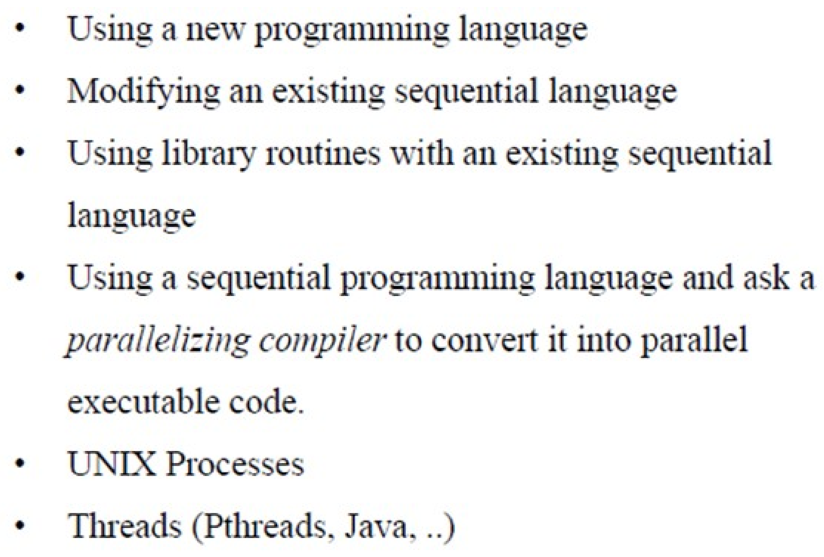
L3 S8 Multithreaded Programming

Several Althernatives for Programming



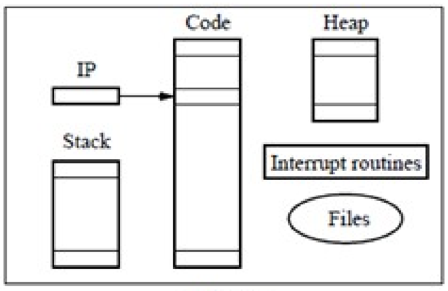
- Using a new programming language
- Modifying an existing sequential language
- Using library routines with an existing sequential language
- Using a sequential programming language and ask a *parallelizing compiler* to convert it into parallel executable code.
- UNIX Processes
- Threads (Pthreads, Java, ..)

UNIX Heavyweight Process

- The UNIX system call *fork()* creates a new process which is an *exact copy* of the calling process except that it has a unique process ID.
- *wait(startup)* delays caller until signal received or one of its child processes terminates or stops
- *exit(status)* terminates a process

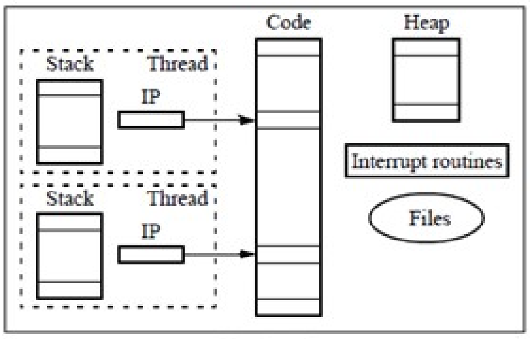
Thread

"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.



(a) Process

Threads - *shares* the same memory space and global variables between routines.



(b) Threads

Detached Threads

- It may be that thread may not be bothered when a thread it creates and in that case a join not be needed. Threads that are not joinned are called detached threads
- 主线程死了，子线程也会死，除了detached

Thread-Safe Routines

- System calls or library routines are called *thread safe* if they can be called from multiple threads simultaneously and always produce correct results.
- The thread-safety aspect of any routine can be avoided by forcing only one thread to execute the routine at a time. This could be achieved by simply enclosing the routine in a critical section but this is very inefficient.

Critical Section

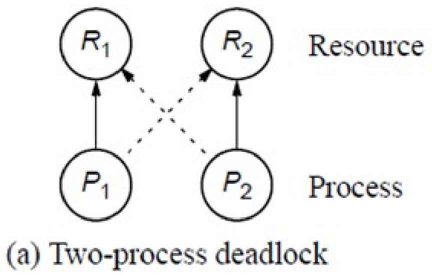
- A mechanism for ensuring that only one process accesses a particular resource at a time is to establish sections of code involving the resource as so-called critical sections.
- *mutal exclusion*

Locks

- Simplest, 1-bit.
- 1: a process has entered the critical section. 0: no process is in the critical section.

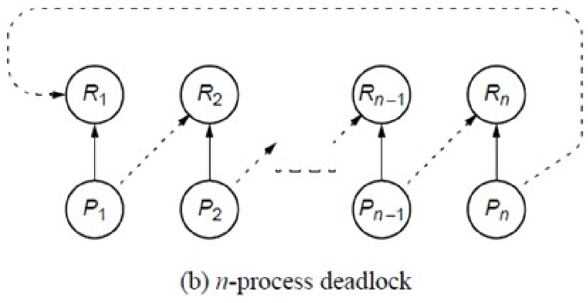
Deadlock

- Two process



'igure 8.8 Deadlock (deadly embrace).

- n-process



Semaphore 信号

- *s* is a positive integer (including zero)
- Process delayed by P(s) are kept in abeyance until released by a V(s) on the same semaphore

P operations, P(s)

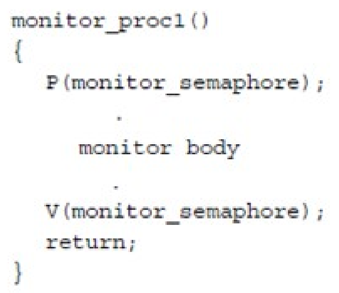
- wait until s is greater than zero and then decrements s by one and allows process to continue

V operation, V(s)

- increments s by one to release one of the waiting processes (if any)

General Semaphore

- Semaphore routines exists for UNIX processes, they don't exist in Pthreads as such, though they can be written and they do exist in the real-time extension to Pthreads
- Reading and writing can only be done by using a monitor procedure, and only one process can use a monitor procedure at any instant.



```
monitor_proc1()
{
   P(monitor_semaphore);
        .
      monitor body
        .
   V(monitor_semaphore);
   return;
}
```

Condition Variables

Operations

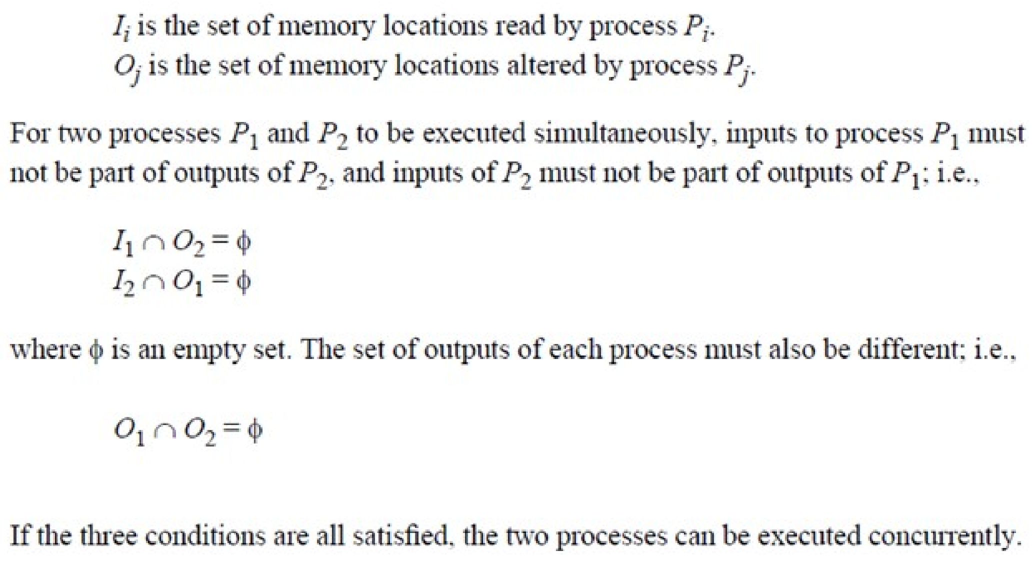
- $wait(cond\_var)$ - wait for a condition to occur
- $singla(cond\_var)$ - signal that the condition has occurred
- $status(cond\_var)$ - return the number of processes waiting for the condition to occur

Language Constructs for Parallelism

- Shared memory variables: `shared int  x`
- for specifying concurrent statements: `par{ code }`
- 类似于openMP的for loop `forall (i=0;i<n;i++){ code }`

Berstein's Conditions

- Set of conditions that are sufficient to determine whether two proceses can be executed simultaneously.



I_i is the set of memory locations read by process P_i.
O_j is the set of memory locations altered by process P_j.

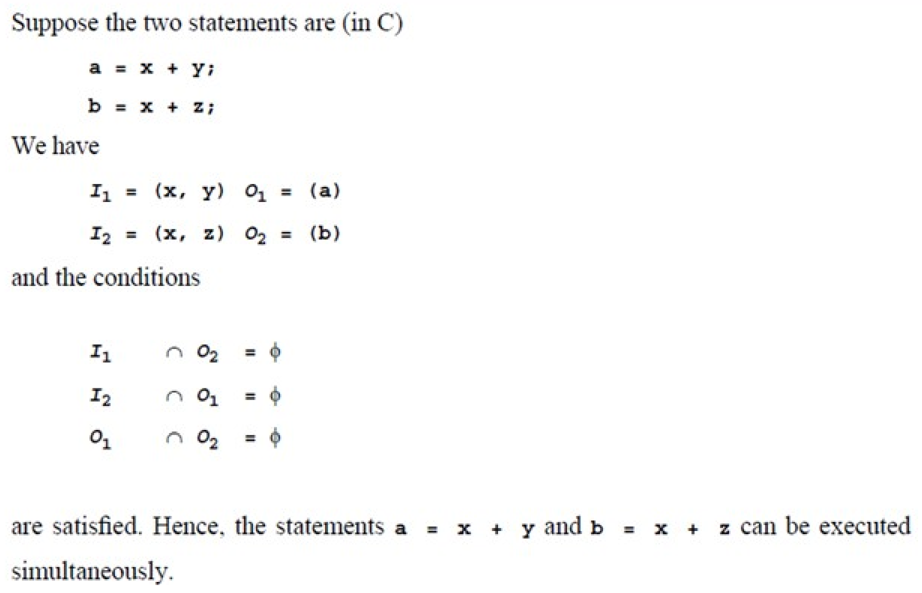
For two processes P_1 and P_2 to be executed simultaneously, inputs to process P_1 must not be part of outputs of P_2, and inputs of P_2 must not be part of outputs of P_1; i.e.,

$$I_1 \cap O_2 = \phi$$
$$I_2 \cap O_1 = \phi$$

where ϕ is an empty set. The set of outputs of each process must also be different; i.e.,

$$O_1 \cap O_2 = \phi$$

If the three conditions are all satisfied, the two processes can be executed concurrently.



Suppose the two statements are (in C)

```
a = x + y;
b = x + z;
```

We have

$I_1 = (x, y) \quad O_1 = (a)$

$I_2 = (x, z) \quad O_2 = (b)$

and the conditions

$$I_1 \cap O_2 = \phi$$
$$I_2 \cap O_1 = \phi$$
$$O_1 \cap O_2 = \phi$$

are satisfied. Hence, the statements `a = x + y` and `b = x + z` can be executed simultaneously.

Shared Data in System with Caches

Cache coherence protocols

- *update policy*: copies of data in all caches are updated at the time one copy is altered
- *invalidate policy*: when one copy of data is altered, the same data in any other cache is invalidated. These copies are only updated when the associated processor makes reference for it.

(2)

False Sharing

Different parts of block required by different processors but not same bytes. If one processor writes to one part of the block, copies of the complete block in other caches must be updated or invalidated though the actual data is not shared.

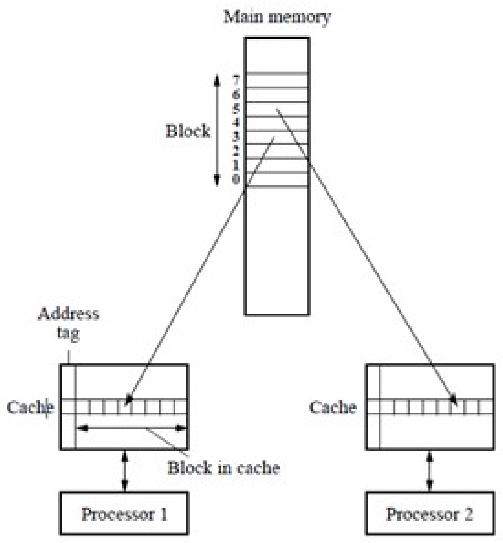


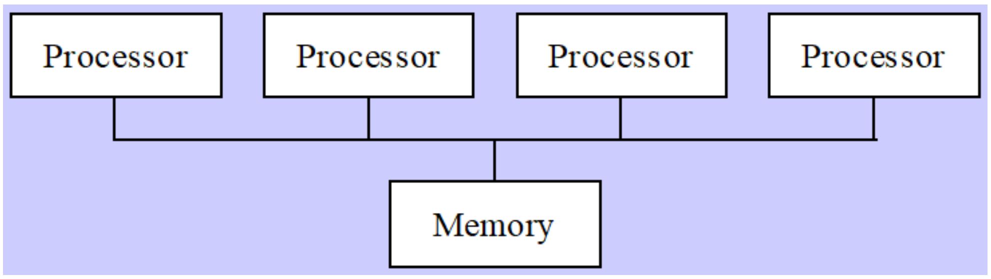
Figure 8.9 False sharing in caches.

Compiler to alter the layout of the data stored in the main memory, separating data only altered by one processor into different blocks.

L4 Quinn Shared-memory Programming

OpenMP

Shared-memory model



Fork/Join Parallelism

- Initially only master thread is active.

Shared-memory Model vs. Message-passing Model

- shared-memory model
  - number active threads 1 at start and finish of program, changes dynamically during execution
  - incrementally make it parallel
- message-passing model
  - all processes active throughout execution of program

    - sequential-to-parallel transformation requires major effort

Incremental Parallelization

- process of converting a sequential program to a parallel program a little bit at a time
- sequential is a special case of shared-m

Parallel for loops

- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads

Pragma

- A compiler directive in C or C++. Stands for "pragmatic information"
- `#pragma omp <rest of pragma>`

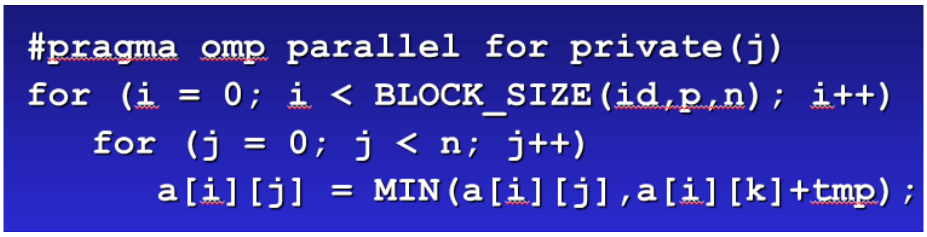
Execution Context

- Every thread has its own execution context (address space containing all of the variables a thread may access)
- Includes:
    - static variables
    - dynamically allocated data structures in the heap
    - variables on the run-time stack
    - additional run-time stack for functions invoked by the thread

Declaring private varibales

- Private clause: directs compiler to make one or more variables privte
    - `private ()

- e.g.

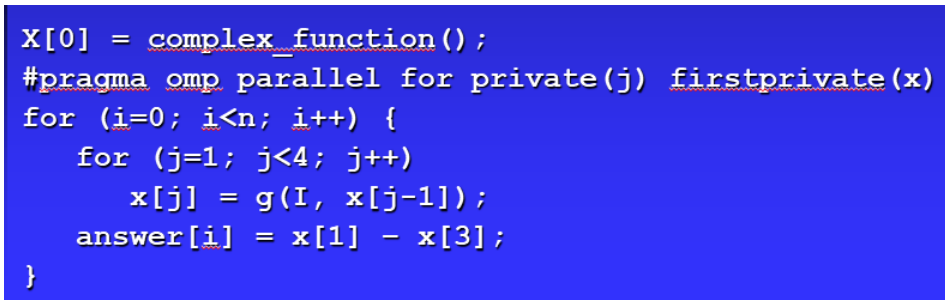


```
#pragma omp parallel for private(j)
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
    for (j = 0; j < n; j++)
        a[i][j] = MIN(a[i][j],a[i][k]+tmp);
```

firstprivate clause

- Variables are initialized once per thread, not once per loop iteration

- 用于继承同名变量的值-并行区域之外的变量的值，进行一次初始化
- If a thread modifies a variables's value in an iteration, subsequent iterations will get the modified value
- e.g.



```
X[0] = complex_function();
#pragma omp parallel for private(j) firstprivate(x)
for (i=0; i<n; i++) {
   for (j=1; j<4; j++)
      x[j] = g(I, x[j-1]);
   answer[i] = x[1] - x[3];
}
```

lastprivate clause

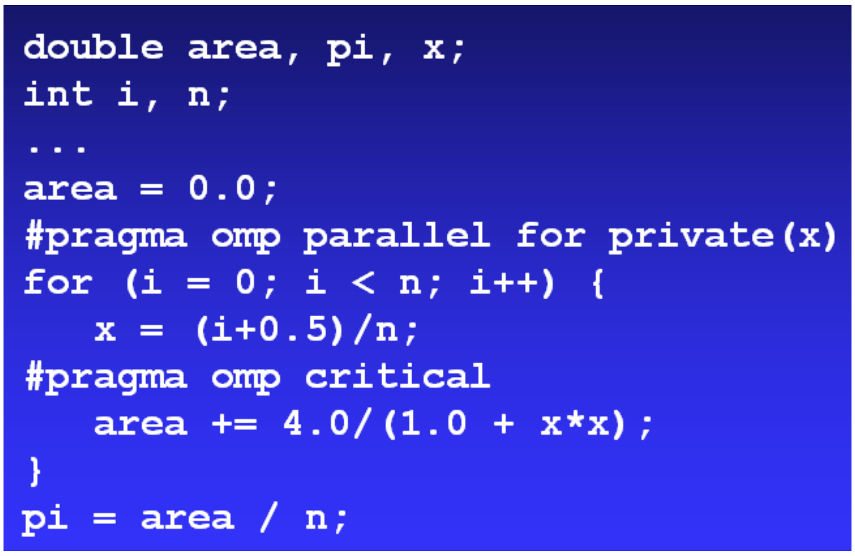
- sequentially last iteration: iteration that occurs last when the loop is executed sequentially
- used to copy back to the master thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration.
- 将值赋予同名的共享变量，sequential的最后一次那个

Critical sections

Race Condition

Critical Pragma

- Critical section: a portion of code that only thread at a time may execute. like mutex
- `#pragma omp critical`
- e.g. (correct, but inefficient code)



```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
#pragma omp critical
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Reductions

- `reduction (<op>:<variable>)`
  - op: +, *, bitwise and, bitwise or, bitwise exclusive or, logical and, logical or

    （1）进入并行区域后，team内的每个新的线程都会对reduction变量构造一个副本，比如上面的例子，假设有四个线程，那么，进入并行区域的初始化值分别为：sum0=100,sum1 = sum2 = sum3 = 0.为何sum0为100呢？因为主线程不是一个新的线程，所以不需要再为主线程构造一个副本（没有找到官方这样的说法，但是从理解上，应该就是这样工作的，只会有一个线程使用到并行区域外的初始值，其余的都是0）。

    （2）每个线程使用自己的副本变量完成计算。

    （3）在退出并行区域时，对所有的线程的副本变量使用指定的操作符进行迭代操作，对于上面的例子，即sum' = sum0'+sum1'+sum2'+sum3'.

    （4）将迭代的结果赋值给原来的变量（sum），sum=sum'.

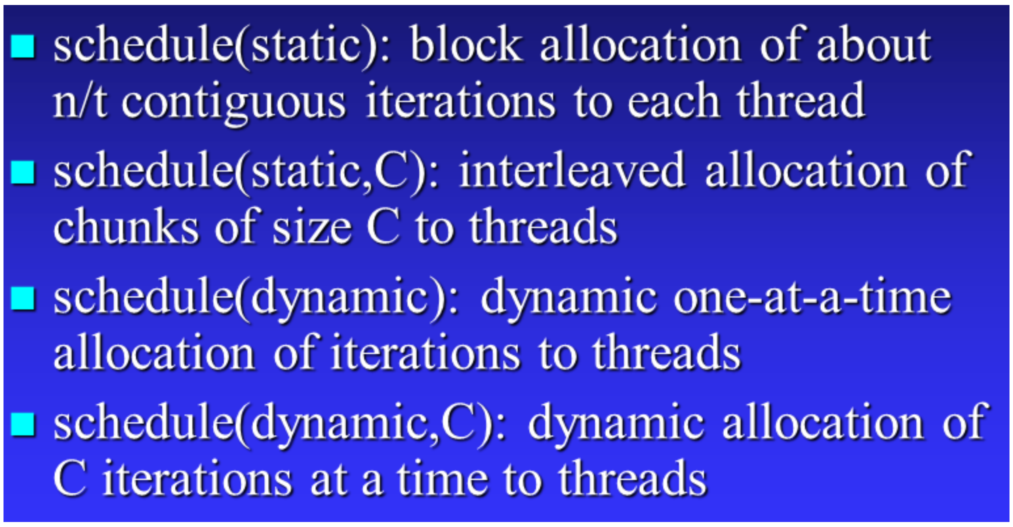
Performance improvements

- Too many fork/joins can lower performance
- If look has too few terations, fork/join overhead is greater than time savings from parallel execution.
  - `#pragma omp parallel for if(n>500)`
- `schedule`
  - static schedule: all iterations allocated to threads before any iterations executed
    - low overhead, may exhibit workload imbalance
  - dynamic schedule: only some iterations allocated to threads at beginning of loop's execution. Remaing iterations allocated to threads that complete their assigned iterations.
    - higher overhead; can reduce workload imbalance

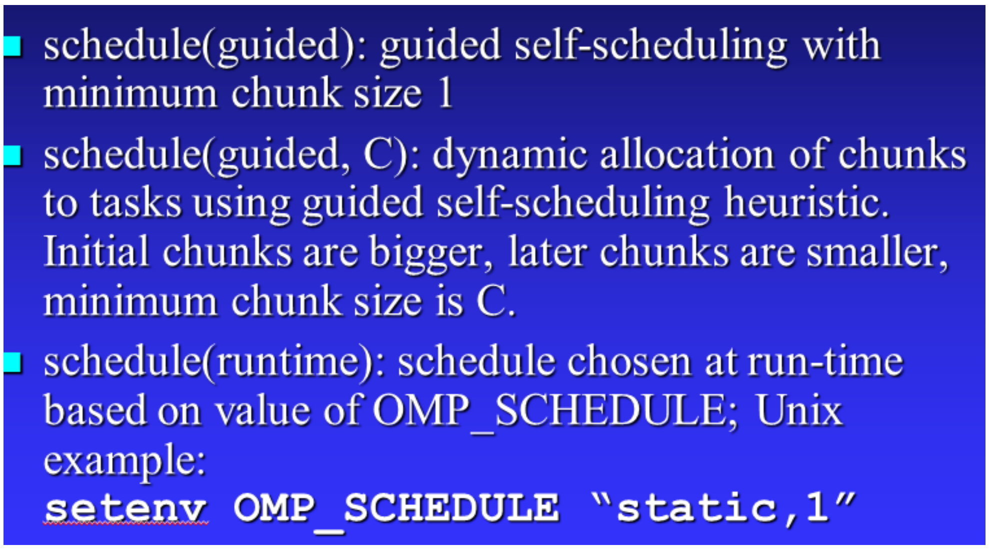
Chunks

- A chunk is a contiguous range of iterations
- Increasing chunk size reduces overhead and may increase cache hit rate.
- Decreasing chunk size allows finer balancing of workloads
- `schedule (<type>[,<chunl.])`

Scheduling options



- schedule(static): block allocation of about n/t contiguous iterations to each thread
- schedule(static,C): interleaved allocation of chunks of size C to threads
- schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
- schedule(dynamic,C): dynamic allocation of C iterations at a time to threads



- schedule(guided): guided self-scheduling with minimum chunk size 1
- schedule(guided, C): dynamic allocation of chunks to tasks using guided self-scheduling heuristic. Initial chunks are bigger, later chunks are smaller, minimum chunk size is C.
- schedule(runtime): schedule chosen at run-time based on value of OMP_SCHEDULE; Unix example:
  `setenv OMP_SCHEDULE "static,1"`

More general data parallelism

Functions for SPMD-style Programming

- `int omp_get_thread_num(void)` return thread identification number
- `int omp_get_num_threads(void)` return the number of active threads

nowait Clause

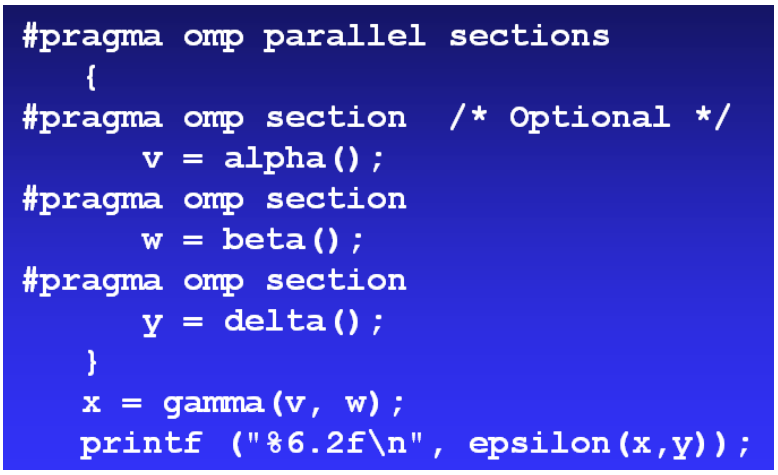
- Compiler puts a barrier synchronization at end of every parallel for statement.

Functional parallelism

parallel sections pragma

- `pragma omp parallel sections`

- e.g.



```
#pragma omp parallel sections
   {
#pragma omp section  /* Optional */
      v = alpha();
#pragma omp section
      w = beta();
#pragma omp section
      y = delta();
   }
   x = gamma(v, w);
   printf ("%6.2f\n", epsilon(x,y));
```

sections和for的区别：

当我们使用for时，openmp是对for语句的所有i值分配到不同线程（或计算节点）进行并行计算。

而当我们使用sections，openmp是对sections中的子section分配到不同计算节点进行并行计算。这只适用于section和section之间没有依赖的情形。