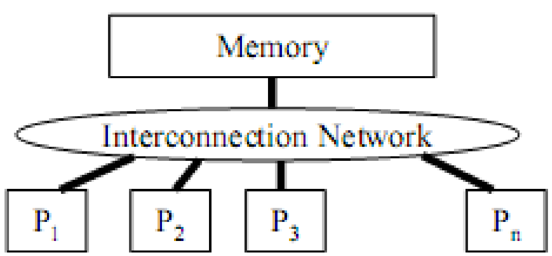
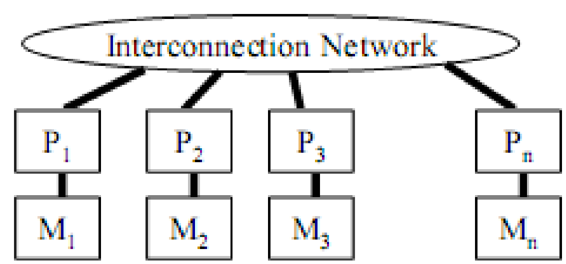
The Two Main Models of Parallel Processing
Distributed Memory (MPI) and Shared Memory (OpenMP)

- **Two different HW arch. models led to two different SW Programming models**
  - **5 years ago, MPI was standard; Now, in 2010, OpenMP is more popular**

- 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





Memory Systems: Distributed Memory

- All memory is associated with processors.
- If processor A needs data in processor B, then B must send a message to A containing the data.

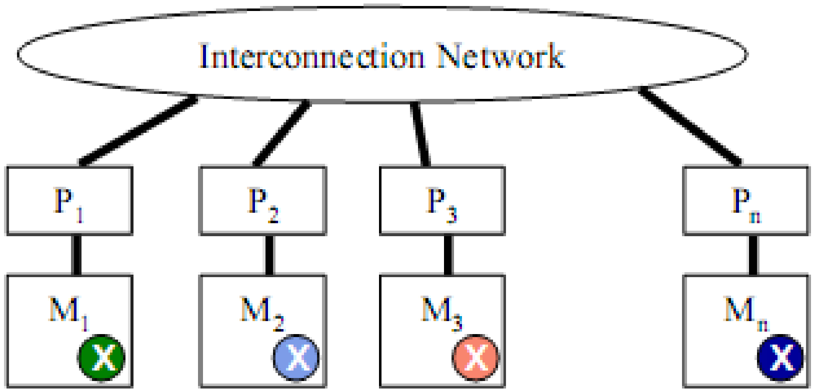
- Advantages:
  - Memory is scalable with number of processors
  - Each processor has rapid access to its own memory
  - *Cost effective and easier to build:* can use commodity parts

Disadvantages

- Programmer is responsible for many of the details of the communication, easy to make mistakes.

- May be difficult to distribute the data structures, often need to revise them to add additional pointers.

Programming Distributed Memory Processors

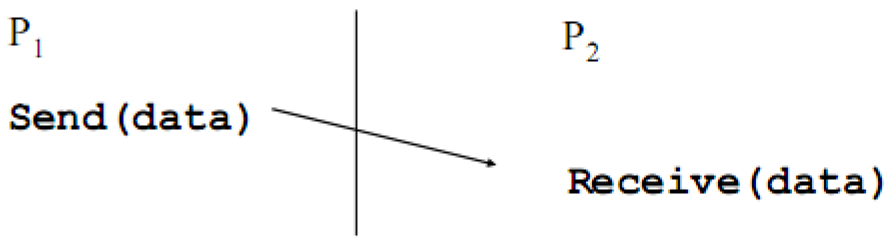


- Processors 1…n ask for X
- There are n places to look
  - Each processor's memory has its own X
  - Xs may vary

- For Processor 1 to look at Processors 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



- Messages are like handshakes.
- They need two partners: a sender and receiver.

MPI – the de facto standard

MPI has become the de facto standard for parallel computing using message passing

What Is MPI?

The ***Message-Passing Interface*** (MPI) is a standard for expressing distributed parallelism via message passing.

MPI consists of a ***header file***, a ***library of routines*** and a ***runtime environment***.

When you compile a program that has MPI calls in it, your compiler links to a local implementation of MPI, and then you get parallelism; if the MPI library isn't available, then the compile will fail.

MPI can be used in Fortran, C and C++.

A Message Passing Library Specification

- MPI: specification
  - Not a language or compiler specification
  - Not a specific implementation or product
  - SPMD model (same program, multiple data)
- For parallel computers, clusters, and heterogeneous networks, multicores
- Full-featured
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication

Where Did MPI Come From?

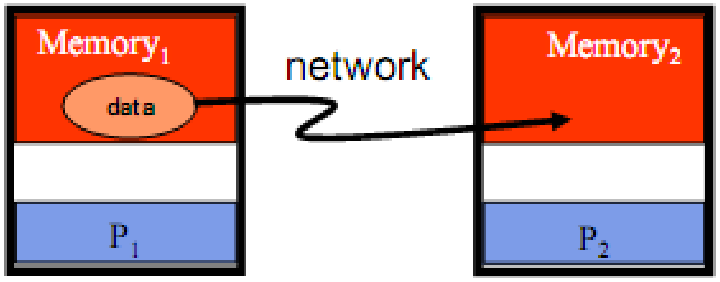
- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum organized in 1992 with broad participation
  - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
  - Portability library writers: PVM, p4
  - Users: application scientists and library writers
  - Finished in 18 months

Communication Patterns

- With message passing, programmer has to understand the computation and orchestrate the communication accordingly
  - Point to Point
  - Broadcast (one to all) and Reduce (all to one)
  - All to All (each processor sends its data to all others)
  - Scatter (one to several) and Gather (several to one)

Point-to-Point

- Basic method of communication between two processors
  - Originating processor "sends" message to destination processor
  - Destination processor then "receives" the message
- The message commonly includes
  - Data or other information
  - Length of the message
  - Destination address and possibly a tag



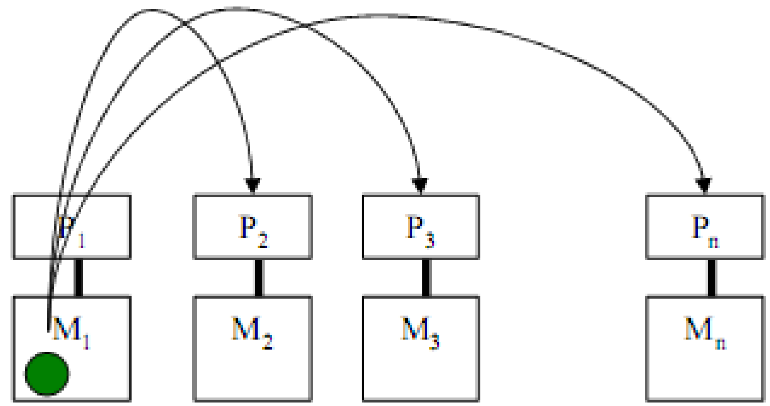
Cell "send" and "receive" commands

```
mfc_get(destination LS addr,
        source memory addr,
        # bytes,
        tag,
        <...>)
```

```
mfc_put(source LS addr,
        destination memory addr,
        # bytes,
        tag,
        <...>)
```

Broadcast

- One processor sends the same information to many other processors
  - **MPI_BCAST**




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

```
A[n] = {…}
B[n] = {…}

Broadcast(B[1..n])
for (i = 1 to n)
  // round robin distribute B
  // to m processors
  Send(A[i % m])
…
```

Reduction

- Example: every processor starts with a value and needs to know the sum of values stored on all processors
- A reduction combines data from all processors and returns it to a single process
  - **`MPI_REDUCE`**
  - Can apply any associative operation on gathered data
    - ADD, OR, AND, MAX, MIN, etc.
  - No processor can finish reduction before each processor has contributed a value
- **`BCAST/REDUCE`** can reduce programming complexity and may be more efficient in some programs

Example Message Passing Program

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

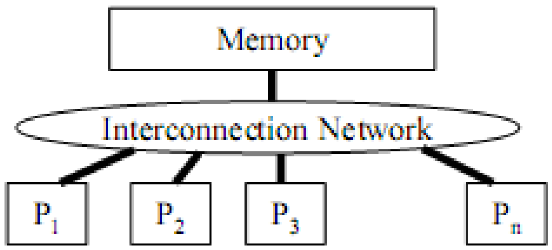
```
A[n] = {…}
B[n] = {…}

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])
```

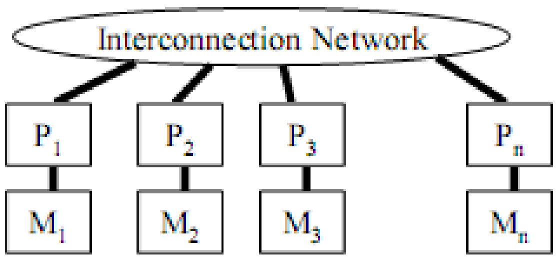
processor 2

```
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])
```

- **In the "old days", each processor was in a separate computer**
  - **So parallel processing was accomplished using a collection of these computers**
    - **Many computers were put together into a Cluster or a Large SuperComputer**
  - **Each had its own Memory → Distributed Memory →Message Passing Needed**
- **Now (2010) Multi-Core and Many-Core Designs put several processors on same chip**
  - **So cores are likely to Share Memory →Shared Mem → Shared Mem Programming**

- 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





Memory Systems: Shared Memory

- Global memory space, accessible by all processors
- Processors may have local memory to hold copies of some global memory.
- Consistency of copies is usually maintained by hardware.

- Advantages:
  - Global address space is user-friendly, program may be able to use global data structures efficiently and with little modification.
  - Data sharing between tasks is fast

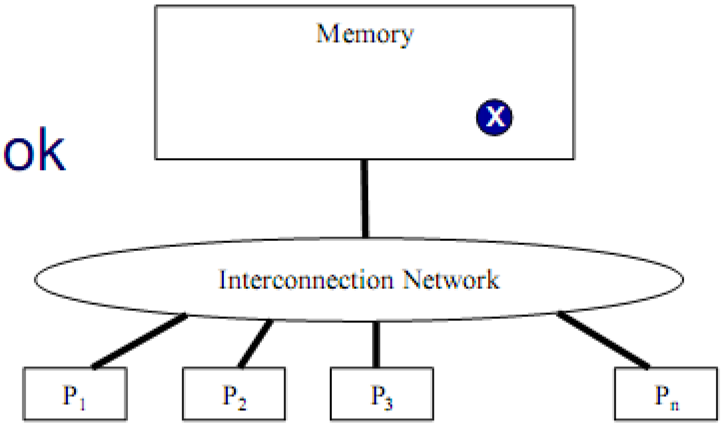
- **Disadvantages:**
- System may suffer from lack of scalability. Adding CPUs increases traffic on shared memory - to - CPU path. This is especially true for cache coherent systems

- Programmer is responsible for correct synchronization

- Systems larger than an SMP need some special-purpose components.

Programming Shared Memory Processors

- 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



- **OpenMP**
  - ***De-facto standard API for writing shared memory parallel applications in C, C++, and Fortran***
  - ***Consists of:***
    - ***Compiler directives***
    - ***Run time routines***
    - ***Environment variables***

- **General Philosophy behind OpenMP is that the compiler doesn't have enough Information at the Source Code Level to do effective Parallelization**

  - ❑ ***The compiler may not be able to do the parallelization in the way you like to see it:***
    - ***It can not find the parallelism***
      - ✔ ***The data dependence analysis is not able to determine whether it is safe to parallelize or not***
    - ***The granularity is not high enough***
      - ✔ ***The compiler lacks information to parallelize at the highest possible level***
  - ❑ ***This is when explicit parallelization through OpenMP directives comes into the picture***

- **Therefore, Programmer must add extra "Comments" to help in Parallelization**

  Comments, or "Directives", are typically focused on Loops in the Code

  Loops (that are data independent per iteration) are prime targets for parallelization

  Loops are typically where the majority of execution time is spent

  Loops are typically modular and operate on Arrays or Matrices

  By focusing on Loops, we can get the best bang per buck

  Minimal extra coding (directives) to get maximum speedup in runtime hotspots

Advantages of OpenMP

- ❑ ***Good performance and scalability***
  - ***If you do it right ....***
- ❑ ***De-facto and mature standard***
- ❑ ***An OpenMP program is portable***
  - ***Supported by a large number of compilers***
- ❑ ***Requires little programming effort***
- ❑ ***Allows the program to be parallelized incrementally***

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP *implementation*!
  - Requires compiler support (C or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than T concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

Motivation

- Unix Forking and Thread libraries are hard to use
  - P-Threads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind

- Synchronization between threads introduces a new dimension of program correctness

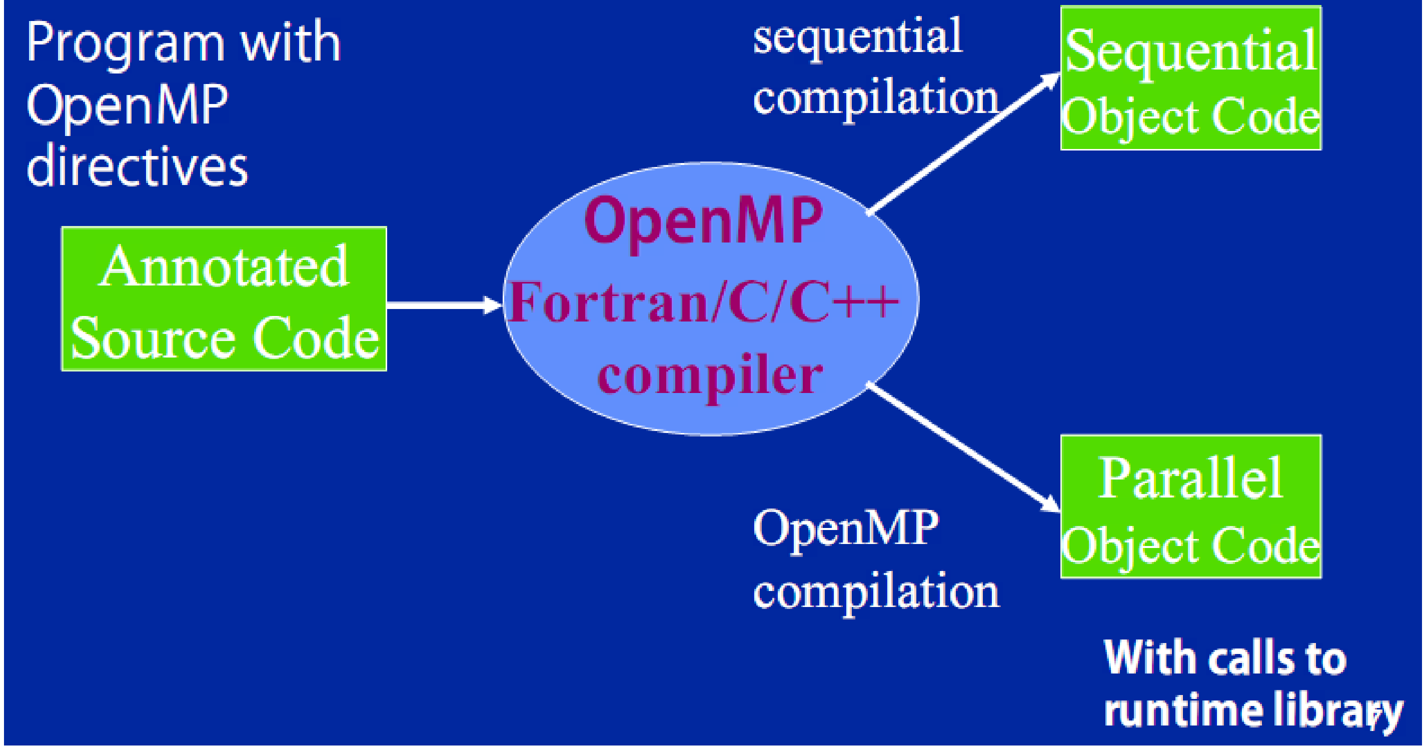
- Wouldn't it be nice to write serial programs and somehow parallelize them "automatically"?
  - OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence
  - It is not automatic: you can still make errors in your annotations

- **Basic Idea Behind OpenMP**

- **User must decide what is parallel in program**
  - **Makes any changes needed to original source code**
  - **E.g. to remove any dependences in parts that should run in parallel**

- **User inserts directives telling compiler how statements are to be executed**
  - **what parts of the program are parallel**
  - **how to assign code in parallel regions to threads**
  - **what data is private (local) to threads**

OpenMP Implementation



- **Sample OpenMP Directives:**

```
#pragma omp parallel for
#pragma omp critical
#pragma omp master
#pragma omp barrier
#pragma omp single
#pragma omp atomic
#pragma omp section
#pragma omp flush
#pragma omp ordered
```

- **Directives look like comments to a non-OpenMP savvy compiler (OpenMP disabled)**
  - **So by not using the –fopenmp compiler option, parallelization is switched off**

- **If program is compiled sequentially**
  - **OpenMP comments and pragmas are ignored**
- **If code is compiled for parallel execution**
  - **comments and/or pragmas are read, and**
  - **drive translation into parallel program**
- **Ideally, one source for both sequential and parallel program (big maintenance plus)**

- **OpenMP can use the same source code for both Sequential and Parallel Execution**
  - **Provides Tremendous Advantages**
    - **Ultimate Scaling (from 1 core sequential up to N cores of parallelism)**
    - **Simplifies Debugging and Optimization**
      - **Optimized sequential program will more likely be an optimized parallel one**
      - **Allows Debugging to occur in simpler, Sequential mode first**
      - **(But a correct sequential program is not necessarily a correct parallel one)**

- **Allows Incremental Parallelization**

  Sequential program a special case of threaded program

  Programmers can add parallelism incrementally

  Profile program execution

  Repeat

    Choose best opportunity for parallelization

    Transform sequential code into parallel code

  Until further improvements not worth the effort

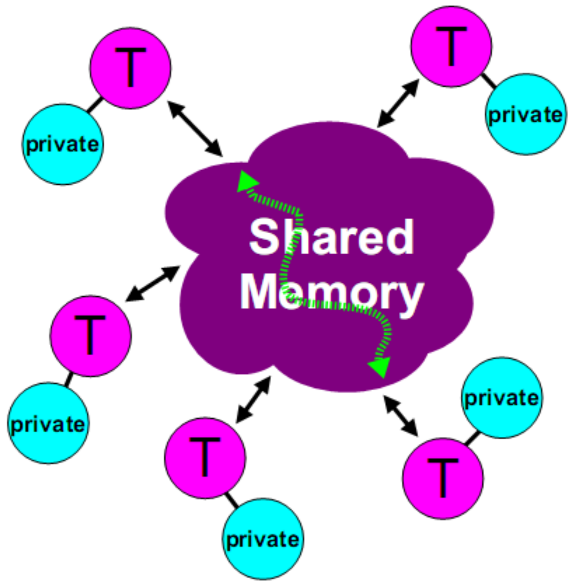
- If Parallelization switch of Compiler is turned on (using the –fopenmp option) then:

  - Compiler generates explicit threaded code
    - shields user from many details of the multithreaded code
  - Compiler figures out details of code each thread needs to execute
  - Compiler does not check that programmer directives are correct!

- Programmer must ensure Code + Directives are correct

  - The program generated by the compiler is executed by multiple threads
    - One thread per processor or core
  - Each thread performs part of the work
    - Parallel parts executed by multiple threads
    - Sequential parts executed by single thread
  - Dependences in parallel parts require synchronization between threads

- **OpenMP's Memory Model**



- *All threads have access to the same, globally shared, memory*
- *Data can be shared or private*
- *Shared data is accessible by all threads*
- *Private data can only be accessed by the thread that owns it*
- *Data transfer is transparent to the programmer*
- *Synchronization takes place, but it is mostly implicit*

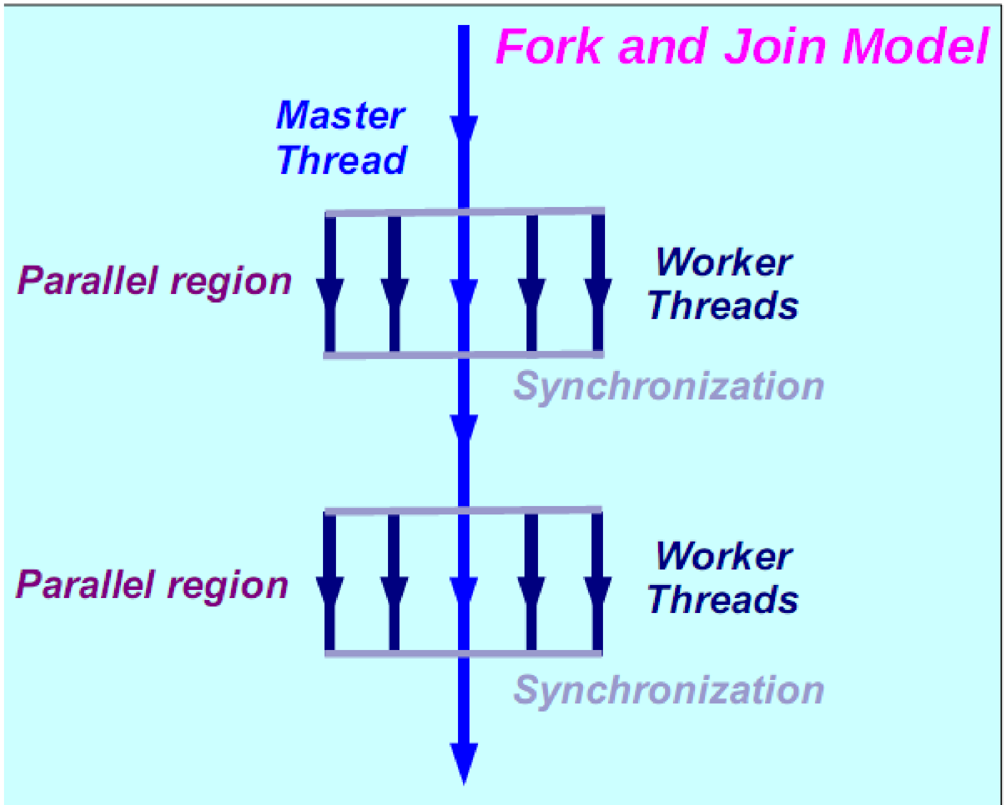
❑ ***In an OpenMP program, data needs to be "labelled"***

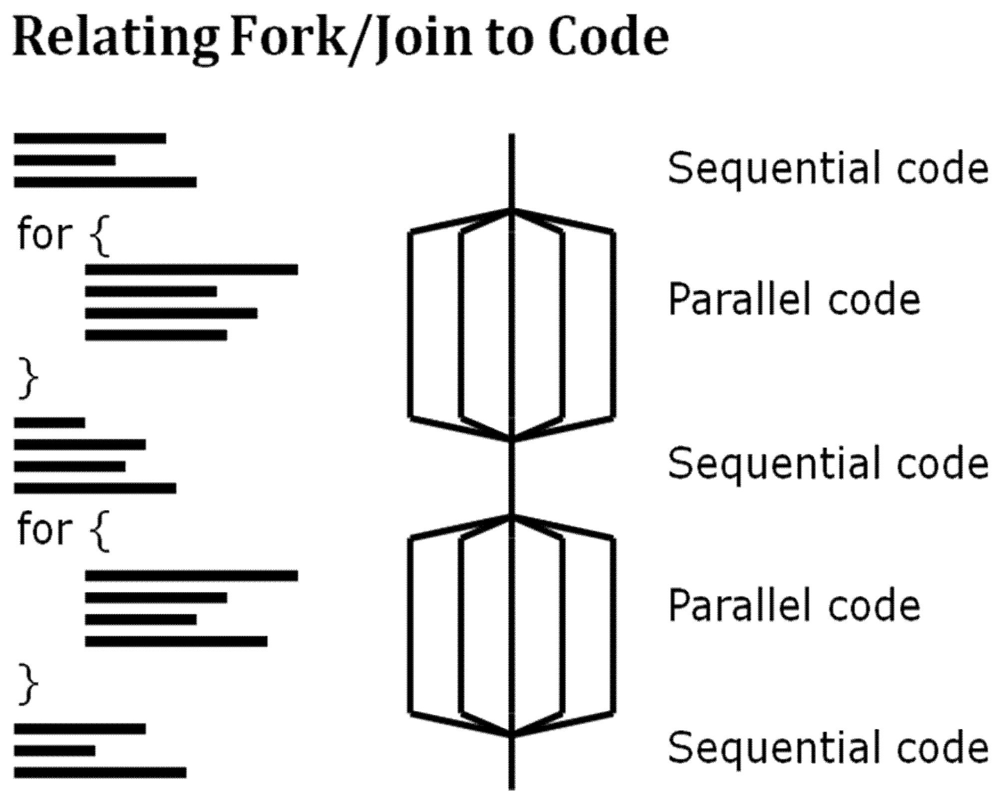
❑ ***Essentially there are two basic types:***

- *Shared*
  - ✓ ***There is only instance of the data***
  - ✓ ***All threads can read and write the data simultaneously, unless protected through a specific OpenMP construct***
  - ✓ ***All changes made are visible to all threads***
    - ◆ ***But not necessarily immediately, unless enforced ......***
- *Private*
  - ✓ ***Each thread has a copy of the data***
  - ✓ ***No other thread can access this data***
  - ✓ ***Changes only visible to the thread owning the data***

- **OpenMP's Execution Model is based on Forks**
  - **Similar to Unix Fork**
  - **Instead of Manually using Fork, Wait and Signal, Programmer uses Directives**

- **When work can be done in Parallel, Programmer Bounds Code with OpenMP Directives**





- **Prime Candidates for Parallelization (and forking of worker threads) are Loops**
  - **Code not in loops are not executed much, so can proceed in Sequential mode**

Why Loops Are Good

- Loops are **very common** in many programs.
- Also, it's easier to optimize loops than more arbitrary sequences of instructions: when a program does **the same thing over and over**, it's **easier to predict** what's likely to happen next.

So, hardware vendors have designed their products to be able to execute loops quickly.

Superscalar Loops

```
DO i = 1, length
  z(i) = a(i) * b(i) + c(i) * d(i)
END DO
```

Each of the iterations is **completely independent** of all of the other iterations; for example,

```
z(1) = a(1) * b(1) + c(1) * d(1)
```

has nothing to do with

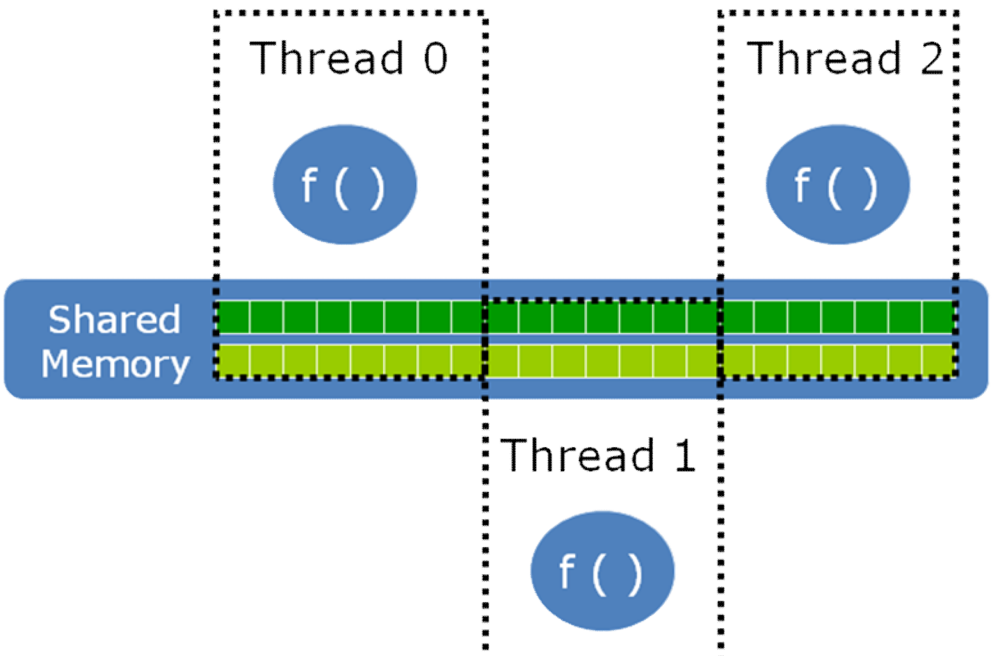
```
z(2) = a(2) * b(2) + c(2) * d(2)
```

Operations that are independent of each other can be performed in **parallel**.

- OpenMP easily parallelizes loops
  - Requires that there be No data dependencies (reads/write or write/write pairs) between iterations!

- Preprocessor calculates loop bounds for each thread directly from *serial* source

Domain Decomposition Using Threads



For-loop with independent iterations

```
for (int i=0; i<n; i++)
   c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for
for (int i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

- **Gives the Following Result for N = 1000**
- **200 Iterations are assigned to each of the 5 Threads**

Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
i=0-199	i=200-399	i=400-599	i=600-799	i=800-999
a[i]	a[i]	a[i]	a[i]	a[i]
+	+	+	+	+
b[i]	b[i]	b[i]	b[i]	b[i]
=	=	=	=	=
c[i]	c[i]	c[i]	c[i]	c[i]

Domain Decomposition

Sequential Code:

```
int a[1000], i;
for (i = 0; i < 1000; i++) a[i] = func(i);
```

Thread 0:

```
for (i = 0; i < 500; i++) a[i] = func(i);
```

Thread 1:

```
for (i = 500; i < 1000; i++) a[i] = func(i);
```

Private Shared

Shared versus Private Variables