Parallel Programming Models

- There are many different parallel programming paradigms. Most are of academic interest only.
- We will present three paradigms that are 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 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 time-consuming functions, and `a`, `b`, `c`, `d`, `e`, `p`, and `q` are 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 several processors if 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)
parallel 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
```