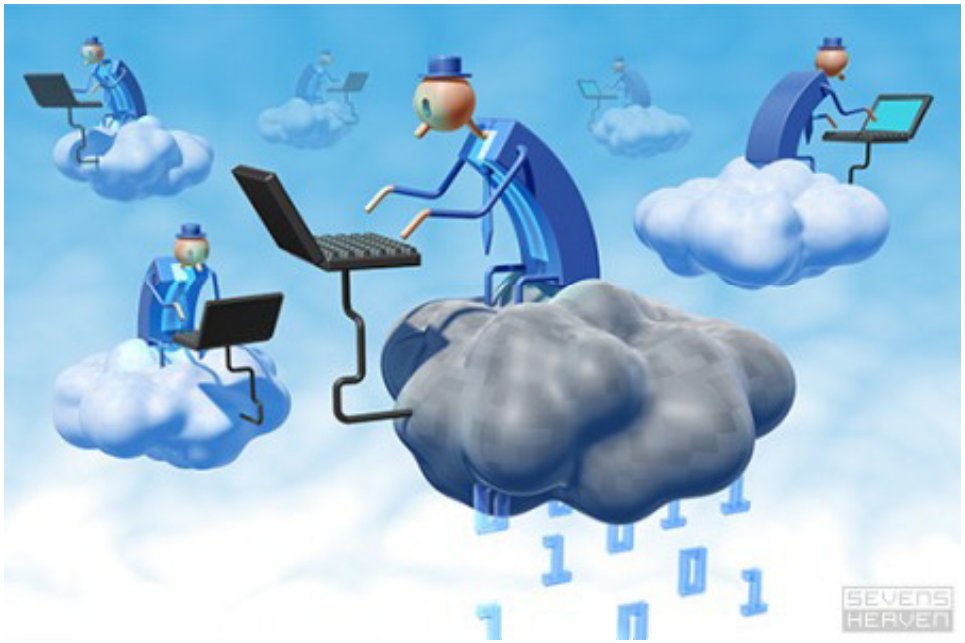
Hanoi Summer School – 20-26 June 2012

L03: Programming the Cloud



Teo Yong Meng
Department of Computer Science
National University of Singapore
Email: teoym@comp.nus.edu.sg
URL: www.comp.nus.edu.sg/~teoym

Outline

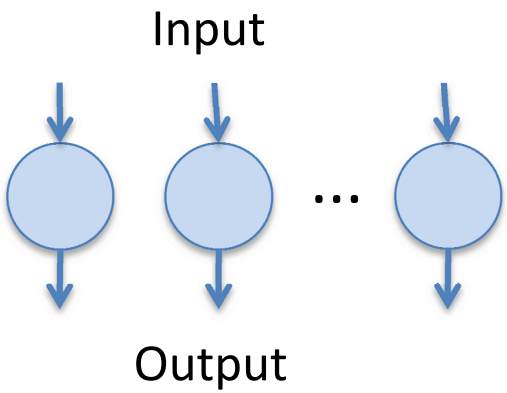
- Types of Parallel Applications
- Writing Parallel (cloud) Programs
- Parallel Programming Models
- Shared-memory Programming
  - Thread Model
  - What is OpenMP?
  - OpenMP Program to Calculate π
- Distributed-memory (message-passing) Programming
  - What is MPI?
  - MPI Program to Calculate π

Outline

- Data-intensive applications
  - What is MapReduce?
  - What is Hadoop?
  - MapReduce Framework
  - Structure of a MapReduce Program
  - High-level View of MapReduce
  - Example: Counting Words
  - Parallelism in MapReduce
  - Applications of MapReduce
- Comparison with Traditional Models
- Summary
- References

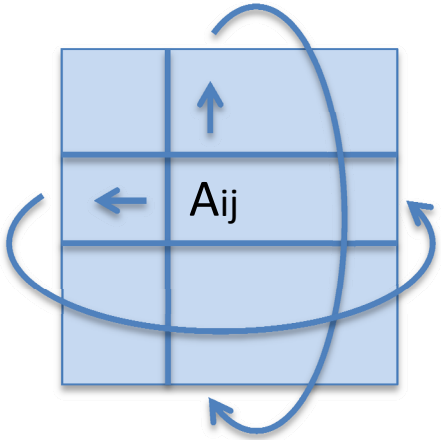
Types of Parallel Applications

Embarrassingly Parallel Applications



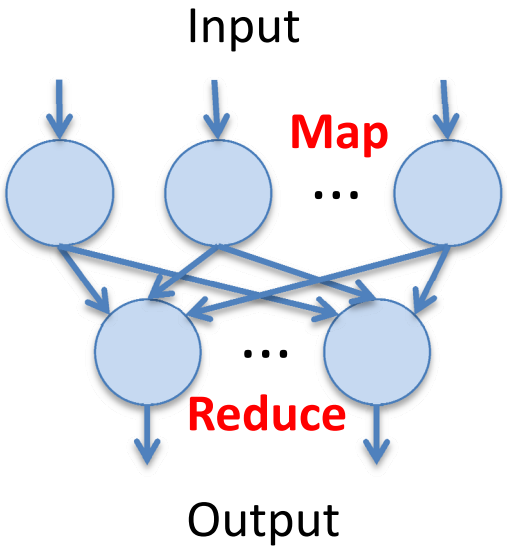
Lots of computation, not much data sharing

Matrix-based Applications



Lots of computation, some data sharing (send/recv)

Data-intensive Parallel Applications

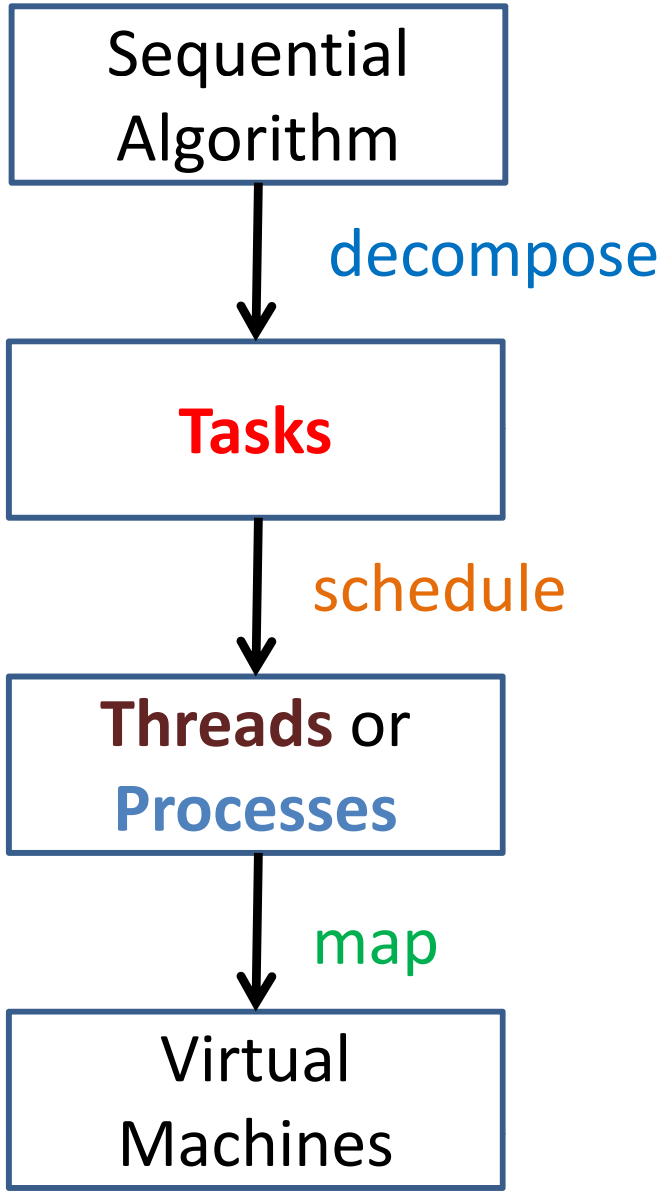


Lots of data to be processed

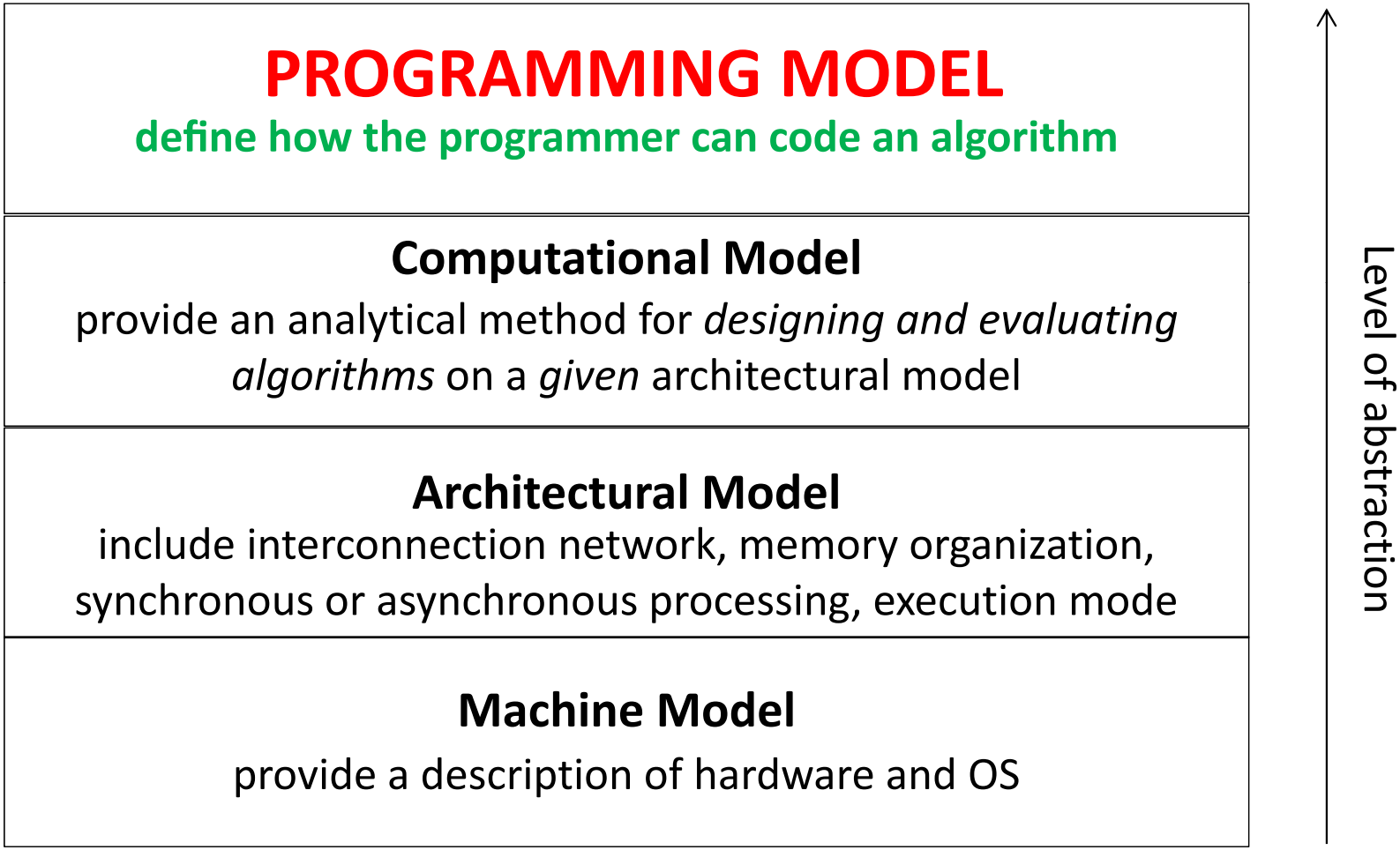
Writing Parallel (Cloud) Programs

Three main steps:

1. Decomposition of the computations
2. Scheduling (assignment of tasks to processes (or threads))
3. Mapping of processes (or threads) to virtual machines (physical processors or cores)



MODELS OF PARALLEL SYSTEMS

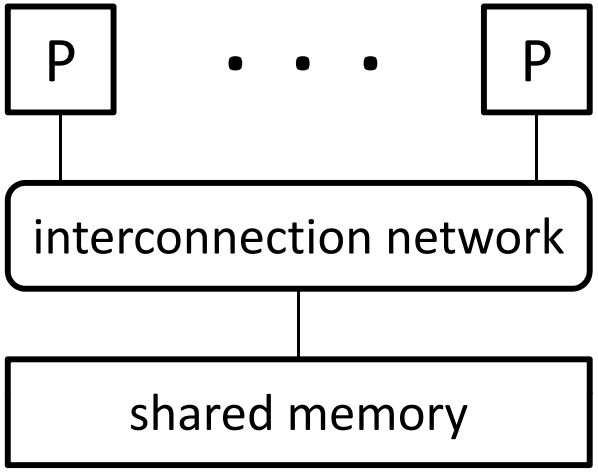


Parallel Programming Models

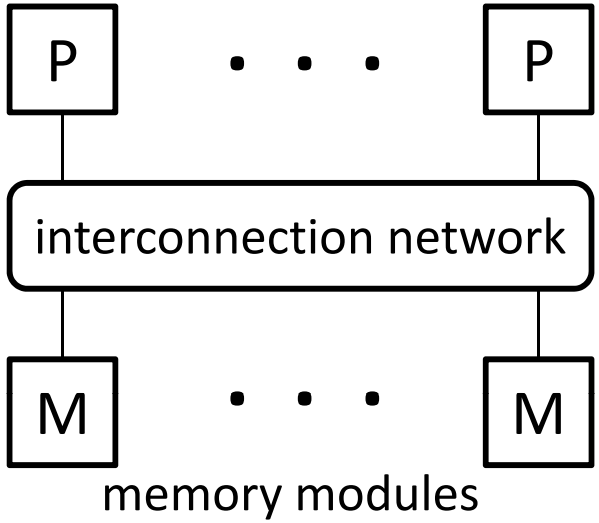
- **Abstraction** above the hardware and memory architecture that provide a mechanism to ***specify parallel programs*** using a programming language

- Differ by several criteria:
  - level of parallelism - instruction level, statement level, procedural level, or parallel loops
  - implicit or user-defined explicit specification of parallelism
  - how parallel program parts are specified, i.e., processes or threads
  - communication mode, i.e., message-passing or shared variables
  - …

Shared-Memory Systems (Multiprocessors)



Abstract View

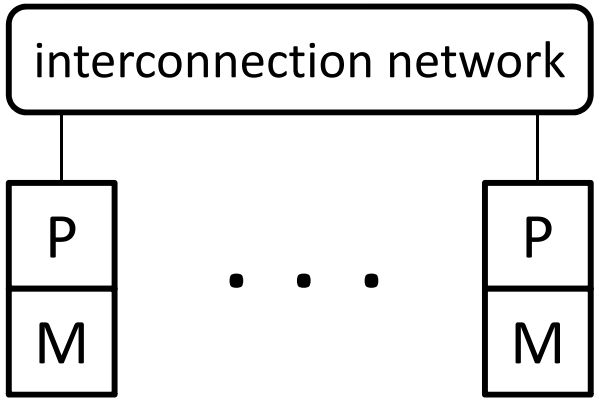


Implementation View

?

- Global memory - a common shared address space
- Data exchanges: **shared variables** (programmed using threads)
- Problems
  - Race conditions
  - Fast access to the global memory for each processor

Distributed-memory Systems (Multicomputers)



P = processor

M = local memory

Abstract View

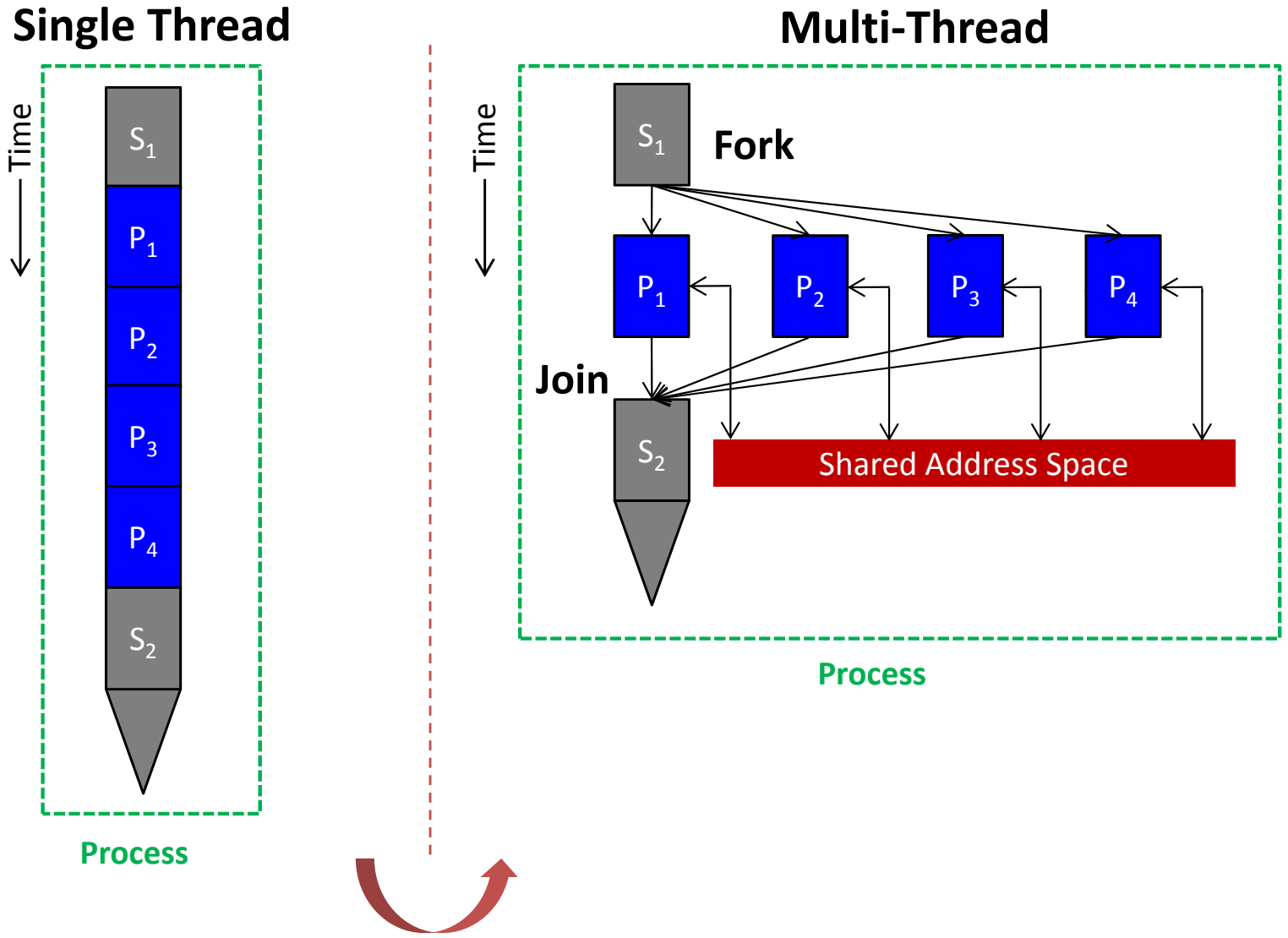
- A **node** is an independent unit, consisting of processor, memory and, sometimes, periphery elements.
- Physically distributed memory -> memory in a node is private
- Data exchanges between nodes -> **message-passing**

SHARED-MEMORY PROGRAMMING

- Thread Model
- What is OpenMP?
- OpenMP Program to Calculate π

Shared-memory Model

S_i = *Serial*
P_j = *Parallel*



Shared-memory Programming Model

- Threads (processes) share a common address space, read and write are asynchronous
- Control access to the shared-memory using as locks /semaphores
- Simplified program development - no need to specify explicitly the communication of data between threads
- But more difficult to understand and manage data locality – beyond the control of the average users
- When multiple processors use the same data, keep data local to the processor that works on it to conserve memory accesses, cache refreshed and bus traffic

Thread Model

- A single program has multiple, concurrent execution paths
- The main program performs some serial work, and then creates a number of tasks (**threads**) that are scheduled and run by the operating system concurrently
- Each thread has local data, but also, shares the entire resources of the main program, including the memory space
- Threads communicate with each other through the global shared memory
- Threads can be created and destroyed, while the main program remains present to provide the necessary shared resources

What is OpenMP?

- Portable standard for the programming of shared memory systems

- API consists of:
  - compiler directives (i.e. pragmas)
  - library routines (i.e. function calls)
  - environmental variables (i.e. shell variables)

- Source code can be translated into multithreaded code or sequential code

Programming Model

- Based on cooperating threads running simultaneously on multiple processors or cores
- Threads are created and destroyed in a fork–join pattern
- An OpenMP program begins with an initial thread, which executes the program sequentially until a first parallel construct is encountered
- Initial thread creates a pool of threads and becomes the master thread of the pool

OpenMP Compiler Directives

- Parallelism is controlled by compiler directive

  #pragma omp directive [clauses [ ] ...]

- Directives are case sensitive and apply only to the next code line

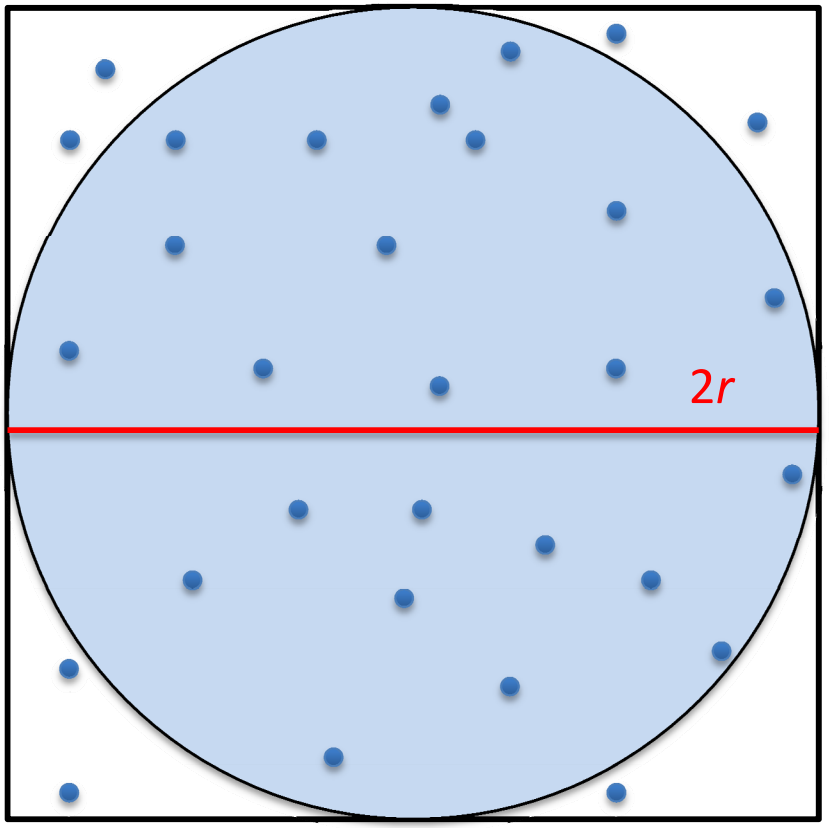
- Clauses are optional

Example: Calculate π

Steps:

1. Inscribe a circle in a square
2. Take random points in the square
3. Determine how many of those points are also in the circle
4. Given $y = \frac{\text{points}_{circle}}{\text{points}_{square}}$ then $\pi \approx 4y$

**** More points used result in better approximation*



$$A_{circle} = \pi r^2$$
$$A_{square} = (2r)^2 = 4r^2$$
$$\pi = 4\frac{A_{circle}}{A_{square}}$$

Serial C Program to Calculate π

```
int main(int argc, char* argv[]) {

        int npoints = 1000000, ncircle = 0;
        double x, y, PI;
        int i, seed = time(NULL);


        for (i=0; i < npoints; i++) {
              x = ((double)rand r(&seed))/RAND MAX;
              y = ((double)rand_r(&seed))/RAND_MAX;
              if (x*x + y*y <= 1) ncircle++;
        }

        PI = 4.0 * ncircle / npoints;
        printf("pi=%.10f\n", PI);
        return 0;
}
```

loop executes 1 **million** times

decompose loop in many smaller loops

OpenMP Program to Calculate π

```
int main(int argc, char* argv[]) {
    #pragma omp parallel num_threads(1000);
        int npoints = 1000000, ncircle = 0;
        double x, y, PI;
        int i, seed = time(NULL);

    #pragma omp parallel for reduction(+: ncircle)
        for (i=0; i < npoints; i++) {
                x = ((double)rand r(&seed))/RAND MAX;
                y = ((double)rand_r(&seed))/RAND_MAX;
                if (x*x + y*y <= 1) ncircle++;
        }

        PI = 4.0 * ncircle / npoints;
        printf("pi=%.10f\n", PI);
        return 0;
}
```

with 1,000 workers, loops now execute 1 **thousand** times

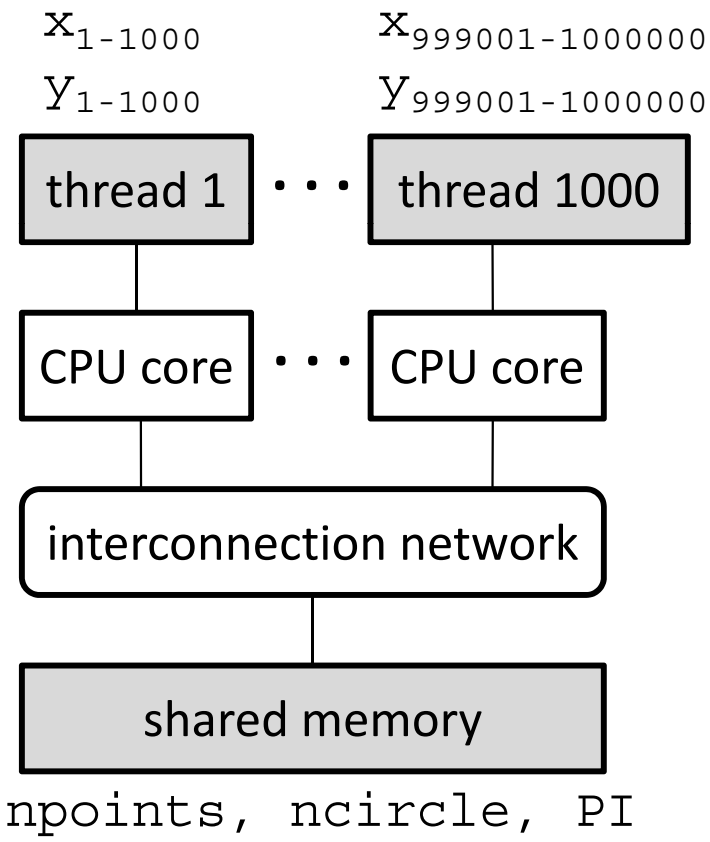
OpenMP Program to Calculate π

All Threads

```
x = ((double)rand_r(&seed))/RAND_MAX;
y = ((double)rand_r(&seed))/RAND_MAX;
if (x*x + y*y <= 1) ncircle++;
```

Master Thread

```
PI = 4.0 * ncircle / npoints;
printf("pi=%.10f\n", PI);
```



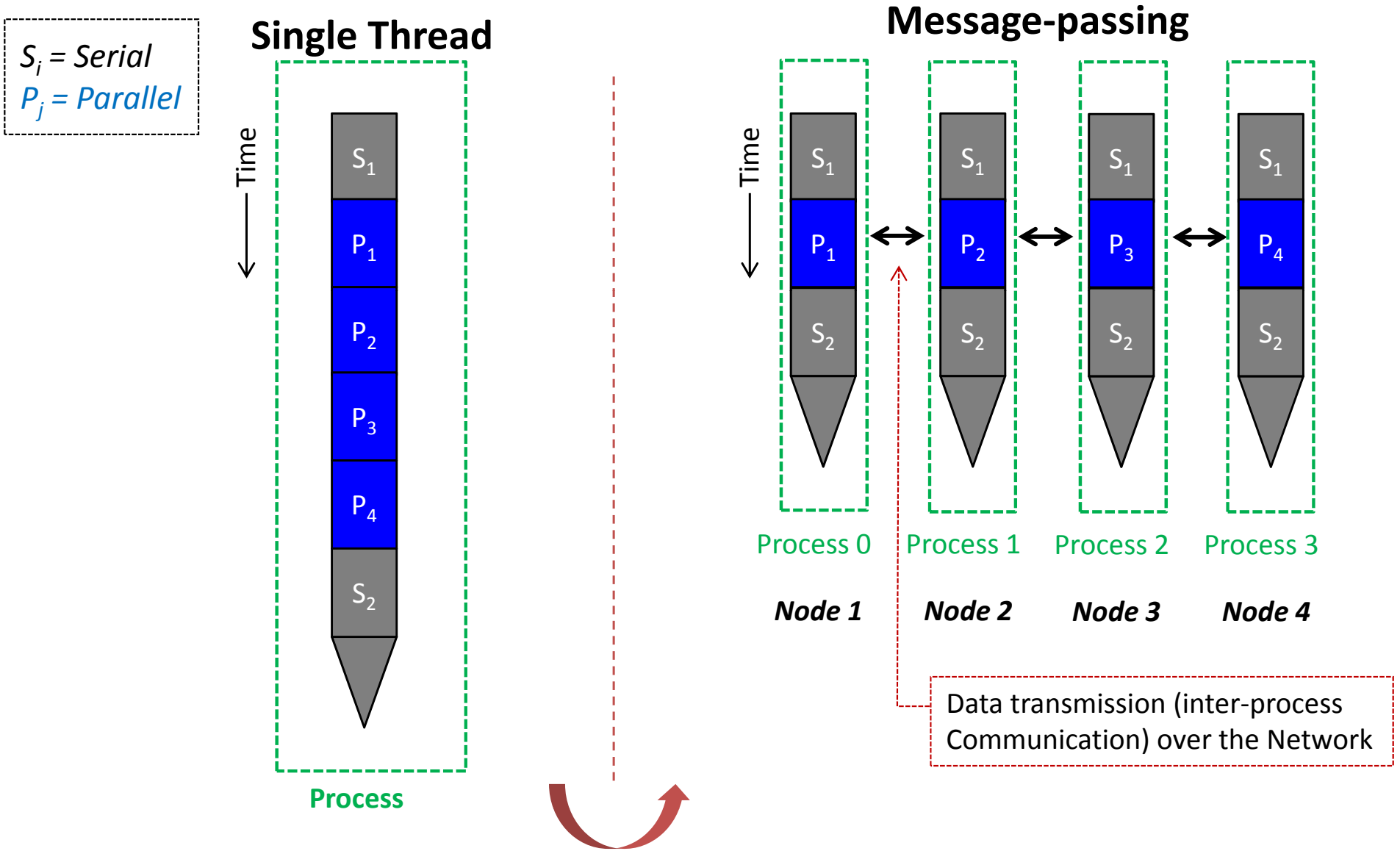
Useful OpenMP directives

Directive	Description
#pragma omp parallel	Defines a parallel region to be run by multiple threads in parallel
#pragma omp atomic	Update memory atomically, don't expose to multiple threads
#pragma omp for	Iterative for-loop, iterations are run in parallel
#pragma omp parallel for	Parallel region containing a single **for** directive
#pragma omp ordered	Execute code in sequential order
#pragma omp critical	Execute code a single thread at a time
#pragma omp barrier	Synchronizes all threads in a parallel region
#pragma omp single	Code is run by only one thread (any)
#pragma omp master	Code is run only by master thread

DISTRIBUTED-MEMORY (MESSAGE-PASSING) PROGRAMMING

- What is MPI?
- MPI Program to Calculate π

Message-passing Model



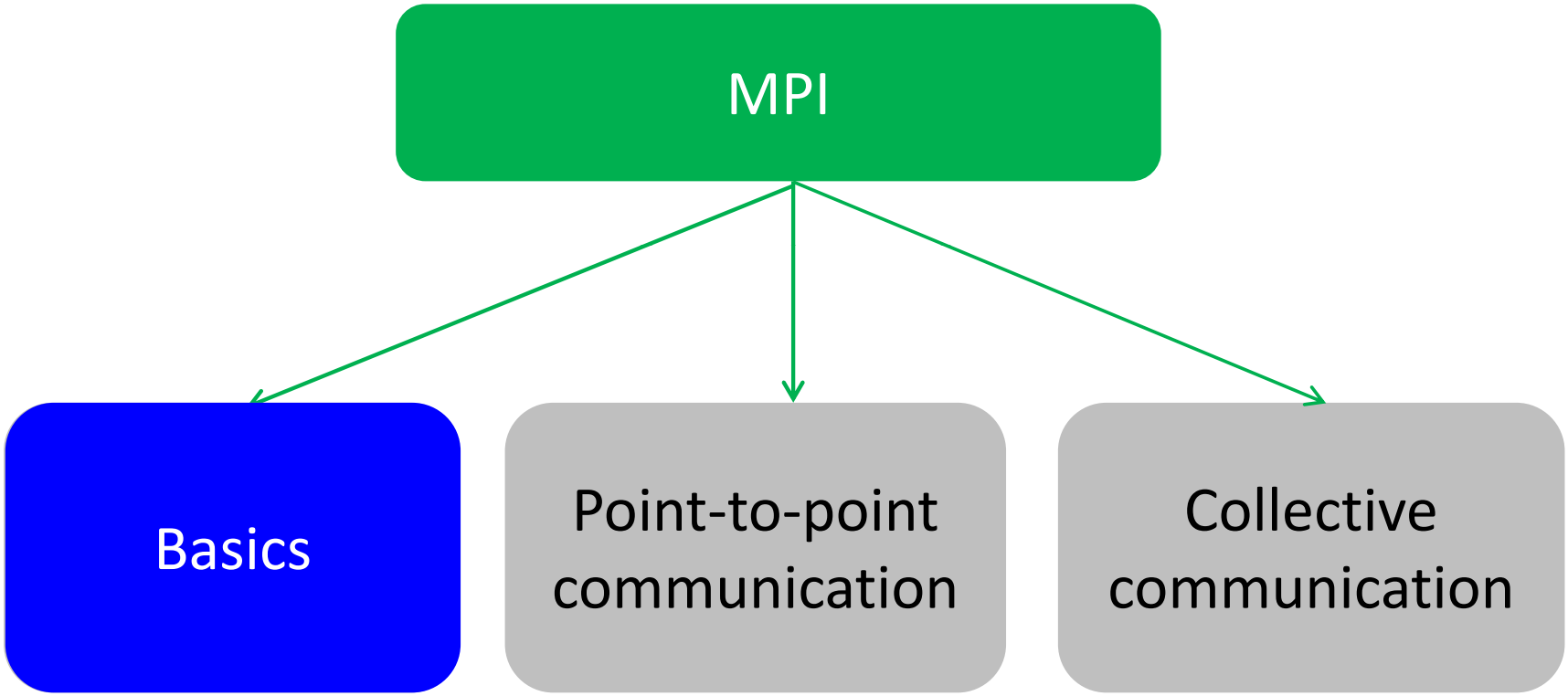
Programming Model

- Processes residing on the same machine and/or across any number of machines use their own local memory during computation
- Data exchange between tasks happens by *sending* and *receiving* messages
- Communication usually require cooperative operations to be performed by each task, e.g., a *send* operation *must have* a matching *receive* operation
- MPI (Message Passing Interface) library is now the “de facto” standard for message passing

What is MPI?

- A message passing library standard for writing message passing programs
- Goal of MPI is to establish a *portable*, *efficient*, and *flexible* standard for message passing
- By itself, MPI is NOT a library but the specification of what such a library should be
- Not an IEEE or ISO standard but has becomes the *industry standard* for writing message-passing programs
- The programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

Message-passing Interface



Reasons for using MPI

Reason	Description
Standardization	MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms
Portability	There is no need to modify your source code when you port your application to a different platform that supports the MPI standard
Performance Opportunities	Vendor implementations should be able to exploit native hardware features to optimize performance
Functionality	Over 115 routines are defined
Availability	A variety of implementations are available, both vendor and public domain

MPI Program to Calculate π

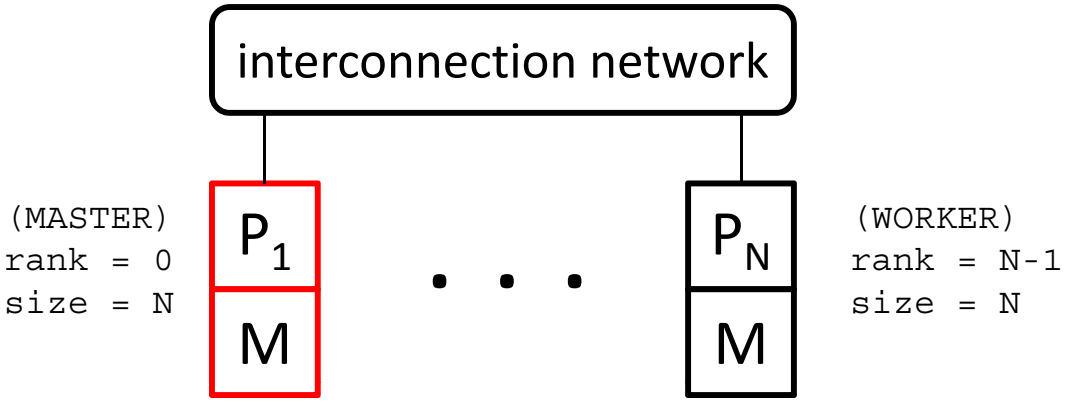
```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char* argv[]) {
    MPI_Init (&argc, &argv);
    int rank, size;
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    for (i=0; i < 1000000/size; i++) {
          x = ((double)rand_r(&rank))/RAND_MAX;
          y = ((double)rand_r(&rank))/RAND_MAX;
          if (x*x + y*y <= 1) ncircle++;
    }
    if (rank != 0) {     /* WORKER NODE CODE */
          MPI_Send (&ncircle, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    } else {             /* MASTER NODE (rank=0) CODE */
          for (w=1; w < size; w++) {
              int worker_result;
              MPI_Recv (&worker_result, 1, MPI_INT, w, 1, MPI_COMM_WORLD);
              ncircle += worker_result;
          }
          PI = 4.0 * ncircle / npoints; /* MASTER NODE COMPUTES PI */
          printf("pi=%.10f\n", PI);
    }
    MPI_Finalize ();
}
```

each node determines its own **rank** and the **size** of the system

all worker nodes send partial results to master node

Calculate π – MPI Send/Receive



$ncircle_0$
+
$worker\ result_1$
+
...
+
$worker\_result_N$

```
for (i=0; i < 1000000/size; i++) {
    x = ((double)rand_r(&rank))/RAND_MAX;
    y = ((double)rand_r(&rank))/RAND_MAX;
    if (x*x + y*y <= 1) ncircle++;
}
ncircle
```

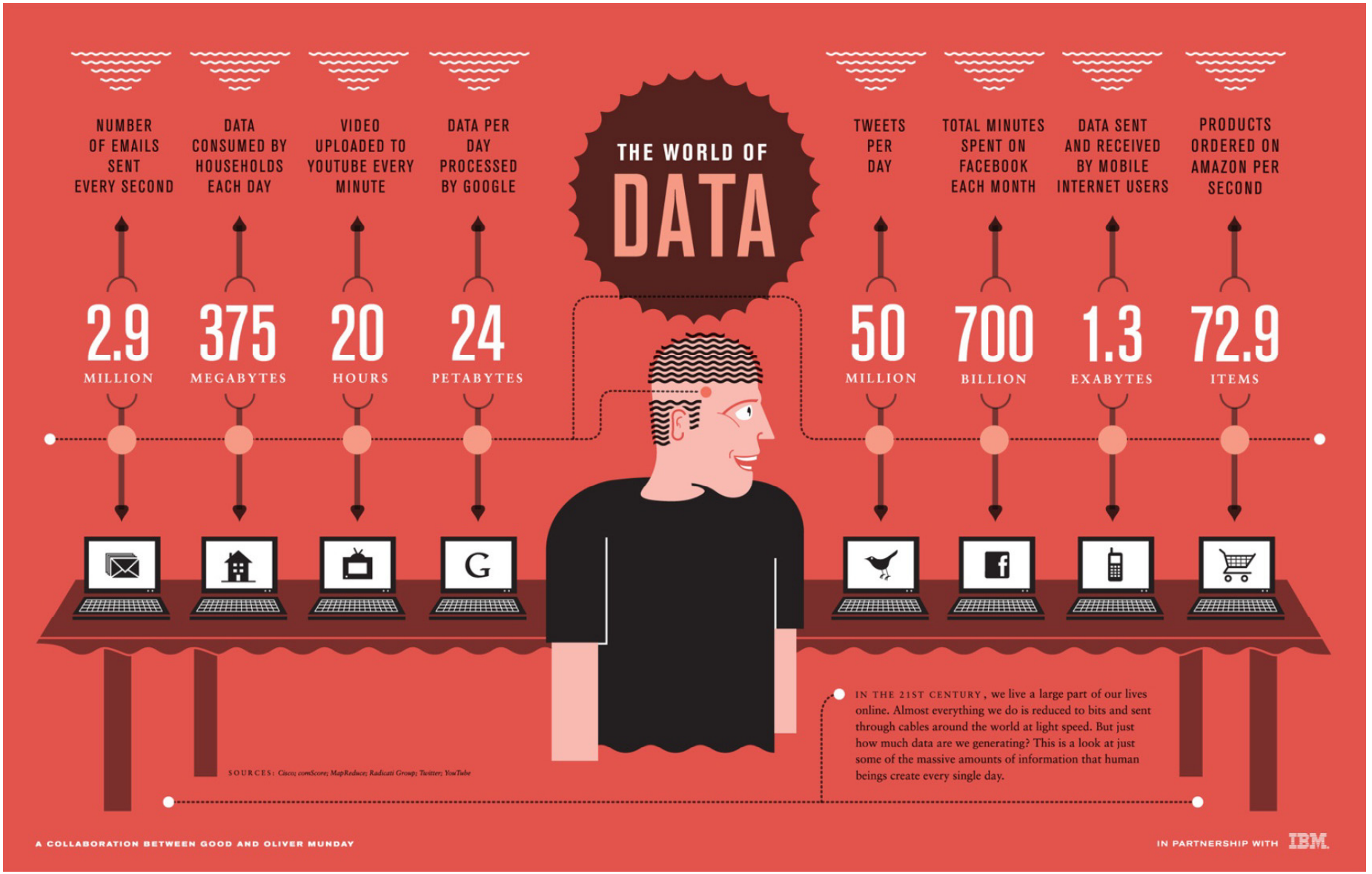
```
MPI_Recv (&worker_result,
          1, MPI_INT,
          w, /* WORKER RANK */
          1, /* TAG ID */
          MPI_COMM_WORLD);
```

```
MPI_Send (&ncircle,
          1, MPI_INT,
          0, /* MASTER RANK */
          1, /* TAG ID */
          MPI_COMM_WORLD);
```

Useful MPI functions

MPI function	Description
MPI_Init	Initialize MPI
MPI_Finalize	Exit MPI
MPI_Comm_rank	Determine processor rank
MPI_Comm_size	Determine number of processes
MPI_Send	Point-to-point message send between processors
MPI_Recv	Point-to-point message receive between processors
MPI_Scatter	Divide array and send each processor a part of it
MPI_Gather	Concatenate arrays from all processors into a large array
MPI_Bcast	Broadcast message from master to all processors

Data-intensive Applications



The World of Data we're Creating on the Internet, Oct 2010

DATA-INTENSIVE APPLICATIONS

- What is MapReduce?
- What is Hadoop?
- MapReduce Framework
- Structure of a MapReduce Program
- High-level View of MapReduce
- Example: Counting Words
- Parallelism in MapReduce
- Applications of MapReduce

What is MapReduce?

- A programming model for data processing
- Supports distributed computing on large data sets on multiple machines (clusters, public or private clouds)
- Ability to scale to 100s or 1000s of computers, each with several processor cores
- How large an amount of work?
  - Web-scale data on the order of 100s of GBs to TBs or PBs
  - Input data set will not likely fit on a single computer's hard drive
  - A distributed file system (e.g., Google File System- GFS) is typically required

What is Hadoop?

- MapReduce has frequently been associated with *Hadoop*
- An *open source* and popular implementation of MapReduce
- Presents MapReduce as an analytics engine with a distributed storage layer referred to as Hadoop Distributed File System (*HDFS*)
- HDFS mimics Google File System (*GFS*)

MapReduce Framework

- Inspired by the **map** and **reduce** functions used in functional programming languages
- Hides the details of parallelization, data distribution, load balancing, fault tolerance (system automatically adapts to the number of cores available)
- Closed source developed by Google (2004) to process large amounts of raw data
- Open source (Hadoop) developed by Apache + Yahoo (Java programming language)
- Amazon Elastic MapReduce creates a Hadoop cluster and handles data transfers between Amazon EC2 (computation) and Amazon S3 (storage)

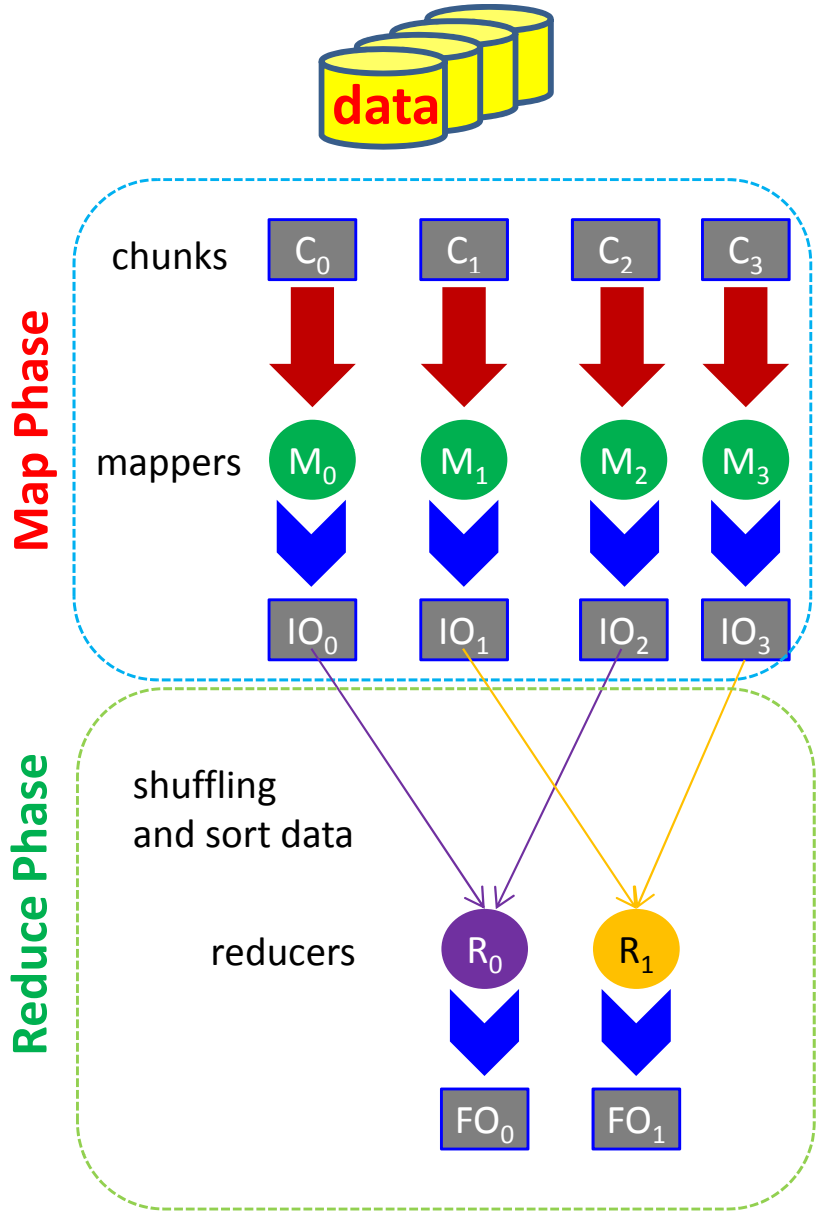
Structure of a MapReduce Program

1. Read (a lot of) data
2. MAP (extract data you need from each record)
3. Shuffle and Sort data
4. REDUCE (aggregate, summarize, filter, transform extracted data)
5. Write the results

Programmers implement the MAP and REDUCE functions to fit their problem

High-level View of MapReduce

- breaks the data flow into two ***map*** and ***reduce*** *phases*
- Chunks are processed in isolation by tasks called *Mappers*
- Outputs from mappers, called intermediate outputs (IOs), are brought into a second set of tasks called *Reducers*
- Process of bringing together IOs into a set of Reducers is called *shuffling*
- Reducers produce the final outputs (FOs)



Example: Word Counting

Count how many times each word appears in a set of documents

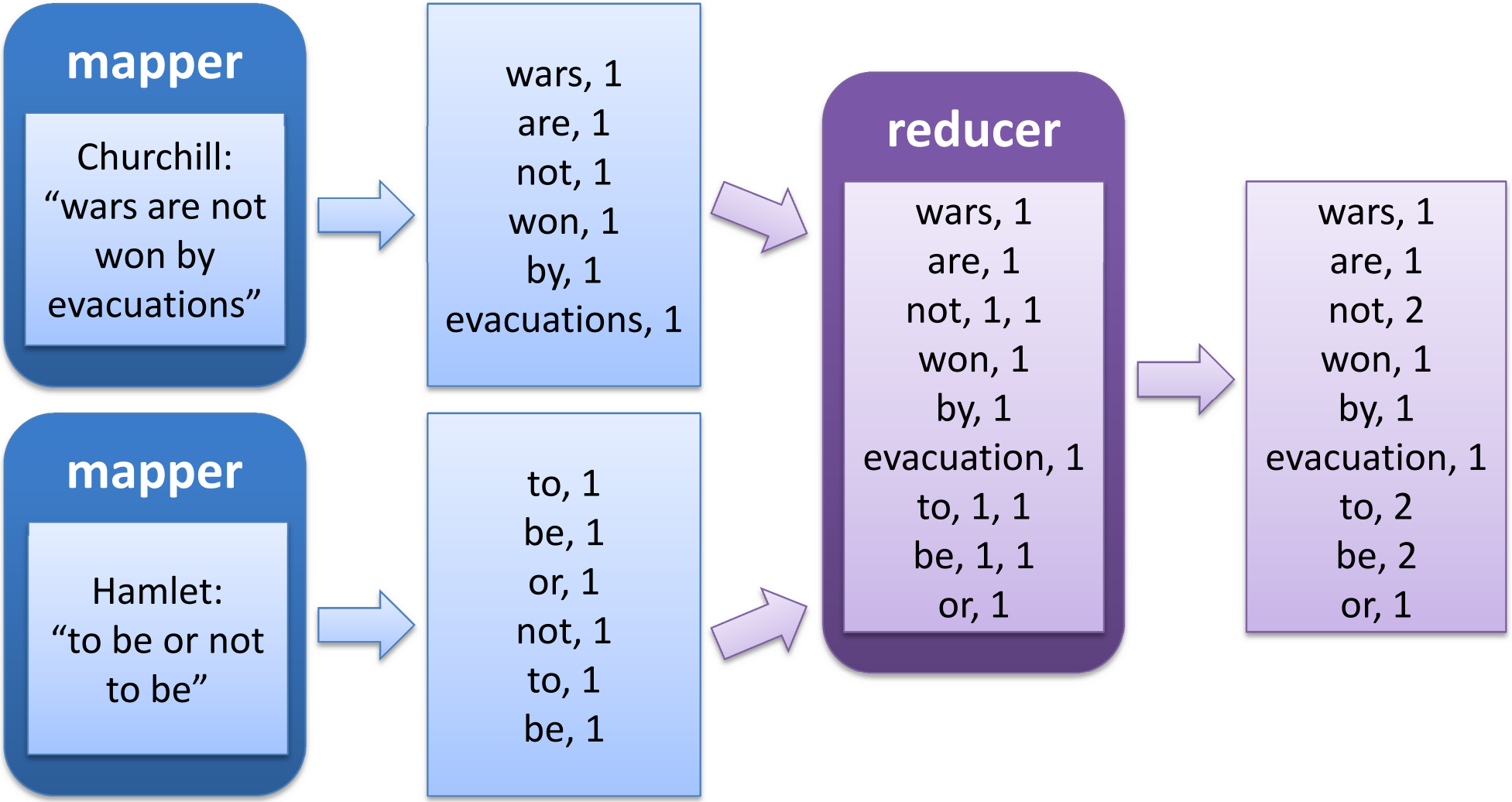
```
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
        StringTokenizer itr = new
StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
        }
}
```

key = file name
value = file contents
write = signal word occurrence

```
public void reduce(Text key, Iterable<IntWritable>
values, Context context) throws IOException,
InterruptedException {
        int sum = 0;
        for (IntWritable val : values)
                sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
}
```

key = word
values = map signals
write = signal total word occurrences

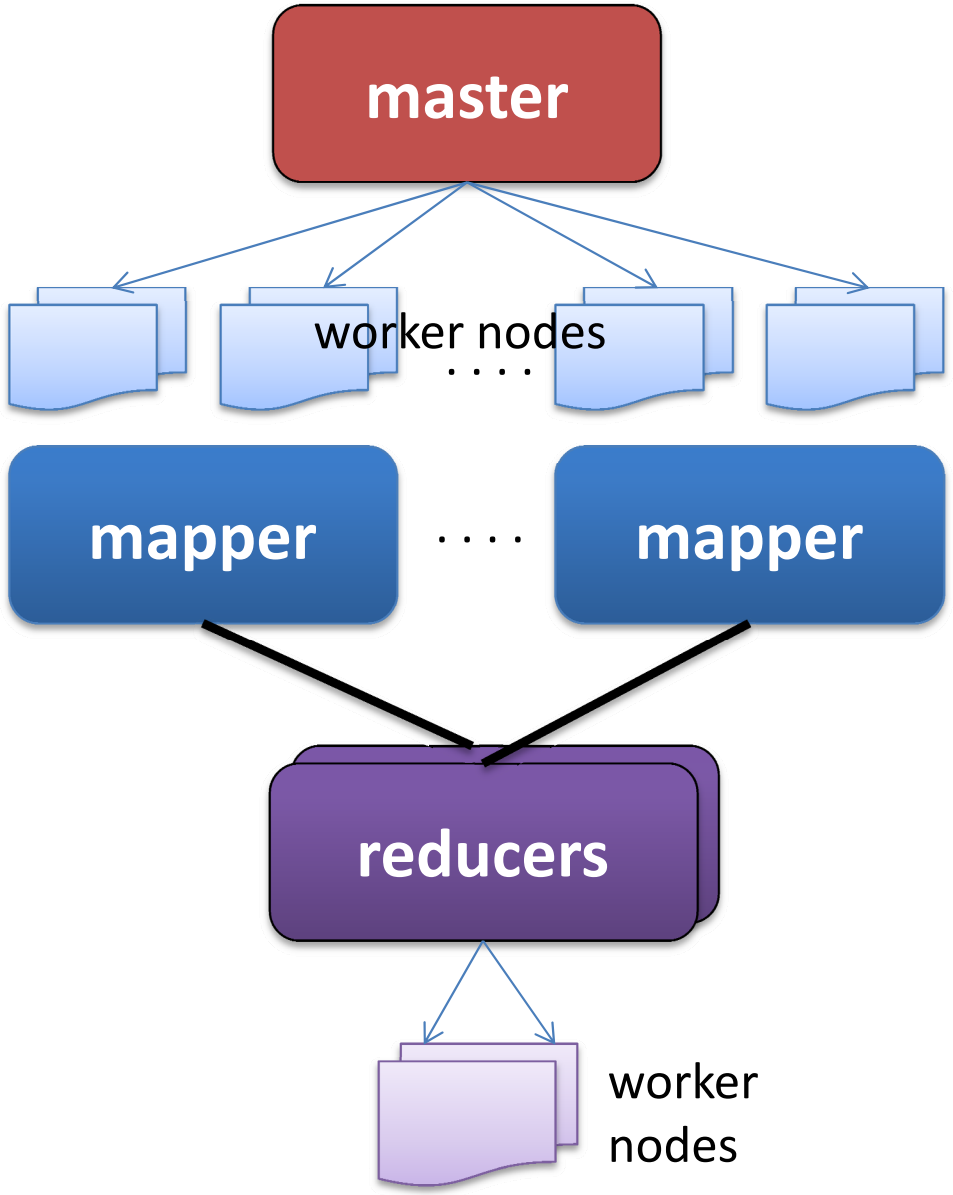
Example: Word Counting*



* for simplicity, we left out the sort phase

Parallelism in MapReduce

- Master program creates data-localized tasks to minimize network communication
- Worker nodes switch between mapper and reducer
- **Map** workers run in parallel
  - each has its own data set
  - each creates intermediate local values from each input data set
- **Reduce** workers run in parallel
  - may reduce intermediate data
  - there is an optimal number of reducers for best performance



Applications of MapReduce

- Indexing for Google search
- Organizing articles in Google news
- Performing various user statistics

- Mapping Yahoo search
- Detecting spam in Yahoo mail

- Data Mining
- Optimizing ads for users
- Detecting spam on Facebook

Comparison with Traditional Models

Aspect	Shared-memory	Message-passing	MapReduce
Communication	Implicit (via loads/stores)	Explicit Messages	Limited and Implicit
Synchronization	Explicit	Implicit (via messages)	Immutable (K, V) Pairs
Hardware Support	Typically Required	None	None
Development Effort	Lower	Higher	Lowest
Tuning Effort	Higher	Lower	Lowest

Summary

- Concepts in major parallel programming models
- OpenMP, MPI, MapReduce/Hadoop
- Comparison of different programming models

References

- OpenMP: http://www.openmp.org/mp-documents/spec25.pdf
- GCC's implementation of OpenMP: http://gcc.gnu.org/onlinedocs/libgomp/
- MPI: http://www.mcs.anl.gov/research/projects/mpi/
- MapReduce: Simplified Data Processing on Large Clusters, Dec 2004 - http://research.google.com/archive/mapreduce.html
- Apache Hadoop: http://hadoop.apache.org/

Exercise - Write a Parallel C Program using OpenMP

Problem

You want to convert a color image to gray scale. An image is a matrix of pixels, where each pixel is described by a triple (R, G, B) of intensities for the colors red, green, and blue. You need to compute a single pixel color value in the gray scale range using the *luminosity* method, which considers a weighted average of the R, G, B components. Human eyes are more perceptive to the green color, so green is weighted more than the other colors. The formula is: **0.21 R + 0.71 G + 0.07 B**

Write a Parallel C Program using OpenMP

Solution

- You are given the serial program below. For large images, the serial solution takes a large amount of time to execute. To speed-up execution, you need to:

  1. Decompose problem into tasks
  2. Map tasks to threads
  3. Parallelize the serial program below using OpenMP
  4. Optimize the OpenMP program to further improve performance

Write a Parallel C Program with OpenMP

1. Decompose problem into tasks (explain)

2. Map tasks to threads (explain)

Write a Parallel C Program

3. Parallelize serial program

```
...
Image *image = ReadImage (image info, error);
int i, j, k, red, green, blue, gray;
PixelPacket *pixels = GetAuthenticPixels (image, 0, 0, image->columns,
image->rows, error); /* read entire image array of pixels from image */

for (i=0; i<image->rows; i++) {
        for (j=0; j<image->columns; j++) {
                k = i * image->rows + j; /* current pixel array index */
                red = pixels[k].red;
                green = pixels[k].green;
                blue = pixels[k].blue;
                gray = 0.21 * red + 0.71 * green + 0.07 * blue;
                pixels[k].red = gray;
                pixels[k].green = gray;
                pixels[k].blue = gray;
        }
}
SyncAuthenticPixels(image, error);
WriteImage (image info, image);
...
```

Write a Parallel C Program

4. Other optimization for the parallel program (explain)

- Amazon EC2 and S3
  - Running serial, OpenMP and MPI programs
  - Summary

- SkyBoxz Federated Cloud
  - Running Hadoop program