High-level Parallel Programming: OpenMP

Contents of Lecture 8

- OpenMP

Parallel Execution of Software

- Ideally optimizing compilers would be able to parallelize existing C/C++ and FORTRAN codes.
- From our example of the dataflow program (and other programs), I think it's rather difficult to write such a compiler.
- Instead of writing new sequential programs we can use e.g.
  - C/Pthreads, C11 Threads, Java Threads
  - Scala Actors
- What about all existing codes?

OpenMP for C/C++ and FORTRAN

- Another option is tool support for manual parallelization:
  - Programmer annotates the source code and is guarantees the validity of parallelization of a loop.
  - Tool support: generating parallel code for a loop
- GCC supports the OpenMP standard for this approach.
- Include `<omp.h>` and annotate e.g. as:

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < n; ++i) {
        /* ... */
}
```

- Compile with `gcc -fopenmp`

The Main Advantage with OpenMP

- We don't want to rewrite millions of C/C++ and FORTRAN codes from scratch.
- Using a new and relatively untested language may be a big risk.
- Untested = less than 20 years of experience...
- With OpenMP we can parallelize our applications **incrementally**.
- We can focus on one for-loop at a time.

Origin of OpenMP

- All supercomputer companies had their own compiler directives to support this ”semi-automatic” parallelization.
- When SGI and Cray (one of the three Cray companies) merged they needed to define a common set of compiler directives.
- Kuck and Associates, a compiler company, and the U.S. ASCI (Accelerated Strategic Computing Initiative) joined SGI and Cray.
- In 1997 IBM, DEC, HP and others were invited to join the group now called OpenMP.
- In 1997 the specification for OpenMP 1.0 for FORTRAN was released.
- Next year the specification for C/C++ was released.
- The current version is OpenMP 4.0 and was published 2013.
- GCC 4.4 supports OpenMP.

OpenMP Components

1. Compiler directives. `#pragma` in C/C++.
2. A runtime library
3. Environment variables, like `OMP_NUM_THREADS`.

Barriers

- A barrier is a synchronization primitive which makes all threads reaching the barrier wait for each other.
- When the last thread has reached the barrier, all threads can proceed and continue after the barrier.

`#pragma omp parallel`

- A **structured block of code** is either
  - a compound statement, i.e. a block enclosed in braces, or
  - a for-loop.
- The pragma `omp parallel` is used before a structured block of code and specifies all threads should execute all of that block.
- Note that this is typically not what we want in a for-loop, see below.
- A default number of threads is used, which can be changed with the environment variable `OMP_NUM_THREADS`, which can be larger than the number of processors in the machine.
- This pragma creates an implicit barrier after the structured block.

Example

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

int main(void)
{
        #pragma omp parallel
        {
                int     tid; // thread id.

                tid = omp_get_thread_num();
                printf("hello world from thread %d\n", tid);
        }

        return 0;
}
```

- Since tid is declared in the compound statement, it becomes private.
- `omp_get_thread_num()` returns an id starting with zero.

OpenMP and Pthreads

- The OpenMP runtime library creates the threads it needs using Pthreads on Linux.
- After a parallel block, the threads wait for the next their work and are not destroyed in between.
- This model of parallelism is called **fork-join** and only the master thread executes the sequential code.
- It's possible to nest parallel regions — but when I tested no additional threads where used (instead the master thread ran the code four times).

Two OpenMP Functions

- To specify in the program how many threads you want, use

```
omp_set_num_threads(nthread);
```

- To measure elapsed wall clock time in seconds, use

```
double  start, end;

start = omp_get_wtime();
/* work. */
end = omp_get_wtime();
```

Parallel For-Loops

- In addition to the `#pragma omp parallel` you must also specify `#pragma omp for`. before the loop.
- Without the second pragma each thread will execute all iterations.
- Note that it's the programmer's responsibility to check that there are no data dependences between loop iterations.

For Loop Scheduling

- There are three ways to schedule for-loops:
- `schedule(static)`
  - The iterations are assigned statically in contiguous blocks of iterations.
  - Static scheduling has the least overhead, obviously, but may suffer from poor load imbalance, e.g. in an LU-decompsition.
- `schedule(dynamic)` or `schedule(dynamic, size)`
  - The default size is one iteration.
  - A thread is assigned size contigouos iterations at a time.
- `schedule(guided)` or `schedule(guided, size)`
  - The default size is one iteration.
  - With a size, a thread never (except possibly at the end) is assigned few than size contigouos iterations at a time.
  - The number of iterations assigned to a thread is proportional to the number of unassigned iterations and the number of threads.
- In addition, `runtime` can be specified which uses the environment variable `OMP_SCHEDULE` which must be one of above three but without the size.

An Example

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

#define N (1024)

float a[N][N];
float b[N][N];
float c[N][N];

int main(void)
{
        int     i;
        int     j;
        int     k;

        #pragma omp parallel private(i,j,k)
        #pragma omp for schedule(static, N/omp_get_num_procs())
        for (i = 0; i < N; ++i)
                for (k = 0; k < N; ++k)
                        for (j = 0; j < N; ++j)
                                a[i][j] += b[i][k] * c[k][j];


        return 0;
}
```

- We need private for j and k since they are declare before the pragma.
- If a function is called in a parallel region, all its local variables become private.

Parallel Tasks

```
#pragma omp sections
{
        #pragma omp section
        {
                work_a();
        }

        #pragma omp section
        {
                work_b();
        }

}
```

- Each section is executed in parallel.

Reductions

- By a **reduction** is meant computing a scalar value from an array such as a sum.
- The loop has a data dependence on the `sum` variable.
- How can we parallelize it anyway?

```
float   a[N];
float   sum;
int     i;

for (sum = i = 0; i < N; ++i)
        sum += a[i];
```

OpenMP Reductions

- By introducing a sum variable private to each thread, and letting each thread compute a partial sum, we can parallelize the reduction:

```
float   a[N];
float   sum;
int     i;

#pragma omp parallel
#pragma omp for
#pragma omp reduction(+:sum)
for (sum = i = 0; i < N; ++i)
        sum += a[i];
```

- We can write the pragmas on one line if we wish:

```
#pragma omp parallel for reduction(+:sum)
for (sum = i = 0; i < N; ++i)
        sum += a[i];
```

- There are reductions for: `+ - * & | ^ && ||` with suitable start values such as 1 for `*` and `~0` for `&`.

Critical Sections

- A critical sections is created as in:

```
#pragma omp critical
{
        point->x += dx;
        point->y += dy;
}
```

Atomic Update

- When one variable should be updated atomically, we can use:

```
#pragma omp atomic
count += 1;
```

Explicit Barriers

- Recall there is an implicit barrier at the end of a parallel region.
- To create a barrier explicitly, we can use:

```
    #pragma omp barrier
```

Work for One Thread

- Recall only the master executes the sequential code between parallel regions.
- If we wish only the master should execute some code in a parallel region, we can use

  ```
  #pragma omp master
  ```

- If it doesn't matter which thread performs the work, we can instead use

  ```
  #pragma omp single
  ```

- There is a difference between the two above constructs: an implicit barrier is created after a `single` directive.

Locks

- OpenMP supports two kinds of locks: plain locks and recursive locks.
- Recall a thread can lock a recursive lock it already owns without blocking for ever.
- Recursive locks are called nested locks in OpenMP.
- The lock functions are `omp_init_lock`, `omp_set_lock`, `omp_unset_lock`, `omp_test_lock` and `omp_destroy_lock`, and `omp_nest_init_lock`, `omp_nest_set_lock`, `omp_nest_unset_lock`, `omp_nest_test_lock` and `omp_nest_destroy_lock`

OpenMP Memory Consistency Model

- Weak ordering is the consistency model for OpenMP.
- The required synchronization instructions are inserted implicitly with the above introduded directives.
- A for loop can be created without an implicit barrier using `nowait` and in that case `#pragma omp flush` makes caches consistent.
- A list of variables to write back can be specified:
  `#pragma omp flush(a,b,c)`

Compiler Support for OpenMP

- The following compilers support OpenMP
- GNU, IBM XL, Oracle, Intel, Portland Group, Pathscale, Absoft, Fujitsu, Microsoft, HP, Cray.