



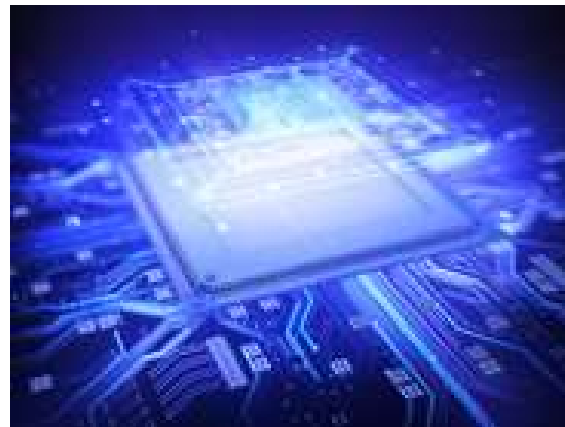
Multicore Processors: Architecture & Programming

Gentle Introduction to Parallel Programming

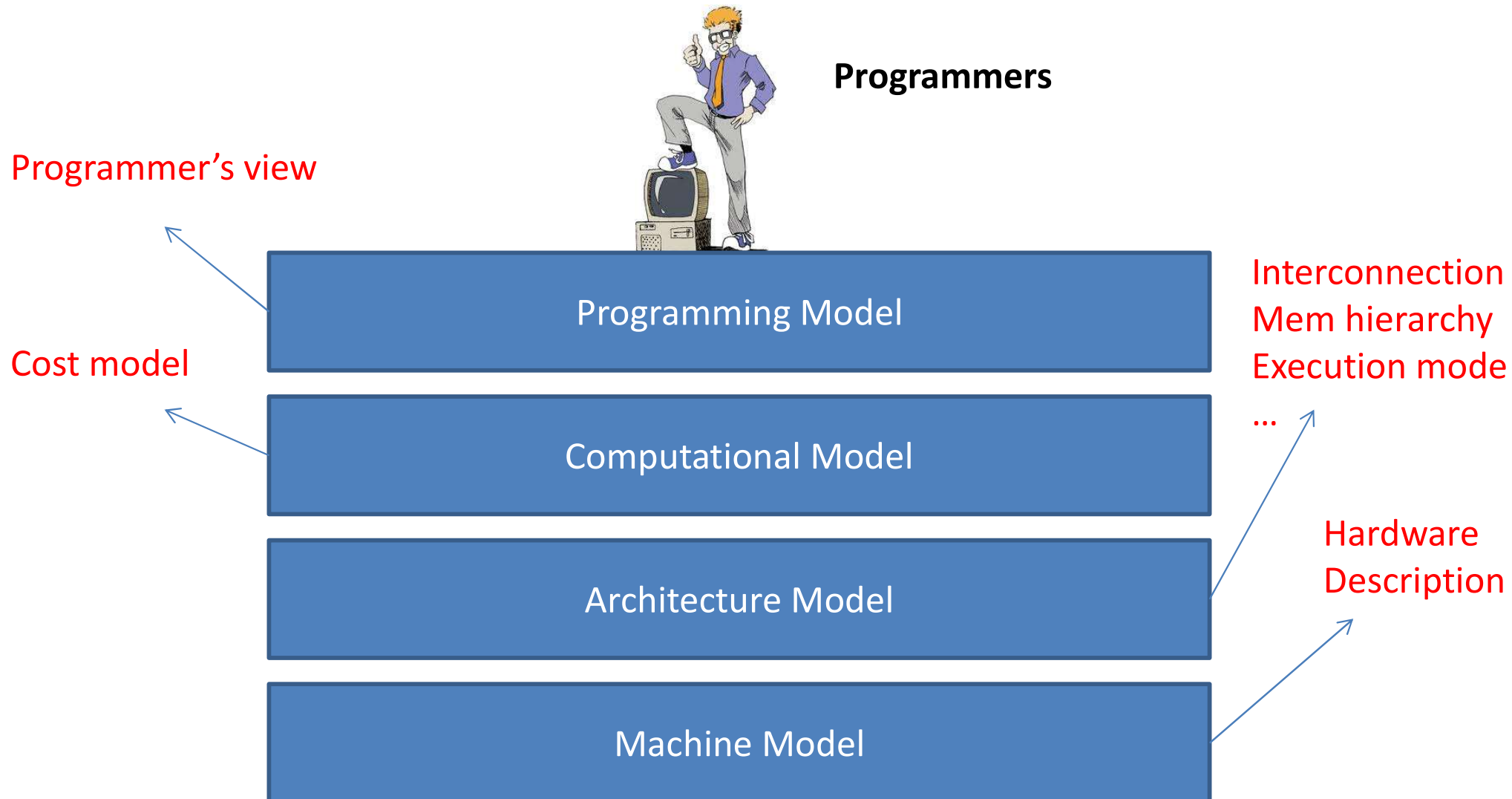
Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

<http://www.mzahran.com>



Models ... Models



Let's Start With A Simple Example


- Compute n values and add them together.
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

Example (cont.)

- We have p cores, p much smaller than n .
- Each core performs a partial sum of approximately n/p values.

```
my_sum = 0;  
my_first_i = . . . ;  
my_last_i = . . . ;  
for (my_i = my_first_i; my_i < my_last_i; my_i++) {  
    my_x = Compute_next_value( . . . );  
    my_sum += my_x;  
}
```



Each core uses its own private variables and executes this block of code independently of the other cores.

Example (cont.)

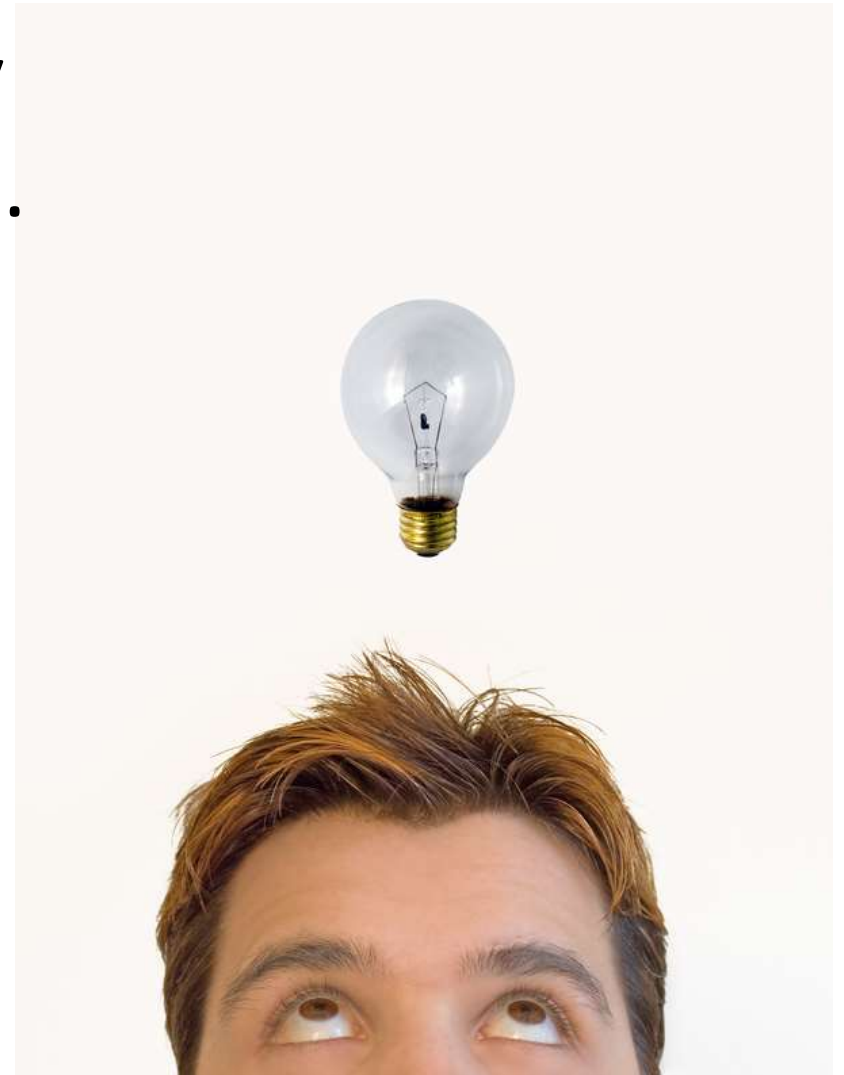
- Once all the cores are done computing their private `my_sum`, they form a global sum by sending results to a designated "master" core which adds the final result.

Example (cont.)

```
if (I'm the master core) {  
    sum = my_x;  
    for each core other than myself {  
        receive value from core;  
        sum += value;  
    }  
} else {  
    send my_x to the master;  
}
```

But wait!

There's a much better way
to compute the global sum.



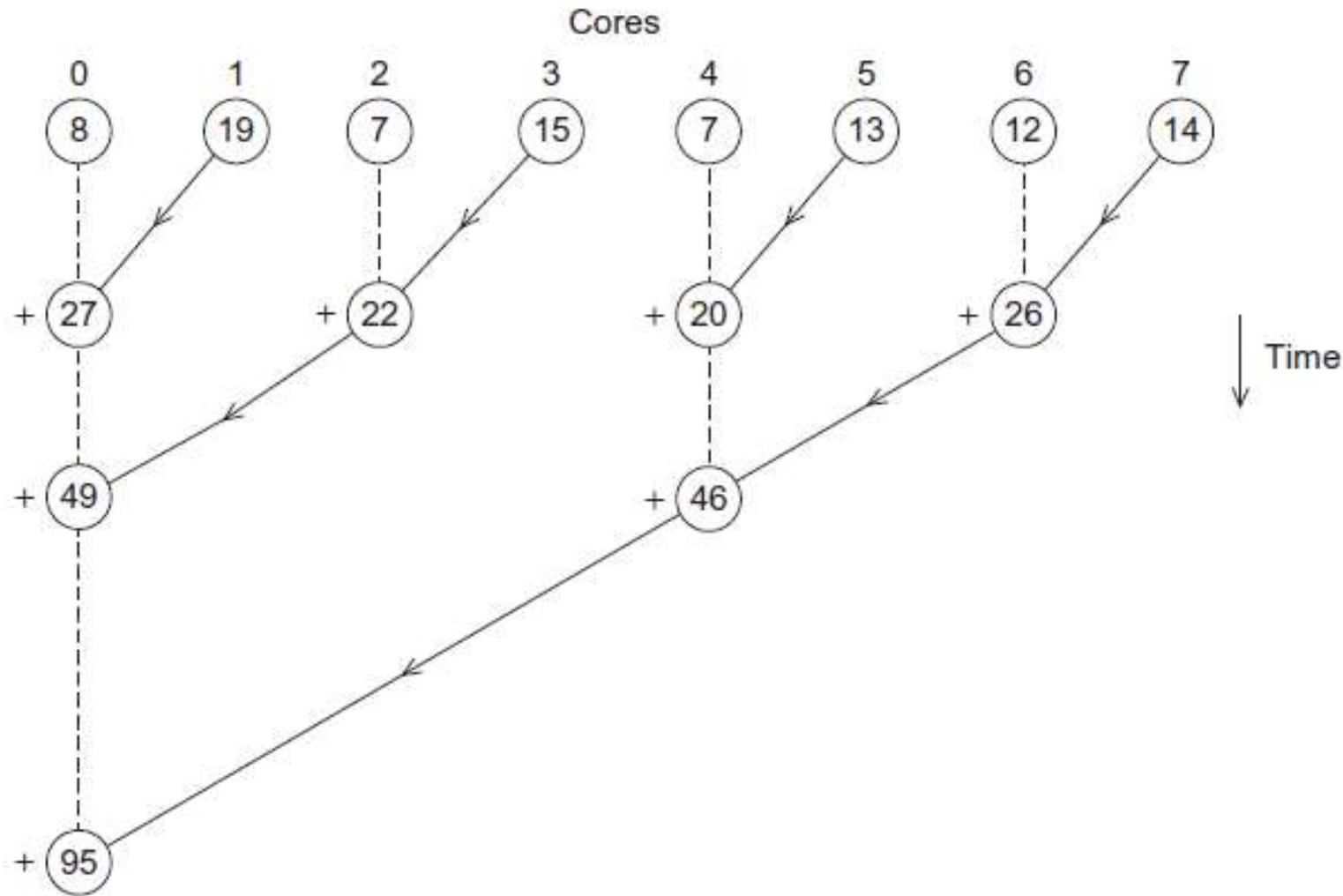
Better parallel algorithm

- Don't make the master core do all the work.
- Share it among the other cores.
- Pair the cores so that:
 - Core 0 adds its result with core 1's result.
 - Core 2 adds its result with core 3's result.
 - etc.

Better parallel algorithm (cont.)

- Repeat the process now with only the evenly ranked cores.
 - Core 0 adds result from core 2.
 - Core 4 adds the result from core 6.
 - etc.
- Now repeat the process with cores divisible by 4, and so forth, until core 0 has the final result.

Multiple cores forming a global sum



Analysis

- In the first example, the master core performs 7 receives and 7 additions.
- In the second example, the master core performs 3 receives and 3 additions.
- The improvement is more than a factor of 2!

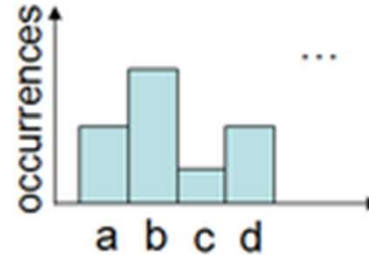
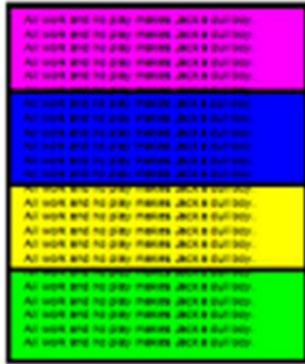
Analysis (cont.)

- The difference is more dramatic with a larger number of cores.
- If we have 1000 cores:
 - The first example would require the master to perform 999 receives and 999 additions.
 - The second example would only require 10 receives and 10 additions.
- That's an improvement of almost a factor of 100!

Another Quick Example

- **Problem:** Count the number of times each ASCII character occurs on a page of text.
- **Input:** ASCII text stored as an array of characters.
- **Output:** A histogram with 128 buckets - one for each ASCII character

Let's See A Quick Example

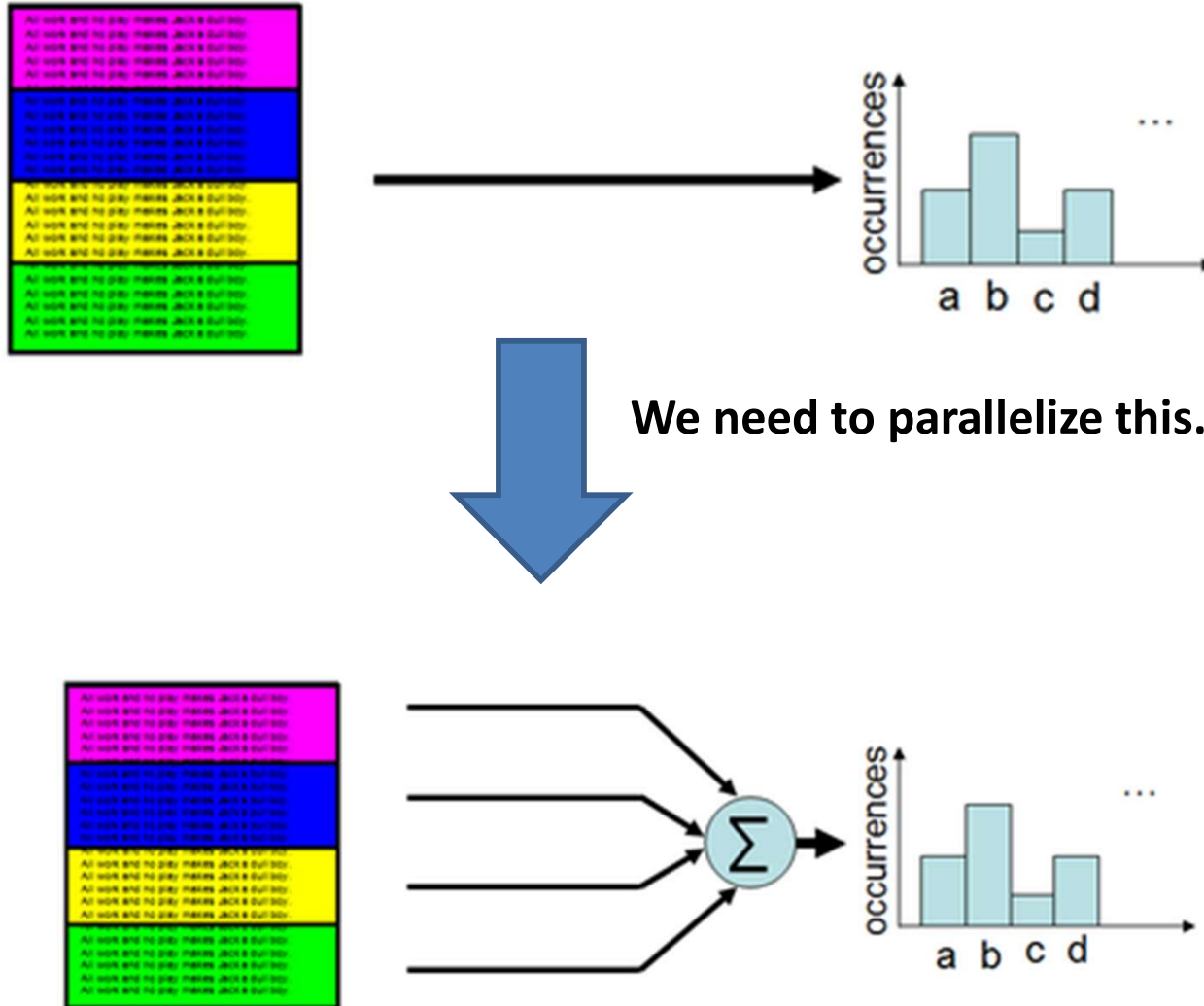


Speed on Quad Core:
10.36 seconds

```
1: void compute_histogram_st(char *page, int page_size, int *histogram){  
2: for(int i = 0; i < page_size; i++){  
3:   char read_character = page[i];  
4:   histogram[read_character]++;  
5: }  
6: }
```

Sequential Version

Let's See A Quick Example



Let's See A Quick Example

```
1: void compute_histogram_st(char *page, int page_size, int *histogram){  
2: #pragma omp parallel for  
3: for(int i = 0; i < page_size; i++){  
4:     char read_character = page[i];  
5:     histogram[read_character]++;  
6: }
```

The above code does not work!! Why?

Let's See A Quick Example

```
1: void compute_histogram_mt2(char *page, int page_size, int *histogram){  
2:  #pragma omp parallel for  
3:  for(int i = 0; i < page_size; i++){  
4:      char read_character = page[i];  
5:      #pragma omp atomic  
6:      histogram[read_character]++;  
7:  }  
8: }
```

Speed on Quad Core:

114.89 seconds

> 10x slower than the single thread version!!

Let's See A Quick Example

```
1: void compute_histogram_mt3(char *page,
                             int page_size,
                             int *histogram, int num_buckets){
2:  #pragma omp parallel
3:  {
4:      int local_histogram[111][num_buckets];
5:      int tid = omp_get_thread_num();
6:      #pragma omp for nowait
7:      for(int i = 0; i < page_size; i++){
8:          char read_character = page[i];
9:          local_histogram[tid][read_character]++;
10:     }
11:     for(int i = 0; i < num_buckets; i++){
12:         #pragma omp atomic
13:         histogram[i] += local_histogram[tid][i];
14:     }
15: }
16: }
```

Runs in 3.8 secs
Why speedup
is not 4 yet?

Let's See A Quick Example

```
void compute_histogram_mt4(char *page, int page_size,  
                           int *histogram, int num_buckets){  
1:   int num_threads = omp_get_max_threads();  
2:   #pragma omp parallel  
3:   {  
4:       __declspec (align(64)) int local_histogram[num_threads+1][num_buckets];  
5:       int tid = omp_get_thread_num();  
6:       #pragma omp for  
7:       for(int i = 0; i < page_size; i++){  
8:           char read_character = page[i];  
9:           local_histogram[tid][read_character]++;  
10:      }  
11:  
12:      #pragma omp single  
13:      for(int t = 0; t < num_threads; t++){  
14:          for(int i = 0; i < num_buckets; i++)  
15:              histogram[i] += local_histogram[t][i];  
16:      }  
17: }
```

Speed is
4.42 seconds.
Slower than the
previous version.

Let's See A Quick Example

```
void compute_histogram_mt4(char *page, int page_size,  
                           int *histogram, int num_buckets){  
1:   int num_threads = omp_get_max_threads();  
2:   #pragma omp parallel  
3:   {  
4:   __declspec (align(64)) int local_histogram[num_threads+1][num_buckets];  
5:   int tid = omp_get_thread_num();  
6:   #pragma omp for  
7:   for(int i = 0; i < page_size; i++){  
8:       char read_character = page[i];  
9:       local_histogram[tid][read_character]++;  
10:  }  
11:  
12:  #pragma omp for  
13:  for(int i = 0; i < num_buckets; i++){  
14:      for(int t = 0; t < num_threads; t++)  
15:          histogram[i] += local_histogram[t][i];  
16:  }  
17: }
```

Speed is
3.60 seconds.

What Can We Learn from the Previous Examples?

- Parallel programming is not only about finding a lot of parallelism.
- Critical section and atomic operations
 - Race condition
 - Again: correctness vs performance loss
- Know your tools: language, compiler and hardware

What Can We Learn from the Previous Examples?

- Atomic operations
 - They are expensive
 - Yet, they are fundamental building blocks.
- Synchronization:
 - correctness vs performance loss
 - Rich interaction of hardware-software tradeoffs
 - Must evaluate hardware primitives and software algorithms together

Sources of Performance Loss in Parallel Programs

- Extra overhead
 - load
 - synchronization
 - communication
- Bigger code than sequential version
- Contention due to hardware resources
- Coherence
- Load imbalance
- Artificial dependence

Artificial Dependencies

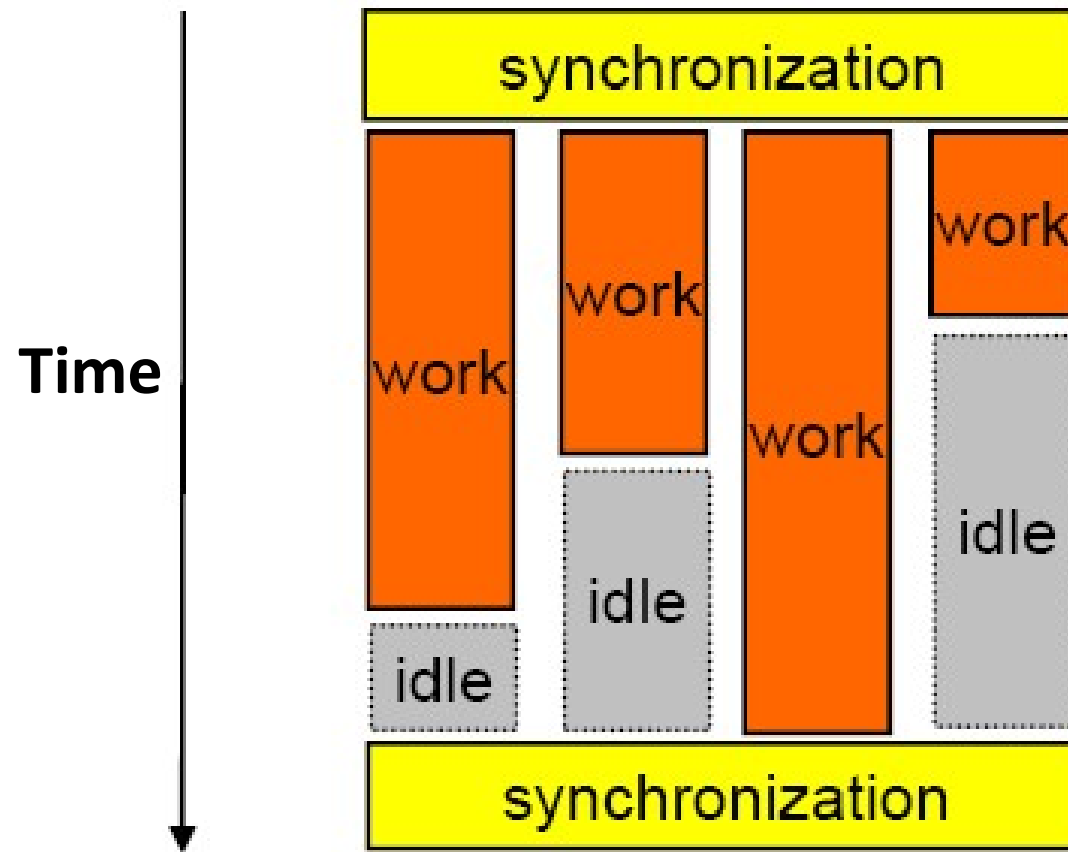
```
int result;  
//Global variable  
  
for (...) // The OUTER loop  
    modify_result(...);  
    if(result > threshold)  
        break;  
  
void modify_result(...)  
    ...  
    result = ...
```

What is wrong with
that program when
we try to parallelize
it?

Coherence

- Extra bandwidth (scarce resource)
- Latency due to the protocol
- False sharing

Load Balancing



Load Balancing

- Assignment of work not data is the key.
- If you cannot eliminate it, at least reduce it.
- Static assignment
- Dynamic assignment
 - Has its overhead

We want to write a parallel program ... Now what?

- We have a serial program.
 - How to parallelize it?
- We know that we need to divide work, ensure load balancing, manage synchronization, and reduce communication! → **Nice! How to do that?**
- Unfortunately: there is no mechanical process.
- **Ian Foster** has some nice framework.
 - His book "Designing and Building Parallel Programs".

Foster's methodology (The PCAM Methodology)

1. **Partitioning**: divide the computation to be performed and the data operated on by the computation into small tasks.

The focus here should be on identifying **tasks that can be executed in parallel**.

This step brings out the parallelism in the algorithm

A checklist for problem partitioning

- Does your partition define at least an order of magnitude more tasks than there are processors/cores in your target computer? If not, you have little flexibility in subsequent design stages.
- Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.
- Are tasks of comparable size? If not, you may face load balancing issues later.
- Does the number of tasks scale with problem size? Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks.
- Have you identified several alternative partitions?

Foster's methodology (The PCAM Methodology)

2. **Communication**: determine what communication needs to be carried out among the tasks identified in the previous step.



A checklist for communication

- Do all tasks perform about the same number of communication operations? Unbalanced communication requirements suggest a nonscalable construct. Revisit your design to see whether communication operations can be distributed more equitably. For example, if a frequently accessed data structure is encapsulated in a single task, consider distributing or replicating this data structure.
- Does each task communicate only with a small number of neighbors? If each task must communicate with many other tasks, evaluate the possibility of formulating this global communication in terms of a local communication structure.
- Are communication operations able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable.

Foster's methodology (The PCAM Methodology)

3. **Agglomeration or aggregation:** combine tasks and communications identified in the first step into larger tasks.

For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.

A checklist for agglomeration

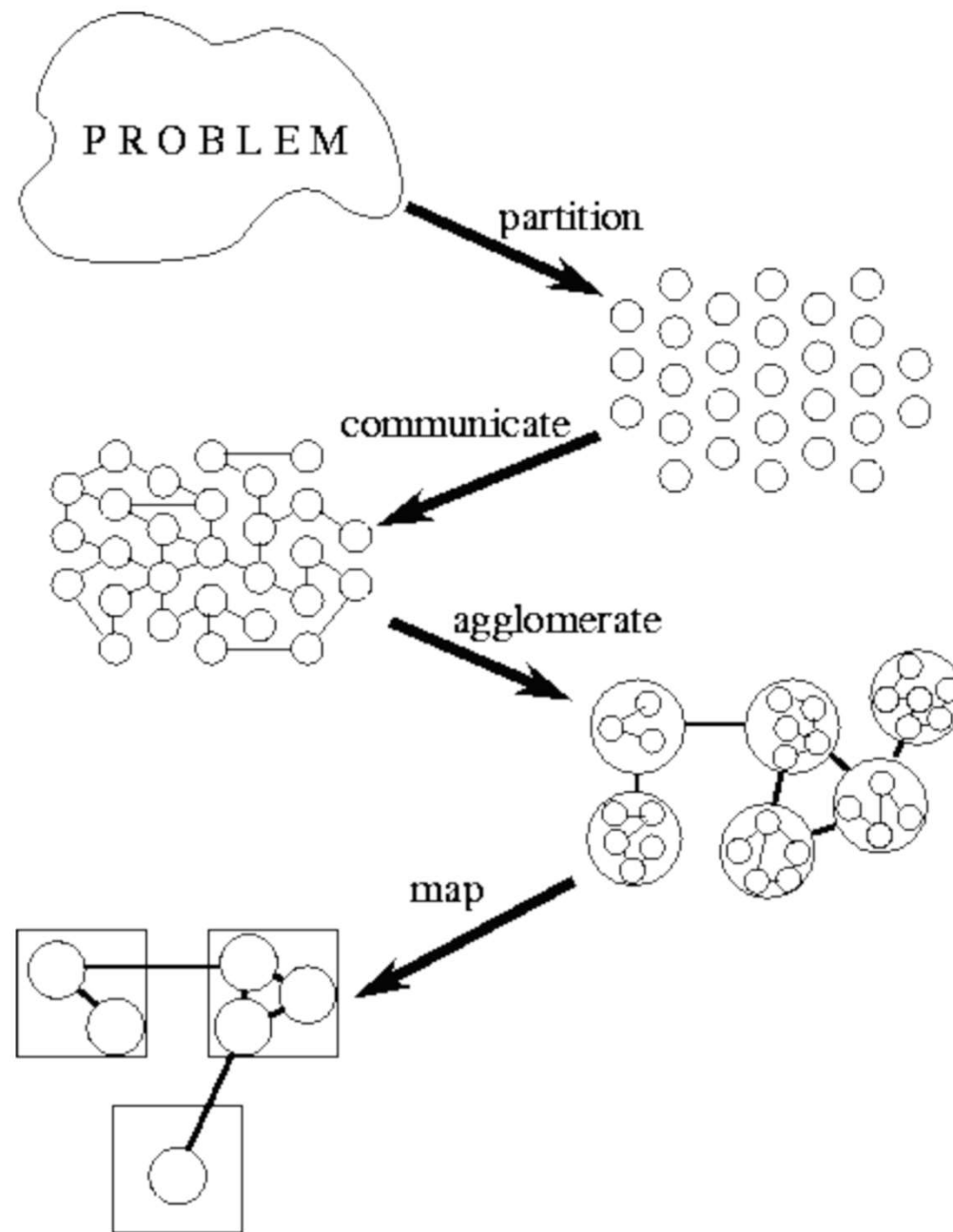
- Has agglomeration reduced communication costs by increasing locality? If not, examine your algorithm to determine whether this could be achieved using an alternative agglomeration strategy.
- Have you explored replicated data or computation to reduce communication cost and explored the benefits and costs?
- Has agglomeration affected load balancing in a negative way? You may need to check several agglomeration alternatives.
- Check the effect of agglomeration on scalability.

Foster's methodology (The PCAM Methodology)

4. Mapping: assign the composite tasks identified in the previous step to processes/threads.

This should be done such that: -

- Communication is minimized.
- Each process/thread gets roughly the same amount of work.



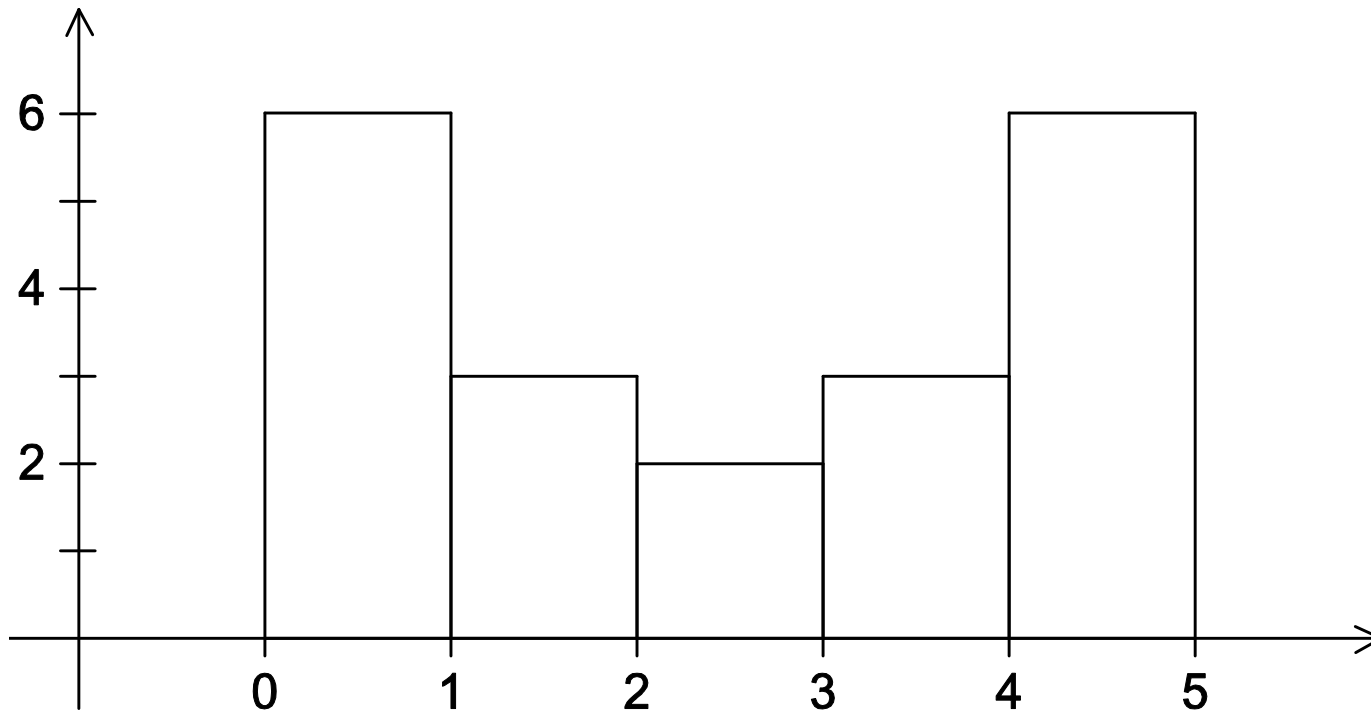
Source: "Designing and Building Parallel Programs" by Ian Foster

In General

- [The P & C in PCAM model] You design your program using machine-independent issues:
 - concurrency
 - scalability
 - ...
- [The A & M in PCAM model] You tweak your program to make the best use of the underlying hardware.

Example - histogram

- 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



Serial program - input

1. The number of measurements: `data_count`
2. An array of `data_count` floats: `data`
3. The minimum value for the bin containing the smallest values: `min_meas`
4. The maximum value for the bin containing the largest values: `max_meas`
5. The number of bins: `bin_count`

- Data[0] = 1.3
- Data[1] = 2.9
- Data[2] = 0.4
- Data[3] = 0.3
- Data[4] = 1.3
- Data[5] = 4.4
- Data[6] = 1.7
- Data[7] = 0.4
- Data[8] = 3.2
- Data[9] = 0.3
- Data[10] = 4.9
- Data[11] = 2.4
- Data[12] = 3.1
- Data[13] = 4.4
- Data[14] = 3.9,
- Data[15] = 0.4
- Data[16] = 4.2
- Data[17] = 4.5
- Data[18] = 4.9
- Data[19] = 0.9

data_count = 20

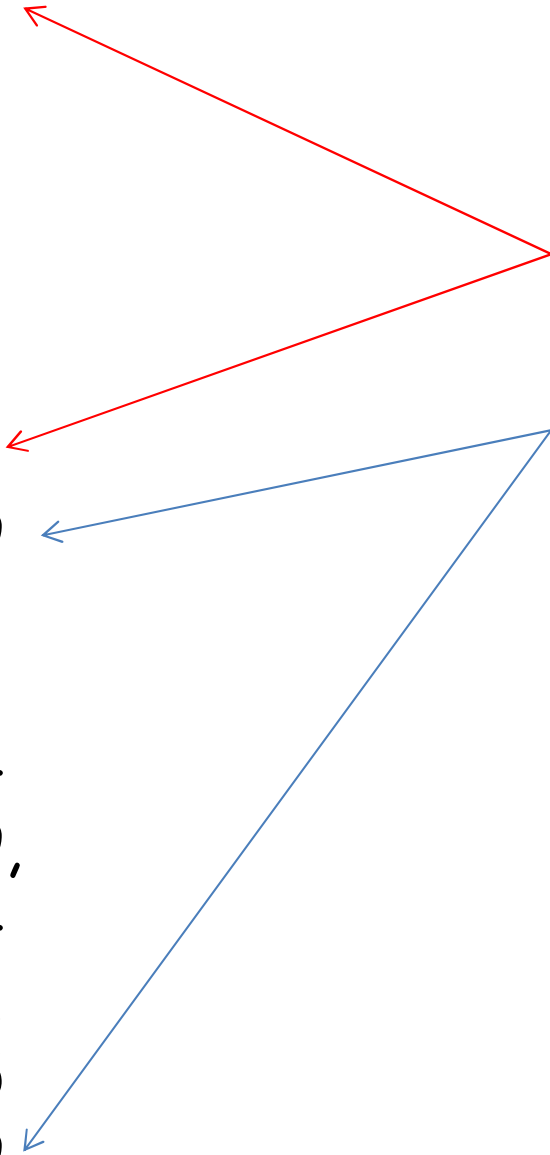
- Data[0] = 1.3
- Data[1] = 2.9
- Data[2] = 0.4
- Data[3] = 0.3
- Data[4] = 1.3
- Data[5] = 4.4
- Data[6] = 1.7
- Data[7] = 0.4
- Data[8] = 3.2
- Data[9] = 0.3
- Data[10] = 4.9
- Data[11] = 2.4
- Data[12] = 3.1
- Data[13] = 4.4
- Data[14] = 3.9,
- Data[15] = 0.4
- Data[16] = 4.2
- Data[17] = 4.5
- Data[18] = 4.9
- Data[19] = 0.9

data_count = 20

min_meas = 0.3

max_meas = 4.9

bin_count = 5



Serial program - output

1. **bin_maxes** : an array of bin_count floats → store the upper bound of each bin
2. **bin_counts** : an array of bin_count ints → stores the number of elements in each bin

- Data[0] = 1.3
- Data[1] = 2.9
- Data[2] = 0.4
- Data[3] = 0.3
- Data[4] = 1.3
- Data[5] = 4.4
- Data[6] = 1.7
- Data[7] = 0.4
- Data[8] = 3.2
- Data[9] = 0.3
- Data[10] = 4.9
- Data[11] = 2.4
- Data[12] = 3.1
- Data[13] = 4.4
- Data[14] = 3.9,
- Data[15] = 0.4
- Data[16] = 4.2
- Data[17] = 4.5
- Data[18] = 4.9
- Data[19] = 0.9

bin_maxes[0] = 0.9

bin_maxes[1] = 1.7

bin_maxes[2] = 2.9

bin_maxes[3] = 3.9

bin_maxes[4] = 4.9

bin_counts[0] = 6

bin_counts[1] = 3

bin_counts[2] = 2

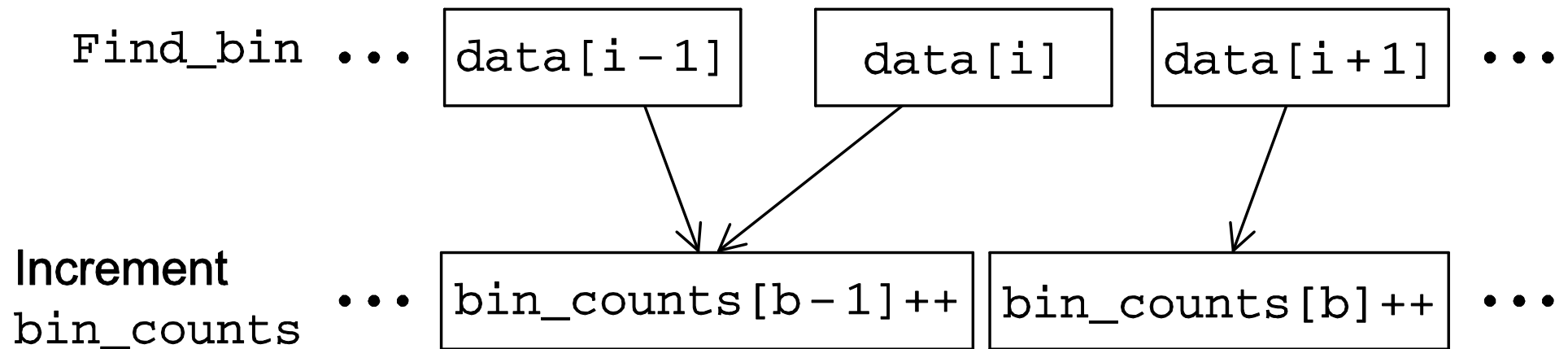
bin_counts[3] = 3

bin_counts[4] = 6

Serial Program

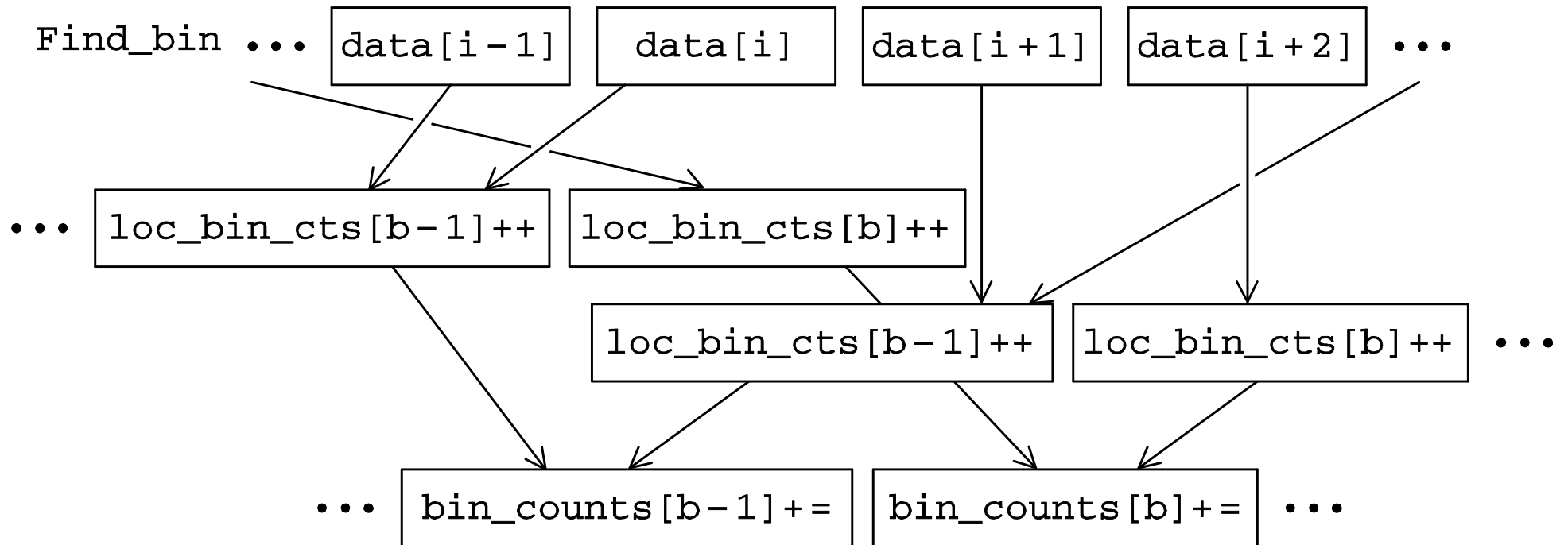
```
int bin = 0;
for( i = 0; i < data_count; i++){
    bin = find_bin(data[i], ...);
    bin_counts[bin]++;
}
```

First two stages of Foster's Methodology

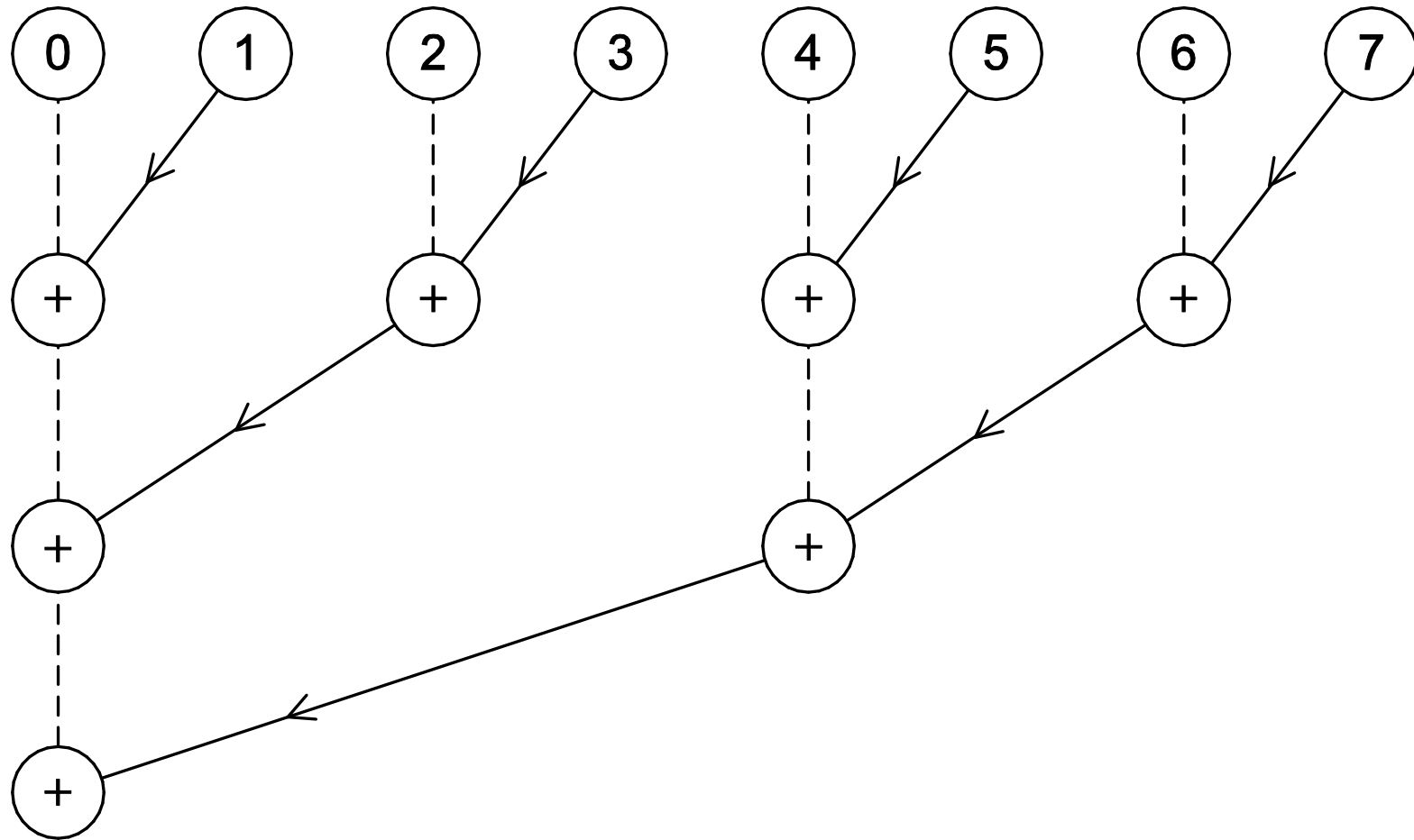


Find_bin returns the bin that `data[i]` belongs to.

Alternative definition of tasks and communication



Adding the local arrays



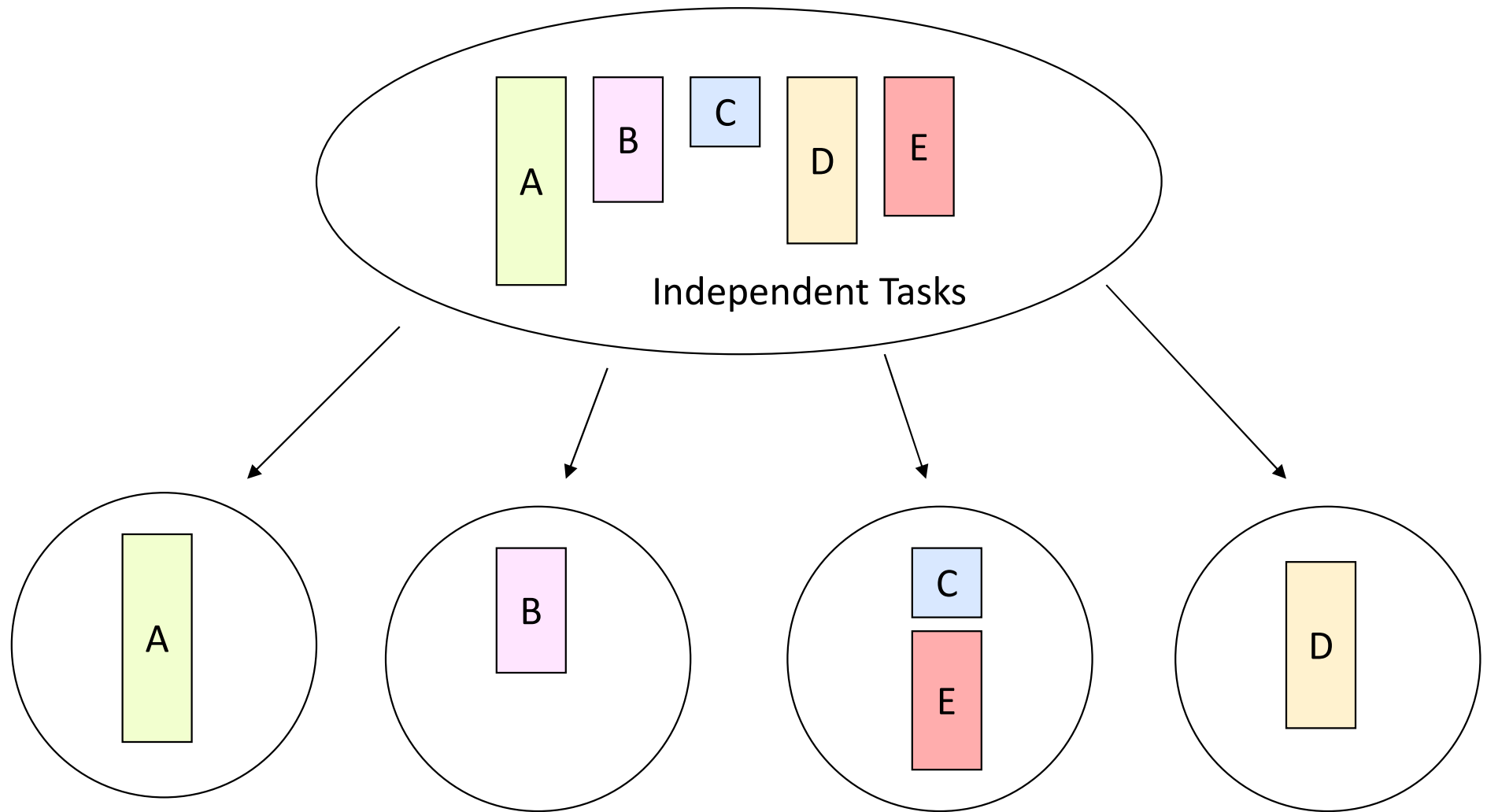
There are several ways for
parallelizing an algorithm ... depending
on the problem at hand.

What are these ways (or patterns)?

Patterns in Parallelism

- Task-level (e.g. Embarrassingly parallel)
- Divide and conquer
- Pipeline
- Iterations (loops)
- Client-server
- Geometric (usually domain dependent)
- Hybrid (different program phases)

Task Level



Task Level

- Break application into tasks, decided offline (a priori).
- Generally this scheme does not have *strong scalability*.

Example

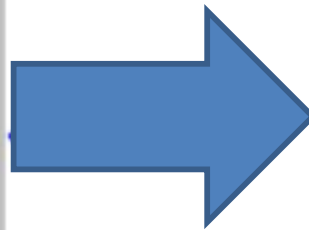
Assume we have a large array and we want to compute its minimum (T1), average (T2), and maximum (T3).

```
#define maxN 1000000000

int m[maxN];
int i;
int min = m[0];
int max = m[0];
double avrg = m[0];

for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
    avrg = avrg + m[i];
    if(m[i] > max)
        max = m[i];
}

avrg = avrg / maxN;
```



```
#define maxN 1000000000
int m[maxN];

int i; int min = m[0];
for(i=1; i < maxN; i++) {
    if(m[i] < min)
        min = m[i];
}

int j;
double avrg = m[0];
for(j=1; j < maxN; j++) {
    avrg = avrg + m[j];
}
avrg = avrg / maxN;

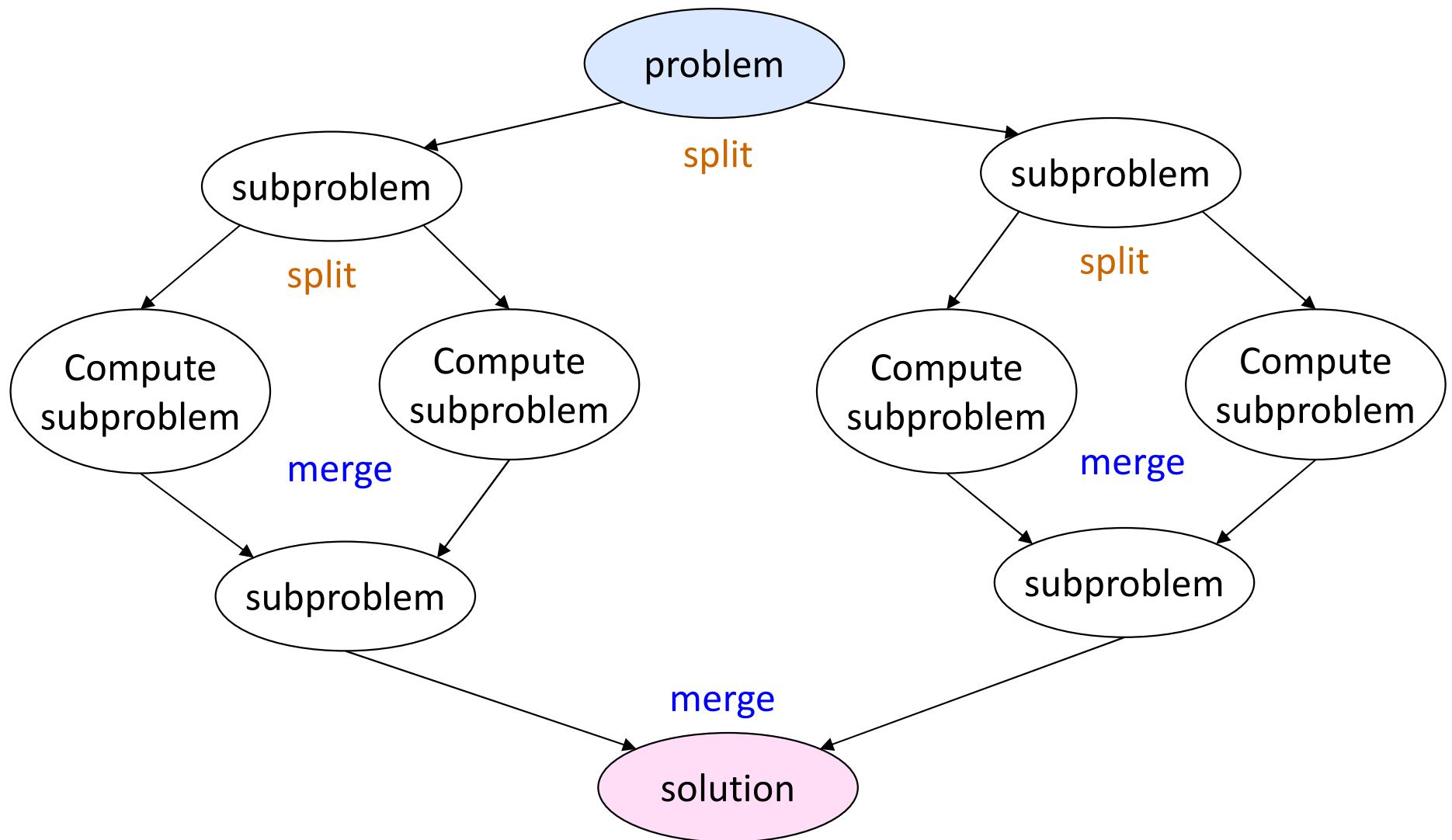
int k; int max = m[0];
for(k=1; k < maxN; k++) {
    if(m[k] > max)
        max = m[k];
}
```

T1

T2

T3

Divide-And-Conquer



Divide-And-Conquer

Sequentially, it looks like this:

```
// Input: A
DnD ( A ) {
    if ( A is a base case )
        return solution ( A ) ;
    else {
        split A into N subproblems B [ N ] ;
        for ( int i=0; i<N; i++)
            sol [ i ] = DnD ( B [ i ] ) ;
        return mergeSolution ( sol ) ;
    }
}
```

Divide-And-Conquer

Parallel Version:

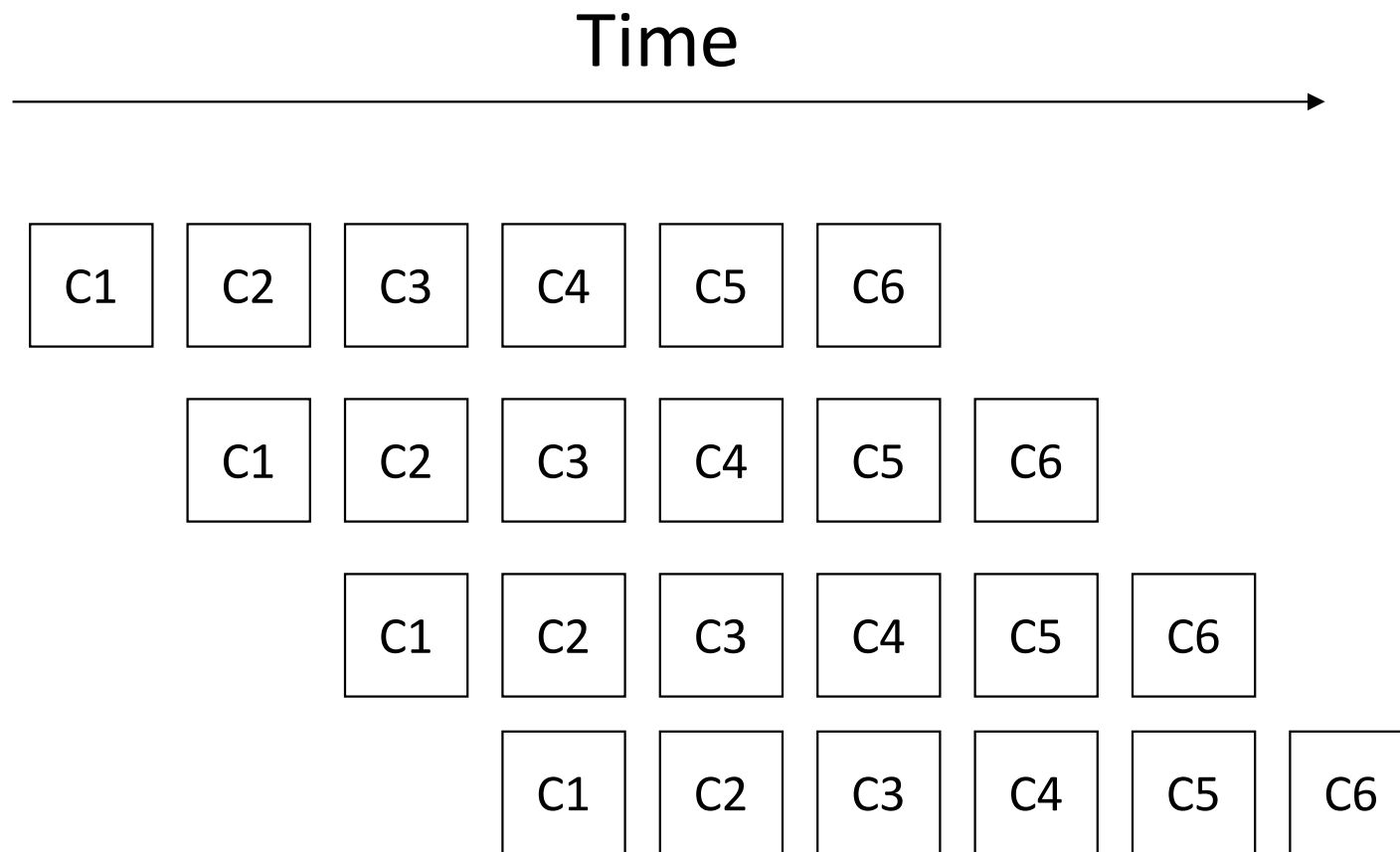
```
// Input: A
DnD(A) {
    if(isBaseCase( A ))
        return solution(A);
    else {
        if( bigEnoughForSplit( A ) ) { // if problem is big enough
            split A into N subproblems B[N];
            for(int i=0;i<N;i++)
                task[i] = newTask( DnD( B[i] ) ); // non-blocking

            for(int i=0;i<N;i++)
                sol[i] = getTaskResult( task[i] ); // blocking results ←
                wait

            return mergeSolution( sol );
        }
        else { // else solve sequentially
            return solution( A );
        }
    }
}
```

Pipeline

A series of **ordered** but **independent** computation stages need to be applied on data.



Pipeline

- Useful for
 - streaming workloads
 - Loops that are hard to parallelize
 - due inter-loop dependence
- How to do it?
 1. Split each loop iteration into independent stages (e.g. S1, S2, S3, ...)
 2. Assign each stage to a thread (e.g. T1 does S1, T2 does S2, ...).
 3. When a thread is done with each stage, it can start the same stage for the following loop iteration (e.g. T1 finishes S1 of iteration 0, then start S1 of iteration 1, etc.).
- Advantages
 - Expose intra-loop parallelism
 - Locality increases for variables used across stages
- How shall we divide an iteration into stages?
 - number of stages
 - inter-loop vs intra-loop dependence

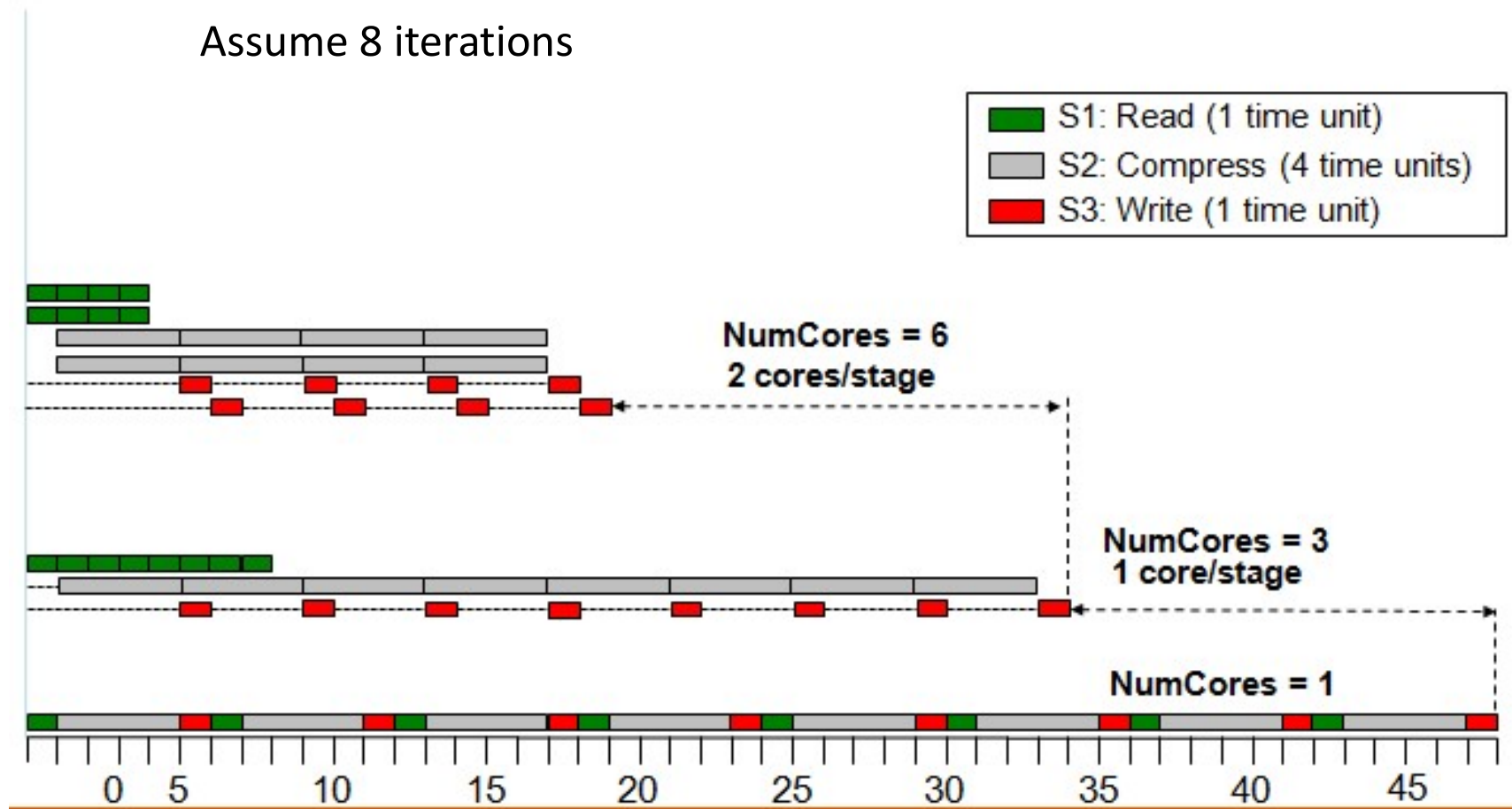
Example of pipeline parallelism

```
while(!done) {  
    Read block;  
    Compress the block;  
    Write block;  
}
```

Source of example:

<http://www.futurechips.org/parallel-programming-2/parallel-programming-clarifying-pipeline-parallelism.html>

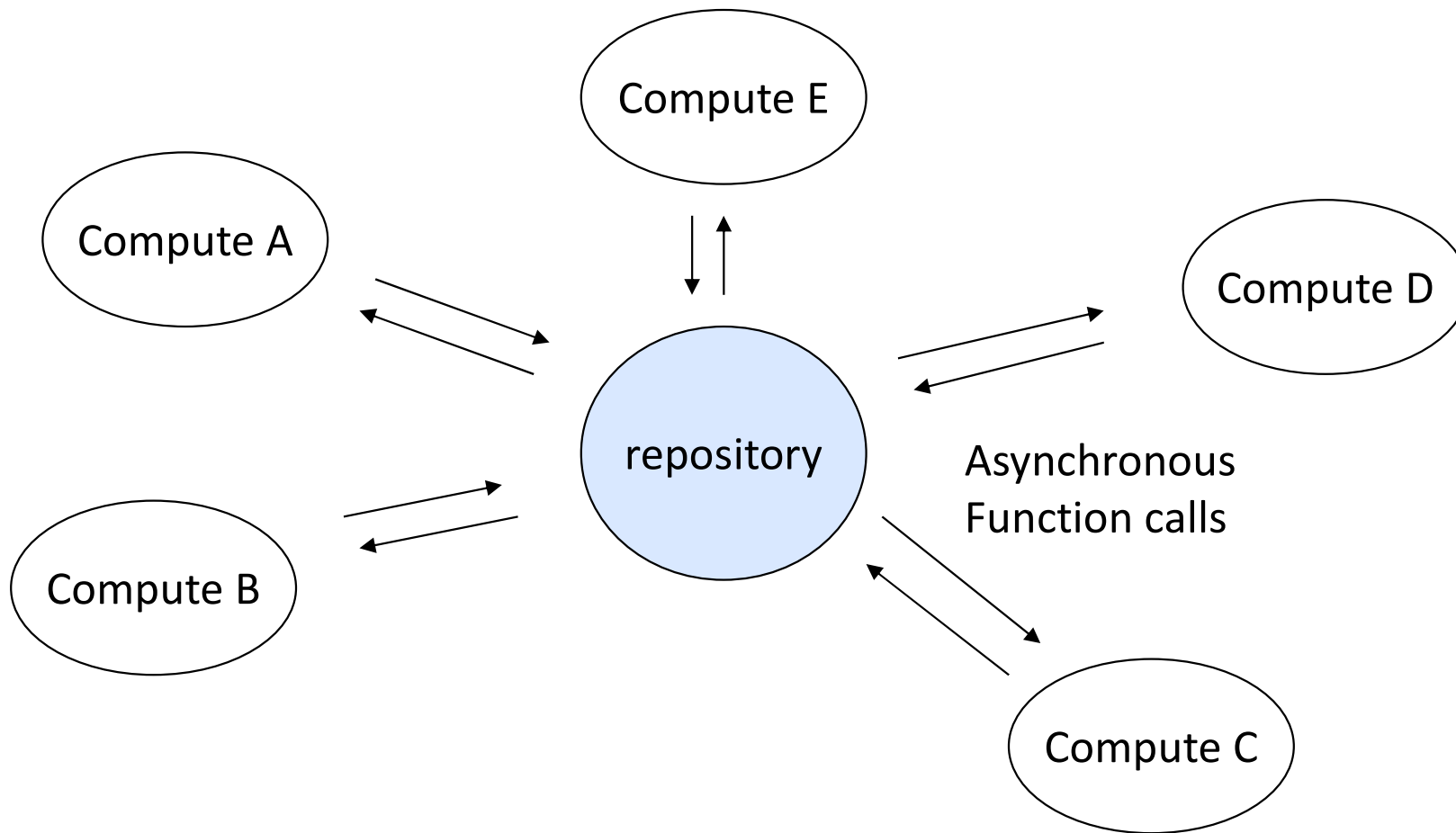
Example of pipeline parallelism



Source of example:

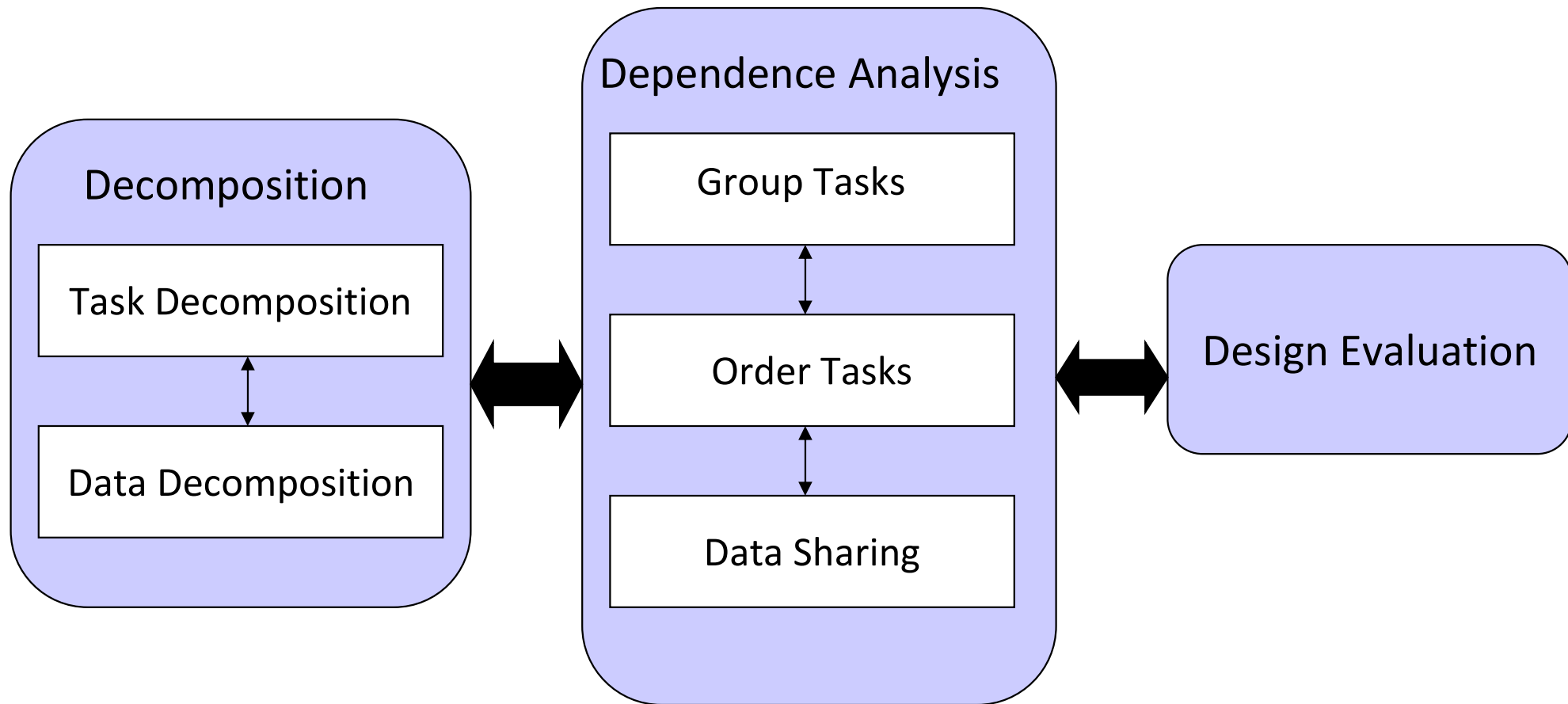
<http://www.futurechips.org/parallel-programming-2/parallel-programming-clarifying-pipeline-parallelism.html>

Repository Model



Whenever a thread is done with its task it can take another one from a repository.

The Big Picture of Parallel Programming



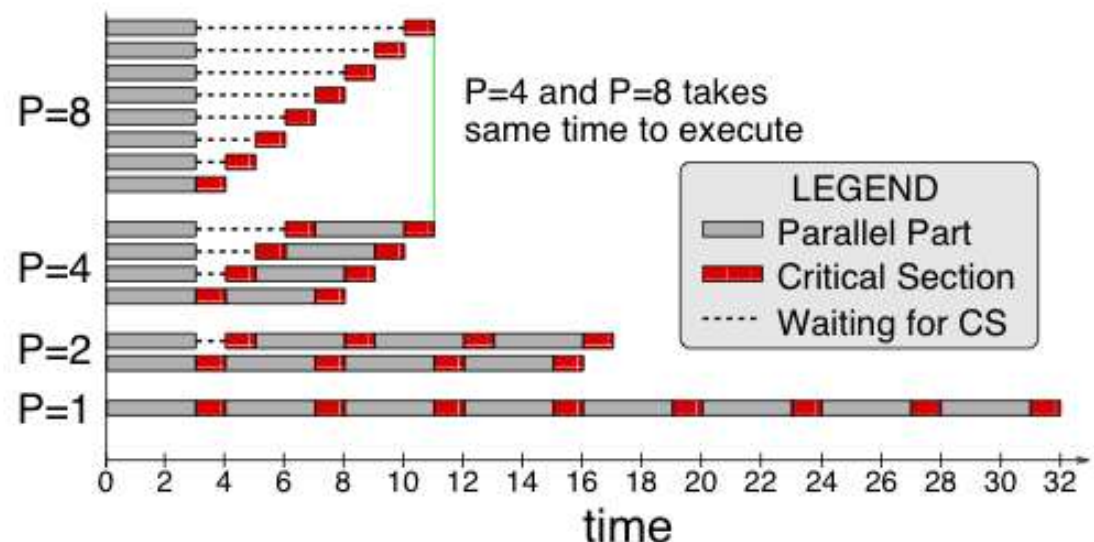
Source: David Kirk/NVIDIA and Wen-mei W. Hwu /UIUC

BUGS

- Sequential programming bugs + more
- Hard to find
- Even harder to resolve 😞
- Major reason for bugs: **race condition**

How to Avoid Race Condition?

- Prohibit more than one thread from reading and writing the shared data at the same time -> **mutual exclusion**
- The part of the program where the shared memory is accessed is called the **critical region**

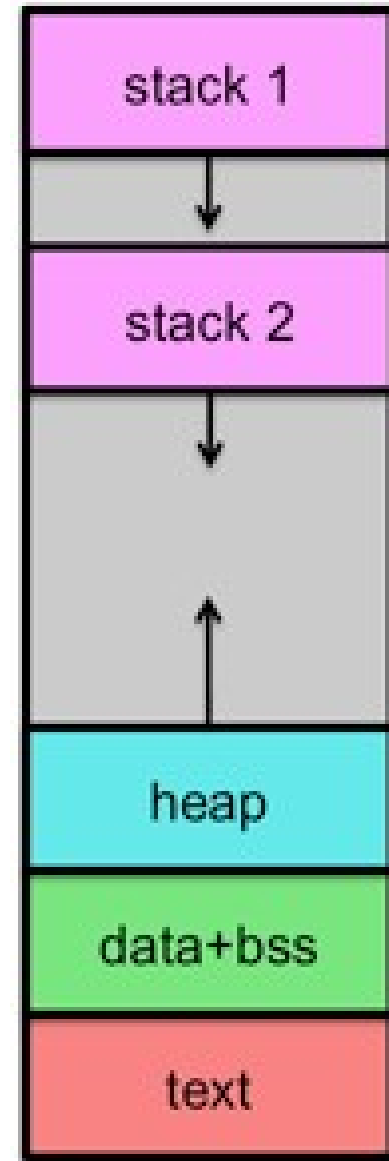


Conditions of Good Solutions to Race Condition

1. No two threads may be simultaneously inside their critical region.
2. No assumptions may be made about speeds or the number of CPUs/Cores.
3. No thread running outside its critical region may block other processes.
4. No thread has to wait forever to enter its critical region.

About Threads

- Thread vs Process
 - Process consists of one or more threads
 - Each thread has its own stack
- Once created a thread can be in one of 4 states: ready, running, waiting (blocked), or terminated.



Memory layout of a process with two threads

Multithreaded Programs

- Using established APIs at the application program
 - Example: Pthreads and OpenMP
- OpenMP:
 - developer-friendly
 - Requires compiler supporting OpenMP API
- Pthreads
 - More lower-level
 - More control and richer constructs
- Higher-level languages exist, but they tend to sacrifice performance to make program-development easier.
 - Example: Haskell

Conclusions

- Pick your programming model
- Task decomposition vs Data decomposition
- Refine based on:
 - What can compiler do.
 - What can runtime do.
 - What does the hardware provide.