CS320/CSE 302/ECE 392

Introduction to Parallel Programming for Scientists and Engineers

Spring 1999

i. Course organization

Instructor:	**Office Hours:**
David Padua. padua@uiuc.edu, 3318 DCL, 3-4223.	By Appointment
T.A.:	**Office Hours:**
Zehra Sura	10:30-11:30 Mondays and Fridays

Textbook

Class notes.

Hyperlinks (mandatory)

URL

http://polaris.cs.uiuc.edu/~padua/cs320

Grading:

5-8 Machine Problems	50%
Midterm (Th. March 11)	25%
Final	25%

Graduate students registered for 1 unit need to complete additional work (associated with each MP).

ii. Topics

- Machine models.
- Parallel programming models.
- Language extensions to express parallelism:

  OpenMP (Fortran) and MPI (Fortran or C).

  If time allows: High-Performance Fortran, Linda, pC++, SplitC.
- Issues in algorithm design

  Parallelism

  Load balancing

  Communication

  Locality

- Algorithms.

  Linear algebra algorithms such as matrix multiplication and equation solvers.

  Symbolic algorithms such as sorting.

  N-body

  Random number generators.

  Asynchronous algorithms.

- Program analysis and transformation

  Dependence analysis

  Race conditions

  Deadlock detection

- Parallel program development and maintenance

  Modularity

  Performance analysis and tuning

  Debugging

Chapter 1. Introduction

Parallelism

- The idea is simple: improve performance by performing two or more operations at the same time.
- Has been an important computer design strategy since the beginning.
- It takes many (complementary forms) within conventional systems like uniprocessor PCs and workstations:

  At the circuit level: Adders and multipliers do not consider one digit at a time but rather operate on several digits at the same time. This design strategy was used even by Charles Babbage in his mechanical computer design of last century.

  At the processor-design level: The execution of instructions and floating-point operations is usually pipelined. Several instructions can execute simultaneously.

  At the system level: Computation and I/O can proceed simultaneously. This is why multiprogramming increases throughput.

- However, the design strategy of interest to us is to attain parallelism by connecting several processors or even several complete computers.

- It has also been claimed that parallel computing is necessary for continuing performance gains given that *“clock times are decreasing slowly and appear to be approaching physical limits”.* This, however, is not universally accepted.

- “Assuming that nothing basic in manufacturing or physics breaks in the next couple of years, there is no reason the historical trend in microprocessor performance will slow.” Robert Colwell, Computer January 1998.

- However, despite all the advances in semiconductor technology, parallel computers are becoming increasingly popular. It is an important strategy to increase performance over what is possible by technology alone.

- Another important reason for the development of parallel systems of the multicomputer variety is availability.

  “Having a computer shrivel up into an expensive doorstop can be a whole lot less traumatic if it’s not unique, but rather one of a herd. The herd should be able to accomodate spares, which can potentially be used to keep the work going; or if one chooses to configure sparelessly, the work that was done by the dear departed sibling can, potentially, be redistributed among the others.” *In search of clusters. G. Pfister. Prentice Hall.*

Moore’s Law

Named after a 1964 observation by Gordon E. Moore of Intel.

It holds that “The number of elements in advanced integrated circuits doubles every year.”

Many claim that the era of this law is coming to an end sometime around 2010 for several reasons:

- The cost of chip factories will increase enormously.
- The small number of electrons moved at these new small feature sizes may not be enough for reliable computing.
- The feature sizes will be so small that quantum effects will start having an impact.

Applications

- Traditionally, highly parallel computers have been used for numerical simulations of complex systems such as the weather, mechanical devices, electronic circuits, manufacturing processes, chemical reactions, etc.

  See the 1991 report from the General Accounting Office on Industry uses of supercomputers and high-speed networks.

- Perhaps the most important government program in parallel computing today is the

  Accelerated Strategic Computing Initiative (ASCI).

  Its main objective is to accurately simulate nuclear weapons in order to verify safety, reliability, and performance of the US nuclear stockpile. Several highly-parallel computers (1000s of processors) from Intel, IBM, and SGI are now being used to develop these simulation

- Commercial applications are also important today. Examples include: transaction processing systems, web servers, junk-mail generators, etc. These applications will probably become the main driving force behind parallel computing in the future.

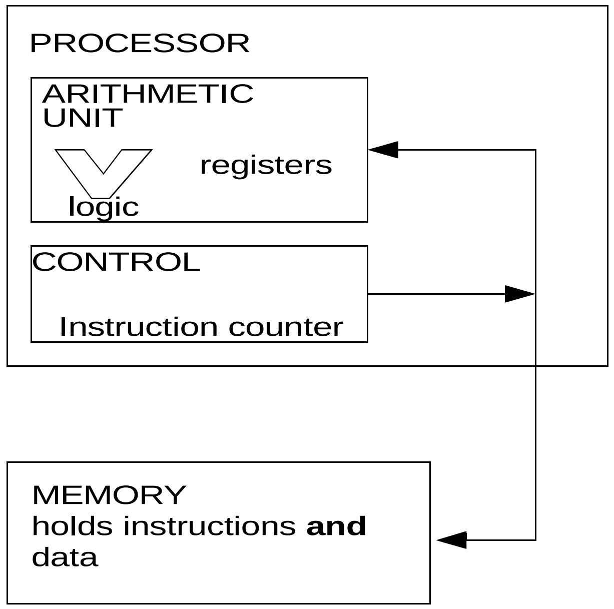
- We will focus on numerical simulations due to their importance for scientists and engineers.
- Computer simulation is considered today as a third mode of scientific research. It complements experimentation and theoretical analysis.
- Furthermore, simulation is an important engineering tool that provides fast feedback on the quality and feasibility of new designs.

Chapter 2. Machine models

2.1 The Von Neumann computational model

Discussion taken from Almasi and Gottlieb: Highly Parallel Computing. Benjamin Cummings, 1988.

- Designed by John Von Neumann about fifty years ago.
- All widely used “conventional” machines follow this model. It is represented next:



- The machine’s essential features are:
  1. A processor that performs instructions such as “add the contents of these two registers and put the result in that register”
  2. A memory that stores both the instructions and data of a program in cells having unique addresses.
  3. A control scheme that fetches one instruction after another from the memory for execution by the processor, and shuttles data one word at a time between memory and processor.
- Notice that machines today usually are programmed in a high level language containing statements such as

  A = B + C

  However, these statements are translated by a compiler into the machine instructions just mentioned. For example, the previous assignment statement would be translated into a sequence of the form:

`LD 1,B` (load B from memory into register 1)

`LD 2,C` (load C from memory into register 2)

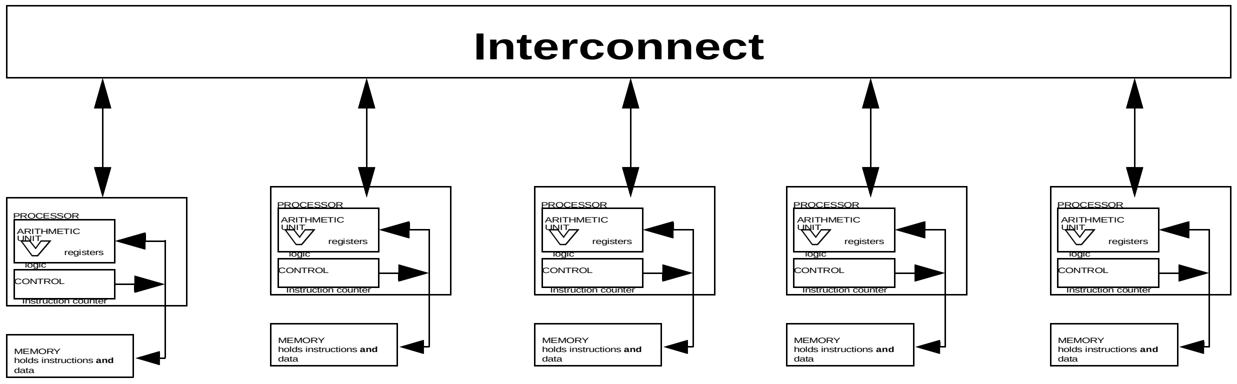
`ADD 3,1,2` (add registers 1 and 2 and put the result into register 3)

`ST 3,A` (store register 3’s contents into variable A’s address in memory)

- It is said that the compiler creates a “virtual machine” with its own language and computational model.
- Virtual machines represented by conventional languages, such as Fortran 77 and C, also follow the Von Neumann model.

2.2 Multicomputers

- The easiest way to get parallelism given a collection of conventional computers is to connect them:



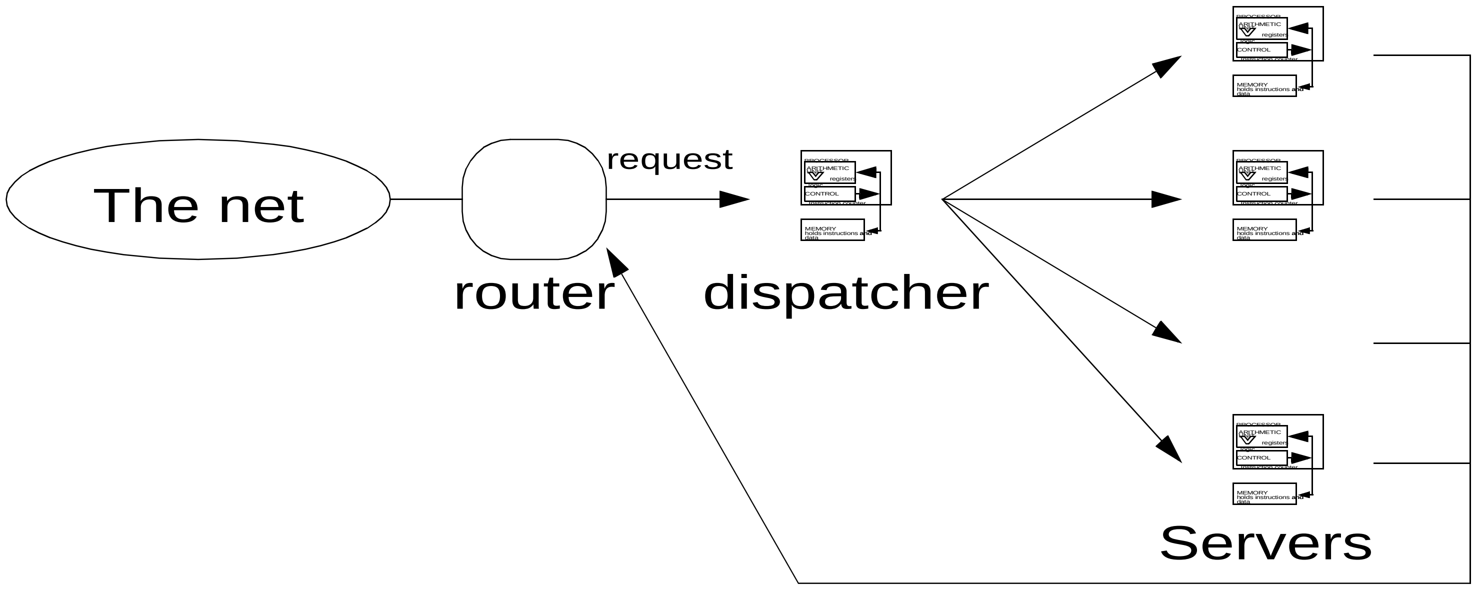
- Each machine can proceed independently and communicate with the others via the interconnection network.

- There are two main classes of multicomputers: clusters and distributed-memory multiprocessors. They are quite similar, but the latter is considered a single computer and is sold as such.

Furthermore, a cluster consists of a collection of interconnected **whole computers** (including I/O) used as a single, unified computing resource.

Not all nodes of a distributed memory multiprocessor (such as IBMs SP-2) need have complete I/O resources.

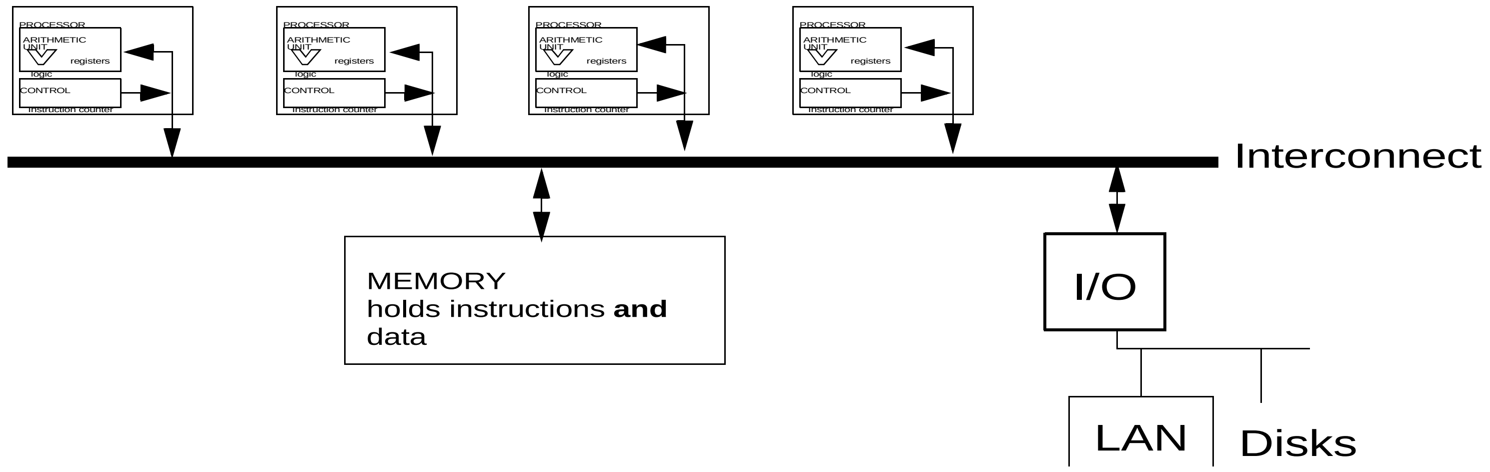
- An example of cluster is a web server



- Another example is the workstation cluster at Fermilab, which consists of about 400 Silicon Graphics and IBM workstations. The system is used to analyze accelerator events. Analyzing any one of those events has nothing to do with analyzing any of the others. Each machine runs a sequential program that analyzes one event at a time. By using several machines it is possible to analyze many events simultaneously.

2.3 Shared-memory multiprocessors

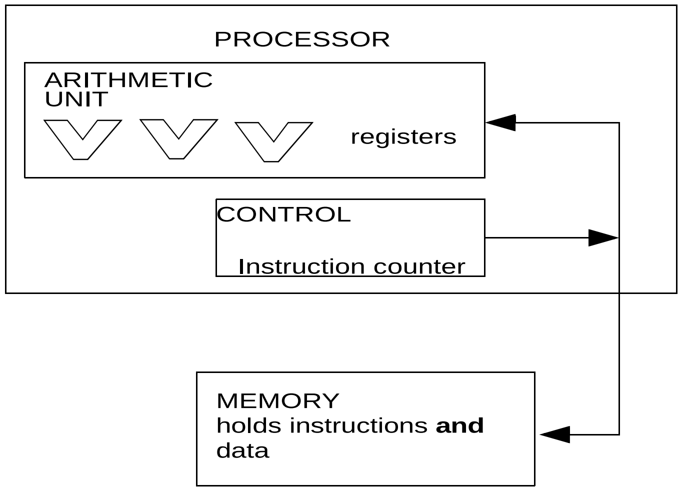
- The simplest form of a shared-memory multiprocessor is the symmetric multiprocessor (SMP). By symmetric we mean that each of the processors has exactly the same abilities. Therefore any processor can do anything: they all have equal access to every location in memory; they all can control every I/O device equally well, etc. In effect, from the point of view of each processor the rest of the machine looks the same, hence the term symmetric.



- An important component of SMPs are caches. These will be discussed later.

2.4 Other forms of parallelism

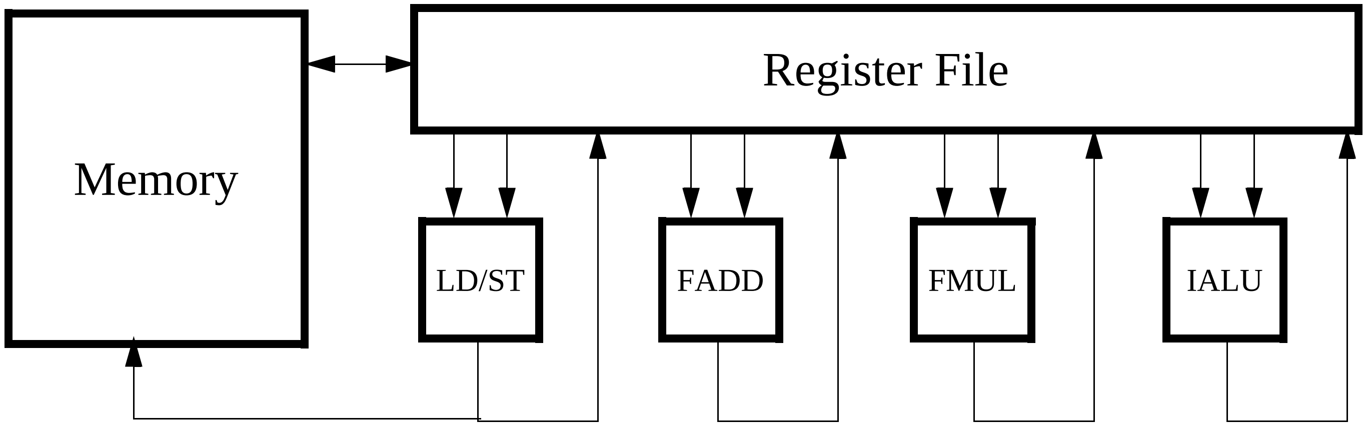
- As discussed above, there are other forms of parallelism that are widely used today. These usually coexist with the coarse grain parallelism of multicomputers and multiprocessors.
- Pipelining of the control unit and/or arithmetic unit.
- Multiple functional units



- Most microprocessors today take advantage of this type of parallelism.

- VLIW (Very Long Instruction Word) processors are an important class of multifunctional processors. The idea is that each instruction may involve several operations that are performed simultaneously.This parallelism is usually exploited by the compiler and not accessible to the high-level language programmer. However, the programmer can control this type of parallelism in assembly language.

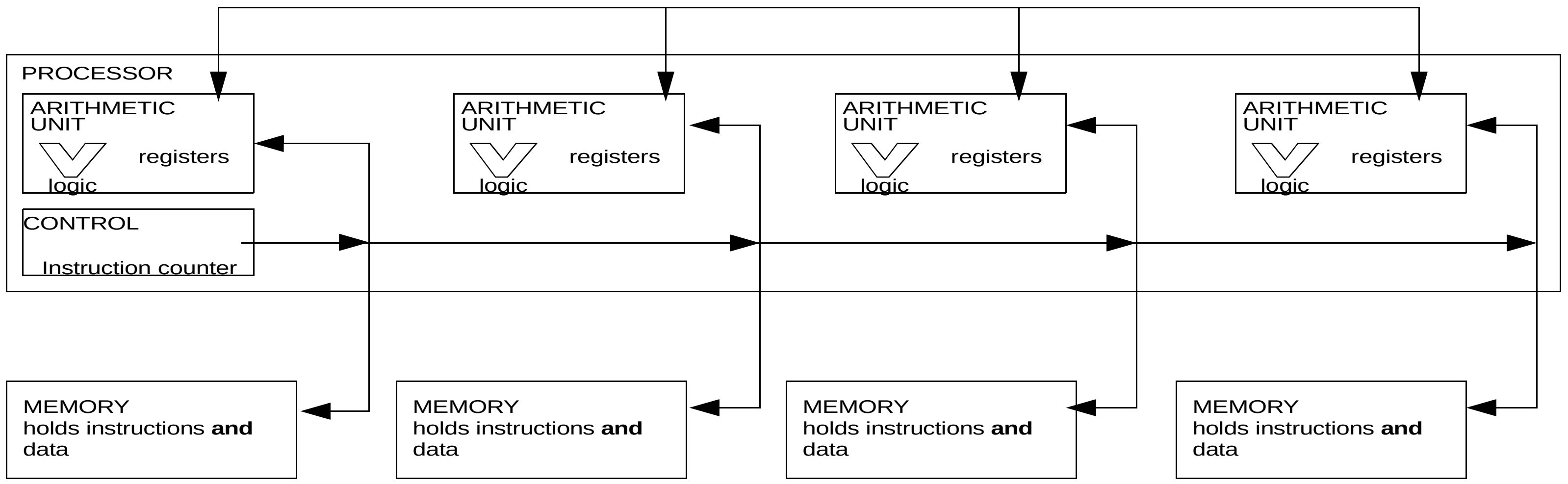
Multifunction Processor (VLIW)



Instruction Word

LD/ST	FADD	FMUL	IALU	BRANCH

- Array processors. Multiple arithmetic units


PROCESSOR
ARITHMETIC UNIT
logic
registers
ARITHMETIC UNIT
logic
registers
ARITHMETIC UNIT
logic
registers
ARITHMETIC UNIT
logic
registers
CONTROL
Instruction counter
MEMORY holds instructions **and** data
MEMORY holds instructions **and** data
MEMORY holds instructions **and** data
MEMORY holds instructions **and** data

- Illiac IV is the earliest example of this type of machine. Each arithmetic unit (processing unit) of the Illiac IV was connected to four others to form a two-dimensional array (torus).

2.5 Flynn’s taxonomy

- Michael Flynn published a paper in 1972 in which he picked two characteristics of computers and tried all four possible combinations. Two stuck in everybody’s mind, and the others didn’t:
- SISD: Single Instruction, Single Data. Conventional Von Neumann computers.
- MIMD: Multiple Instruction, Multiple Data. Multicomputers and multiprocessors.
- SIMD: Single Instruction, Multiple Data. Array processors.
- MISD: Multiple Instruction, Single Data. Not used and perhaps not meaningful.

Chapter 3. Parallel Programming Models

- There are many different parallel programming paradigms. Most are of academic interest only.
- We will present three paradigms that are (or are likely to become) popular with real application programmers:

  Shared-memory programming
  Message-passing programming
  Array programming
- We will start by introducing the notion of task

Tasks

Tasks are a central concept in parallel programming.

A task is a sequential program under execution. The programmer may assume that there is a processor devoted to each task. (There may not be a physical processor, but the operating system will time-share the real processors to give the illusion of one processor per task. It is said that the operating system creates a “virtual machine”.)

Parallel programs consist of two or more tasks. Each task may contain private data (local memory). That is, data that only the tasks can access.

There are two main programming approaches used frequently to generate tasks:

1. Explicit spawning.
2. Programming in the SPMD (Single Program Multiple Data) model.

The SPMD model will be discussed shortly.

We will illustrate the explicitly spawning strategy next in the context of shared-memory parallel programming.

Shared-Memory Parallel Programming

To illustrate this model ,consider the following very simple program

```
read b,c,e,g
a=f1(b,c)
h=f2(e)
d=f3(h, g)
q=f4(d,a)
print q
end
```

where `f1`, `f2`, `f3`, and `f4` are very complex functions, and a, b, c, d, e, p, and q are complex data structures.

A simple parallel program for this is:

```
read b,c,e,g
start_task sub(e,g,d)
a=f1(b,c)
wait_for_all_tasks_to_complete
q=f4(d,a)
print q
end


subroutine sub(e,g,d)
local h
h=f2(e)
d=f3(h, g)
end sub
```

The program starts as a single task program and then a second task is initiated. The second task proceeds by executing subroutine sub.

The computations of variable a and variable d proceed in parallel.

The original task waits for the second task to complete before proceeding with the computation of q. Waiting is necessary to guarantee that d has been computed before it is used.

In this program, all variables except for h can be shared by the two tasks. In fact, only variables e, g, and d are accessed by both tasks.

Notice that because h is private to the task, a new copy of h will be created every time `start_task` sub is executed.

Consider the following program:

```
read b,c,e,g
start_task sub(b,c,a)
call sub(e,g,d)
wait_for_all_tasks_to_complete
q=f4(d,a)
print q
end


subroutine sub(e,g,d)
local h
h=f2(e)
d=f3(h, g)
end sub
```

Two copies of sub will be executing simultaneously, and each will have its own copy of h.

Channels and Message Passing

In message passing programming, all variables are private. Variable are shared by explicitly “writing” and “reading” data to and from tasks.

The following code is equivalent to the first shared-memory code presented above:

```
read b,c,e,g
start_task sub()
send x(e,g)
a=f1(b,c)
receive y(d)
q=f4(d,a)
print q
end

subroutine sub()
local m,n,r,h
receive x(m,n)
h=f2(m)
r=f3(h, n)
send y(r)
end sub
```

Here, `x` and `y` are communication channels. The `send` operation is asynchronous: it completes immediately.

The `receive` operation is synchronous: it halts execution of the task until the necessary data is available.

Thus, thanks to the `send` and `receive` operations, the values of variables `e` and `g` are tranferred from the original task to the created task, and the value of `d` is transferred in the opposite direction.

Notice that no wait operation is needed. The `receive y(d)` operation does the necessary synchronization.

An alternative approach, message passing, uses the names of the destination tasks rather than channels for communication.

Usually, message passing systems start one task per processor, all executing the same code. This is the SPMD model mentioned above. It is the most popular model today.

The previous program in SPMD and message passing:

```
if my_proc().eq. 0 then
   read b,c,e,g
   send (e,g) to 1
   a=f1(b,c)
   receive (d) from 1
   q=f4(d,a)
   print q
else /* my_proc() == 1  */
   receive (m,n) from 0
   h=f2(m)
   r=f3(h, n)
   send (r) to 0
end if
```

Later in the semester we will study this approach in more detail

Parallel loops

One of the most popuar constructs for shared-memory programming is the parallel loop.

Parallel loops are just like any usual iterative statement, such as for or do, except that it doesn’t actually iterate. Instead, it says “just get all these things done, in any order, using as many processors as possible.”

The number of processors available to the job may be specified or limited in some way, but that’s usually outside the domain of the parallel loop construct.

An example of a parallel loop is:

```
c = sin (d)
parallel do i=1 to 30
   a(i) = b(i) + c
end parallel do
e = a(20)+ a(15)
```

Parallel loops are implemented using tasks. For example, the previous program could be translated by the compiler into something similar to the following program:

```
c = sin (d)
start_task sub(a,b,c,1,10)
start_task sub(a,b,c,11,20)
call sub(a,b,c,21,30)
wait_for_all_tasks_to_complete
e = a(20)+ a(15)
...

subroutine sub(a,b,c,k,l)
   ...
   for i=k to l
      a(i) = b(i) + c
   end for
end sub
```

Notice that, in this program, arrays a and b are shared by the three processors cooperating in the execution of the loop.

This program assigns to each processor a fixed segment of the iteration space. This is called static scheduling.

Scheduling also could be dynamic. Thus, the compiler could generate the following code instead:

```
c = sin (d)
start_task sub(a,b,c)
start_task sub(a,b,c)
call sub(a,b,c)
wait_for_all_tasks_to_complete
e = a(20)+ a(15)
...

subroutine sub(a,b,c)
logical empty
   ...
   call get_another_iteration(empty,i)
   while .not. empty do
      a(i) = b(i) + c
      call get_another_iteration(empty,i)
   end while
end sub
```

Here, the `get_another_iteration()` subroutine accesses a pool containing all n iteration numbers , gets one of them, and removes it from the pool. When all iterations have been assigned, and therefore the pool is empty, the function returns `.true.` in variable `empty`.

A third alternative is to have `get_another_iteration()` return a range of iterations instead of a single iteration:

```
subroutine sub(a,b,c,k,l)
logical empty
   ...
   call get_another_iteration(empty,i,j)
   while .not. empty do
      for k=i to j
         a(k) = b(k) + c
      end for
      call get_another_iteration(empty,i,j)
   end while
end sub
```

SPMD model and parallel loops

Starting a task is usually very time consuming.

For that reason, current implementations of extensions involving parallel loops start a few tasks at the beginning of the program or when the first parallel loop is to be executed.

These tasks, called *implicit tasks*, will be idle during the execution of the sequential components (that is, outside the parallel loops). The task that starts exectuiton of the program is called the *master task*. When the master task finds a parallel loop, it awakes the implicit tasks who join in the execution of the parallel loop.

An asternative strategy is to use the SPMD model as illustrated next:

```
c = sin (d)
i=my_proc()*10
call sub(a,b,c,i+1,i+10)
call barrier()
if my_proc() .eq. 0 then
   e = a(20)+ a(15)
end if
...

subroutine sub(a,b,c,k,l)
   ...
   for i=k to l
      a(i) = b(i) + c
   end for
end sub
```

Array Programming

In these languages, array operations are written in a compact form that often makes programs more readable.

Consider the loop:

```
s=0
do i=1,n
   a(i)=b(i)+c(i)
   s=s+a(i)
end do
```

It can be written (in Fortran 90 notation) as follows:

```
a(1:n) = b(1:n) +c(1:n)
s=sum(a(1:n))
```

Perhaps the most important array language is Kenneth Iverson’s APL, developed ca. 1960.

A popular array language today is MATLAB.

Although these languages were not developed for parallel computing but rather for expressiveness, they can be used to express parallelism since array operations can be easily executed in parallel.

Thus, all the arithmetic operations (+, -, * /, **) involved in a vector expression can be performed in parallel. Intrinsic reduction functions ,such as `sum` above, also can be performed in parallel but in a less obvious manner.

Vector operations can be easily executed in parallel using almost any form of parallelism, from pipelining to multicomputers.

Vector programs are easily translated for execution on shared-memory machines. The code segment:

```
c = sin(d)
a(1:30)=b(2:31) + c
e=a(20)+a(15)
```

is equivalent to the following code segement:

```
c = sin (d)
pafallel do i=1 to 30
   a(i) = b(i+1) + c
end parallel do
e = a(20)+ a(15)
```

Going in the other direction, it is not always simple to transform forall loops into vector operations. For example, how would you transform the following loop into vector form?

```
parallel do i=1 to n
   if c(i) .eq. 1 then
      while a(i) .gt. eps do
         a(i) = a(i) - a(i) / c
      end while
   else
      while a(i) .lt. upper do
         a(i) = a(i) + a(i) * d
      end while
   end if
end parallel do
```