OpenMP Basics

Ned Nedialkov

McMaster University

14-16 February 2023

Outline

Intro

- OpenMP, Open Multi-Processing
- Application programming interface (API) for shared-memory programming in C, C++, and Fortran
- Consists of
  - compiler directives
  - library routines
  - environment variables

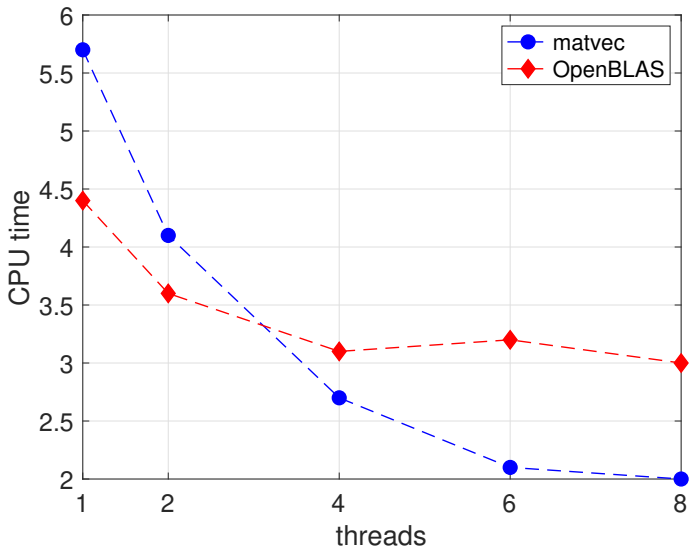
Example

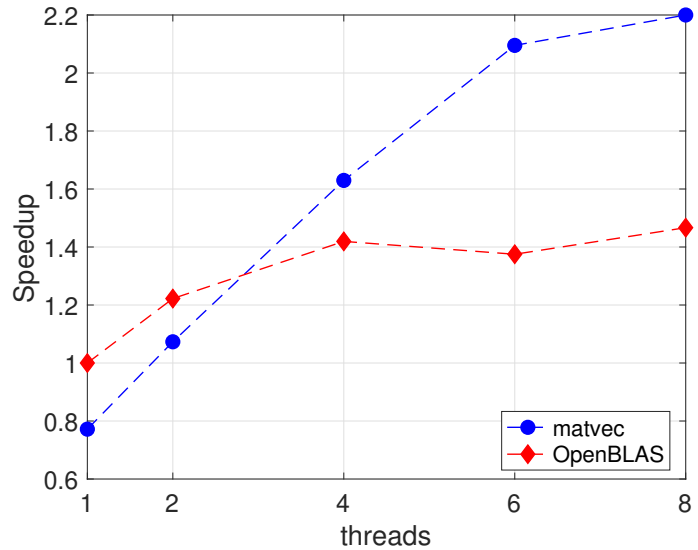
Consider the following code for computing $y = Ax$, where A is an nrows $\times$ ncols matrix.

```
void matvec(int nrows, int ncols, const double *A, const double *b, double
        *y) {
#pragma omp parallel for
  for (int i = 0; i < nrows; i++)
    {
      y[ i ] = 0;
      for (int j = 0; j < ncols; j++)
        y[ i ] += A[ i * ncols + j ] * b[ j ];
    }
}
```

`#pragma omp parallel for`

tells the compiler to parallelize a loop by dividing the loop iterations among threads.





- `matvec` vs OpenBLAS `cblas_dgemv`
- A is 40,000×40,000.
- Apple M1 Pro, clang-15 with `-O3 -march=native -fopenmp`
- Speedup plot of `matvec` is wrt to `cblas_dgemv` on one core

omp parallel

`#pragma omp parallel`

- Specifies a region (or block) to be executed in parallel.
  Also called parallel region (or block).

```
#pragma omp parallel
  {
   // each thread executes this block
  }
```

- Number of threads is specified by the environment variable `OMP_NUM_THREADS`.
- One can also specify the number of threads as:

```
#pragma omp parallel num_threads ( n )
  {
   // each of n threads executes this block
  }
```

Example

```
// parall.c
#include <omp.h>
#include <stdio.h>

int main()
{
   int num_procs = omp_get_num_procs();
   printf("Number of processors %d\n", num_procs);

#pragma omp parallel
   {
      int thread_num = omp_get_thread_num();
      printf("thread number %d \n", thread_num);
   }
   return 0;
}
```

- To compile with `gcc`/`clang`, flag: `-fopenmp`.
- Run as
  `OMP_NUM_THREADS=4 ./parall`
  or
  ```
  export OMP_NUM_THREADS=4
  ./parall
  ```
- On my machine with 10 cores and 4 threads, one output is:

```
Number of processors 10
thread number 3
thread number 2
thread number 0
thread number 1
```

Fork/join execution model

- A single thread executes until a parallel region.
- At the entrance of it, each of the (specified number of) threads starts executing the statements in the parallel region.
- Threads share work through work-sharing constructs, e.g. parallel loops, sections.
  If no such constructs, each thread executes the region redundantly.
- There is an implicit barrier at the end of a parallel region.
- When all threads are finished, a single thread continues.

omp for

```
#pragma omp parallel
{
#pragma omp for
   for (...)
   {
      // ...
   }
}
```

`#pragma omp for`

- work-sharing construct
- distribute loop iterations among the threads of a parallel region.

`#pragma omp parallel for`

parallelize a loop by dividing the loop iterations among threads.

Subtlety

```
#pragma omp parallel for
for (...)
  {
    // ...
  }
#pragma omp parallel for
for (...)
  {
    // ...
  }
```

Parallel region cost incurred twice.

vs

```
#pragma omp parallel
{
#pragma omp for
  for (...)
    {
      // ...
    }
#pragma omp for
  for (...)
    {
      // ...
    }
}
```

Parallel region cost incurred once.

Sections

```
#pragma omp sections
```

- Specifies a non-iterative work-sharing construct within a parallel region.
- Each section is executed by a single thread.
- If more threads than sections, some will have no work and will "jump" to the implied barrier at the end of the sections construct.

Example

```
#pragma omp parallel
#pragma omp sections
  {
#pragma omp section
    { printf("section 1 thread %d\n", omp_get_thread_num()); }
#pragma omp section
    { printf("section 2 thread %d\n", omp_get_thread_num()); }
#pragma omp section
    { printf("section 3 thread %d\n", omp_get_thread_num()); }
  }
```

output can be

```
section 1 thread 0
section 3 thread 2
section 2 thread 1
```

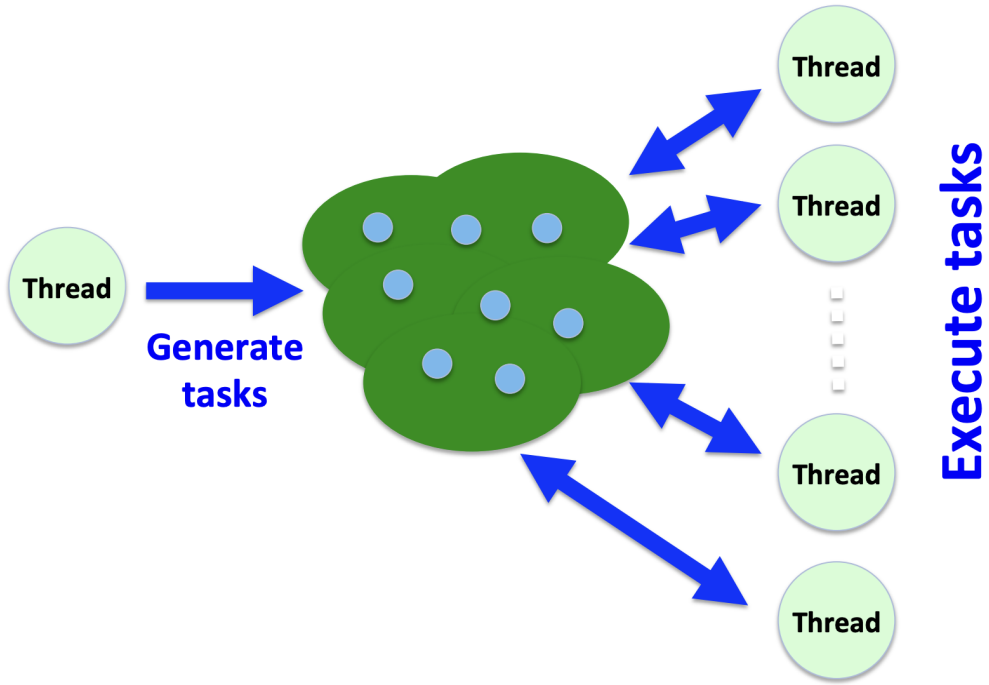
Example

```
#pragma omp parallel
  {
#pragma omp for
   for (int i=0;i<4; i++)
     printf("for loop i=%d thread %d\n", i, omp_get_thread_num());
#pragma omp sections
   {
#pragma omp section
     { printf("section 1 thread %d\n", omp_get_thread_num()); }
#pragma omp section
     { printf("section 2 thread %d\n", omp_get_thread_num()); }
   }
  }
```

output (with 8 threads)

```
for loop i=0  thread 0
for loop i=2  thread 2
for loop i=1  thread 1
for loop i=3  thread 3
section 1 thread 0
section 2 thread 1
```

Tasks



- Allow units of work to be generated dynamically.
- A thread generates tasks.
- Different threads can execute them.
- When they are executed is up to the runtime system.

See ▸ Ruud van der Pas. OpenMP Tasking Explained (picture from there).

```
#pragma omp task
  {
  // code for a task to be generated
  }
```

Usually a single thread creates tasks through

```
#pragma omp single
  {
  // this code is run by a single thread
  }
```

Example

```
#pragma omp parallel
  {
#pragma omp single
    {
      printf("Single block on thread %d\n", omp_get_thread_num());
#pragma omp task
      { printf("task 1 on thread %d\n", omp_get_thread_num()); }
#pragma omp task
      { printf("task 2 on thread %d\n", omp_get_thread_num()); }
#pragma omp task
      { printf("task 3 on thread %d\n", omp_get_thread_num()); }
    }
  }
```

output (using 8 threads)

```
Single block on thread 8
task 1 on thread 1
task 2 on thread 7
task 3 on thread 1
```

If `omp single` is missing, each thread will create tasks.

Completion of tasks can be ensured through task synchronization:

```
#pragma omp barrier
```

or

```
#pragma omp taskwait
```

Example

```
#pragma omp parallel
  {
#pragma omp single
    {
      printf("Single block on thread %d\n", omp_get_thread_num());
#pragma omp task
      { printf("task 1 on thread %d\n", omp_get_thread_num()); }
#pragma omp task
      { printf("task 2 on thread %d\n", omp_get_thread_num()); }
#pragma omp task
      { printf("task 3 on thread %d\n", omp_get_thread_num()); }
    }
#pragma omp taskwait
       printf("After taskwait thread %d\n", omp_get_thread_num());
  }
```

output (using 4 threads)

```
Single block on thread 0
task 3 on thread 0
task 2 on thread 2
task 1 on thread 1
After taskwait thread 0
After taskwait thread 3
After taskwait thread 2
After taskwait thread 1
```

Some data-sharing attributes

Each of the following specifies for a variable or a list of variables their sharing/accessibility.

`private` each thread has a copy of a variable and it is accessed by a single thread only.

`firstprivate` like private, but should be initialized.

`lastprivate` the value of the enclosing version of the variable is the value that is last assigned to it by a thread.

`shared` shared variables between threads.
A variable declared before a parallel block is shared. If read-only, not an issue; if assigning values, potential race conditions.

`default` specifies the behavior of unscoped variables in a parallel region.
`default(none)` forces one to specify shared, private, etc.

Example

Fibonacci numbers $f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$.

Computing them recursively is not efficient (exponential time).
Here we illustrate tasks.

Serial implementation:

```
unsigned long fib_serial(unsigned n)
{
  if (n < 2) return n;
  unsigned long f1 = fib_serial(n - 1);
  unsigned long f2 = fib_serial(n - 2);
  return f1 + f2;
}
```

Parallel, version 1:

```
unsigned long fib_parallel_1(unsigned n)
{
  if (n < 2) return n;
  unsigned long f1, f2;
#pragma omp task default(none) shared(f1) firstprivate(n)
  f1 = fib_parallel_1(n - 1);
#pragma omp task default(none) shared(f2) firstprivate(n)
  f2 = fib_parallel_1(n - 2);
#pragma omp taskwait // wait for tasks to finish
  return f1 + f2;
}
```

We call it as

```
#pragma omp parallel
#pragma omp single
  fibn = fib_parallel_1(n);
```

Parallel, version 2:

```
unsigned long fib_parallel_2(unsigned n)
{
  if (n <= 30) return fib_serial(n);
  unsigned long f1, f2;
#pragma omp task default(none) shared(f1) firstprivate(n)
  f1 = fib_parallel_2(n - 1);
#pragma omp task default(none) shared(f2) firstprivate(n)
  f2 = fib_parallel_2(n - 2);
#pragma omp taskwait
  return f1 + f2;
}
```

Parallel, version 3:

```
unsigned long fib_parallel_3(unsigned n)
{
  if (n < 2) return n;
  unsigned long f1, f2;
#pragma omp task default(none) shared(f1) firstprivate(n) if (n > 30)
  f1 = fib_parallel_3(n - 1);
#pragma omp task default(none) shared(f2) firstprivate(n) if (n > 30)
  f2 = fib_parallel_3(n - 2);
#pragma omp taskwait
  return f1 + f2;
}
```

On a 10-core M1 Mac with 4 threads (time is in seconds):

```
fib_serial(45)     = 1134903170, time  3.46
fib_parallel_1(45) = 1134903170, time 47.22, speedup  0.07
fib_parallel_2(45) = 1134903170, time  0.90, speedup  3.83
fib_parallel_3(45) = 1134903170, time 28.45, speedup  0.12

with 8:

fib_serial(45)     = 1134903170, time  3.45
fib_parallel_1(45) = 1134903170, time 24.62, speedup  0.14
fib_parallel_2(45) = 1134903170, time  0.47, speedup  7.41
fib_parallel_3(45) = 1134903170, time 14.70, speedup  0.23
```