An Introduction to
Parallel Programming

Ing. Andrea Marongiu
(a.marongiu@unibo.it)

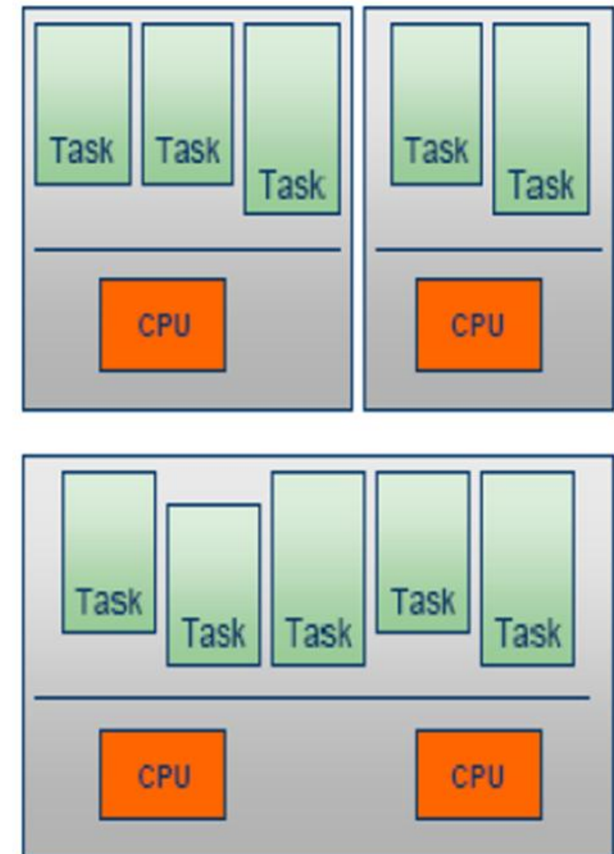
Includes slides from "*Multicore Programming Primer*" course at Massachusetts Institute of Technology (MIT) by Prof. Saman Amarasinghe and Dr. Rodric Rabbah

The Multicore Revolution

- More instruction-level parallelism hard to find
  - Very complex designs needed for small gains
  - Thread-level parallelism appears live and well
- Clock frequency scaling is slowing drastically
  - Too much power and heat when pushing the envelope
- Cannot communicate across chip fast enough
  - Better to design small local units with short paths
- Effective use of billion of transistors
  - Easier to reuse a basic unit many times
- Potential for very easy scaling
  - Just keep adding cores for higher (peak) performance

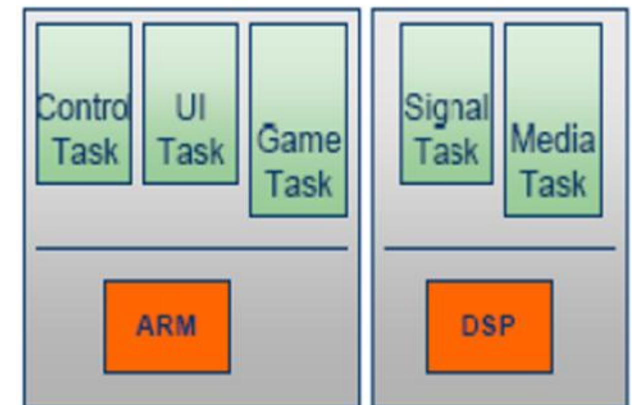
Vocabulary in the multi-era

- AMP (Asymmetric MP)
  - Each processor has local memory
  - Tasks statically allocated to one processor
- SMP (Symmetric MP)
  - Processors share memory
  - Tasks dynamically scheduled to any processor

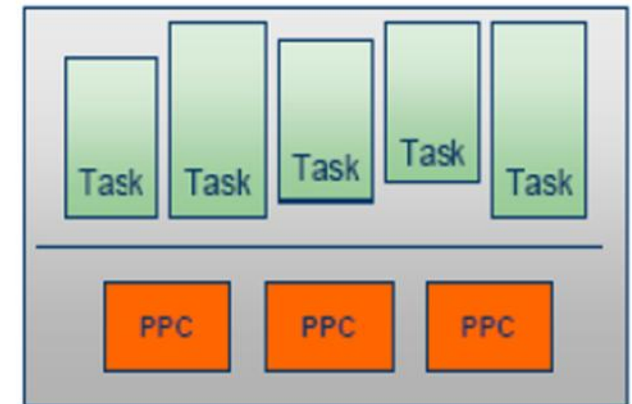


Vocabulary in the multi-era

- Heterogeneous:
  - Specialization among processors
  - Often different instruction sets
  - Usually AMP design

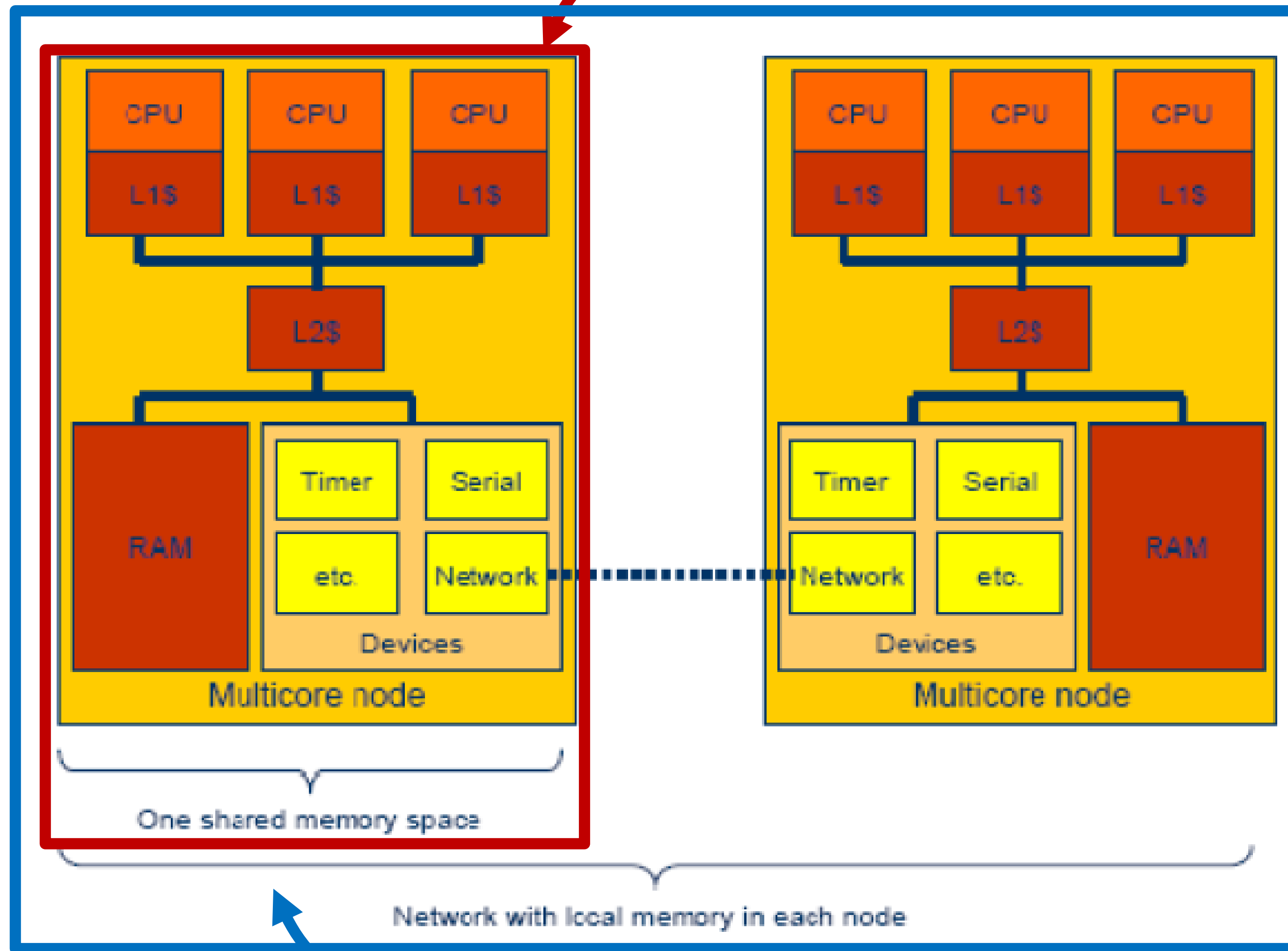


- Homogeneous:
  - All processors have the same instruction set
  - Processors can run any task
  - Usually SMP design



Future many-cores

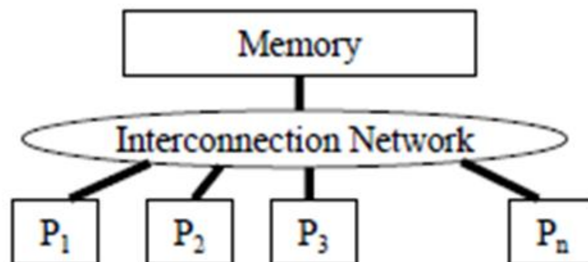
Locally homogeneous

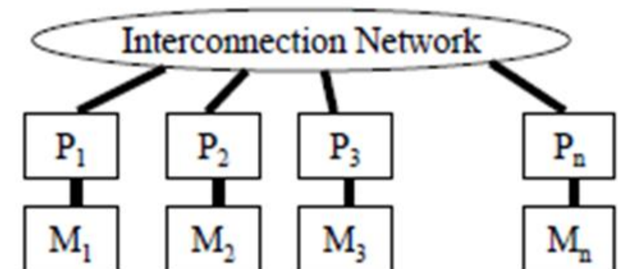


Globally heterogeneous

Multicore design paradigm

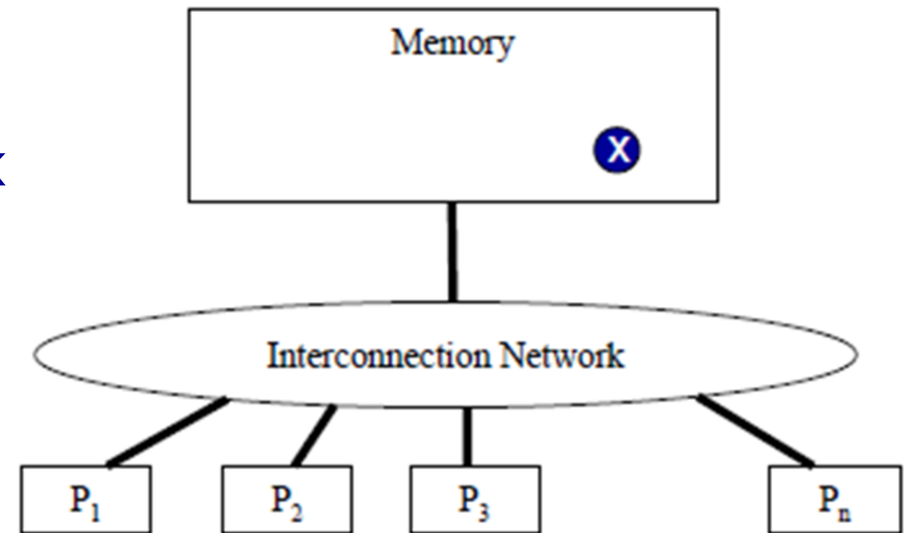
- Two primary patterns of multicore architecture design
  - Shared memory
    - Ex: Intel Core 2 Duo/Quad
    - One copy of data shared among many cores
    - Atomicity, locking and synchronization essential for correctness
    - Many scalability issues
  - Distributed memory
    - Ex: Cell
    - Cores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





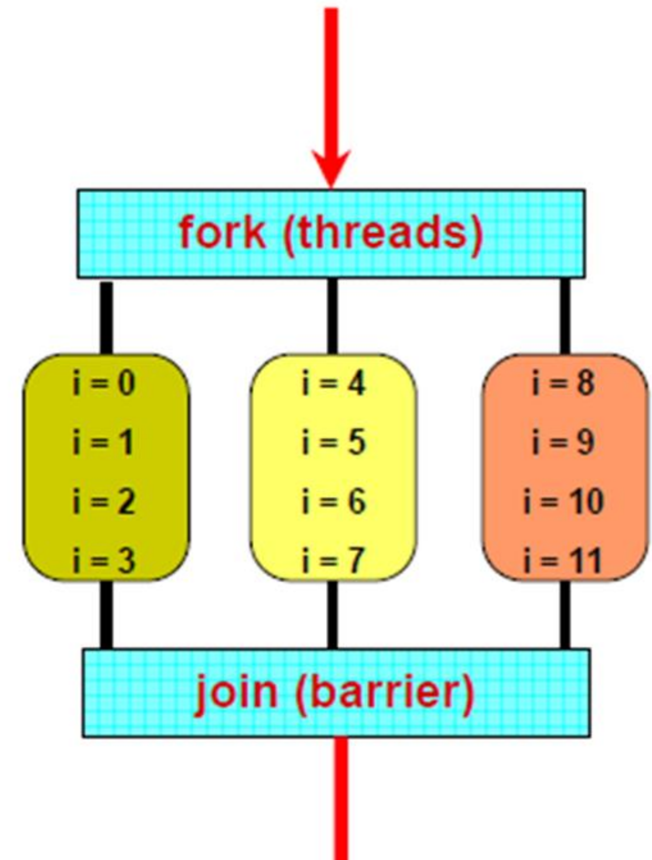
Shared Memory Programming

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



Example

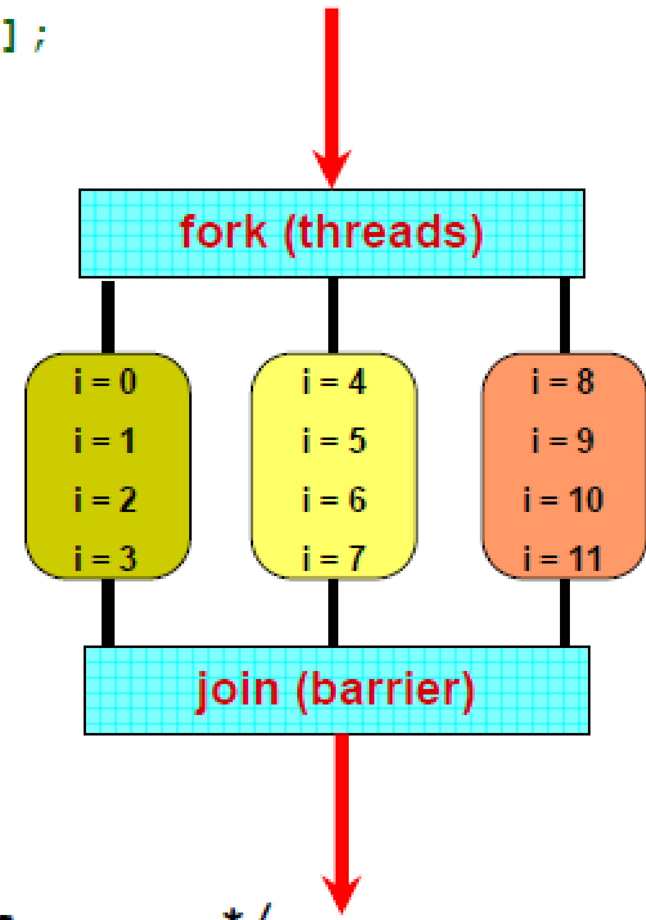
```
for (i = 0; i < 12; i++)
     C[i] = A[i] + B[i];
```



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

Pthreads

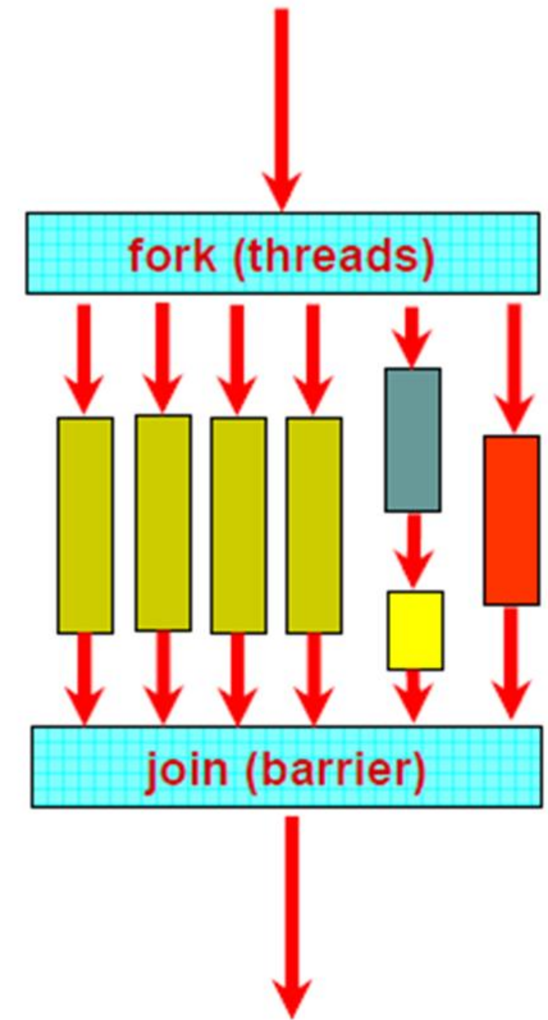
```
int A[12] = {...}; int B[12] = {...}; int C[12];

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

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



Types of Parallelism

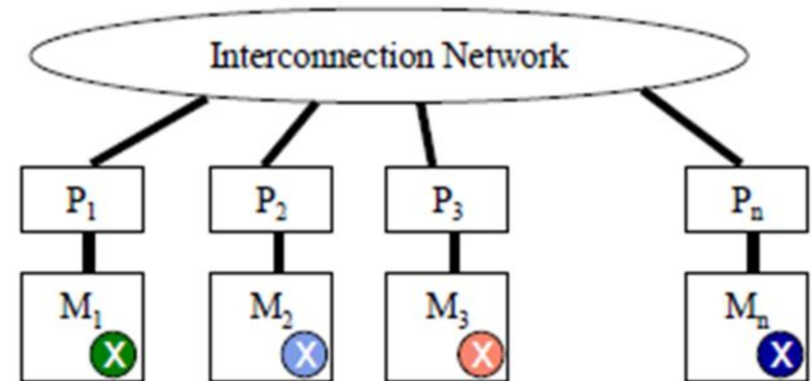
- Data parallelism
  - Perform same computation but operate on different data
- Task (control) parallelism
  - Perform different functions

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



Distributed Memory Programming

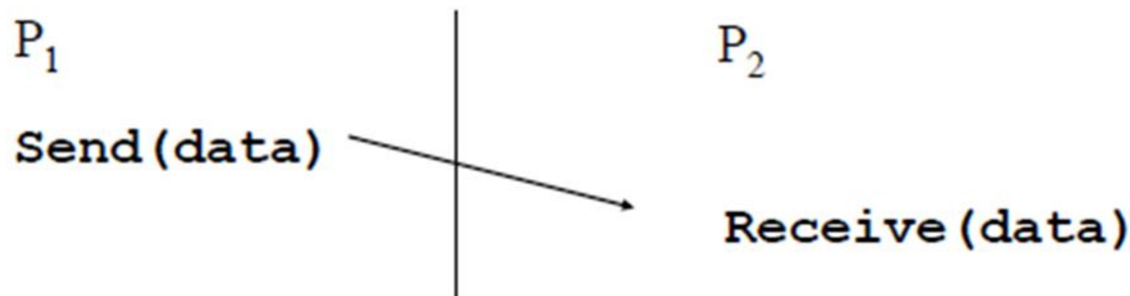
- Processor 1..n ask for X
- There are n places to look
  - Each processor's memory has its own X
  - Xs may vary



- For Processor 1 to look at Processor's 2's X
  - Processor 1 has to request X from Processor 2
  - Processor 2 sends a copy of its own X to Processor 1
  - Processor 1 receives the copy
  - Processor 1 stores the copy in its own memory

Message Passing

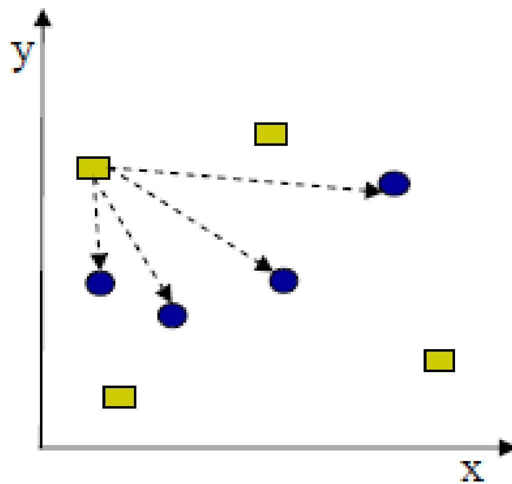
- Architectures with distributed memories use explicit communication to exchange data
  - Data exchange requires synchronization (cooperation) between senders and receivers



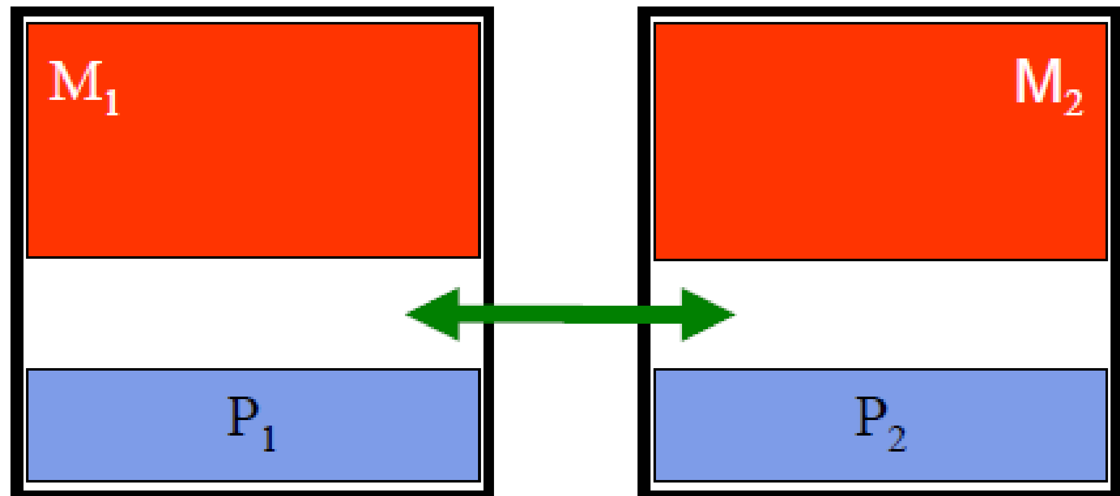
- How is "data" described
- How are processes identified
- Will receiver recognize or screen messages
- What does it mean for a send or receive to complete

Example

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

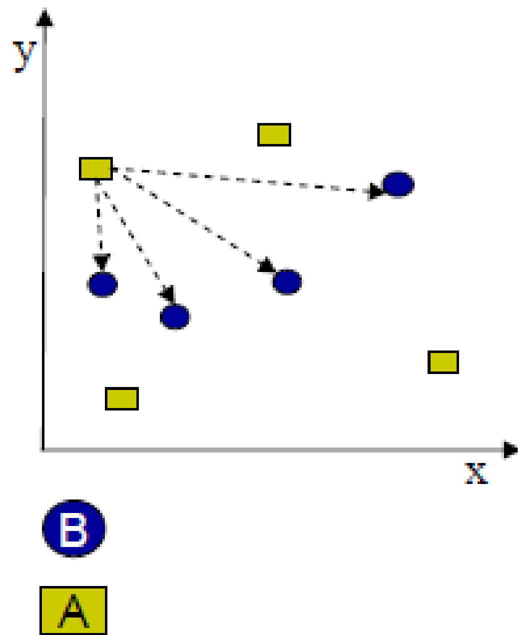



```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

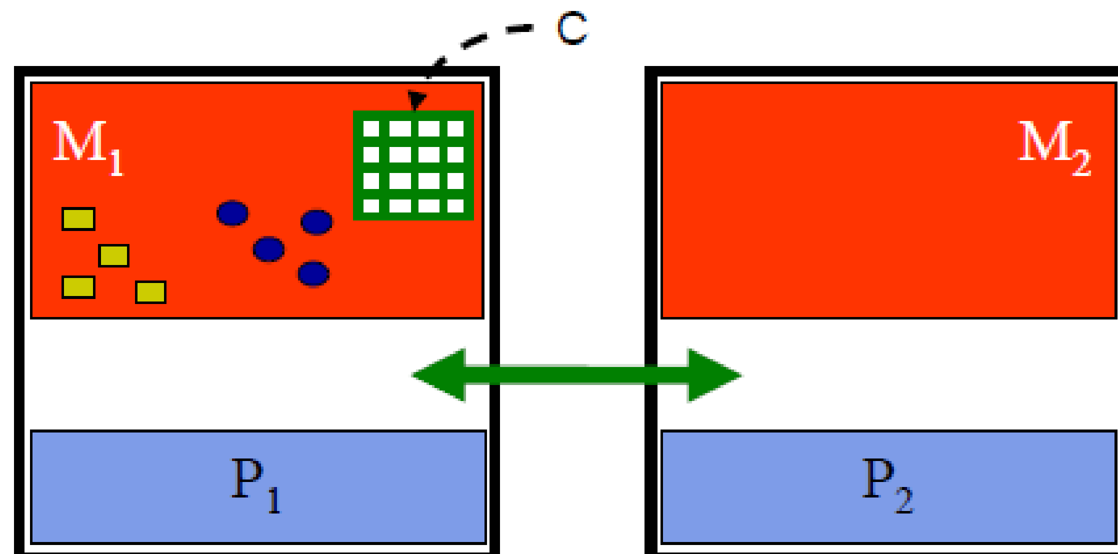


Example

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




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

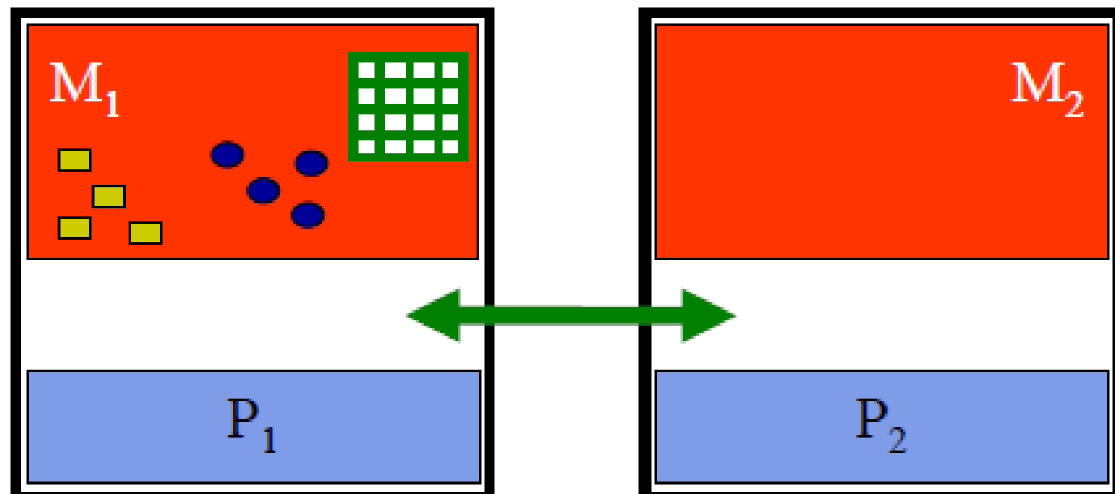


Example

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

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

- Can break up work between the two processors
  - P_1 sends data to P_2

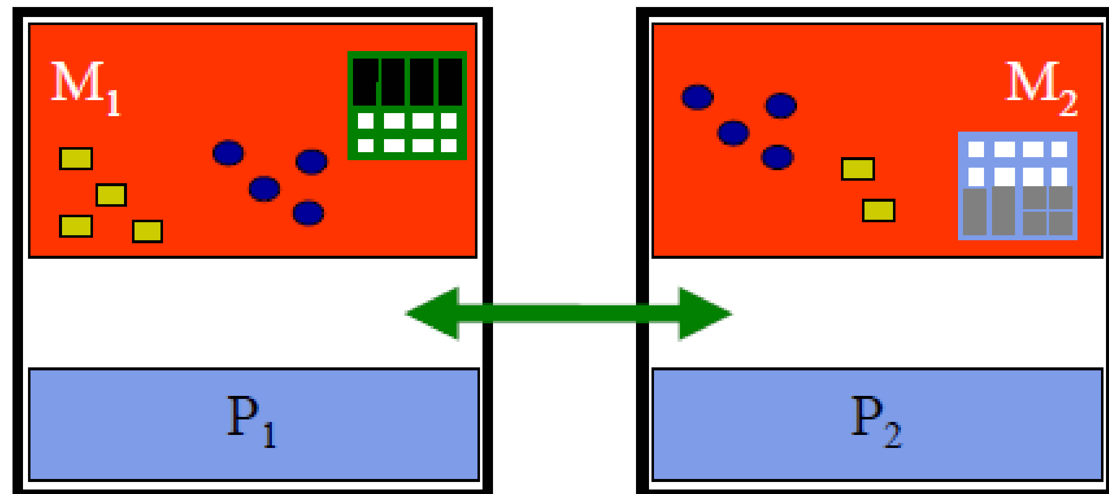


Example

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

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

- Can break up work between the two processors
  - P_1 sends data to P_2
  - P_1 and P_2 compute

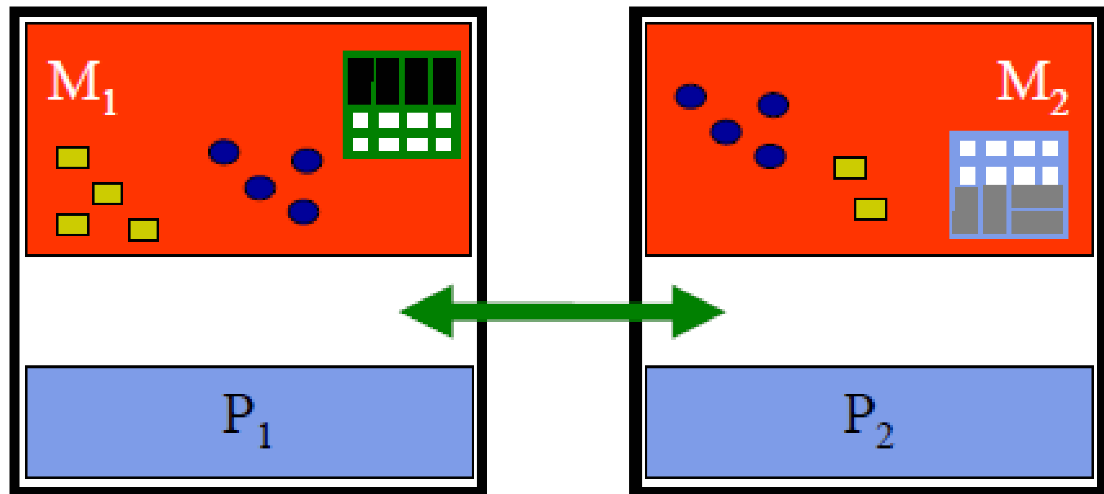


Example

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

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

- Can break up work between the two processors
  - P_1 sends data to P_2
  - P_1 and P_2 compute
  - P_2 sends output to P_1



Example

processor 1

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

sequential

parallel with messages

processor 1

```
A[n] = {…}
B[n] = {…}

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

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

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

processor 2

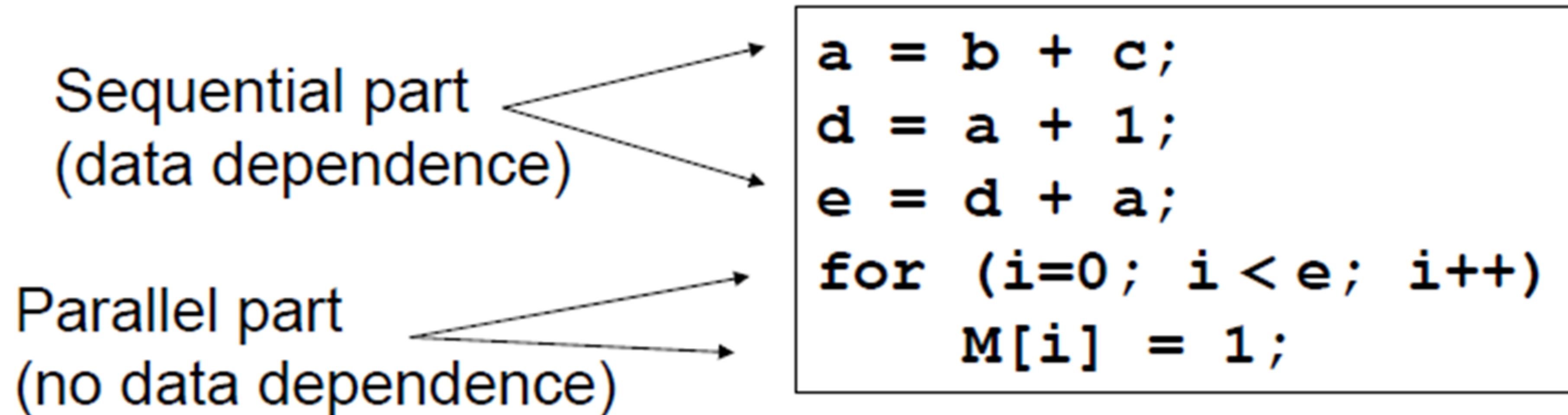
```
A[n] = {…}
B[n] = {…}

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

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

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

Understanding Performance

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

Coverage

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

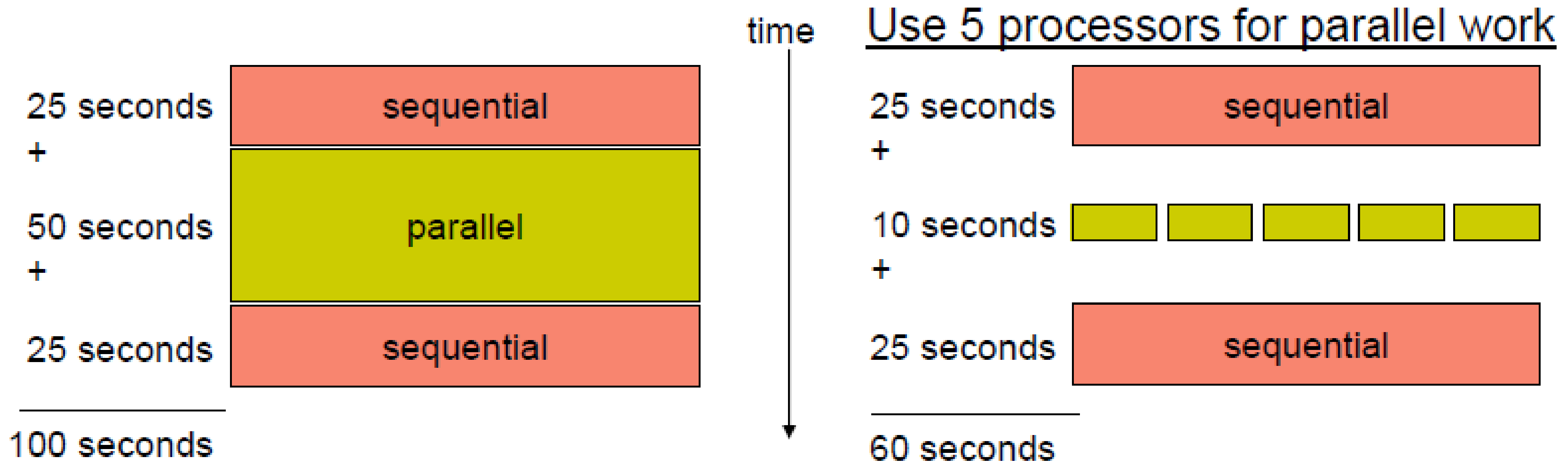


Amdahl's Law

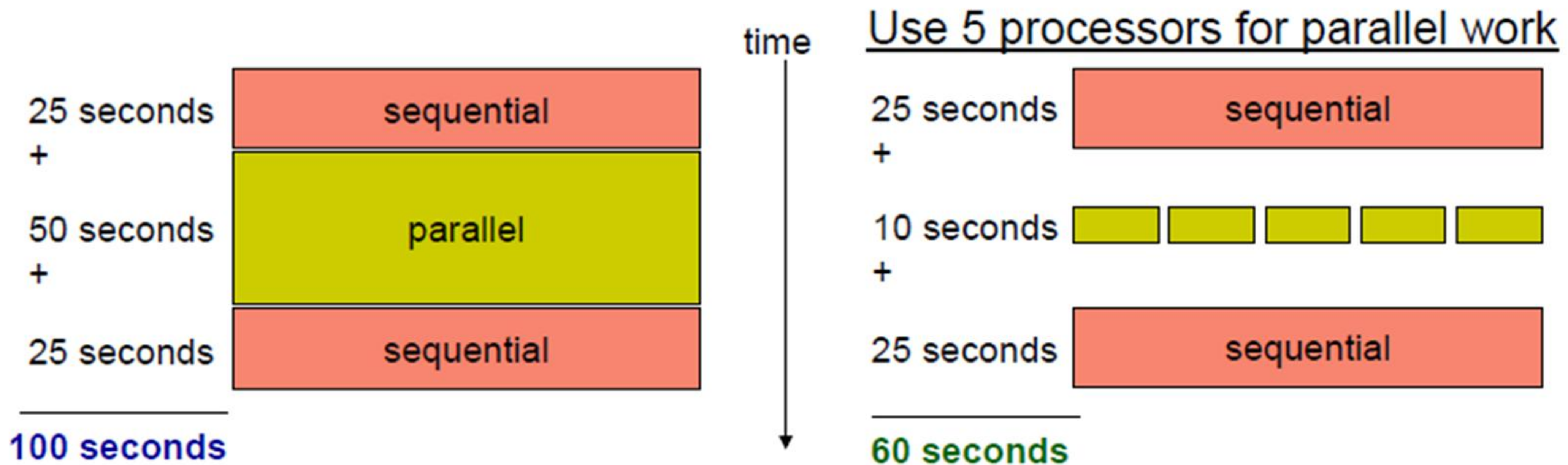
- **Amdahl's Law:** *The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used*

Amdahl's Law

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



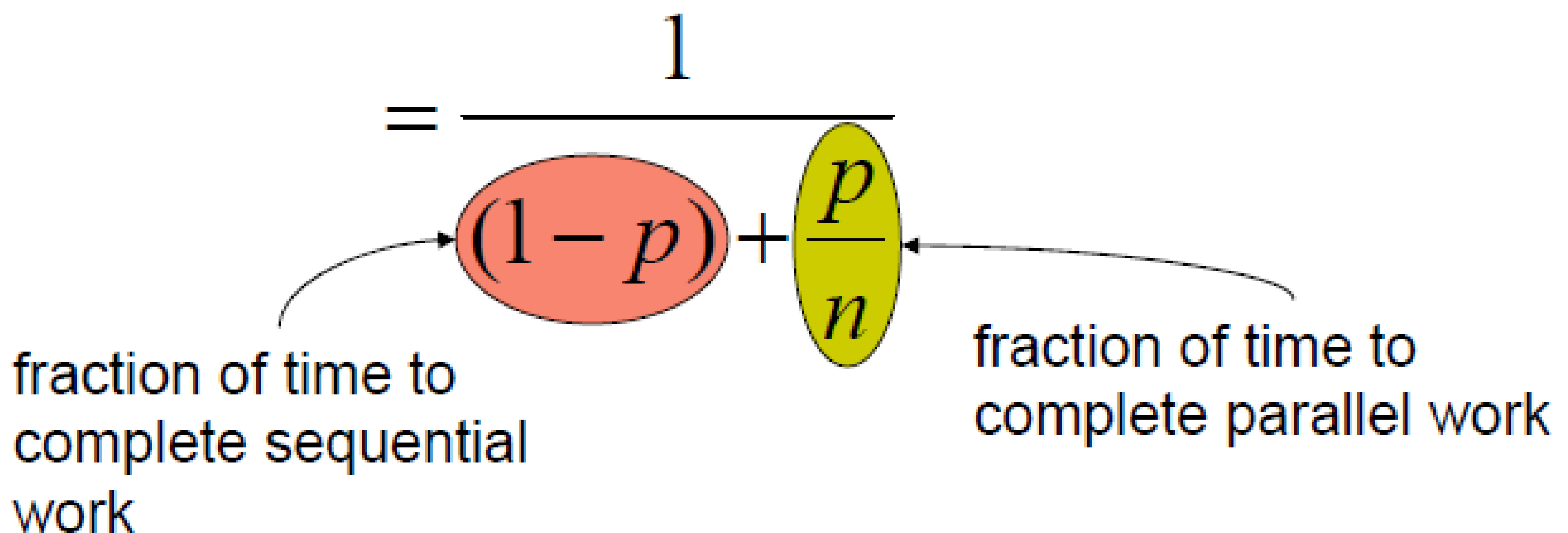
Amdahl's Law



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

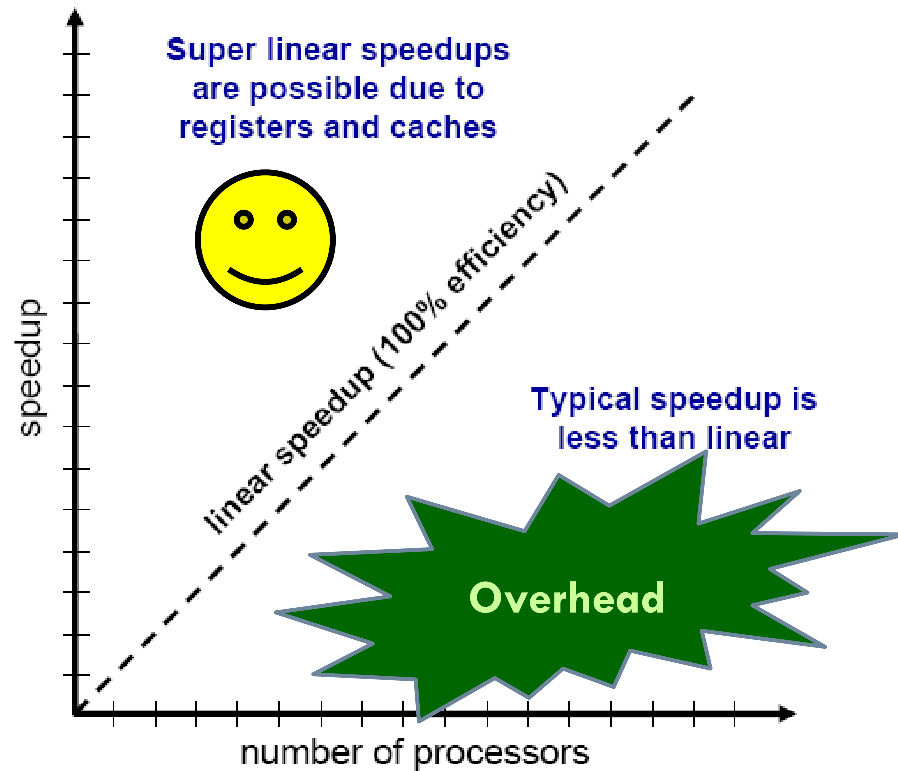
Amdahl's Law

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



Amdahl's Law

- Speedup tends to 1/(1-p) as number of processors tends to infinity
- Parallel programming is worthwhile when programs have a lot of work that is parallel in nature



Understanding Performance

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

Granularity

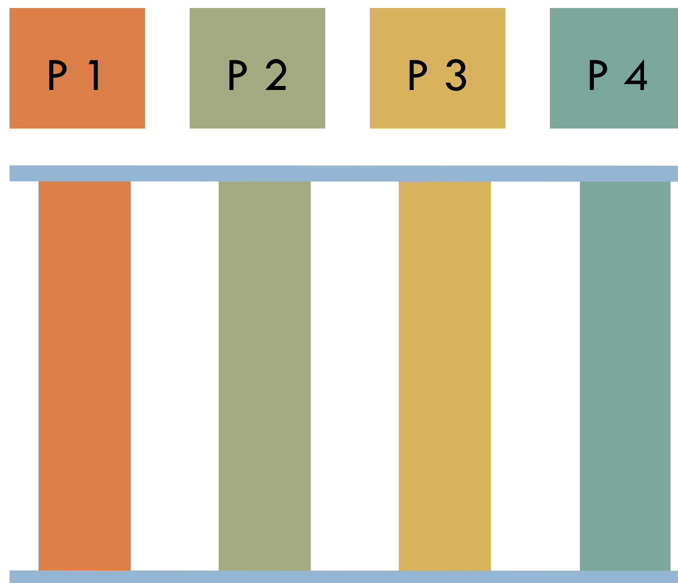
- Granularity is a qualitative measure of the ratio of computation to communication
- Computation stages are typically separated from periods of communication by synchronization events

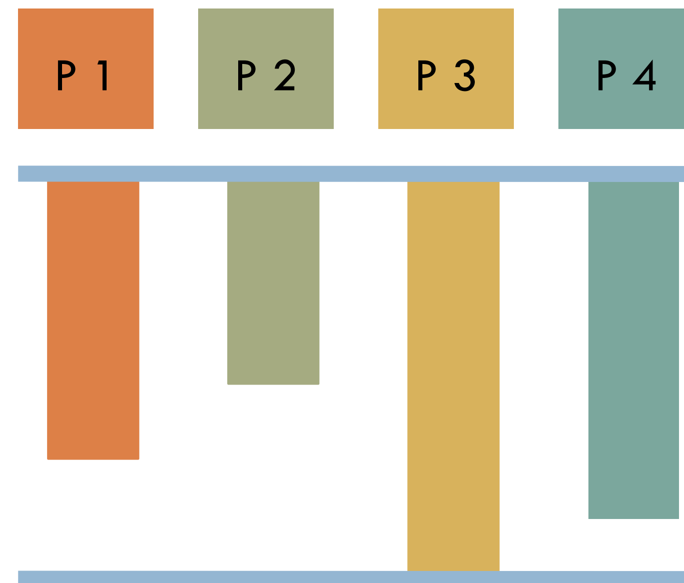
Granularity

- Fine-grain Parallelism
  - Low computation to communication ratio
  - Small amounts of computational work between communication stages
  - Less opportunity for performance enhancement
  - High communication overhead
- Coarse-grain Parallelism
  - High computation to communication ratio
  - Large amounts of computational work between communciation events
  - More opportunity for performance increase
  - Harder to load balance efficiently

Load Balancing

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

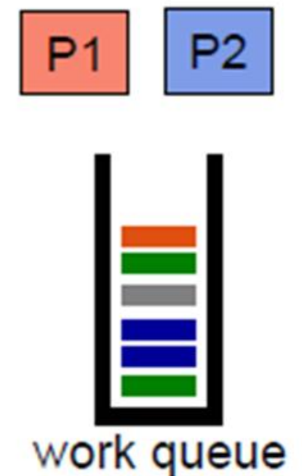




Slowest core dictates overall execution time

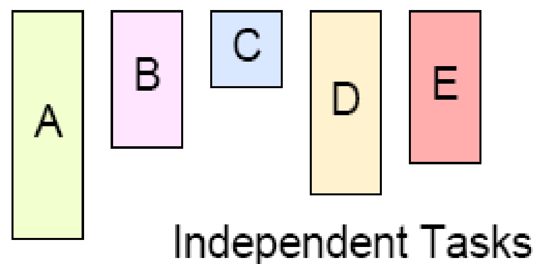
Static Load Balancing

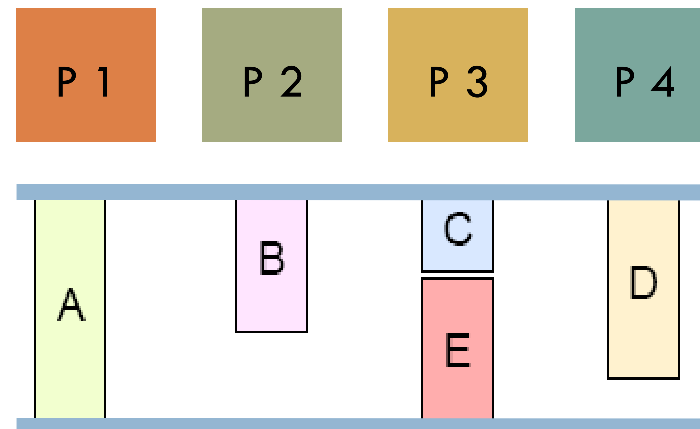
- Programmer make decisions and assigns a fixed amount of work to each processing core a priori
- Works well for homogeneous multicores
  - All core are the same
  - Each core has an equal amount of work
- Not so well for heterogeneous multicores
  - Some cores may be faster than others
  - Work distribution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



Dynamic Load Balancing

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





Communication and Synchronization

- In parallel programming processors need to ***communicate*** partial results on data or ***synchronize*** for correct processing
- In **shared memory** systems
  - Communication takes place ***implicitly*** by concurrently operating on shared variables
  - Synchronization primitives must be ***explicitly*** inserted in the code
- In **distributed memory** systems
  - Communication primitives (send/receive) must be ***explicitly*** inserted in the code
  - Synchronization is ***implicitly*** achieved through message exchange

Communication Cost Model

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

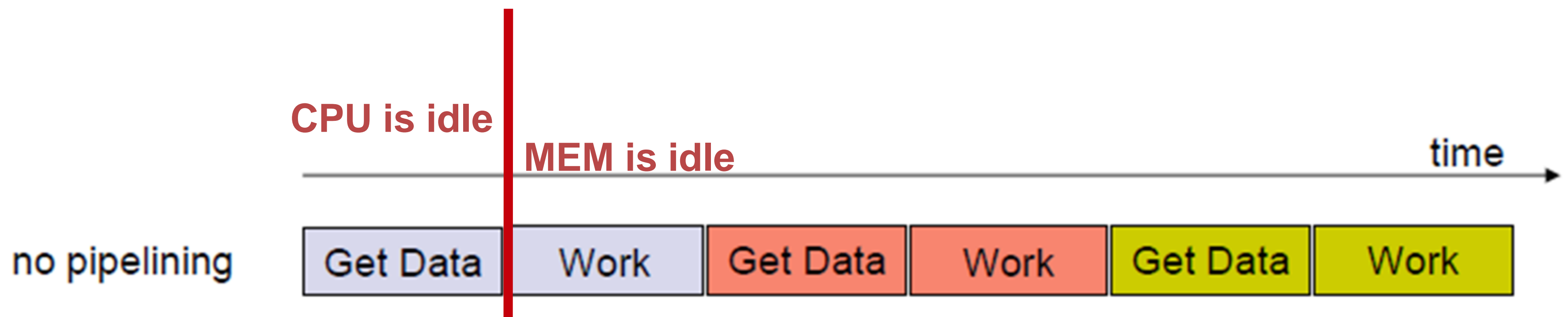
- f: frequency of messages
- o: overhead per message (at both ends)
- l: network delay per message
- n: total data sent
- m: number of messages
- B: bandwidth along path (determined by network)
- t: cost induced by contention per message
- $overlap$: (circled)

Types of Communication

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

Overlapping messages and computation

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



Overlapping messages and computation

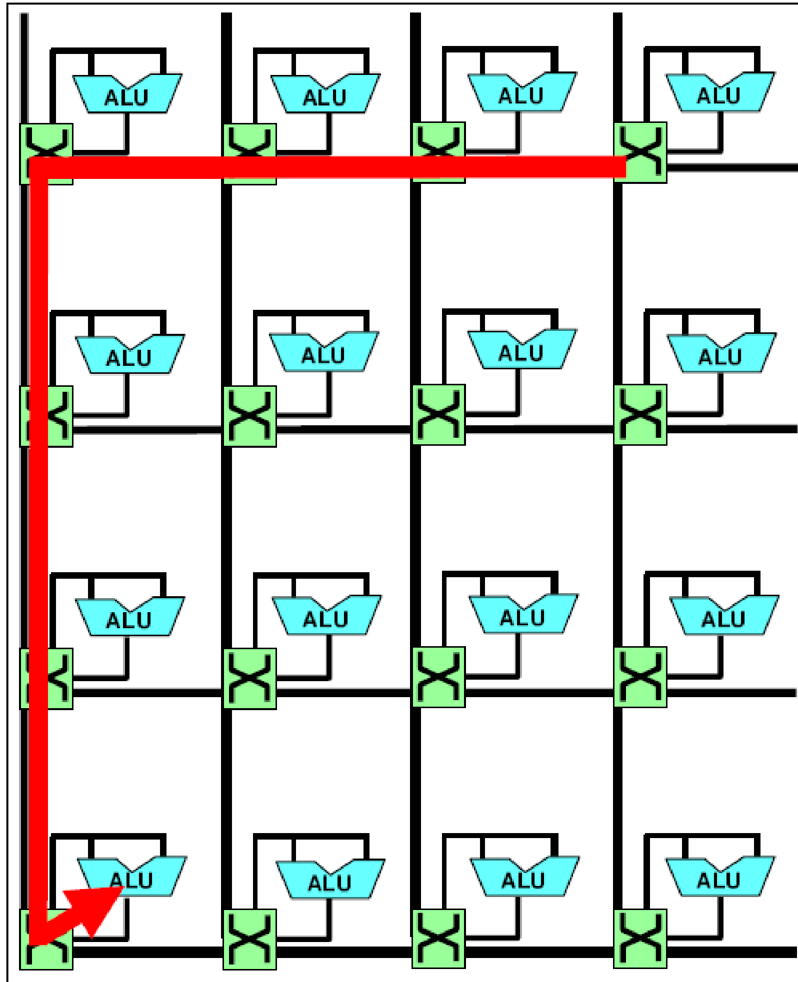
- Computation and communication concurrency can be achieved with pipelining
  - Think instruction pipelining in superscalars
- Essential for performance on Cell and similar distributed memory multicores

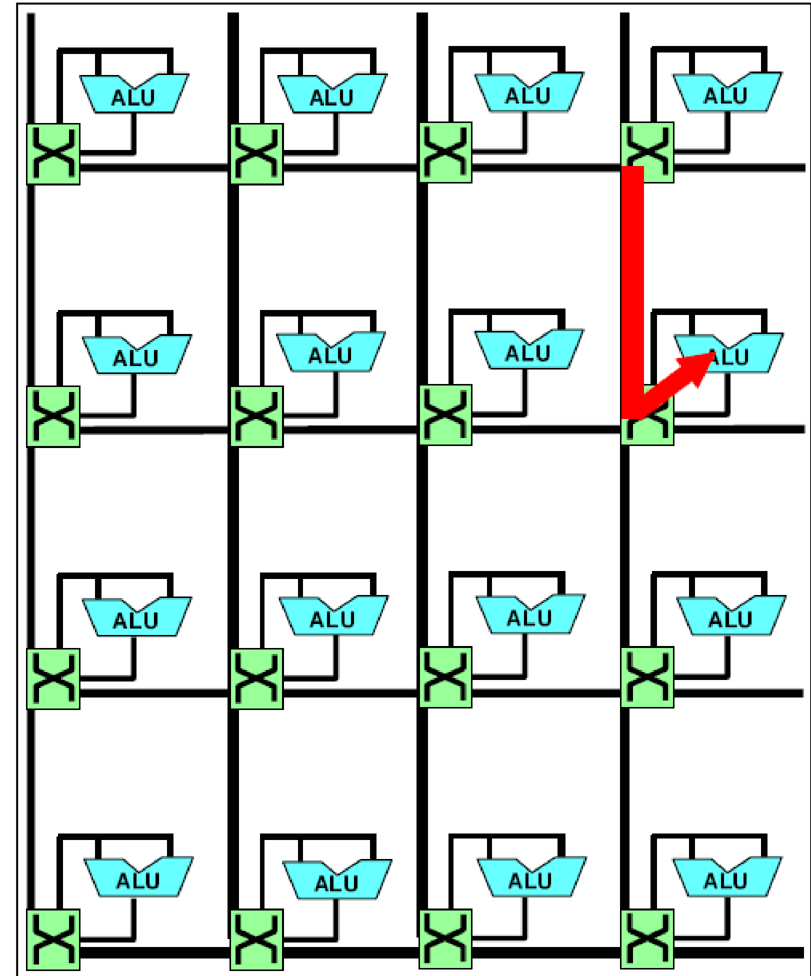
```
// Start transfer for first buffer
id = 0;
mfc_get(buf[id], addr, BUFFER_SIZE, id, 0, 0);
id ^= 1;

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

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

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

Understanding Performance

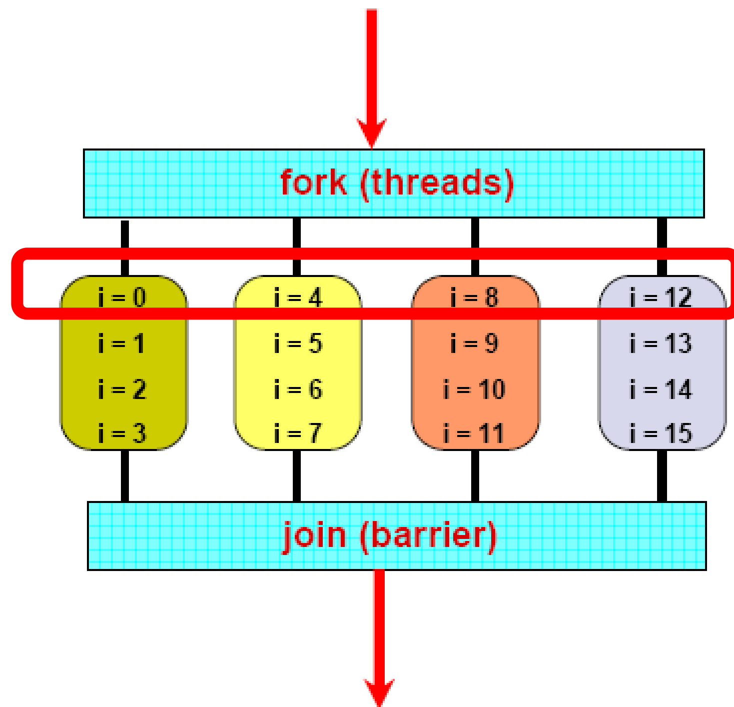
- **Coverage** or extent of parallelism in algorithm
- **Granularity** of partitioning among processors
- **Locality** of computation and communication

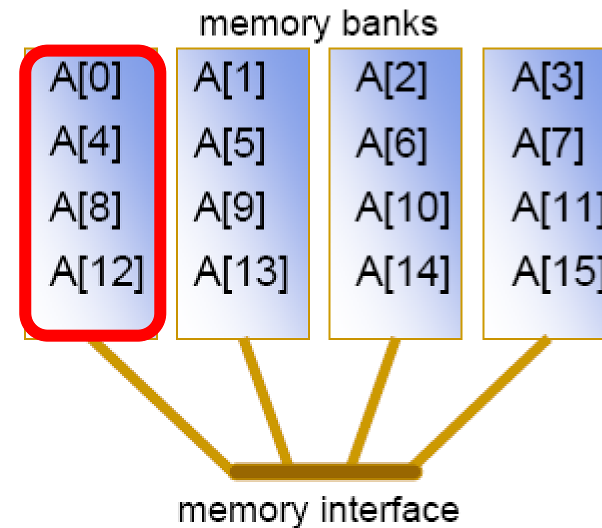
Locality in communication (message passing)



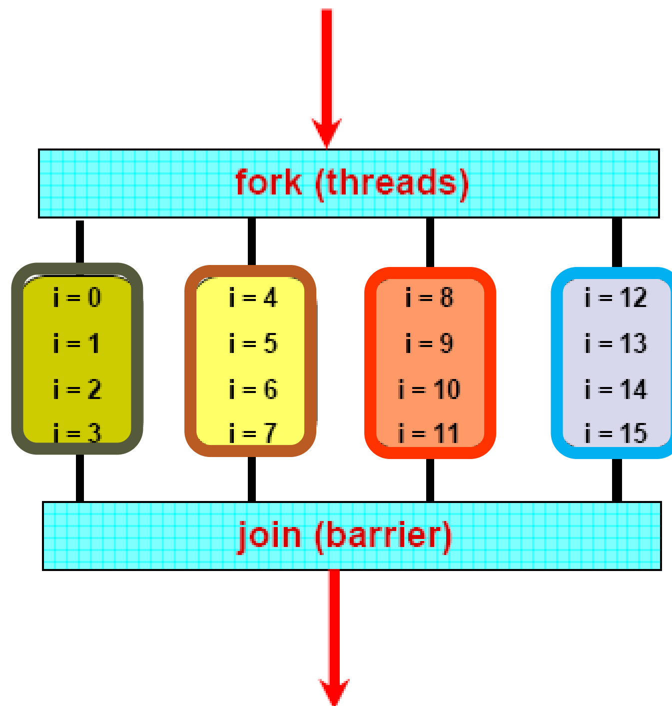


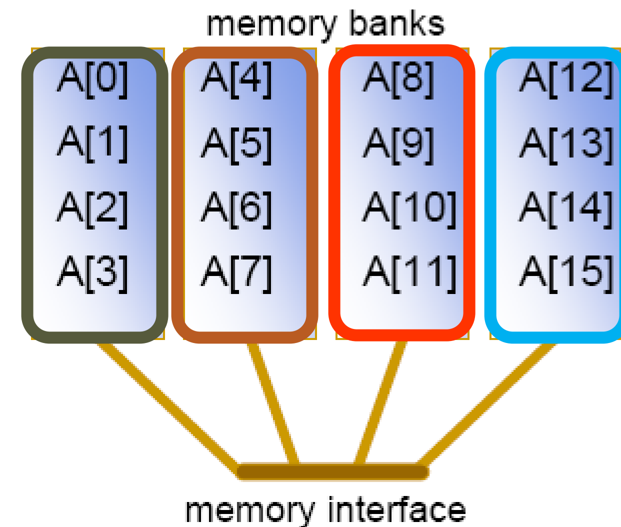
Locality of memory accesses (shared memory)





Locality of memory accesses (shared memory)



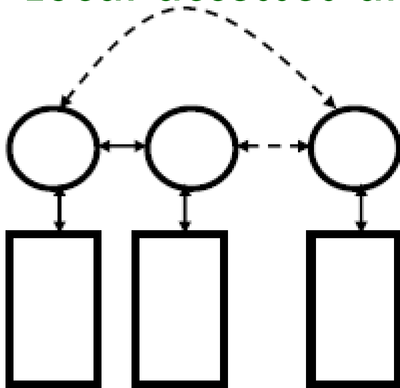


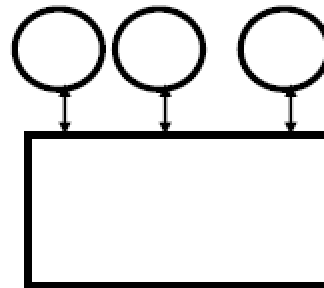
Memory access latency in shared memory architectures

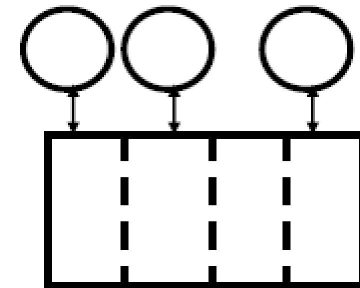
- Uniform Memory Access (UMA)
  - Centrally located memory
  - All processors are equidistant (access times)
- Non-Uniform Access (NUMA)
  - Physically partitioned but accessible by all
  - Processors have the same address space
  - Placement of data affects performance

Distributed Shared Memory

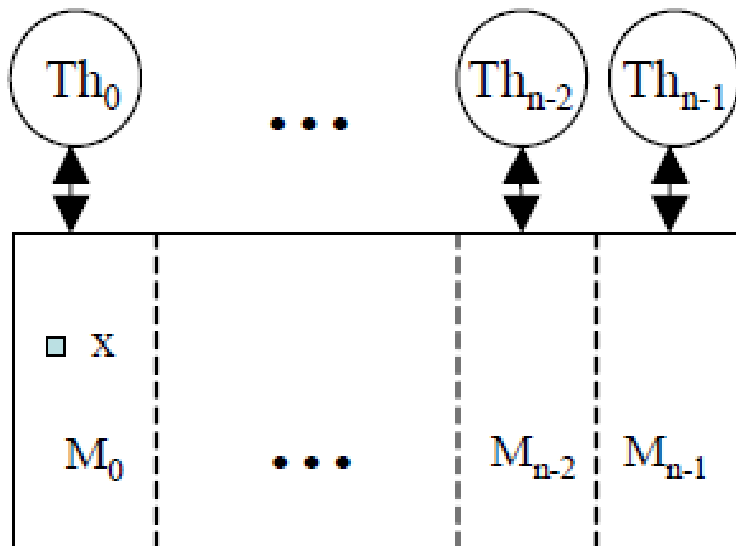
- A.k.a. **P**artitioned **G**lobal **A**ddress **S**pace (PGAS)
  - Each processor has a local memory node, globally visible by every processor in the system
  - Local accesses are fast, remote accesses are slow (NUMA)



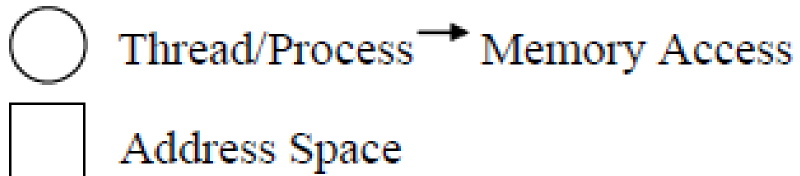




Distributed Shared Memory



Legend



- Concurrent threads with a partitioned shared space
- Similar to the shared memory
- Memory partition Mi has affinity to thread Th_i
  - ▲ Helps exploiting locality
  - ▲ Simple statements as SM
  - ▼ Synchronization

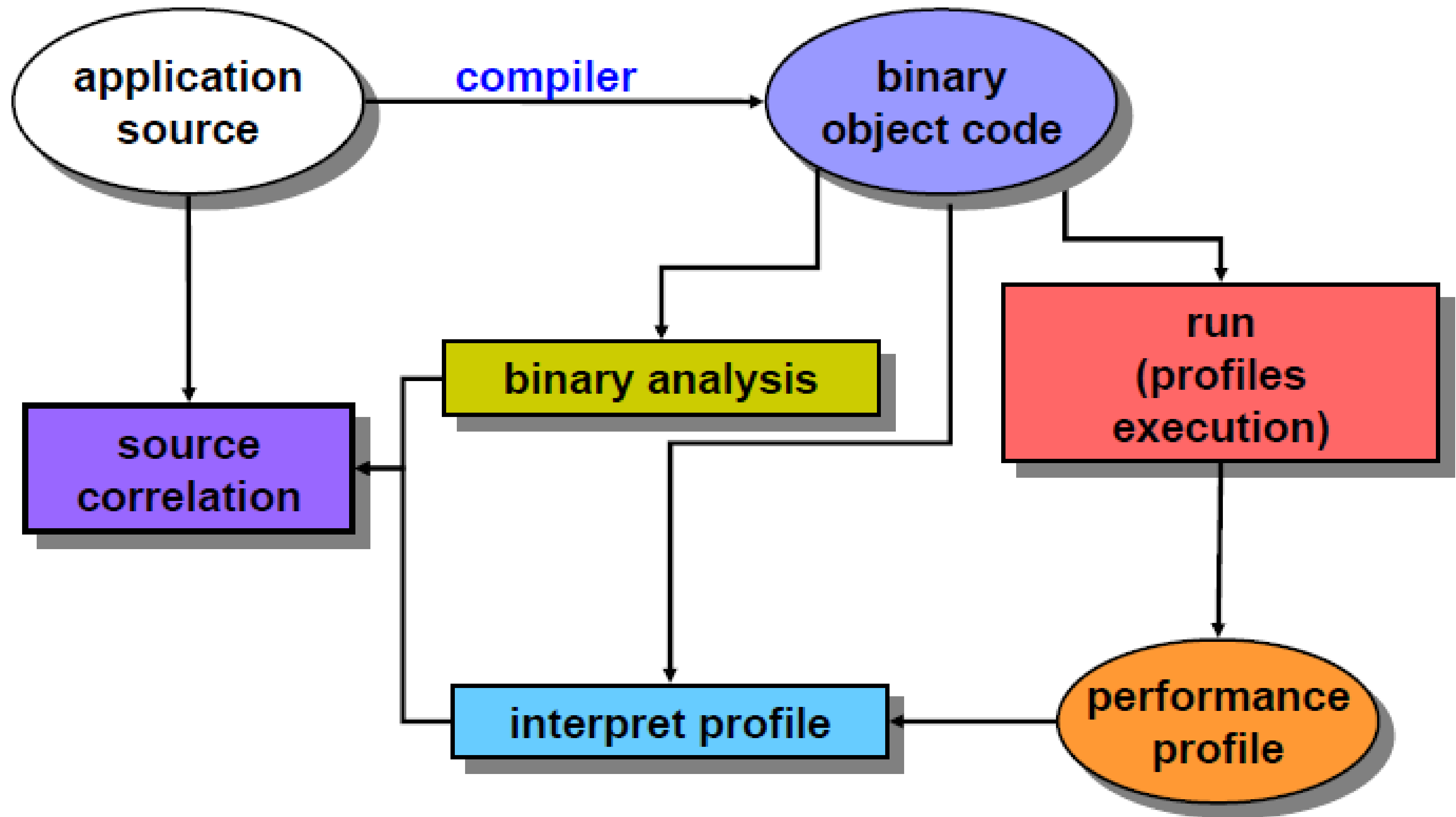
Summary

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

Program Parallelization

- Define a testing protocol
- Identify program hot spots: where is most of the time spent?
  - Look at code
  - Use profiling tools
- Parallelization
  - Start with hot spots first
  - Make sequences of small changes, each followed by testing
  - Pattern provides guidance

Common profiling workflow



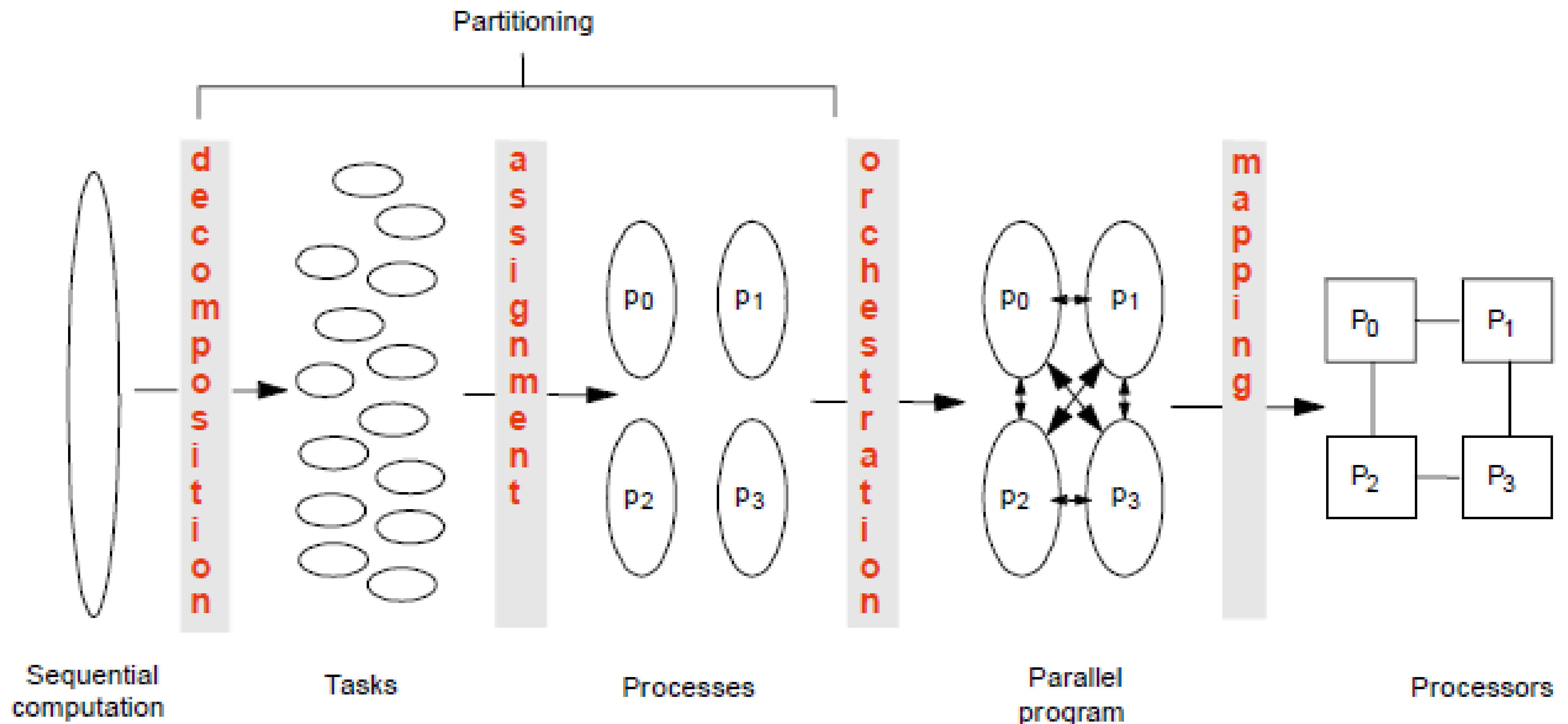
A simple example for code profiling GNU Gprof

- Useful to identify most time-consuming program parts
- These parts are good candidates for parallelization

```
gcc -pg mpeg.c -o mpeg            /* compile for instrumentation */
./mpeg <program flags>            /* run program */
gprof mpeg                        /* launch profiler */
```

```
%    cumulative   self              self      total
time   seconds   seconds    calls  ns/call   ns/call   name
90.48      0.19      0.19     7920 23989.90 23989.90   Reference_IDCT
 4.76      0.20      0.01     2148  4655.49  4655.49   Decode_MPEG1_Intra_Block
```

4 common steps to creating a parallel program



Decomposition (Amdahl's Law)

- Identify concurrency and decide at what level to exploit it

- Break upon computation into tasks to be divided among processors
  - Tasks may become available dynamically
  - Number of tasks may vary with time

- Enough tasks to keep processors busy
  - Number of tasks available at a time is upper bound on achievable speedup

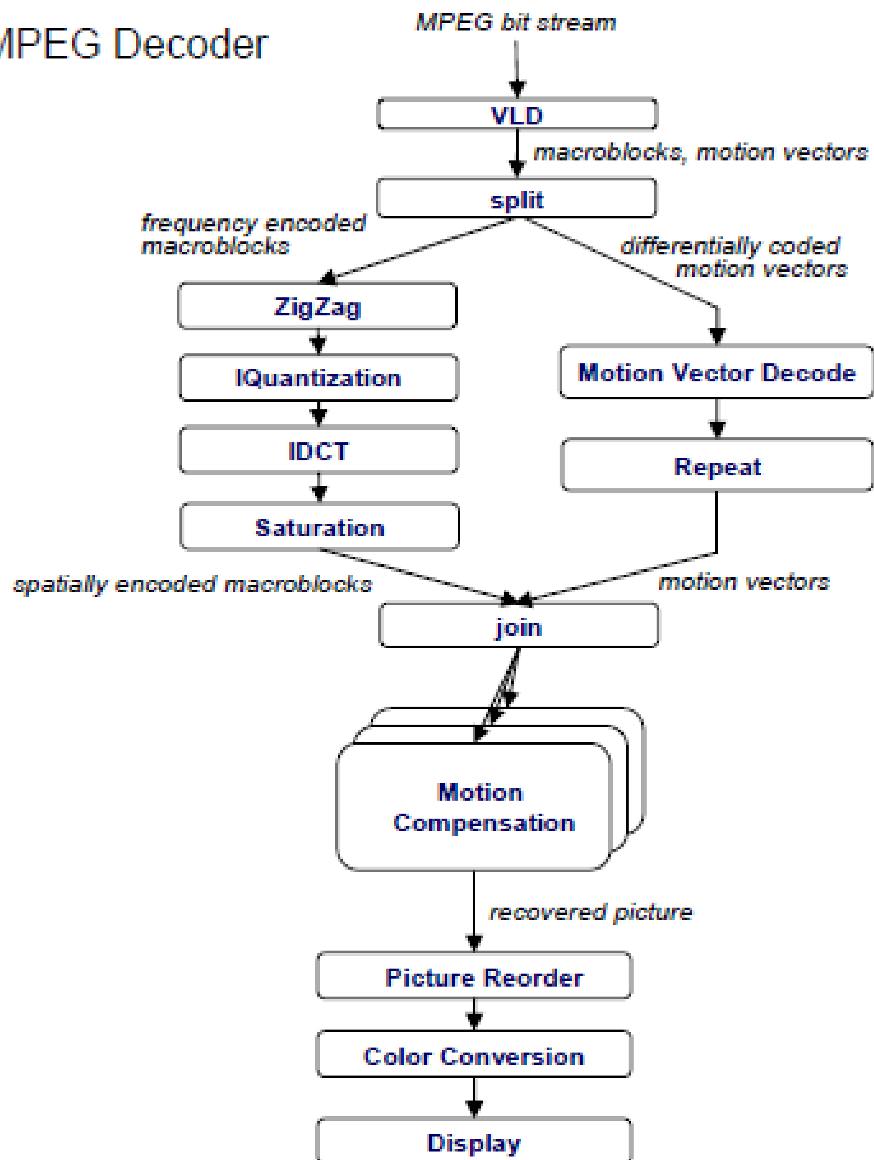
Assignment (Granularity)

- Specify mechanism to divide work among core
  - Balance work and reduce communication
- Structured approaches usually work well
  - Code inspection or understanding of application
  - Well-known design patterns
- Programmers worry about partitioning first
  - Independent of architecture or programming model
  - But complexity often affect decisions!

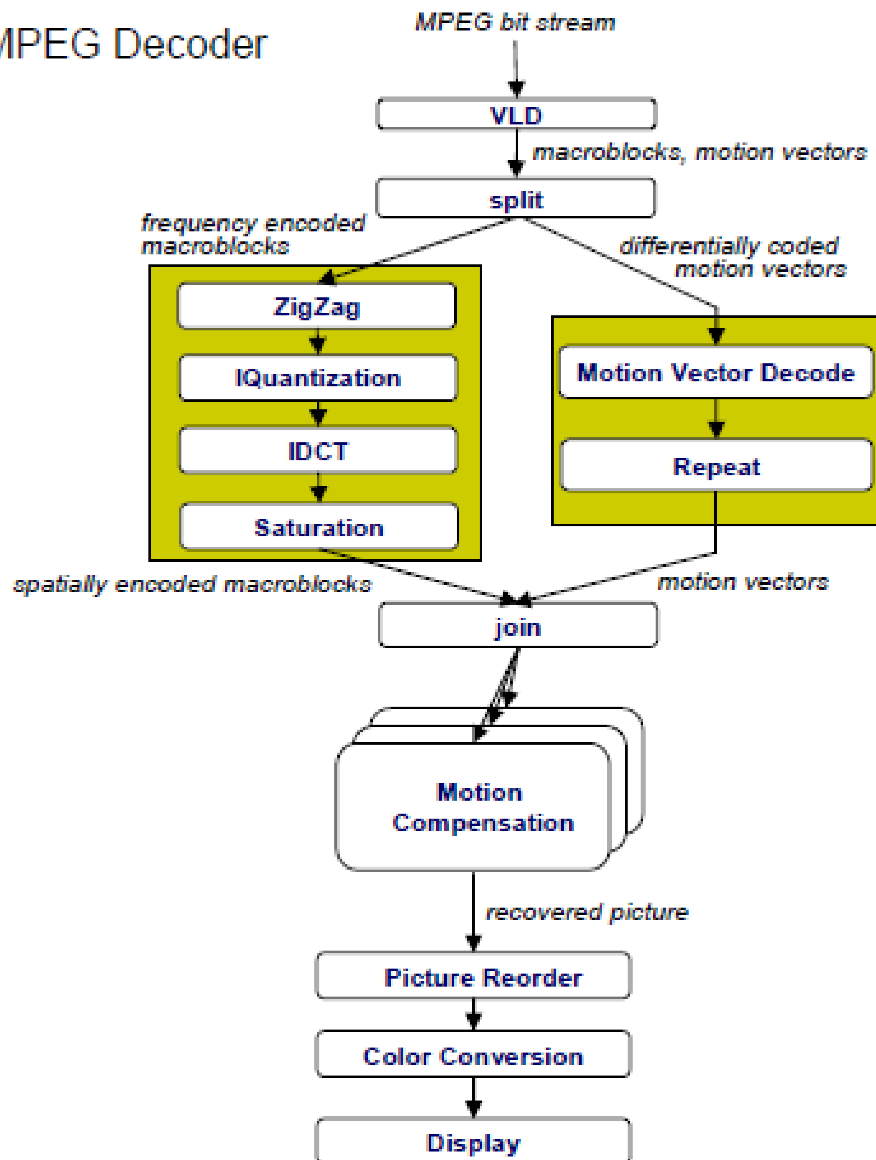
Orchestration and mapping (Locality)

- Computation and communication concurrency
- Preserve locality of data
- Schedule tasks to satisfy **dependences** early

Where's the parallelism?

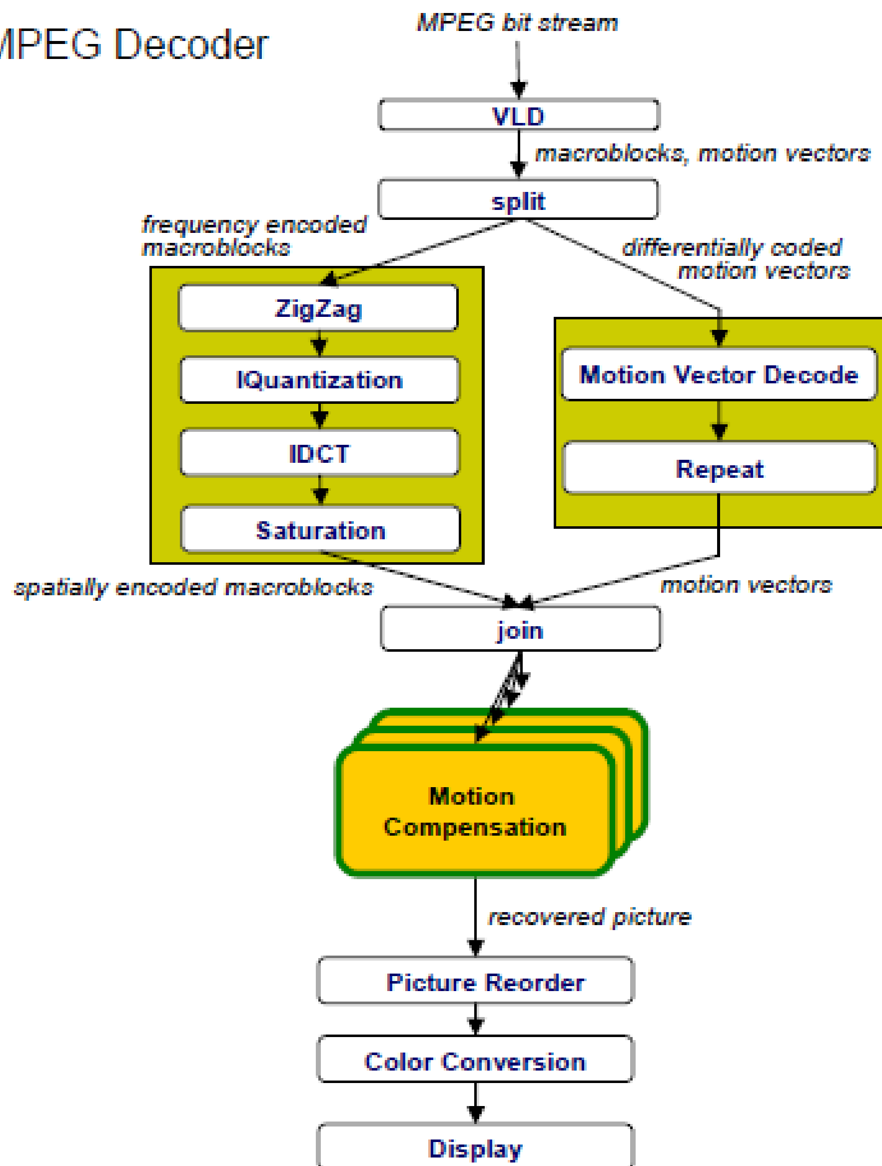


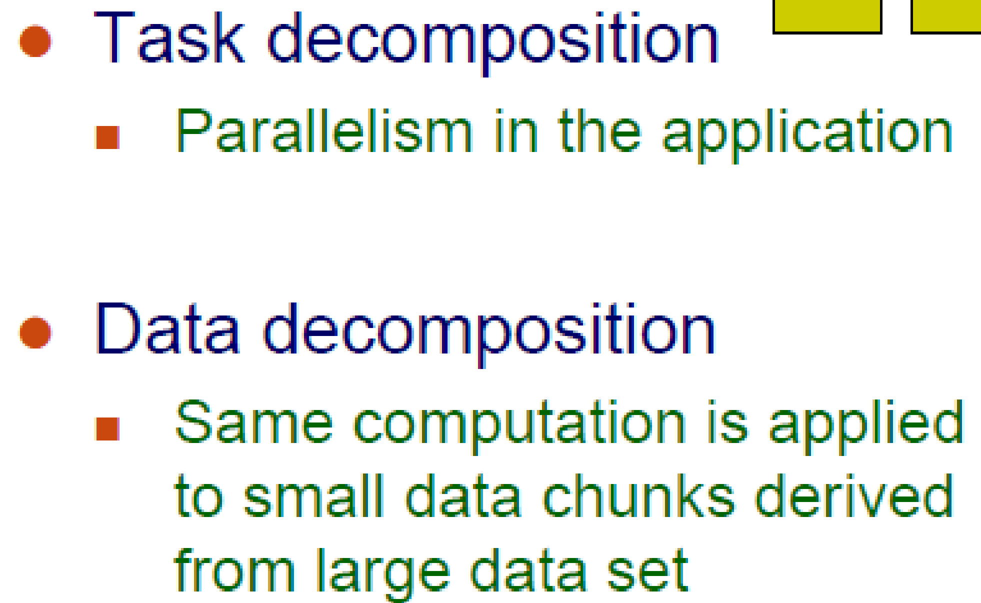
Where's the parallelism?



- Task decomposition
  - Independent coarse-grained computation
  - Inherent to algorithm
- Sequence of statements (instructions) that operate together as a group
  - Corresponds to some logical part of program
  - Usually follows from the way programmer thinks about a problem

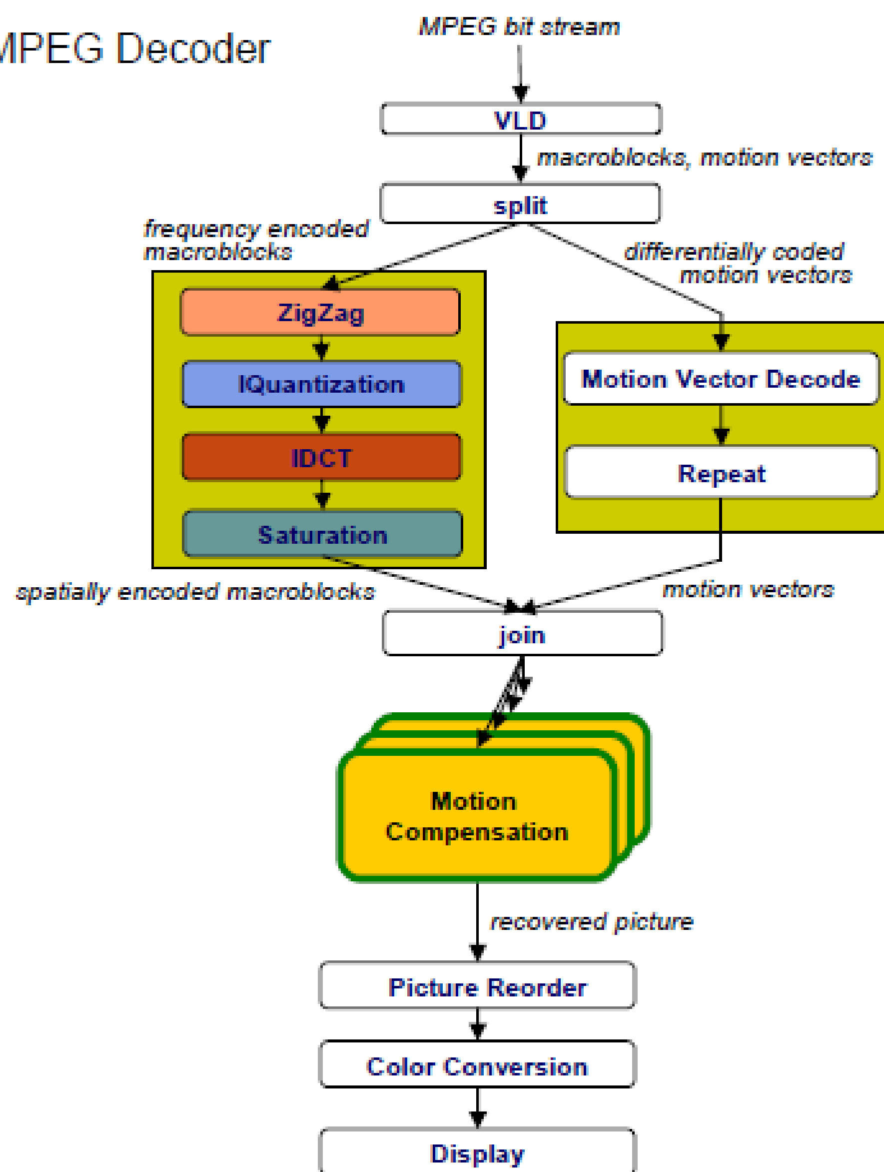
Where's the parallelism?

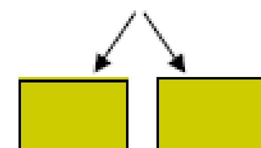
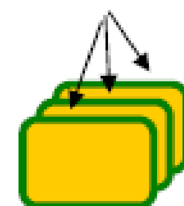


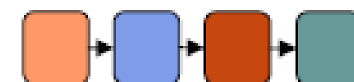


- Task decomposition
  - Parallelism in the application
- Data decomposition
  - Same computation is applied to small data chunks derived from large data set

Where's the parallelism?



- Task decomposition
  
  - Parallelism in the application
- Data decomposition
  
  - Same computation many data
- Pipeline decomposition
  - Data assembly lines
  - Producer-consumer chains



Guidelines for **TASK** decomposition

- Algorithms start with a good understanding of the problem being solved
- Programs often naturally decomposes into tasks
  - Two common decompositions are
    - Function calls and
    - Distinct loop iterations
- Easier to start with many tasks and later fuse them, rather than too few tasks and later try to split them

Guidelines for **TASK** decomposition

- Flexibility
  - Program design should afford flexibility in the number and size of tasks generated
    - Tasks should not be tied to a specific architecture
    - Fixed tasks vs. parameterized tasks
- Efficiency
  - Tasks should have enough work to amortize the cost of creating and managing them
  - Tasks should be sufficiently independent so that managing dependencies does not become the bottleneck
- Simplicity
  - The code has to remain readable and easy to understand, and debug

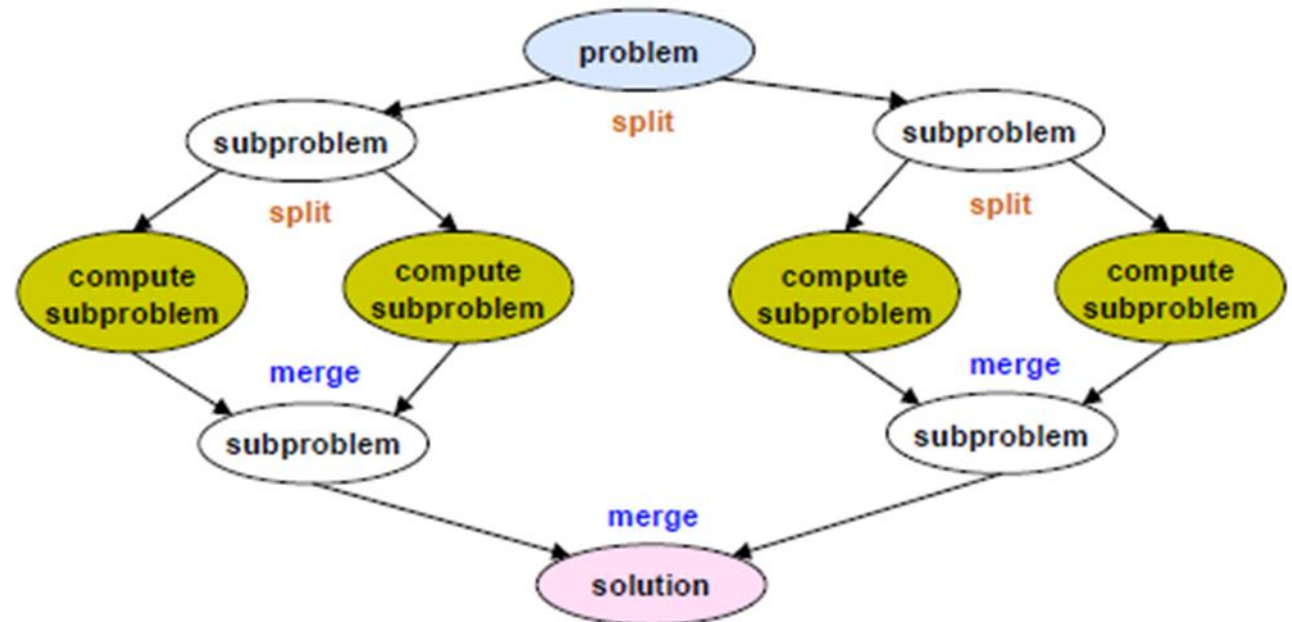
Guidelines for **DATA** decomposition

- Data decomposition is often implied by task decomposition
- Programmers need to address task and data decomposition to create a parallel program
  - Which decomposition to start with?
- Data decomposition is a good starting point when
  - Main computation is organized around manipulation of a large data structure
  - Similar operations are applied to different parts of the data structure

Common **DATA** decompositions

- Array data structures
  - Decomposition of arrays along rows, columns, blocks

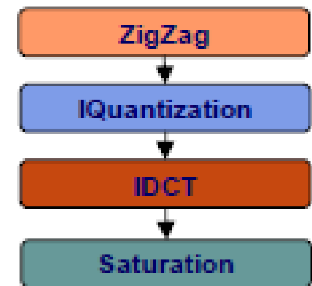
- Recursive data structures
  - Example: decomposition of trees into sub-trees



Guidelines for data decomposition

- Flexibility
  - Size and number of data chunks should support a wide range of executions
- Efficiency
  - Data chunks should generate comparable amounts of work (for load balancing)
- Simplicity
  - Complex data compositions can get difficult to manage and debug

Case for pipeline decomposition



- Data is flowing through a sequence of stages
  - Assembly line is a good analogy
- What's a prime example of pipeline decomposition in computer architecture?
  - Instruction pipeline in modern CPUs
- What's an example pipeline you may use in your UNIX shell?
  - Pipes in UNIX: cat foobar.c | grep bar | wc
- Other examples
  - Signal processing
  - Graphics

Dependence Analysis

- Given two tasks how to determine if they can safely run in parallel?

- R_i: set of memory locations read (input) by task T_i
- W_i: set of memory locations written (output) by task T_i

- Two tasks T_1 and T_2 are parallel if
  - Input to T_1 is not part of output from T_2
  - Input to T_2 is not part of output from T_1
  - Outputs from T_1 and T_2 do not overlap

Example

T$_1$

```
a = x + y
```

T$_2$

```
b = x + z
```

$R_1 = \{ x, y \}$
$W_1 = \{ a \}$

$R_2 = \{ x, z \}$
$W_2 = \{ b \}$

$$R_1 \cap W_2 = \phi$$

$$R_2 \cap W_1 = \phi$$

$$W_1 \cap W_2 = \phi$$

Common decomposition patterns

- SPMD
- Loop parallelism
- Master/Worker
- Fork/Join

SPMD pattern

- Single Program Multiple Data: create a single source code image that runs on each processor
  - Initialize
  - Obtain a unique identifier
  - Run the same program on each processor
    - Identifier and input data differentiate behavior
  - Distribute data
  - Finalize

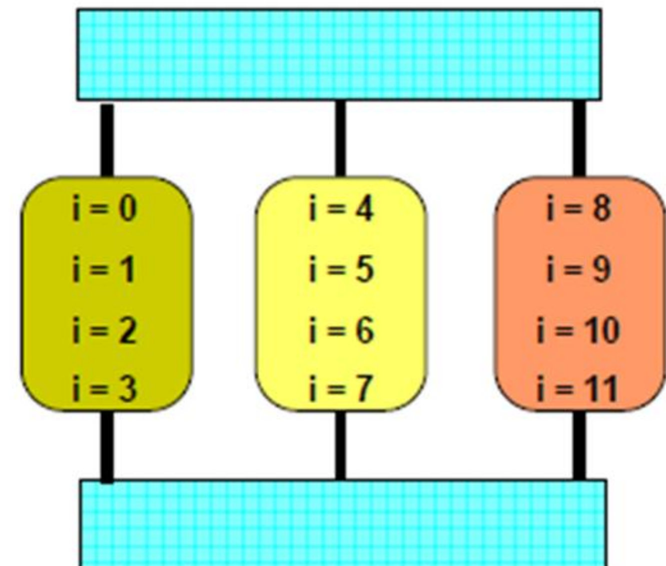
SPMD challenges

- Split data correctly
- Correctly combine the results
- Achieve an even distribution of the work
- For programs that need dynamic load balancing, an alternative pattern is more suitable

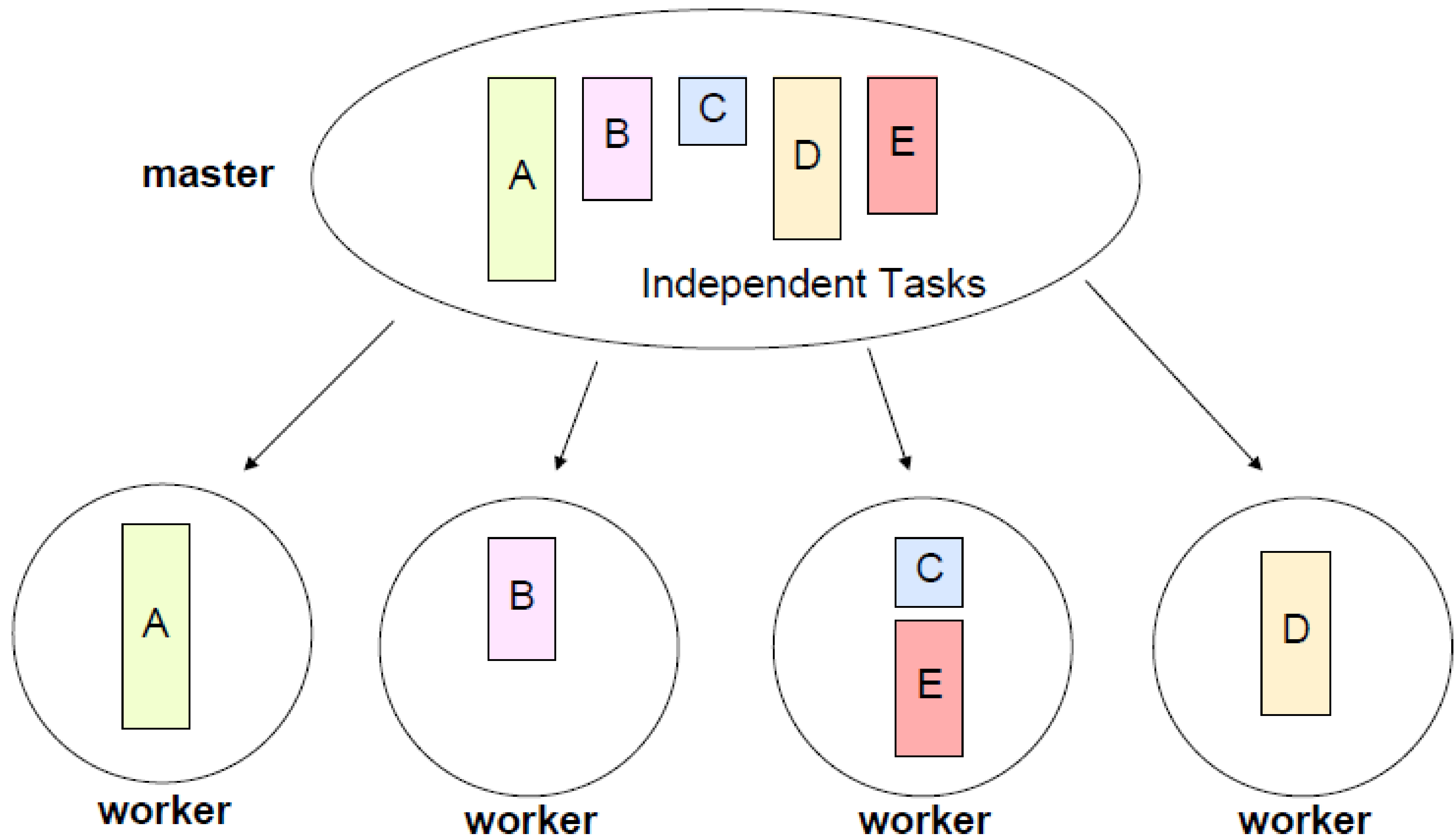
Loop parallelism pattern

- Many programs are expressed using iterative constructs
  - Programming models like OpenMP provide directives to automatically assign loop iterations to execution units
  - Especially good when code cannot be massively restructured

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



Master-Worker pattern



Master-Worker pattern

- Particularly relevant for problems using task parallelism pattern where tasks have no dependencies
  - Embarassingly parallel problems
- Main challenge in determining when the entire problem is complete

Fork-Join pattern

- Tasks are created dynamically
  - Tasks can create more tasks
- Manages tasks according to their relationship
- Parent task creates new tasks (fork) then waits until they complete (join) before continuing on with the computation