OpenMP: An API for Portable Shared Memory Programming

Alfred Park

February 26, 2003

OpenMP: What is it?

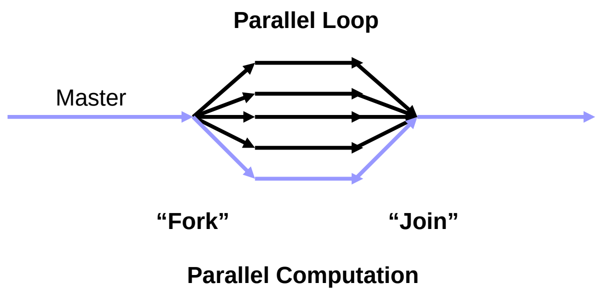
- A standard developed under the review of many major software and hardware developers, government, and academia
- Facilitates simple development of programs to take advantage of SMP architectures
  - SMP: Symmetric multi-processing, access time to memory is approx. equal for all processors (usually 2-16 processors)
  - Shared Memory: memory local to all processors in an SMP domain
  - Distributed Memory: remote memory access (non-local memory) – NUMA (clusters, grids)

OpenMP: What is it?

- OpenMP API is comprised of:
  - Compiler directives
  - Library routines
  - Environment variables
- OpenMP language support:
  - Fortran, C, C++
- Compilers supporting OpenMP:
  - Intel Compilers, Portland Group (PGI), IBM, Compaq
  - Omni, OdinMP can be used with gcc

Fork-Join Parallelism

- Master-Worker Team thread pattern



Behind the Scene

- Thread communication through shared variables (shared memory)
- Threads can be “carried through” from one parallel “region” to the next
  - Important! Need to amortize thread fork cost and minimize thread joins
- Number of threads can be dynamically altered during runtime
- Support for nested parallelism exists in some compilers

Syntax

- Compiler directives:
  - C/C++
    - `#pragma omp` *`directive [clause, …]`*
  - Fortran
    - `!$OMP` *`directive [clause, …]`*
    - `C$OMP` *`directive [clause, …]`*
    - `*$OMP` *`directive [clause, …]`*
- We will focus on C syntax

Parallel Regions

- Fundamental OpenMP construct:
  - `#pragma omp parallel`

```
#pragma omp parallel
{
  printf( “hello world from thread %d of
     %d\n”, omp_get_thread_num(),
     omp_get_num_threads() );
}
```

Sample Output

- From an 8-processor machine:

```
hello world from thread 0 of 8
hello world from thread 2 of 8
hello world from thread 3 of 8
hello world from thread 7 of 8
hello world from thread 6 of 8
hello world from thread 1 of 8
hello world from thread 4 of 8
hello world from thread 5 of 8
```

Another Example

```
double xyz[5000][3];

printf( “entering parallel region\n” );      ← Master only
#pragma omp parallel                         ← Thread Forks
{                                            ← Thread Private
                                               Space
   int tid;
   tid = omp_get_thread_num();
   compute_edges( tid, xyz );
}                                            ← Implicit barrier,
printf( “parallel computation completed\n” );  Thread join
                                             ← Master only
```

Note: xyz is shared between all threads!

Work-sharing Constructs

- `#pragma omp for`
  - Each thread receives a portion of work to accomplish – data parallelism
- `#pragma omp section`
  - Each section executed by a different thread – functional parallelism
- `#pragma omp single`
  - Serialize a section of code, only one thread executes code block (good for I/O)

Data Parallelism Example

```
int a[10000], b[10000], c[10000];

#pragma omp parallel
#pragma omp for
  for (i = 0; i < 10000; i++) {
     a[i] = b[i] + c[i];
  }
```

No specified schedule, each thread gets a chunk of the for loop to process

Implicit barrier at the end of the for loop, can be disabled with the `nowait` clause

Work-sharing Scheduling

- `schedule(static [,chunk])`
  - Threads get a chunk of data to iterate over
- `schedule(dynamic [,chunk])`
  - Threads grab chunk iterations off work queue until all work is exhausted
- `schedule(guided [,chunk])`
  - Threads grab large chunk sizes and decreases to specified chunk size as the computation progresses
- `schedule(runtime)`
  - Use the schedule defined at runtime by the OMP_SCHEDULE environment variable

Functional Parallelism

```
#pragma omp parallel
#pragma omp sections nowait
{
   thread1_work();
#pragma omp section
   thread2_work();
#pragma omp section
   thread3_work();
#pragma omp section
   for (i = 0; i < 10000; i++) {
      quick_transform(xyz);
   }
}
```

Probably a good idea to equally distribute work between sections!

Combining Work-sharing Constructs

```
int a[10000], b[10000], c[10000];

#pragma omp parallel for
  for (i = 0; i < 10000; i++) {
     a[i] = b[i] + c[i];
  }
```

Good for single parallel loops or nested loops

Can combine `parallel` with `sections` as well

If we had multiple for loops and did the above directive for each one, we would have a non-optimal solution. Why?

Data Scope and Protection

- Shared memory programming
  - OpenMP usually defaults to shared data
- Variables declared outside of parallel regions are implicitly carried into threads as shared by default
- Variables declared within parallel regions are private by default
- Functions called within a parallel region or section have their own private stack space

Data Scope Storage Attributes

- `private(var, …)`
  - Uninitialized, thread local instance of the variable
- `shared`
  - Explicitly share variables across all threads
- `firstprivate`
  - Initialize local instance of the variable from master thread
- `lastprivate`
  - Upon the end of the last iteration, value of the variable is copied back out to the master thread

Data Scope Storage Attributes

- `threadprivate`
  - Global data (local file scope in C/C++ or common blocks in Fortran) is private to each thread and persistent throughout lifetime of program
- `default`
  - For the corresponding parallel directive, variables will be default to either the specified `private` or `shared` scope
- `copyin`
  - Initialize value of threadprivate variables to the value reported by the master thread

Example

```
int x;

x = 0;                                           ← Initialize x to zero
#pragma omp parallel for firstprivate(x)         ← Copy value of x from master
for (i = 0; i < 10000; i++) {
  x = x + i;
}
printf( “x is %d\n”, x );                        ← Print out value of x
```

Oops! The value x is undefined!

Need `lastprivate(x)` to copy value back out to master

Global Reduction

- It is often necessary to accumulate (or perform some other operation) on a single variable for all threads and return a single value at the end of the computation
- OpenMP provides a reduction directive
  - `reduction(op: list)`
    - op must not be overloaded
    - op can be +, *, -, /, &, ^, |, &&, ||
      - Binary bitwise operations allowed as well

Synchronization

- As with any parallel programming interface, there is always potential for:
  - Deadlocks
  - Race conditions
- OpenMP provides synchronization directives

Synchronization Constructs

- `critical`
  - Creates critical section, only one thread can enter at a time
- `atomic`
  - Special version of critical, for atomic ops (e.g. updating a single memory location)
- `barrier`
  - Synchronization point for all threads in parallel region
- `ordered`
  - Forces sequential execution of the following block

Synchronization Constructs

- `flush`
  - Synchronization point forcing program to provide a consistent view of memory
- `single`
  - Mentioned in work-sharing construct, not a real synchronization construct
- `master`
  - Not really a synchronization construct – only the master thread executes code block, all other threads skip it (no implied barriers or flushes)

Environment Variables

- OMP_NUM_THREADS
  - Sets max number of threads to use
- OMP_SCHEDULE
  - Scheduling algorithm for “parallel for” regions
- OMP_DYNAMIC
  - Dynamic adjustment of threads for parallel regions (TRUE, FALSE)
- OMP_NESTED
  - Enables or disables nested parallelism (TRUE, FALSE)

OpenMP Library Routines

- Always prefixed with `omp_`
- Too many to list here, see references slide for sites with an OpenMP API listing

An OpenMP Example

- Simple Monte-Carlo approximation for the volume of a sphere
  - $x^2 + y^2 + z^2 = 4$; x, y, z >= 0
- Embarrassingly Parallel (EP) class, should achieve good speedup: close to linear with many iterations

Extending OpenMP

- OpenMP can be used in conjunction with distributed memory message passing
- Message Passing Interface (MPI) can be used to manage computations between shared memory machines
  - For example, data sets in different files
    - Each SMP reads their own data file
    - Performs computation on data set, returns an array of reductions
    - MPI could reduce each component of the array from all SMP machines and return a single globally reduced array

References

- OpenMP: http://www.openmp.org/
- Introduction to OpenMP: http://www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html
- SC'99 OpenMP Tutorial: http://www.openmp.org/presentations/index.cgi?sc99_tutorial