F21DP2 - Distributed and Parallel Technology

Phil Trinder

Topic 4: Shared Memory (Multicore)
Parallelism: OpenMP

Numerous Tutorials on the web
Chapters of Multicore Programming Textbooks
Wikipedia article

Parts of this lecture are adapted from computing.llnl.gov/tutorials/openMP/

---

**Multicore Architectures**

- Physical limits of semiconductor technology and improved manufacturing technologies make multicores the dominant *Central* processor (CPU) technology
- Of course there are still lots of unicore, e.g. in embedded systems
- The architectures are shared-memory, and cores share some level of cache
- The terminology is (deliberately?) confused: cores (C) may reside on the same
  - (S)ilicon Chip
  - (P)ackage
  - Board, connected by a synchronising bus

---

**Multicore Architectures**

These components are often combined e.g. the '8-core' Dell PowerEdge machines comprise a pair of quad cores in 2 packages.

- Major performance issues are
  - *cache coherence* - ensuring that each core’s caches are consistent after memory writes
  - *contention* - synchronising memory read/writes between cores
- The trend is towards more cores: many cores
Multicore Programming

- The architectures encourages a programming model with lightweight threads (sometimes called filaments)
- Stateful programming (e.g. assignment) causes problems with cache coherence
- **Caution**: much threaded code is not safe for parallel execution!
- There are a number of programming models being explored, e.g.
  - Concurrent Collection libraries, e.g. java.util.concurrent
  - High-level Parallelism, e.g. JaSkel, GpH, Erlang
  - Thread-based models, e.g. Intel Thread Library, OpenMP

What is OpenMP?

- Open Multi-Processing
- An API for multi-threaded, shared memory parallelism
- API components:
  1. Compiler Directives
  2. Runtime Library Routines
  3. Environment Variables
- Available for
  - C, C++ and Fortran(s)
  - Unix & some Windows platforms

OpenMP Does Not

- Guarantee efficient use of shared memory
- Check for data dependencies, data conflicts, race conditions, or deadlocks
- Guarantee that I/O (e.g. to the same file) is synchronous

History

- Developed by a consortium of companies: HP, Intel, IBM, ...
- 1997 1st Standard
- 2008 OpenMP 3.0
1. Compiler Directives

- `#pragma omp` directive [ clause, clause, ... ]
- A directive typically applies to the following structured block
- No. threads determined by:
  - Setting of the NUM_THREADS clause
  - Use of the `omp_set_num_threads()` library function
  - Setting of the OMP_NUM_THREADS environment variable
  - Implementation default - usually the number of cores, though it could be dynamic
  - others ...

```c
#include <omp.h>

main () {
  int var1, var2, var3;
  Serial code

  // Beginning of parallel section. Fork a team of threads and specify variable scoping
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    // Parallel section executed by all threads
    All threads join master thread and disband
  }
  Resume serial code

  // Only master thread does this
  if (tid == 0)
  {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
  }
}
```
Dual Core execution:

jove% helloPar
Hello World from thread = 1
Hello World from thread = 0
Number of threads = 2
jove%

8-core execution:
lxpara3% helloPar
Hello World from thread = 4
Hello World from thread = 3
Hello World from thread = 5
Hello World from thread = 1
Hello World from thread = 6
Hello World from thread = 2
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 7
lxpara3%

Data Parallelism: for Directive

```cpp
#pragma omp for [clause ...]
    schedule (type [,chunk])
    ordered
    collapse
    nowait
    reduction (operator: list)
    ...
```

- **schedule** specifies how loop iterations are divided amongst threads
  - **static** iterations of size chunk are statically assigned
  - **dynamic** iterations of size chunk are assigned, and when a thread finishes it dynamically collects the next chunk
  - **auto** scheduling delegated to compiler/runtim
  - ...
- **nowait** threads don’t synchronise at the end of the loop

```cpp
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main ()
{
    int i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
{
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
} /* end of parallel section */
}```
Control Parallelism: sections Directive

`#pragma omp sections [clause ...]
   reduction (operator: list)
   nowait`

- Create threads to evaluate different control structures in a program

```
include <omp.h>
#define N 1000

main () {
  int i;
  float a[N], b[N], c[N], d[N];

  for (i=0; i < N; i++) { /* Initialise */
    a[i] = i * 1.5;
    b[i] = i + 22.35;
  }

  #pragma omp parallel shared(a,b,c,d) private(i):
  {
    #pragma omp sections nowait
    {
      #pragma omp section
      for (i=0; i < N; i++)
        c[i] = a[i] + b[i];

      #pragma omp section
      for (i=0; i < N; i++)
        d[i] = a[i] * b[i];
    } /* end of sections */
  } /* end of parallel section */
}
```

Synchronisation Constructs

- `#pragma omp critical [ name ]`
  structured_block
  If a thread is executing in a critical region, any other thread reaching the region blocks until the first thread exits

- `#pragma omp barrier`
  A thread reaching a barrier waits until all other threads have reached the barrier. Thereafter all threads resume parallel execution.

- ...  

Data Sharing Clauses

- As OpenMP is a shared memory paradigm variables are shared by default

- However it is important for each thread to keep a private copy of some variables, including loop index variables, thread id, etc.

- See previous programs for examples
2. Run-Time Library Functions

Library functions support

- Querying the number of threads/processors, setting number of threads
- Semaphores
- Portable wall clock timing
- Setting execution environment for functions

Example functions

- Getting and setting number of threads:
  ```c
  void omp_set_num_threads(int num_threads)
  int omp_get_num_threads(void)
  ```

- Get thread Id (see 1st example program):
  ```c
  int omp_get_thread_num(void)
  ```

- Get no. processors:
  ```c
  int omp_get_num_procs(void)
  ```

- Set/Unset locks
  ```c
  void omp_set_lock(omp_lock_t *lock)
  void omp_unset_lock(omp_lock_t *lock)
  ```

- Get wallclock time
  ```c
  double omp_get_wtime(void)
  ```

3. Environment Variables

- OpenMP parallel evaluation can be controlled from the environment, for example:

  - To control parallel for loops:
    ```bash
    export OMP_SCHEDULE="dynamic"
    ```

  - To set the maximum number of threads to use during execution:
    ```bash
    export OMP_NUM_THREADS=6
    ```
    For example:
    ```bash
    lxpara3% export OMP_NUM_THREADS=4
    lxpara3% helloPar
    Hello World from thread = 2
    Hello World from thread = 3
    Hello World from thread = 1
    Hello World from thread = 0
    Number of threads = 4
    lxpara3%
    ```

  - ...

Critique of OpenMP

- Provides high-level parallelism compared with MPI
- Programmer must
  - identify paradigm(s) and introduce them e.g. data parallelism, control parallelism
  - control thread granularity
  - identify shared and private variables
  - synchronise on shared variables
- The directives are:
  - elaborate - how do you chose the best ones?
  - a fixed set: there's no way to introduce a new directive
  - there are only fixed ways of combining directives. In contrast GpH evaluation strategies can be arbitrarily sequenced, composed, or passed as an argument to another strategy.