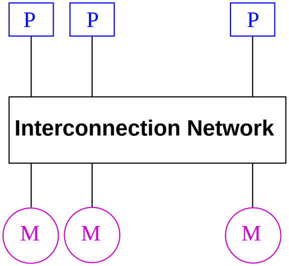
Shared Memory and OpenMP

- Background
  - Shared Memory Hardware
  - Shared Memory Languages
- OpenMP

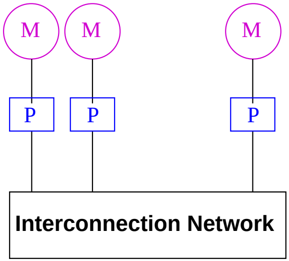
Parallel Hardware

Shared Memory Machines global memory can be acessed by all processors or cores. Information exchanged between threads using shared variables written by one thread and read by another. Need to coordinate access to shared variables.



Parallel Hardware

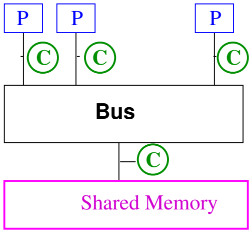
Distributed Memory Machines private memory for each processor, only accessible this processor, so no synchronization for memory accesses needed. Information exchanged by sending data from one processor to another via an interconnection network using explicit communication operations.



Hybrid approach increasingly common

Shared Memory Systems

Symmetric Multiprocessors (SMP): processors all connected to a large shared memory. Examples are processors connected by crossbar, or multicore chips. Key characteristic is *uniform memory access (UMA)*

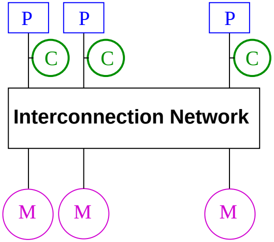


Caches are a problem - need to be kept *coherent* = when one CPU changes a value in memory, then all other CPUs will get the same value when they access it.

Shared Memory Systems

Distributed Shared Memory Memory is logically shared but physically distributed. Has *non-uniform memory access (NUMA)*

- Any processor can access any address in memory
- Cache lines (or pages) passed around machine. Difficulty is *cache coherency protocols*.
- CC-NUMA architecture (if network is cache-coherent)



(SGI Altix at NASA Ames - had 10,240 cpus of Itanium 2 nodes connected by Infiniband, was ranked 84 in June 2010 list, ranked 3 in 2008. Expensive!)

Parallel Programming Models

Programming model gives an abstract view of the machine describing

- Control
  - how is parallelism created?
  - what ordering is there between operations?
- Data
  - What data is private or shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations are used to coordinate parallelism
  - What operations are atomic (indivisible)?

Shared Memory Programming Model

Program consists of *threads* of control with

- shared variables
- private variables
- threads communicate implicitly by writing and reading shared variables
- threads coordinate by synchronizing on shared variables

Threads can be dynamically created and destroyed.

Other programming models: distributed memory, hybrid, data parallel programming model (single thread of control), shared address space,

What’s a thread? A process?

Processes are independent execution units that contain their own state information and their own address space. They interact via interprocess communication mechanisms (generally managed by the operating system). One process may contain many threads. Processes are given system resources.

All threads within a process share the same address space, and can communicate directly using shared variables. Each thread has its own stack but only one data section, so global variables and heap-allocated data are shared (this can be dangerous).

What is state?

- instruction pointer
- Register file (one per thread)
- Stack pointer (one per thread)

Multithreaded Processors

- Both the above (SMP and Distributed Shared Memory Machines) are *shared address space* platforms.
- Also can have multithreading on a single processor. Switch between threads for long-latency memory operations
    - multiple thread *contexts* without full processors
    - Memory and some other state is shared
    - Can combine multithreading and multicore, e.g. Intel Hyperthreading, more generally SMT (simultaneous multithreading).
    - Cray MTA (MultiThreaded Architecture, hardware support for context switching every cycle), and Eldorado processors. Sun Niagra processors (multiple FPU and ALU per chip, 8 cores handle up to 8 threads per core)

Shared Memory Languages

- pthreads - POSIX (Portable Operating System Interface for Unix) threads; heavyweight, more clumsy
- PGAS languages - Partitioned Global Address Space UPC, Titanium, Co-Array Fortran; not yet popular enough, or efficient enough
- OpenMP - newer standard for shared memory parallel programming, lighter weight threads, not a programming language but an API for C and Fortran

OpenMP Overview

OpenMP is an API for multithreaded, shared memory parallelism.

- A set of compiler directives inserted in the source program
  - pragmas in C/C++ (pragma = compiler directive external to prog. lang. for giving additional info., usually non-portable, treated like comments if not understood)
  - (specially written) comments in fortran
- Library functions
- Environment variables

Goal is standardization, ease of use, portability. Allows incremental approach. Significant parallelism possible with just 3 or 4 directives. Works on SMPs and DSMs.

Allows fine and coarse-grained parallelism; loop level as well as explicit work assignment to threads as in SPMD.

What is OpenMP?

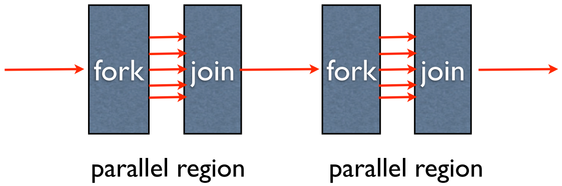
- http://www.openmp.org
- Maintained by the OpenMP Architecture Review Board (ARB) (non-profit group of organizations that interpret and update OpenMP, write new specs, etc. Includes Compaq/Digital, HP, Intel, IBM, KAI, SGI, Sun, DOE. (Endorsed by software and application vendors).
- Individuals also participate through cOMPunity, which participates in ARB, organizes workshops, etc.
- Started in 1997. OpenMP 3.0 just recently released.
- For Fortran (77,90,95), C and C++, on Unix, Windows NT and other platforms.

OpenMP = Open specifications for MultiProcessing

Basic Idea

Explicit programmer control of parallelization using fork-join model of parallel execution

- all OpenMP programs begin as single process, the master thread, which executes until a parallel region construct encountered
- FORK: master thread creates team of parallel threads
- JOIN: When threads complete statements in parallel region construct they synchronize and terminate, leaving only the master thread. (similar to fork-join of Pthreads)



Basic Idea

- User inserts directives telling compiler how to execute statements
    - which parts are parallel
    - how to assign code in parallel regions to threads
    - what data is private (local) to threads
    - `#pragma omp` in C and `!$omp` in Fortran
- Compiler generates explicit threaded code
- Rule of thumb: One thread per core (2 or 4 with hyperthreading)
- Dependencies in parallel parts require synchronization between threads

Simple Example

```c
#include <omp.h>
#include <stdio.h>

int main() {

#pragma omp parallel
printf("Hello world from thread %d\n",omp_get_thread_num());

return 0;
}
```

Compile line:

```
gcc -fopenmp helloWorld.c
icc -openmp helloWorld.c
```

Simple Example

Sample Output:

```
MacBook-Pro% a.out
Hello world from thread 1
Hello world from thread 0
Hello world from thread 2
Hello world from thread 3

MacBook-Pro% a.out
Hello world from thread 0
Hello world from thread 3
Hello world from thread 2
Hello world from thread 1
```

(My laptop has 2 cores)
(Demos)

Setting the Number of Threads

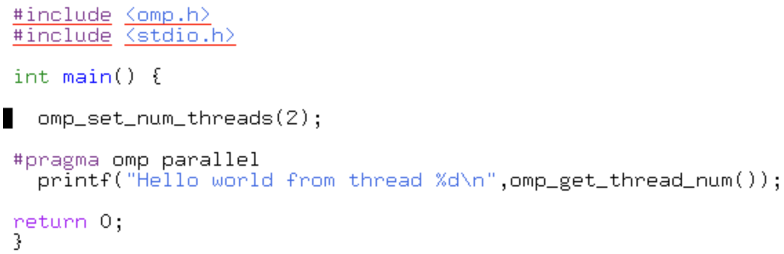
Environment Variables:

`setenv OMP_NUM_THREADS 2` (cshell)

`export OMP_NUM_THREADS=2` (bash shell)

Library call:

`omp_set_num_threads(2)`



```c
#include <omp.h>
#include <stdio.h>

int main() {

  omp_set_num_threads(2);

#pragma omp parallel
  printf("Hello world from thread %d\n",omp_get_thread_num());

return 0;
}
```

Parallel Construct

```
#include <omp.h>

int main(){
  int var1, var2, var3;

  ...serial Code

  #pragma omp parallel private(var1, var2) shared (var3)
 {
    ...parallel section
 }

 ...resume serial code

}
```

Parallel Directives

- When a thread reaches a PARALLEL directive, it becomes the master and has thread number 0.
- All threads execute the same code in the parallel region (Possibly redundant, or use work-sharing constructs to distribute the work)
- There is an implied barrier* at the end of a parallel section. Only the master thread continues past this point.
- If a thread terminates within a parallel region, all threads will terminates, and the result is undefined.
- Cannot branch into or out of a parallel region.

barrier - all threads wait for each other; no thread proceeds until all threads have reached that point

Parallel Directives

- If program compiled serially, openMP pragmas and comments ignored, stub library for omp library routines
- easy path to parallelization
- One source for both sequential and parallel helps maintenance.

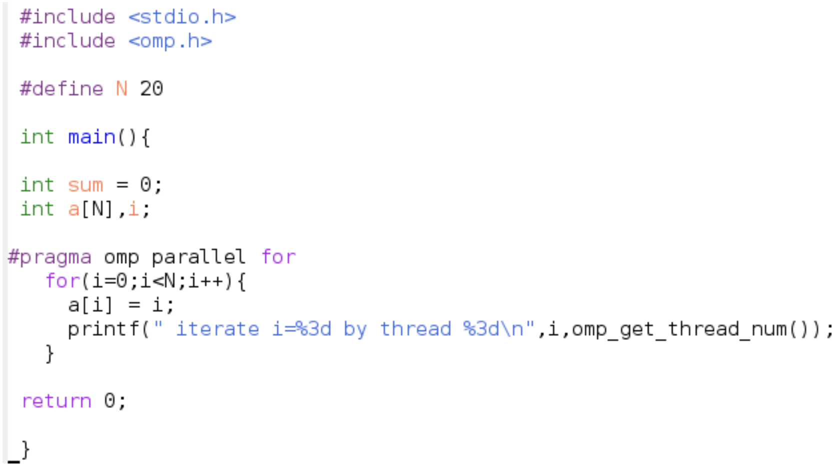
Work-Sharing Constructs

- work-sharing construct divides work among member threads. Must be dynamically within a parallel region.
- No new threads launched. Construct must be encountered by all threads in the team.
- No implied barrier on entry to a work-sharing construct; Yes at end of construct.

3 types of work-sharing construct (4 in Fortran - array constructs):

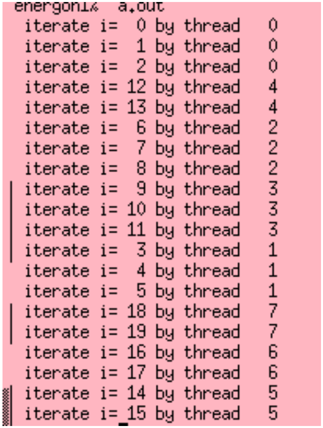
- for loop: share iterates of `for` loop (“data parallelism”) iterates must be independent
- sections: work broken into discrete section, each executed by a thread (“functional parallelism”)
- single: section of code executed by one thread only

FOR directive schedule example

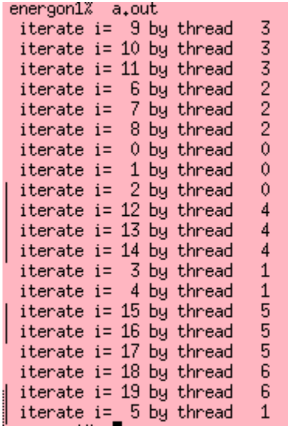


```
 #include <stdio.h>
 #include <omp.h>

 #define N 20

 int main(){

 int sum = 0;
 int a[N],i;

#pragma omp parallel for
   for(i=0;i<N;i++){
     a[i] = i;
     printf(" iterate i=%3d by thread %3d\n",i,omp_get_thread_num());
   }

 return 0;

}
```

FOR directive schedule example



```
energon1%  a.out
 iterate i=  0 by thread   0
 iterate i=  1 by thread   0
 iterate i=  2 by thread   0
 iterate i= 12 by thread   4
 iterate i= 13 by thread   4
 iterate i=  6 by thread   2
 iterate i=  7 by thread   2
 iterate i=  8 by thread   2
 iterate i=  9 by thread   3
 iterate i= 10 by thread   3
 iterate i= 11 by thread   3
 iterate i=  3 by thread   1
 iterate i=  4 by thread   1
 iterate i=  5 by thread   1
 iterate i= 18 by thread   7
 iterate i= 19 by thread   7
 iterate i= 16 by thread   6
 iterate i= 17 by thread   6
 iterate i= 14 by thread   5
 iterate i= 15 by thread   5
```



```
energon1%  a.out
 iterate i=  9 by thread   3
 iterate i= 10 by thread   3
 iterate i= 11 by thread   3
 iterate i=  6 by thread   2
 iterate i=  7 by thread   2
 iterate i=  8 by thread   2
 iterate i=  0 by thread   0
 iterate i=  1 by thread   0
 iterate i=  2 by thread   0
 iterate i= 12 by thread   4
 iterate i= 13 by thread   4
 iterate i= 14 by thread   4
 iterate i=  3 by thread   1
 iterate i=  4 by thread   1
 iterate i= 15 by thread   5
 iterate i= 16 by thread   5
 iterate i= 17 by thread   5
 iterate i= 18 by thread   6
 iterate i= 19 by thread   6
 iterate i=  5 by thread   1
```

for loop with 20 iterations and 8 threads:

icc: 4 threads get 3 iterations and 4 threads get 2
gcc: 6 threads get 3 iterations, 1 thread gets 2, 1 gets none

OMP Directives

All directives:

```
#pragma omp directive  [clause ...]
                        if (scalar_expression)
                        private (list)
                        shared (list)
                        default (shared | none)
                        firstprivate (list)
                        reduction (operator: list)
                        copyin (list)
                        num_threads (integer-expression)
```

Directives are:

- Case sensitive (not for Fortran)
- Only one directive-name per statement
- Directives apply to at most one succeeding statement, which must be a structured block.
- Continue on succeeding lines with backslash (`"\"`)

FOR directive

```
#pragma omp for [clause ...]
                       schedule (type [,chunk])
                       private (list)
                       firstprivate(list)
                       lastprivate(list)
                       shared (list)
                       reduction (operator: list)
                       nowait
```

SCHEDULE: describes how to divide the loop iterates

- **static** = divided into pieces of size *chunk*, and statically assigned to threads. Default is approx. equal sized chunks (at most 1 per thread)
- **dynamic** = divided into pieces of size *chunk* and dynamically scheduled as requested. Default chunk size 1.
- **guided** = size of chunk decreases over time. (Init. size proportional to the number of unassigned iterations divided by number of threads, decreasing to *chunk size*)
- **runtime**=schedule decision deferred to runtime, set by environment variable OMP_SCHEDULE.

FOR example

```
#pragma omp parallel shared(n,a,b,x,y), private(i)
{ // start parallel region

   #pragma omp for nowait
   for (i=0;i<n;i++)
     b[i] = += a[i];

   #pragma omp for nowait
   for (i=0;i<n;i++)
     x[i] = 1./y[i];

} // end parallel region  (implied barrier)
```

Spawning tasks is expensive: reuse if possible.
nowait clause: minimize synchronization.

SECTIONS directive

```
#pragma omp sections [clause ...]
                   private (list)
                   firstprivate(list)
                   lastprivate(list)
                   reduction (operator: list)
                   nowait
{
 #pragma omp section
   structured block
 #pragma omp section
   structured block
}
```

- implied barrier at the end of a SECTIONS directive, unless a NOWAIT clause used
- for different numbers of threads and SECTIONS some threads get none or more than one
- cannot count on which thread executes which section
- no branching in or out of sections

Sections example

```
#pragma omp parallel shared(n,a,b,x,y), private(i)
{ // start parallel region
   #pragma omp sections nowait
   {
      #pragma omp section
      for (i=0;i<n;i++)
        b[i] = += a[i];

      #pragma omp section
      for (i=0;i<n;i++)
        x[i] = 1./y[i];

   } // end sections
} // end parallel region
```

SINGLE directive

```
#pragma omp single [clause ...]
                   private (list)
                   firstprivate(list)
                   nowait
 structured block
```

- SINGLE directive says only one thread in the team executes the enclosed code
- useful for code that isn't thread-safe (e.g. I/O)
- rest of threads wait at the end of enclosed code block (unless NOWAIT clause specified)
- no branching in or out of SINGLE block

firstprivate example

What is wrong with this code snippet?

```
#pragma omp parallel for
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

firstprivate example

What is wrong with this code snippet?

```
#pragma omp parallel for
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

By default, `x` is shared variable (i is private).

Could have: Thread 0 set `x` for some `i`.
Thread 1 sets `x` for different `i`.
Thread 0 uses `x` but it is now incorrect.

firstprivate example

Instead use:

```
#pragma omp parallel for private(x)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

What about `i`,`dx`,`y`?

firstprivate example

Instead use:

```
#pragma omp parallel for private(x)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

What about `i,dx,y`?

By default `dx,n,y` shared.
`dx,n` used but not changed. `y` changed, but independently for each `i`

firstprivate example

What is wrong with this code?

```
dx = 1/n.;
#pragma omp parallel for private(x,dx)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

firstprivate example

What is wrong with this code?

```
dx = 1/n.;
#pragma omp parallel for private(x,dx)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

Specifying `dx` private creates a new private variable for each thread, but it is not initialized.

firstprivate clause creates private variables and initializes to the value from the master thread before the loop.

lastprivate copies last value computed by a thread (for *i=n*) to the maser thread copy to continue execution.

Clauses

These clauses not strictly necessary but may be convenient (and may have performance penalties too).

- lastprivate private data is undefined after parallel construct. this gives it the value of last iteration (as if sequential) or sections construct (in lexical order).
- firstprivate pre-initialize private vars with value of variable with same name before parallel construct.
- default (`none | shared`). In fortran can also have private. Then only need to list exceptions. (none is better habit).
- nowait suppress implicit barrier at end of work sharing construct. Cannot ignore at end of parallel region. (But no guarantee that if have 2 `for` loops where second depends on data from first that same threads execute same iterates)

More Clauses

- if (logical expr) true = execute parallel region with team of threads; false = run serially (loop too small, too much overhead)
- reduction for assoc. and commutative operators compiler helps out; reduction variable is shared by default (no need to specify).

  ```
  #pragma omp parallel for default(none) \
                           shared(n,a) \
                           reduction(+:sum)
    for (i=0;i<n;i++)
      sum += a[i]
    /* end of parallel reduction  */
  ```

  Also other arithmetic and logical ops., `min`,`max` instrinsics in Fortan only.
- copyprivate only with single direction. one thread reads and initializes private vars. which are copied to other threads before they leave barrier.
- threadprivate variables persist between different parallel sections (unlike private vars). (applies to global vars. must have dynamic false)

Race Condition* Example

```
#include <stdio.h>
#include <omp.h>

int main(){

  int x = 2;

  #pragma omp parallel shared(x) num_threads(2)
  {
    if (1 == omp_get_thread_num()){
      x = 5;
    }
    else {
      printf("1: Thread %d has x = %d\n",omp_get_thread_num(),x);
    }

    #pragma omp barrier

    if (0 == omp_get_thread_num()) {
      printf("2: thread %d has x = %d\n",omp_get_thread_num(),x);
    }
    else {
      printf("3: thread %d has x = %x\n",omp_get_thread_num(),x);
    }

  }
  return 0;
}
```

*race condition= 2 or more threads access shared variable without synchronization and at least one is a write.

Synchronization

- Implicit barrier synchronization at end of parallel region (no explicit support for synch. subset of threads). Can invoke explicitly with `#pragma omp barrier`. All threads must see same sequence of work-sharing and barrier regions .
- critical sections: only one thread at a time in critical region with the same name. `#pragma omp critical [(name)]`
- atomic operation: protects updates to individual memory loc. Only simple expressions allowed. `#pragma omp atomic`
- locks: low-level run-time library routines (like mutex vars., semaphores)
- flush operation - forces the executing thread to make its values of shared data consistent with shared memory
- master (like single but not implied barrier at end), *ordered*, ...

At all these (implicit or explicit) synchronization points OpenMP ensures that threads have consistent values of shared data.

Critical Example

```
#pragma omp parallel sections
{
  #pragma omp section
  {
      task = produce_task();
      #pragma omp critical (task_queue)
      {
         insert_into_queue(task);
      }
  }
  #pragma omp section
  {
     #pragma omp critical (task_queue)
     {
       task = delete_from_queue(task);
     }
     consume_task(task);
  }
}
```

Atomic Examples

```
#pragma omp parallel shared(n,ic) private(i)
   for (i=0;i<n;i++){
     #pragma omp atomic
          ic = ic +1;
   }
```

`ic` incremented atomically

```
#pragma omp parallel shared(n,ic) private(i)
   for (i=0;i<n;i++){
     #pragma omp atomic
        ic = ic + bigfunc();
   }
```

`bigfunc` not atomic, only `ic` update

allowable atomic operations:

```
 x binop=  expr      x++ or ++x        x-- or --x
```

Atomic Example

```
int sum = 0;
#pragma omp parallel for shared(n,a,sum)
{
  for (i=0; i<n; i++){
  #pragma omp atomic
     sum = sum + a[i];
  }
}
```

Better to use a *reduction* clause:

```
int sum = 0;
#pragma omp parallel for shared(n,a)   \
                     reduction(+:sum)
{
  for (i=0; i<n; i++){
     sum += a[i];
  }
}
```

Locks

Locks control access to shared resources. Up to implementation to use spin locks (busy waiting) or not.

- Lock variables must be accessed only through locking routines:

  ```
  omp_init_lock  omp_destroy_lock
  omp_set_lock   omp_unset_lock  omp_test_lock
  ```

- In C, lock is a type `omp_lock_t` or `omp_nest_lock_t` (In Fortran lock variable is integer)
- initial state of lock is unlocked.
- `omp_set_lock(omp_lock_t *lock)` forces calling thread to wait until the specified lock is available. (Non-blocking version is `omp_test_lock`

Examing and setting a lock must be *uninterruptible* operation.

Lock Example

```
#include <stdio.h>
#include <omp.h>

omp_lock_t my_lock;

int main() {
   omp_init_lock(&my_lock);

   #pragma omp parallel
   {
      int tid = omp_get_thread_num( );
      int i, j;

      for (i = 0; i < 5; ++i) {
         omp_set_lock(&my_lock);
         printf("Thread %d - starting locked region\n", tid);

         printf("Thread %d - ending locked region\n", tid);
         omp_unset_lock(&my_lock);
      }
   }

   omp_destroy_lock(&my_lock);
}
```

Deadlock

Runtime situation that occurs when a thread is waiting for a resource that will never be available. Common situation is when two (or more) actions are each waiting for the other to finish (for example, 2 threads acquire 2 locks in different order)

```
work1() {  /* do some work */
  #pragma omp barrier
}
work2(){ /* do some work */
}
main(){
   #pragma omp parallel sections
   {
     #pragma omp section
         work1();

     #pragma omp section
         work2();
   }
} /* end main */
```

Also livelock: state changes but no progress is made.

Nested Loops

Which is better (assuming $m \approx n$)?

```
#pragma omp parallel for private(i)
for (j=0;j<m;j++)
   for (i=0;i<n;i++)
       a[j][i] = 0.;
```

or

```
for (j=0;j<m;j++)
#  pragma omp parallel for
   for (i=0;i<n;i++)
       a[jk][i] = 0.;
```

Nested Loops

Which is better (assuming $m \approx n$)?

```
#pragma omp parallel for private(i)
for (j=0;j<m;j++)
   for (i=0;i<n;i++)
       a[j][i] = 0.;
```

or

```
for (j=0;j<m;j++)
#  pragma omp parallel for
   for (i=0;i<n;i++)
       a[jk][i] = 0.;
```

First has less overhead: threads created once instead of m times.
What about order of indices?

Memory Model

- By default data is shared among all thread.
- Private data - each thread has its own copy (e.g.loop variables); reduces hot spots, avoid race conditions and synchronizations, increases memory (need larger stack).
- OpenMP rule: shared objects must be made available to all threads at synchronization points (relaxed consistency model)
- Between synch., may keep values in local cache, so threads may see different values for shared objects. If one thread needs another's value, must insert synch. point.
- *flush* operation synchronizes values for shared objects. (a.k.a. memory fence). Writes to shared vars. must be committed to memory; all refs. to shared vars. after a fence must be read from memory. `#pragma omp flush [varlist]`. Default is all shared vars.

Memory Model

OpenMP provides relaxed- consistency view of thread memory. Threads can cache their data and are not required to maintain exact consistency with real memory all the time.

A thread is allowed to have its own *temporary view* of memory (incuding cache, registers) to avoid having to go to memory for every reference to a variable.

Memory Consistency Problem

A quick aside to present the issue. More discussion later.

In uniprocessor, when new value computed it is written back to cache, where either

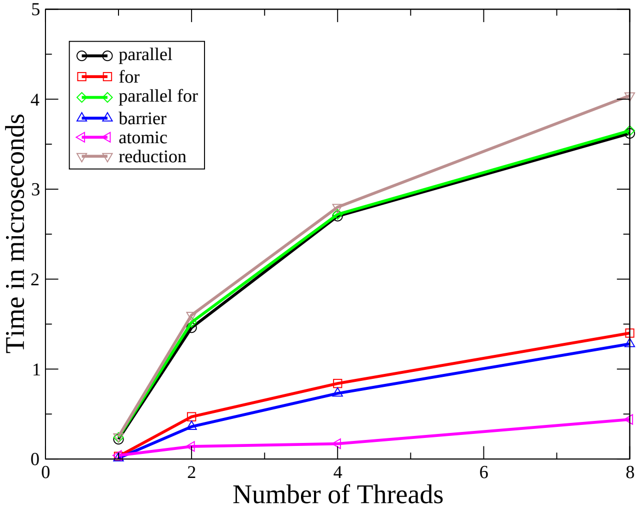
- whenever a write is performed the cache is updated and a write to main memory is issued (write-through). This maintains cache coherence - cache accurately reflects the contents of main memory. (But main memory is slow; may writes slows down reads; try to buffer the writes - postpone until done working on that memory location)
- new value stays in cache until cache line needed for other data. Before evicted it is written back to main memory (write-back). More complicated- need flag to indicate *cleanliness* for each cache entry; *dirty* indicates value at that location has changed; and before eviction needs to be written back to memory. If there are dirty cache entries; cache is not coherent with that memory location..

Memory Consistency Problem

- In SMP, suppose one processor has an updated result in private cache. Second processor wants to access that memory location - but a read from memory will get the old value since original data not yet written back. When is new value available?
- What if old value in second processor's cache? How do other processors know to *invalidate* the cache? (There are *snoopy cache protocols* and *directory-based protocols* to tell a processor when this is necessary).

OpenMP Overhead

Results (selected) of running epcc micro-benchmarks on one node of Union Square



Conditional Compilation

Maintain single source for sequential and parallel code even with runtime functions.

```
#ifdef _OPENMP
   #include <omp.h>
#else
   #define omp_get_thread_num() 0
#endif

int  tid = omp_get_thread_num();
```

Compilers that support openMP define `_OPENMP` for testing.

Runtime Environment

Can set runtime vars (or query from within program) to control:

- OMP_NUM_THREADS - sets number of threads to use. (omp_set_num_threads(pos. integer) at runtime)
- OMP_DYNAMIC true/false - to permit or disallow system to dynamically adjust number of threads used in future parallel regions. (omp_set_dynamic(flag) at runtime)
- OMP_NESTED to find out if parallel nesting allowed (omp_set_nested or omp_get_nested at runtime)
- OMP_SCHEDULE to set default scheduling type for parallel loops of type *runtime*

Also runtime calls `omp_get_num_threads()`, `omp_in_parallel()`, `omp_get_thread_num()`, `omp_get_num_procs()`

May need to change stack size `limit stacksize unlimited` (default on Gauss is 8M)

Dynamic/Nested Threads

If supported, dynamic threads set by

- the `omp_set_dynamic()` library routine
- setting the **OMP_DYNAMIC** environment variable TRUE

Allows number of threads to be controlled at runtime using num_threads clause or omp_set_num_threads() function.

Same for Nested Threads (`omp_set_nested()` or **OMP_NESTED**) API also provides for (but implementation may not support)

- Nested parallelism (parallel constructs inside other parallel constructs)
- dynamically altering number of threads in different parallel regions

Standard says nothing about parallel I/O.

Number of Threads

The number of threads is determined in order of precedence by:

- Evaluation of **if** clause (if evaluates to zero - false- serial execution)
- Setting the **num_threads** clause
- the **omp_set_num_threads()** library function
- the **OMP_NUM_THREADS** environment variable
- Implementation default

Threads numbers from 0 (master thread) to N-1.

False Sharing

False Sharing = when two threads update different data elements in the same cache line.

- Side effect of cache line granularity.
- Can be problem on shared memory machines
- Any time cache line is modified, cache coherence mech. notifies other caches with copies that cache line has been modified elsewhere. Local cache line invalidated, even if different bytes modified. Cache line hops from one cache to the other.

For ex., parallel for loop updating an array with chunk size 1. Suppose 8 threads want to update a[0] to a[7]. Thread 0 updates a[0], invalidating cache line for threads 1 through 7.

Scalability

OpenMP so far for loop level parallelism

- Can use as with MPI - explicitly divide up work between threads. Need to logically partition data and associate with each thread (SPMD). (typically work sharing based on distributing major data structures among threads, most of the data usually private).

- Can use in hybrid programming model - MPI between nodes in a cluster of SMPS, OpenMP for the SMP. Also nested OpenMP.

Memory Placement

- Data allocation controlled by operating system
- On cc-numa architectures, pages of data may be distributed across nodes of a system
- Common allocation policy is *First Touch*: the thread initializing a data objects get the page with that data in its local memory. Threads then follow *owner computes* rule. Need to disallow *thread migration* (threads suspended, when resume are on different cpus) .
- If changing memory access patterns, round robin placement might mitigate bottlenecks. Also could investigate *migrate on next touch* support. (some systems can migrate memory pages during different program phases via library routines or compiler directives - can be expensive).

Typical Bugs *

Data race conditions: hard to find, not reproducible, answer varies with number of threads.

```
for (i=0; i<n-1; i++)  a[i] = a[i] + b[i]
```

Iterations of above loop are completely independent of order of execution.

```
for (i=0; i<n-1; i++)  a[i] = a[i+1] + b[i]
```

This loop has loop-carried dependence, destroys parallelism.

Use checksum approach to insure answers don't change. (How much do answers change when rearranging order of operations?)

*Examples from *Using OpenMP*, by Chapman, Jost and Van Der Pas

Typical Bugs

Default behavior for parallel variables is shared.

```
void compute(int n){
  int i;
  double h,x,sum;

  h = 1.0/(double)/n;
  sum = 0.0;
#pragma omp for reduction (+:sum) shared(h)
  for (i=1; i<=n; i++){
    x = h*((double)i - 0.5);
    sum += (1.0)/(1.0+x*x));
  }
  pi = h * sum;
}
```

Race condition due to forgetting to declare x as private.

Typical Bugs

Default for index variables of parallel for loops is private, but not for loops at a deeper nesting level.

```
int i,j;
#pragma omp parallel for
for (i=0;i<n;i++)
  for (j=0;j<m;j++){
    a[i][j] = compute(i,j)
  }
```

Loop variable `j` shared by default – data race. Explicitly declare private or use `for (int j ....` (Different rules in Fortran - always private) (changed in 3.0?)

Typical Bugs

Problems with private variables:

```
void main (){
. . .
#pragma omp parallel for private(i,a,b)
for (i=0;i<n;i++){
   b++;
   a = b+i;
} /* end parallel for */
c = a + b;
```

- Remember that value of a private copy is uninitialized on entry to parallel region (unless use `firstprivate(b)`)
- the value of the original variable is undefined on exit from the parallel region (unless use `lastprivate(a,b)`)

Good habit to use `default(none)` clause

Typical Bugs

Problems with `master` construct:

```
void main (){
  int Xinit, Xlocal;
. . .
#pragma omp parallel shared(Xinit) private(Xlocal)
{
  #pragma omp master
  {Xinit = 10;}

  Xlocal = Xinit;
} /* end parallel region */
```

`master` doesn’t have implied barrier, so Xinit might not be available or might be flushed to memory when another thread reaches it.

Typical Bugs

`nowait` causes problems:

```
#pragma omp parallel
{
  #pragma omp for schedule(static) nowait
  for (i=0;i<n;i++)
    b[i] = (a[i]+a[i-1])/2.0;
  #pragma omp for schedule(static) nowait
   for (i=0;i<n;i++)
     z[i] = sqrt(b[i]);
 }
```

cannot assume which thread executes which loop iterations. If `n` not a multiple of # threads several algs. exist for distributing the remaining iterations. There is no requirement that the same alg. has to be used in different loops. Second loop might read values of `b` not yet written in first loop. (changed in 3.0?)

Typical Bugs

Illegal use of barrier:

```
#pragma omp parallel
 {
  if (omp_get_thread_num()==0)
  { ...
  #pragma omp barrier
  }
  else
  { ...
  #pragma omp barrier
  }
} /* end parallel region */
```

Each barrier must be encountered by all threads in a team. The runtime behavior of this is undefined.

Typical Bugs

Missing curly braces:

```
#pragma omp parallel
{
   work1();  /* executed in parallel */
   work2();  /* executed in parallel */
}

#pragma omp parallel
   work1();  /* executed in parallel */
   work2();  /* executed sequentially */
```

Need curly brackets for parallel region more than a single statement. (Fortran has explicit mark at end of parallel region)

Typical Bugs

Library or function call must be thread-safe (= able to be accessed concurrently by multiple flows of control.

- Global data isn’t thread-safe (and if meant to have global data then the code must be written to protect it from concurrent writes.)
- Indirect accesses through pointers
- Library routines in Fortran must allocate local data on the stack for thread-safety. (no SAVE statement). In C no volatile variables. No shared class objects in C++)

Also called re-entrant code: code can be partially executed by one task, reentered by another task, and then resumed from the original task. State info. and local variables saved on the stack.

Typical Bugs

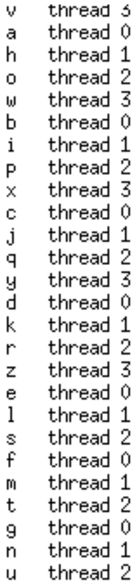
How many times is the alphabet printed in each block?

```
int i;
#pragma omp parallel for
for (i='a'; i<= 'z'; i++)
  printf ("%c",i);
```

```
int i;
#pragma omp parallel
for (i='a'; i<='z';i++)
  printf("%c",i);
```

Typical Bugs

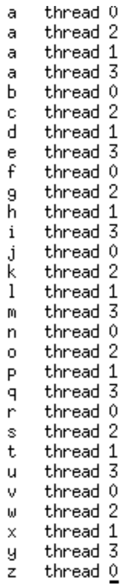
```
int i;
#pragma omp parallel for
for (i=’a’; i<= ’z’; i++)
  printf ("%c",i);
```



```
v    thread 3
a    thread 0
h    thread 1
o    thread 2
w    thread 3
b    thread 0
i    thread 1
p    thread 2
x    thread 3
c    thread 0
j    thread 1
q    thread 2
y    thread 3
d    thread 0
k    thread 1
r    thread 2
z    thread 3
e    thread 0
l    thread 1
s    thread 2
f    thread 0
m    thread 1
t    thread 2
g    thread 0
n    thread 1
u    thread 2
```

Typical Bugs

```
int i;
#pragma omp parallel
for (i='a'; i<= 'z'; i++)
  printf ("%c",i);
```



```
a    thread 0
a    thread 2
a    thread 1
a    thread 3
b    thread 0
c    thread 2
d    thread 1
e    thread 3
f    thread 0
g    thread 2
h    thread 1
i    thread 3
j    thread 0
k    thread 2
l    thread 1
m    thread 3
n    thread 0
o    thread 2
p    thread 1
q    thread 3
r    thread 0
s    thread 2
t    thread 1
u    thread 3
v    thread 0
w    thread 2
x    thread 1
y    thread 3
z    thread 0
```

Memory Model

When are new values of shared data guaranteed to be available to other threads besides the one that did the update? (i.e. when should the updating thread write back to memory or invalidate other copies of data). In what order should the updates be performed?

Memory Model

When are new values of shared data guaranteed to be available to other threads besides the one that did the update? (i.e. when should the updating thread write back to memory or invalidate other copies of data). In what order should the updates be performed?

Sequential consistency (Lamport): multiprocessor is sequentially consistent if

(1) the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and

(2) the operations of each individual processor appear in this sequence in the order specified by its program.

Memory Model

When are new values of shared data guaranteed to be available to other threads besides the one that did the update? (i.e. when should the updating thread write back to memory or invalidate other copies of data). In what order should the updates be performed?

Sequential consistency (Lamport): multiprocessor is sequentially consistent if

(1) the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and

(2) the operations of each individual processor appear in this sequence in the order specified by its program.

Other models of relaxed consistency have been proposed. (i.e. a read can complete before an earlier write to a different address, but a read cannot return the value of a write by another processor unless all processors see the write (but it returns the value of its own write before others see it).

Memory Model

Both threads can end up with old values, which violates sequential consistency.

```
Processor P1                Processor P2

x = new;                     y = new;
y_copy = y;                  x_copy = x;
```

In any serialization at least one of the processors should end up with a new value.

Very readable summary ”Shared Memory Consistency Models: A Tutorial”, by Adve and Gharachorloo, in IEEE Computer, Dec., 1996.

Memory Model

OpenMP defines points in program at which data in shared memory must be made current and at which each thread must make its modifications available to all other threads. The threads do not need to update data in any specific order. As long as there are no race conditions there should be no impact on resulting values.

OpenMP model: each thread has its own temporary view of the values of shared data between explicit or implicit barrier synchronization points. If shared variables accessed between these points, user must take care to keep the temporary memory view of the threads consistent.

Memory Model

Cache coherence not enough to prevent race conditions - is only a protocol for updating cache lines, but does not specify when and in what order results written back to memory - that is specified by the memory consistency model. Relaxed consistency does not guarantee that a write operation from one thread finishes before another thread reads from same address.

CC triggers on a store. If variables kept in registers modifications may not propagate back to memory without explicit *flush* directives.

Compiler may reorder a flush operation relative to code that does not affect the variables being flushed. Compilers may move flush operations with respect to each other.

Memory Model

Reasoning about *flush* directive is tricky.

```
signal = 0;
#pragma omp parallel default(none) shared(signal,newdata) \
                                  private(TID,localdata)
{
  TID = omp_get_thread_num();
  if (TID == 0) {
    newdata = 10;
    signal = 1;
  }
  else {
    while (signal == 0) {}
    localdata = newdata;
  }
} /* end parallel */
```

Memory Model

Wrong use of flush to synchronize! Compiler can reorder flush statements.

```
signal = 0;
#pragma omp parallel shared(signal,newdata) private(TID,locald
{
  TID = omp_get_thread_num();
  if (TID == 0) {
    newdata = 10;
#pragma omp          flush(newdata)
    signal = 1;
#pragma omp          flush(signal)
  }
  else {
#pragma omp          flush(signal)
    while (signal == 0) {
#pragma omp          flush(signal)
     }
#pragma omp          flush(newdata)
    localdata = newdata;
   }
} /* end parallel */
```

Memory Model

Need fence with both items to prevent interchange of assignments.

```
signal = 0;
#pragma omp parallel shared(signal,newdata)  private(TID,local
{
  TID = omp_get_thread_num();
  if (TID == 0) {
    newdata = 10;
#pragma omp           flush(newdata,signal)
    signal = 1;
#pragma omp           flush(signal)
  }
  else {
#pragma omp           flush(signal)
    while (signal == 0) {
#pragma omp           flush(signal)
      }
#pragma omp           flush(newdata)
    localdata = newdata;
    }
} /* end parallel */
```

Memory Model

Common Fortran technique:

```
program main
!$OMP PARALLEL DEFAULT(NONE) PRIVATE(...)
   call mysubroutine()
!$OMP END PARALLEL
stop
end
subroutine mysubroutine()
implicit none
real, save      :: a
real            :: a_inv
logical, save   :: first
data first/.true./

if (first) then
  a = 10.0
  first = .false.
end if
a_inv = 1.0/a
return
end
```

Need `!$omp single` and `!$omp end single` around initialization.

Performance Issues

- If data carefully mapped via First Touch placement use of `single` construct could destroy performance. (e.g. say data was initialized by master thread, then use `master` construct instead. Or explicitly map work to individual threads based on location of data.
- Use private data - allocated on stack, and stack in thread's local memory. This is not the case with shared data. Shared data could be fetched and cached for reuse, but if evicted from cache another expensive transfer from remote memory will be needed, so private data better.

Performance Issues

Use profiling to show where program spends most of its time.

- state of a thread: waiting for work, synchronizing, forking, joining, doing useful work.
- Time spent in parallel regions and work-sharing constructs
- time spent in user and system level routines
- hardware counter info: CPU cycles, instructions, cache miches, tlb misses
- time spend in communcation, message length, number of messages

Look at wall-clock time (=elapsed time) vs. CPU time (=user + system time summed over all threads, so will not decrease, and likely will increase due to overhead).

Speedup and Efficiency

- On p processors speedup $S(p) = T_1/T_p$
- Linear (ideal) speedup: on p procs code is p times faster
  - this doesn't usually happen due to overhead, contention, other bottlenecks
  - however can sometimes observe *superlinear* speedup due to cache effects (smaller problem fits more easily in cache)
- Efficiency $E = S(p)/p$
  - Ideal speedup $S = T_1/T_p = T_1/(T_1/p) = p$ has 100% efficiency
  - Amdahl's law for serial fraction of code f says $T_p = f + (1-f)/p \quad \text{so} \quad S < 1/f$

    $E = S/p = T_1/(T_p\,p) = 1/(f\,p) \to 0$

Scaled Speedup

Strong scaling as above, run the same code using increasing number of processors; the problem size stays *constant*.

Weak scaling increase *size* of the problem as increase # cpus.

- originally due to Gustafson to get around Amdahl's law. Keep problem size n constant on *each processor*, so total problem size grows linearly with p
- Problem if time complexity of algorithm grows superlinearly with problem size. For example,

  for data size n, let $T_1 = n^2$

  for data size pn, parallel time is $T_p = p^2 n^2 / p = p \cdot T_1$
  so parallel time increases linearly with p even with ideal scaling.
- Hard to measure fairly. Problem doesn't all fit on 1 cpu.

See Dave Bailey's article "Twelve Ways to Fool the Masses"

`http://crd.lbl.gov/~dhbailey/dhbpapers/twelve-ways.pdf`

References

- http://computing.llnl.gov/tutorials/openMP/
  very complete description of OpenMP for Fortran and C
- Rauber and Runger text
  text has small OpenMP section in chapter 6.
- *Using OpenMP*
  Portable Shared Memory Parallel Programming
  by Chapman, Jost and Van Der Pas
- http://www.openmp.org