Advanced OpenMP Topics

NAS Webinar
September 12, 2012
NASA Advanced Supercomputing Division
Outline

• The *fork-join* model (a refresh)
• Nested parallelism
• OpenMP tasking
  – Task execution model
  – Data scoping
  – Task synchronization
• Performance considerations
• Correctness issues
• Future OpenMP extensions
The **Fork-Join Model**

- Multiple threads are forked at a *parallel* construct
  - The *master* thread is part of the new thread team
- *Worksharing* constructs distribute work in the parallel region
  - *for* or *do, sections, single*
- Synchronization primitives synchronize threads
  - *barrier, critical, locks*
- Threads join at the end of the parallel region and the *master* thread continues
Nested Parallelism

- Parallel regions can be nested inside another
  - Exploiting parallelism at multiple nesting levels since single level may not be enough

```c
#pragma omp parallel for num_threads(2)
for (j=0; j<m; j++) {
    #pragma omp parallel for num_threads(3)
    for (i=0; i<n; i++) {
        c[j][i] = a[j][i] + b[j][i];
    }
}
```

- To enable nested parallel regions
  - `OMP_NESTED=true` or call `omp_set_nested(1)`
  - If not, the inner parallel region will be started with a team of one thread

- To set the number of threads
  - Call `omp_set_num_threads()` or use the `num_threads` clause
  - `OMP_NUM_THREADS=2,3` (OpenMP 3.1)
Nested Parallelism (cont.)

- Issues with nested parallel regions
  - Performance is a concern
    - Overhead from fork and join
    - Issue with data locality and data reuse
    - *Implicit barrier* at the end of each inner parallel region
  - Not all compilers (such as PGI compiler) provide the support

- The **collapse** clause for multiple loops (OpenMP 3.0)
  - Combines closely nested loops into one
  - More efficient than nested parallel regions

```c
#pragma omp parallel for collapse(2)
for (j=0; j<m; j++) {
    for (i=0; i<n; i++) {
        c[j][i] = a[j][i] + b[j][i];
    }
}
```

Combines both i and j loops
Tasking in OpenMP

• Limitation of the fork-join model with worksharing constructs
  – Work units statically determined in worksharing constructs
    • No easy method to dynamically generate work units
  – Lack of support for recursive algorithms
    • For example, no easy way to traverse a tree in parallel

• Tasking model
  – Introduced in OpenMP 3.0
  – Complimentary to the thread-centric model
  – Ability to express parallelism for recursive algorithms, pointer chasing, which are commonly encountered in C/C++
  – Constructs for task generation and task synchronization
  – Concept of task switching
Basic Task Concept

• OpenMP task
  – A code entity including control flow and its data environment, executed by a thread

• Implicit and explicit tasks
  – Implicit tasks generated via the parallel directive
  – Explicit tasks generated via the task directive

• Task synchronization
  – The taskwait directive to wait for all child tasks of the current task
  – Implicit or explicit barriers to wait for all explicit tasks

• Data environment is associated with tasks except for threadprivate storage

• Locks are owned by tasks
  – Set by a task, unset by the same task
Task Execution Model

- Starts with the *master* thread
- Encounters a **parallel** construct
  - Creates a team of threads, *id 0* for the *master* thread
  - Generates implicit tasks, one per thread
  - Threads in the team executes implicit tasks
- Encounters a worksharing construct
  - Distributes work among threads (or implicit tasks)
- Encounters a **task** construct
  - Generates an explicit task
  - Execution of the task could be deferred
- Execution of explicit tasks
  - Threads execute tasks at a *task scheduling point* (such as *task*, *taskwait*, *barrier*)
  - Thread may switch from one task to another task
- At the end of a **parallel** construct
  - All tasks complete their execution
  - Only the *master* thread continues afterwards

```
Task Pool

Task 1
Task 2
Task 3
...
```

```
Parallel construct

T0
T1
T2
T3
```

```
End parallel construct

Schedule and exec tasks

master continues
```

```
   itask exec
   itask exec
   itask exec
   itask exec

master
```

- *implicit tasks cannot be deferred*
- *explicit tasks could be deferred*
Thread versus Task

• Threading model
  – Thread and work (or task) go together
  – No concept of deferred execution

• Tasking model
  – Task generation and task execution are separate
  – There is no direct control on when a task gets executed
  – Thread is an execution engine
  – It is a more dynamic environment

• OpenMP supports both models
Tasking Example: Fibonacci Number

The code builds a binary task tree. Parallelism comes from the execution of tasks on the leaf nodes. 

*But don’t expect any performance from this version!*
Data Sharing in Tasks

• Default sharing attribute rules
  – Shared for implicit tasks
  – For explicit tasks
    • If a variable is determined to be shared in the parallel region, default is shared
    • Otherwise, default is firstprivate (to avoid out-of-scope data access)

• Use data sharing clauses explicitly, in particular if you are not sure
  – shared, private, firstprivate, etc.

Example of pointer chasing

```
node_t *node_head, *p;
int n = 40;
#pragma omp parallel private(p)
{
    #pragma omp master
    {
        p = node_head;
        while (p) {
            #pragma omp task
            process(p,n);
            p = p->next;
        }
    }
    #pragma omp taskwait
}
```

“p” is private and “n” is shared in the parallel region

“p” is firstprivate and “n” is shared for the task
Common Problems in Using OpenMP

• Code is not scaling – possible issues:
  - Overhead of OpenMP constructs
  - Granularity of work units
  - Remote data access and NUMA effect
  - Load imbalance
  - False sharing of cache
  - Poor resource utilization

• Parallel code gives a slightly different result than the serial code
  - Understanding parallel reduction

• Code crashes or gives different results from run to run
  - Stack size limitation
  - Data race
Overhead and Granularity

• Overhead from OpenMP constructs
  – Fork-join of threads
  – Barrier
  – Creation and scheduling of tasks
  – May be measured with the EPCC microbenchmarks

• Not enough granularity in work unit

• Possible solutions
  – Increase work and exploit parallelism at coarser level
  – Merge parallel regions if possible
  – Avoid barrier if possible (e.g., with `nowait` clause)
  – Use `atomic` over `critical` or `reduction`
Reducing Overhead

Example 1

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

#pragma omp parallel for
for (i=0; i<n; i++)
d[i] = e[i] + f[i];
```

Example 2

```c
for (i=0; i<m; i++) {
    #pragma omp parallel for
    for (j=0; j<n; j++)
        a[i][j] += a[i-1][j] + a[i+1][j];
}
```

- Merge parallel regions
- Use `nowait` if no data dependence between worksharing regions

```c
#pragma omp parallel private(i)
for (i=0; i<m; i++) {
    #pragma omp for
    for (j=0; j<n; j++)
        a[i][j] += a[i-1][j] + a[i+1][j];
}
```
Fibonacci Number – Increased Granularity

```c
int fib(int n)
{
    int x, y;
    if (n < 2) return n;
    if (n < 30)
        return (fib(n-1)+fib(n-2));
    #pragma omp task shared(x)
        x = fib(n-1);
    #pragma omp task shared(y)
        y = fib(n-2);
    #pragma omp taskwait
    return(x+y);
}
```

IF condition added to avoid fine-grained tasks and increase task granularity

Each task performs some amount of work!

Performance from the naïve version is not shown here – it is more than 10-fold worse and does not scale.
- Measure extra time spent (or overhead) by each OpenMP construct as a function of thread counts on the SGI Altix
- Intel OpenMP compiler was used
- Constructs such as parallel, reduction, barrier have very large overhead
Remote Data Access and NUMA Effect

- Remote data access is more expensive
  - May cause memory access bottleneck
- Possible solutions
  - Use thread-local data (private or threadprivate) if possible
  - Add the *first touch* loop

- Performance of BT from the NAS Parallel Benchmarks on the SGI Altix
- Four types of data layout based on how data are initially distributed

![Graph showing performance comparison of BT CLASS=B and BT CLASS=C]

Better
Other Performance Issues

• Load imbalance
  – Try the *dynamic* loop schedule
  – Increase iteration space by using the *collapse* clause for nested loops

• False sharing
  – Caused by multiple threads updating data in the same cache line
  – Work-around
    • Pad array dimension of shared data
    • Use private data if possible

• A good practice
  – Use `omp_get_wtime()` to get timing profile for code sections in question
Thread-Processor Binding

• Or thread affinity
  – May improve performance by reducing OS scheduling overhead and improving resource utilization
  – Reduce run-to-run timing variation
  – But no standard way currently to control the affinity setting
    • For Intel compiler, set `KMP_AFFINITY={scatter,compact..}`

Example of using thread binding from two types of affinity settings to improve resource utilization

![Graph showing Packed vs. Scatter (% Change) for different benchmarks with omp threads = 8 and omp threads = 16 on Intel SandyBridge Node (#cores=16).]

Scatter is better
Thread Affinity Types

Examples of Intel Compiler, **OMP_NUM_THREADS=8**, two quad-core sockets

**KMP_AFFINITY=compact**
better cache sharing between threads

**KMP_AFFINITY=scatter**
maximizing memory bandwidth utilization

– “scatter” usually gives better results for most cases

**KMP_AFFINITY=explicit, proclist=[0-7]**
user specifies the proc list explicitly

For more details, see
www.nas.nasa.gov/hecc/support/kb/60/
Code Correctness Issues

• Parallel reduction
  - May not be bit reproducible as the serial result
  - Mathematically associative: \((x + y) + z = x + (y + z)\), but machine accuracy is limited for floating point
  - Use double precision over single precision for reduction variables

• Some common programming errors
  - Incorrect variable scoping
  - Accessing reduced variable without a barrier
  - Master versus single
    • Master doesn’t have a barrier, but single does
  - Race condition
Race Condition

• Commonly encountered in shared memory programming
  - Results are not deterministic
  - Unintentional (programming error), intentional (one thread polling a flag that is updated by another thread)

• Occurs when all the following hold
  - Multiple threads access the same memory location concurrently
  - One of the access is write
  - Access is not protected (e.g., by critical construct)

```
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  n = tid;
}
```

*Updating shared variable “n” from multiple threads causes a race. Race condition should be avoided by all means.*
Code Correctness Issues (cont.)

- Code crashes
  - Caused by programming errors
    - Debugging the code with a debugger (gdb, totalview, etc.)
  - Runtime stack size limitation
    - Default thread stack size can be easily exhausted
    - Reset stack size for master threads via shell command
      limit stacksize unlimit (csh)
      ulimit –s unlimited (sh)
    - Reset stack size for worker threads via environment variable
      setenv OMP_STACKSIZE 12m (csh)
      export OMP_STACKSIZE=12m (sh)
Software Tools

• Correctness checking
  – Variable scoping
    • “Auto” scoping supported by the Oracle OpenMP compiler
  – Race condition detection
    • Intel Thread Checker (or Parallel Inspector)
    • Oracle Thread Analyzer

• Performance tools
  – Compiler feedback
  – Profiling tools
    • ompP (UCB), PerfSuite (NCSA), Vtune (Intel), TAU (U.Oregon), etc.

• Parallelization assistant
  – Compiler auto-parallelization
  – Semi-automatic parallelization tools (CAPO/Parawise)
Future OpenMP Extensions

• Work in progress within the OpenMP language committee
  – Public draft of the 4.0 specification by the end of the year

• New features under consideration
  – User-defined reduction
  – Error handling
    • The `cancel` construct for parallel and worksharing
    • `Cancellation` points
  – Fortran 2003 support
  – Thread affinity
    • Logical processor units via the `OMP_PLACES` environment variable
    • Affinity policy (`compact, scatter, master`) for threads in parallel regions
    • Handling thread affinity in nested parallel regions
  – **Atomic** construct for sequential consistency
    • `atomic seq_cst`
Support for Accelerator Devices

• Such as GPUs, Intel Xeon Phi (MIC)
  – Many cores, large amount of parallelism
  – Disjoint device memory from the host

• Programming models
  – Low level models (CUDA, OpenCL) exist, but hard to use
  – High level models are being developed

• OpenACC model (for GPUs)
  – Based on the PGI Accelerator programming model, defined by multi-vendors (www.openacc-standard.org)
  – Using compiler directives, as in OpenMP
  – Offloading work to the device
  – Data transfer between the host and the device
  – Intend to merge into OpenMP eventually
Summary

• OpenMP provides a programming model for shared memory systems
• Compilers with OpenMP support are widely available
• The tasking model opens up opportunities for a wider range of applications
• Several issues to consider for developing efficient OpenMP codes
  – OpenMP overhead
  – Data locality
  – In some cases trade-off between easy of use and performance
    • With some extra effort, scalability can be achieved in many cases
References

• OpenMP specifications

• Resources
  – www.openmp.org/wp/resources/
  – www.compunity.org/

• Benchmarks
  – OpenMP Microbenchmarks from EPCC
    (www.epcc.ed.ac.uk/research/openmpbench)
  – NAS Parallel Benchmarks
    (www.nas.nasa.gov/publications/npb.html)

• Porting applications to Pleiades
  – www.nas.nasa.gov/hecc/support/kb/52/
  – www.nas.nasa.gov/hecc/support/kb/60/