

## **Control, Synchronization, Scheduling**

- Three main activities:
  - scheduling – deciding what is done where, i.e., assigning specific pieces of work to specific processors
  - synchronization – deciding when things are done, i.e., making sure data that is needed is actually valid
  - control – moving between sequential and parallel execution, distributing information needed to perform work, i.e., some of the support operations needed to support parallel operation, scheduling and synchronization

- require a combination of HW and SW support
- we will concentrate on these operations on a shared memory machine (physically and software paradigmatically)

## Scheduling

- Given parallel execution of a set of tasks, our main goal is to minimize execution time.
- a key task is to schedule the work on the processors in such a way as to have the completion times of the processors as close as possible
- in this case the load is said to be balanced and, assuming the tasks are all independent and overhead is ignored, this minimizes execution time

- considerations:
  - scheduling overhead
  - task granularity
  - task time profile

- two main strategy types – static and dynamic
- static
  - various forms
  - grouping of tasks into assignments of work to particular processors is done at compile time
  - distribution may be completely independent of runtime configuration but usually depends on one runtime system parameter value – the number of processors

- dynamic
  - various forms
  - often depends on efficient HW and low-level SW support
  - grouping of tasks into assignments of work to particular processors is done at runtime
  - actual assignment is often nondeterministic
  - best for load balancing

## Static blockwise scheduling:

- given  $p$  processors and  $n$  tasks allocate contiguous blocks of  $b$  tasks
- if  $n = q \times p + r$  then  $r$  processors get  $b = q + 1$  and the rest get  $q$
- Example:  $n = 5, p = 3 \rightarrow q = 1$  and  $r = 2$  therefore processor 1 gets tasks 1,2 processor 2 gets tasks 3,4 and processor 3 gets task 5
- reasons: easy, maximum granularity if all tasks equal size
- problems: load imbalance if not equal sizes, size may not be known

## Static interleaved (simple and reflected)

- suppose tasks 1 to  $n$  take decreasing amounts of time, e.g., task 1 takes  $W$  units, task 2 takes  $W - 1$ ,  $\dots$ .
- static scheduling of contiguous blocks of tasks will lead to placing all large tasks on one processor leading to imbalance
- static interleaved attempts to remedy this
- two types simple and reflected
- both depend on assumption of decreasing task sizes

Assume Task 1 takes 6 units, Task 2 takes 5 units,  
Task 3 takes 4 units, Task 4 takes 3 units,  
Task 5 takes 2 units, Task 6 takes 1 unit

Contiguous:	Proc 1	Proc 2	Proc 3
	6	4	2
	5	3	1
	11 total	7 total	3 total

therefore parallel time of 11

factor of almost 4 between earliest completion and

terrible load balancing

Interleaved:	Proc 1	Proc 2	Proc 3
	6	5	4
	3	2	1
	9 total	7 total	5 total

therefore parallel time of 9

Reflected:	Proc 1	Proc 2	Proc 3
	6	5	4
	1	2	3
	7 total	7 total	7 total

therefore parallel time of 7

## Dynamic scheduling

- next task to next available processor
- reason: best load balance independent of task time profile
- problems: overhead

Assume tasks 1 to 6 have the following times

6 5 4 3 2 1 then dynamic yields  
the same as reflected

Dynamic:	Proc 1	Proc 2	Proc 3
	6	5	4
	1	2	3
	7 total	7 total	7 total

therefore parallel time of 7

Assume tasks 1 to 6 have the following times

6 1 3 4 5 2 then dynamic yields

Dynamic:	Proc 1	Proc 2	Proc 3
	6	1	3
		4	5
		2	
	6 total	7 total	8 total

therefore parallel time of 8

- we have assumed that all tasks were independent and that all processors were already involved in their execution
- if there are dependences we must guarantee correct access patterns to shared memory (synchronization)
- once the system is running we must be able to involve all processors in sharing work and reverting to a single processor performing work
- often this is referred to as **fork/join** (not this is not equivalent to the Unix terminology)

- **fork** is executed by the single processor performing work sequentially in order to signal the other processors that they should begin aiding the execution of the work in the code (the particular work is identified through scheduling information)
- **join** – **Exactly one processor** can proceed beyond the synchronization point once **all** processors have reached the synchronization point. The processors that were active but that do not take over the sequential execution become idle until the next fork reactivates them.

- some times more than one processor proceeds after the synchronization point
- A **barrier** synchronization point in a code is such that **all** processors can proceed beyond the synchronization point once **all** processors have reached the synchronization point.
- a join is a special case of the barrier
- consider a case where all elements of an array are repeatedly updated in parallel, e.g., an image processing type application

Proc 1

$y(1) =$   
 $C + x(1)$

barrier

$x(1) = y(1)$

barrier

$y(1) =$   
 $C + x(1)$

barrier

$x(1) = y(1)$

Proc 2

$y(2) =$   
 $x(2) + x(3)$

barrier

$x(2) = y(2)$

barrier

$y(2) =$   
 $x(2) + x(3)$

barrier

$x(2) = y(2)$

Proc 3

$y(3) =$   
 $x(3) + x(4)$

barrier

$x(3) = y(3)$

barrier

$y(3) =$   
 $x(3) + x(4)$

barrier

$x(3) = y(3)$

Proc 4

$y(4) =$   
 $x(4) + D$

barrier

$x(4) = y(4)$

barrier

$y(4) =$   
 $x(4) + D$

barrier

$x(4) = y(4)$

- all processors execute barrier
- assumes all memory transactions are completed before any processor proceeds
- barrier is a **global synchronization construct**
- fork, join, barrier often supported with special HW
- local synchronization constructs also exist

A **critical section** of code is one that can be executed by **exactly one processor at any given time**.

This does not mean that only one processor in the system is active if a critical section is being executed.

Other processors may be executing different critical sections, i.e., a code may have more than one critical section.

Other processors may be executing noncritical sections or they may be idle and not waiting at a synchronization point.

Crucial performance considerations include: the length of time it takes to execute a critical section; the number of times a processor must enter a critical section; the number of processors that must complete the critical section; and the amount of parallel work to be done relative to the amount done in a critical section.

There are many ways to implement the synchronization around a critical section.

Often a lock/unlock pair has been used to control admission to the critical section of code.

- A call to lock(i) checks the state of the lock variable i. If it is “unlocked” the processor succeeds in seizing the lock, locks it, and proceeds with the critical section. (In some dialects the variable i must be specifically declared a lock variable.)
- This locking is guaranteed to be indivisible, i.e., only one processor can succeed at any given time.
- A call to unlock changes the state of the lock to “unlocked” and may notify based on some priority scheme the next processor waiting on the lock that it can now proceed with the locking procedure.

- suppose you want to sum all  $n$  elements of a vector
- $s = x(1) + \dots + x(n)$
- use static interleaved scheduling
- $slocal$  is a variable private to each processor
- $sum$  is an intrinsic function that sums the elements of the vector passed as an argument
- $s$  is a shared variable
- $lk$  is a shared lock variable assumed to be initially unlocked

$x(a : b : c)$  refers to the elements of a vector starting at  $x(a)$  using stride  $c$  to get to (or no further than)  $x(b)$

$X(1:11:3)$

$X(1), X(4), X(7), X(10)$

Proc 1

```
slocal=  
sum(x(1:n:3))
```

```
call lock(lk)  
s = s + slocal  
call unlock(lk)
```

Proc 2

```
slocal=  
sum(x(2:n:3))
```

```
call lock(lk)  
s = s + slocal  
call unlock(lk)
```

Proc 3

```
slocal=  
sum(x(3:n:3))
```

```
call lock(lk)  
s = s + slocal  
call unlock(lk)
```

- Lock/unlock can be implemented at several levels from special memory hardware to operating system software.
- The special memory hardware typically exploits an indivisible memory operation referred to as a **test-and-set**. A memory location is tested for a condition and if the condition is not found it is set – indivisibly, i.e., only one reference to the memory location can be done at a time. There are various ways to implement this primitive depending on the network and memory system.
- Since it requires a special operation in the memory system and is therefore more expensive than a normal read/write it is often the case that one tests the location first before attempting the TAS. If the condition is not set then a TAS is done. This reduces the number of failed TAS requests in the memory system.