# Lecture 18—Automatic Parallelization, OpenMP

## ECE 459: Programming for Performance

February 13, 2015

# Road Map

- Previously: compilers & automatic parallelization, OpenMP.
- Now: More OpenMP.
- Soon: Reading week.

# Reductions

Recall that we introduced the concept of a reduction, e.g.

```
for (int i = 0; i < length; i++)
    total += array[i];
```

What is the appropriate scope for `total`?

# Reductions

Recall that we introduced the concept of a reduction, e.g.

```
for (int i = 0; i < length; i++)
    total += array[i];
```

What is the appropriate scope for `total`?
Well, it should be shared.

- We want each thread to be able to write to it.

# Reductions

Recall that we introduced the concept of a reduction, e.g.

```
for (int i = 0; i < length; i++)
    total += array[i];
```

What is the appropriate scope for `total`?
Well, it should be shared.

- We want each thread to be able to write to it.
- But, is there a race condition? (of course)

Aha! OpenMP can deal with reductions as a special case:

```
#pragma omp parallel for reduction (+:total)
```

specifies that the `total` variable is the accumulator for a reduction over the + operator.

# Accessing Private Data outside a Parallel Region

Sometimes you want private variables, but want them initialized before the loop.

Consider this (silly) code:

```
int data=1;
#pragma omp parallel for private(data)
for (int i = 0; i < 100; i++)
    printf ("data=%d\n", data);
```

- `data` is private, so OpenMP will not copy in initial 1.
- To make OpenMP copy the data before the threads start, use `firstprivate(data)`.
- To publish a variable after the (sequentially) last iteration of the loop, use `lastprivate(data)`.

# Thread-Private Data

You might have a global variable, for which each thread should have a persistent local copy—lives across parallel regions.

- Use the `threadprivate` directive.
- Add `copyin` if you want something like `firstprivate`.
- There is no `lastprivate` since the data is accessible after the loop.

# Thread-Private Data Example (1)

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

int tid, a, b;

#pragma omp threadprivate(a)

int main(int argc, char *argv[])
{
    printf("Parallel #1 Start\n");
    #pragma omp parallel private(b, tid)
    {
        tid = omp_get_thread_num();
        a = tid;
        b = tid;
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    printf("Sequential code\n");
```

# Thread-Private Data Example (2)

```
    printf("Parallel #2 Start\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    return 0;
}
```

```
% ./a.out
Parallel #1 Start
T6: a=6, b=6
T1: a=1, b=1
T0: a=0, b=0
T4: a=4, b=4
T2: a=2, b=2
T3: a=3, b=3
T5: a=5, b=5
T7: a=7, b=7
```

```
Sequential code
Parallel #2 Start
T0: a=0, b=0
T6: a=6, b=0
T1: a=1, b=0
T2: a=2, b=0
T5: a=5, b=0
T7: a=7, b=0
T3: a=3, b=0
T4: a=4, b=0
```

# Collapsing Loops

- Normally, it's best to parallelize the outermost loop.

Consider this code:

```
#include <math.h>
int main() {
    double array[2][10000];
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < 2; i++)
      for (int j = 0; j < 10000; j++)
        array[i][j] = sin(i+j);
    return 0;
}
```

- Would parallelizing this outer loop benefit us? What about the inner loop?

OpenMP supports *collapsing* loops:

- Creates a single loop for all the iterations of the two loops.
- Outer loop only enables the use of 2 threads.
- Collapsed loop lets us use up to 20,000 threads.

# Better Performance Through Scheduling: An Example

Default mode: *Static scheduling*.

- Assumes each iteration takes the same running time.

Does that assumption hold for this code?

```
double calc(int count) {
    double d = 1.0;
    for (int i = 0; i < count*count; i++) d += d;
    return d;
}

int main() {
    double data[200][100];
    int i, j;
    #pragma omp parallel for private(i, j) shared(data)
    for (int i = 0; i < 200; i++) {
        for (int j = 0; j < 200; j++) {
            data[i][j] = calc(i+j);
        }
    }
    return 0;
}
```

# Better Performance Through Scheduling

- In example, earlier iterations are faster than later iterations.
  Result: sublinear scaling—wait for all iterations to finish.

- Turn on *dynamic schedule* mode
  by adding `schedule(dynamic)` to the pragma:
  - Breaks the work into chunks;
  - Distributes the work to each thread in chunks;
  - Higher overhead;
  - Default chunk size of 1
    (can modify, e.g. `schedule(dynamic, n/50)`).

# More Scheduling

Other schedule modes exist, e.g. `guided`, `auto` and `runtime`.

- `guided` changes the chunk size based on work remaining.
  - Default minimum chunk size $= 1$ (can modify)
- `auto` lets OpenMP decide what's best.
- `runtime` doesn't pick a mode until runtime.
  - Tune with `OMP_SCHEDULE` environment variable

# Part I

# Beyond `for` loops: OpenMP Parallel Sections and Tasks

# Why more than for?

So far, we can parallelize (some) `for` loops with OpenMP.

Less powerful than Pthreads. (Also harder to get wrong.)

Reflects OpenMP's scientific-computation heritage.

Today, we need more general parallelism, not just matrices.

# Parallel Sections Example: Linked Lists (1)

Purely-static mechanism for specifying independent work units which should run in parallel.

Linked list example:

```
#include <stdlib.h>

typedef struct s { struct s* next; } S;

void setuplist (S* current) {
  for (int i = 0; i < 10000; i++) {
    current->next = (S*) malloc (sizeof(S));
    current = current->next;
  }
  current->next = NULL;
}
```

# Parallel Sections Example: Linked Lists (2)

(Exactly) 2 linked lists:

```
int main() {
  S var1, var2;
  #pragma omp parallel sections
  {
    #pragma omp section
    { setuplist (&var1); }
    #pragma omp section
    { setuplist (&var2); }
  }
  return 0;
}
```

Parallelism structure explicitly visible.

Finite number of threads.

(What's another barrier to parallelism here?)

# Nested Parallelism

Sometimes you don't want to collapse loops.

Example: (better example in PDF notes!)

```
#pragma omp parallel sections
{
  #pragma omp section
  {
      #pragma omp parallel for
      for (int i = 0; i < 1000; i++) { ... }
  }
  #pragma omp section
  {
      #pragma omp parallel for
      for (int i = 0; i < 1000; i++) { ... }
  }
}
```

To enable nested parallelism, call `omp_set_nested(1)` or set `OMP_NESTED`. (Runtime might refuse.)

# OpenMP Tasks

Main new feature in OpenMP 3.0.

`#pragma omp task`:
code splits off and scheduled to run later.

More flexible than parallel sections:

- can run as many threads as needed;
- tasks do not need to join (like detached threads).

OpenMP does the task-to-thread mapping—lower overhead.

# Examples of tasks

Two examples:

- web server

  unstructured requests
- user interface

  allows users to start concurrent tasks

# Boa webserver main loop example

```
#pragma omp parallel
  /* a single thread manages the connections */
  #pragma omp single nowait
  while (!end) {
    process any signals
    foreach request from the blocked queue {
      if (request dependencies are met) {
        extract from the blocked queue
        /* create a task for the request */
        #pragma omp task untied
          serve_request(request);
      }
    }
    if (new connection) {
      accept_connection();
      /* create a task for the request */
      #pragma omp task untied
        serve_request(new connection);
    }
    select();
  }
```

# Other OpenMP qualifiers

`untied`: lifts restrictions on task-to-thread mapping.

`single`: only one thread runs the next statement (not $N$ copies).

`flush` directive: write all values in registers or cache to memory.

`barrier`: wait for all threads to complete. (OpenMP also has implicit barriers at ends of parallel sections.)

OpenMP also supports critical sections (one thread at a time), atomic sections, and typical mutex locks (`omp_set_lock`, `omp_unset_lock`).