  - If no chunksize specified, default chunk is single iteration

```
#pragma omp parallel default(none)
        shared(a, b, c, n)
        num_threads(8)
{
  #pragma omp for schedule(dynamic)
  for(int i = 0; i < n; i++) {
     for(int j = 0; j < n; j++) {
       c[i][j] = 0;
       for(int k = 0; k < n; k++) {
          c[i][j] += a[i][k] * b[k][j];
       }
     }
  }
}
```

Dynamic scheduling: Load Imbalance

- Imagine 1000 iterations and chunk size = 50 => 20 chunks
  - 16 threads => ?

Guided scheduling

- Imagine 1000 iterations and chunk size = 50 => 20 chunks
  - 16 threads => 12 threads get 1 chunk each, 4 threads get 2 each
  - Load imbalance: 12 threads are idle for possibly a long time!
- Guided scheduling idea: start with big chunks but reduce size as computation progresses
  - Chunks get smaller and smaller as computation progresses, if load gets imbalanced

    `schedule(guided[, chunksize])`
  - Parameter `chunksize` specifies the minimum size chunk to use
  - If not specified, default `chunksize` is 1

Runtime scheduling

- Delay scheduling decision until runtime

- Environment variable OMP_SCHEDULE determines class and chunksize
- If no schedule type is identified at runtime, the runtime system will choose the most appropriate schedule (also called AUTO scheduling)

Restrictions on `for` directive

```
#pragma omp for
for (int i=0; i < n; i++) {
```

Must be integer type!

Must be an integer assignment

Must be a <, >, <=, or >= expression

Must have integer increments only

```
  // statements
```

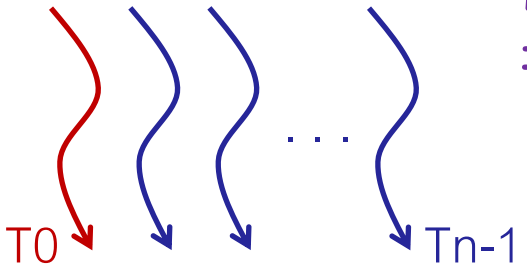
No `break` instructions!

```
}
```

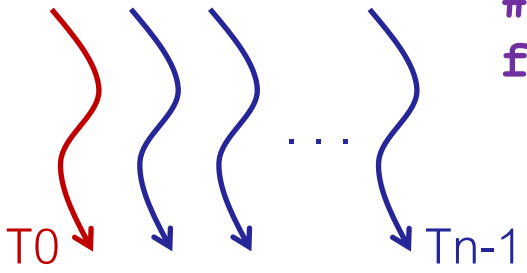
- Compiler will typically prompt you to the problem
- Refer to the OpenMP manual when it doubt!

Sequences of `for` directives

- A sequence of `for` directives: implicitly adds barrier after each one



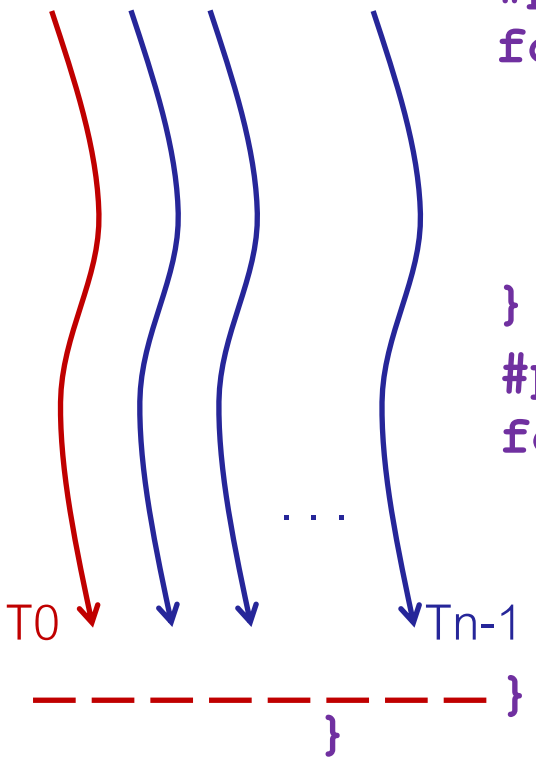

```
#pragma omp parallel {
      // Spawn threads

      #pragma omp for
      for(...) {

        // Parallel execution of loop
      }
      #pragma omp for
      for(...) {

        // Parallel execution of loop
      }
```



`nowait` clause

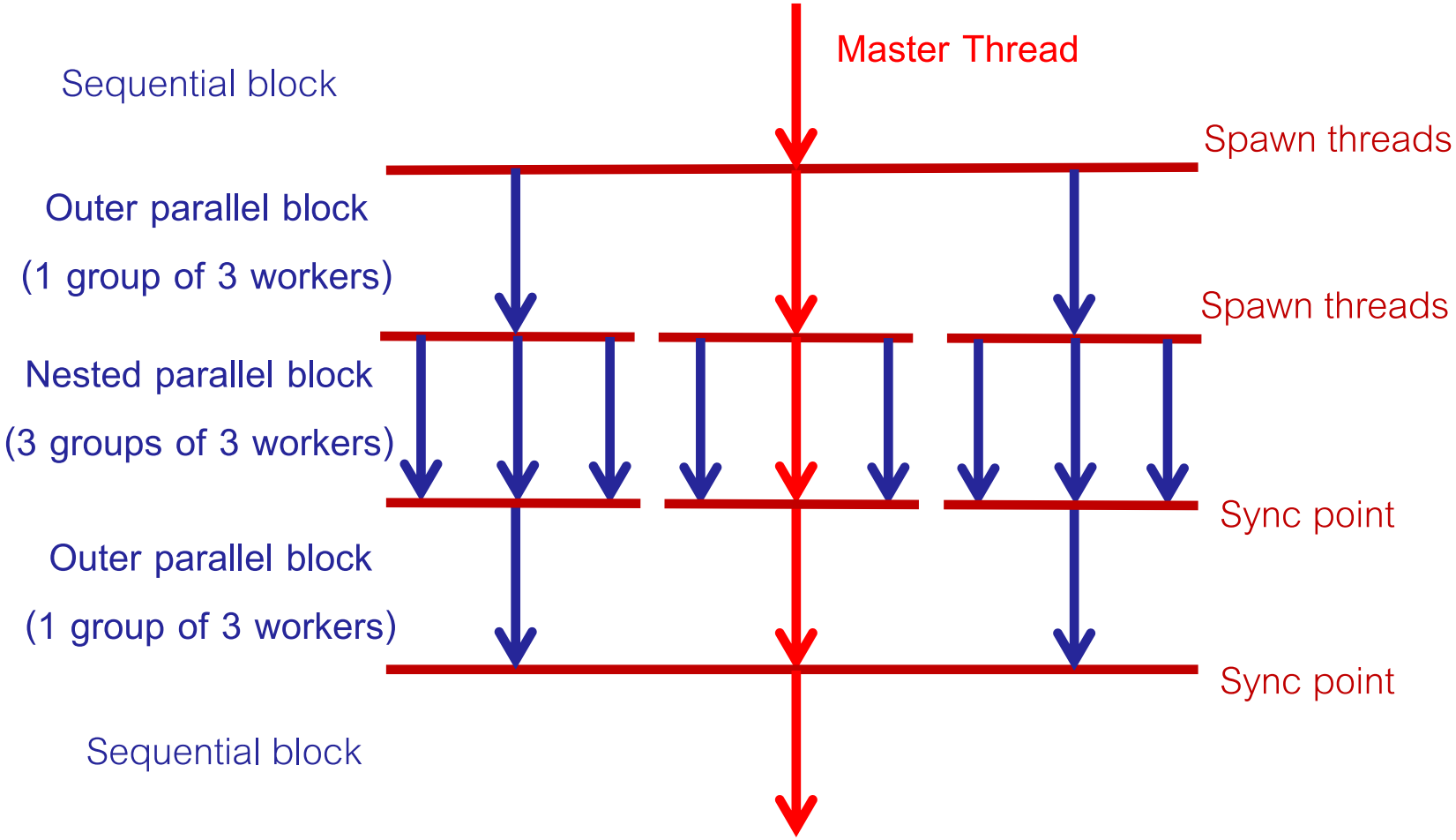
- Let threads proceed without an implicit barrier
  - Might be useful if there is no need to wait for results from previous loop

```
#pragma omp parallel
{       // Spawn threads

  #pragma omp for nowait
  for(...) {

    // Parallel execution of loop

  }
  #pragma omp for
  for(...) {

    // Parallel execution of loop
}
}
```



Nested parallelism

- OpenMP supports arbitrarily deep nesting of parallel regions



Sections

- So far, threads do the same work: same block, or the same loop code
- What about diverging tasks?

```
#pragma omp parallel {
   #pragma omp sections [clauses] {
      // Specify different tasks as "section" blocks
      #pragma omp section {
         // task 1
      }
      #pragma omp section {
         // task 2
      }
      ...
      #pragma omp section {
         // task n
      }
   }
}
```

Tasks scheduled in parallel

- 1 or more threads assigned to each section

- each thread gets 1 or more sections

Sections

- So far, threads do the same work: same block, or the same loop code
- What about diverging tasks?

```
#pragma omp parallel
{
    #pragma omp sections [clauses]
    {
        // Specify different tasks as "section" blocks
        #pragma omp section
        { // task 1
        }
        #pragma omp section
        { // task 2
        }
        ...
        #pragma omp section
        { // task n
        }
     }
}
```

Clauses:

- private, firstprivate, lastprivate
- reduction
- nowait

The omp task clause

`#pragma omp task/s` is another directive that is very similar to the section directive.

An implicit barrier exists at the end of the omp sections clause.

However, omp tasks does not have this implicit barrier, hence the existence of the taskwait clause. See the omp manual for more!

Example: Fibonacci numbers

```
int fib (int n)
{
   int x,y;
   if (n < 2) return n;

   x = fib(n-1);
   y = fib (n-2);
   return (x+y);
}

int main()
{
   int NW = 5000;
   fib(NW);
}
```

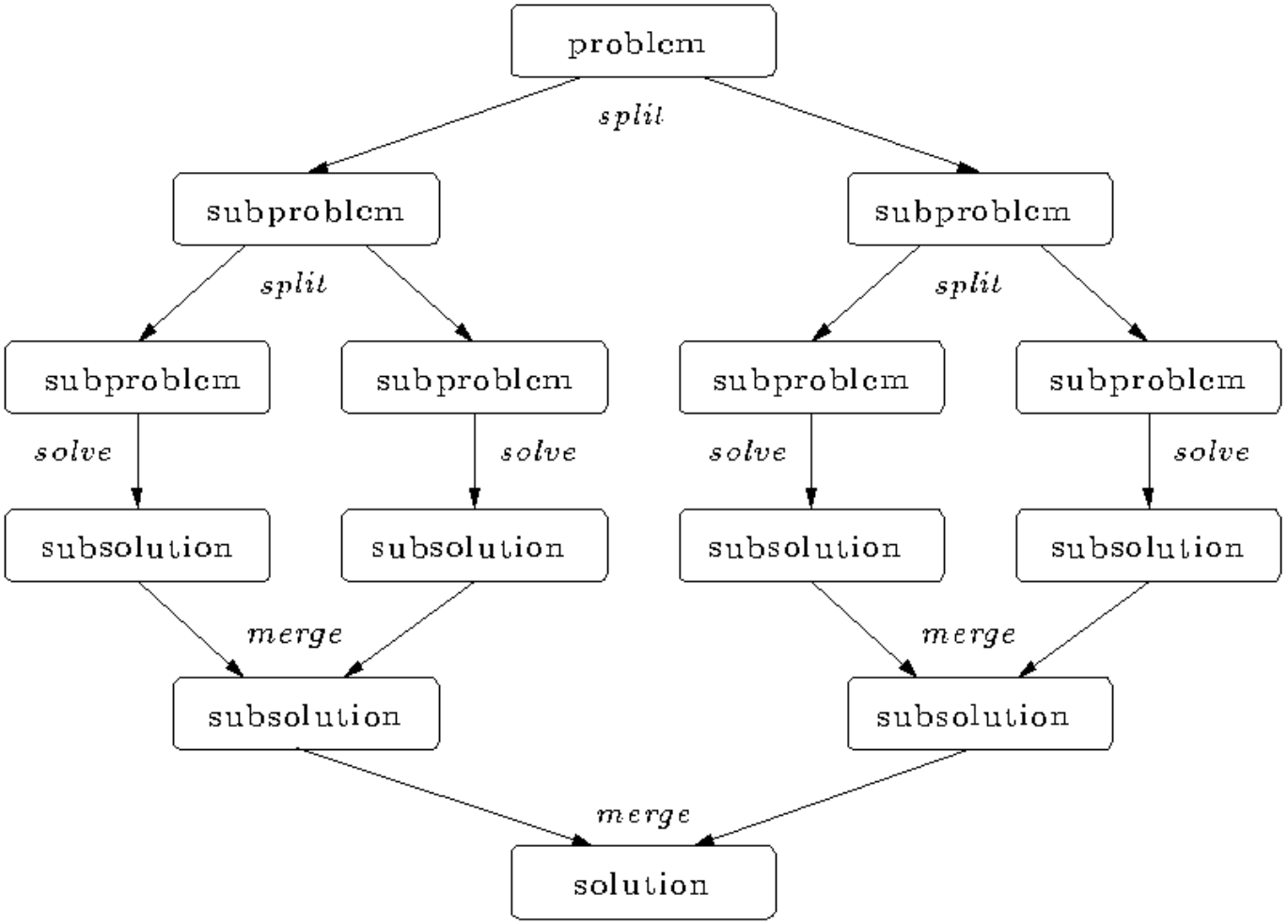
- $F_n = F_{n-1} + F_{n-2}$
- Inefficient $O(n^2)$ recursive implementation!

Parallel Fibonacci

```
int fib (int n)
{   int x,y;
   if (n < 2) return n;

#pragma omp task shared(x)
   x = fib(n-1);
#pragma omp task shared(y)
   y = fib (n-2);
#pragma omp taskwait
   return (x+y);
}

Int main()
{  int NW = 5000;
   #pragma omp parallel
   {
      #pragma omp single
         fib(NW);
   }
}
```

- taskwait enforces that a task not complete until all tests below it in the tree are finished
- **`x,y`** are local to the thread/s that executes a task however they must be shared on child tasks!
- Although one thread executes the single directive, all threads will participate in executing the tasks.

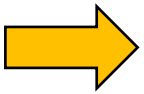
Parallel Fibonacci



Merging directives

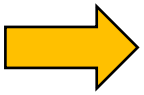
- `parallel` needed to spawn threads => merge it with `for` or `sections`

```
#pragma omp parallel shared(n)
{
  #pragma omp for
  for(int i = 0; i < n; i++) {
     // parallel execution
  }
}
```



```
#pragma omp parallel for shared(n)
{
  for(int i = 0; i < n; i++) {
     // parallel execution
  }
}
```

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    { // task 1
    }
    #pragma omp section
    { // task 2
    } ...
    #pragma omp section
    {  // task n
    }
  }
}
```



```
#pragma omp parallel sections
{
   #pragma omp section
   { // task 1
   }
   #pragma omp section
   {   // task 2
   } ...
   #pragma omp section
   {   // task n
   }
}
```

Synchronization constructs

- Directives
  - barrier
  - single, master
  - critical
  - atomic
  - ordered
  - flush
- Explicit locks

Barriers

- Wait for all threads spawned by closest enclosing `parallel` directive
  - Either all or none of the threads must hit the barrier point
- Example:

```
#pragma omp parallel num_threads(4)
{
   foo();

   // All threads stop here and wait for each other
   #pragma omp barrier

   // All threads have executed foo(), before any calls bar()
   bar();
}
```

- Remember: `for` and `sections` use implicit barriers at the end, unless the `nowait` clause is present

Single directive

- When a task in a parallel block must be executed by only one thread
  - first thread to encounter the single directive is the only one executing it
- Calculate the class average and report which students are above it
  - reduction on the sum, then only one thread computes the average

```
float sum = 0.0, average;
#pragma omp parallel reduction(+:sum)
        shared(average, num_students, names, grades)
        num_threads(8)
{
  #pragma omp for
  for(i = 0; i < num_students; i++) {
    sum += grades[i];
  }
  #pragma omp single
  { average = sum / num_students;
  }
  #pragma omp for
  for(i = 0; i < num_students; i++) {
    if (grades[i] > average)  printf("%s is above average\n", names[i]);
  }
}
```

Clauses: private, firstprivate, nowait

Implicit barrier, unless using nowait!

Master directive

- Similar to single, only the master thread executes this block
  - Single: only one thread (any of them) can execute the single block
  - Master: only the master thread executes the master block
- Unlike single, no implicit barrier after the `master` block

Critical sections

- Critical region where threads must serialize to avoid race conditions

```
#pragma omp critical [(name)]
// critical section
```

- Simple example:

```
int data = 100;
#pragma omp parallel sections
{
  #pragma omp section
  {
    #pragma omp critical (datacrit)
    {
      data += 42;
    }
  }
  #pragma omp section
  {
    #pragma omp critical (datacrit)
    {
      data += 5;
    }
  }
}
printf("Data=%d\n", data);
```

Critical sections

- In our previous example:

```
int dotprod = 0;
#pragma omp parallel num_threads(8) \
            shared(size, a, b, dotprod)
{

   #pragma omp for
   for(int i = 0; i < size; i++) {
      #pragma omp critical
      dotprod += a[i] * b[i];
   }
}
```

- Correctness ✓
- What about performance? ✗

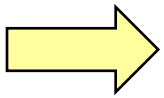
Atomic construct

- When the critical section is just an update to a single memory location
  - The following instruction after the `atomic` directive is executed .. atomically

```
#pragma omp atomic [read|write|update|capture]
var op= expr
```

- Limitations:
  - Only simple operations (e.g., +=, -=, &=, etc., or ++, --)
  - Expression is simple, does not include reference to variable var
  - Consult the OpenMP manual!
- Can be translated into a `critical` directive

```
#pragma omp atomic
data += 5;
```



```
#pragma omp critical(data)
data += 5;
```

Atomic construct

- In our previous example:

```
int dotprod = 0;
#pragma omp parallel num_threads(8) \
            shared(size, a, b, dotprod)
{

   #pragma omp for
   for(int i = 0; i < size; i++) {
      #pragma omp atomic update
      dotprod += a[i] * b[i];
   }
}
```

- Correctness ✓
- What about performance?
  - Better, much lower overhead
  - More specialized than critical directive => can use H/W support for atomic instructions
  - Still not great though…

Atomic - optimizations

- What if we calculate local (partial) dotproducts, and sum them up after for loop?

```
int dotprod = 0;
#pragma omp parallel num_threads(8) \
            shared(size, a, b, dotprod)
{
   int mydotprod = 0;

   #pragma omp for
   for(int i = 0; i < size; i++) {
      mydotprod += a[i] * b[i];
   }
   #pragma omp atomic update
   dotprod += mydotprod;
}
```

- Remember - localization
  - Only 8 atomic operations, compared to `size` atomics in the previous code
- Try reduction clause too…

Explicit locks

- In some cases, must have the ability to use explicit locks
- Self-explanatory, similar to pthread mutexes (check specs!)

```
void omp_init_lock(omp_lock_t *lock); // pthread_mutex_init
void omp_destroy_lock(omp_lock_t *lock);// pthread_mutex_destroy
void omp_set_lock(omp_lock_t *lock); // pthread_mutex_lock
void omp_unset_lock(omp_lock_t *lock); // pthread_mutex_unlock
int omp_test_lock(omp_lock_t *lock); // pthread_mutex_trylock
```

- Supports nestable locks (can be locked multiple times by same thread)
  - Lock will be unlocked only once it's been unset the same number of times

```
void omp_init_nest_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t *lock);
```

Performance profiling

- Built-in timing functions

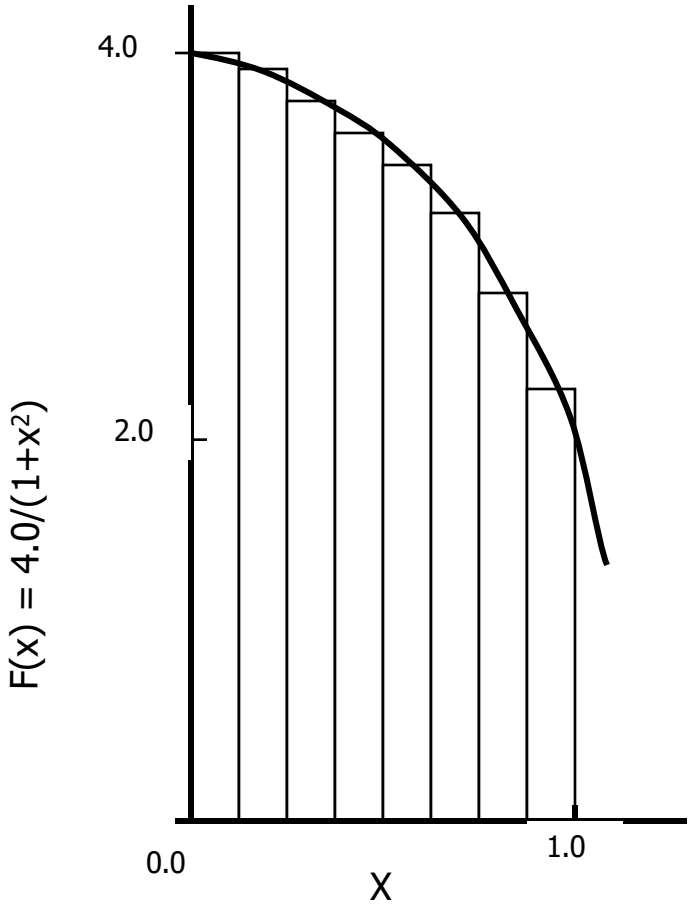
```
double omp_get_wtime(); // wall clock time in seconds
```

  - The time is measured per thread, no guarantee can be made that two distinct threads measure the same time

```
double start_time = omp_get_wtime();
#pragma omp parallel [...]
{
   // parallel block
}
double elapsed_time = omp_get_wtime() - start_time;
```

- See also `omp_get_wtick()`

Example: Numerical integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

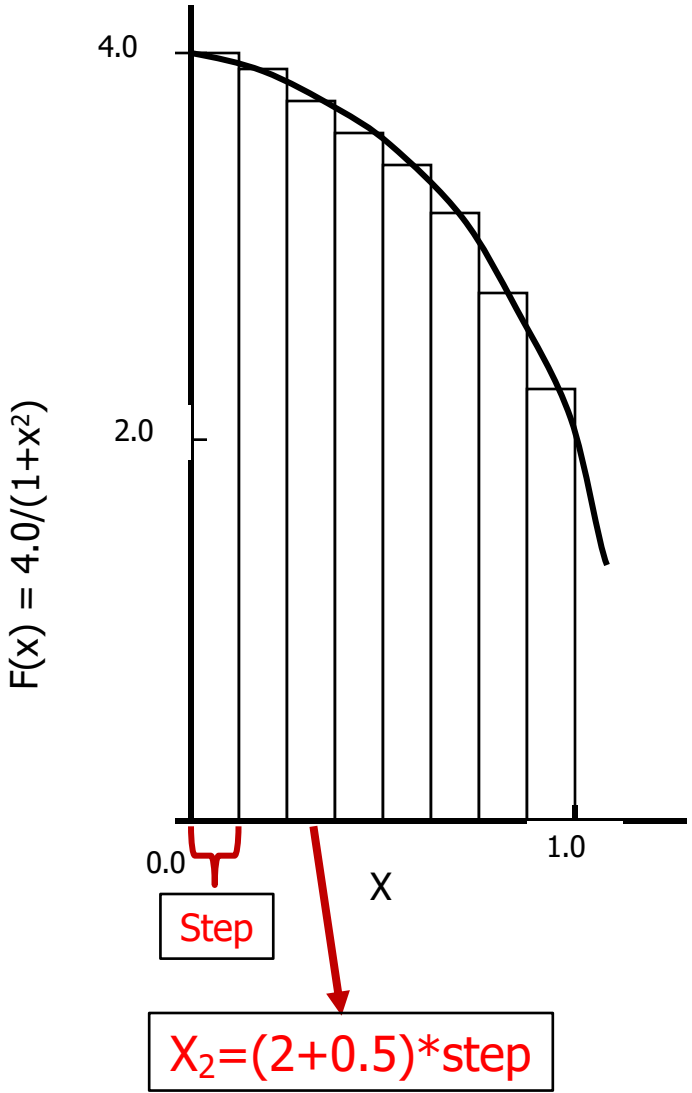
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial pi program

```
static long num_steps = 100000;
double step;
int main ()
{           int i; double x, pi, 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;
}
```

Example: Numerical integration




```
static long num_steps = 100000;
double step;
int main ()
{              ….
 step = 1.0/(double) num_steps;

  for (i=0;i< num_steps; i++){
              x = (i+0.5)*step;
              sum=sum+f(x)
              }
  ….
}
```

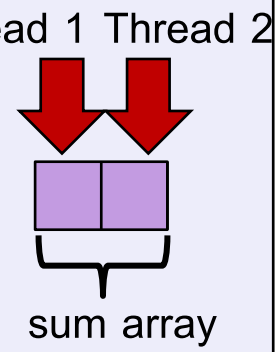
Serial pi program

```
#include <omp.h>
static long num_steps = 100000;
double step;
int main ()
{           int i;      double x, pi, sum = 0.0, tdata;

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

The library routine get_omp_wtime() is used to find the elapsed "wall time" for blocks of code

Parallel pi program

```
#include <omp.h>
static long num_steps = 100000;  double step;
#define NUM_THREADS 2
void main ()
{  int i, nthreads;  double pi, sum[NUM_THREADS]
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {        int i, id,nthrds;
             double x;
             id = omp_get_thread_num();
             nthrds = omp_get_num_threads();
             if (id == 0)   nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                          x = (i+0.5)*step;
                          sum[id] += 4.0/(1.0+x*x);
             }
     }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

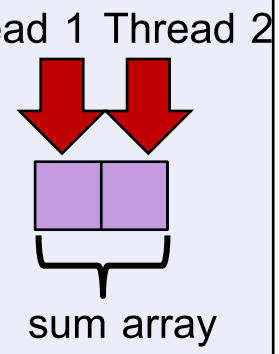


threads	1st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

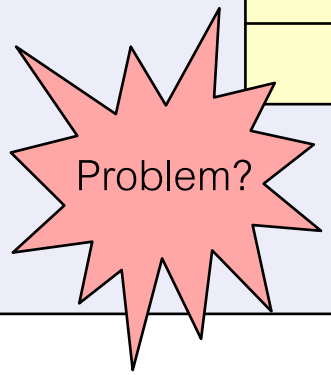
*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Parallel pi program

```
#include <omp.h>
static long num_steps = 100000;  double step;
#define NUM_THREADS 2
void main ()
{  int i, nthreads;  double pi, sum[NUM_THREADS]
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {        int i, id,nthrds;
             double x;
             id = omp_get_thread_num();
             nthrds = omp_get_num_threads();
             if (id == 0)   nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                          x = (i+0.5)*step;
                          sum[id] += 4.0/(1.0+x*x);
             }
     }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

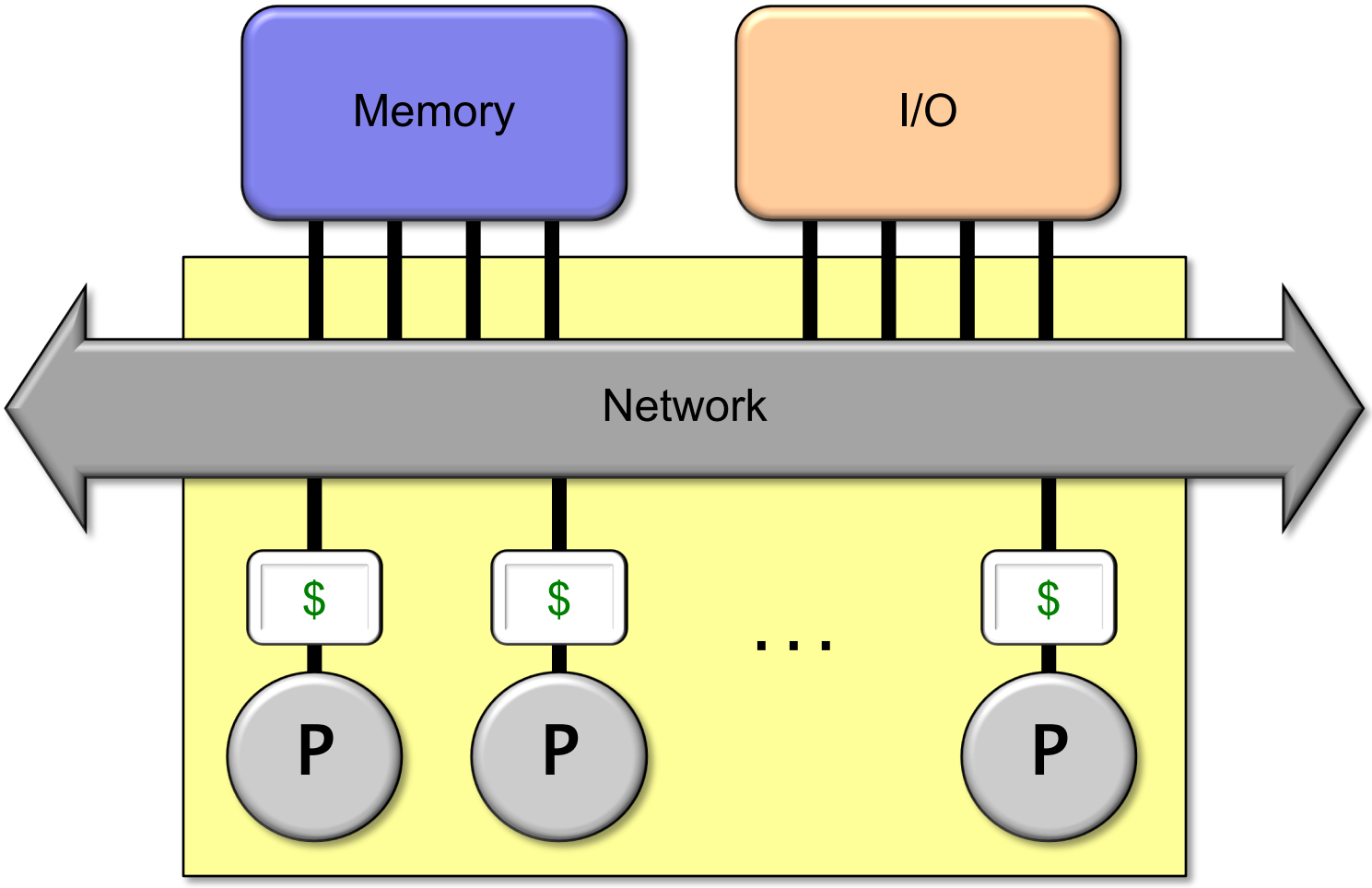


threads	1st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97



*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

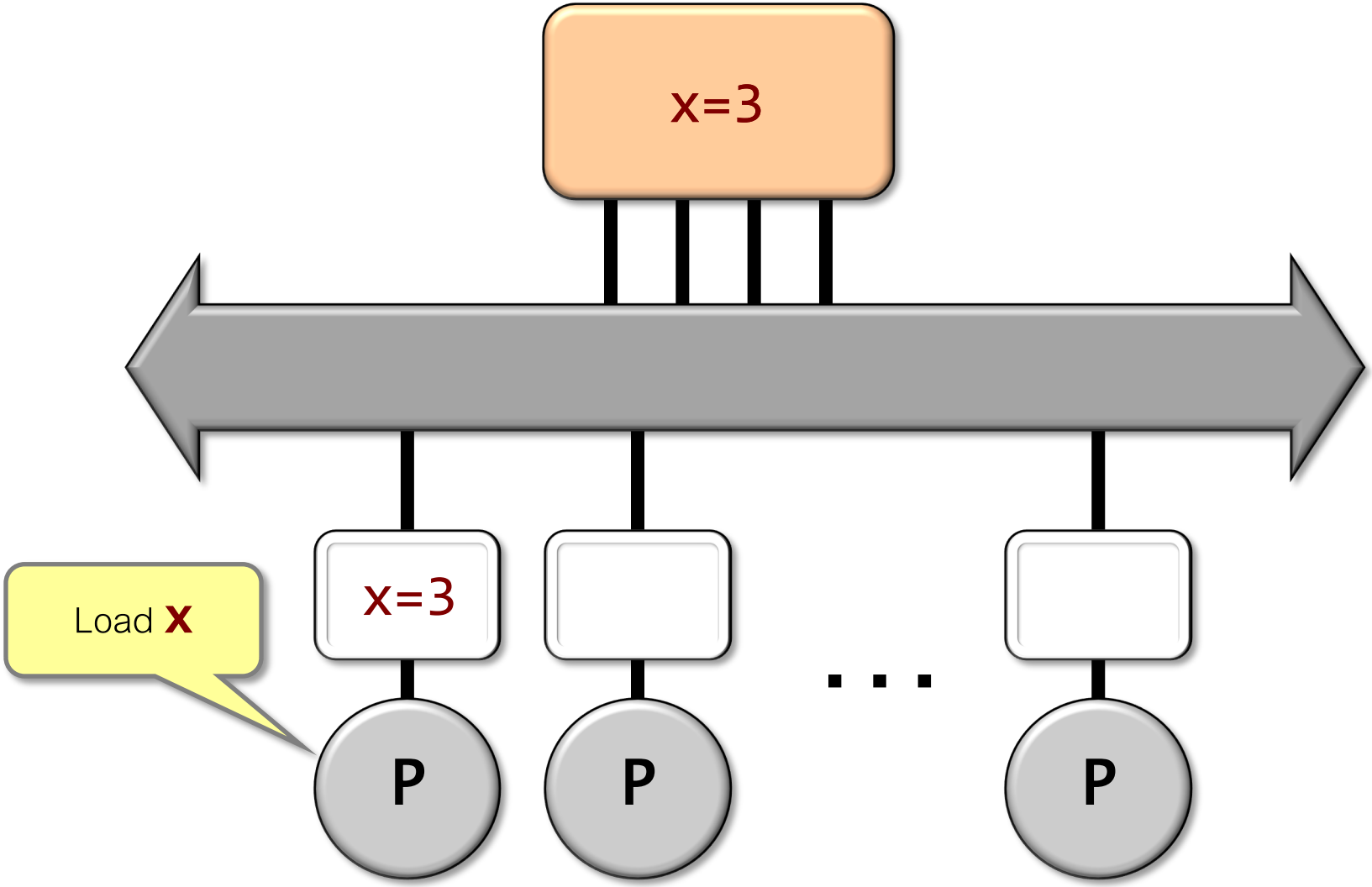
Multicore architecture with caches



$=Cache

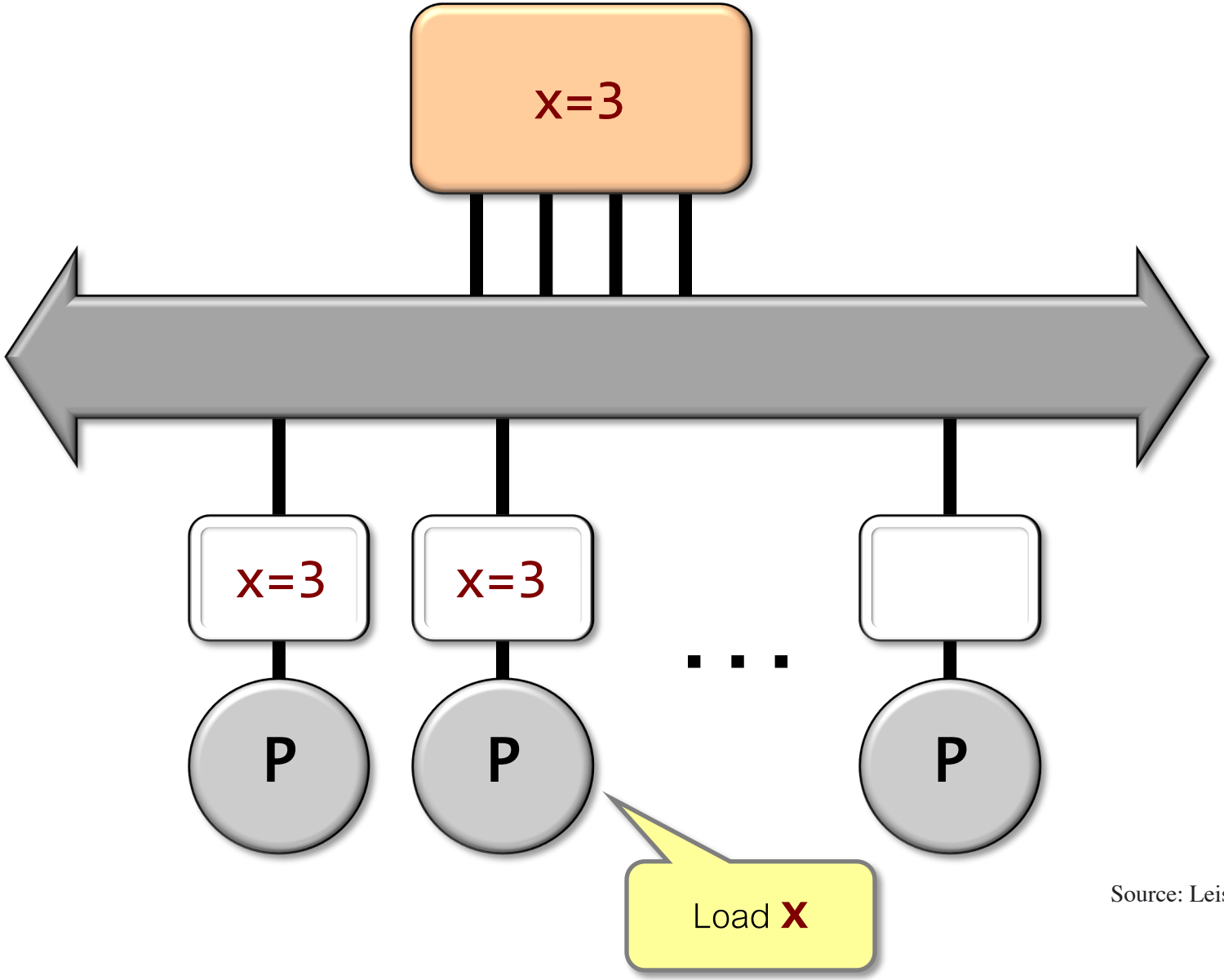
Chip Multiprocessor (CMP)

Cache coherence



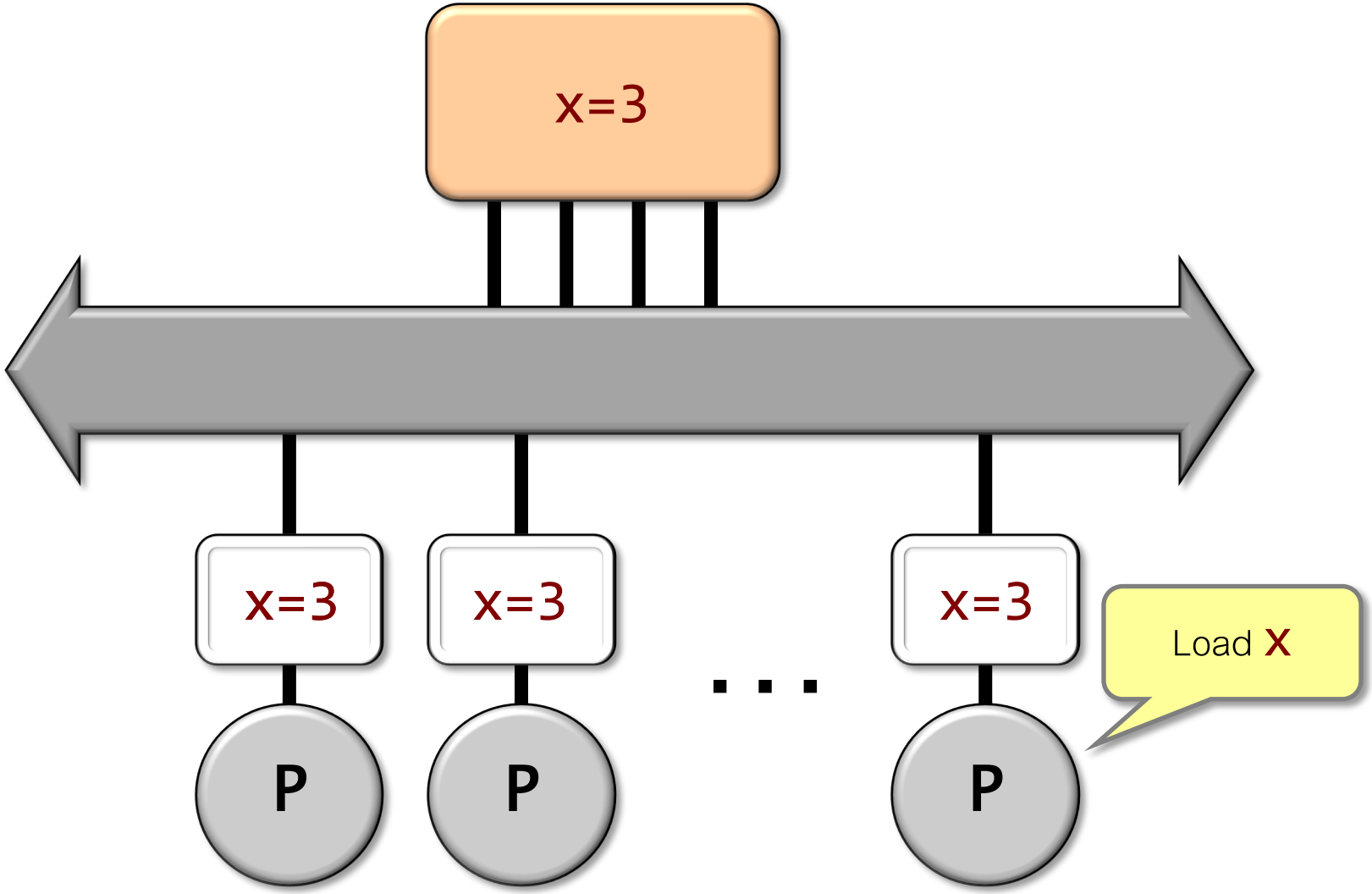
Source: Leiserson

Cache coherence



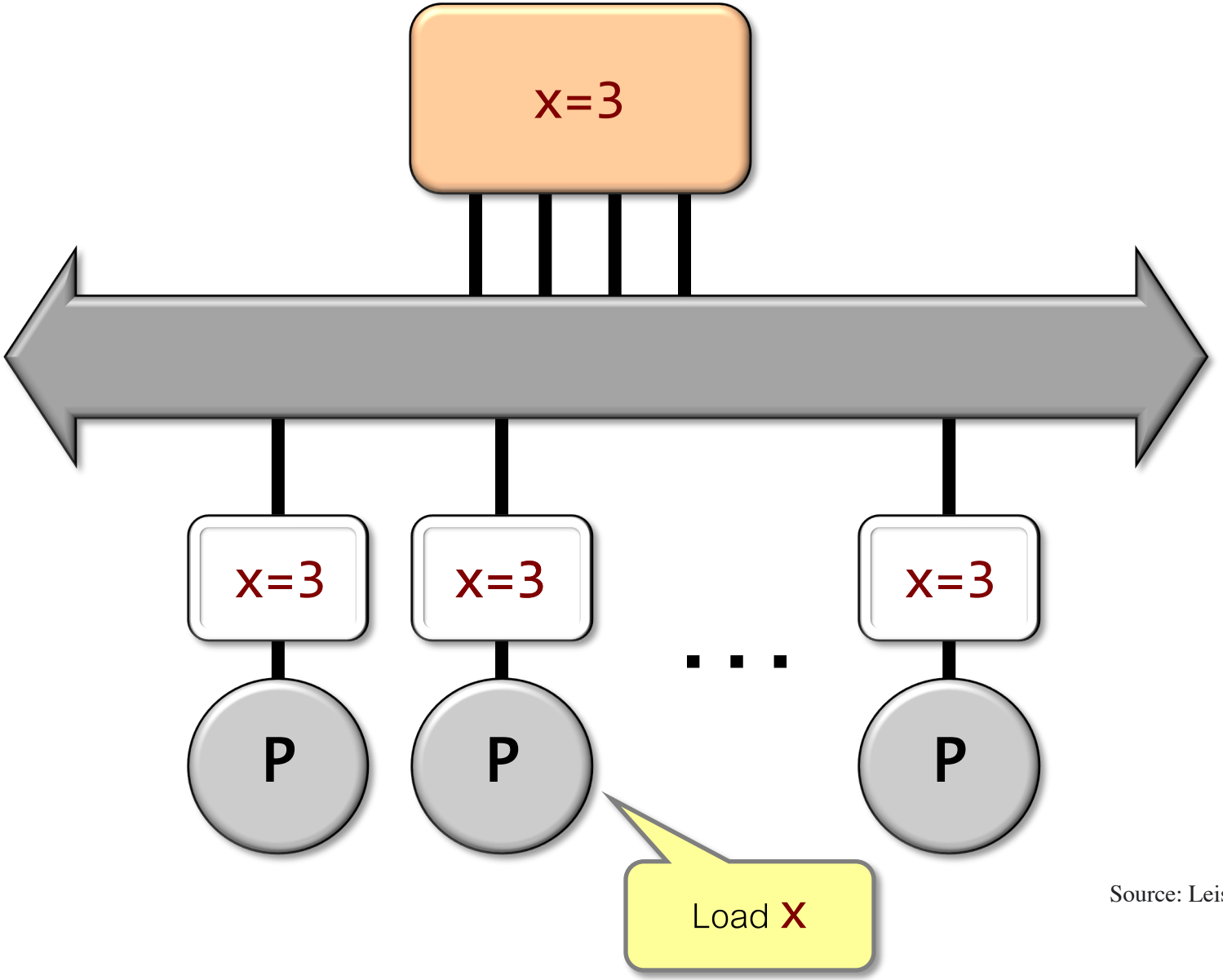
Source: Leiserson

Cache coherence



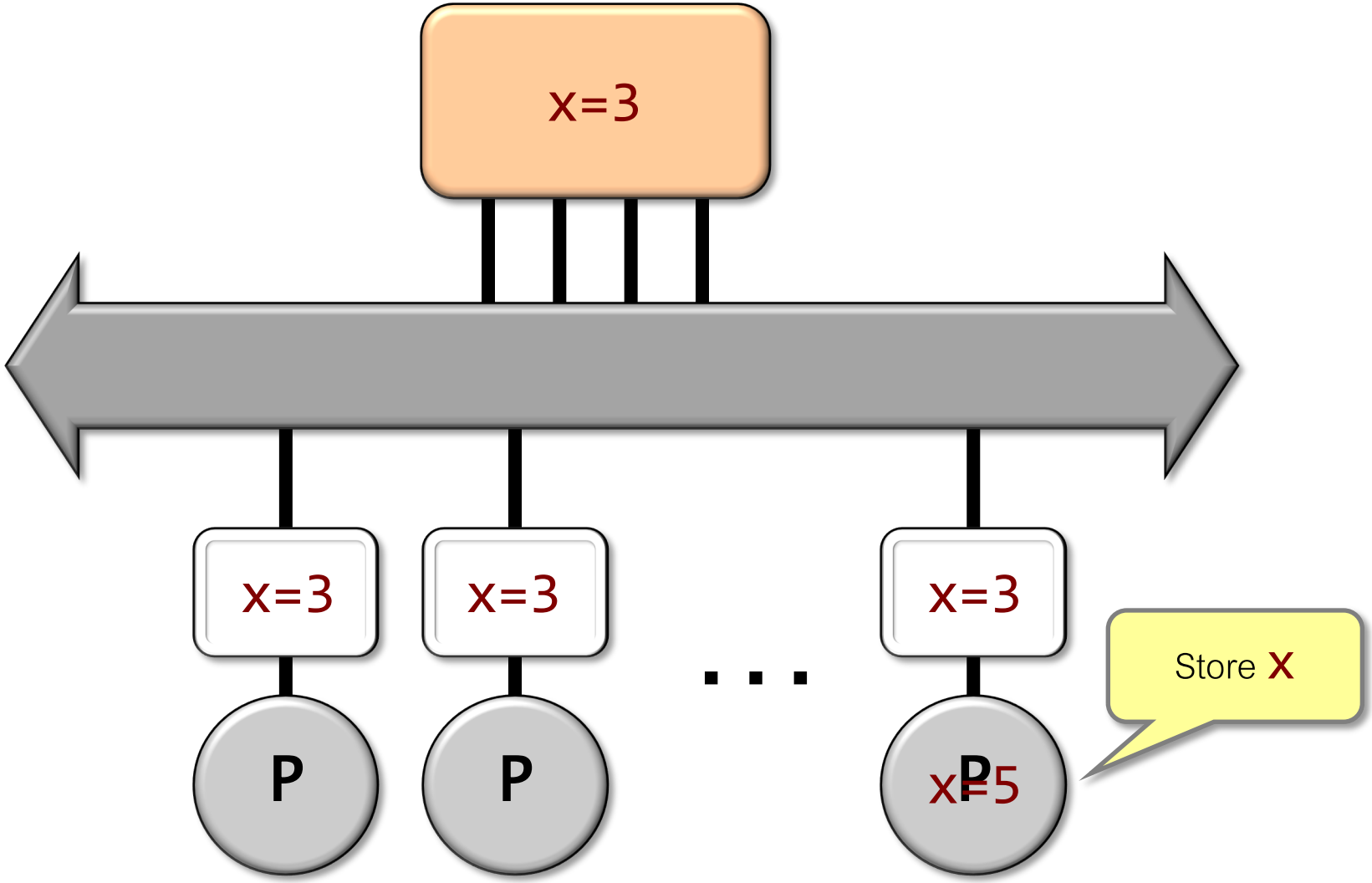
Source: Leiserson

Cache coherence



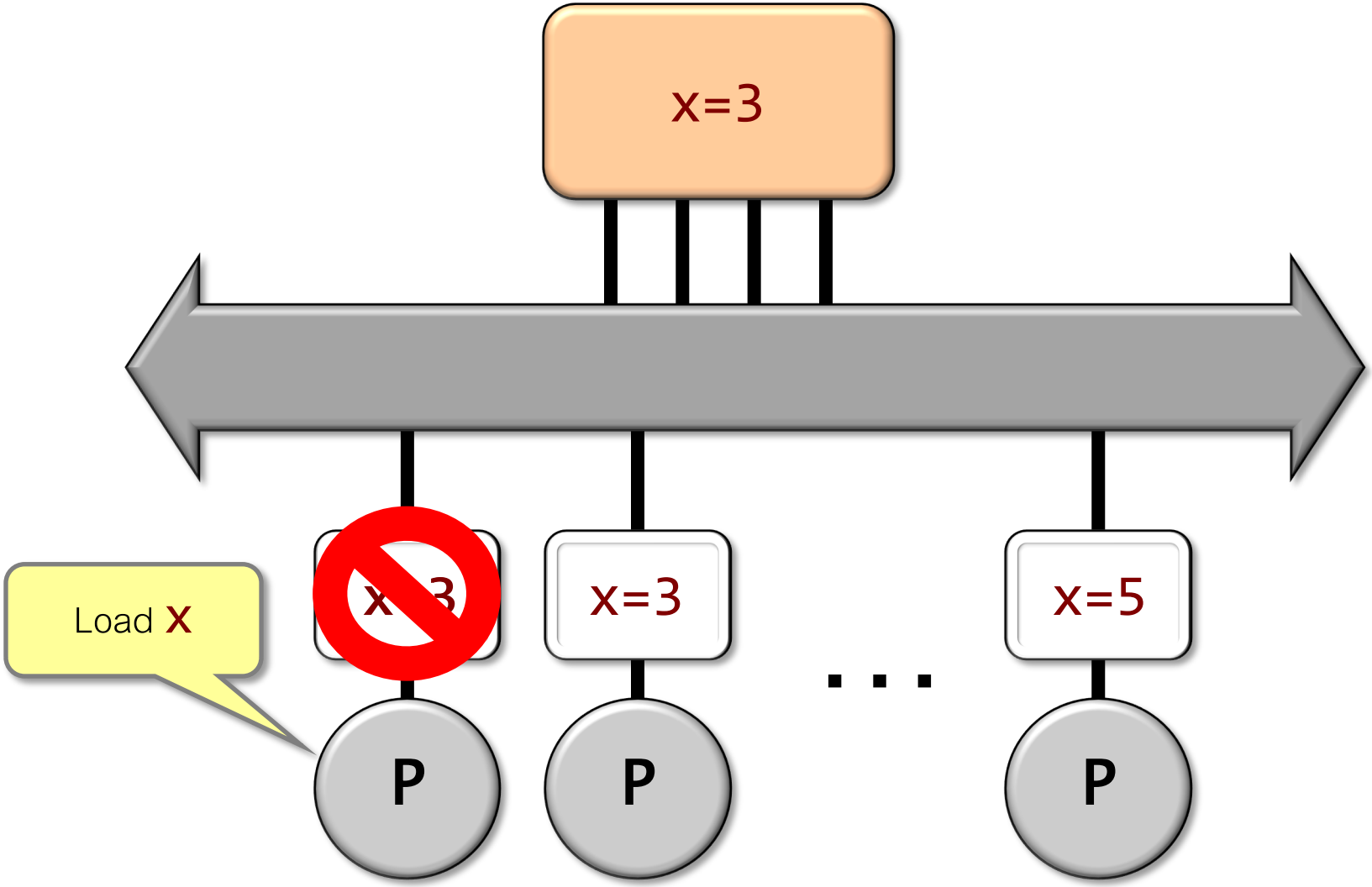
Source: Leiserson

Cache coherence



Source: Leiserson

Cache coherence

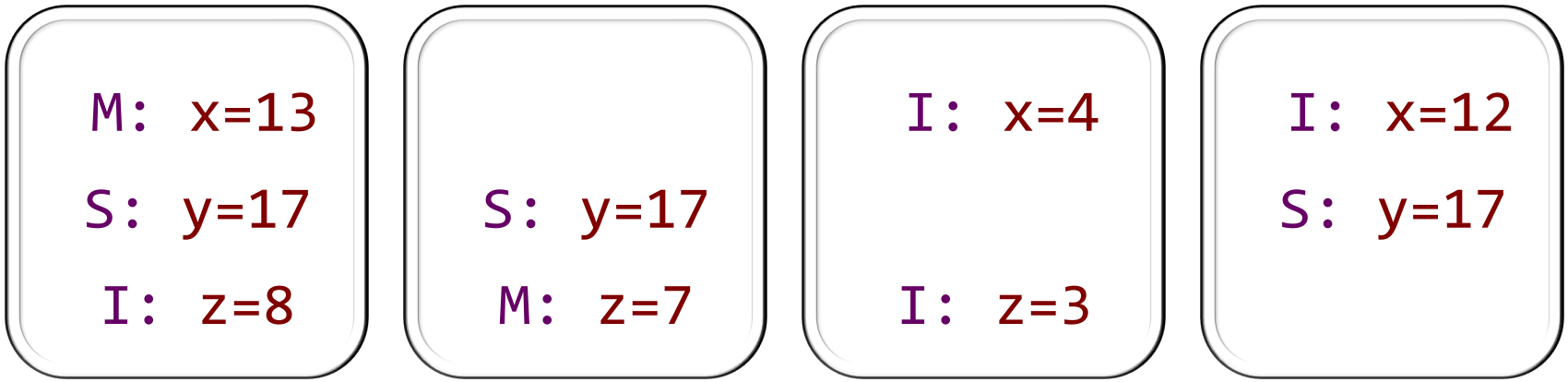


Source: Leiserson

MSI protocol

Each cache line is labeled with a state:

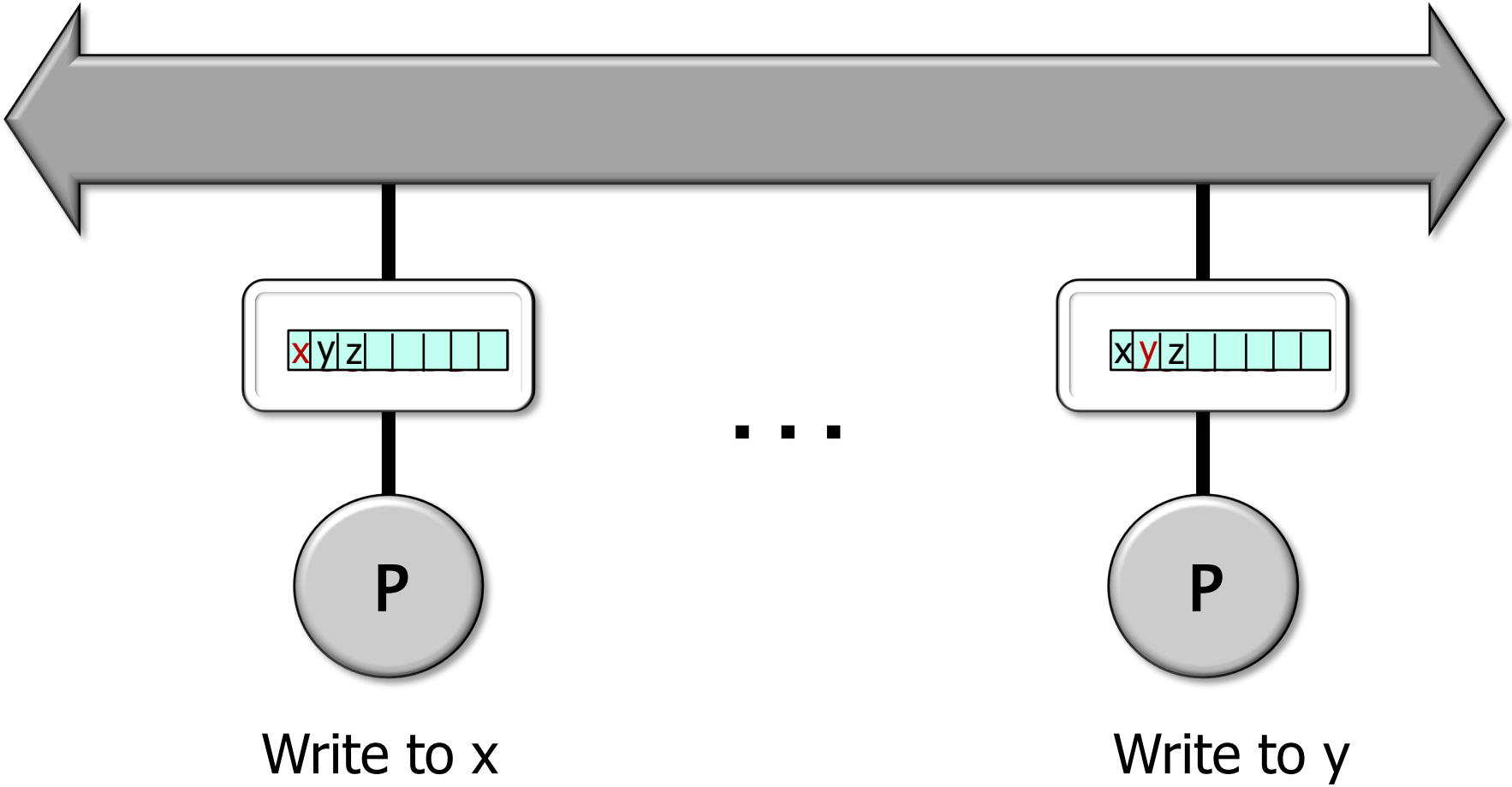
- M: cache block has been modified. No other caches contain this block in M or S states.
- S: other caches may be sharing this block.
- I: cache block is invalid (same as not there).



Before a cache modifies a location, the hardware first invalidates all other copies.

False sharing in multicores

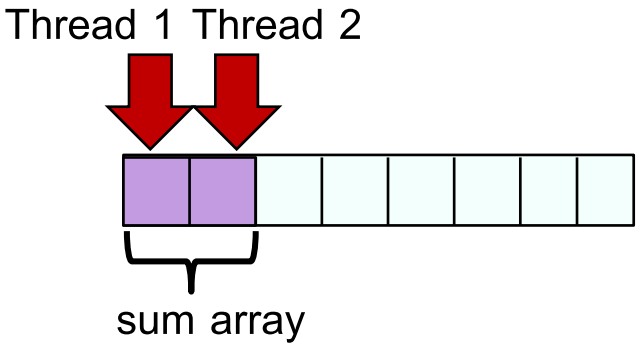
The cache line is bounced between processors because x and y are in the same cache line and because caches have to be coherent! This is called false sharing!



Parallel pi program: false sharing!

```
#include <omp.h>
static long num_steps = 100000;  double step;
#define NUM_THREADS 2
void main ()
{  int i, nthreads;  double pi, sum[NUM_THREADS];
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {        int i, id,nthrds;
             double x;
             id = omp_get_thread_num();
             nthrds = omp_get_num_threads();
             if (id == 0)   nthreads = nthrds;
             for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                          x = (i+0.5)*step;
                          sum[id] += 4.0/(1.0+x*x);
             }
     }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Below shows a cache line and the sum array in the cache line (cache line size is 64 bytes!)



Threads false share a cache line

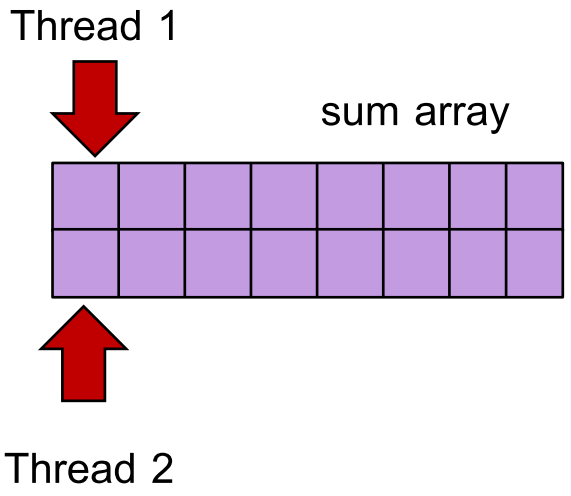
The pi Example: Eliminate false sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000; double step;
#define    PAD      ?     // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{   int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {         int i, id,nthrds;
              double x;
              id = omp_get_thread_num();
              nthrds = omp_get_num_threads();
              if (id == 0)    nthreads = nthrds;
              for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                                  x = (i+0.5)*step;
                                  ?+= 4.0/(1.0+x*x);
              }
     }
   for(i=0, pi=0.0;i<nthreads;i++)pi += ?* step;
}
```

How can padding resolve false sharing?

The pi Example: Eliminate false sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000; double step;
#define    PAD      8     // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{   int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {         int i, id,nthrds;
              double x;
              id = omp_get_thread_num();
              nthrds = omp_get_num_threads();
              if (id == 0)    nthreads = nthrds;
              for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                                  x = (i+0.5)*step;
                                  sum[id][0]+= 4.0/(1.0+x*x);
              }
     }
   for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0]* step;
}
```



Pad the array so each sum value is in a different cache line

Results: pi program padding

```
#include <omp.h>
static long num_steps = 100000; double step;
#define    PAD      8     // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{   int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {         int i, id,nthrds;
              double x;
              id = omp_get_thread_num();
              nthrds = omp_get_num_threads();
              if (id == 0)   nthreads = nthrds;
              for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                                  x = (i+0.5)*step;
                                  sum[id][0]+= 4.0/(1.0+x*x);
              }
     }
   for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0]* step;
}
```

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

Results: pi program critical section

```
#include <omp.h>
static long num_steps = 100000;  double step;
#define NUM_THREADS 2
void main ()
{ int nthreads; double  pi=0.0;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
   {          int i, id, nthrds;
              double x, sum;
              id = omp_get_thread_num();
              nthrds = omp_get_num_threads();
              if (id == 0)    nthreads = nthrds;
              for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
                              x = (i+0.5)*step;
                              sum += 4.0/(1.0+x*x);
              }
             #pragma omp critical
                        pi += sum * step;
  }
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don’t conflict

Example: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;  double step;
#define NUM_THREADS 2
void main ()
{ int nthreads; double  pi=0.0;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
   {          int i, id, nthrds;
              double x, sum;
              id = omp_get_thread_num();
              nthrds = omp_get_num_threads();
              if (id == 0)    nthreads = nthrds;
              for (i=id, sum=0.0;i< num_steps; i=i+nthrd
                                  x = (i+0.5)*step;
                                  sum += 4.0/(1.0+x*x);
              }
             #pragma omp critical
                        pi += sum * step;
  }
}
```

threads	1st SPMD	1st SPMD padded	SPMD critical
1	1.86	1.86	1.87
2	1.03	1.01	1.00
3	1.08	0.69	0.68
4	0.97	0.53	0.53

Example: pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;  double step;
void main ()
{    int i;     double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel
      {
           double x;
          #pragma omp for 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;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.

Results: pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;  double step;
void main ()
{    int i;     double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
          double x;
        #pragma omp for reductic
                for (i=0;i< num_step
                        x = (i+0.5)*s
                        sum = sum +
                }
      }
     pi = step * sum;
}
```

threads	1st SPMD	1st SPMD padded	SPMD critical	PI Loop and reduction
1	1.86	1.86	1.87	1.91
2	1.03	1.01	1.00	1.02
3	1.08	0.69	0.68	0.80
4	0.97	0.53	0.53	0.68

Performance and scalability considerations

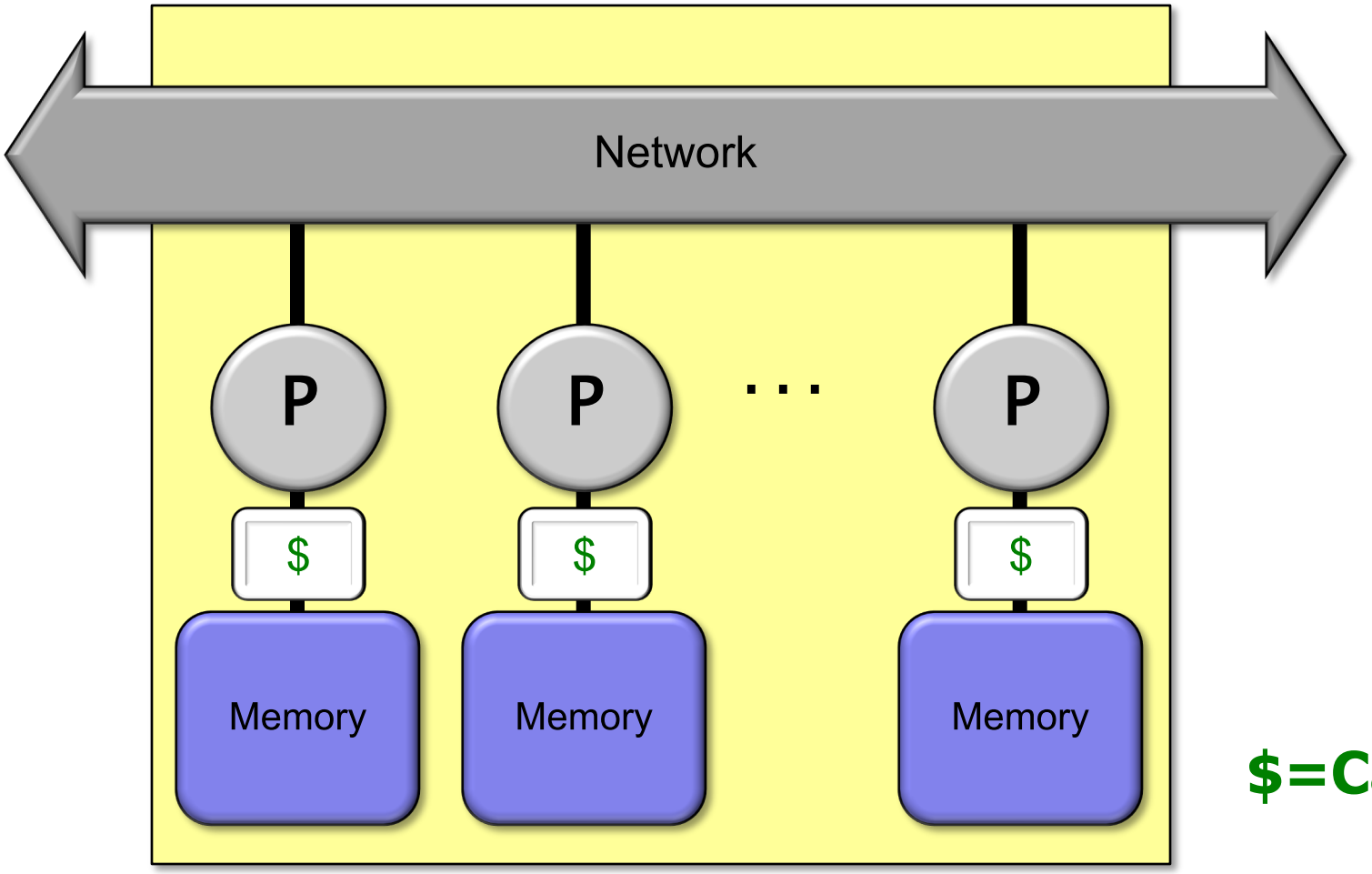
- Starting a parallel OpenMP region does not come for free
  - Overheads involved => performance impact
- Scalability tips
  - Try to parallelize the most outer loop possible
  - Consider cache locality, false sharing, etc.
  - Make sure parallel regions get sufficient speedup to overcome overhead
  - Overheads are dependent on machine, OS, and compiler!
  - Keep in mind Amdahl's law – what fraction of your program runs in parallel?

CSC367 Parallel computing

Distributed Memory Architectures and their Parallel Programming Model

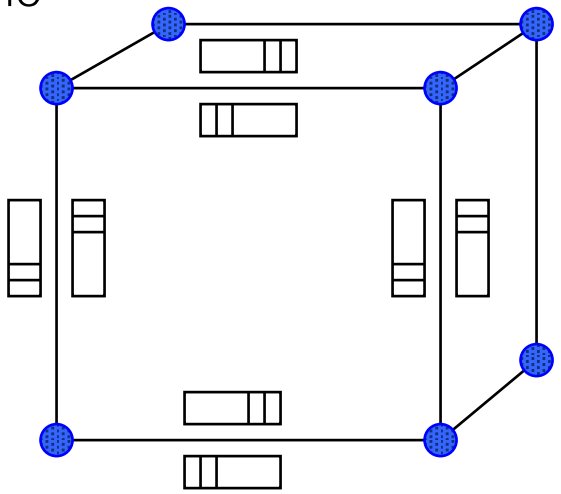
Thanks to Scinet, NSF XSEDE, James Demmel, and Kathy Yelick

Distributed Memory Architecture



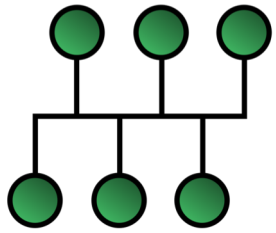
Historical Perspective

- Early distributed memory machines were:
  - Collection of microprocessors.
  - Communication was performed using bi-directional queues between nearest neighbors.
- Messages were forwarded by processors on path.
  - “Store and forward” networking
- There was a strong emphasis on topology in algorithms, in order to minimize the number of hops = minimize time



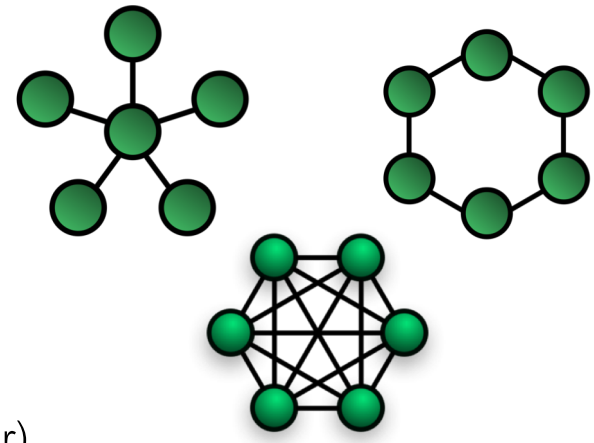
Network Analogy

- To have a large number of different transfers occurring at once, you need a large number of distinct wires
  - Not just a bus, as in shared memory



Bus-shared memory

- Networks are like streets:
  - Link = street.
  - Switch = intersection.
  - Distances (hops) = number of blocks traveled.
  - Routing algorithm = travel plan.



Network topologies

- Properties:
  - Latency: how long to get between nodes in the network.
    - Street: time for one car = dist (miles) / speed (miles/hr)
  - Bandwidth: how much data can be moved per unit time.
    - Street: cars/hour = density (cars/mile) * speed (miles/hr) * #lanes
    - Network bandwidth is limited by the bit rate per wire and #wires

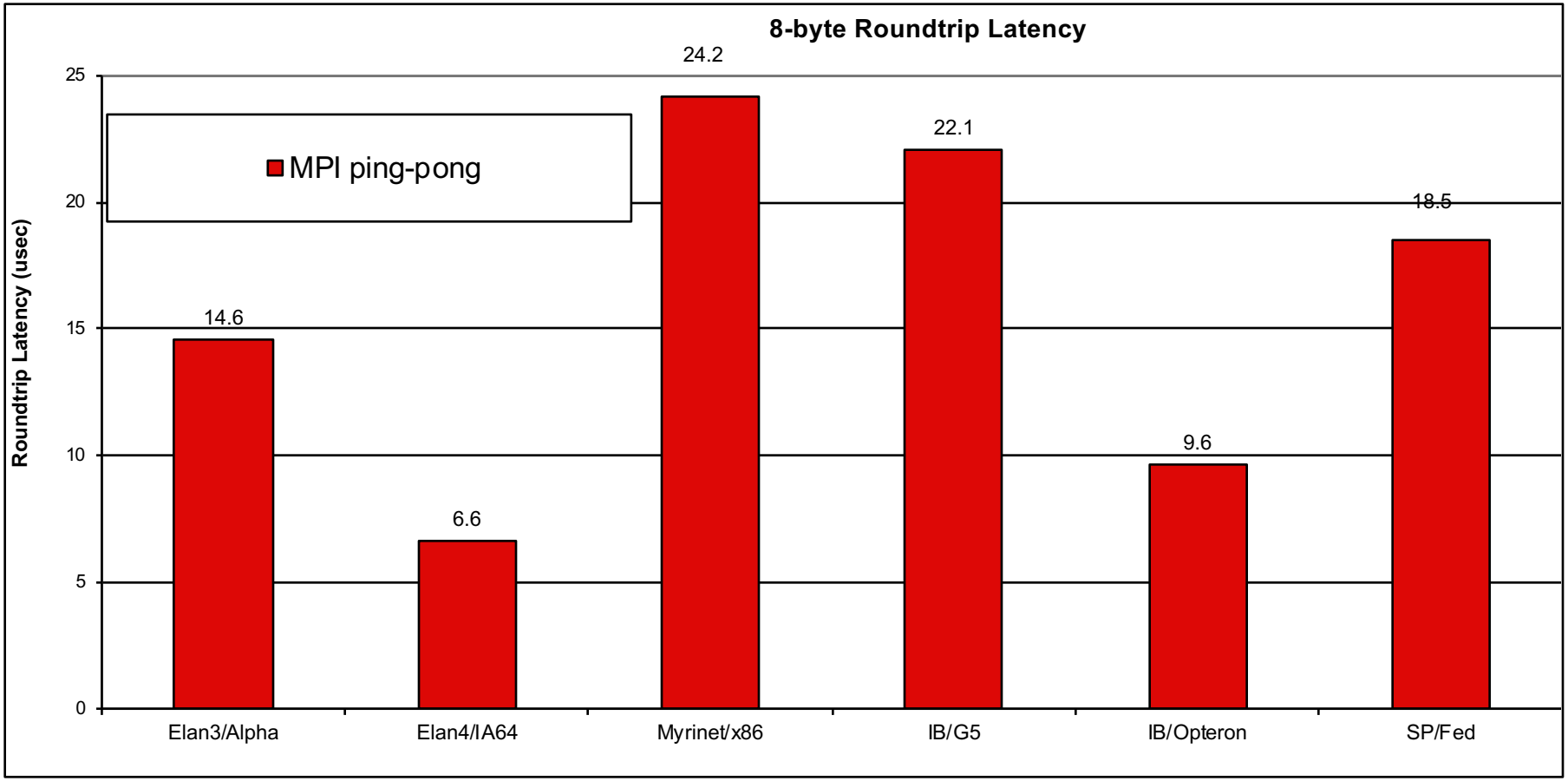
Design Characteristics of a Network

- Topology (how things are connected)
  - Crossbar; ring; 2-D, 3-D, higher-D mesh or torus; hypercube; tree; butterfly, …
- Routing algorithm:
  - Example in 2D torus: all east-west then all north-south (avoids deadlock).
- Switching strategy:
  - Circuit switching: full path reserved for entire message, like the telephone.
  - Packet switching: message broken into separately-routed packets, like the post office, or internet
- Flow control (what if there is congestion):
  - Stall, store data temporarily in buffers, re-route data to other nodes, tell source node to temporarily halt, discard, etc.

Performance Properties of a Network: Latency

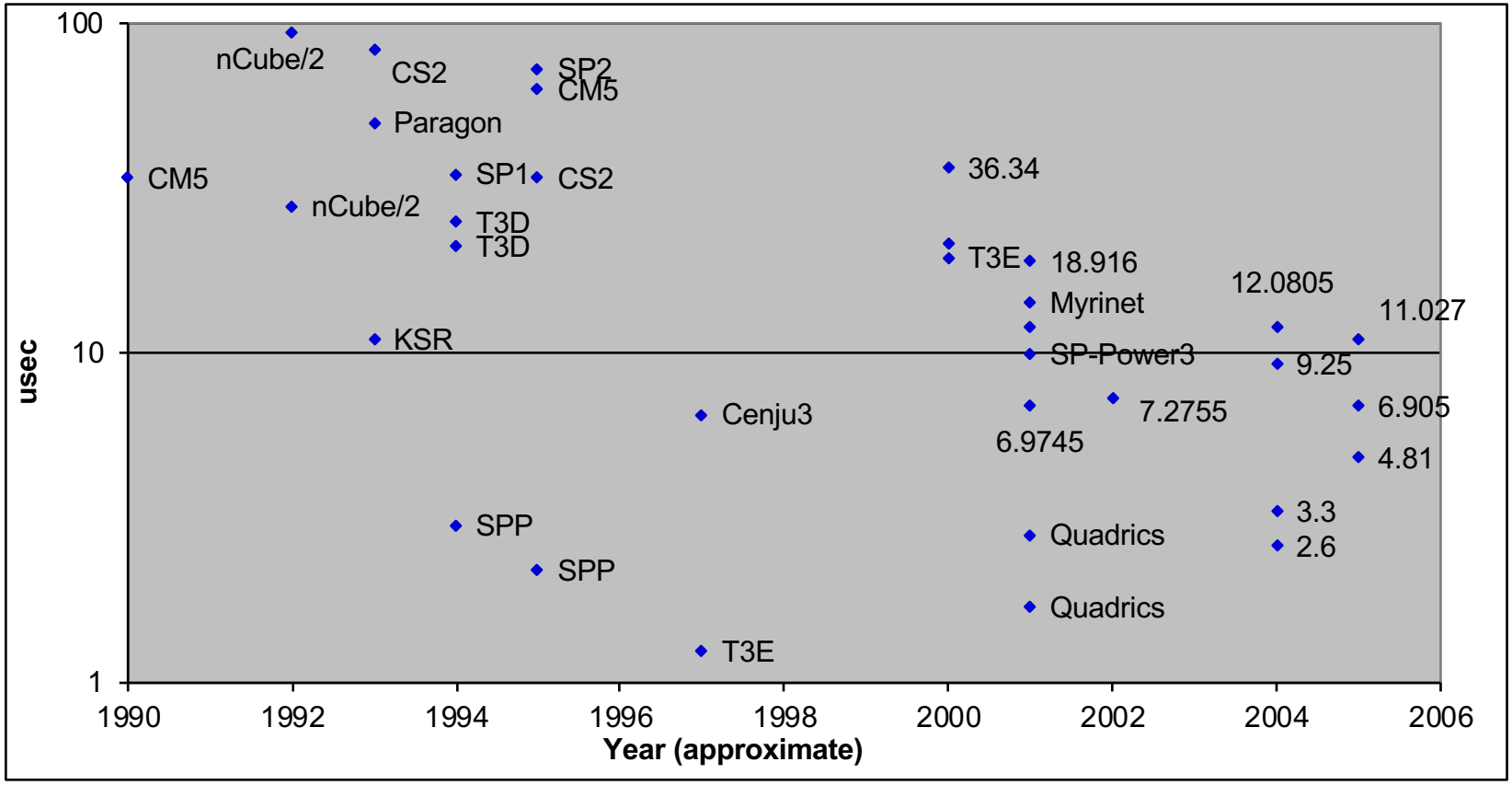
- Diameter: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes.
- Latency: delay between send and receive times
  - Latency tends to vary widely across architectures
  - Vendors often report hardware latencies (wire time)
  - Application programmers care about software latencies (user program to user program)
- Latency is key for programs with many small messages

Latency on Some Machines/Networks



- Latencies shown are from a ping-pong test using MPI
- These are roundtrip numbers: many people use ½ of roundtrip time to approximate 1-way latency (which can't easily be measured)

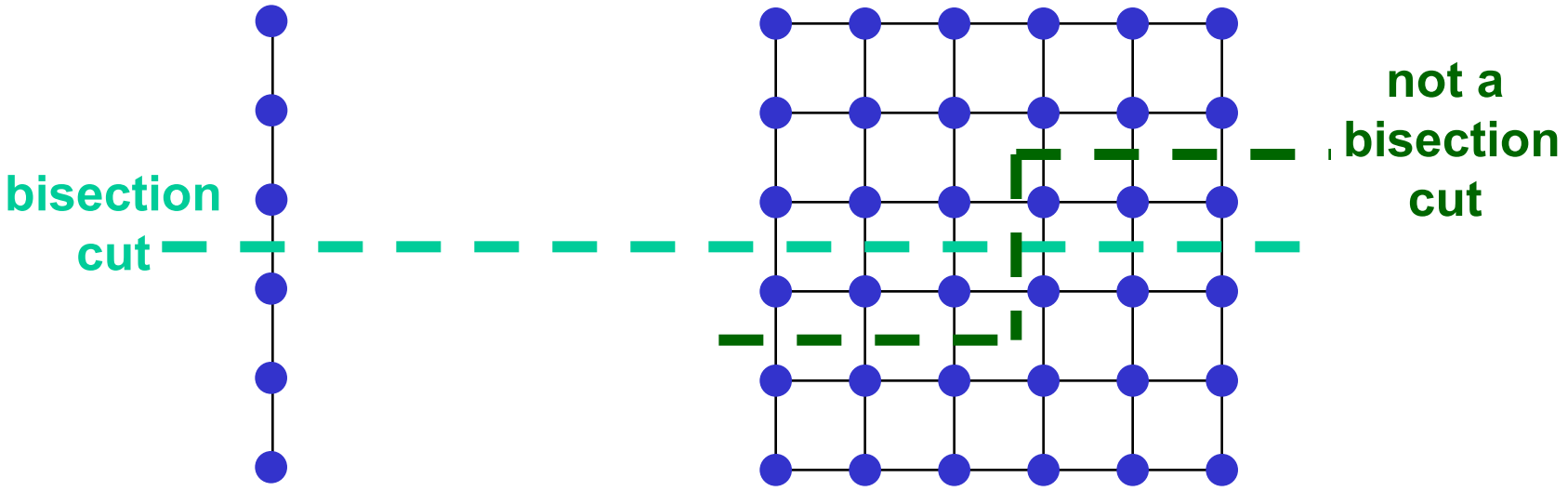
End to End Latency (1/2 roundtrip) Over Time



- Latency has not improved significantly, unlike Moore's Law
  - T3E (shmem) was lowest point – in 1997

Performance Properties of a Network: Bisection Bandwidth

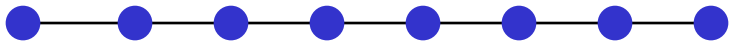
- Bisection bandwidth: bandwidth across smallest cut that divides network into two equal halves
- Bandwidth across "narrowest" part of the network



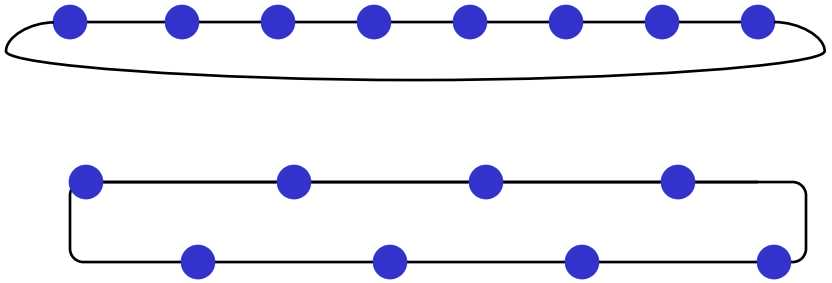
- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

Linear and Ring Topologies

- Linear array (1D mesh)



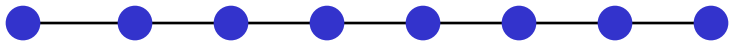
  - Diameter = ?;
  - Bisection bandwidth = ?(in units of link bandwidth).
- Torus or Ring



  - Diameter = ?;
  - Bisection bandwidth = ?.
  - Natural for algorithms that work with 1D arrays.

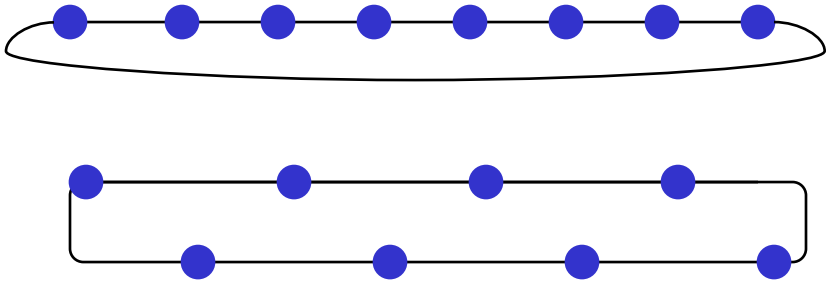
Linear and Ring Topologies

- Linear array (1D mesh)



  - Diameter = n-1;
  - Bisection bandwidth = 1 (in units of link bandwidth).

- Torus or Ring

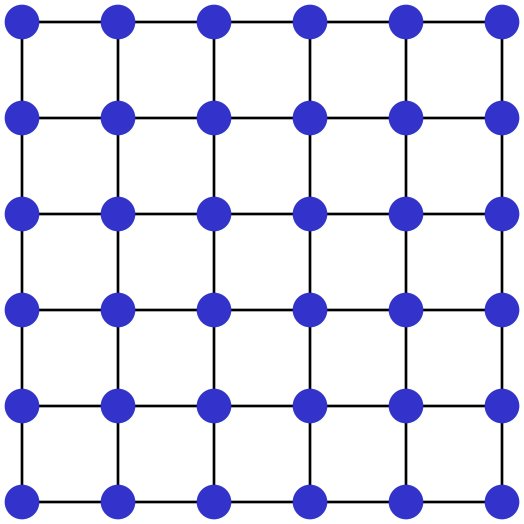


  - Diameter = n/2;
  - Bisection bandwidth = 2.
  - Natural for algorithms that work with 1D arrays.

Meshes and Tori

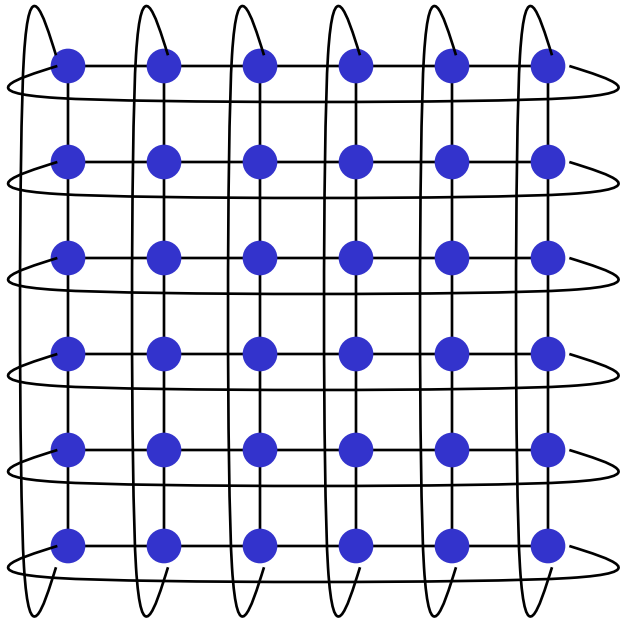
Two dimensional mesh

- Diameter = ?
- Bisection bandwidth = ?



Two dimensional torus

- Diameter = ?
- Bisection bandwidth = ?

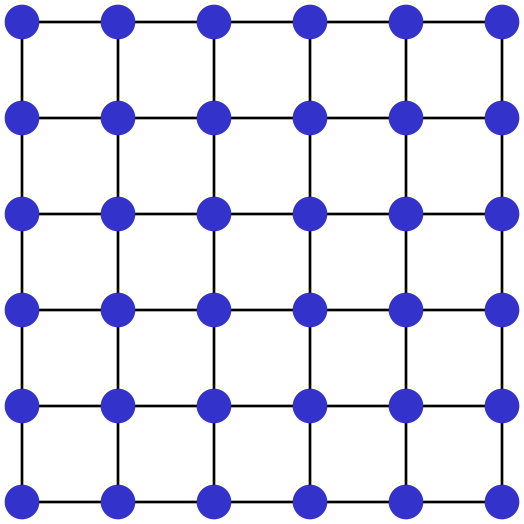


- Generalizes to higher dimensions
  - Cray XT (eg Hopper@NERSC) uses 3D Torus
- Natural for algorithms that work with 2D and/or 3D arrays (matmul)

Meshes and Tori

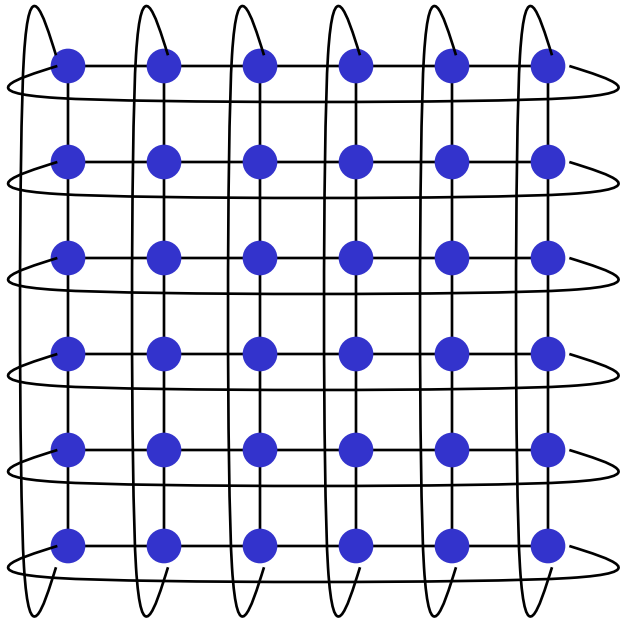
Two dimensional mesh

- Diameter = 2 * (sqrt(n) – 1)
- Bisection bandwidth = sqrt(n)



Two dimensional torus

- Diameter = ?
- Bisection bandwidth = ?

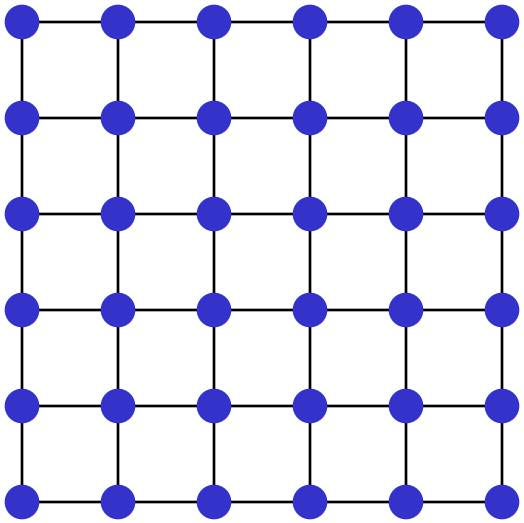


- Generalizes to higher dimensions
  - Cray XT (eg Hopper@NERSC) uses 3D Torus
- Natural for algorithms that work with 2D and/or 3D arrays (matmul)

Meshes and Tori

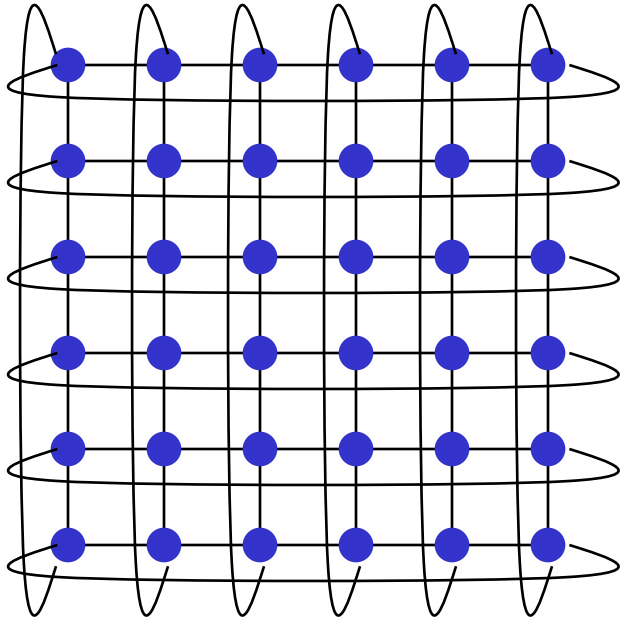
Two dimensional mesh

- Diameter = 2 * (sqrt(n) – 1)
- Bisection bandwidth = sqrt(n)



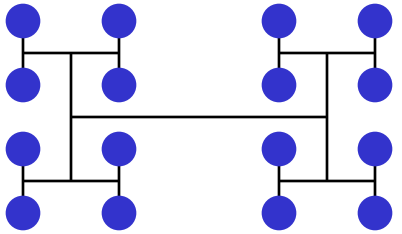
Two dimensional torus

- Diameter = sqrt(n)
- Bisection bandwidth = 2 * sqrt(n)

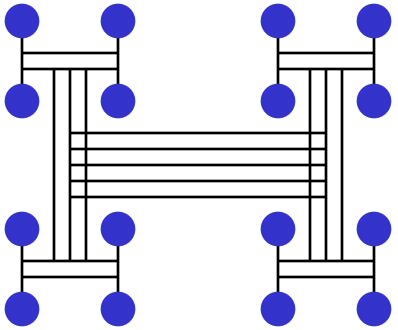


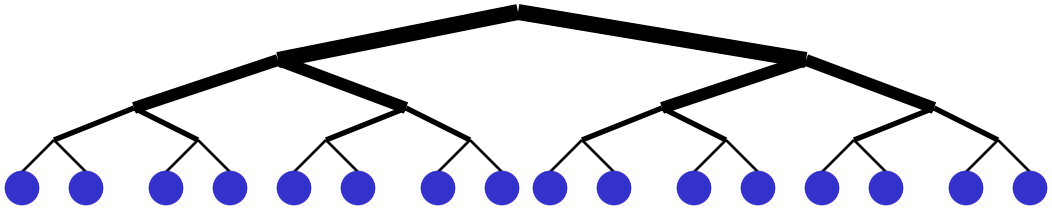
- Generalizes to higher dimensions
  - Cray XT (eg Hopper@NERSC) uses 3D Torus
- Natural for algorithms that work with 2D and/or 3D arrays (matmul)

Trees



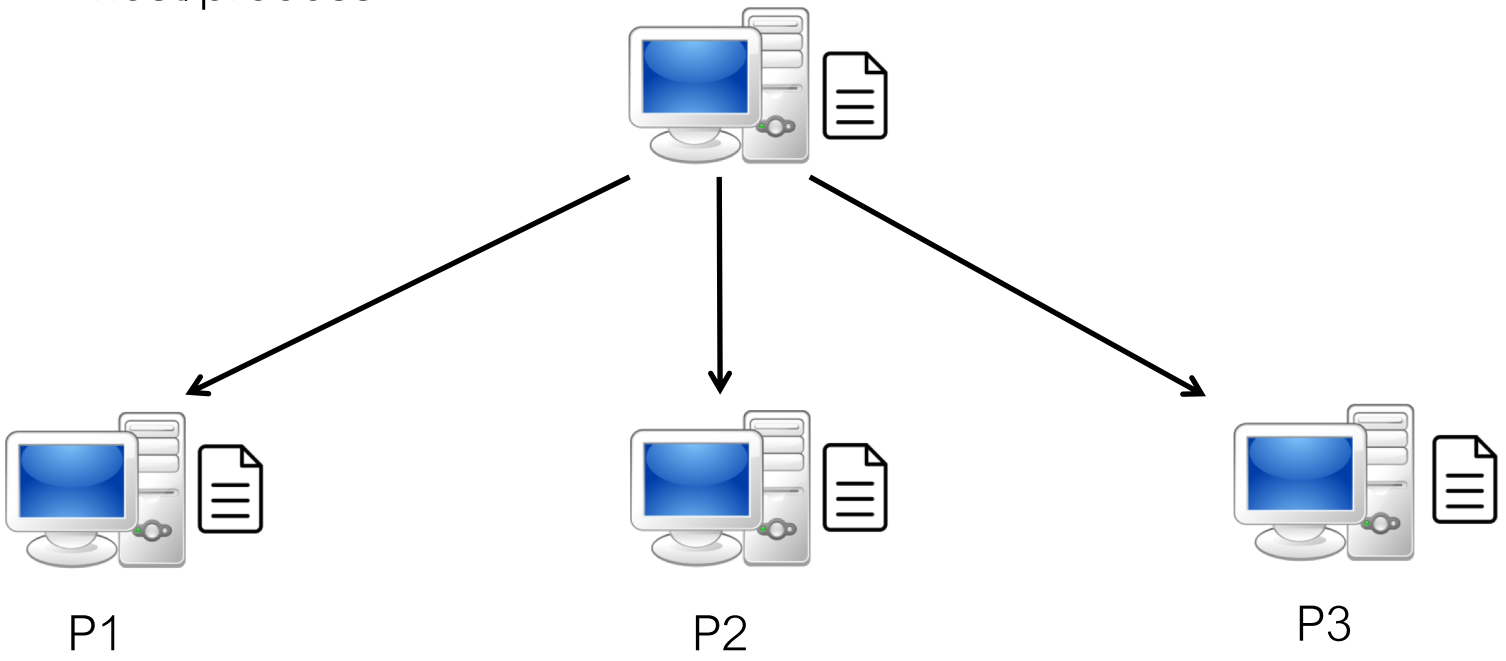
- Bisection bandwidth = 1.
- Easy layout as planar graph.
- Many tree algorithms benefit from tree topologies (e.g., summation).
- Fat trees avoid bisection bandwidth problem:
  - More (or wider) links near top.
  - Example: Thinking Machines CM-5.





Message passing model

- Single Program Multiple Data (SPMD)
  - All processes execute same code, communicate via messages
  - Technically, it does support executing different programs on each host/process



Why message passing?

- Build a parallel multi-computer from lots and lots of nodes
  - Nodes are connected with a network
  - Partition the data across the nodes, computation in parallel
  - If local data is needed on remote node, send it over the interconnect
  - Computation is done collectively
  - Can always add more and more nodes => bottleneck is not the number of cores or memory on a single node
    - Scale out instead of scale up: Can increase the data size while increasing processors
- Downside: harder to program
  - Every communication has to be hand-coded by the developer