Parallel Programming with OpenMP

Troels Henriksen

Cons and pros of programming with pthreads

+ Full low-level control over what threads are doing.
+ Can implement complicated synchronisation.
+ Useful for *task parallelism*, where threads do wildly different things (e.g. a thread per request to web server).

Cons and pros of programming with pthreads

\+ Full low-level control over what threads are doing.

\+ Can implement complicated synchronisation.

\+ Useful for *task parallelism*, where threads do wildly different things (e.g. a thread per request to web server).

\- Extremely tedious to use.

\- Very easy to make mistakes.

\- Flexibility is not needed for much scientific computing.

- There is a zoo of higher-level parallel programming libraries and languages.
- We will look at a particularly popular and simple one: **OpenMP**.

Parallel loops

For scientific computing, we are mostly concerned with parallelising straightforward loops.

Matrix multiplication

```
void matmul(int n,
            const double *x, const double *y, double *out) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      out[i*n+j] = 0;
      for (int l = 0; l < n; l++) {
        out[i*n+j] += x[i*n+j] * y[i*n+j];
      }}}} // To fit on slide. Don't code like this.
```

- Iterations of the outer two loops are *independent*.
- Can be computed by different threads.

Simplest OpenMP example

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
  A[i] = A[i]*2;
}
```

- *Directives* used to indicate how run C program in parallel.
- *Clauses* (covered later) can be used to customise the behaviour.
- Semantics are the *sequential elision*—how the program would behave if we ignored the directives.
- **We are only scratching the surface of OpenMP in this course!**

Fork-join programming model

```
printf("Program starts\n");      Sequential
N = 1000;                            ↓

#pragma omp parallel for            Parallel
for (i=0; i<N; i++)                ↓↓↓↓↓↓
  A[i] = B[i] + C[i];              ↓↓↓↓↓↓

M = 500;                           Sequential
                                     ↓

#pragma omp parallel for            Parallel
for (j=0; j<M; j++)                ↓↓↓↓↓↓
  p[j] = q[j] - r[j];              ↓↓↓↓↓↓

printf("Program done\n");        Sequential
exit(0);                             ↓
```

- Program starts sequential.
- Parallel regions split across multiple threads.
- Parallel region ends when *all* threads done.
- Worker threads kept running in background.

Compilation

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int n = 100000000;

  int *arr = malloc(n*sizeof(int));

#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    arr[i] = i;
  }

  free(arr);
}
```

```
$ gcc -o openmp-example openmp-example.c -fopenmp
```

Controlling number of threads

```
$ time OMP_NUM_THREADS=1  ./openmp-example
real     0m0.124s
user     0m0.034s
sys      0m0.090s
$ time OMP_NUM_THREADS=2  ./openmp-example
real     0m0.076s
user     0m0.033s
sys      0m0.104s
$ time OMP_NUM_THREADS=4  ./openmp-example
real     0m0.054s
user     0m0.039s
sys      0m0.133s
$ time OMP_NUM_THREADS=8  ./openmp-example
real     0m0.046s
user     0m0.054s
sys      0m0.184s
```

Memory model

- Variables declared inside a loop iteration are *private*.
- Variables declared outside are *shared*.
- **As always, be extremely careful when modifying shared variables—OpenMP will not protect you!**

```
double sum = 0;
#pragma omp parallel for
for (int i=0; i < n; i++) {
  sum += A[i];
}
```

Mutexes are against the spirit of OpenMP—so how do we parallelise a loop like this?

Reductions

Instead of doing a summation sequentially

$$(((((((x_0 + x_1) + x_2) + x_3) + x_4) + x_5) + x_6) + x_7)$$

we can do it like

$$(x_0 + x_1 + x_2 + x_3) + (x_4 + x_5 + x_6 + x_7)$$

and have one compute the left part, and a second one compute the right, combining their results at the end.

- Is that valid?

Reductions

Instead of doing a summation sequentially

$$(((((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus x_5) \oplus x_6) \oplus x_7)$$

we can do it like

$$(x_0 \oplus x_1 \oplus x_2 \oplus x_3) \oplus (x_4 \oplus x_5 \oplus x_6 \oplus x_7)$$

and have one compute the left part, and a second one compute the right, combining their results at the end.

- Is that valid?
- What about now?

What must hold for an operator $\oplus$ for such a rewrite to be valid?

Commutativity and associativity

A binary operator $\oplus : \alpha \rightarrow \alpha \rightarrow \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

Commutativity and associativity

A binary operator $\oplus : \alpha \rightarrow \alpha \rightarrow \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

It is *associative* if

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

We can do a parallel "sum" with any associative operator, called a *reduction*.

Commutativity and associativity

A binary operator $\oplus : \alpha \rightarrow \alpha \rightarrow \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

It is *associative* if

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

We can do a parallel "sum" with any associative operator, called a *reduction*.

For convenience, we also tend to require a *neutral element* $0_{\oplus}$:

$$x \oplus 0_{\oplus} = 0_{\oplus} \oplus x = x$$

OpenMP dot product with reduction clause.

```
double dotprod(int n, double *x, double *y) {
  double sum = 0;
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < n; i++) {
    sum += x[i] * y[i];
  }
  return sum;
}
```

- Must initialise accumulator variable to neutral element.
- Must explicitly tell OpenMP the combining operator (+, *, -, &&, ||, &, |, ^, max, or min).

Parallelising matrix multiplication

```
void matmul_seq(int n,
                const double *x, const double *y,
                double *out) {
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
      double acc = 0;
      for (int l = 0; l < n; l++)
        out[i*n+j] += x[i*n+j] * y[i*n+j];
      out[i*n+j] = acc;
    }
}
```

- Runtime for `n=1000`: 0.87s.
- Three nested loops.
- Which do we parallelise, and how?

Parallelising outermost loop

```
void matmul_outer(int n,
                  const double *x, const double *y,
                  double *out) {
#pragma omp parallel for
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
      double acc = 0;
      for (int l = 0; l < n; l++)
        acc += x[i*n+j] * y[i*n+j];
      out[i*n+j] = acc;
    }
}
```

- Runtime: 0.059s (vs. 0.87s sequential) — pretty good for adding one line of code!
- But this only parallelises across the rows of the result matrix.

Parallelising both outer loops

```
void matmul_collapse(int n,
                     const double *x, const double *y,
                     double *out) {
#pragma omp parallel for collapse(2)
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
      double acc = 0;
      for (int l = 0; l < n; l++)
        acc += x[i*n+j] * y[i*n+j];
      out[i*n+j] = acc;
    }
}
```

- `collapse(2)` clause combines loops to one parallel `n*n` iteration loop,
- Runtime: 0.055s (vs. 0.059s before) — not much impact here.
- Would matter more if we had fewer iterations in outer loop.

Parallelising the innermost (dot product) loop

```
void matmul_inner(int n,
                  const double *x, const double *y,
                  double *out) {
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
      double acc = 0;
#pragma omp parallel for reduction(+:acc)
      for (int l = 0; l < n; l++)
        acc += x[i*n+j] * y[i*n+j];
      out[i*n+j] = acc;
    }
}
```

- Runtime: 3.95s (vs. 0.87s sequential)—this sucks.
- **Almost always better to parallelise outermost loops.**

Scheduling clauses

- By default, OpenMP splits the iterations of a parallel loop evenly among the threads (*static scheduling*).
- This is not always optimal.

Scheduling clauses

- By default, OpenMP splits the iterations of a parallel loop evenly among the threads (*static scheduling*).
- This is not always optimal.

```
int fib(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

- Consider a loop that computes `fib(i)` for each `i<n`.
- Since time to compute `fib(i)` is over twice that of `fib(i-1)`, the threads with the early iterations finish much faster than the threads with the later iterations.

Static scheduling

```
#pragma omp parallel for schedule(static)
  for (int i = 0; i < n; i++) {
    fibs[i] = fib(i);
  }
```

- For `n=45` my machine runs this in 5.2*s*.
- Using a process monitor (`htop` on Unix) I can see that many of my processing cores are idle for most of the run-time.

Dynamic scheduling

- *Dynamic scheduling* assigns each thread an iteration, and is given more iterations when it finishes.
- **No idle threads** (as long as there are unclaimed iterations to run).

```
#pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < n; i++) {
    fibs[i] = fib(i);
  }
```

- Runs in 2.27—much better!
- By default, assigns single iterations at a time, which is slow for very large loops with quick iterations.
- Use `schedule(dynamic,K)` to schedule `K` iterations at a time to threads.

Summary

- OpenMP is a simple language extension for parallelising loops in C programs.
- Use `#pragma omp parallel for` to parallelise a **for**-loop.
- Remember to compile with `-fopenmp`.
- Use the `reduce` clause for aggregation loops.
- Use the `schedule` clause to tweak how work is allocated to threads.