Multiprocessor Systems

Arquitecturas Paralelas de Computadoras

Profr. Carlos Ernesto Carrillo Arellano[1]

[1]Universidad Autónoma Metropolitana - Unidad Azcapotzalco
Departamento de Ingeniería Electrónica
Correo electrónico: *ceca@xanum.uam.mx*
Personal web page: *http://ecarrillo.ddns.net*

Mayo, 2019

Contenido

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Contenido

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Introduction

1. Since the beginning of the history of computer systems, the demand for **more performance** has been the most important driving force for evolution in computer architecture
2. Many important **applications** demand more performance than a single (serial) processor core can provide, and historically have pushed parallel architecture technology
3. **Examples**: Numerical programs used in computer simulation to analyze and solve problems in science and engineering, such as climate modeling, weather forecasting, or computer-aided design

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Introduction

1. ¿One processor form many tasks?
2. ¿One processor for an intensive task?
3. Interest in **multiprocessor** architecture research is focus os increasing the performance of computer systems.
4. A multiprocessor is a computer system with **two or more central processing units** (CPUs) **linked together** to enable parallel processing
5. **Cooperation and communication**
6. The key objective of using a multiprocessor is to boost the **system's execution speed**, with other objectives being fault tolerance and multitasking

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

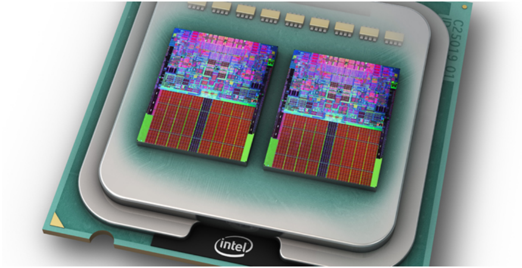
Introduction

1. The main reason for parallel programming is to **execute code efficiently**, since parallel programming saves time, allowing the **execution of applications in a shorter wall-clock time.**
2. As a consequence of executing code efficiently, p**arallel programming often scales with the problem size, and thus can solve larger problems.**
3. Parallel programming is a means of providing concurrency, particularly performing **simultaneously multiple actions at the same time**.

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared-memory and message-passing

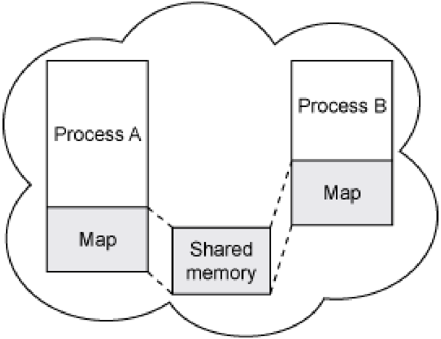
1. There are two multiprocessor architectural styles
2. **Shared-memory and message-passing** multiprocessor systems
3. Both styles use multiple processors with the goal of achieving a **linear speedup of computational power with the number of processors.**
4. They differ in the method by which the **processors exchange data**.



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared-memory and message-passing

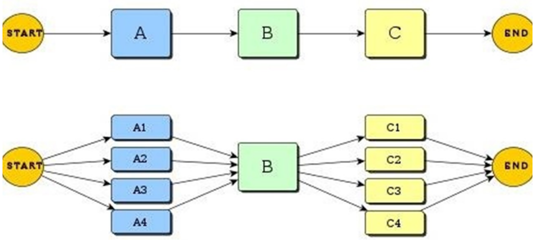
1. Processors in shared-memory multiprocessors share the same address space and can exchange data through **shared-memory locations by regular load and store instructions.**
2. Processors in message-passing multiprocessor systems have their own (private) **address space** and communicate data between their address spaces by exchanging **messages**
3. A fundamental aspect is that software must be written in such a way that parallelism is exposed



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared-memory and message-passing

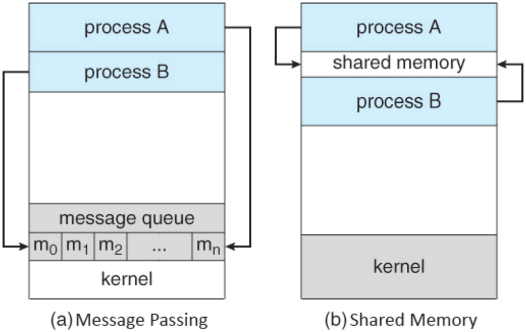
1. Computational tasks that can run in parallel must first be **identified and partitioned across the processors in a balanced fashion**
2. **Dependences** between tasks and **communication** needed to transfer results among tasks, **collectively called coordination**, must be taken into deep consideration
3. **Primitives for synchronization and communication** constitute an important part of what the underlying architecture has to support and expose to the software

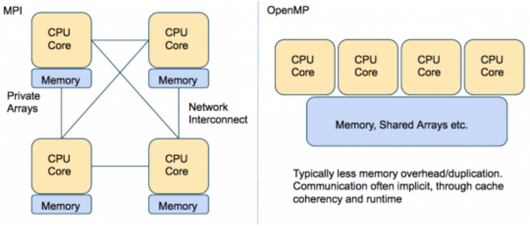


Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared-memory and message-passing

1. In shared-memory multiprocessor systems, communication is intuitively supported by the inherent shared-memory address space offered by these systems, although **explicit support for synchronization is needed**
2. Message-passing systems, on the other hand, offer explicit primitives for synchronization as well as communication: **send and receive primitives**





Parallel-programming model

1. A parallel-programming model defines how parallel computations can be expressed in a **high level programming language**
2. Popular parallel programming models have been implemented as simple **extensions** to commonly used programming languages
3. **Message Passing Interface** (MPI) for message-passing
4. **OpenMP** for shared-memory systems

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Parallel-programming model

1. Regardless of the parallel computer architecture-style (shared-memory or message-passing) **programmers or compilers must be able to express parallelism**
2. **Work partitioning** and **coordination**
3. Work that can be carried out in parallel must be **identified and partitioned among the processors**
4. **Thread** or **process** interchangeably: code that is run on a single processor (or core) in a parallel computer

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

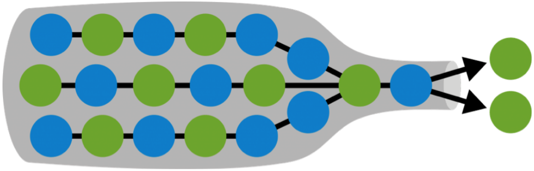
Parallel-programming model

1. **Work** done in parallel by threads running on different processors must be **coordinated** so that **the end result is the same as if the work was done by a single processor**.
2. Coordination involves two actions
3. One to **synchronize** parallel threads
4. **Communicate** partial results between threads
5. **Threads need to exchange information** either through the **memory**, or by **explicit messaging**

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Syncronization and Coordination

1. Regardless of the communication model, **communication** obviously **takes time** and has an impact on how fast a problem can be solved on a parallel computer
2. From an architecture point of view, it is important to provide adequate **support for synchronization and communication** so that they can be carried out efficiently.
3. Remove bottlenecks in a shared-memory system or in the interconnection network.



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

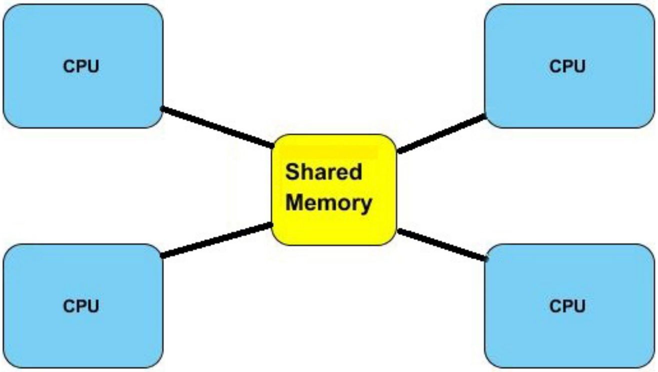
Speedup

1. A key reference point for both the architect and the application developer is how the use of parallelism **improves the performance of the application**.
2. A typical measure is the improvement in **execution time**
3. For a single, fixed problem, the performance of the machine on the problem is simply the **reciprocal of the time to complete the problem**

$$\text{Speedup}_{\text{fixed problem}}(p \text{ processors}) = \frac{\text{Time}(1 \text{ processor})}{\text{Time}(p \text{ processors})}.$$

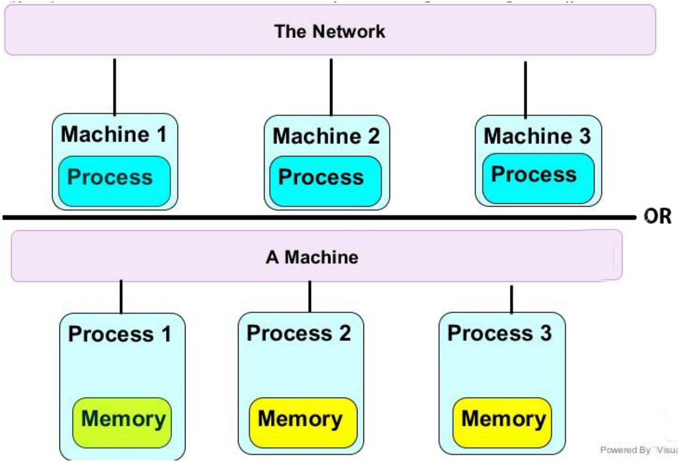
Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared memory vs message passing



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared memory vs message passing

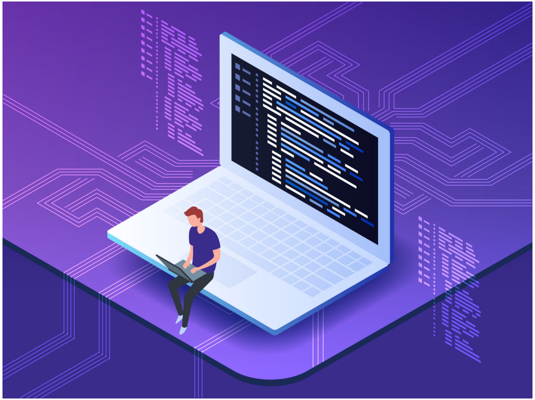


	Message Passing	**Shared Memory**
Who does communication?	Programmer	Automatic
Data distribution	Manual	Automatic
HW Support	Simple (NIC)	Extensive
Programmer		
Correctness	Difficult	Less Difficult
Performance	Dificult	Very Difficult

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Parallel-programming

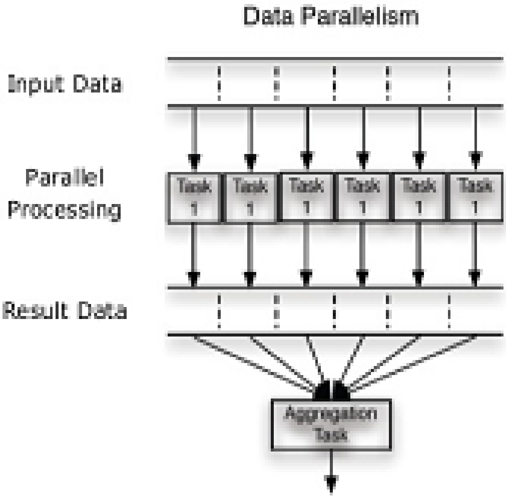
1. **Starting** from a sequential program, the **programmer or compiler** must first identify the **parts of the program which can be run in parallel**.
2. S_1 and S_2 are two **program segments** that are executed one after the other in the sequential program
3. Can run in parallel if, and only if, S_1 is **independent** of S_2 , meaning that S_1 does not produce data used by S_2
4. Running S_1 and S_2 in parallel yields the same result as if they **were executed one after the other**
5. The parallel program then conforms to **sequential semantics**



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Parallel-programming

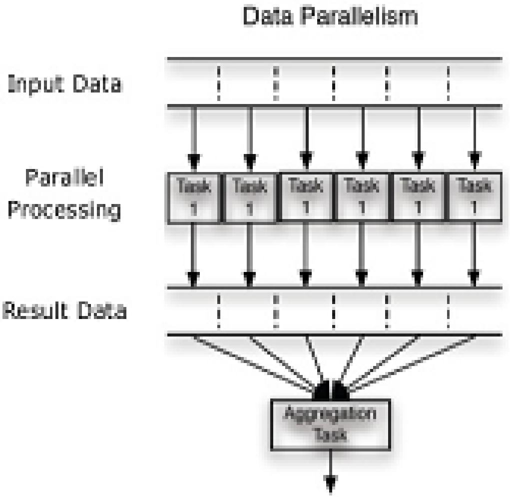
1. Finding independent code segments in a program is **key to unlocking the parallelism** exploitable by multiprocessors
2. **Data-level** parallelism means that computations of different data elements are **independent of each other**.
3. **Data-level** parallelism is often expressed in **loops** and is one of the main targets for parallelizing compilers to unlock parallelism
4. To exploit data-level parallelism in loops, it must make **sure that there is no loop-carried dependency**, meaning that the computations in different iterations are independent of each other.



Metropolitana
Casa abierta al tiempo Azcapotzalco

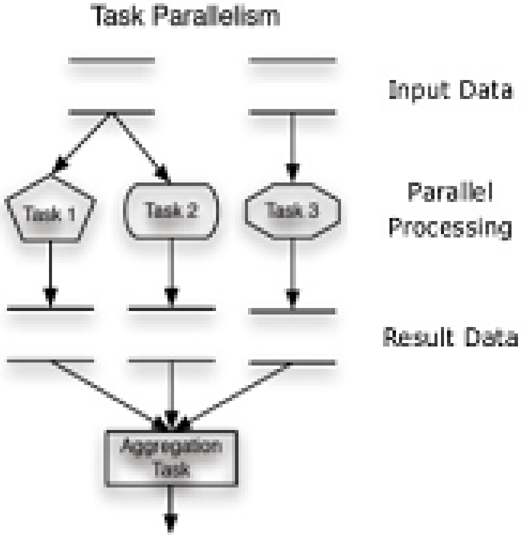
Data level parallelism

1. When the **same computation is applied to all data elements** in an array, the term SPMD parallelism is often used
2. **SPMD** stands for **single-program-multiple-data**
3. The **same function (program code) is applied to all data elements**
4. Data-level parallelism has the attractive property that **as the problem size is scaled up, more parallelism can often be found.**



Task Level Parallelism

1. In this form of parallelism **independent functions are executed on different processors.**
2. These functions can then form a **function pipeline** in which the data stream is processed by applying a **sequence of functions to it.**
3. **Data-level parallelism is applied to each function of a function pipeline to exploit parallelism at two levels.**



Contenido

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

```
sum = 0;
for (i=0,i<N,i++)
    for (j=0,j<N,j++){
        C[i,j] = 0;
        for (k=0,k<N,k++)
            C[i,j] = C[i,j] + A[i,k]*B[k,j];
        sum += C[i,j];
   }
```

Problem

1. Multiplication of two matrices **A and B**
2. Each with **N rows and N columns**
3. The **sum of all matrix elements**
4. The result C is a N$\times$N matrix and a scalar variable sum stores the sum of all matrix elements
5. This program has lots of **data-level parallelism** because the calculation of each individual matrix element is independent of the calculation of others.
6. All matrix elements could be **calculated in parallel**.

```
sum = 0;
for (i=0,i<N,i++)
    for (j=0,j<N,j++){
        C[i,j] = 0;
        for (k=0,k<N,k++)
            C[i,j] = C[i,j] + A[i,k]*B[k,j];
        sum += C[i,j];
    }
```

Problem

1. Multiplication of two matrices **A and B**
2. Each with **N rows and N** columns
3. The **sum of all matrix elements**
4. The result C is a $N \times N$ matrix and a scalar variable sum stores the sum of all matrix elements
5. This program has lots of **data-level parallelism** because the calculation of each individual matrix element is independent of the calculation of others.
6. All matrix elements could be **calculated in parallel**.

Problem

1. Multiplication of two matrices **A and B**
2. Each with **N rows and N columns**
3. The **sum of all matrix elements**
4. The result C is a N×N matrix and a scalar variable sum stores the sum of all matrix elements
5. This program has lots of **data-level parallelism** because the calculation of each individual matrix element is independent of the calculation of others.
6. All matrix elements could be **calculated in parallel**.

```
sum = 0;
for (i=0,i<N,i++)
    for (j=0,j<N,j++){
        C[i,j] = 0;
        for (k=0,k<N,k++)
            C[i,j] = C[i,j] + A[i,k]*B[k,j];
        sum += C[i,j];
   }
```

Shared Memory Systems

1. The **parallelism granularity** is the **amount of work that is carried out by a thread in parallel with other threads.**
2. For a fixed problem size, for example a matrix with $N \times N$ elements, **the granularity decreases with the number of processors.**
3. Under a shared-memory programming model, **variables can be defined globally as shared**.
4. This has the attractive property that the source data **can be accessed by all threads**.
5. Each thread can **read** from the input matrices A and B and **store** its results into matrix C using regular loads and stores

Shared Memory Systems

1. The **parallelism granularity** is the **amount of work that is carried out by a thread in parallel with other threads.**
2. For a fixed problem size, for example a matrix with $N \times N$ elements, **the granularity decreases with the number of processors.**
3. Under a shared-memory programming model, **variables can be defined globally as shared**.
4. This has the attractive property that the source data **can be accessed by all threads**.
5. Each thread can **read** from the input matrices A and B and **store** its results into matrix C using regular loads and stores

Shared Memory Systems

1. The **parallelism granularity** is the **amount of work that is carried out by a thread in parallel with other threads.**
2. For a fixed problem size, for example a matrix with $N \times N$ elements, **the granularity decreases with the number of processors.**
3. Under a shared-memory programming model, **variables can be defined globally as shared**.
4. This has the attractive property that the source data **can be accessed by all threads**.
5. Each thread can **read** from the input matrices A and B and **store** its results into matrix C using regular loads and stores

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. The **parallelism granularity** is the **amount of work that is carried out by a thread in parallel with other threads.**
2. For a fixed problem size, for example a matrix with $N \times N$ elements, **the granularity decreases with the number of processors.**
3. Under a shared-memory programming model, **variables can be defined globally as shared**.
4. This has the attractive property that the source data **can be accessed by all threads**.
5. Each thread can **read** from the input matrices A and B and **store** its results into matrix C using regular loads and stores

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. The **parallelism granularity** is the **amount of work that is carried out by a thread in parallel with other threads.**
2. For a fixed problem size, for example a matrix with $N \times N$ elements, **the granularity decreases with the number of processors.**
3. Under a shared-memory programming model, **variables can be defined globally as shared**.
4. This has the attractive property that the source data **can be accessed by all threads**.
5. Each thread can **read** from the input matrices A and B and **store** its results into matrix C using regular loads and stores

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. Each thread calculates its partial sum and **the partial sums must be accumulated in the global sum**
2. Since there is a **risk** that **multiple threads could each read the global sum into a processor register**, add their **partial sum to it, and then write back the result into the global sum variable, some of the partial sums could be overwritten by others**.
3. Therefore, the partial sums must be added to the **global sum in a serial fashion**, inside a **critical section**.

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. Each thread calculates its partial sum and **the partial sums must be accumulated in the global sum**
2. Since there is a **risk** that **multiple threads could each read the global sum into a processor register**, add their **partial sum to it, and then write back the result into the global sum variable, some of the partial sums could be overwritten by others**.
3. Therefore, the partial sums must be added to the **global sum in a serial fashion**, inside a **critical section**.

Shared Memory Systems

1. Each thread calculates its partial sum and **the partial sums must be accumulated in the global sum**
2. Since there is a **risk** that **multiple threads could each read the global sum into a processor register**, add their **partial sum to it, and then write back the result into the global sum variable, some of the partial sums could be overwritten by others**.
3. Therefore, the partial sums must be added to the **global sum in a serial fashion**, inside a **critical section**.

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. The semantics of a critical section ensures that **at most one thread can execute the code inside of the section at a time.**
2. All threads add their partial sums to the global sum one at a time so that the end result is the correct sum of all partial sums.
3. Since threads run **asynchronously**, it may happen that one thread is finished with its partial matrix product before another one has even started.
4. No thread can enter the critical section and update sum until all threads are done with their part of the matrix multiplication algorithm
5. **Barrier synchronization** forces every thread to stall until all threads have reached the barrier.

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. The semantics of a critical section ensures that **at most one thread can execute the code inside of the section at a time.**
2. All threads add their partial sums to the global sum one at a time so that the end result is the correct sum of all partial sums.
3. Since threads run **asynchronously**, it may happen that one thread is finished with its partial matrix product before another one has even started.
4. No thread can enter the critical section and update sum until all threads are done with their part of the matrix multiplication algorithm
5. **Barrier synchronization** forces every thread to stall until all threads have reached the barrier.

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. The semantics of a critical section ensures that **at most one thread can execute the code inside of the section at a time.**
2. All threads add their partial sums to the global sum one at a time so that the end result is the correct sum of all partial sums.
3. Since threads run **asynchronously**, it may happen that one thread is finished with its partial matrix product before another one has even started.
4. No thread can enter the critical section and update sum until all threads are done with their part of the matrix multiplication algorithm
5. **Barrier synchronization** forces every thread to stall until all threads have reached the barrier.

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. The semantics of a critical section ensures that **at most one thread can execute the code inside of the section at a time.**
2. All threads add their partial sums to the global sum one at a time so that the end result is the correct sum of all partial sums.
3. Since threads run **asynchronously**, it may happen that one thread is finished with its partial matrix product before another one has even started.
4. No thread can enter the critical section and update sum until all threads are done with their part of the matrix multiplication algorithm
5. **Barrier synchronization** forces every thread to stall until all threads have reached the barrier.

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. The semantics of a critical section ensures that **at most one thread can execute the code inside of the section at a time.**
2. All threads add their partial sums to the global sum one at a time so that the end result is the correct sum of all partial sums.
3. Since threads run **asynchronously**, it may happen that one thread is finished with its partial matrix product before another one has even started.
4. No thread can enter the critical section and update sum until all threads are done with their part of the matrix multiplication algorithm
5. **Barrier synchronization** forces every thread to stall until all threads have reached the barrier.

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Shared Memory Systems

1. Shared-memory multiprocessors have the attractive property that communication and synchronization can be carried out by primitives available in the ISA
2. Loads and stores may sometimes suffer from very long access latencies
3. The design of the underlying memory system is critical to performance.

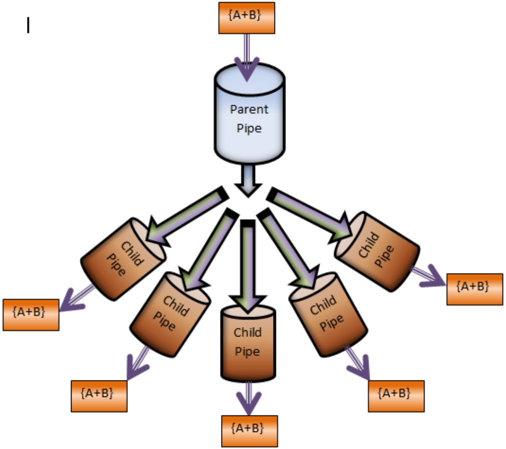
```
/* A, B, C, BAR, LV and sum are shared
/* All other variables are private
low = pid*N/nproc;    /* pid=0...nproc-1
hi = low + N/nproc;   /* identifies rows of A
mysum = 0; sum = 0;
for (i=low,i<hi,i++)
    for (j=0,j<N,j++){
        C[i,j] = 0;
        for (k=0,k<N,k++)
            C[i,j] = C[i,j] +  A[i,k]*B[k,j];
        mysum +=C[i,j];
    }
BARRIER(BAR);
LOCK(LV);
    sum += mysum;
UNLOCK(LV);
```

Contenido

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Introduction

1. In shared-memory systems, where **coordination happens through an address space shared among all threads**
2. In the message-passing programming model, each thread or process has **its own address space**.
3. **Coordination** is carried out by sending **explicit messages** between the threads.



Message-passing systems

1. Message passing can be supported in any system that consists of a number of **interconnected computational nodes** and where **each node has at least one processor and some memory**.
2. Two major implications:
3. Data structures must be explicitly **distributed to the private address spaces**
4. Results from the partial computations performed by the threads must be **collected at the end**

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Message-passing systems

1. Multiplication of matrices A and B
2. A and B are initially kept in one computational node, the **master node**
3. The master node **partitions** the matrices across the nodes
4. **Send** and **Receive**
5. **SEND** copies data from the sender's local address space to a buffer at the receiver's side
6. **RECV** copies data from that buffer to the local address space at the receiver's side

```
myN = N/nproc;
if(pid == 0)
  for(i=1; i<nproc;i++){
    k=i*N/nproc;
    SEND(&A[k][0],myN*N*sizeof(float),i,IN1);
    SEND(&B[0][0],N*N*sizeof(float),i,IN2);
  } else {
    RECV(&A[0][0],myN*N*sizeof(float),0,IN1);
    RECV(&B[0][0],N*N*sizeof(float),0,IN2);
  }
mysum = 0;
for (i=0,i<myN, i++)
  for (j=0,j<N, j++){
     C[i,j] = 0;
     for (k=0,k<N, k++)
        C[i,j] = C[i,j] + A[i,k]*B[k,j];
     mysum += C[i,j];
  }
 if (pid == 0){
   sum = mysum;
   for(i = 1;i<nproc;i++){
      RECV(&mysum,sizeof(float),i,SUM);
      sum += mysum;
   }
   for(i=1; i<nproc;i++){
      k=i*N/nproc;
      RECV(&C[k][0],myN*N*sizeof(float),i,RES);
   }
 } else{
     SEND(&mysum,sizeof(float),0,SUM);
     SEND(&C[0][0],myN*N*sizeof(float),0,RES);
   }
```

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

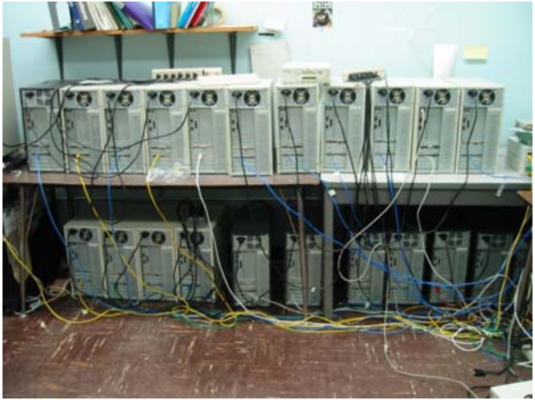
Message-passing systems

1. Each process works on the partition assigned
2. Local portions of the result matrix must later be copied back to the master node
3. In the message-passing system implementation, **synchronization** is **implicit** in the message-passing primitives.
4. SENDs and RECVs usually come in twomain flavors: **synchronous** and **asynchronous**
5. Synchronous SENDs and RECVs **block until both parties have notified each other that the message has been exchanged**

```
myN = N/nproc;
if(pid == 0)
  for(i=1; i<nproc;i++){
    k=i*N/nproc;
    SEND(&A[k][0],myN*N*sizeof(float),i,IN1);
    SEND(&B[0][0],N*N*sizeof(float),i,IN2);
  } else {
    RECV(&A[0][0],myN*N*sizeof(float),0,IN1);
    RECV(&B[0][0],N*N*sizeof(float),0,IN2);
  }
mysum = 0;
for (i=0,i<myN, i++)
  for (j=0,j<N, j++){
     C[i,j] = 0;
     for (k=0,k<N, k++)
        C[i,j] = C[i,j] + A[i,k]*B[k,j];
     mysum += C[i,j];
  }
 if (pid == 0){
   sum = mysum;
   for(i = 1;i<nproc;i++){
      RECV(&mysum,sizeof(float),i,SUM);
      sum += mysum;
   }
   for(i=1; i<nproc;i++){
      k=i*N/nproc;
      RECV(&C[k][0],myN*N*sizeof(float),i,RES);
   }
 } else{
     SEND(&mysum,sizeof(float),0,SUM);
     SEND(&C[0][0],myN*N*sizeof(float),0,RES);
   }
```

Message-passing multiprocessor

1. In Message-passing multiprocessor architectures **participating nodes exchange explicit messages** among each other
2. Only the nodes involved in a particular exchange are contacted
3. Message-passing systems can be easily built on top of a cluster of desktop/laptop machines as well as embedded on top of a shared-memory system.



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Message-passing primitives

1. **Message exchanges** are the fundamental primitive for **synchronization** as well as **communication**
2. Synchronous message-passing
3. A matching pair of SEND and RECV primitives sets up a communication path
4. Data can be copied from a designated location in the sender's local address space to a designated location in the receiver's local address space

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Message-passing primitives

1. **SEND** (location in the local address space (&A), length of the message (sizeof(A)), ID of the receiver (P2), tag to distinguish the message from other messages sent between P1 and P2)
2. RECV (location in the receiver's local address space (&B),the size of B (which should match the size in the SEND), the ID of the sender (P1), and the same tag as the matching SEND)
3. P1 and P2 run **asynchronously**, the issue for the SEND RECV protocol is to **guarantee** that the content of A will be transferred and correctly copied into B before P2 executes its statement after returning from the RECV, which actually happens to change the content of B.

```
Code for process P1:
A=10;
SEND(&A,sizeof(A),P2,SEND_A);
A=A+1;
RECV(&C,sizeof(C),P2,SEND_B);
printf(C);
```

```
Code for process P2:
B=5;
 RECV(&B,sizeof(B),P1,SEND_A);
B=B+1;
 SEND(&B,sizeof(B),P1,SEND_B);
```

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Synchronous message passing

1. The sender is **block** until the message has been received by the receiver
2. The receiver is **block** until the message is available and has been copied into the designated data structure in its local address space.
3. There are **two disadvantages** with synchronous message exchanges: one is that they are prone to **deadlock**
4. The second is that they do not allow overlapping communication with computation

```
Code for process P1:
A = 10;
SEND(&A,sizeof(A),P2,SEND_A);
RECV(&B,sizeof(B),P2,SEND_B);
```

```
Code for process P2:
B = 5;
SEND(&B,sizeof(B),P1,SEND_B)
RECV(&A,sizeof(B),P1,SEND_A);
```

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Asynchronous message passing

```
Code for process P1:
A=10;
SEND(&A,sizeof(A),P2,SEND_A);
    <UNRELATED COMPUTATION;>
RECV(&B,sizeof(B),P2,SEND_B);
```

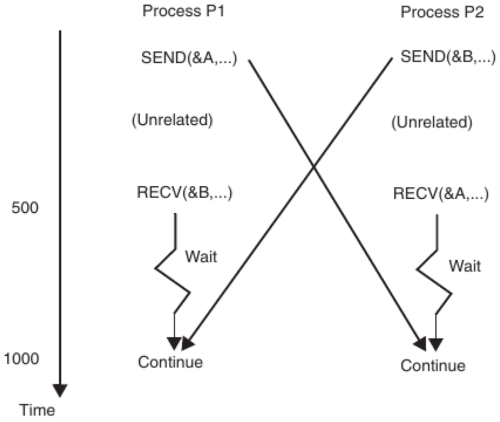
```
Code for process P2:
B=5;
SEND(&B,sizeof(B),P1,SEND_B);
    <UNRELATED COMPUTATION;>
RECV(&A,sizeof(B),P1,SEND_A);
```

1. Synchronous messages transfer is that they **combine synchronization with communication**
2. **Long-latency operations**, and packing them into a single primitive can lead to **performance losses**
3. **Separating** them could allow the sender to **do useful work while the message is in transit**.
4. **Asynchronous message passing** primitives do just that

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Asynchronous message passing

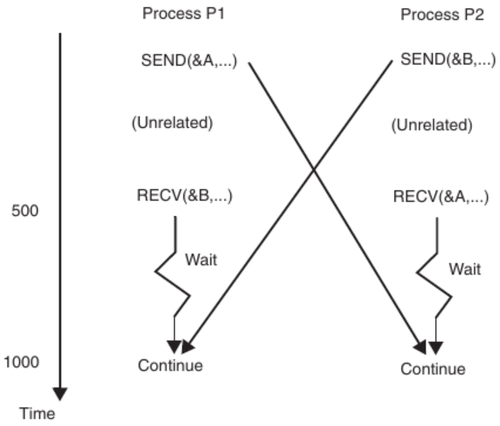
1. P1 sends the message and then moves on to execute the code after its SEND, **without waiting** for the receiver to copy the message into its address space.
2. The code after SEND is, in this case, **unrelated** in the sense that it does not change the content of A, nor does it need the value of B



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Asynchronous message passing

1. It could happen that the content of A that is copied does not correspond to what it was at the time the SEND was executed, as the message transfer happens asynchronously with the execution of the code
2. **Two forms of asynchronous message-passing primitives exist, blocking and non-blocking message-passing primitives.**
3. In **asynchronous message** passing the sender does not necessarily wait until the data in its local address space have been copied.



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Blocking and non-blocking message-passing

1. A **blocking asynchronous** SEND gives back control to the sending process once a copy of the local data making up the message has been buffered somewhere and cannot be affected by the execution of the sending process
2. A **blocking RECV** does not give back control to the receiving process until after the message has been copied into the local address space of the receiver.
3. **Bon-blocking asynchronous** message-passing primitives, where the control is returned to the sender and receiver immediately and where the transfer happens in the background

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

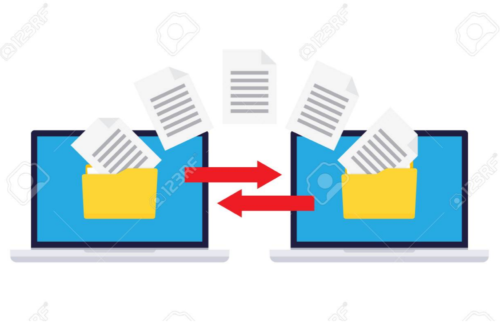
Probe function

1. Some message-passing systems also provide a probe function that can **interrogate the status of the message transfer**
2. Probe functions make it possible to **check whether data has been copied from the local address space of the sender** to a **buffer** or whether it has been copied into the **local address space of the receiver**

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Introduction

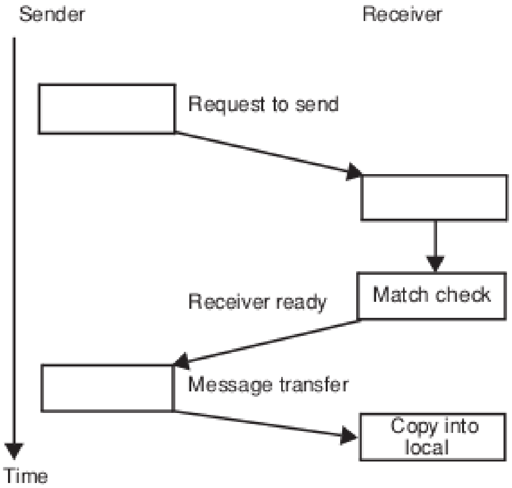
1. Consider a **synchronous message transfer**
2. When sender executes its SEND, it **needs to synchronize** with the receiver to make sure that it has reached its matching RECV.
3. At this point, the message with **the data copied from the local address space of the sender** can be sent off to the receiver, which then **copies it into the location specified in its local address space**



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Message-passing protocol

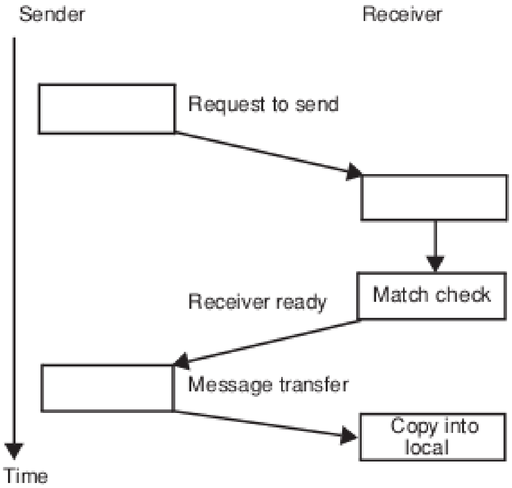
1. A protocol is used to implement the message transfer
2. When the sender process invokes the SEND function, a **message asking whether the receiver is ready** is sent to the receiver ("Request to send")
3. In order for the receiving node to figure out whether it is ready for the message transfer, it **interrogates a table – the match table –** which keeps track of the status of all RECVs



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

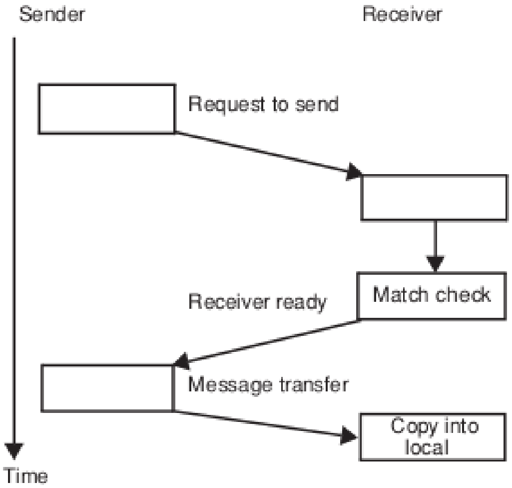
Message-passing protocol

1. There are two cases: **(1) the matching RECV has already been executed**, or **(2) the matching RECV has not yet been executed.**
2. If the receiver process has **already executed the matching RECV**, there will be a **match in the table** and the receiving node notifies the sending node that it is ready for the message transfer ("**Receiver ready**")



Message-passing protocol

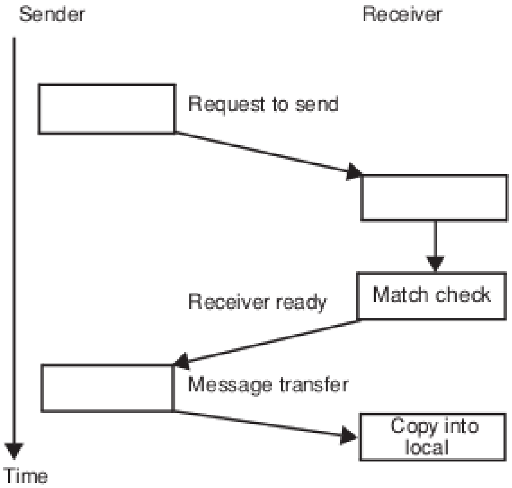
1. There are two cases: **(1) the matching RECV has already been executed**, or **(2) the matching RECV has not yet been executed.**
2. If the receiver process has **already executed the matching RECV**, there will be a **match in the table** and the receiving node notifies the sending node that it is ready for the message transfer ("**Receiver ready**")
3. The sender then sends the message to the receiver, which copies it into the specified area in its local address space.



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Message-passing protocol

1. If the receiving node has not yet executed the RECV function, a **“Receiver ready” message does not go out until the matching RECV function is executed** and the match table is updated accordingly
2. A three-phase protocol is needed to send the message.
3. Protocol for synchronous message-passing primitives
4. This means that the sender and the receiver processes are **both blocked until the message transfer is completed**



Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Message-passing protocol

1. ¿What happens under blocking asynchronous message passing?
2. Under blocking asynchronous message passing, the sender can continue executing past the SEND function **once the message is buffered**
3. Buffering space has to be reserved to host a copy of the message data of the sender before the sender can resume its execution past the SEND function
4. copy of the content of the local data structure is created in parallel with sending a request to the receiver
5. Once the copy is created, the sender can resume its execution past the SEND function.

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Introduction

1. Message-passing systems can be built on nodes connected by an interconnection network
2. The protocols can be built upon the supported low-level network transaction primitives.
3. These primitive network transactions are provided by general interconnection networks.
4. Additional hardware support for message transfer aims to cut down the latency of sending a message from one node to another

Universidad
Autónoma
Metropolitana
Casa abierta al tiempo Azcapotzalco