CSCI 5451: Introduction to Parallel Computing

Lecture 6: Threads

computer science
& engineering

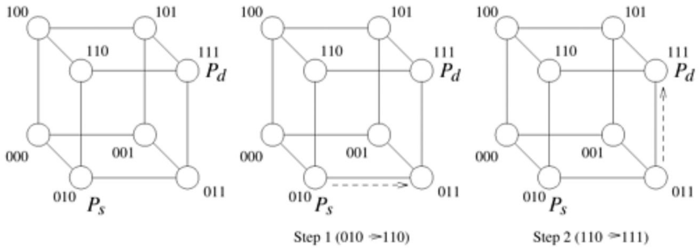
University of Minnesota
Driven to Discover®

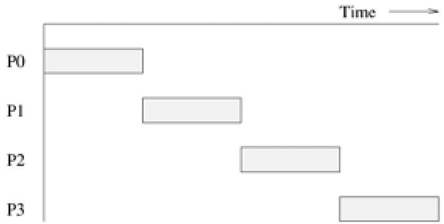
Lecture Overview

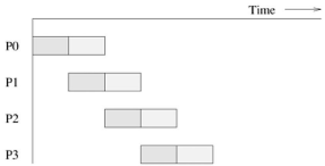
- **Recap**
- Wrap Up Mapping (cont'd)
  - Minimizing Interactions
  - Processes to processors
- Threads
  - Background
  - Pi Computation Example
  - Threading in Detail

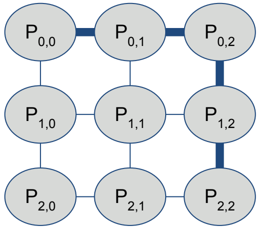
Recap

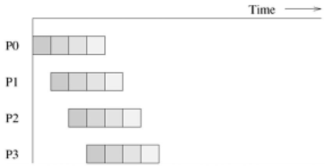
Message Passing





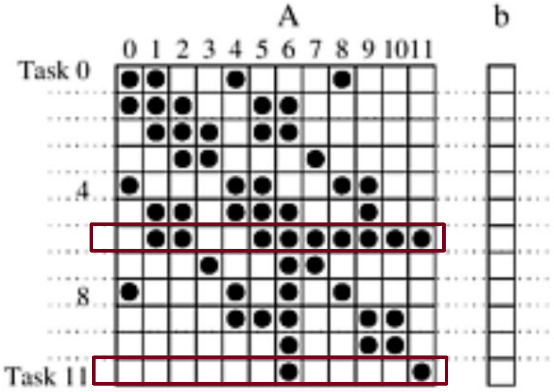


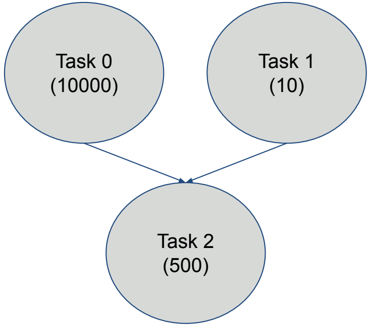




Recap

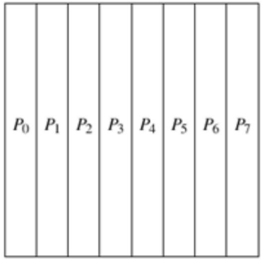
Load Balancing

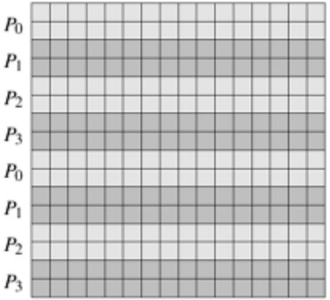


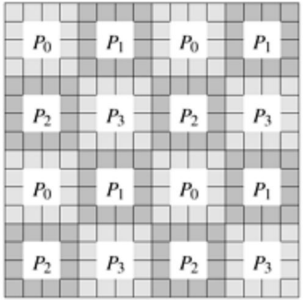


row-wise distribution

column-wise distribution







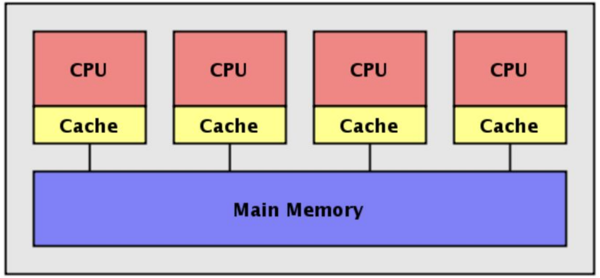
Lecture Overview

- Recap
- **Wrap Up Mapping (cont'd)**
  - **Minimizing Interactions**
  - Processes to processors
- Threads
  - Background
  - Pi Computation Example
  - Threading in Detail

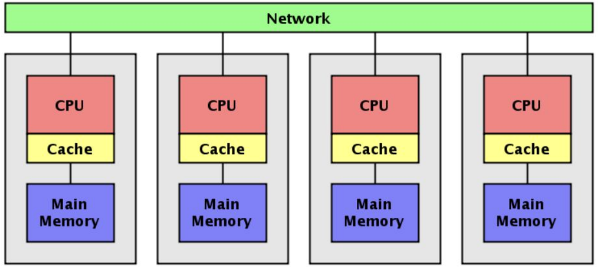
Interactions in Shared & Distributed Memory

Interactions occur both in shared-address space & distributed memory settings

- ❑ Shared-Address Space → Cache coherence overhead
- ❑ Distributed Memory → Time spent on communication



Source: Kaminsky/Parallel Java



Source: Kaminsky/Parallel Java

General Strategies for Minimizing Interaction

- Minimize Volume of Data Exchange
- Minimize Frequency of Interactions
- Minimize Contentions/Hot Spots
- Overlap comms + Interactions
- Replicate Data + Computations
- Use Optimized Collective Interactions (MPI)
- Overlap interactions with other interactions

General Strategies for Minimizing Interaction

Recap

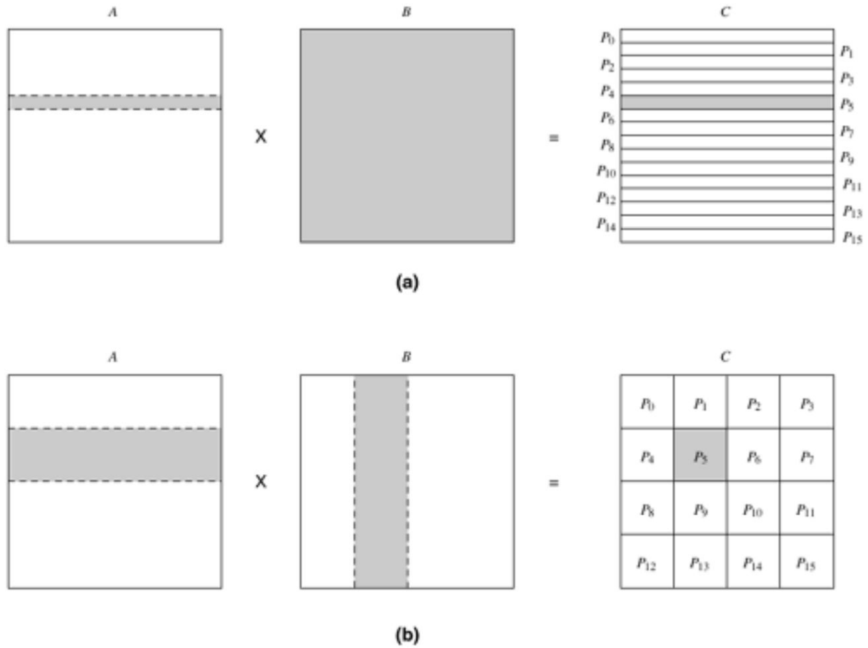
- Minimize Volume of Data Exchange

$$t_{comm} = t_s + kt_h + t_w \boxed{m}$$

Minimizing message size lowers the above term.

General Strategies for Minimizing Interaction

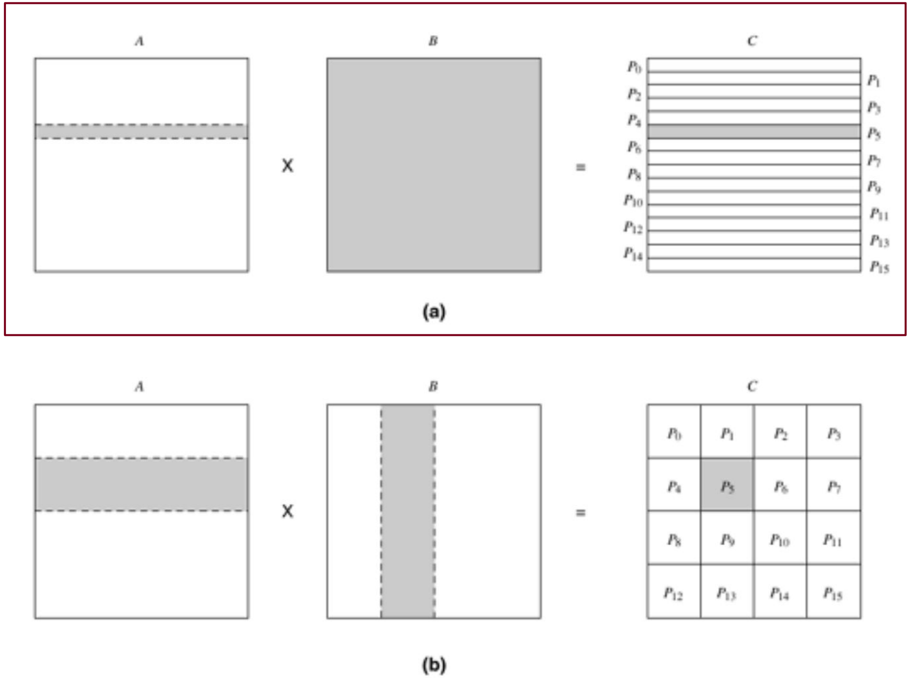
- Minimize Volume of Data Exchange



Suppose we use an output decomposition for matrix multiplication, but matrices A and B are too large to fit on any one processor in a distributed memory setting (i.e. each processor only has some subset of A and B). Therefore, we will need the processors to communicate their values of A,B.

General Strategies for Minimizing Interaction

- Minimize Volume of Data Exchange

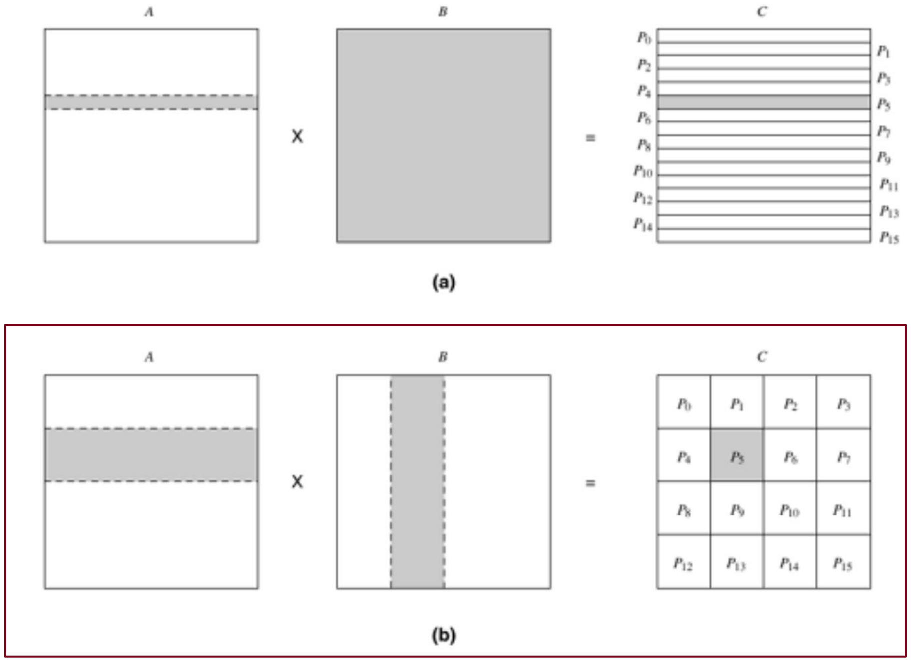


Bad Mapping

Each process requires $n^2/p + n^2$ accesses to the data → We will have to communicate all of B to calculate each row of C

General Strategies for Minimizing Interaction

- Minimize Volume of Data Exchange

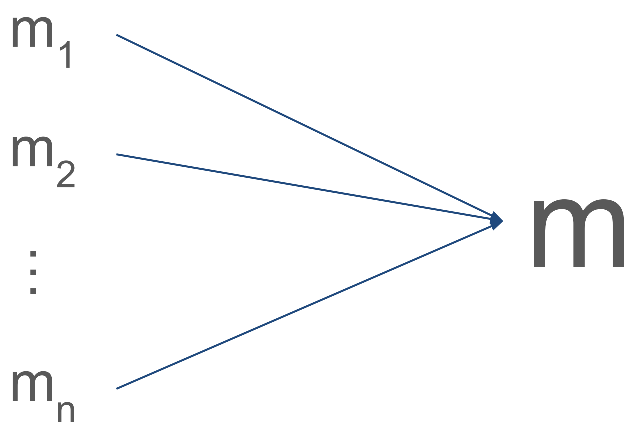


Good Mapping

Each process requires $2n^2/\text{sqrt}(p)$ accesses to the data → We can dramatically reduce how much information must be shared

General Strategies for Minimizing Interaction

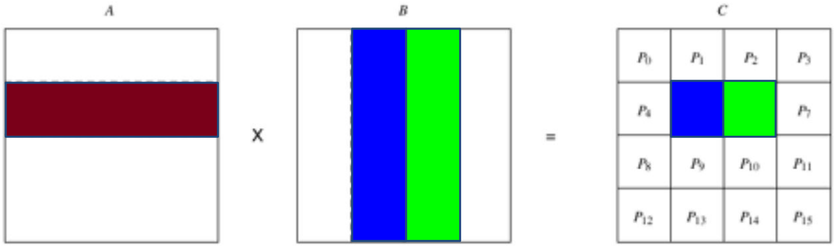
- Minimize Frequency of Interactions



Grouping messages, eliminating unnecessary messages

General Strategies for Minimizing Interaction

- ❑ Minimize Contentions/Hot Spots

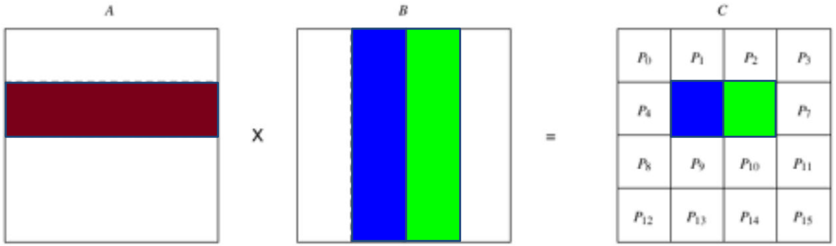


(b)

Both the blue + green processes at right need to access the same elements of A → If they try to access similar cache lines at the same time on a shared memory system, there can be delays

General Strategies for Minimizing Interaction

- Minimize Contentions/Hot Spots



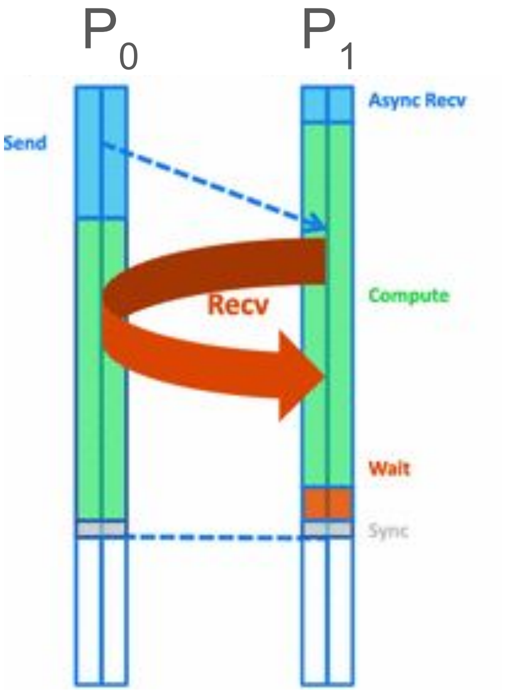
(b)

Both the blue + green processes at right need to access the same element of A → If they try to execute communications with the process which currently holds a given row of A in a distributed memory setting, there might be communication delays

General Strategies for Minimizing Interaction

- Overlap comms + Interactions

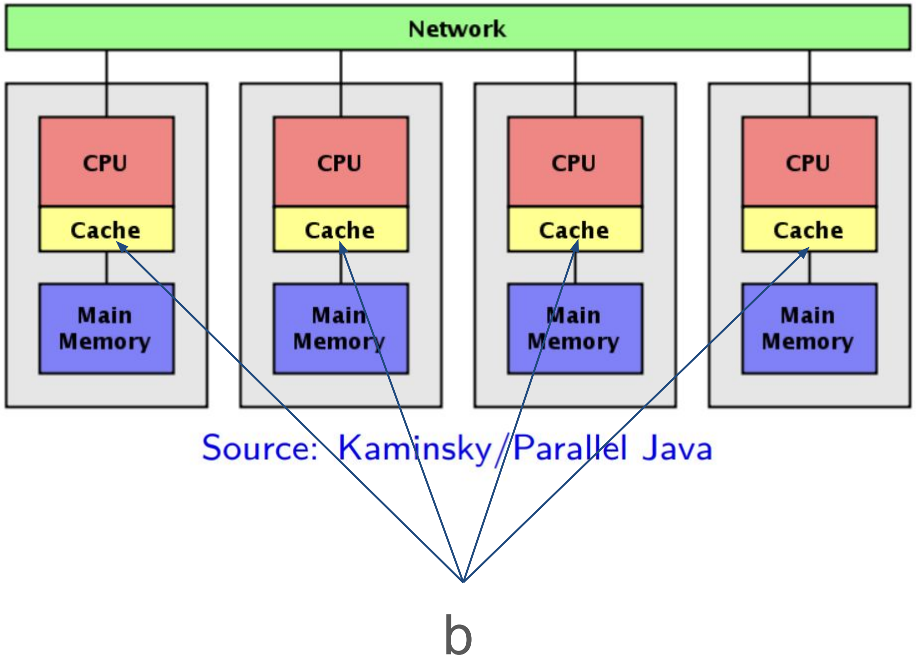
Prepare communications in advance so that we can always be computing with the data we already have (similar to pre-fetching)



General Strategies for Minimizing Interaction

- ❑ Replicate Data + Computations

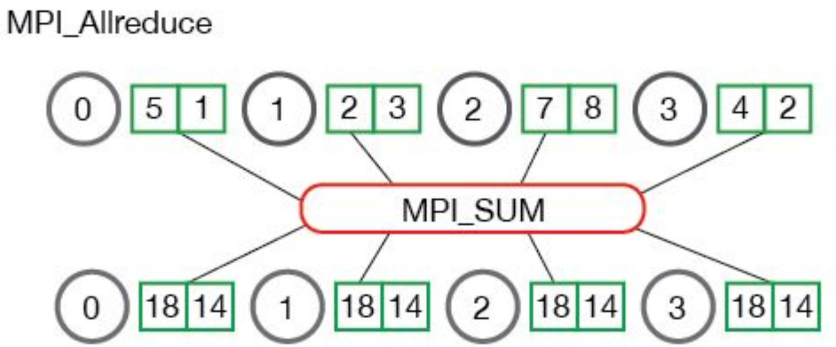
Copy important data structures across each process, rather than re-sending or recomputing each time they are needed (e.g. Ab = y)



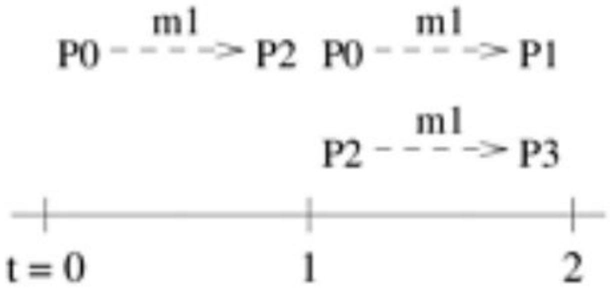
General Strategies for Minimizing Interaction

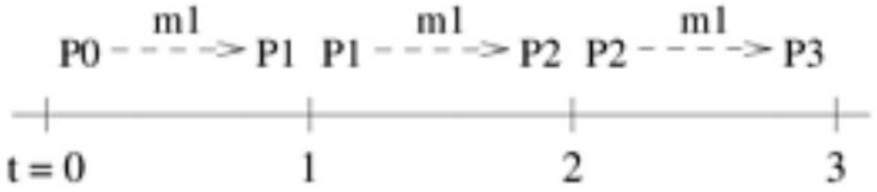
To be covered in greater detail in the next few weeks

- ❑ Use Optimized Collective Interactions (MPI)



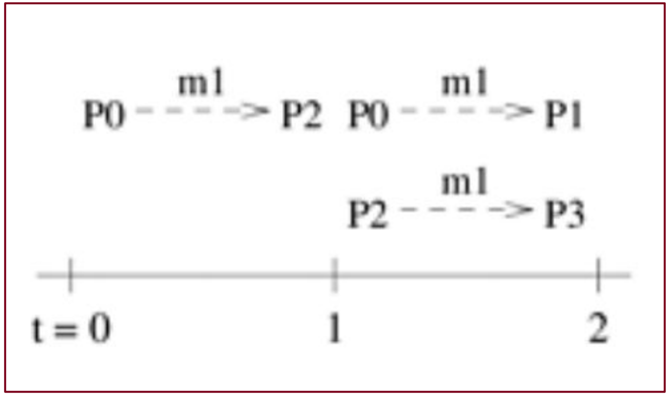
General Strategies for Minimizing Interaction

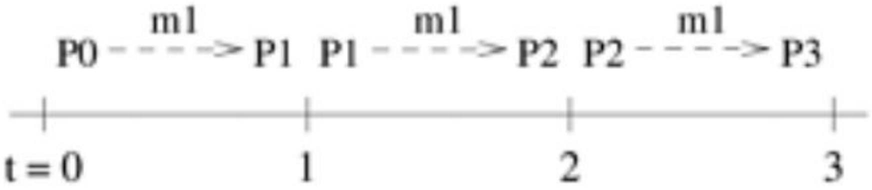




- ❑ Overlap interactions with other interactions

General Strategies for Minimizing Interaction

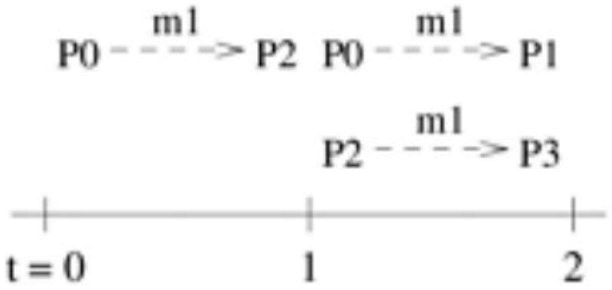


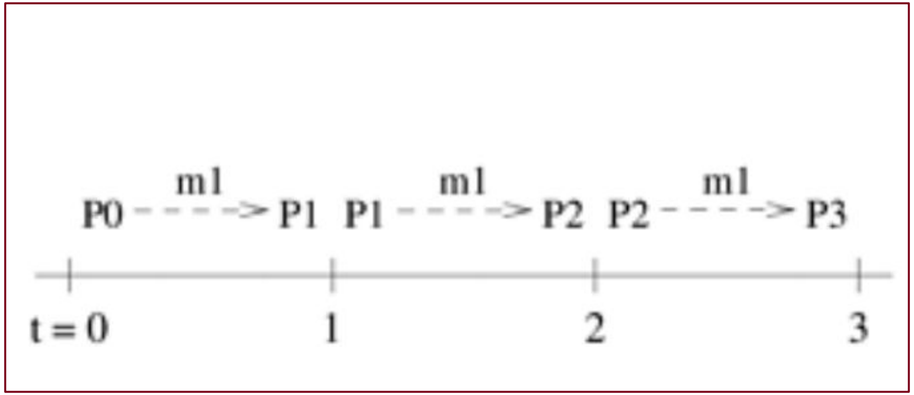


Fast Broadcast

- Overlap interactions with other interactions

General Strategies for Minimizing Interaction





Slow Broadcast

- Overlap interactions with other interactions

Lecture Overview

- Recap
- **Wrap Up Mapping (cont'd)**
  - Minimizing Interactions
  - **Processes to processors**
- Threads
  - Background
  - Pi Computation Example
  - Threading in Detail

Tasks vs. Processes vs. Processors

- In general, creating a well running parallel program requires performing the following in order:

1. Define the tasks
2. Define task interactions + TDG
3. Determine which processes will work on which tasks
4. Make sure that each process is properly mapped onto hardware

Tasks vs. Processes vs. Processors

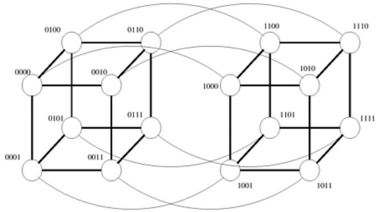
- In general, creating a well running parallel program requires performing the following in order:

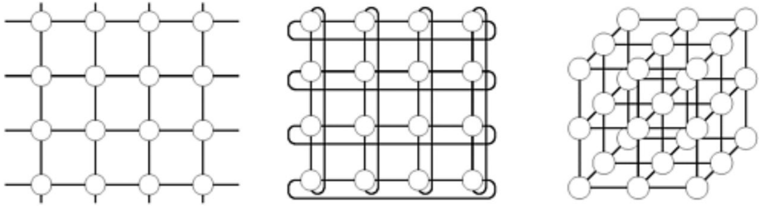
1. Define the tasks
2. Define task interactions + TDG
3. Determine which processes will work on which tasks
4. Make sure that each process is properly mapped onto hardware

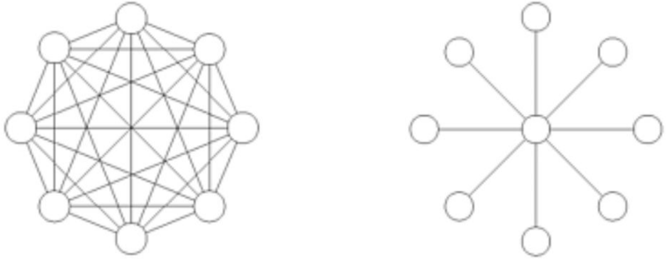
We want to ensure that the way we organize our processes logically is how they are mapped onto the hardware.

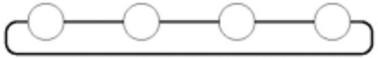
Process communications

- Similar to constructing the TDG, we must consider how each process will communicate with one another
- We use the same topologies we have previously discussed to organize these communications (hypercube, linear array, 2-d mesh, etc.)
- We can think of this as grouping together task-level communications



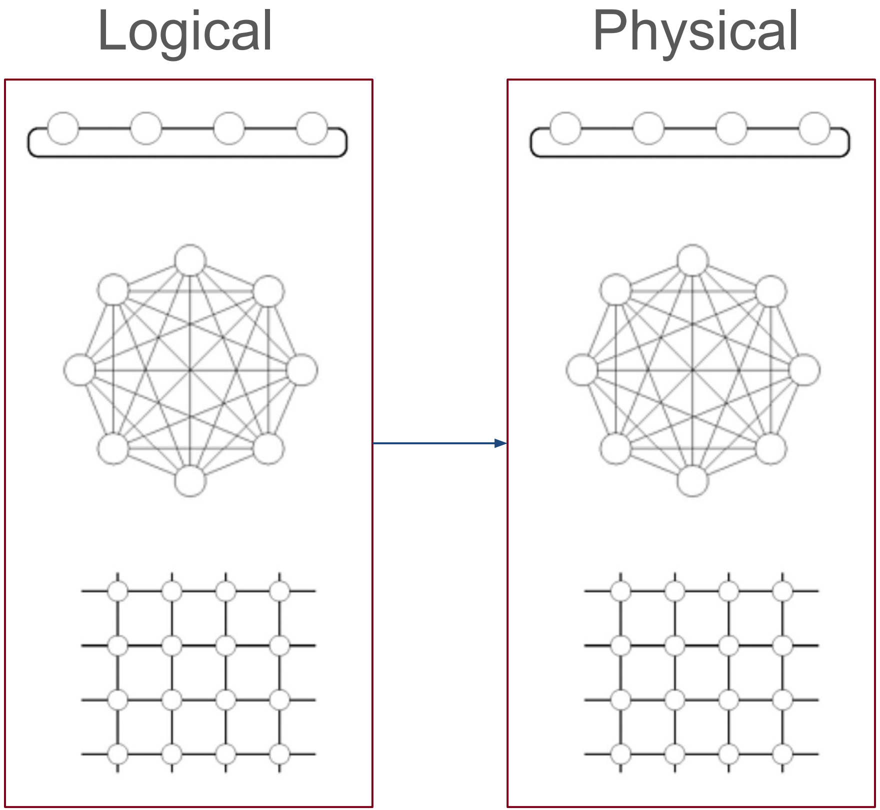






Logical to Physical

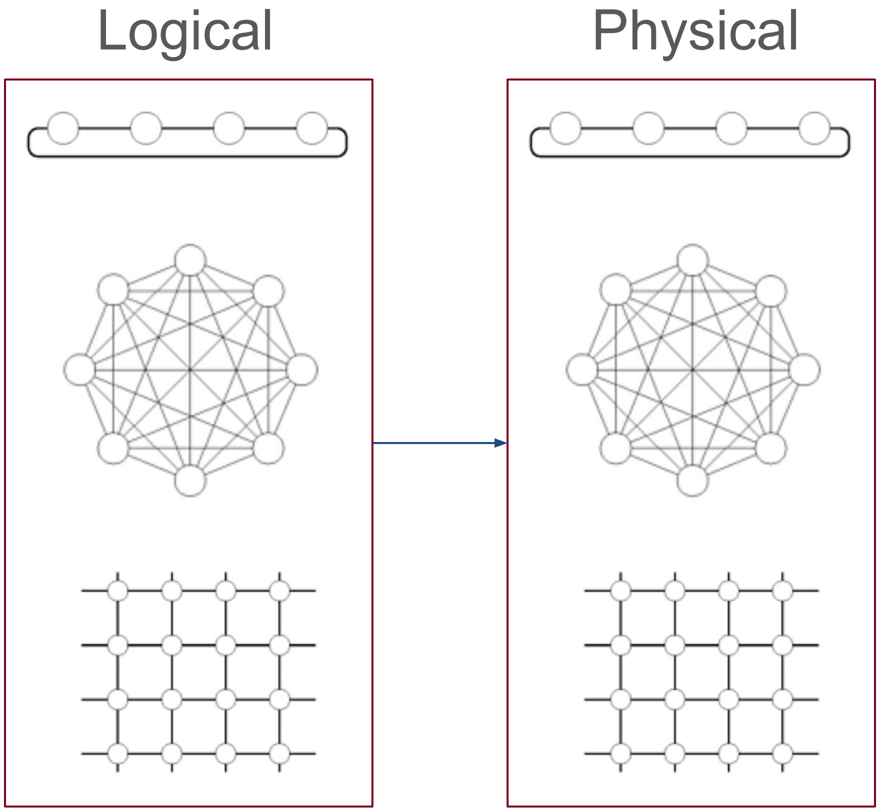
- Once we have this *logical* organization of processes, we have to map them to our physical hardware
- Setting the logical organization of processes to be equivalent to the hardware organization usually helps when programming
- In other words, we want the logical topology equivalent to the physical topology



Logical to Physical

How do we define the goodness of a mapping?
What will this mapping look like in greater detail?

- Once we have this *logical* organization of processes, we have to map them to our physical hardware
- Setting the logical organization of processes to be equivalent to the hardware organization usually helps when programming
- In other words, we want the logical topology equivalent to the physical topology

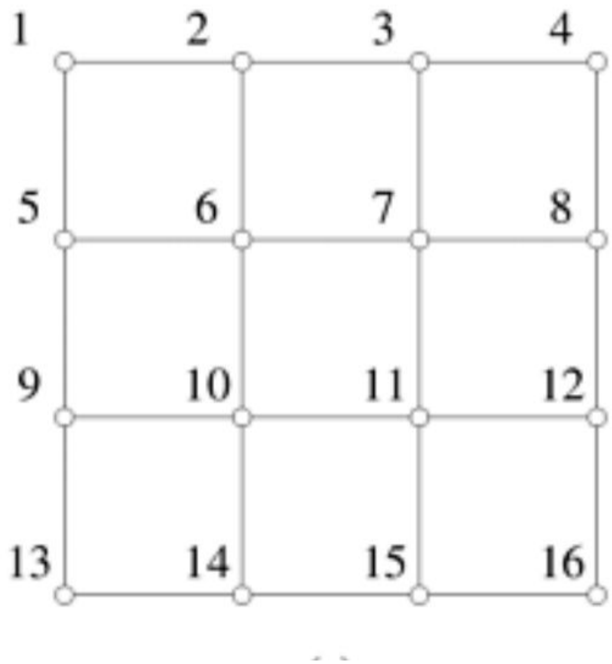


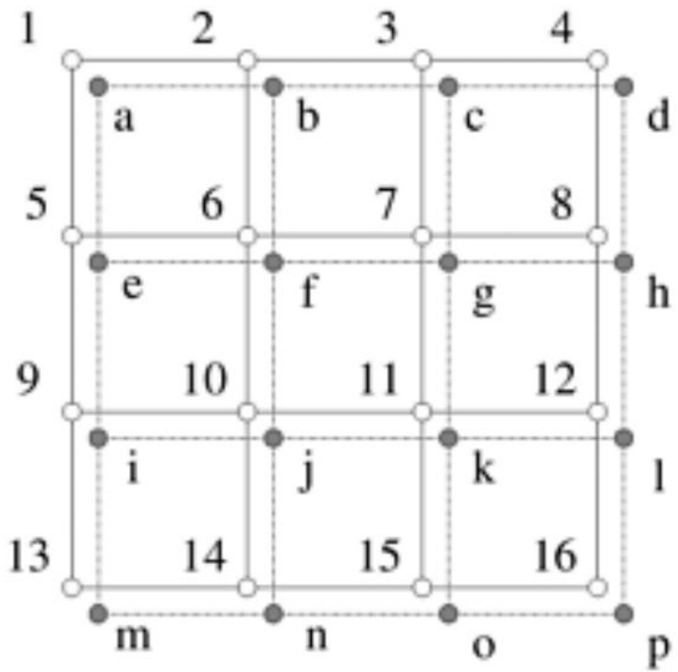
Mapping Metrics

- Congestion
  - How many links in the logical process topology are mapped onto the same link in the physical processor topology.
  - Defines how many communications may wait to use the same link
- Dilation
  - What is the longest number of links a communication must hop in the physical processor topology that took only one hop in the logical process topology.
  - Defines how much ***farther*** communication is in the worst case

Process - Processor Mappings

Good Mapping

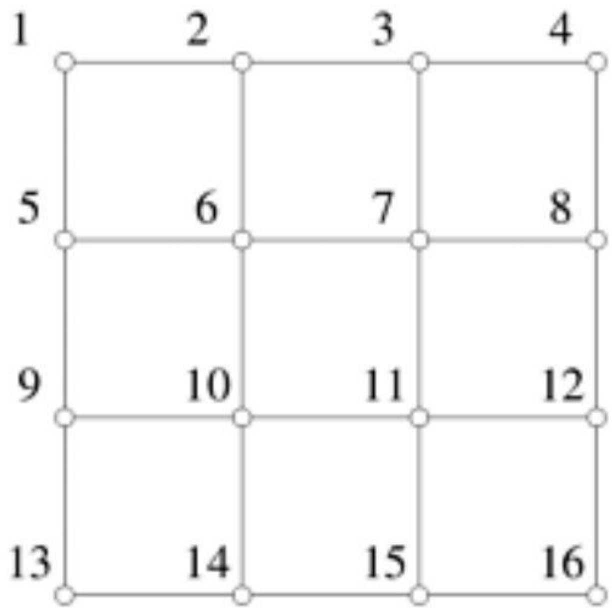


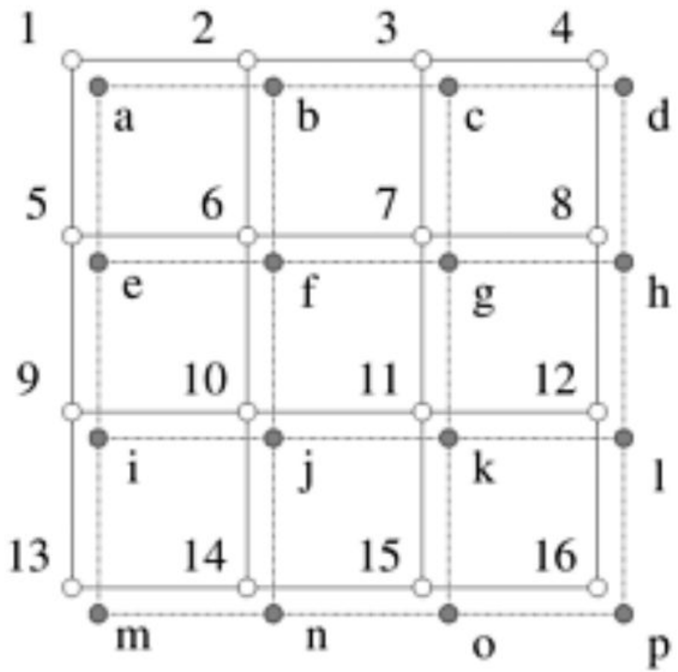


Process - Processor Mappings

Congestion?
Dilation?

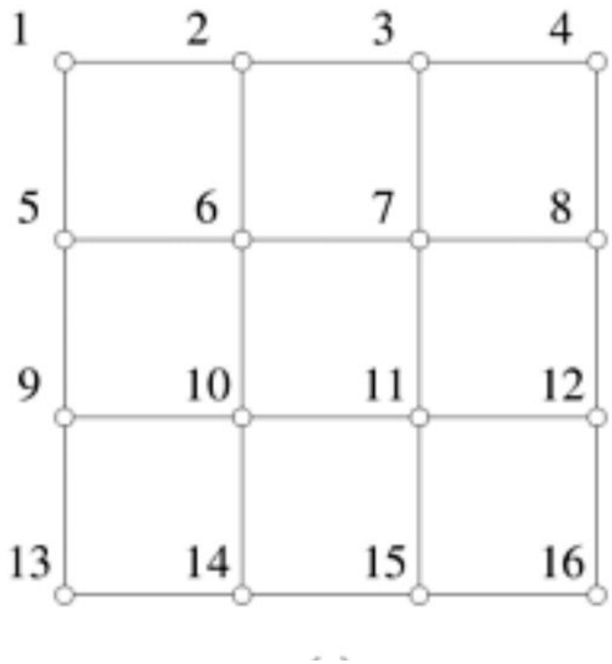
Good Mapping

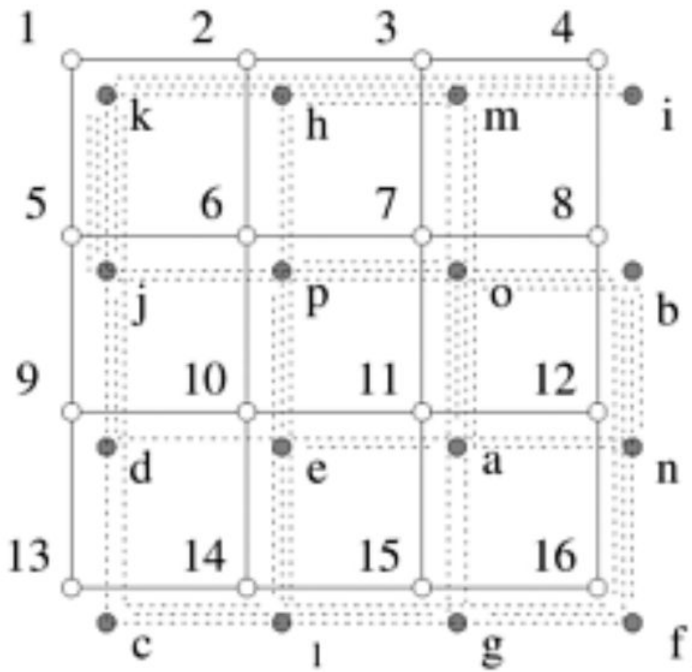




Process - Processor Mappings

Bad Mapping

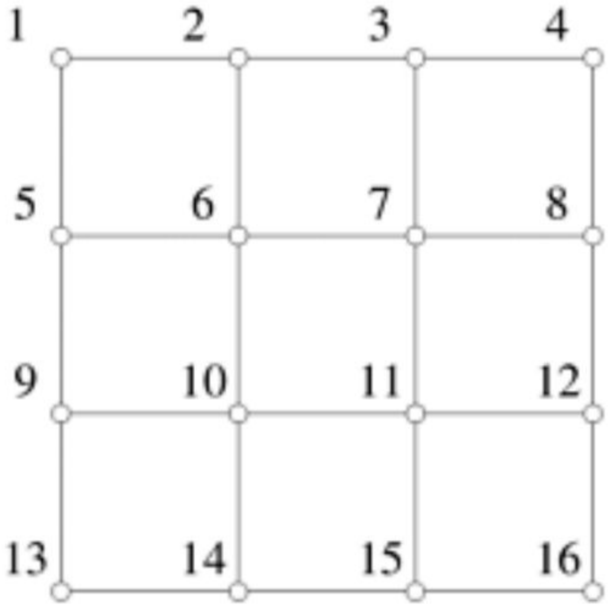


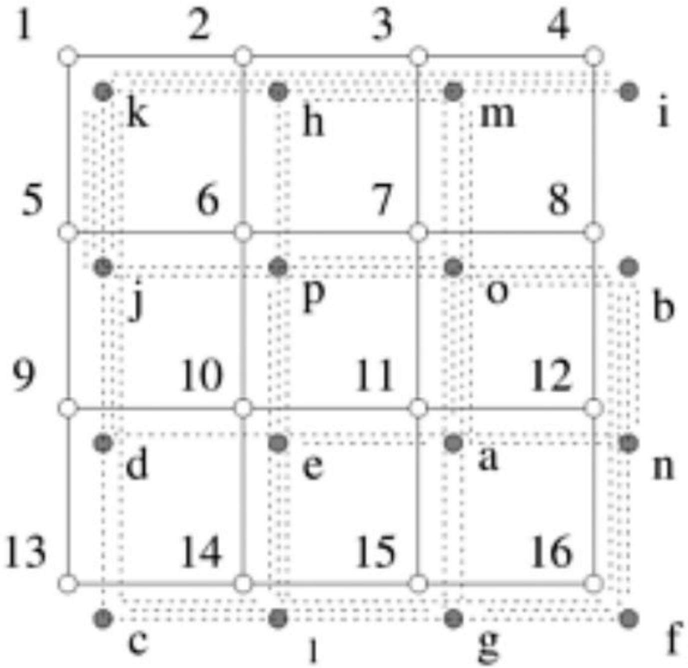


Process - Processor Mappings

Congestion?
Dilation?

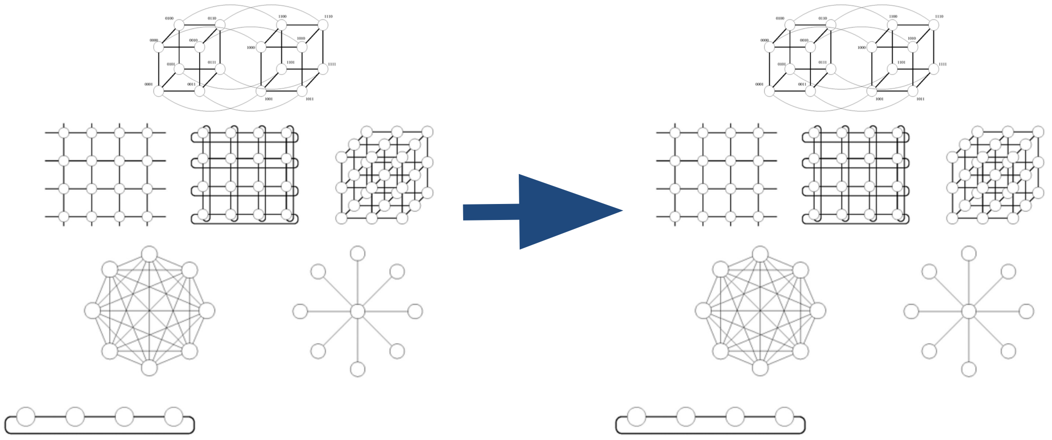
Bad Mapping





Mapping Between different graphs

- Sometimes, the logical process topology is different from the physical processor topology
- This can lead to performance issues downstream
- Usually, we want to make sure they are they same
- Sometimes, we do not have control of the hardware, or do not want to re-design the algorithm from scratch



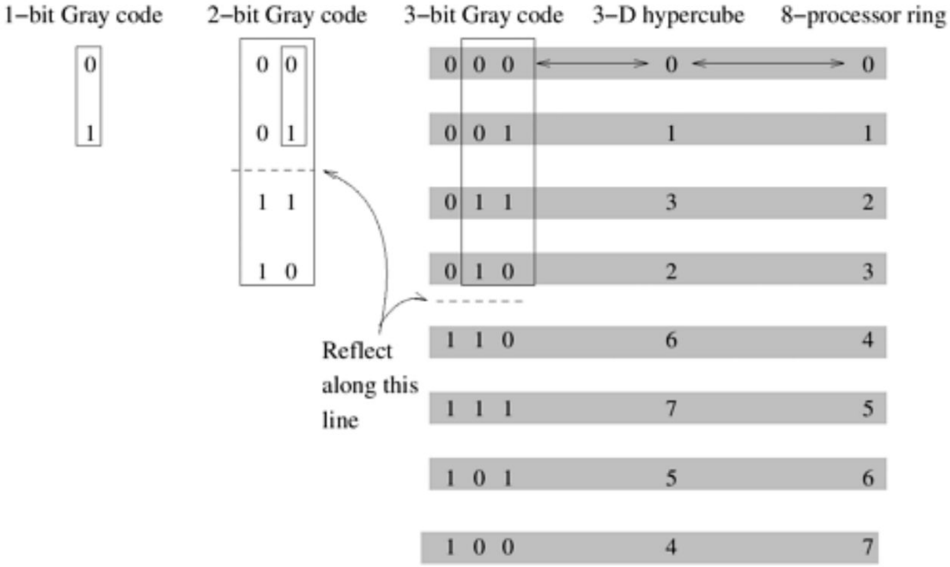
Linear Array to a Hypercube

How can we map effectively from a linear array (w/o wraparound) onto a hypercube so that we limit congestion & dilation?

Linear Array to a Hypercube

How can we map effectively from a linear array (w/o wraparound) onto a hypercube so that we limit congestion & dilation?

- Gray Codes

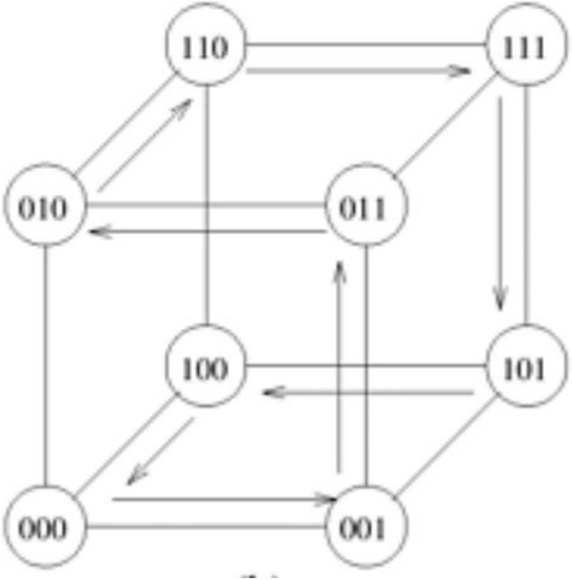


$$
\begin{aligned}
G(0,1) &= 0\\
G(1,1) &= 1\\
G(i,x+1) &= \begin{cases} G(i,x), & i < 2^x \\ 2^x + G(2^{x+1}-1-i, x), & i \geq 2^x \end{cases}
\end{aligned}
$$

Linear Array to a Hypercube

How can we map effectively from a linear array (w/o wraparound) onto a hypercube so that we limit congestion & dilation?

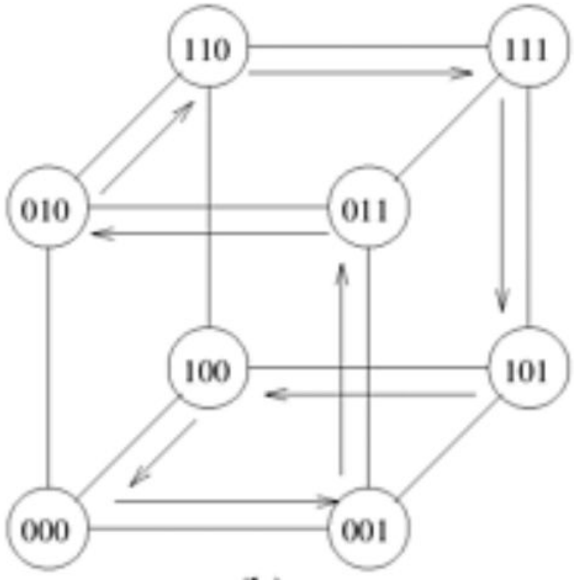
- Gray Codes



Linear Array to a Hypercube

How can we map effectively from a linear array (w/o wraparound) onto a hypercube so that we limit congestion & dilation?

- Gray Codes



Dilation? Congestion?

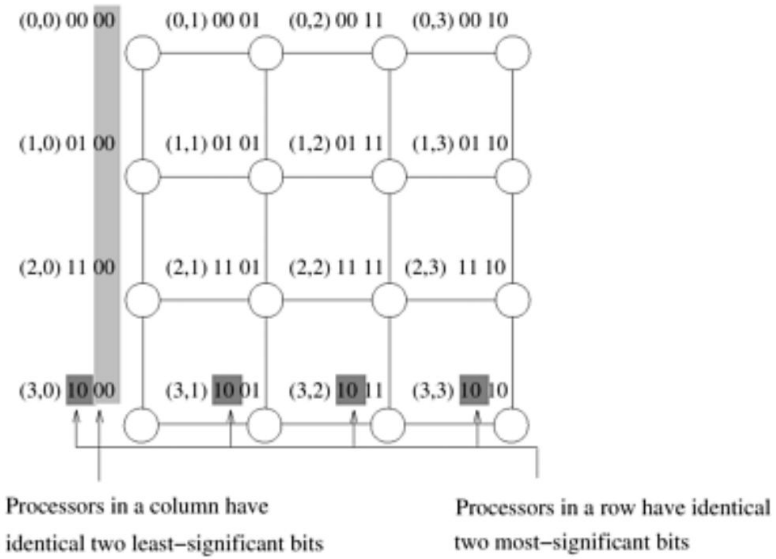
Mesh to a Hypercube

How can we extend these gray codes so that we are able to map from a 2-d mesh to a hypercube?

Mesh to a Hypercube

How can we extend these gray codes so that we are able to map from a 2-d mesh to a hypercube?

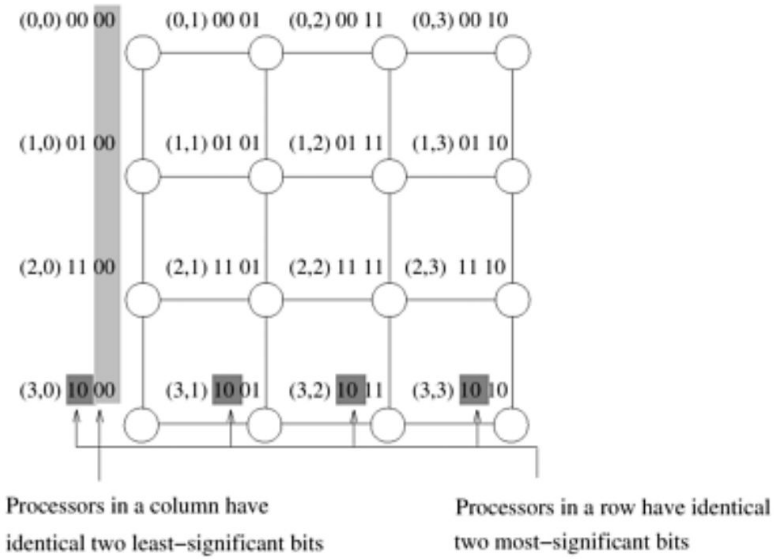
- Use gray codes along each dimension separately.

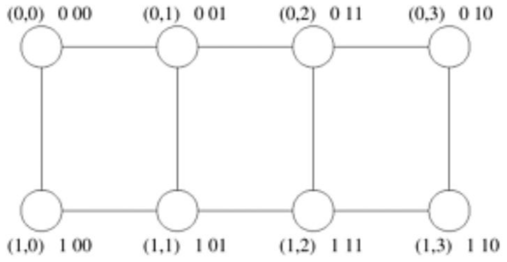


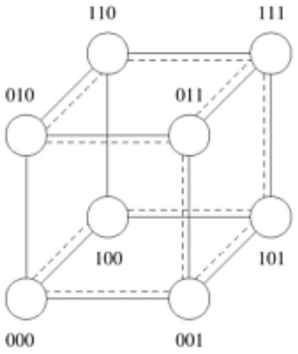
Mesh to a Hypercube

How can we extend these gray codes so that we are able to map from a 2-d mesh to a hypercube?

- Use gray codes along each dimension separately.



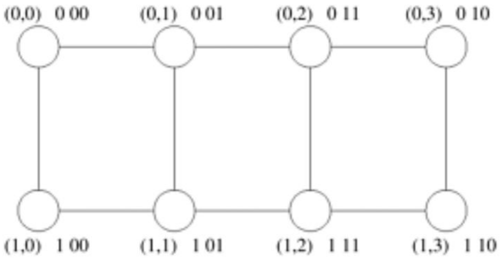


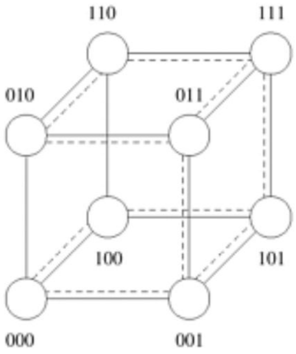


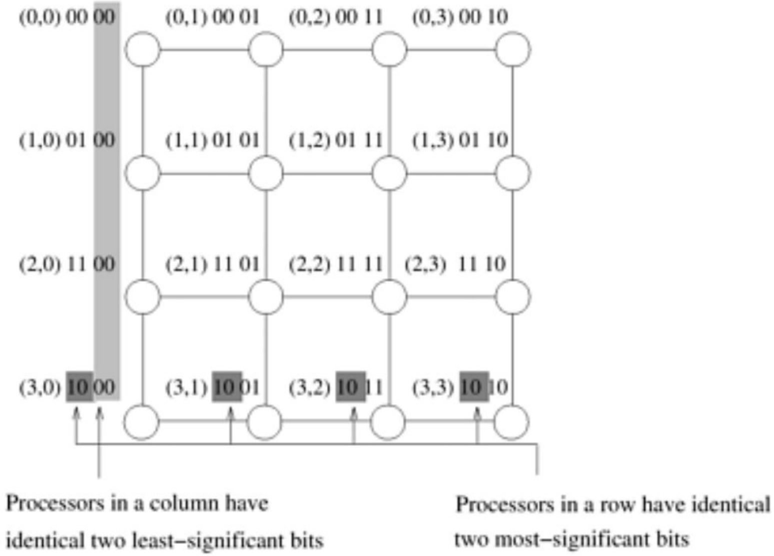
Mesh to a Hypercube

How can we extend these gray codes so that we are able to map from a 2-d mesh to a hypercube?

- Use gray codes along each dimension separately.







Dilation? Congestion?

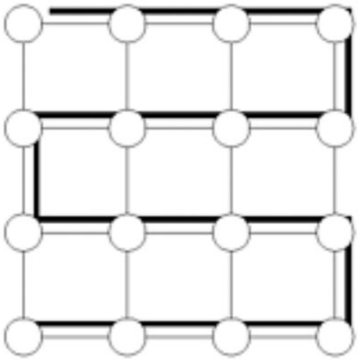
Why have we been able to keep the congestion and dilation at 1 in the previous two examples?

Why have we been able to keep the congestion and dilation at 1 in the previous two examples?

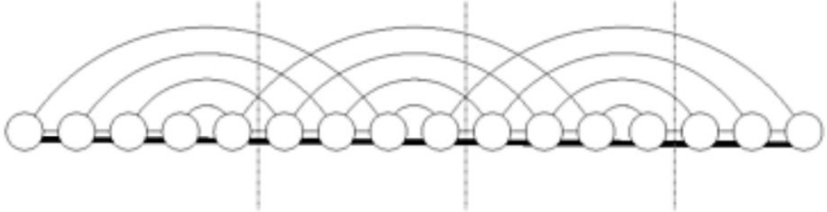
We are mapping sparser
networks onto denser networks.

Mesh to Linear Array

If we go in the other direction (denser to sparser networks), we end up with larger values for dilation + congestion



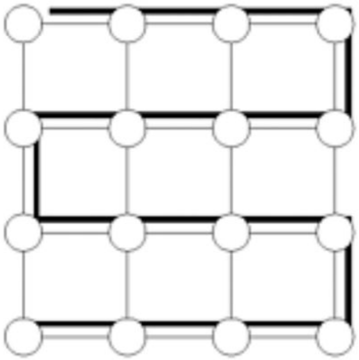
(a) Mapping a linear array into a 2D mesh (congestion 1).



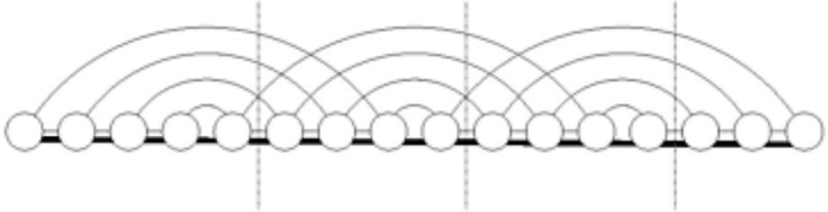
Mesh to Linear Array

If we go in the other direction (denser to sparser networks), we end up with larger values for dilation + congestion

Dilation? Congestion?

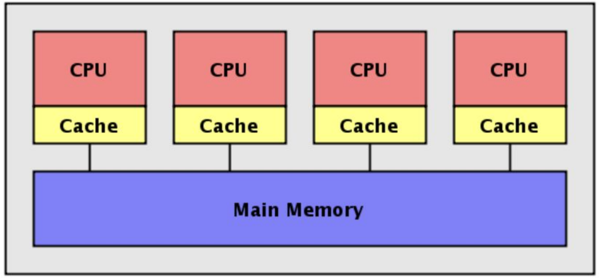


(a) Mapping a linear array into a 2D mesh (congestion 1).



Processor mapping on Shared-Address Space?

Is it necessary to determine how logical process topology maps onto physical processor topology when considering a shared-address space machine?

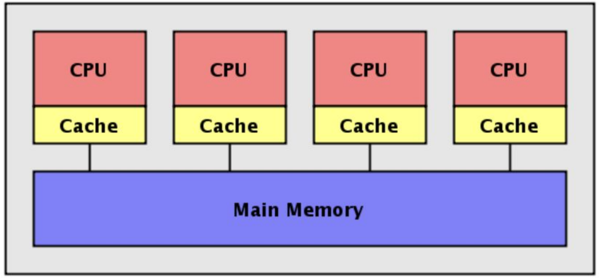


Source: Kaminsky/Parallel Java

Processor mapping on Shared-Address Space?

Is it necessary to determine how logical process topology maps onto physical processor topology when considering a shared-address space machine?

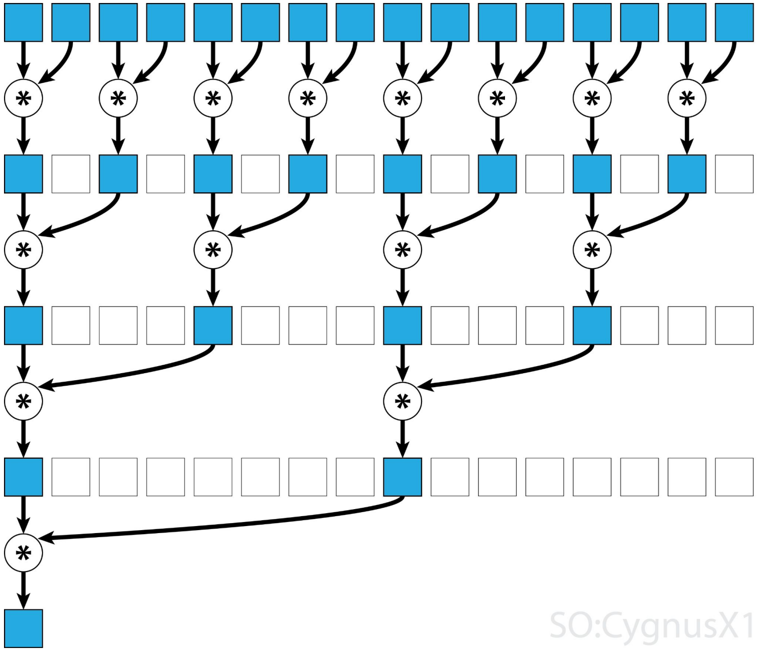
- No. Shared Address machines do not (typically) have any topology associated with them. This step is usually more of a requirement when in the distributed memory setting.



Source: Kaminsky/Parallel Java

End-to-End Example (Array Reduction)

- Sum array of 128 elements
- Assume we have 8 physical processors
- Assume our hardware is a linear array (w/o wraparound) + only one communication can occur on each link at a given time (but multiple communications can occur concurrently on different links)
- Full Pipeline
  - Define tasks
  - Define Task Decomposition Graph
  - Split tasks into processes
  - Define logical organization of process communications
  - Map processes onto processors

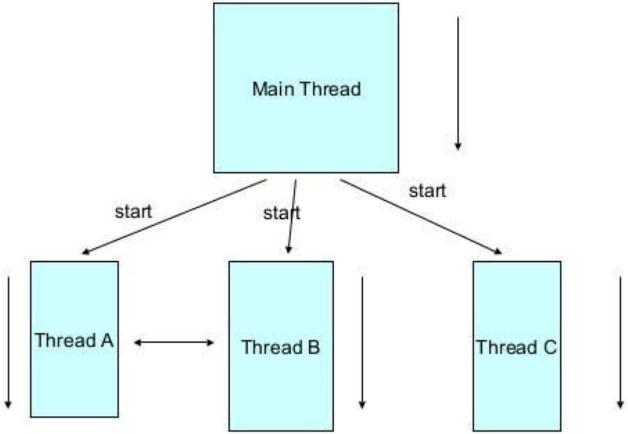


Lecture Overview

- Recap
- Wrap Up Mapping (cont'd)
  - Minimizing Interactions
  - Processes to processors
- **Threads**
  - **Background**
  - Pi Computation Example
  - Threading in Detail

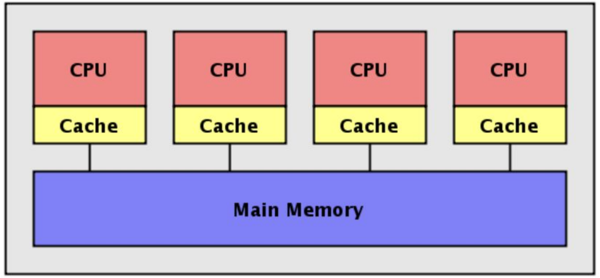
Threads Basics

- **Independent execution units** – Threads are lightweight entities that can run simultaneously on separate CPU cores.
- **Enabling parallelism** – By distributing threads across cores, programs can execute multiple tasks at the same time
- **Shared memory with coordination** – Threads share the same process memory, requiring synchronization tools (e.g., locks, barriers) to avoid race conditions and ensure correctness.



Threads are Shared Memory Processes

- Threads (in our context) share some global memory
- There is no need to explicitly move data between threads - it is shared in some global location among all threads
- We ***do not*** use the operating system definition of a process here - threads are better understood as shared memory processes



Source: Kaminsky/Parallel Java

Matrix Multiplication with Threads

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

How might we parallelize this using some form of threads?

Matrix Multiplication with Threads

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

Parallelize the inner for loop

Matrix Multiplication with Threads

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(dot_product(get_row(a, row),
                                      get_col(b, col)));
```

Matrix Multiplication with Threads

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

Each call to *create_thread* will result in a new thread running in parallel (we do not block on each *create_thread* call)

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(dot_product(get_row(a, row),
                                      get_col(b, col)));
```

Matrix Multiplication with Threads

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

What kind of decomposition have we used here (data, recursive, speculative, etc)?

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(dot_product(get_row(a, row),
                                      get_col(b, col)));
```

Matrix Multiplication with Threads

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

It is important to note that this is not the actual threads API (examples of this will be in the upcoming slides).

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(dot_product(get_row(a, row),
                                      get_col(b, col)));
```

Matrix Multiplication with Threads

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(dot_product(get_row(a, row),
                                      get_col(b, col)));
```

What is a potential issue with this (think back to one of the issues we discussed regarding shared-memory architectures)?

Matrix Multiplication with Threads

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(dot_product(get_row(a, row),
                                      get_col(b, col)));
```

What is a potential issue with this (think back to one of the issues we discussed regarding shared-memory architectures)?

False Sharing

Why Use Threading?

- Portability - Threads can work in both parallel & serial settings. They can further be used across different architectures with minimal code changes.
- Latency Hiding - Some threads can compute while others wait on high-latency operations (I/O, memory access, communications)
- Scheduling/Load Balancing - Useful for dynamic mapping settings.
- Ease of use - Much easier to write programs for than many message passing programs. Wide use + development also encourages faster start-up time for new programmers. Easy to build on top of to create new libraries.

Lecture Concluded Here. Remaining Elements will be covered in the lecture on Wednesday.

Creating a Thread (API)

- We create a thread using *pthread_create*
- The arguments for this function are
  - *thread_handle* → Pointer to thread being created
  - *attribute* → sets thread attributes such as stack size, priority, etc. Typically, you can just use NULL for default settings
  - *thread_function* → The function we want each thread to run. Its signature should be (void *) thread_function(void *)
  - arg → A pointer to the argument passed to *thread_function* (more complex arguments are typically wrapped in a struct)
- The function returns 0 if the thread completes successfully, otherwise an error code (nonzero)

```
#include <pthread.h>
int
pthread_create (
    pthread_t    *thread_handle,
    const pthread_attr_t    *attribute,
    void *    (*thread_function)(void *),
    void    *arg);
```

Exiting a Thread (API)

- We exit a thread with *pthread_exit*
- The arguments for this function are
  - *retval* → A pointer to the value we want to return to the callee thread
- No return value

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Waiting for Threads to Complete (API)

- We wait for threads on the main thread using *pthread_join*
- The arguments for this function are
  - *thread* → The thread we are waiting on to complete
  - *ptr* → The value returned by the thread upon exiting
- Returns 0 when there is no error code, a nonzero value otherwise

```
int
pthread_join (
     pthread_t thread,
     void **ptr);
```

Simple Example

```
#include <stdio.h>
#include <pthread.h>
#include <stdint.h>   // for intptr_t

void* worker(void* arg) {
    int error_code = 42;  // nonzero value
    pthread_exit((void*)(intptr_t)error_code);
}

int main() {
    pthread_t thread;
    void* retval;

    pthread_create(&thread, NULL, worker, NULL);
    pthread_join(thread, &retval);

    int result = (int)(intptr_t)retval;  // retrieve the integer
    printf("Thread returned: %d\n", result);  // prints 42

    return 0;
}
```

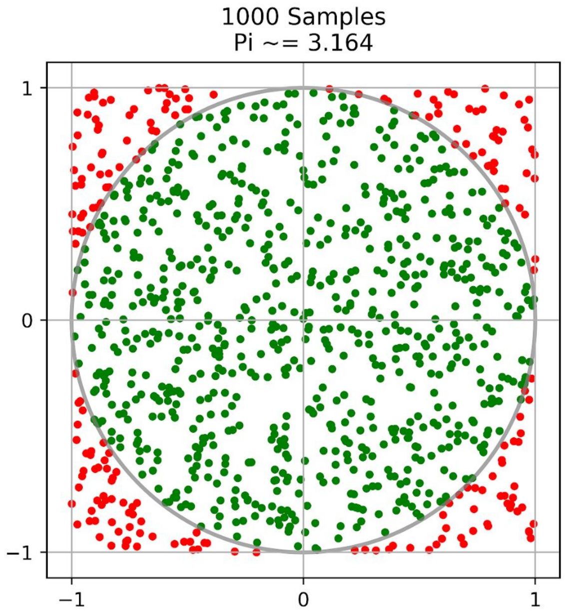
Lecture Overview

- Recap
- Wrap Up Mapping (cont'd)
  - Minimizing Interactions
  - Processes to processors
- **Threads**
  - Background
  - **Pi Computation Example**
  - Threading in Detail

Computing Pi with Monte Carlo Estimation

- We randomly sample x, y points in [-1,1]x[-1,1] (in our program, we use [0,1]x[0,1], but the principle is the same)
- If a point is less than a distance of 1 from [0,0], then we call it a hit (green),
- If it is greater than a distance of 1 from [0,0] we call it a miss (red)
- Pi is then approximately equal to

hits/(hits+misses)



Computing Pi with Monte Carlo Estimation

```c
#include <pthread.h>
#include <stdlib.h>

#define MAX_THREADS 512
void *compute_pi (void *);

int total_hits, total_misses, hits[MAX_THREADS],
    sample_points, sample_points_per_thread, num_threads;

main() {
    int i;
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    double computed_pi;
    double time_start, time_end;
    struct timeval tv;
    struct timezone tz;

    pthread_attr_init (&attr);
    pthread_attr_setscope (&attr,PTHREAD_SCOPE_SYSTEM);
    printf("Enter number of sample points: ");
    scanf("%d", &sample_points);
    printf("Enter number of threads: ");
    scanf("%d", &num_threads);

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
                 (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Computing Pi with Monte Carlo Estimation

```
#include <pthread.h>
#include <stdlib.h>

#define MAX_THREADS 512
void *compute_pi (void *);

int total_hits, total_misses, hits[MAX_THREADS],
    sample_points, sample_points_per_thread, num_threads;

main() {
    int i;
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    double computed_pi;
    double time_start, time_end;
    struct timeval tv;
    struct timezone tz;

    pthread_attr_init (&attr);
    pthread_attr_setscope (&attr,PTHREAD_SCOPE_SYSTEM);
    printf("Enter number of sample points: ");
    scanf("%d", &sample_points);
    printf("Enter number of threads: ");
```

In this example, an attributes object is created that enables threads to compete with all other threads in the system. Helpful for ensuring each CPU core is used. Is the default on most Linux/Unix systems.

Computing Pi with Monte Carlo Estimation

We create *num_threads* threads, pass in the attribute variable, a reference to the *compute_pi* function, and the location of the *hits* array we want this thread to update

```
    scanf("%d", &num_threads);

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
                 (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
         pthread_create(&p_threads[i], &attr, compute_pi,
             (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
         ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
         rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
         rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Computing Pi with Monte Carlo Estimation

Each thread samples *sample_points_per_thread*, determining the total number of *local_hits*.
Crucially, this is done in parallel.

```
    scanf("%d", &num_threads);

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
                 (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

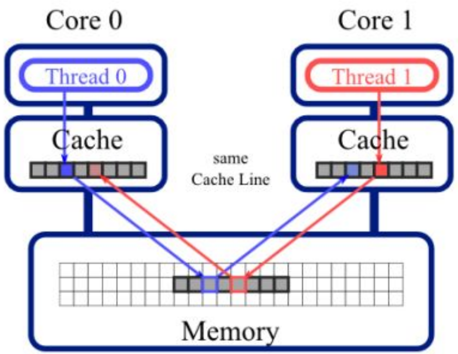
Computing Pi with Monte Carlo Estimation

Why are we not updating the *hit_pointer* directly (i.e. why do we create a local *local_hits* variable)?

```
    scanf("%d", &num_threads);

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
                 (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Computing Pi with Monte Carlo Estimation

False Sharing




```c
    scanf("%d", &num_threads);

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
                 (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
         pthread_create(&p_threads[i], &attr, compute_pi,
             (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
         pthread_join(p_threads[i], NULL);
         total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
         ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
         rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
         rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
             (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Computing Pi with Monte Carlo Estimation

We make sure to set the *hit_pointer* to the *local_hits* computed on this thread.

```
    scanf("%d", &num_threads);

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
                 (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Computing Pi with Monte Carlo Estimation

The main thread waits for all of the threads to complete. Once they are complete, the *total_hits* variable accumulates the value in each thread.

```
    scanf("%d", &num_threads);

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
                 (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Computing Pi with Monte Carlo Estimation

Once all other threads have completed execution, the main thread has the complete number of hits (*total_hits*) and can finish the estimation of pi.

```
    scanf("%d", &num_threads);

    gettimeofday(&tv, &tz);
    time_start = (double)tv.tv_sec +
                 (double)tv.tv_usec / 1000000.0;

    total_hits = 0;
    sample_points_per_thread = sample_points / num_threads;
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    computed_pi = 4.0*(double) total_hits /
        ((double)(sample_points));
    gettimeofday(&tv, &tz);
    time_end = (double)tv.tv_sec +
               (double)tv.tv_usec / 1000000.0;

    printf("Computed PI = %lf\n", computed_pi);
    printf(" %lf\n", time_end - time_start);
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Lecture Overview

- Recap
- Wrap Up Mapping (cont'd)
  - Minimizing Interactions
  - Processes to processors
- **Threads**
  - Background
  - Pi Computation Example
  - **Threading in Detail**

Race Conditions in Threads

- ❑ Sometimes concurrent threads may issue updates to the same variable
- ❑ In general, if the outcome of a multithreaded program depends on the order of execution of the threads, then we say the program has a race condition

```c
#include <stdio.h>
#include <pthread.h>

int counter = 0;  // shared global variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // <-- race condition occurs here
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // create two threads that increment the same counter
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

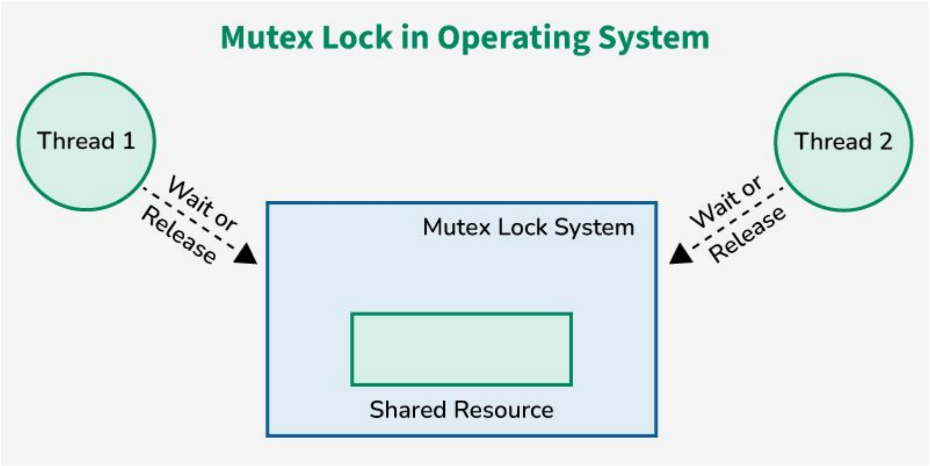
Race Conditions in Threads

How do we resolve this?

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;  // shared global variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // <-- race condition occurs here
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // create two threads that increment the same counter
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Race Conditions in Threads

How do we resolve this?




```c
#include <stdio.h>
#include <pthread.h>

int counter = 0;  // shared global variable

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // <-- race condition occurs here
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // create two threads that increment the same counter
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

Mutex APIs

- For all three functions below:
  - Returns 0 if successful, nonzero for error code
  - Pointer to the lock (**mutex_lock)* for intended use as first argument
- *int pthread_mutex_lock(pthread_mutex_t *mutex_lock)*
  - Locks thread. If the lock is already held by another thread, then this call blocks.
- *Int pthread_mutex_unlock(pthread_mutex_t *mutex_lock)*
  - Unlocks thread. Allows other threads to lock the thread after waiting.
- *Int pthread_mutex_init(pthread_mutex_t *mutex_lock, const pthread_mutexattr_t *lock_attr)*
  - Initializes the lock to unlocked state
  - Allows attributes to alter the default state of the lock
- *Int pthread_mutex_destroy(pthread_mutex_init(pthread_mutex_t *mutex_lock)*
  - Destroys the given mutex lock and clears up memory usage by the lock

Mutex Example (Counter)

We alter the earlier example to use mutexes and resolve the race condition

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;                    // shared global variable
pthread_mutex_t counter_mutex;    // mutex to protect counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&counter_mutex);  // lock before accessing counter
        counter++;
        pthread_mutex_unlock(&counter_mutex); // unlock after updating
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // initialize the mutex
    pthread_mutex_init(&counter_mutex, NULL);

    // create two threads
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);  // should always be 200000

    // destroy the mutex
    pthread_mutex_destroy(&counter_mutex);

    return 0;
}
```

Mutex Example (Counter)

Is this a good way to parallelize this program with mutexes?

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;                    // shared global variable
pthread_mutex_t counter_mutex;     // mutex to protect counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&counter_mutex);  // lock before accessing counter
        counter++;
        pthread_mutex_unlock(&counter_mutex); // unlock after updating
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // initialize the mutex
    pthread_mutex_init(&counter_mutex, NULL);

    // create two threads
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);  // should always be 200000

    // destroy the mutex
    pthread_mutex_destroy(&counter_mutex);

    return 0;
}
```

Mutex Example (Counter)

Is this a good way to parallelize this program with mutexes?

No

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;                    // shared global variable
pthread_mutex_t counter_mutex;    // mutex to protect counter

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&counter_mutex);  // lock before accessing counter
        counter++;
        pthread_mutex_unlock(&counter_mutex); // unlock after updating
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // initialize the mutex
    pthread_mutex_init(&counter_mutex, NULL);

    // create two threads
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final counter value: %d\n", counter);  // should always be 200000

    // destroy the mutex
    pthread_mutex_destroy(&counter_mutex);

    return 0;
}
```

Mutex Example (Counter)

Better Counter Program

Perform updates on a local counter first, then sum together into global counter

```
// We ignore headers for visualization purposes
int counter = 0;                 // shared global variable
pthread_mutex_t counter_mutex;   // mutex to protect counter
#define INCREMENTS 100000

void* increment(void* arg) {
    int local_counter = 0;
    for (int i = 0; i < INCREMENTS; i++) {
        local_counter++;
    }
    pthread_mutex_lock(&counter_mutex);
    counter += local_counter;
    pthread_mutex_unlock(&counter_mutex);
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&counter_mutex, NULL);

    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&counter_mutex);

    return 0;
}
```