OpenMP

Introduction

- Enhanced performance for applications is the only reason to go to parallel computers
- Parallel systems (with multiple, low cost, commodity microprocessors) can have significant cost advantage over even the best single processor computer
- Application developer has to design and program correct and efficient parallel code for multiprocessors
- Result is the combined effect of application performance and price performance

Introduction : Performance with OpenMP

- OpenMP provides the option for *incremental parallelization*
- Ability to parallelize an application at a rate where the developers feel additional effort is necessary and provides the performance for the price
- MPI might have more of a large investment type approach from the beginning
- Incremental parallelization can be done in different versions of a code allowing a conservative development path
- Suitable for apps that have been developed and tested for many years

Introduction : OpenMP code

- OpenMP codes are closer to sequential codes
  - What needs to be done
  - Input/output
  - Algorithms designed to do the work
  - In case of parallel program how the work is distributed among processors
- OpenMP supports the final step
- OpenMP works with Fortran and C/C++ (C++ might be limited)
- OpenMP is a set of compiler directives to describe parellelism in the source code
- OpenMP also has library routines

Introduction : First OpenMP code

```
        program hello
        print *, “Hello from threads:”
!$OMP  parallel
        print *, omp_get_thread_num()
!$OMP end parallel
        print*, “Back to sequential:”
        end
```

Introduction : OpenMP code

- Set OMP_NUM_THREADS to 4
- Within OpenMP directives three additional copies of the code are started (what does this mean?)
- Each copy is called a thread or thread of execution
- The OpenMP routine omp_get_thread_num() reports unique thread# between 0 and OMP_NUM_THREADS-1

Introduction : Output of code

Output after running on 4 threads :

Hello from threads:

1

0

3

2

Back to sequential:

Analysis of OpenMP output :

Threads are working completely independent

Threads may be forced to cooperate to produce correct and efficient results - leads to synchronization

Introduction : OpenMP Parallel Computers

- Primarily for shared memory parallel computers
- All processors are able to directly access all of the memory
- UMA/SMPs (Compaq AlphaServers, all multiprocessor PC and workstations, SUN Enterprise) and distributed shared memory (DSM) or ccNUMA (SGI Origin 2000, HP V-Class)
- SMPs are ~32 procs and ccNUMAs can go upto from 100s to even 1000s procs
- OpenMP codes usually result in few % to at most 20% increase in code size compare to sequential codes (MPI codes can result in 50% to few 100% increase in size)

OpenMP Partners

- OpenMP web site at http://www.openMP.org

The OpenMP Architecture Review Board (1997) is comprised of the following organizations.

- Compaq
- Hewlett-Packard Company
- Intel Corporation
- International Business Machines (IBM)
- Kuck & Associates, Inc. (KAI)
- Silicon Graphics, Inc.
- Sun Microsystems, Inc.
- U.S. Department of Energy ASCI program

The following software vendors also endorse the OpenMP API:

- Absoft Corporation
- Edinburgh Portable Compilers
- Etnus, Inc.
- GENIAS Software GmBH
- Myrias Computer Technologies, Inc.
- The Portland Group, Inc. (PGI)

Summary of OpenMP Introduction

- MPI has its advantages and disadvantages
- Pthreads is an accepted standard for shared memory in the low end
  - Not for HPC
  - Little support for Fortran
  - More suitable for task than data parallel
- OpenMP : incremental parallelization
- OpenMP : compiler directives and library calls
- Directive allows to write portable codes since they are ignored by non-openMP compilers

High Level

OpenMP

OpenMP: Fortran

- OpenMP compiler directives :
  12345 6 (columns)
  !$omp <directive>
  C$omp <directive>
  *$omp <directive>
  must contain a space or zero in the 6th column
- Treated as OpenMP directive by an OpenMP compiler and treated as a comment by non-openMP compilers
- In fixed form a line begins with !$omp; in free format in any column we have !$omp preceded only by white spaces
- Continuation expressed as :

  !$omp <directive>&

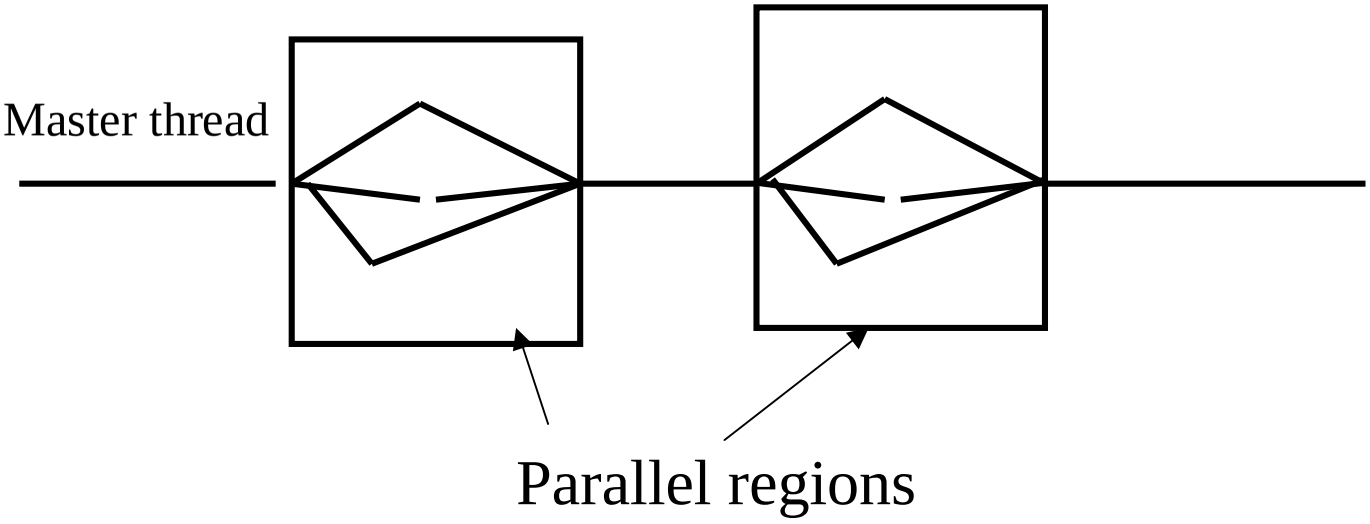
OpenMP : C

- #pragma omp

- In general make sure conditional compilation is used with care

OpenMP: Programming Model

- Fork/Join parallelism
  - Master thread spawns threads as needed
  - Parallelism is added incrementally i.e. the sequential program evolves into a parallel program



- Two basic kinds of constructs for controlling parallelism :
- Directive to create multiple threads of execution that execute concurrently
  - Used to execute multiple structured blocks concurrently
  - “parallel” directive
- Directive to divide work among existing set of parallel threads
  - Used to do loop level parallelism
  - “do” in fortran and “for” in C

OpenMP Data Sharing

- OpenMP begins with single thread of control that has the execution context or data environment : master thread
- The execution context is the data address space containing all the variables in the program : all the global, subroutine variables (allocated on the stack) and dynamically allocated variables (in the heap)
- Master thread exists for the duration of the entire program
- During parallel construct new threads of execution are created

- Each thread has its own stack within its execution context, hence multiple threads can individually invoke subroutines and execute safely without interfering the stack frames of other threads
- For other program variables OpenMP parallel construct:
  - Can share a single copy between all the threads
  - Can provide each thread with its own copy
- Same variable can be shared within one parallel construct and private in another

Shared Data

- A ***shared*** variable will have single storage location in the memory for the entire duration of that parallel construct
- All the threads will access the same memory location
- Read/Write operations will allow communication between multiple OpenMP threads

Private Data

- A variable that has ***private*** scope will have multiple storage locations
- Execution context of each thread will have a copy of the variable for the duration of the parallel construct
- All read/write operations on that variable by a thread will refer to the private copy
- This memory location is inaccessible to other threads

Reduction Data

- ***reduction*** variables have both private and shared behavior
- These variables are the target of an arithmetic operation
- Example is final summation of temporary local variables at the end of a parallel construct

Synchronization

- Multiple OpenMP threads communicate with each other through ordinary reads and writes
- Coordination is necessary so that they don't simultaneously attempt to modify variables or read when a variable is being modified
  - This can lead to incorrect results without warning
  - This can also produce different results in different runs
- Mutual exclusion : ***critical*** directive allows a thread exclusive access to a shared variable for the duration of the construct
- Event synchronization : ***barrier*** directive signals occurrence of an event across multiple threads
- There are other synchronization constructs

Simple Loop Parallelization

```
subroutine sub1(z, a, x, y, n)
integer i , n
real z(n), a, x(n), y
do i = 1, n
      z(i) = a*x(i) + y
enddo
return
end
```

- No dependences in the above loop : result of one iteration doesn’t depend on result of any other iteration
- 2 procs can simultaneously execute two iterations
- Use **parallel do**

Fortran Parallel Loop

```
      subroutine sub1(z, a, x, y, n)
      integer i , n
      real z(n), a, x(n), y
!$omp parallel do
      do i = 1, n
            z(i) = a*x(i) + y
      enddo
      return
      end
```

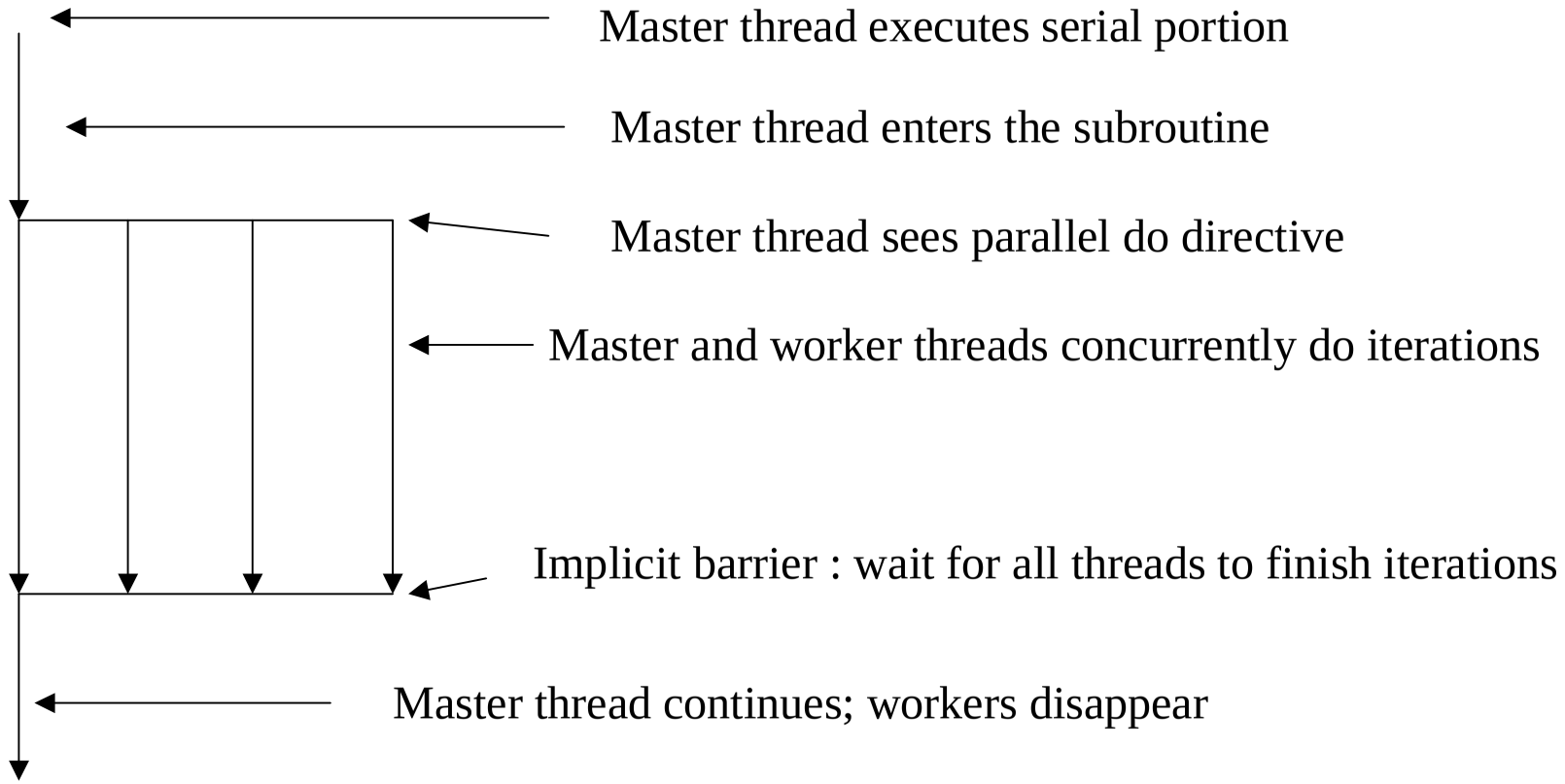
- Directive followed by do loop construct says to execute the iterations concurrently across multiple threads
- An openMP compiler creates multiple threads and distributes the iterations of the loop across threads for parallel execution

C Parallel Loop

```
void routine (float z[n], float a, float x[n], float y, n) {
   int i ;

     #pragma omp parallel for
      for ( i = 0 ;  i <n ; i++ )  {
        z[i] = a*x[i] + y ;
       }
   }
```

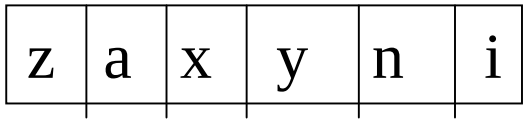
Execution Model



- OpenMP doesn’t specify:
  - How threads are implemented
  - How unique and distinct set of iterations are assigned to threads

Data Sharing

Global shared memory

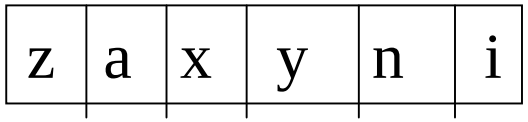


Serial execution (master thread only)

All data reference to global shared

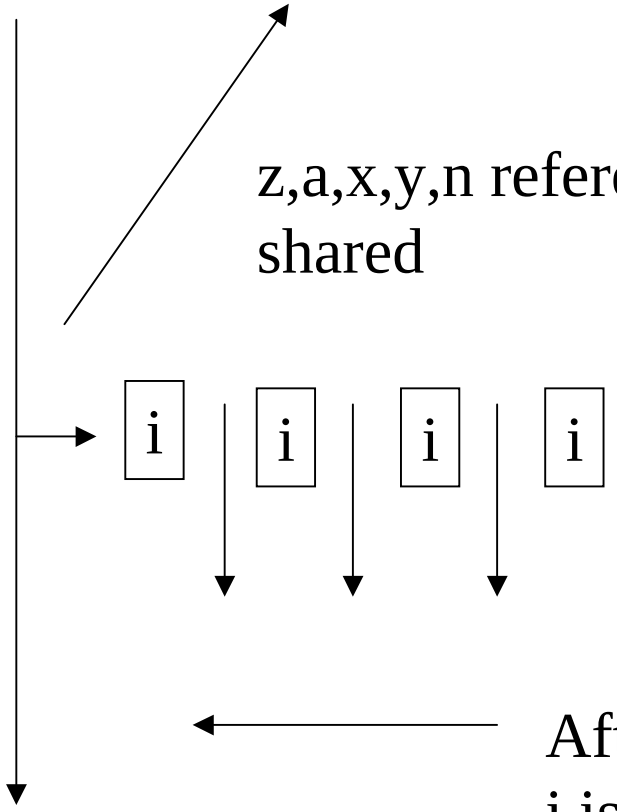
Data Sharing

Global shared memory



Parallel execution (multiple threads)

Each thread has a private copy of i referenced by each thread



z,a,x,y,n references to global shared

Initial i undefined

After parallel construct i is also undefined

Synchronization

- Synchronization for Z
  - Multiple threads modify shared variable Z
  - Each thread modifies distinct element of Z
  - No data conflict; no explicit synchronization
- Master thread needs to see all updated value of Z after the parallel loop
  - Only master thread executes after parallel do/for
  - Parallel do/for has **implied barrier** at the end for all threads including master thread – guarantees that all iterations have completed and all Z values updated

Simple Loop Parallelization

- Easy to express
- Can be used to parallelize large codes by incrementally parallelizing individual loops
- Problems :
  - Some apps may not have many loops
  - Overhead of joining threads at the end of each loop – this is a synchronization point and need to wait for the slowest one

More Loop Parallelization

```
real*8 x, y, x1, y1
integer i, j, m, n
integer distance(*,*)
integer function
……..
x1 = 10.
y1 = 20.
do i = 1, m
   do j = 1, n
         x = i / real(m)
         y = j / real(n)
         distance(i, j) = function(x, y, x1, y1)
   enddo
enddo
```

- function takes x, y, x1 and y1 and calculates the distance between two points (x,y) and (x1, y1)
- Are different iterations independent of each other ?
- Look at source code of function to see if thread safe
- Scalar variables j, x and y are assigned and function called in the innermost loop
- **parallel do** will distribute iterations of the outermost loop among threads
- By default i is private and everything else is shared
- m and n are only read – ok to have them shared
- Loop index j needs to be private – why ?
- x and y need to be private also – why ?
- distance is modified inside the loop – synchronization
- No explicit sync required due to implicit barrier of parallel do

Parallel code

```
          ……….

          x1 = 10.
          y1 = 20.
!$omp parallel do private(j, x, y)
          do i = 1, m
          do j = 1, n
                    x = i / real(m)
                    y = j / real(n)
                    distance(i, j) = function(x, y, x1, y1)
           enddo
          enddo
```

Synchronization

```
x1 = 10.
y1 = 20.
total_dist = 0.
do i = 1, m
do j = 1, n
        x = i / real(m)
        y = j / real(n)
        distance(i, j) = function(x, y, x1, y1)
        total_dist = total_dist + distance(i,j)
 enddo
enddo
```

How does parallelization change ?

- Cannot just make total_dist shared
- total_dist needs to be shared but total_dist (a shared variable) is modified in the parallel portion of the code
- Multiple threads write to total_dist – no guaranteed ordered among multiple threads reads/writes – many possibilities of wrong result
- This is race condition on access to a shared variable
- Access needs to be controlled through synchronization
- **critical/end critical** construct can be executed by one thread at a time
- First thread to reach critical executes code – all other wait until current thread is done
- One thread at a time updates value total_dist

```
            x1 = 10.
            y1 = 20.
            total_dist = 0.
!$omp parallel do private(j, x, y)
            do i = 1, m
                  do j = 1, n
                        x = i / real(m)
                        y = j / real(n)
                        distance(i, j) = function(x, y, x1, y1)
!$omp critical
                         total_dist = total_dist + distance(i,j)

!$omp end critical
                  enddo
            enddo
```

(Explicit synchronization - inserted by programmer)

Reduction clause

- critical section protects shared variable total_dist
- Basic operation was sum reduction
- Reductions are common enough that openMP has reduction data scope clause

```
                total_dist = 0.
!$omp parallel do private(j, x, y)
!$omp& reduction (+:total_dist)
                do i = 1, m
                        do j = 1, n
                            x = i / real(m)
                            y = j / real(n)
                              distance(i, j) = function(x, y, x1, y1)
                              total_dist = total_dist + distance(i,j)

                        enddo
                enddo
```

- Reduction clause tells compiler that total_dist is the target of a sum reduction
- Many other mathematical operations possible in reduction
- Compiler and runtime environment will implement the reduction in an efficient manner for the target machine
- Reduction is a data attribute distinct from either shared or private and has elements of both shared and private data