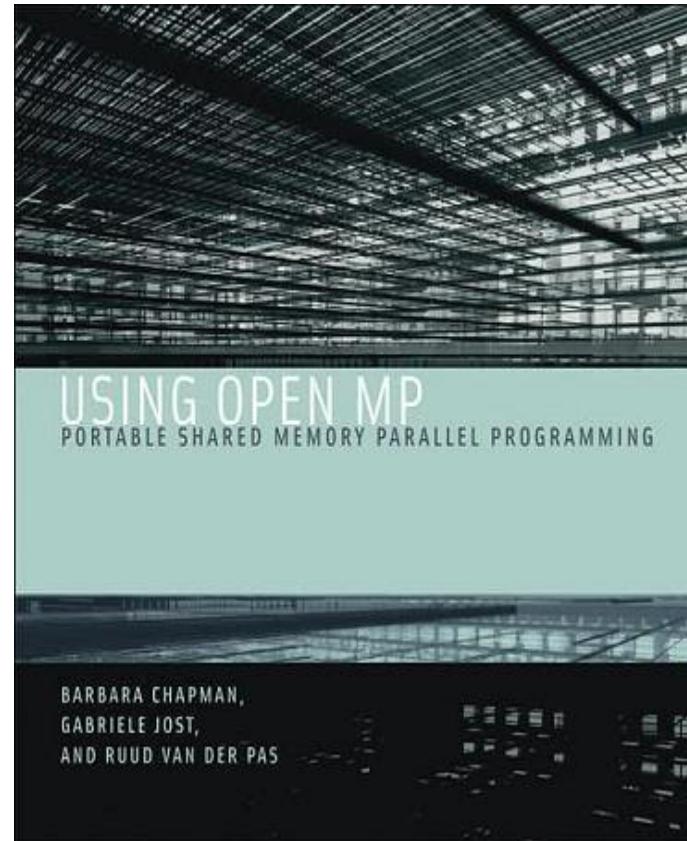


# Introduction to OpenMP



# Agenda

- Introduction to parallel computing
- Classification of parallel computers
- What is OpenMP?
- OpenMP examples



# Parallel computing

A form of computation in which many instructions are carried out simultaneously operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (in parallel).

Parallel computing has become a dominant paradigm in computer architecture, mainly in the form of multi-core processors.

Why is it required?



## Demand for parallel computing

With the increased use of computers in every sphere of human activity, computer scientists are faced with two crucial issues today:

- Processing has to be done faster
- Larger or more complex problems need to be solved

# Parallelism



Increasing the number of transistors as per Moore's law is not a solution, as it increases power consumption.

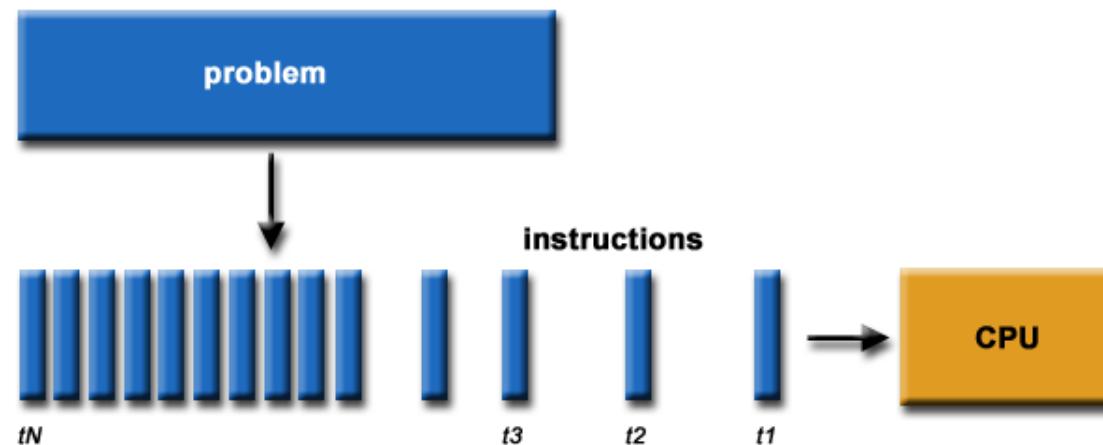
Power consumption has been a major issue recently, as it causes a problem of processor heating.

The perfect solution is **parallelism**, in hardware as well as software.

Moore's law (1965): The density of transistors will double every 18 months.

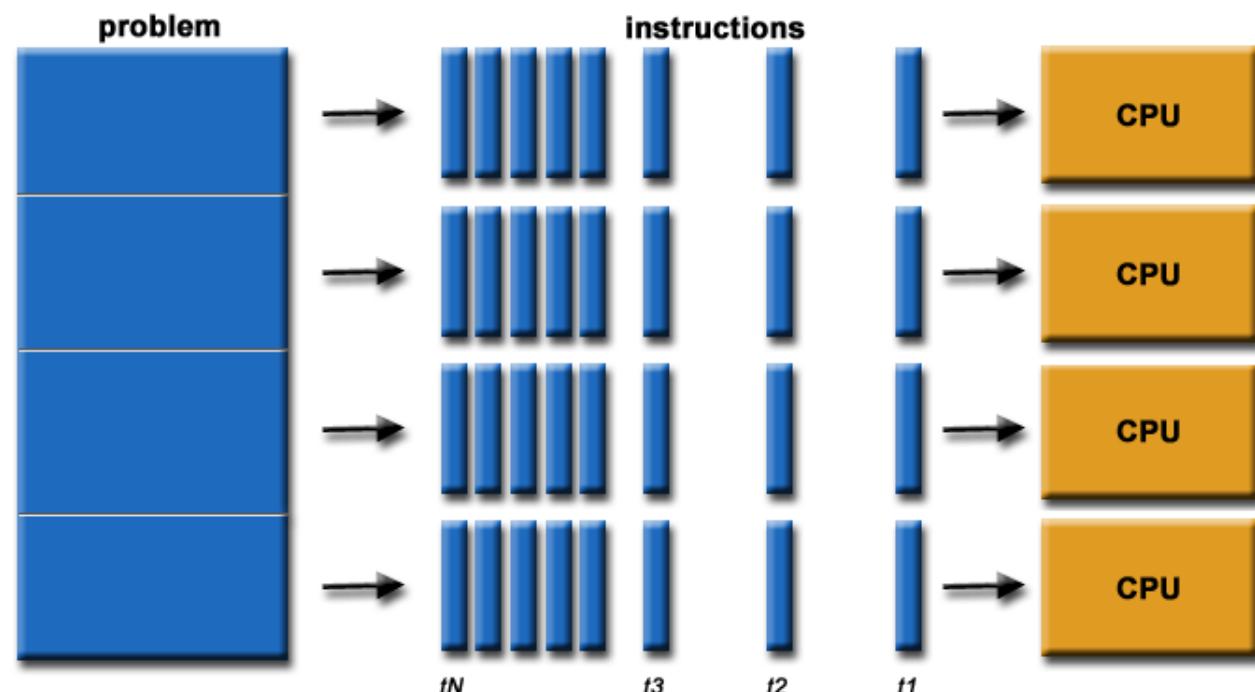
# Serial computing

A problem is broken into a discrete series of instructions.  
Instructions are executed one after another.  
Only one instruction may execute at any moment in time.



# Parallel computing

A problem is broken into discrete parts that can be solved concurrently.  
Each part is further broken down to a series of instructions.  
Instructions from each part execute simultaneously on different CPUs



# Forms of parallelism



- Bit-level parallelism

More than one bit is manipulated per clock cycle

- Instruction-level parallelism

Machine instructions are processed in multi-stage pipelines

- Data parallelism

Focuses on distributing the data across different parallel computing nodes. It is also called *loop-level parallelism*

- Task parallelism

Focuses on distributing tasks across different parallel computing nodes. It is also called *functional* or *control parallelism*

# **Key difference between data and task parallelism**



<b>Data parallelism</b>	<b>Task parallelism</b>
A task ‘A’ is divided into sub-parts and then processed	A task ‘A’ and a task ‘B’ are processed separately by different processors

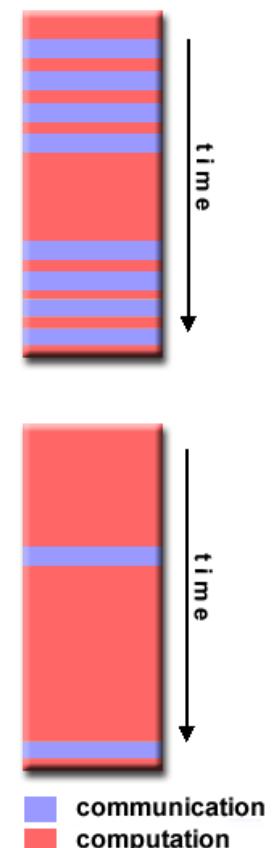


# Parallelism versus concurrency

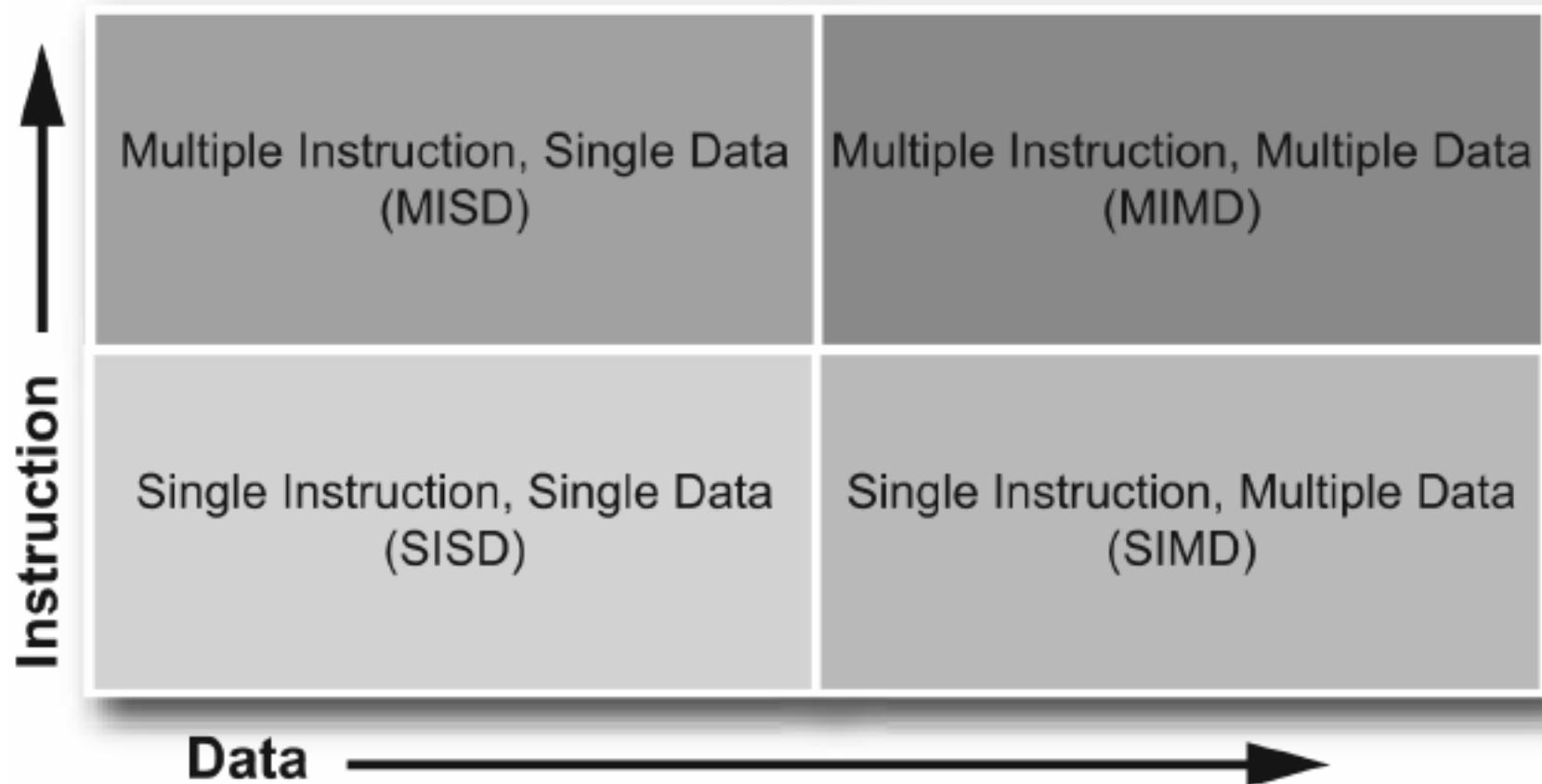
- *Concurrency* is when two tasks can start, run and complete in overlapping time periods. It doesn't necessarily mean that they will ever be running at the same instant of time. E.g., multitasking on a single-threaded machine.
- *Parallelism* is when two tasks literally run at the same time, e.g. on a multi-core computer.

# Granularity

- *Granularity* is the ratio of computation to communication.
- *Fine-grained parallelism* means individual tasks are relatively small in terms of code size and execution time. The data are transferred among processors frequently in amounts of one or a few memory words.
- *Coarse-grained* is the opposite: data are communicated infrequently, after larger amounts of computation.

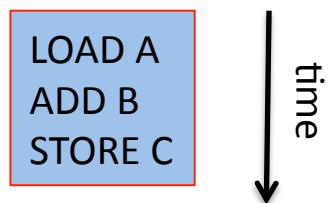


# Flynn's classification of computers



M. Flynn, 1966

# Flynn's taxonomy: SISD



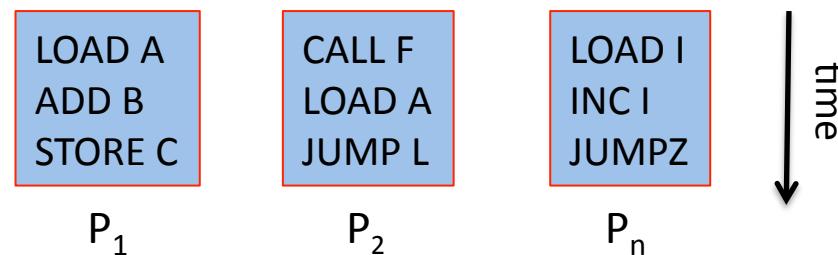
A serial (non-parallel) computer

**Single Instruction:** Only one instruction stream is acted on by the CPU in one clock cycle

**Single Data:** Only one data stream is used as input during any clock cycle

Examples: uni-core PCs, single CPU workstations and mainframes

# Flynn's taxonomy: MIMD

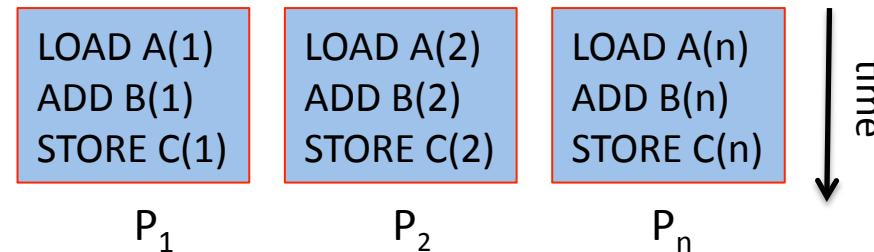


**Multiple Instruction:** Every processing unit may be executing a different instruction at any given clock cycle

**Multiple Data:** Each processing unit can operate on a different data element

Examples: Most modern computers fall into this category

# Flynn's taxonomy: SIMD



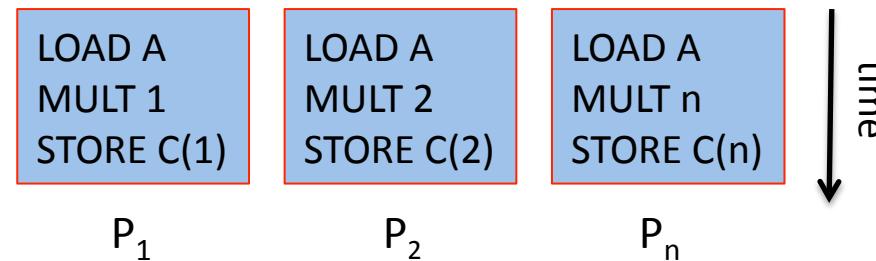
**Single Instruction:** All processing units execute the same instruction at any given clock cycle

Best suited for problems of high degree of regularity, such as image processing

**Multiple Data:** Each processing unit can operate on a different data element

Examples: Connection Machine CM-2, Cray C90, NVIDIA

# Flynn's taxonomy: MISD



A single data stream is fed into multiple processing units

Each processing unit operates on the data independently via independent instruction streams

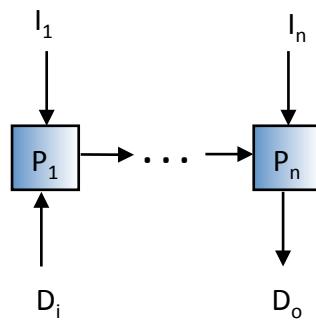
Examples: Few actual examples of this category have ever existed

# Flynn's Taxonomy

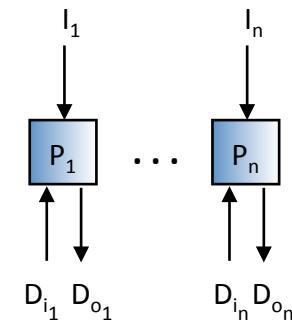
(1966)



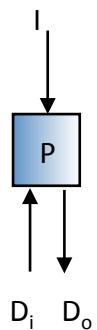
MISD (Multiple-Instruction Single-Data)



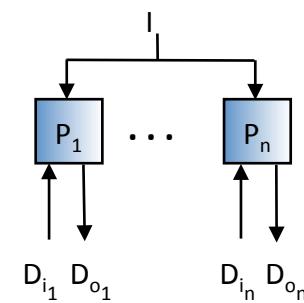
MIMD (Multiple-Instruction Multiple-Data)

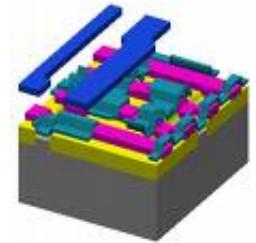


SISD (Single-Instruction Single-Data)



SIMD (Single-Instruction Multiple-Data)





# Parallel classification

Parallel computers:

- SIMD (Single Instruction Multiple Data):  
Synchronized execution of the same instruction on a set of processors
- MIMD (Multiple Instruction Multiple Data):  
Asynchronous execution of different instructions

# Parallel machine memory models

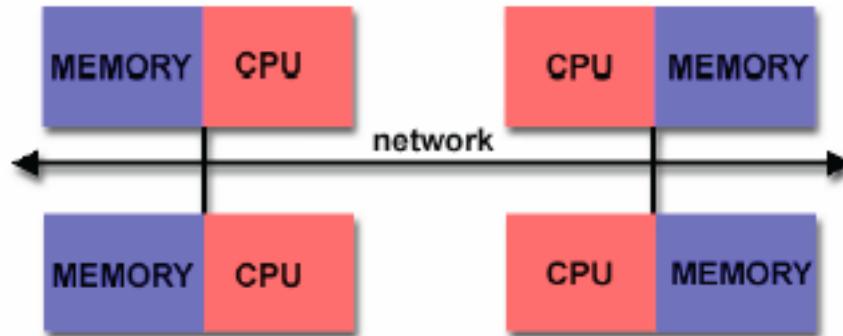


Distributed memory model

Shared memory model

Hybrid memory model

# Distributed memory



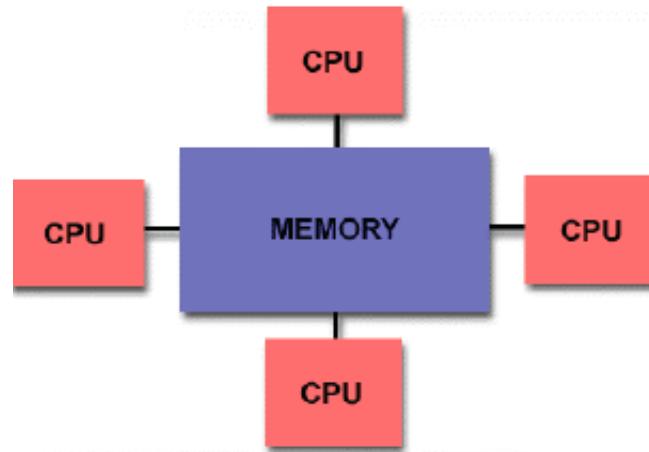
Processors have local (non-shared) memory

Requires communication network.  
Data from one processor must be communicated if required

Synchronization is programmer's responsibility

- + Very scalable
- + No cache coherence problems
- + Use commodity processors
- Lot of programmer responsibility
- NUMA (Non-Uniform Memory Access) times

# Shared memory



Multiple processors operate independently but access global memory space

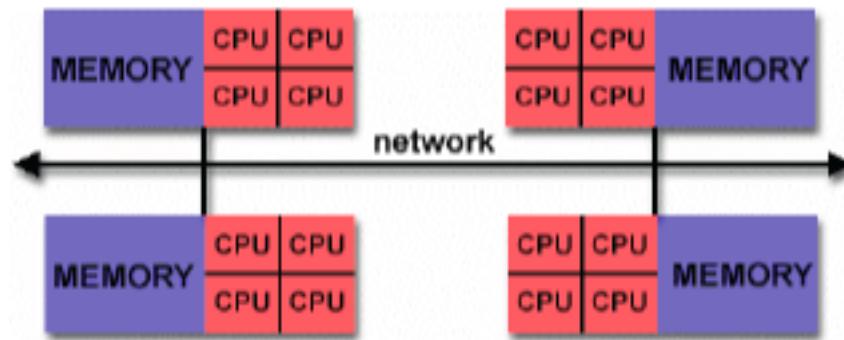
Changes in one memory location are visible to all processors

- + User friendly
- + UMA (Uniform Memory Access) times. UMA = SMP (Symmetric Multi-Processor)

- Expense for large computers

Poor scalability is “myth”

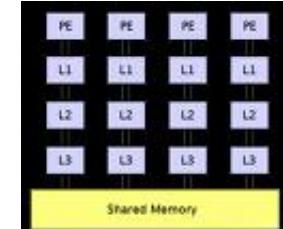
# Hybrid memory



Biggest machines today have both

Advantages and disadvantages are  
those of the individual parts

# Shared memory access



Uniform Memory Access – UMA:  
(Symmetric Multi-Processors – SMP)

- centralized shared memory, accesses to global memory from all processors have same latency

Non-uniform Memory Access – NUMA:  
(Distributed Shared Memory – DSM)

- memory is distributed among the nodes, local accesses much faster than remote accesses

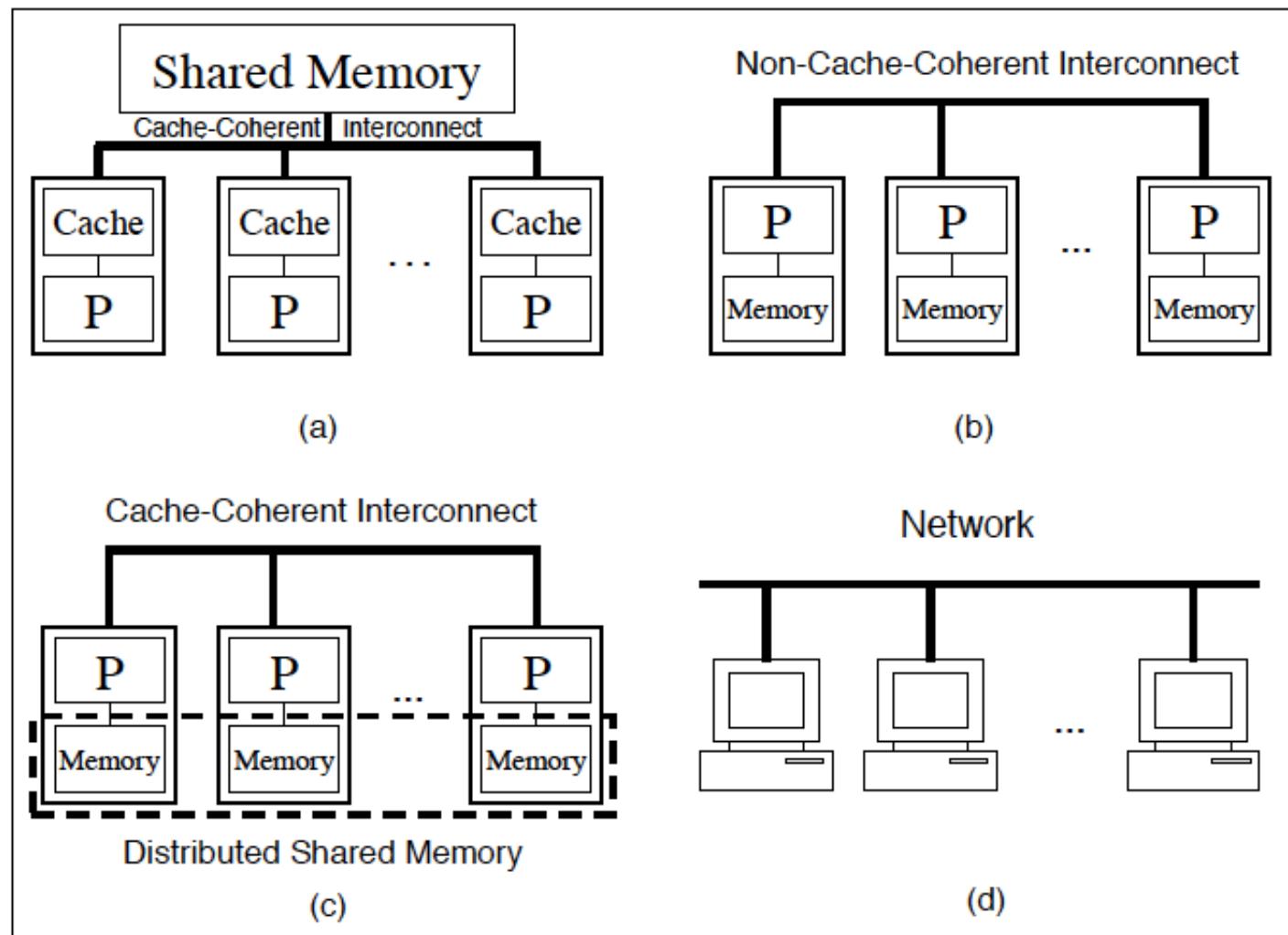


Figure 1.2: **Distributed- and shared-memory computers** – The machine in (a) has physically shared memory, whereas the others have distributed memory. However, the memory in (c) is accessible to all processors.

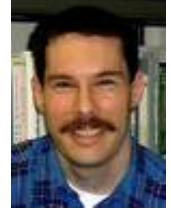
# Moore's Law

(1965)



Moore's Law, despite predictions of its demise, is still in effect. Despite power issues, transistor densities are still doubling every 18 to 24 months.

With the end of frequency scaling, these new transistors can be used to add extra hardware, such as additional cores, to facilitate parallel computing.



# Parallelism in hardware and software

- Hardware
  - going parallel (many cores)
- Software?
  - Herb Sutter (Microsoft):  
*“The free lunch is over. Software performance will no longer increase from one generation to the next as hardware improves ... unless it is parallel software.”*

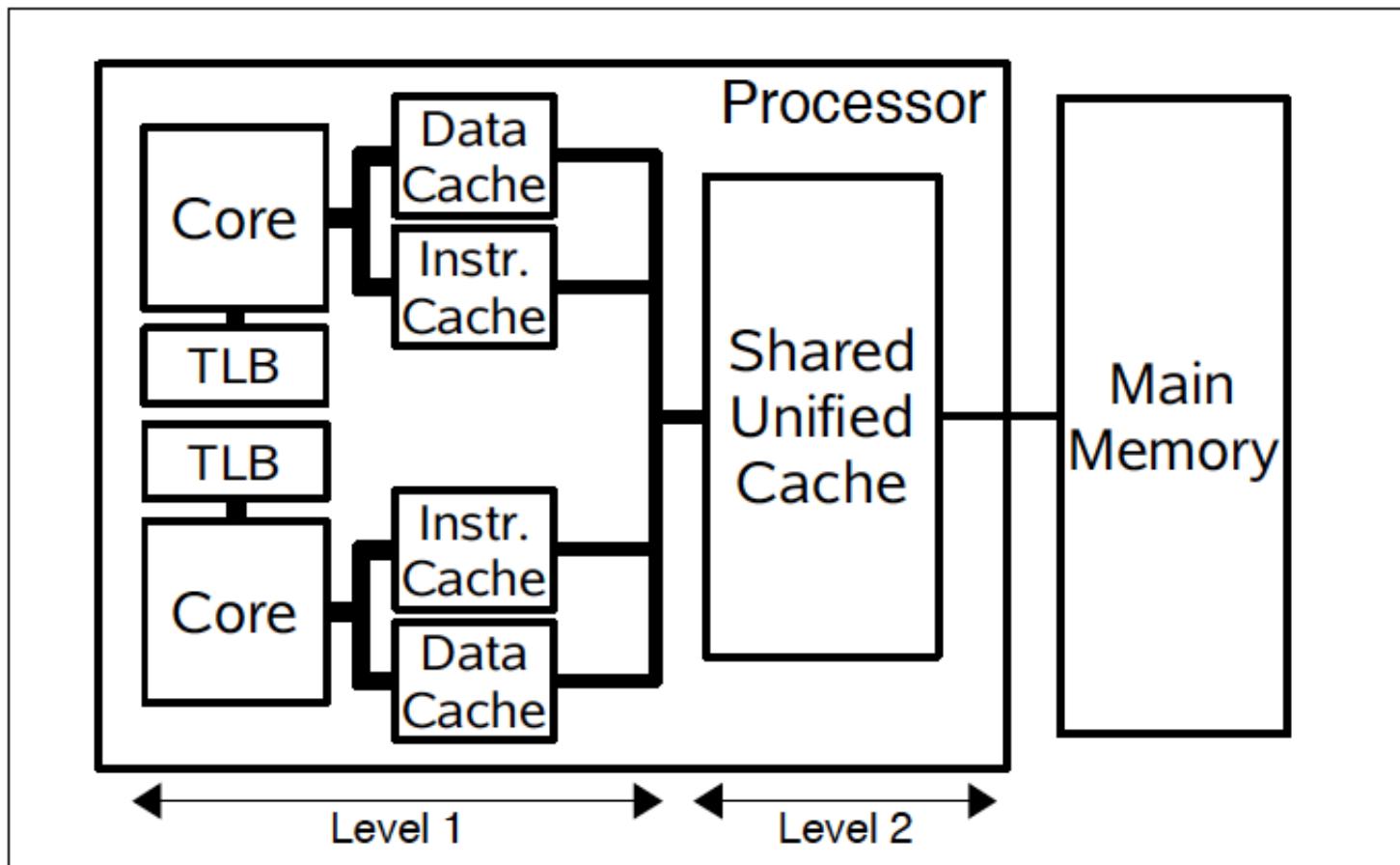


Figure 1.1: Block diagram of a generic, cache-based dual core processor



## Amdahl's Law [1967]

Gene Amdahl, IBM designer

Assume fraction  $f_p$  of execution time is parallelizable

No overhead for scheduling, communication, synchronization, etc.

Fraction  $f_s = 1 - f_p$  is serial (not parallelizable)

Time on 1 processor:  $T_1 = f_p T_1 + (1 - f_p) T_1$

Time on  $N$  processors:  $T_N = \frac{f_p T_1}{N} + (1 - f_p) T_1$

$$\text{Speedup} = \frac{T_1}{T_N} = \frac{1}{\frac{f_p}{N} + (1 - f_p)} \leq \frac{1}{1 - f_p}$$

$$Speedup = \frac{1}{\frac{f_p}{N} + (1 - f_p)}$$

## Example

95% of a program's execution time occurs inside a loop that can be executed in parallel.

What is the maximum speedup we should expect from a parallel version of the program executing on 8 processors?

$$Speedup \leq \frac{1}{0.95 / 8 + 0.05} \approx 5.9$$

What is the maximum speedup achievable by this program, regardless of how many processes are used?

$$Speedup \leq \frac{1}{0.05} = 20$$

$$Speedup = \frac{1}{\frac{f_p}{N} + (1 - f_p)}$$

## Amdahl's Law

For mainframes Amdahl expected  $f_s = 1 - f_p = 35\%$

For a 4-processor: speedup  $\approx 2$

For an infinity-processor: speedup  $< 3$

Therefore, stay with mainframes with only  
one/few processors!

Amdahl's Law prevented designers from exploiting parallelism for many years as even the most trivially parallel programs contain a small amount of natural serial code

# Limitations of Amdahl's Law



Two key flaws in past arguments that Amdahl's law is a fatal limit to the future of parallelism are

- (1) Its proof focuses on the steps in a particular algorithm, and does not consider that other algorithms with more parallelism may exist
- (2) **Gustafson's Law:** The proportion of the computations that are sequential normally decreases as the problem size increases



## Gustafson's Law [1988]

The key is in observing that  $f_s$  and  $N$  are not independent of one another.

Parallelism can be used to increase the size of the problem. Time gained by using  $N$  processors (in stead of one) for the parallel part of the program can be used in the serial part to solve a larger problem.

Gustafson's speedup factor is called *scaled speedup factor*.

## Gustafson's Law (cont'd)

Assume  $T_N$  is fixed to one, meaning the problem is scaled up to fit the parallel computer.

Then execution time on a single processor will be  $f_s + Nf_p$  as the  $N$  parallel parts must be executed sequentially. Thus,

$$\text{Speedup} = \frac{T_1}{T_N} = \frac{f_s + Nf_p}{f_s + f_p} = f_s + Nf_p = \boxed{N + (1 - N)f_s}$$

The serial fraction  $f_s$  decreases with increasing problem size. Thus, “**Any sufficiently large program scales well.**”

$$\text{Speedup} = N + (1 - N)f_s$$

## Example

95% of a program's execution time occurs inside a loop that can be executed in parallel.

What is the scaled speedup on 8 processors?

$$\text{Speedup} = 8 + (1 - 8) \cdot 0.05 = 7.65$$



# What is OpenMP?

Portable, shared-memory threading API for Fortran, C, and C++

Standardizes data- and task-level parallelism

Supports coarse-grained parallelism

Combines serial and parallel code in single source

- mainly by preprocessor (compiler) directives
- few runtime routines
- environment variables

# What does OpenMP stand for?



Short version: **Open Multi-Processing**

Long version: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

# Advantages of OpenMP



- Good performance and scalability
  - If you do it right
- De-facto standard
- Portable across shared-memory architectures
  - Supported by a large number of compilers
- Requires little programming effort
- Maps naturally into a multi-core architecture
- Allows the program to be parallelized incrementally

# OpenMP properties



OpenMP allows for a developer to parallelize their applications **incrementally**.

By simply compiling with or without the `openmp` compiler flag OpenMP can be turned on or off.

Code compiled without the `openmp` flag simply ignores the OpenMP directives, which allows simple access back to the original serial application.



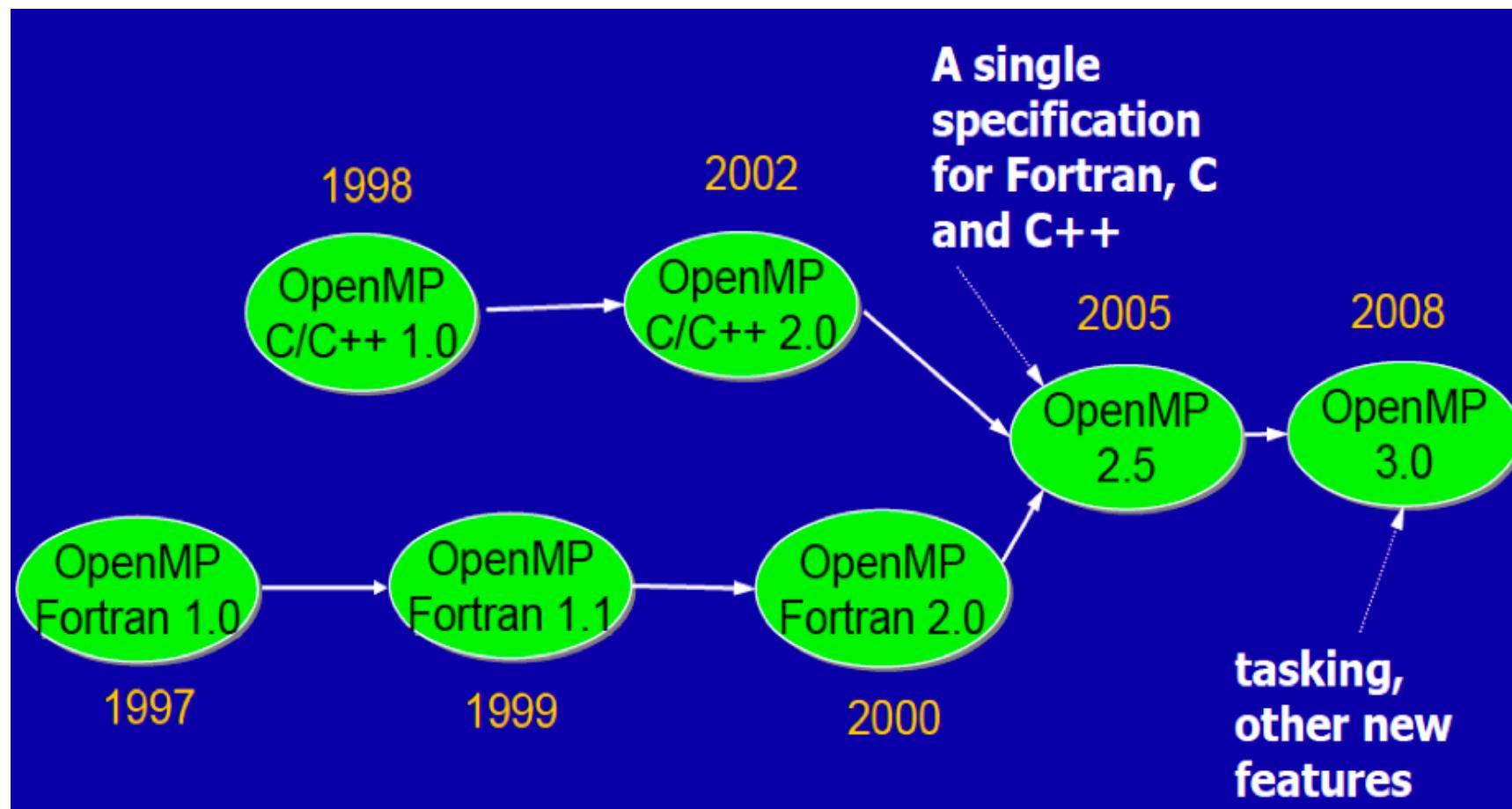
# Programmer's view of OpenMP

- OpenMP will:
  - Allow the programmer to separate a program into *parallel regions*, rather than concurrently executing threads.
  - Hide stack management
  - Provide synchronization primitives
- OpenMP will not:
  - Parallelize automatically
  - Check for data dependencies, race conditions, or deadlocks
  - Guarantee speedup

# OpenMP versus MPI

OpenMP	MPI
Only for shared memory computers	Portable to all platforms
Easy to parallelize incrementally, more difficult to write highly scalable programs	Parallelize all or nothing. Significant complexity even for implementing simple and localized constructs
Small API based on compiler directives and limited library routines	Vast collection of library routines (more than 300)
Same program can be used for serial and parallel execution	Possible but difficult to use same program for serial and parallel execution
Shared variable versus private variable can cause confusion	Variables are local to each processor

# OpenMP release history



# OpenMP core syntax

Most of the constructs of OpenMP are compiler directives.

`#pragma omp construct [clause [,] [clause] ... ] new-line`

Example

```
#pragma omp parallel numthreads( 4 )
```

Function prototypes and types in a file:

```
#include <omp.h>
```

Most OpenMP constructs apply to a *structured block* – a block of one or more statements with one entry point at the top and one point of exit at the bottom. It is OK to have an `exit()` within the structured block.

# OpenMP constructs

## 1. Parallel regions

```
#pragma omp parallel
```

## 2. Work sharing

```
#pragma omp for
```

```
#pragma omp sections
```

## 3. Data environment

```
#pragma omp parallel shared/private(...)
```

## 4. Synchronization

```
#pragma omp barrier
```

## 5. Runtime functions

```
int my_thread_id = omp_get_thread_num();
```

## 6. Environment variables

```
export OMP_NUM_THREADS=4
```

# Execution model

## Fork/join

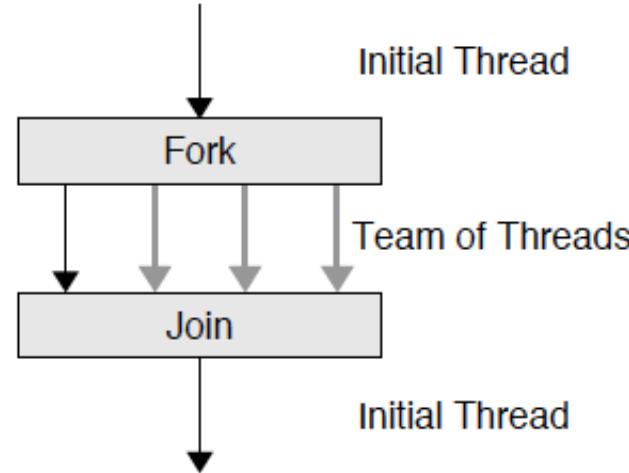
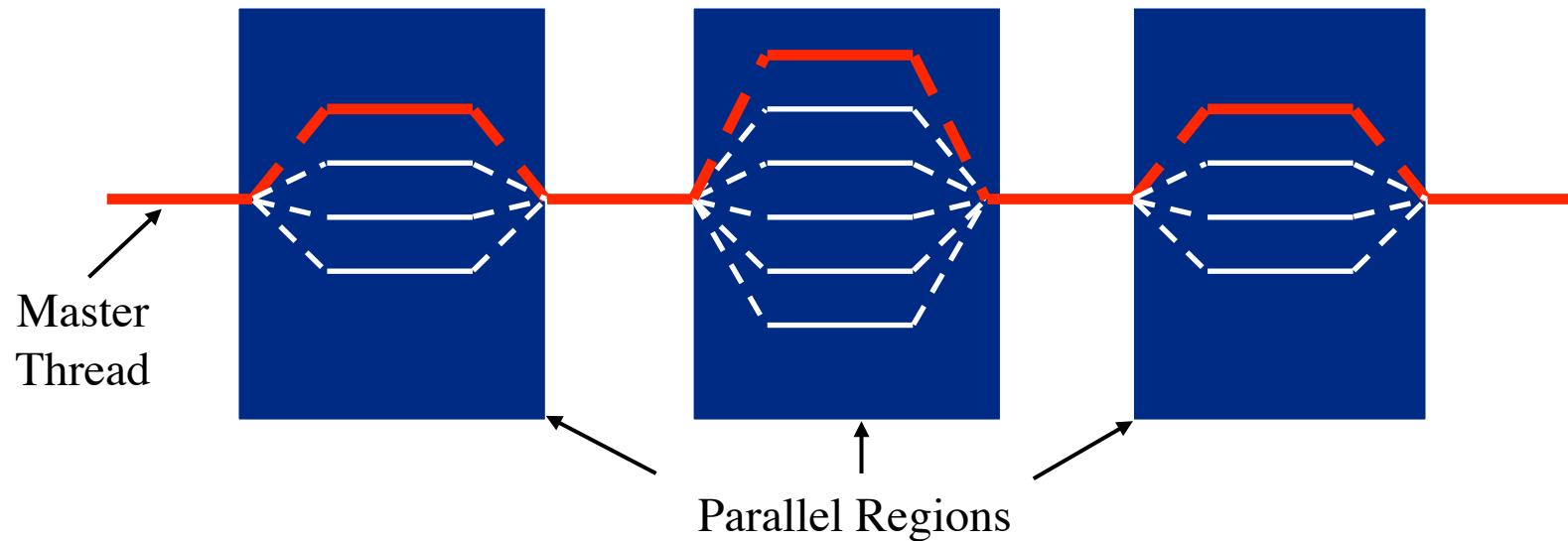


Figure 2.1: The fork-join programming model supported by OpenMP – The program starts as a single thread of execution, the initial thread. A team of threads is forked at the beginning of a parallel region and joined at the end.

# The fork/join model

- Master thread spawns a team of threads as needed.
- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program.



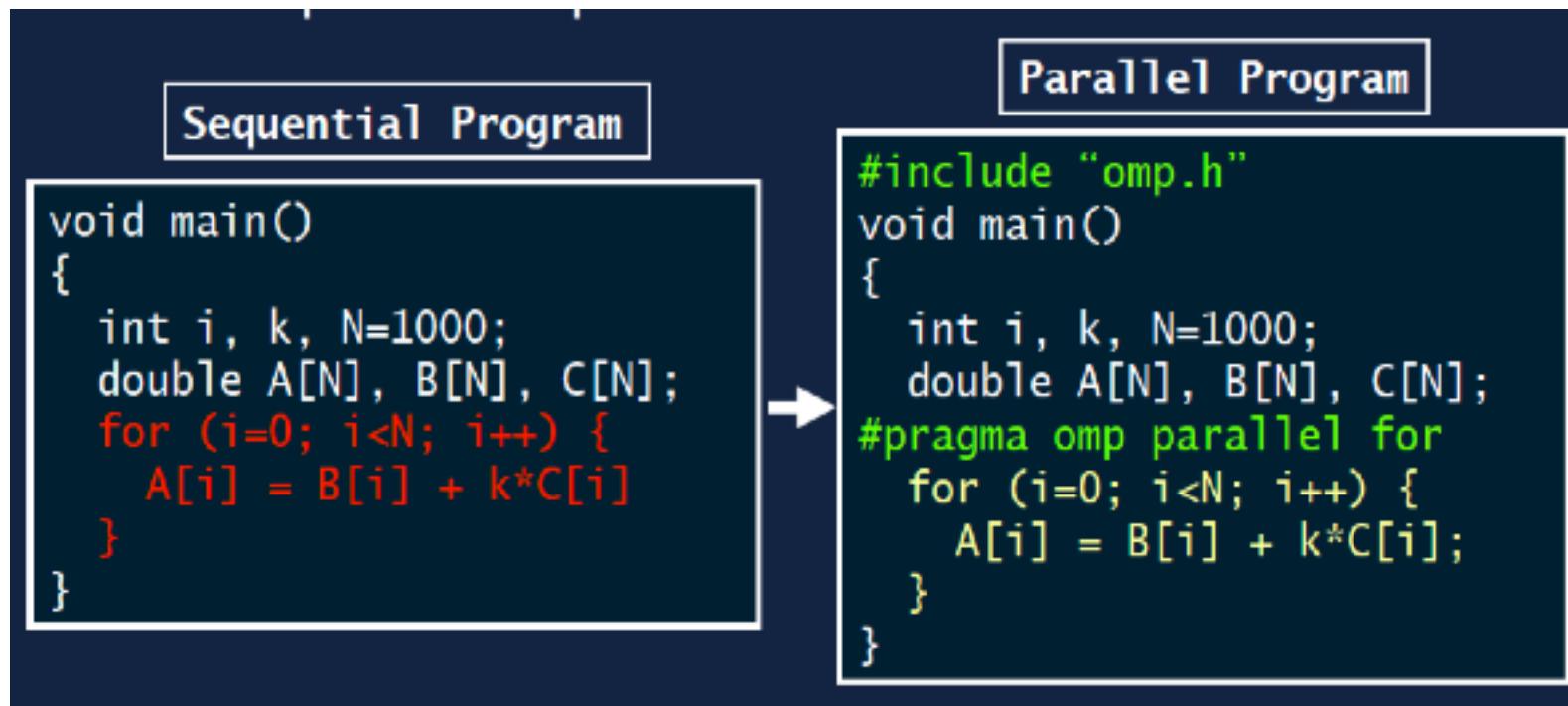
# How do threads interact?

- OpenMP is a multi-threading, shared address model
  - Threads communicate by sharing variables
- Unintended sharing of data causes race conditions
  - Race condition: when the program's outcome changes as the threads are scheduled differently
- To control race conditions:
  - Use synchronization to protect data conflicts. Synchronization is expensive, so change how data are accessed to minimize the need for synchronization

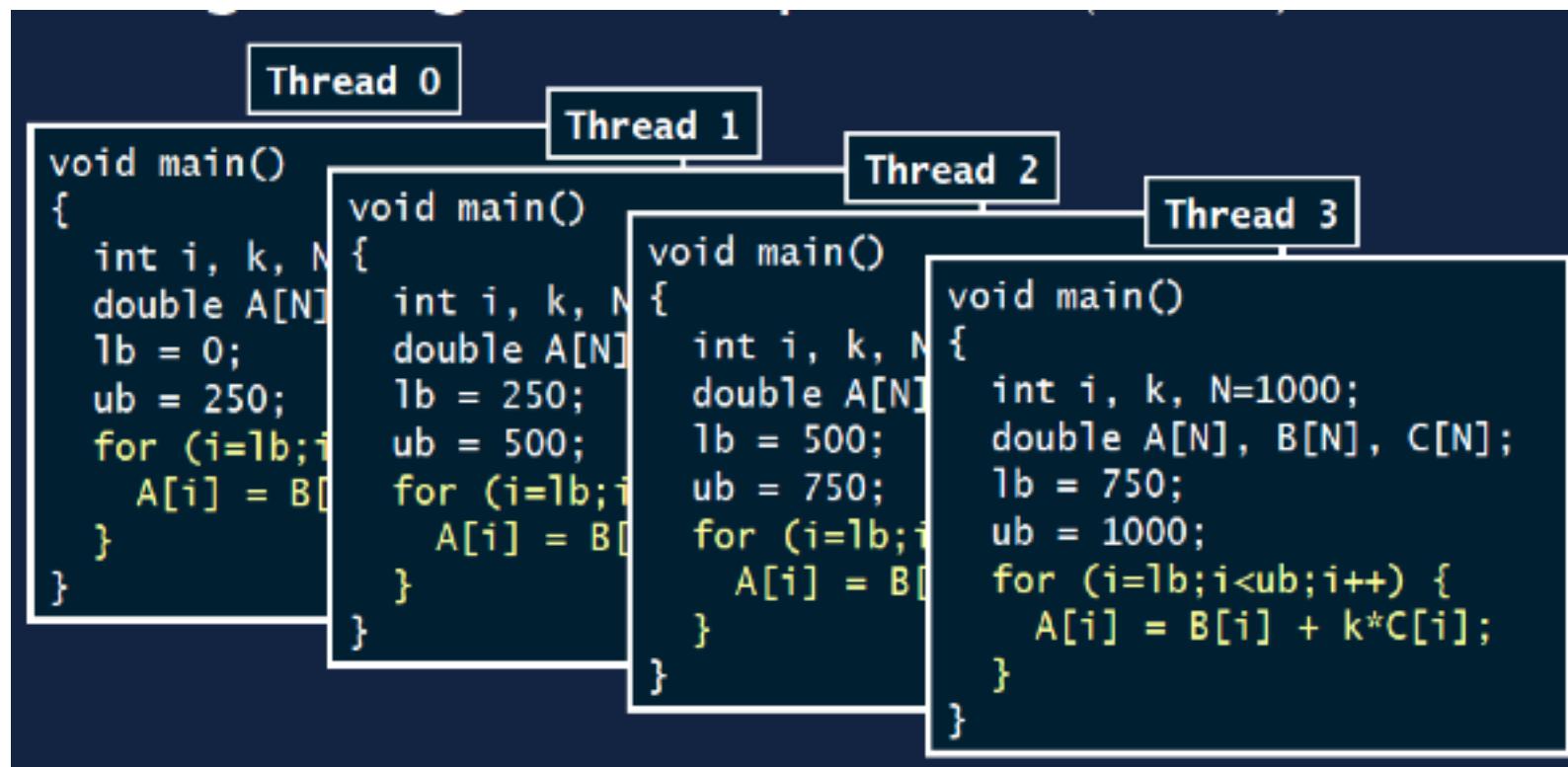
# How is OpenMP typically used?

OpenMP is typically used to parallelize loops:

1. Find your most time-consuming loops
2. Split them up between threads



# Single Program Multiple Data (SPMD)



# Example (one parallel region)

Program: hello.c

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

int main (int argc, char *argv[ ]) {
    #pragma omp parallel
    printf("Hello from thread %d out of %d\n",
           omp_get_thread_num(),
           omp_get_num_threads());
}
```

# **Example**

## **Compilation and execution**

Compilation:

```
gcc -o hello -fopenmp hello.c
```

Execution:

```
./hello
```

```
Hello from thread 2 of 4 threads  
Hello from thread 3 of 4 threads  
Hello from thread 1 of 4 threads  
Hello from thread 0 of 4 threads
```

# Dot-product example

Problem: Write a parallel program for computing the dot-product of two vectors,  $\vec{a}$  and  $\vec{b}$ .

$$\vec{a} \cdot \vec{b} = \sum_{i=0}^{n-1} a_i b_i$$

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i] * b[i];
```

# Dot-product - Serial C version

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    double sum, a[256], b[256];
    int i, n = 256;
    for (i = 0; i < n; i++) {
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }
    sum = 0;
    for (i = 0; i < n; i++)
        sum += a[i] * b[i];
    printf("a*b = %f\n", sum);
}
```

# Dot-product - Serial Java version

```
class DotSerial {
    public static void main(String[] args) {
        int n = 256;
        double[] a = new double[n];
        double[] b = new double[n];
        for (int i = 0; i < n; i++) {
            a[i] = i * 0.5;
            b[i] = i * 2.0;
        }
        double sum = 0;
        for (int i = 0; i < n; i++)
            sum += a[i] * b[i];
        System.out.println("a*b = " + sum);
    }
}
```

# Dot-product - Java thread version

```
class Task extends Thread {  
    Task(double[] a, double[] b, int iStart, int iEnd) {  
        this.a = a; this.b = b;  
        this.iStart = iStart;  
        this.iEnd = iEnd;  
        start();  
    }  
  
    double[] a, b;  
    int iStart, iEnd;  
    double sum;  
  
    public void run() {  
        sum = 0;  
        for (int i = iStart; i < iEnd; i++)  
            sum += a[i] * b[i];  
    }  
  
    public double getSum() {  
        try {  
            join();  
        } catch(InterruptedException e) {}  
        return sum;  
    }  
}
```

```
import java.util.*;

class DotThreads {
    public static void main(String[] args) {
        int n = 256;
        double[] a = new double[n];
        double[] b = new double[n];
        for (int i = 0; i < n; i++) {
            a[i] = i * 0.5;
            b[i] = i * 2.0;
        }
        int numThreads = 4;
        List<Task> tasks = new ArrayList<Task>();
        for (int i = 0; i < numThreads; i++)
            tasks.add(new Task(a, b, i * n / numThreads,
                               (i + 1) * n / numThreads));
        double sum = 0;
        for (Task t : tasks)
            sum += t.getSum();
        System.out.println("a*b = " + sum);
    }
}
```

# Dot-product - Java Callable version

```
class Task implements Callable<Double> {
    Task(double[] a, double[] b, int iStart, int iEnd) {
        this.a = a;
        this.b = b;
        this.iStart = iStart;
        this.iEnd = iEnd;
    }

    double[] a, b;
    int iStart, iEnd;

    public Double call() {
        double sum = 0;
        for (int i = iStart; i < iEnd; i++)
            sum += a[i] * b[i];
        return sum;
    }
}
```

```
import java.util.concurrent.*;
import java.util.*;

class DotCallable {
    public static void main(String[] args) {
        int n = 256;
        double[] a = new double[n];
        double[] b = new double[n];
        for (int i = 0; i < n; i++) {
            a[i] = i * 0.5;
            b[i] = i * 2.0;
        }
    }
}
```

cont'd on next page

```
int numThreads = 4;
ExecutorService executor =
    Executors.newFixedThreadPool(numThreads);
List<Future<Double>> futures =
    new ArrayList<Future<Double>>();
for (int i = 0; i < numThreads; i++)
    futures.add(executor.submit(
        new Task(a, b, i * n / numThreads,
                 (i + 1) * n / numThreads)));
executor.shutdown();
double sum = 0;
for (Future<Double> f : futures) {
    try {
        sum += f.get();
    } catch (Exception e) { e.printStackTrace(); }
}
System.out.println("a*b = " + sum);
}
```

# Dot-product - Java (alternative version)

```
class Dotter {
    Dotter() {
        for (int i = 0; i < n; i++) {
            a[i] = i * 0.5;
            b[i] = i * 2.0;
        }
        for (int i = 0; i < n; i++)
            new DotThread(this).start();
    }

    int n = 256, index = 0, sum = 0,
        numThreads = 4, terminated = 0;
    double[] a = new double[n];
    double[] b = new double[n];
```

cont'd on next page

```
synchronized int getNextIndex() {  
    return index < n ? index++ : -1;  
}  
  
synchronized void addPartialSum(double partialSum) {  
    sum += partialSum;  
    if (++terminated == numThreads)  
        System.out.println("a*b = " + sum);  
}  
}
```

cont'd on next page

```
class DotThread extends Thread {  
    DotThread(Dotter parent) {  
        this.parent = parent;  
    }  
  
    Dotter parent;  
  
    public void run() {  
        int i;  
        double sum = 0;  
        while ((i = parent.getNextIndex()) != -1)  
            sum += parent.a[i] * parent.b[i];  
        parent.addPartialSum(sum);  
    }  
}
```

# Dot-product - MPI version

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

int main(int argc, char *argv[]) {
    double sum, sum_local, a[256], b[256];
    int i, n = 256, id, np;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    for (i = id; i < n; i += np) {
        a[i] = i * 0.5;
        b[i] = i * 2.0; ← Initialization
    }
    sum_local = 0;
    for (i = id; i < n; i += np)
        sum_local += a[i] * b[i];
    MPI_Reduce(&sum_local, &sum, 1, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);
    if (id == 0)
        printf("a*b = %f\n", sum);
    return 0;
}
```

# Dot-product - MPI version (scatter)

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

int main(int argc, char *argv[]) {
    double sum, sum_local, a[256], b[256];
    int *disps, *cnts;
    double *a_part, *b_part;
    int i, n = 256, id, np, share;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    share = n / np;
    if (id == 0) {
        for (i = 0; i < n; i++) {
            a[i] = i * 0.5;
            b[i] = i * 2.0;
        }
    }
```

cont'd on next page

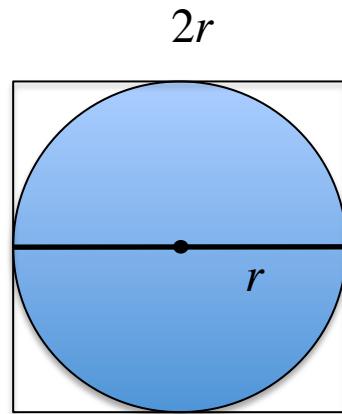
```
disps = (int *) malloc(np * sizeof(int));
cnts = (int *) malloc(np * sizeof(int));
offset = 0;
for (i = 0; i < np; i++) {
    disps[i] = offset;
    cnts[i] = i < n - npp * share ? share + 1 : share;
    offset += cnts[i];
}
a_part = (double *) malloc(cnts[id] * sizeof(double));
b_part = (double *) malloc(cnts[id] * sizeof(double));
MPI_Scatterv(a, cnts, disps, MPI_DOUBLE, a_part,
              n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatterv(b, cnts, disps, MPI_DOUBLE, b_part,
              n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
sum_local = 0;
for (i = 0; i < cnts[id]; i++)
    sum_local += a_part[i] * b_part[i];
MPI_Reduce(&sum_local, &sum, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);
if (id == 0)
    printf("a*b = %f\n", sum);
return 0;
}
```

# Dot-product - OpenMP C version

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    double sum, a[256], b[256];
    int i, n = 256;
    for (i = 0; i < n; i++) {
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }
    sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < n; i++)
        sum += a[i] * b[i];
    printf("a*b = %f\n", sum);
}
```

# Monte Carlo algorithm for estimating $\pi$



Throw darts randomly within the square

The chance of hitting the circle  $P$  is proportional to the ratio of areas:

$$\text{Area}_{\text{circle}} = \pi r^2$$

$$\text{Area}_{\text{square}} = 4r^2$$

$$P = \text{Area}_{\text{circle}} / \text{Area}_{\text{square}} = \frac{\pi}{4}$$

Compute  $\pi$  by randomly choosing points.  
Count the fraction that falls in the circle.  
Compute  $\pi (= 4P)$ .

# Monte Carlo program

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

int main(int argc, char *argv[ ]) {
    int trials = 100000, count = 0, i;
    #pragma omp parallel for reduction(+:count)
    for (i = 0; i < trials; i++) {
        double x = (double) rand() / RAND_MAX;
        double y = (double) rand() / RAND_MAX;
        if (x * x + y * y <= 1)
            count++;
    }
    printf("Estimate of pi    = %f\n", 4.0 * count / trials);
    printf("True value of pi = 3.141593\n");
}
```

Estimate of pi = 3.141520  
True value of pi = 3.141593

# Matrix times vector example

Problem: Write a parallel OpenMP program for computing the product of an  $m \times n$  matrix  $B$  and an  $n \times 1$  vector  $c$ .

$$a_i = \sum_{j=0}^{n-1} B_{i,j} c_j$$

$$a_i = \sum_{j=0}^{n-1} B_{i,j} c_j$$

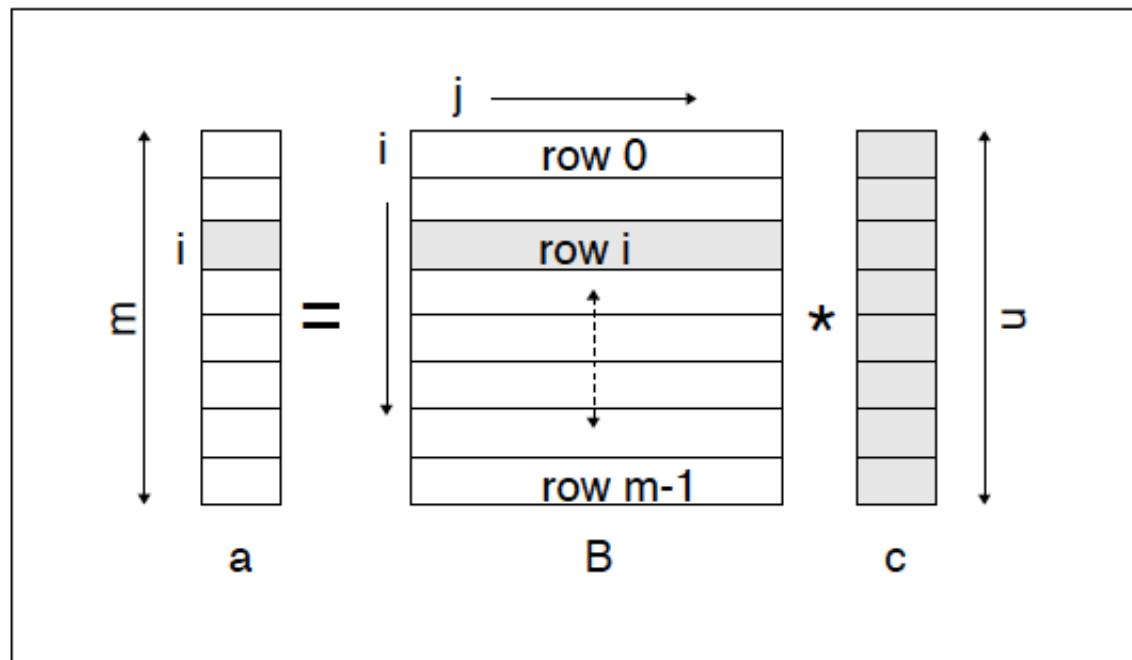


Figure 3.9: Graphical representation of the row variant of the matrix times vector operation – Element  $a_i$  is obtained by computing the dot product of row  $i$  of matrix  $B$  with vector  $c$ .

```
int main(int argc, char *argv[ ]){  
    double *a, *B, *c;  
    int i, j, m, n;  
    printf("Please give m and n: ");  
    scanf("%d %d",&m, &n);  
    printf("\n");  
    if ((a = (double *) malloc(m * sizeof(double))) == NULL)  
        perror("memory allocation for a");  
    if ((B = (double *) malloc(m * n * sizeof(double))) == NULL)  
        perror("memory allocation for b");  
    if ((c = (double *) malloc(n * sizeof(double))) == NULL)  
        perror("memory allocation for c");  
    printf("Initializing matrix B and vector c\n");  
    for (j = 0; j < n; j++)  
        c[j] = 2.0;  
    for (i = 0; i < m; i++)  
        for (j = 0; j < n; j++)  
            B[i * n + j] = i;  
    printf("Executing mxv function for m = %d n = %d\n", m, n);  
    mxv(m, n, a, B, c);  
    free(a); free(B); free(c);  
    return(0);  
}
```

```
void mxv(int m, int n, double *a,
         double *B, double *c) {
    int i, j;
    #pragma omp parallel for default(none) \
        shared(m,n,a,B,c) private(i,j)
    for (i = 0; i < m; i++) {
        a[i] = 0.0;
        for (j = 0; j < n; j++)
            a[i] += B[i * n + j] * c[j];
    }
}
```

# Conditional compilation

Keeping sequential and parallel programs  
as a single source code

```
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

int id = omp_get_thread_num();
```