Lecture 14

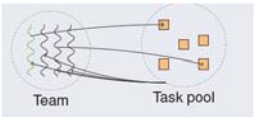
Review

- Structured block
- Parallel construct
  - clauses
- Working-Sharing contructs
  - for, single, section
  - for construct with different scheduling strategies

Work Tasking

Tasking

- New feature in OpenMP 3.0 (200805)
- Used to parallelize very unstructured parallelism Unbounded loops, recursive functions, ...
- Tasks are work units whose execution may be deferred or may be executed immediately (
- Synchronize through #pragma omp taskwait - suspends the current task until all children tasks are completed.
  - Just direct children, not descendants



Compiler directives

task *construct*

#pragma omp task *[clause [[,] clause] …] new-line*
structured-block

Clauses
default(shared|none)
private(list)
firstprivate(list)
shared(list)

if (scalar-expression)
untied – untied task, which can be executed by any thread in the team

•The task construct defines an explicit task.
•The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs

34a.cpp

```
unsigned long fib(unsigned long n) {
  if(n<2) return n;
  unsigned long x, y;
  #pragma omp task shared(x)
  x = fib(n-1);
  #pragma omp task shared(y)
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}
int main() {
  #pragma omp parallel
  {
    #pragma omp single nowait
    cout << fib(35) << " ";
  }
  return 0;
}
```

```
time ./34a.exe
9227465

real    0m11.608s
user    0m17.665s
sys     0m5.352s
```

35a.cpp

```
unsigned long fib(unsigned long n) {
  if(n<2) return n;
  unsigned long x, y;
  #pragma omp task shared(x)
  x = fib(n-1);
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}
int main() {
  #pragma omp parallel
  {
    #pragma omp single nowait
    cout << fib(35) << " ";
  }
  return 0;
}
```

```
time ./35a.exe
9227465

real    0m5.806s
user    0m9.345s
sys     0m2.004s
```

7

Compiler directives
task *construct*, if *clause*

- When the if clause argument is false
  - The encountering task is suspended.
  - The new task is executed immediately by the encountering thread.
  - The data environment is still local to the new task...
  - ...and it's still a different task with respect to synchronization.
  - The parent task resumes when the task finishes.
- It's a user directed optimization
  - when the cost of deferring the task is too great compared to the cost of executing the task code.
  - to control cache and memory affinity.
  - Avoid creating small tasks

36a.cpp

```
unsigned long fib(unsigned long n) {
  if(n<2) return n;
  unsigned long x, y;
  #pragma omp task shared(x) if(0)
  x = fib(n-1);
  y = fib(n-2);

  return x+y;
}
int main() {
  #pragma omp parallel
  {
    #pragma omp single nowait
    cout << fib(35) << " ";
  }
  return 0;
}
```

if (false) makes the encountering thread immediately executes the new task (and suspend the current running task …)

```
time ./36a.exe
9227465

real    0m4.606s
user    0m4.792s
sys     0m0.000s
```

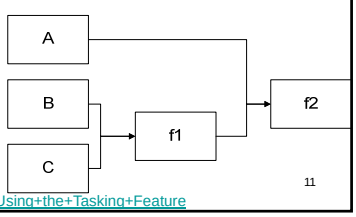
9

Compiler directives
task *construct*, untied *clause*

- By default, tasks are tied to the thread that first executes them
  - not the creator
- Tied tasks can be scheduled as the implementation wishes
  - Constraints:
    - Only the thread that the task is tied to can execute it
    - A task can only be suspended at a suspend point
      - task creation, task finish, taskwait, barrier
    - If the tasks is not suspended in a barrier it can only switch to a direct descendant of all tasks tied to the thread
- Untied tasks
  - Can be suspended at any point
  - Can switch to any task
  - Be careful: variable scoping

10

```
/* Compute f2 (A, f1 (B, C)) */
void foo () {
   int a, b, c, x, y;
   #pragma omp task shared(a)
      a = A();
   #pragma omp task if (0) shared (b, c, x) untied
   {
      #pragma omp task shared(b)
            b = B();
      #pragma omp task shared(c)
            c = C();
      #pragma omp taskwait
   }
   x = f1 (b, c);
   #pragma omp taskwait
   y = f2 (a, x);
}
```



11

http://wikis.sun.com/display/openmp/Using+the+Tasking+Feature

- Library functions
  - Timing
  - Locking
  - Thread information and control
- Environment Variables

Synchronization

- Synchronization refers to cooperate/coordiate multiple threads to work in a desired manner/order. OpenMP provides the following directives for synchronization:
  - **#pragma omp master**
  - **#pragma omp barrier**
  - **#pragma omp critical**
  - **#pragma omp atomic**
  - **#pragma omp flush**
  - **#pragma omp ordered**
- Note synchronization implies threads to be coordinated, and usually results in performance degradation.

13

Compiler directives for sychronization
#pragma omp master

- The **master** construct specifies a structured block that is executed by the master thread (thread number 0).

#pragma omp master *new-line*
structured-block

- Very much similar to **#pragma omp single** previously introduced, but:
  - The structured block is executed by the master thread.
  - There is no implicit barrier at the end of the structured block.

14

18a.cpp
```
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
  int i, n=10, sum=0;
  #pragma omp parallel default(none) \
       shared(cin, cout, n, sum) private(i)
  {
    int ID = omp_get_thread_num();
    #pragma omp master
    {
      cout << "(" << ID << ") Please enter n: ";
      cin >> n;
    }
    #pragma omp barrier
    #pragma omp for reduction(+:sum)
    for(i=0;i<=n;i++) {
      sum += i;
    }
  }
  cout << "Sum: " << sum;
  return 0;
}
```

15

Compiler directives for sychronization
#pragma omp barrier

- The **barrier** construct specifies an explicit barrier at the point at which the construct appears.

#pragma omp barrier *new-line*

- **Barrier**: *A point in the execution of a program encountered by a team, beyond which no thread in the team may execute until all threads in the team have reached that point.*
- Recall: MPI_Barrier().
- The following constructs have implicit barrier at the end of the structured block: **parallel, for, sections, *critical, single***

16

Compiler directives for sychronization
#pragma omp critical

- The **critical** construct restricts execution of the associated structured block to a single thread at a time.

#pragma omp critical *[(name)] new-line*
structured-block

- *In concurrent programming, a **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.*
- Different threads may enter critical sections of different names.
- Often used to update shared variables, or to call **thread-unsafe** functions (e.g. rand(), file I/O, …).
  - Thread unsafe functions are usually those who keep their own "states" (e.g. static variables, rand())

17

04a.cpp
```
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel shared(count)
  {
    for(int i=0;i<1000000;i++) {
      count++;
    }
    cout << "My count: " << count << endl;
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1089747
My count: 2126479
My count: 2240935
My count: 3165806
My count: 3345987
My count: 3364074
My count: 4210268
My count: 4425132
Main count: 4425132
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1152417
My count: 2249624
My count: 2274868
My count: 3346454
My count: 3858762
My count: 4402852
My count: 4406426
My count: 4465075
Main count: 4465075
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: My count:
10919331091933
My count: 2170209

My count: 3213347
My count: 3751776
My count: 4269572
My count: 4272164
My count: 4347902
Main count: 4347902
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1501215
My count: 2501215
My count: 3501215
My count: 4501215
My count: 5501215
My count: 6501215
My count: 7501215
My count: 1000017
Main count: 1000017
```

```
OMP_NUM_THREADS=8 ./04a.exe
My count: 1000017
My count: 2000017
My count: 3000017
My count: 4000017
My count: 5000017
My count: 6000017
My count: 7000017
My count: 8000017
Main count: 8000017
```

18

19a.cpp

```
#include <iostream>
using namespace std;

int main() {
  int count=17;
  #pragma omp parallel shared(count)
  {
    for(int i=0;i<1000000;i++) {
      #pragma omp critical
      count++;
    }
  }
  cout << "Main count: " << count << endl;
  return 0;
}
```

```
OMP_NUM_THREADS=8 ./19a.exe
Main count: 8000017
```

```
OMP_NUM_THREADS=8 ./19a.exe
Main count: 8000017
```

```
OMP_NUM_THREADS=8 ./19a.exe
Main count: 8000017
```

```
OMP_NUM_THREADS=8 ./19a.exe
Main count: 8000017
```

NOTE: Critical section induces a great synchronization between threads, and may significantly reduce performance due to this synchronization!

Compiler directives for sychronization
#pragma omp atomic

- The atomic construct ensures that a specific storage location is updated **atomically**, rather than exposing it to the possibility of multiple, simultaneous writing threads.

  #pragma omp atomic *new-line*
  expression-statement

- This is a special case of a critical section that can only be used for certain simple statements: =, ++, --, +=, -=, *=, /=, &=, ^=, …
- And it is more efficient …

20

Compiler directives for sychronization
#pragma omp flush

- In OpenMP, threads have temporary view of memory (in cache or in register). This temp. view of memory may be different from the actual content in the main memory.
- The flush construct enforces consistency between the temporary view and memory. (i.e. flush variables from cache to main memory, and "invalid" cached variable so that the next the variable has to be read from the main memory)

  #pragma omp flush (variable list) *new-line*
- If the list is ignored, all thread-local variables are flushed. This is implied at the following regions: barrier region; entry to/exit from parallel, critical, and ordered; exit from work-sharing; lock API function calls; before & after task scheduling point.

21

Synchronization

- Synchronization refers to cooperate/coordiate multiple threads to work in a desired manner/order. OpenMP provides the following directives for synchronization:
  - **#pragma omp master**
  - **#pragma omp barrier**
  - **#pragma omp critical**
  - **#pragma omp atomic**
  - **#pragma omp flush**
  - **#pragma omp ordered**
- Note synchronization implies threads to be coordinated, and usually results in performance degradation.

22

Guess what is the outcome?

```
#include <iostream>
#include <omp.h>

using namespace std;
#define NUM_THREADS 16
int synch[NUM_THREADS];
int output[NUM_THREADS];


int main() {
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    int id = omp_get_thread_num();
    int next = (id>0? id: NUM_THREADS) - 1;

    #pragma omp for
    for(int i=0;i<NUM_THREADS;i++) {
      synch[i] = 1;
    }
    int status = synch[next];

    synch[id] = 5;
    output[id] = status * 10 + synch[next];
  }

  for(int i=0;i<NUM_THREADS;i++) cout << output[i] << " ";
  return 0;
}
```

23

Guess what is the outcome?

```
#include <iostream>
#include <omp.h>

using namespace std;
#define NUM_THREADS 16
int synch[NUM_THREADS];
int output[NUM_THREADS];


int main() {
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    int id = omp_get_thread_num();
    int next = (id>0? id: NUM_THREADS) - 1;

    #pragma omp for
    for(int i=0;i<NUM_THREADS;i++) {
      synch[i] = 1;
    }
    #pragma omp barrier
    int status = synch[next];

    synch[id] = 5;
    output[id] = status * 10 + synch[next];
  }
  for(int i=0;i<NUM_THREADS;i++) cout << output[i] << " ";
  return 0;
}
```

24

Guess what is the outcome?

```
#include <iostream>
#include <omp.h>

using namespace std;
#define NUM_THREADS 16
int synch[NUM_THREADS];
int output[NUM_THREADS];


int main() {
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    int id = omp_get_thread_num();
    int next = (id>0? id: NUM_THREADS) - 1;

    #pragma omp for
    for(int i=0;i<NUM_THREADS;i++) {
      synch[i] = 1;
    }
    int status = synch[next];
    #pragma omp barrier

    synch[id] = 5;
    output[id] = status * 10 + synch[next];
  }
  for(int i=0;i<NUM_THREADS;i++) cout << output[i] << " ";
  return 0;
}
```

25

Guess what is the outcome?

```
#include <iostream>
#include <omp.h>

using namespace std;
#define NUM_THREADS 16
int synch[NUM_THREADS];
int output[NUM_THREADS];


int main() {
  #pragma omp parallel num_threads(NUM_THREADS)
  {
    int id = omp_get_thread_num();
    int next = (id>0? id: NUM_THREADS) - 1;

    #pragma omp for
    for(int i=0;i<NUM_THREADS;i++) {
      synch[i] = 1;
    }
    int status = synch[next];
    #pragma omp barrier

    synch[id] = 5;
    output[id] = status * 10 + synch[next];
  }
  for(int i=0;i<NUM_THREADS;i++) cout << output[i] << " ";
  return 0;
}
```

26

OpenMP

Compiler directives

Library functions

Environment variables

OpenMP Library Functions

- #include <omp.h>
- Once you use OpenMP library functions, the code can no longer be compiled on compilers that doesn't support OpenMP.
- Three categories
  - Timing
    - omp_get_wtime(): get a wallclock time reference
    - omp_get_wtick(): get the resolution of wallclock time.
  - Thread information and control
  - Locking

Threading

omp_get_num_procs(): get the number of available processors

omp_get_num_threads(): get the number of threads currently using in parallel execution

omp_get_max_threads(): get the max. number of threads that can be used for parallel execution

omp_get_thread_num(): get the identifier (~ MPI_Comm_rank()) of the current thread.

omp_in_parallel(): returns true if the code is in a parallel region

omp_set_num_threads(): suggest the number of threads to be used in parallel execution.

Locking

- Locks are used for synchronization
- OpenMP Locks are represented by "**lock variables**", which can only be accessed through OpenMP locking functions.
- An OpenMP lock has one of the following three states: uninitialized, unlocked, or locked.

Locking

omp_init_lock(): initializes a simple lock.

omp_destroy_lock(): uninitializes a simple lock.

omp_set_lock(): **waits** until a simple lock is available, and then sets it.

omp_test_lock(): **tests** a simple lock, and **sets** it if it is available.

omp_unset_lock(): unsets a simple lock.

20a.cpp

```
#include <iostream>
using namespace std;
#include "omp.h"
void doSomething() {
  cout << "Executed by many threads at the same time." << endl;}
void alone() {
  cout << "Executed by exactly one thread at any time." << endl;}

int main() {
  int id;
  omp_lock_t Lock;
  omp_init_lock(&Lock);
  #pragma omp parallel
  {
    id = omp_get_thread_num();
    omp_set_lock(&Lock);
    cout << "Thread ID " << id << " entered! " << endl;
    omp_unset_lock(&Lock);
    while(! omp_test_lock(&Lock)) {
        doSomething();
    }
    alone();
    omp_unset_lock(&Lock);
  }
  return 0;
}
```

32

OpenMP

Compiler directives

Library functions

Environment variables

Environment Variables

- **OMP_SCHEDULE**
  - **#pragma omp for schedule(runtime)**
  - Linux, bash: export OMP_SCHEDLE='dynamic, 3072'
  - Windows: set OMP_SCHEDULE=dynamic, 3072
- **OMP_NUM_THREADS**
  - *Suggests* the number of threads for parallel execution

OpenMP vs. MPI

Only for shared memory computers (?)
• Easy to incrementally parallelize
– More difficult to write highly scalable programs
• Small API based on compiler directives and limited library routines
• Same program can be used for sequential and parallel execution
• Shared vs private variables can cause confusion

Portable to all platforms
• Parallelize all or nothing
• Vast collection of library routines
• Possible but difficult to use same program for serial and parallel execution
• variables are local to each processor

Performance Considerations

- Converge and Granularity
- Load balance
- Locality
- Synchronization

36

References

- T. Mattson and R. Eigenmann (1999), "OpenMP: An API for Writing Portable SMP Application Software", SC'99 Tutorial
- Ruud van der Pas (2005), "An Introduction Into OpenMP," IWOMP 2005
- OpenMP Application Program Interface, Version 2.5 May 2005
- OpenMP Application Program Interface, Version 3.0, May 2008

Other Online References

- Lawrence Livemore online tutorial
  - http://www.llnl.gov/computing/training/
- Multimedia tutorial at Boston University:
  - http://scv.bu.edu/SCV/Tutorials/OpenMP/
- European workshop on OpenMP (EWOMP)
  - http://www.epcc.ed.ac.uk/ewomp2000/
- Wikipedia
  - http://en.wikipedia.org/wiki/OpenMP

Notice

- 6/7 will be the last lecture…
- Can we have a discussion on assignments on 6/14? (no class is scheduled from school's point of view!)
- Final exam on 6/21
  - Written part (open book)
  - Programming (take home or ??)
- Presentation on 6/28

39