Parallel Computing with OpenMP
Parallel Computing

• Parallel computing benefits
  » decrease the wall-time taken by program execution
    » Several processes work together to solve the problem
    » Total CPU time does not decrease
  » increase the size of the problem that can be solved

• Parallel programming is more difficult than sequential programming
Why OpenMP?

• Parallelism in selected regions
• OpenMP is a scalable, portable, incremental approach to designing shared memory parallel programs
• OpenMP supports
  – fine and coarse grained parallelism
  – data and control parallelism
What is OpenMP?

Three components:

- **Set of compiler directives for**
  - creating teams of threads
  - sharing the work among threads
  - synchronizing the threads

- **Library routines** for setting and querying thread attributes

- **Environment variables** for controlling runtime behavior of the parallel program
Elements of OpenMP

- Parallel regions and work sharing
- Data scoping
- Synchronization
- Compiling and running OpenMP programs
Parallelism in OpenMP

- The *parallel region* is the construct for creating multiple threads in an OpenMP program.
- A *team of threads* is created at run time for a parallel region.
- A *nested* parallel region is allowed, but may contain a team of one thread.
- Nested parallelism is enabled with `setenv OMP_NESTED TRUE`.
Parallelism in OpenMP

beginning of parallel region

fork
join
fork
join

end of parallel region
Hello World in OpenMP

#include <omp.h>
int main() { 
    int iam =0, np = 1;
    #pragma omp parallel private(iam, np) 
    {
        #if defined (_OPENMP)
            np = omp_get_num_threads();
            iam = omp_get_thread_num();
        #endif
        printf("Hello from thread %d out of %d \n", iam, np);
    }
}
Specifying Parallel Regions

• Fortran

```fortran
!$OMP PARALLEL [clause [clause...]]
! Block of code executed by all threads
!$OMP END PARALLEL
```

• C and C++

```c
#pragma omp parallel [clause [clause...]]
{
/* Block executed by all threads */
}
```
Fortran Syntax

!$OMP PARALLEL
!$OMP DO ! clause on
!$OMP&PRIVATE(I) ! next line
  DO I=1,n
    A(I)= A(I) * A(I) - 3.
  END DO
!$OMP END DO NOWAIT ! clause
!$OMP END PARALLEL
```c
#include <omp.h>

#pragma omp parallel
{
    #pragma omp for nowait private(i)
        for(i=0; i<n; i++)
}
```
Compiling and Linking

- **MIPSpro**: compile & link with \(-mp\) option
  - **Fortran**:
    - `f90 [options] -mp -O3 prog.f`
    - \(-\text{freeform}\) needed for free form source
    - \(-\text{cpp}\) needed when using \#ifdefs
  - **C/C++**:
    - `cc -mp -O3 prog.c`
    - `CC -mp -O3 prog.C`

- **AIX/ XL Fortran**: use \(-qsmp\) option
  - `xlf_r -qsmp -O3 prog.f`
Running

• Standard environment variable determines the number of threads:
  – tcsh
    setenv OMP_NUM_THREADS 8
  – sh/bash
    export OMP_NUM_THREADS=8

• Run program and get wall-time:
  time a.out
Work sharing in OpenMP

• Two ways to specify parallel work:
  – Explicitly coded in parallel regions
  – Work-sharing constructs
    » DO and for constructs: parallel loops
    » sections
    » single

• SPMD type of parallelism supported
Work and Data Partitioning

Loop parallelization
• distribute the work among the threads, without explicitly distributing the data.
• scheduling determines which thread accesses which data
• communication between threads is implicit, through data sharing
• synchronization via parallel constructs or is explicitly inserted in the code
Work Sharing Constructs

- **DO** and **for**: parallelizes a loop, dividing the iterations among the threads.
- **sections**: defines a sequence of contiguous blocks, the beginning of each block being marked by a section directive. The block within each section is assigned to one thread.
- **single**: assigns a block of a parallel region to a single thread.
Specialized Parallel Regions

Work-sharing can be specified combined with a parallel region

- **parallel DO and parallel for**: a parallel region which contains a parallel loop
- **parallel sections**: a parallel region that contains a number of section constructs
Parallel Sections

```c
#pragma omp parallel shared(A,B) private(i) 
{
    #pragma omp sections nowait 
    {
        #pragma omp section
        for(i=0; i<n; i++)
        #pragma omp section
        for(i=0; i<n; i++)
            B[i] = B[i]*B[i] + 9.;
    } // end omp sections
} // end omp parallel
```
Message passing vs multithreading

• Process versus thread address space
  – threads have shared address space, but the thread stack holds thread-private data
  – processes have separate address spaces
• For message passing multiprocessing, e.g., MPI, all data is explicitly communicated, no data is shared
• For OpenMP, threads in a parallel region reference both private and shared data
• Synchronization: explicit or embedded in communication
program mat_init
  implicit none
  integer, parameter::N=1024
  real A(N,N)
  integer :: iam, np
  iam = 0
  np = 1
  !$omp parallel private(iam,np)
  np = omp_get_num_threads()
  iam = omp_get_thread_num()
  ! Each thread calls work
  call work(N, A, iam, np)
  !$omp end parallel
end

subroutine work(n, A, iam, np)
  integer n, iam, n
  real A(n,n)
  integer :: chunk,low,high,i,j
  chunk = (n + np - 1)/np
  low = 1 + iam*chunk
  high=min(n,(iam+1)*chunk)
  do j = low, high
    do I=1,n
      A(I,j)=3.14 + &
      sqrt(real(i*i*i+j*j+i*j*j))
    enddo
  enddo
return
Pros and Cons of SPMD

» Potentially higher parallel fraction than with loop parallelism

» The fewer parallel regions, the less overhead

» More explicit synchronization needed than for loop parallelization

» Does not promote incremental parallelization and requires manually assigning data subsets to threads
Extent of directives

Most directives have as extent a structured block, or basic block, i.e., a sequence of statements with a flow of control that satisfies:

- there is only one entry point in the block, at the beginning of the block
- there is only one exit point, at the end of the block; the exceptions are that exit() in C and stop in Fortran are allowed
Scheduling

• Scheduling assigns the iterations of a parallel loop to the team of threads

• The directives `[parallel] do` and `[parallel] for` take the clause
  
  `schedule(type [,chunk])`

• The optional `chunk` is a loop-invariant positive integer specifying the number of contiguous iterations assigned to a thread
Scheduling

The type can be one of:

- **static** threads are statically assigned chunks of size $\text{chunk}$ in a round-robin fashion. The default for $\text{chunk}$ is $\text{ceiling}(N/p)$ where $N$ is the number of iterations and $p$ is the number of processors.

- **dynamic** threads are dynamically assigned chunks of size $\text{chunk}$, i.e.,
Scheduling

when a thread is ready to receive new work, it is assigned the next pending chunk. Default value for chunk is 1.

• guided a variant of dynamic scheduling in which the size of the chunk decreases exponentially from chunk to 1. Default value for chunk is $\text{ceiling}(N/p)$
Scheduling

- runtime indicates that the schedule type and chunk are specified by the environment variable `OMP_SCHEDULE`. A chunk cannot be specified with runtime.

- Example of run-time specified scheduling
  
  ```
  setenv OMP_SCHEDULE "dynamic,2"
  ```
Scheduling

• **If the `schedule` clause is missing, an implementation dependent schedule is selected.** MIPSpro selects by default the static schedule.

• **Static scheduling has low overhead and provides better data locality.**

• **Dynamic and guided scheduling may provide better load balancing.**
Directive Binding

- Work sharing directives (do, for, sections, and single) as well as master and barrier bind to the dynamically closest parallel directive, if one exists, and have no effect when they are not in the dynamic extent of a parallel region.

- The ordered directive binds to the enclosing do or for directive having the ordered clause.
Directive Binding

- critical (and atomic) provide mutual exclusive execution (and update) with respect to all the threads in the program
Data Scoping

- Work-sharing and parallel directives accept *data scoping* clauses.
- Scope clauses apply to the static extent of the directive and to variables passed as actual arguments.
- The *shared* clause applied to a variable means that all threads will access the single copy of that variable created in the master thread.
Data Scoping

• The **private** clause applied to a variable means that a volatile copy of the variable is cloned for each thread.

• Semi-private data for parallel loops:
  
  – *reduction*: variable that is the target of a reduction operation performed by the loop, e.g., sum.
  
  – *firstprivate*: initialize the private copy from the value of the shared variable.
  
  – *lastprivate*: upon loop exit, master thread holds the value seen by the thread assigned the last loop iteration (for parallel loops only).
Threadprivate Data

• The `threadprivate` directive is associated with the declaration of a static variable (C) or common block (Fortran) and specifies `persistent` data (spans parallel regions) cloned, but not initialized, for each thread.

• To guarantee persistence, the dynamic threads feature must be disabled:
  ```
  setenv OMP_DYNAMIC FALSE
  ```
Threadprivate Data

- threadprivate data can be initialized in a thread using the `copyin` clause associated with the parallel, parallel do/for, and parallel sections directives.
- The value stored in the master thread is copied into each thread in the team.
- Syntax: `copyin (name [,name])`
  
  where `name` is a variable (C) or a named common block (Fortran).
Scoping Rules

• Data declared outside a parallel region region is shared by default, except for
  – loop index variable of `parallel do`
  – data declared as `threadprivate`

• Local data in the dynamic extent of a parallel region is private:
  – subroutine local variables, and
  – C/C++ blocks within a parallel region
Scoping Restrictions

• The private clause for a directive in the dynamic extent of a parallel region can be specified only for variables that are shared in the enclosing parallel region
  – That is, a privatized variable cannot be privatized again
• The shared clause is not allowed for the DO (Fortran) or for (C) directive
Shared Data

• Access to shared data must be mutually exclusive: a thread at a time

• For shared arrays, when different threads access mutually exclusive subscripts, synchronization is not needed

• For shared scalars, critical sections or atomic updates must be used

• Consistency operation: flush directive
Synchronization

Explicit, via directives:

• **critical**, implements the critical sections, providing mutual exclusion

• **atomic**, implements atomic update of a shared variable

• **barrier**, a thread waits at the point where the directive is placed until all other threads reach the barrier
  
  – Cannot appear in the extent of a parallel loop
Synchronization

- **ordered**, preserves the order of the sequential execution; can occur at most once inside a parallel loop
- **flush**, creates consistent view of thread-visible data
- **master**, block in a parallel region that is executed by the master thread and skipped by the other threads; unlike **single**, there is no implied barrier
Implicit Synchronization

• There is an implied barrier at the end of a parallel region, and of a work-sharing construct for which a nowait clause is not specified.

• A flush is implied by an explicit or implicit barrier as well as upon entry and exit of a critical or ordered block.
Directive Nesting

- A parallel directive can appear in the dynamic extent of another parallel, i.e., parallel regions can be nested.
- Work-sharing directives binding to the same parallel directive cannot be nested.
- An ordered directive cannot appear in the dynamic extent of a critical directive.
Directive Nesting

- A barrier or master directive cannot appear in the dynamic extent of a work-sharing region (DO or for, sections, and single) or ordered block.
- In addition, a barrier directive cannot appear in the dynamic extent of a critical or master block.
## Environment Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_NUM_THREADS</td>
<td>positive number</td>
</tr>
<tr>
<td>OMP_DYNAMIC</td>
<td>TRUE or FALSE</td>
</tr>
<tr>
<td>OMP_NESTED</td>
<td>TRUE or FALSE</td>
</tr>
<tr>
<td>OMP_SCHEDULE</td>
<td>“static,2”</td>
</tr>
</tbody>
</table>

- Online help: `man openmp`
Library Routines

OpenMP defines library routines that can be divided in three categories

1. *Query and set multithreading*
   - get/set number of threads or processors
     - `omp_set_num_threads`, `omp_get_num_threads`, `omp_in_parallel`, ...
   - get thread ID:
     - `omp_get_thread_num`


Library Routines

2. Set and get execution environment

- Inquire/set nested parallelism:
  
  `omp_get_nested`
  
  `omp_set_nested`

- Inquire/set dynamic number of threads in different parallel regions:

  `omp_set_dynamic`
  
  `omp_get_dynamic`
3. API for manipulating locks

- A lock variable provides thread synchronization, has C type `omp_lock_t` and Fortran type `integer*8`, and holds a 64-bit address.
- Locking routines: `omp_init_lock`, `omp_set_lock`, `omp_unset_lock`...

Man pages: `omp_threads`, `omp_lock`
Reality Check

Irregular and ambiguous aspects are sources of language- and implementation dependent behavior:

- `nowait` clause is allowed at the beginning of `[parallel] for` (C/C++) but at the end of `[parallel] do` (Fortran)

- `default` clause can specify `private` scope in Fortran, but not in C/C++
Reality Check

• Can only privatize full objects, not array elements, or fields of data structures

• For a threadprivate variable or block one cannot specify any clause except for the copyin clause

• In MIPSpro 7.3.1 one cannot specify in the same directive both the firstprivate and lastprivate clauses for a variable
Reality Check

• With MIPSpro 7.3.1, when a loop is parallelized with the `do` (Fortran) or `for` (C/C++) directive, the indexes of the nested loops are, by default, private in Fortran, but shared in C/C++

  *Probably, this is a compiler issue*

• Fortunately, the compiler warns about unsynchronized accesses to shared variables

• This does not occur for parallel `do` or parallel `for`
Compiling and Running

• Use MIPSpro with the option `-mp` both for compiling and linking

⚠️ default `-MP:open_mp=ON` must be in effect

• Fortran:

  ```shell
  f90 [-freeform] [-cpp]-mp prog.f
  -freeform needed for free form source
  -cpp needed when using `#ifdef`s
  ```

• C/C++:

  ```shell
  cc -mp -O3 prog.c
  CC -mp -O3 prog.C
  ```
Setting the Number of Threads

• Environment variables:
  ```
  setenv OMP_NUM_THREADS 8
  ```

  !

  if `OMP_NUM_THREADS` is not set, but `MP_SET_NUM_THREADS` is set, the latter defines the number of threads

• Environment variables can be overridden by the programmer:
  ```
  omp_set_num_threads(int n)
  ```
Scalable Speedup

• Most often the memory is the limit to the performance of a shared memory program
• On scalable architectures, the latency and bandwidth of memory accesses depend on the locality of accesses
• In achieving good speedup of a shared memory program, data locality is an essential element
What Determines Data Locality

• Initial data distribution determines on which node the memory is placed
  – first touch or round-robin system policies
  – data distribution directives
  – explicit page placement

• Work sharing, e.g., loop scheduling, determines which thread accesses which data

• Cache friendliness determines how often main memory is accessed
Parallelizing Code 1

• Optimize single-CPU performance
  – maximize cache reuse
  – eliminate cache misses
  – compile flags:  -LNO:fusion=2:cache_size2=8m
    -OPT:IEEE_arithmetic=3  -Ofast=ip35

• Parallelize as high a fraction of the work as possible
  – preserve cache friendliness
Parallelizing Code 2

– avoid synchronization and scheduling overhead: partition in few parallel regions, avoid reduction, single and critical sections, make the code loop fusion friendly, use static scheduling

– partition work to achieve load balancing

• **Check correctness of parallel code**

  – run OpenMP compiled code first on one thread, then on several threads
Synchronization Overhead

- Parallel regions, work-sharing, and synchronization incur overhead
- Edinburgh OpenMP Microbenchmarks, version 1.0, by J. Mark Bull, are used to measure the cost of synchronization on a 32 processor Origin 2000, with 300 MHz R12000 processors, and compiling the benchmarks with MIPSpro Fortran 90 compiler, version 7.3.1.1m
Synchronization Overhead

Synchronization Overhead on SGI Origin2000

- barrier
- parallel
- parallel do
- do
- parallel + reduction
- single

Log$_{10}$ (time in cycles)

Number of Threads

32 R12000 300 MHz CPUs
MIPSpro f90 compiler, version 7.3.1.1m
Synchronization Overhead on SGI Origin2000

32 R12000 300 MHz CPUs
MIPSpro f90 compiler, version 7.3.1.1m

- **barrier**
- **atomic**
- **critical**
- **lock/unlock**

Time in cycles vs. Number of Threads
Insights

• cost (DO) \sim cost(barrier)
• cost (parallel DO) \sim 2 \times cost(barrier)
• cost (parallel) > cost (parallel DO)
• atomic is less expensive than critical
• bad scalability for
  – reduction
  – mutual exclusion: critical, (un)lock
  – single
Loop Parallelization

• Identify the loops that are bottleneck to performance

• Parallelize the loops, and ensure that
  – no data races are created
  – cache friendliness is preserved
  – page locality is achieved
  – synchronization and scheduling overheads are minimized
Hurdles to Loop Parallelization

- Data dependencies among iterations caused by shared variables
- Input/Output operations inside the loop
- Calls to thread-unsafe code, e.g., the intrinsic function \texttt{rtc}
- Branches out of the loop
- Insufficient work in the loop body
- The MIPSpro auto-parallelizer helps in identifying these hurdles
Data Races

• Parallelizing a loop with data dependencies causes *data races*: unordered or interfering accesses by multiple threads to shared variables, which make the values of these variables different from the values assumed in a serial execution.

• A program with data races produces unpredictable results, which depend on thread scheduling and speed.
Types of Data Dependencies

- Reduction operations:

```c
const int n = 4096;
int a[n], i, sum = 0;
for (i = 0; i < n; i++) {
    sum += a[i];
}
```

- Easy to parallelize using reduction variables
Types of Data Dependencies

• Carried dependence on a shared array, e.g., recurrence:

```c
const int n = 4096;
int a[n], i;
for (i = 0; i < n-1; i++) {
    a[i] = a[i+1];
}
```

• Non-trivial to eliminate, the auto-parallelizer cannot do it
Parallelizing the Recurrence

Idea: Segregate even and odd indices

```c
#define N 16384
int a[N], work[N+1];

// Save border element
work[N] = a[0];

// Save & shift even indices
#pragma omp parallel for
for (i = 2; i < N; i+=2)
    work[i-1] = a[i];

// Update even indices from odd
#pragma omp parallel for
for (i = 0; i < N-1; i+=2)
    a[i] = a[i+1];

// Update odd indices with even
#pragma omp parallel for
for (i = 1; i < N-1; i+=2)
    a[i] = work[i];

// Set border element
a[N-1] = work[N];
```
Performing Reduction

The bad scalability of the reduction clause affects its usefulness, e.g., bad speedup when summing the elements of a matrix:

```c
#define N 1<<12
#define M 16
int i, j;
double a[N][M], sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        sum += a[i][j];
```
Parallelizing the Sum

Idea: Use explicit partial sums and combine them atomically

```c
#define N 1<<12
#define M 16

int main() {
    double a[N][M], sum = 0.0;
    #pragma distribute a[block][*]
    int i, j = 0;

    #pragma omp parallel private(i,j)
    {
        double mysum = 0.0;
        // initialization of a
        // not shown
        // compute partial sum
        #pragma omp for nowait
        for (i = 0; i < N; i++)
            for (j = 0; j < M; i++)
                mysum += a[i][j];
    }

    // each thread adds its
    // partial sum
    #pragma omp atomic
    sum += mysum;
}
```
Sum and Product Speedup

Summing and multiplying the elements of a matrix

- reduction
- atomic
- ideal speedup

4096 x 16 matrix
Origin2000, 32 R12000 300 MHz CPUs
MIPSpro f90 compiler, version 7.3.1.1m
Loop Fusion

• Increases the work in the loop body
• Better serial programs: fusion promotes software pipelining and reduces the frequency of branches
• Better OpenMP programs: fusion reduces synchronization and scheduling overhead
  – fewer parallel regions and work-sharing constructs
Promoting Loop Fusion

• Loop fusion inhibited by statements between loops which may have dependencies with data accessed by the loops
• Promote fusion: reorder the code to get loops which are not separated by statements creating data dependencies
• Use one parallel do construct for several adjacent loops; may leave it to the compiler to actually perform fusion
Fusion-friendly code

Unfriendly

integer, parameter :: n = 4096
real :: sum, a(n)
do i = 1, n
  a(i) = sqrt(dble(i*i + 1))
enddo
sum = 0.d0
do i = 1, n
  sum = sum + a(i)
enddo

Friendly

integer, parameter :: n = 4096
real :: sum, a(n)
sum = 0.d0
do i = 1, n
  a(i) = sqrt(dble(i*i + 1))
enddo
sum = 0.d0
do i = 1, n
  sum = sum + a(i)
enddo
Tradeoffs in Parallelization

• To increase parallel fraction of work when parallelizing loops, it is best to parallelize the outermost loop of a nested loop.

• However, doing so may require loop transformations such as loop interchanges, which can destroy cache friendliness, e.g., defeat cache blocking.
Tradeoffs in Parallelization

- Static loop scheduling in large chunks per thread promotes cache and page locality but may not achieve load balancing.
- Dynamic and interleaved scheduling achieve good load balancing but cause poor locality of data references.
The Future of OpenMP

• Data placement directives will become part of OpenMP
  – affinity scheduling may be a useful feature
• It is desirable to add parallel input/output to OpenMP
• Java binding of OpenMP
References

• Introduction to OpenMP
  Lawrence Livermore National Laboratory
  www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html
  Ohio Supercomputing Center
  oscinfo.osc.edu/training/openmp/big
  Minnesota Supercomputing Institute
  www.msi.umn.edu/tutorials/shared_tutorials/openMP
References

• **SC2000 Tutorial**
  Mattson and Eigenmann Tutorial
  dynamo.ecn.purdue.edu/~eigenman/EE563/Handouts/sc00introOMP.ppt

• **SC2000 Advanced OpenMP**
  Mattson and Eigenmann Presentation
  dynamo.ecn.purdue.edu/~eigenman/EE563/Handouts/sc00advancedOMP.ppt

• **OpenMP Benchmarks**
  Edinburgh OpenMP Microbenchmarks
  www.epcc.ed.ac.uk/research/openmpbench
Acknowledgements

Special thanks to Gabriel Mateescu from National Research Council Canada.