OpenMP programming

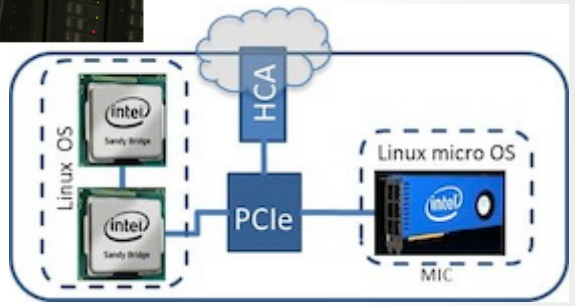
Thomas Hauser
Director Research Computing
thomas.hauser@colorado.edu

Outline

- OpenMP
- Shared-memory model
- Parallel `for` loops
- Declaring private variables
- Critical sections
- Reductions

Parallelism at All Levels

- Parallelism across multiple nodes or process - MPI
- Parallelism across threads - OpenMP
- Parallelism across instructions
- Parallelism on data – SIMD Single Instruction Multiple Data

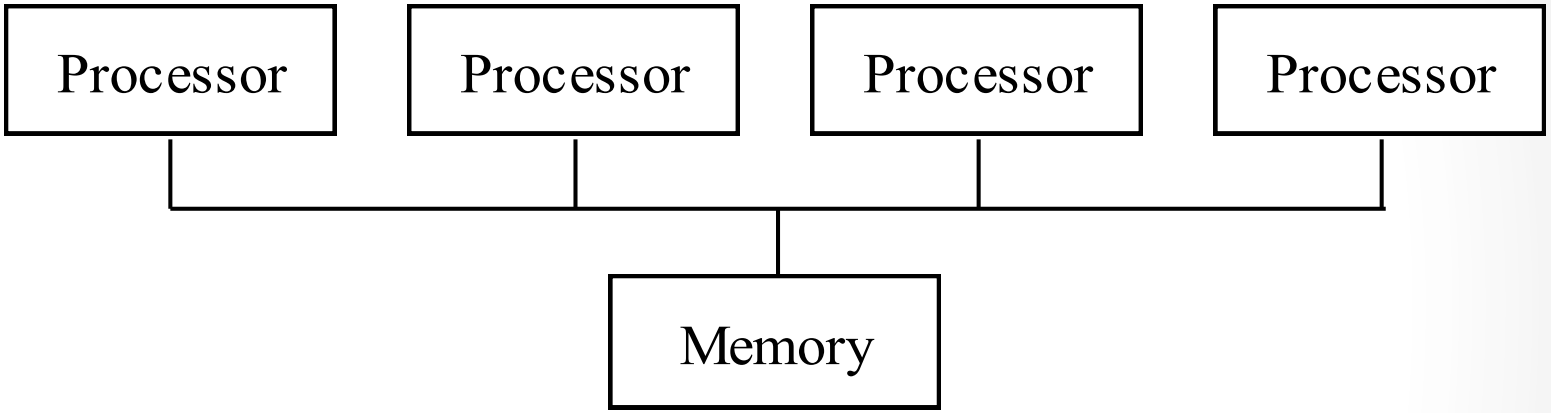


www.scan.co.uk

OpenMP

- OpenMP: An application programming interface (API) for parallel programming on multiprocessors
  - Compiler directives
  - Library of support functions
  - Environment variables
- OpenMP works in conjunction with Fortran, C, or C++
- http://openmp.org

Shared-memory Model

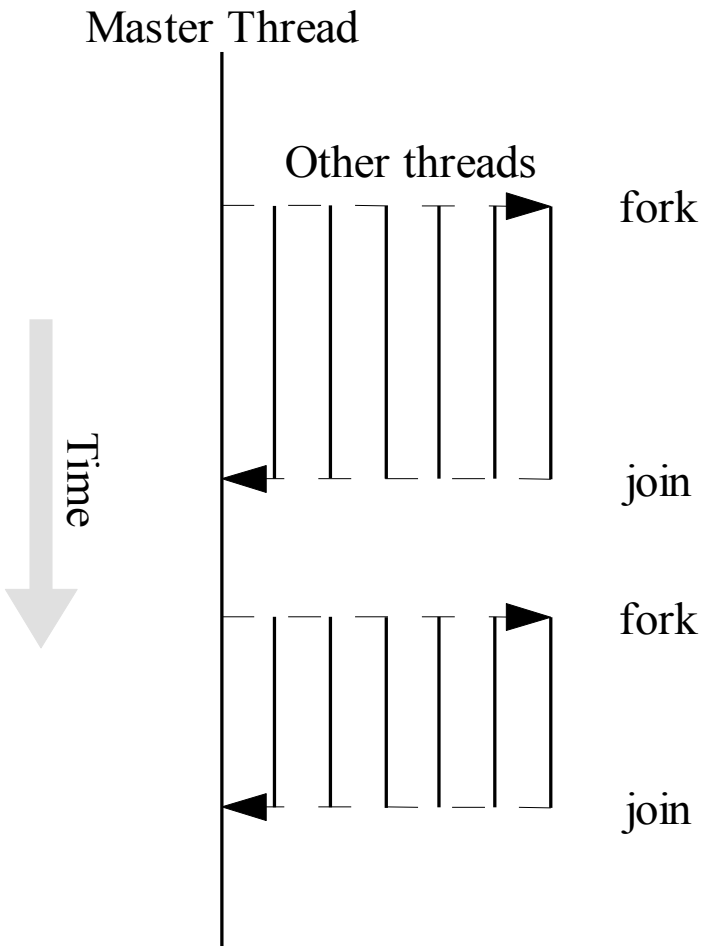


Processors interact and synchronize with each other through shared variables.

Fork/Join Parallelism

- Initially only master thread is active
- Master thread executes sequential code
- Fork: Master thread creates or awakens additional threads to execute parallel code
- Join: At end of parallel code created threads die or are suspended

Fork/Join Parallelism



Shared-memory Model vs. Message-passing Model (#1)

- Shared-memory model
  - Number active threads 1 at start and finish of program, changes dynamically during execution
- Message-passing model
  - All processes active throughout execution of program

Incremental Parallelization

- Sequential program a special case of a shared-memory parallel program
- Parallel shared-memory programs may only have a single parallel loop
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit at a time

Shared-memory Model vs. Message-passing Model (#2)

- Shared-memory model
  - Execute and profile sequential program
  - Incrementally make it parallel
  - Stop when further effort not warranted
- Message-passing model
  - Sequential-to-parallel transformation requires major effort
  - Transformation done in one giant step rather than many tiny steps

Parallel for Loops

- Fortran programs often express data-parallel operations as **do** loops

```
do i=1,nx, bs
   a(i) = b(i)+c(i)
enddo
```

- OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel
- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads

Compiler directive

- Compiler directive in Fortran
- A way for the programmer to communicate with the compiler
- Compiler free to ignore directives
- Syntax:

```
!$OMP <rest of directive>
!$OMP& continuation line
!$ INTEGER :: i !something only in the
  parallel version of the program
```

Pragma in C or C++

- Syntax:

```
#pragma omp <rest of directive>
```

Parallel do directive

- Fortran:

```
!$OMP parallel do
do i=1,10
    a(i) = b(i) + c(i)
end do
!$OMP end parallel do
```

- Compiler must be able to verify the run-time system will have information it needs to schedule loop iterations

- C

```
#pragma omp for
for (i=0;i<10;i++){
    a[i] = b[i] + c[i];
}
```

Execution Context

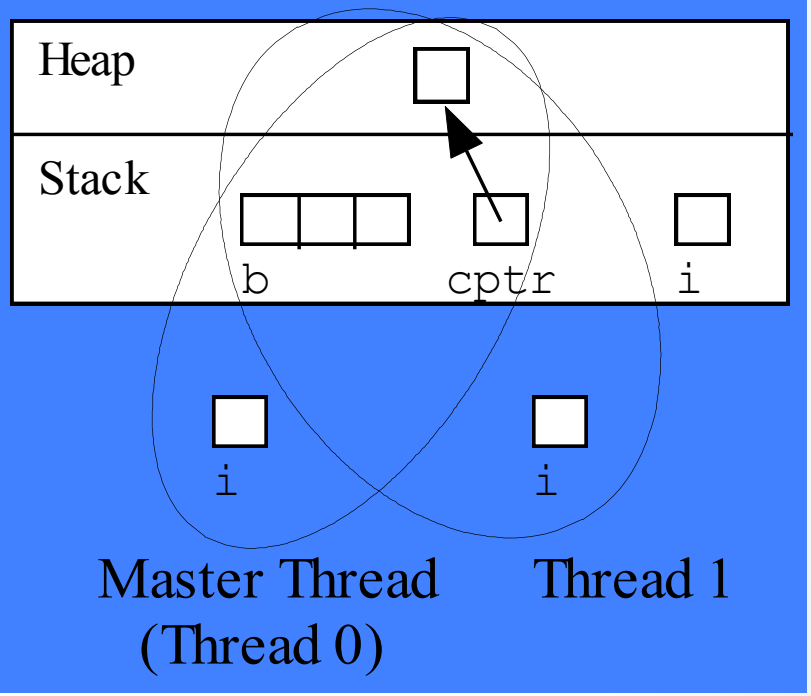
- Every thread has its own execution context
- Execution context: address space containing all of the variables a thread may access
- Contents of execution context:
  - static variables
  - dynamically allocated data structures in the heap
  - variables on the run-time stack
  - additional run-time stack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

Shared and Private Variables

- Shared variable: has same address in execution context of every thread
- Private variable: has different address in execution context of every thread
- A thread cannot access the private variables of another thread

Shared and Private Variables

```
int main (int argc, char *argv[])
{
   int b[3];
   char *cptr;
   int i;

   cptr = malloc(1);
#pragma omp parallel for
   for (i = 0; i < 3; i++)
      b[i] = i;
```



Using the GCC compilers with OpenMP

- gfortran -fopenmp –g basic.f90
- gcc -fopenmp -g basic.c
- setenv OMP_NUM_THREADS 16 if tcsh/csh
- export OMP_NUM_THREADS=16 if bash/sh
- ./a.out

Using the Intel compilers with OpenMP

- ifort -openmp –g basic.f90
- icc -openmp -g basic.c
- setenv OMP_NUM_THREADS 16 if tcsh/csh
- export OMP_NUM_THREADS=16 if bash/sh
- ./a.out

Stream example

- Python notebook demo

Function omp_get_num_procs

- Returns number of physical processors available for use by the parallel program
- C: **int omp_get_num_procs**(void);
- Fortran:

```
interface
   function omp_get_num_procs ()
      use omp_lib_kinds
      integer ( kind=omp_integer_kind ) :: omp_get_num_procs
   end function omp_get_num_procs
end interface
```

Function omp_get_thread_num

- This function returns the thread identification number
- If there are *t* threads, the ID numbers range from 0 to *t*-1
- The master thread has ID number 0

```
integer omp_get_thread_num ()
```

single Directive

- Suppose we only want to see the output once
- The **`single`** directive directs compiler that only a single thread should execute the block of code the directive precedes
- Syntax:

```
!$omp single
```

Use of single Pragma

```
!$omp parallel private(i,j)
Do i = 1, m
   low = a(i)
   high = b(i)
   if (low > high) then
      !$OMP single
      write(*,*) "Exiting", I
      !$OMP end single
      exit
   endif
End do
!$omp do
do j = low, high
  c(j) = (c(j) - a(i))/b(i)
End do
!$omp end do
!$omp end parallel
```

Master directive

- Code is executed by the master tread
- Otherwise very similar to the single directive

Function omp_set_num_threads

- Uses the parameter value to set the number of threads to be active in parallel sections of code
- May be called at multiple points in a program

```
subroutine omp_set_num_threads (t)
integer, intent(in) :: t
```

void omp_set_num_threads(int *num_threads*);

Hello World - OpenMP

- See ipython notebook

Declaring Private Variables

```
do j = 1,nj
  do i = 1, ni
    a(i,j) = min(a(i,j), a(i,j)+tmp)
  end do
end do
```

- Either loop could be executed in parallel
- We prefer to make outer loop parallel, to reduce number of forks/joins
- We then must give each thread its own private copy of variable `i`

private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private

```
private ( <variable list> )
```

Example Use of private Clause

```
!$OMP parallel do private(i)
DO j = 1, nj
   DO i = 1, ni
      a(i,j) = MIN(a(i,j),a(i,k)+tmp)
   END DO
END DO
```

```
#pragma omp for private(i)
for (j=0;j<nj;j++)
  for (i=0; i<ni; i++) {
    a[j,i] = a[j,i]+tmp
}
```

firstprivate Clause

- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
- Variables are initialized once per thread, not once per loop iteration
- If a thread modifies a variable's value in an iteration, subsequent iterations will get the modified value

Use of firstprivate

```
program wrong
I = 10
!$OMP PARALLEL PRIVATE(I)
I= I + 1
print *, I
!$OMP END PARALLEL
```

```
program correct
I = 10
!$OMP PARALLEL FIRSTPRIVATE(I)
I= I + 1
print *, I
!$OMP END PARALLEL
```

lastprivate Clause

- Sequentially last iteration: iteration that occurs last when the loop is executed sequentially
- **`lastprivate`** clause: 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

Numerical Integration

$$\pi = \int_0^1 \frac{4.0}{1+x^2} dx$$

$$\sum_{i=0}^{N} f(x_i) dx_i \approx \pi$$

Critical Sections

```
integer n, i
double precision w, x, area,
pi, f, a

! function to integrate
f(a) = 4.d0 / (1.d0 + a*a)

w = 1.0d0/n
area = 0.0d0

do i = 1, n
  x = w * (i - 0.5d0)
  area = area + f(x)
end do
pi = w * area
```

Race Condition (cont.)

- If we simply parallelize the loop...

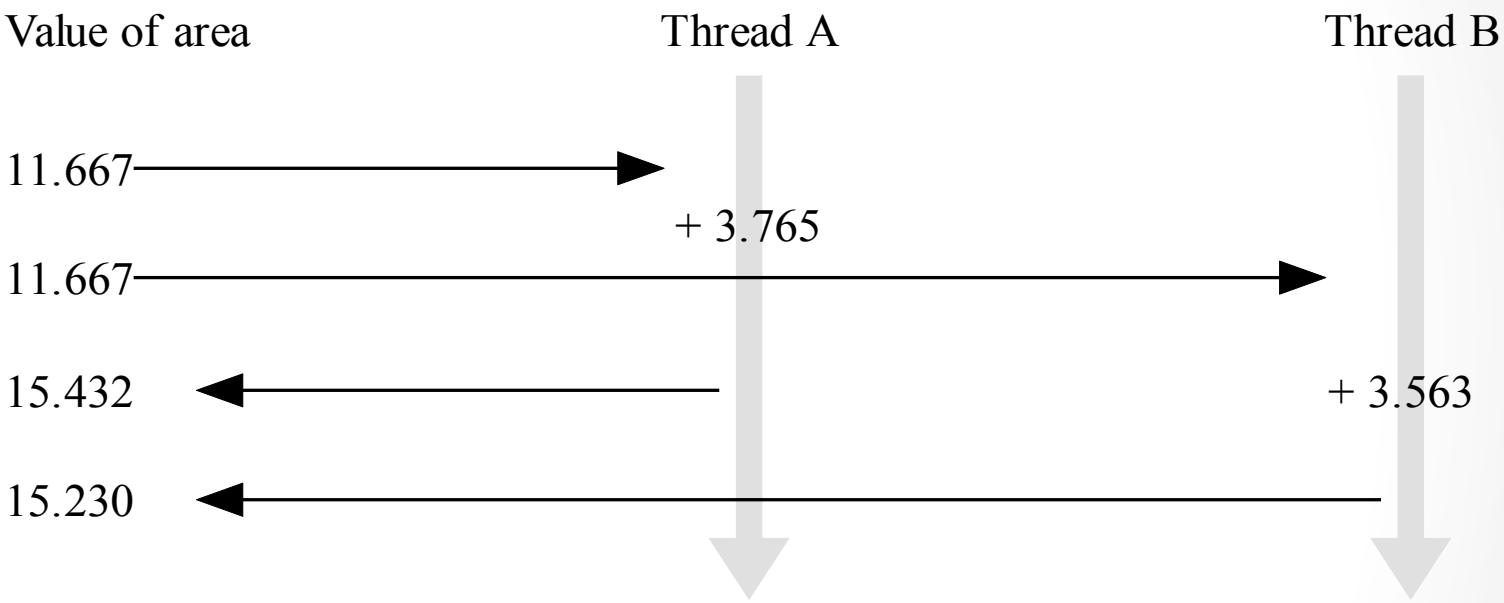
```
area = 0.0d0

!$OMP PARALLEL DO PRIVATE(x)
do i = 1, n
  x = w * (i - 0.5d0)
  area = area + f(x)
end do
!$OMP END PARALLEL DO
pi = w * area
```

Race Condition (cont.)

- ... we set up a race condition in which one process may "race ahead" of another and not see its change to shared variable **`area`**


```
area = area +4.0/(1.0 + x*x)
```

Race Condition Time Line



critical directive

- Critical section: a portion of code that only thread at a time may execute
- We denote a critical section by putting the directive

  !$OMP critical

  in front of a block of Fortran code

Correct, But Inefficient, Code

```
! function to integrate
f(a) = 4.d0 / (1.d0 + a*a)

w = 1.0d0/n
area = 0.0d0
!$OMP PARALLEL DO PRIVATE(X), SHARED(w,n)
do i = 1, n
  x = w * (i - 0.5d0)
  !$OMP CRITICAL
  area = area + f(x)
  !$OMP END CRITICAL
end do
!$OMP END PARALLEL DO
pi = w * area
```

Source of Inefficiency

- Update to **`area`** inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code
- Time to execute statement significant part of loop
- By Amdahl's Law speedup will be severely constrained

Reductions

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to **`parallel do`** directive
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop

reduction Clause

- The reduction clause has this syntax:
  `reduction (<op> :<variable>)`
- Operators
  - +
  - *
  - -
  - .AND.
  - .OR.
  - .EQV.
  - .NEQV.
  - MAX
  - MIN
  - IAND
  - IOR
  - IEOR

π-finding Code with Reduction Clause

```
! function to integrate
f(a) = 4.d0 / (1.d0 + a*a)

w = 1.0d0/n
area = 0.0d0
!$OMP PARALLEL DO PRIVATE(X), SHARED(w,n)
!$OMP& REDUCTION(+:area)
do i = 1, n
  x = w * (i - 0.5d0)
  area = area + f(x)
end do
!$OMP END PARALLEL DO
pi = w * area
```

Summary

- OpenMP an API for shared-memory parallel programming
- Shared-memory model based on fork/join parallelism
- Data parallelism
  - parallel for pragma
  - reduction clause