Multicore Programming: OpenMP

Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

https://gforgeron.gitlab.io/progsys/

The OpenMP standard (www.openmp.org)

- Parallel Programming Interface designed for shared-memory multiprocessor machines
  - Language extensions to C, C++ and Fortran

- Incremental parallelization
  - `#pragma omp` *`directive`*
  - Less intrusive than adding calls to libraries (e.g. POSIX threads)
  - Pragmas can be ignored to easily switch back to the original sequential code
    - Hmm, really?

The OpenMP standard (www.openmp.org)

- Incremental parallelization
  - Pragmas are like "On my honor, I swear that this code is parallel"
    - Compiler will trust you! (no check)
  - `#pragma omp` *`directive clause clause`* …
    - The more you say, the more performance you can get (hopefully)
  - Seems like a piece of cake, uh?
- The OpenMP standard keeps evolving
  - Architecture Review Board (Intel, IBM, AMD, Microsoft, Oracle, etc.)

Our first “Hello World” program

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

int main ()
{
#pragma omp parallel
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  hello.c  -o hello
[my-machine] ./hello
Hello world!
Bye!
```

Our first “Hello World” program

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

int main ()
{
#pragma omp parallel
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

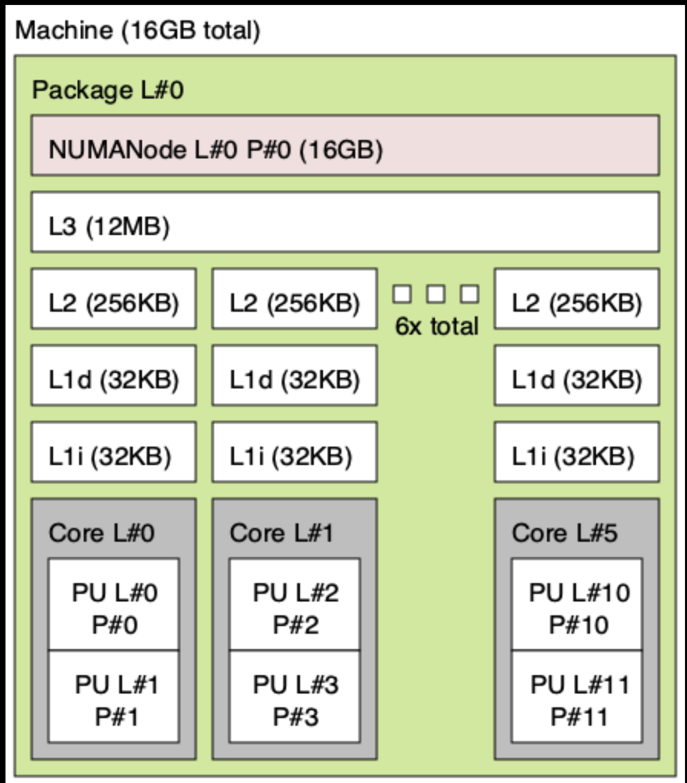
```
[my-machine] make
gcc -Wall  -fopenmp hello.c  -o hello
[my-machine] ./hello
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Bye!
```

Our first “Hello World” program

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

int main ()
{
#pragma omp parallel
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  -fopenmp hello.c  -o hello
[my-machine] ./hello  | cat -n
    1  Hello world!
    2  Hello world!
    3  Hello world!
    4  Hello world!
    5  Hello world!
    6  Hello world!
    7  Hello world!
    8  Hello world!
    9  Hello world!
   10  Hello world!
   11  Hello world!
   12  Hello world!
   13  Bye!
```

Our first “Hello World” program



Output of the “lstopo” command on my-machine

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] ./hello  | cat -n
     1  Hello world!
     2  Hello world!
     3  Hello world!
     4  Hello world!
     5  Hello world!
     6  Hello world!
     7  Hello world!
     8  Hello world!
     9  Hello world!
    10  Hello world!
    11  Hello world!
    12  Hello world!
    13  Bye!
```

Our first “Hello World” program

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

int main ()
{
#pragma omp parallel
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] OMP_NUM_THREADS=4 ./hello | cat -n
     1  Hello world!
     2  Hello world!
     3  Hello world!
     4  Hello world!
     5  Bye!
```

Our first "Hello World" program

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

int main ()
{
#pragma omp parallel num_threads(6)
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] ./hello | cat -n
   1  Hello world!
   2  Hello world!
   3  Hello world!
   4  Hello world!
   5  Hello world!
   6  Hello world!
   7  Bye!
```

Our first "Hello World" program

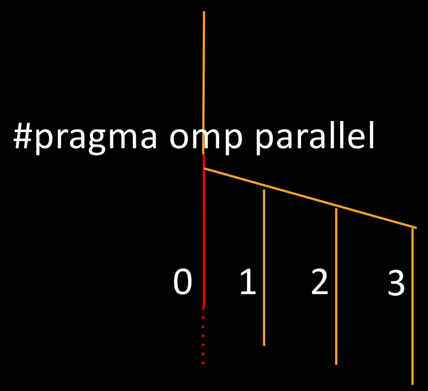
```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main ()
{
#pragma omp parallel num_threads(6)
  printf ("Hello world!\n");
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

Usually not a good idea

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] ./hello | cat -n
   1  Hello world!
   2  Hello world!
   3  Hello world!
   4  Hello world!
   5  Hello world!
   6  Hello world!
   7  Bye!
```

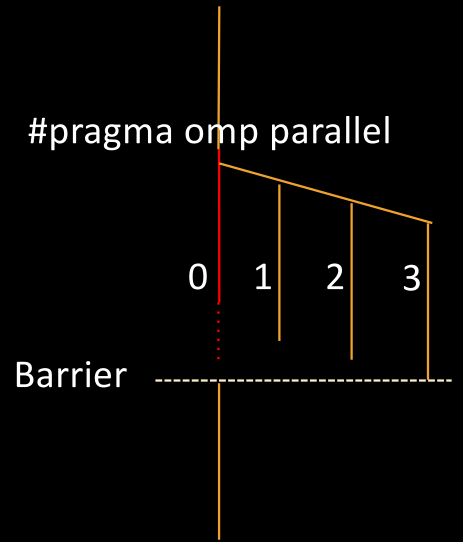
Fork-Join parallelism

- A single thread initially executes the main function
- When it reaches a “parallel” directive
  - A team of threads is created
  - The initial thread is part of the team (and is the master)
  - Each thread executes the parallel region



Fork-Join parallelism

- At the end of the parallel region
  - All threads enter a synchronization barrier (*rendez-vous*)
  - When all threads have reached the barrier, all threads but the master are freed
  - The master thread can then continue executing code beyond the region



How to introduce divergence?

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

int main ()
{
#pragma omp parallel
  printf ("Hello from %d!\n", omp_get_thread_num());
  printf ("Bye!\n");

  return EXIT_SUCCESS;
}
```

```
[my-machine] make
gcc -Wall  -fopenmp hello.c   -o hello
[my-machine] OMP_NUM_THREADS=4 ./hello
Hello from 0!
Hello from 3!
Hello from 1!
Hello from 2!
Bye!
```

How to introduce divergence?

```
int main()
{
#pragma omp parallel
  {
    switch (omp_get_thread_num())
    {
      case 0:
        f(); break;
      case 1:
        g(); break;
      ...
    }
  }
  return EXIT_SUCCESS;
}
```

- Not a sound solution
  - Parallelism is usually not linked to the number of OpenMP threads!
- Our program is definitely not an “incremental” evolution of a sequential one…

Loop parallelism

```
int main ()
{


    for (int i = 0; i < 10; i++)
      f (i);

  return EXIT_SUCCESS;
}
```

- We assume that f(i) calls can be performed in parallel

Loop parallelism

```
int main ()
{
#pragma omp parallel
  {
    for (int i = 0; i < 10; i++)
      f (i);
  }
  return EXIT_SUCCESS;
}
```

- We assume that f(i) calls can be performed in parallel
- In the current code
  - f(0) is executed by all threads
  - So are f(1), f(2), …

Loop parallelism

```
int main ()
{
#pragma omp parallel
  {
    for (int i = 0; i < 10; i++)
      f (i);
  }
  return EXIT_SUCCESS;
}
```

- We assume that f(i) calls can be performed in parallel
- In the current code
  - f(0) is executed by all threads
  - So are f(1), f(2), …
- We'd like to distribute the iteration range to the thread!

Loop parallelism

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 10; i++)
      f (i);
  }
  return EXIT_SUCCESS;
}
```

Distribute iteration range

- We assume that f(i) calls can be performed in parallel
- In the current code
  - f(0) is executed by all threads
  - So are f(1), f(2), …
- We'd like to distribute the iteration range to the thread!

Loop parallelism

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 10; i++)
      printf("f(%d) computed by %d\n",
             i, omp_get_thread_num());
  }
  return EXIT_SUCCESS;
}
```

```
[my-machine] OMP_NUM_THREADS=4 ./loop
f(0) computed by 0
f(1) computed by 0
f(8) computed by 3
f(9) computed by 3
f(6) computed by 2
f(7) computed by 2
f(2) computed by 0
f(3) computed by 1
f(4) computed by 1
f(5) computed by 1
```

Loop parallelism

```c
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 10; i++)
      printf("f(%d) computed by %d\n",
             i, omp_get_thread_num());
  }
  return EXIT_SUCCESS;
}
```

- By default (with gcc), the iteration range is splitted in chunks
  - Each thread was assigned one chunk of contiguous iterations
  - That is: static partitioning

Loop parallelism

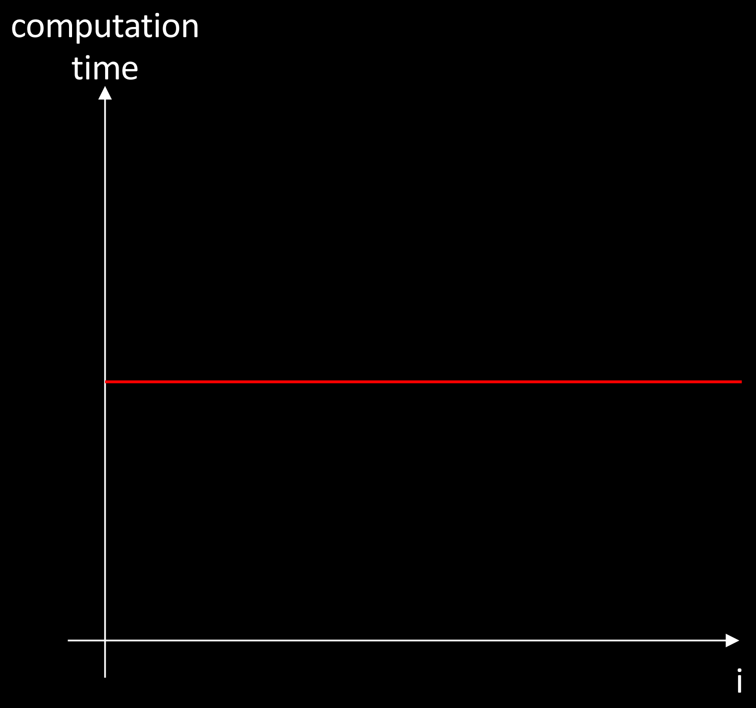
```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 10; i++)
      printf("f(%d) computed by %d\n",
             i, omp_get_thread_num());
  }
  return EXIT_SUCCESS;
}
```

- By default (with gcc), the iteration range is splitted in chunks
  - Each thread was assigned one chunk of contiguous iterations
  - That is: static partitioning
- Side note: an implicit barrier takes place at the end of the loop

Parallelizing computations

- How good is a static *block* distribution?
  - OK if the computation time of f(i) is constant
    - I.e. does not depend on the value of i

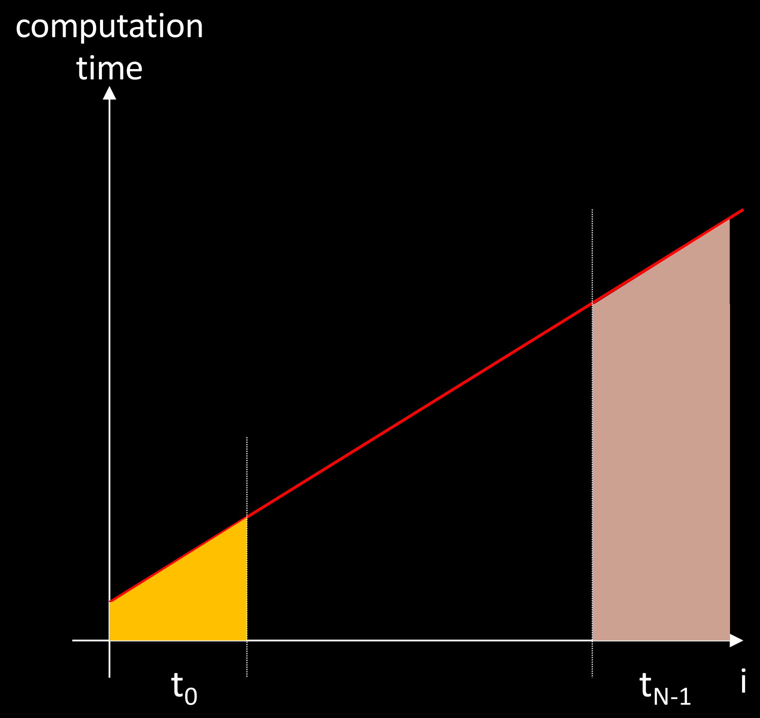
```
#pragma omp for schedule (static)
  for (int i = 0; i < 10; i++)
    f (i);
```



Parallelizing computations

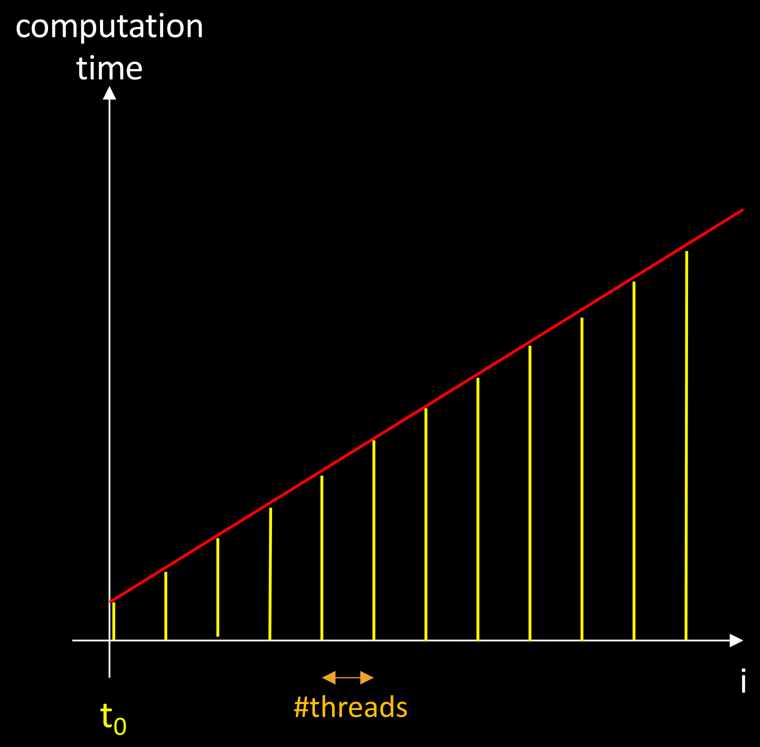
- What if the computation time is linearly increasing?
  - Our block distribution is no longer relevant
    - Well, using a mirror block distribution assigning two blocks per thread would work…

- What kind of distribution should we use?



Parallelizing computations

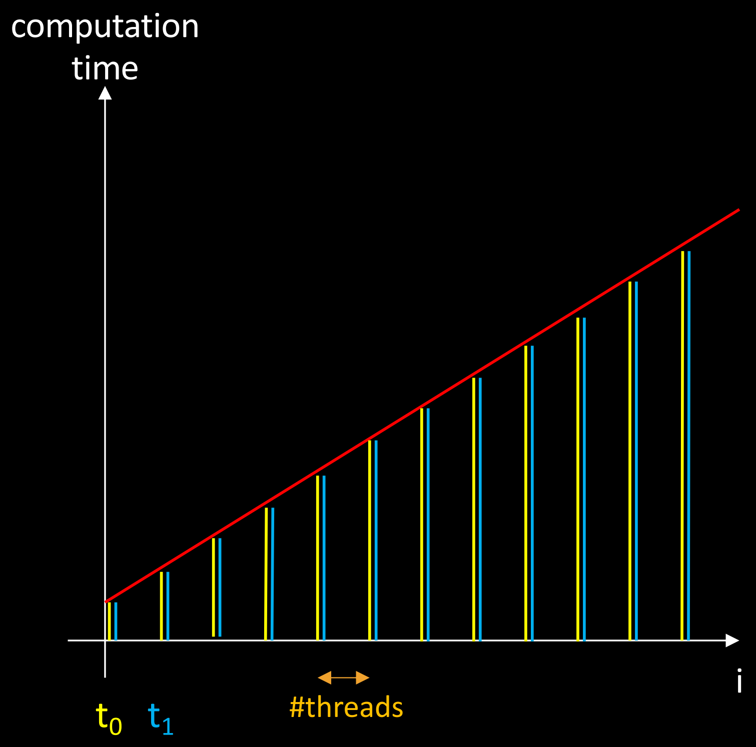
- What if the computation time is linearly increasing?
  - A cyclic distribution of indexes would be a good option



Parallelizing computations

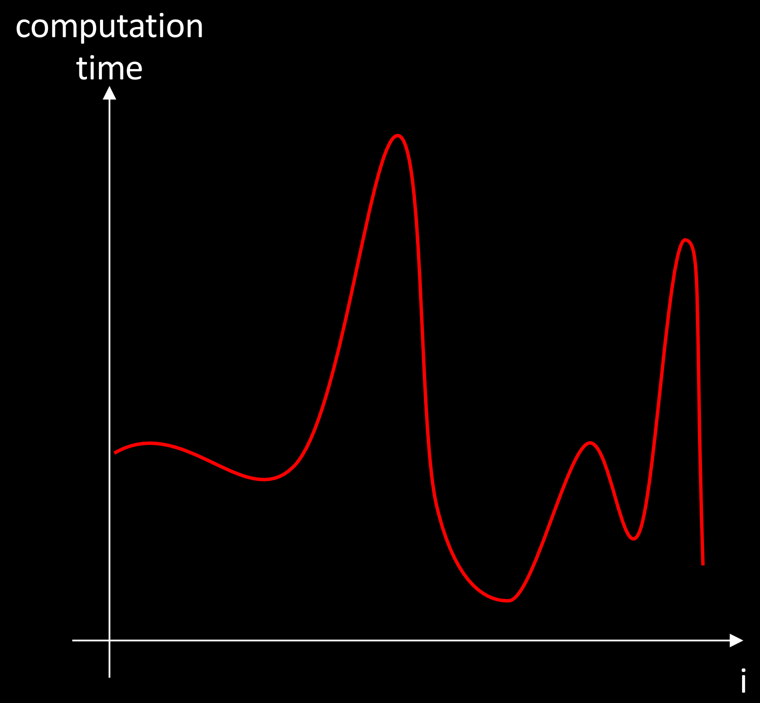
- What if the computation time is linearly increasing?
  - A cyclic distribution of indexes would be a good option

```
#pragma omp for schedule (static, 1)
  for (int i = 0; i < 10; i++)
    f (i);
```



Parallelizing computations

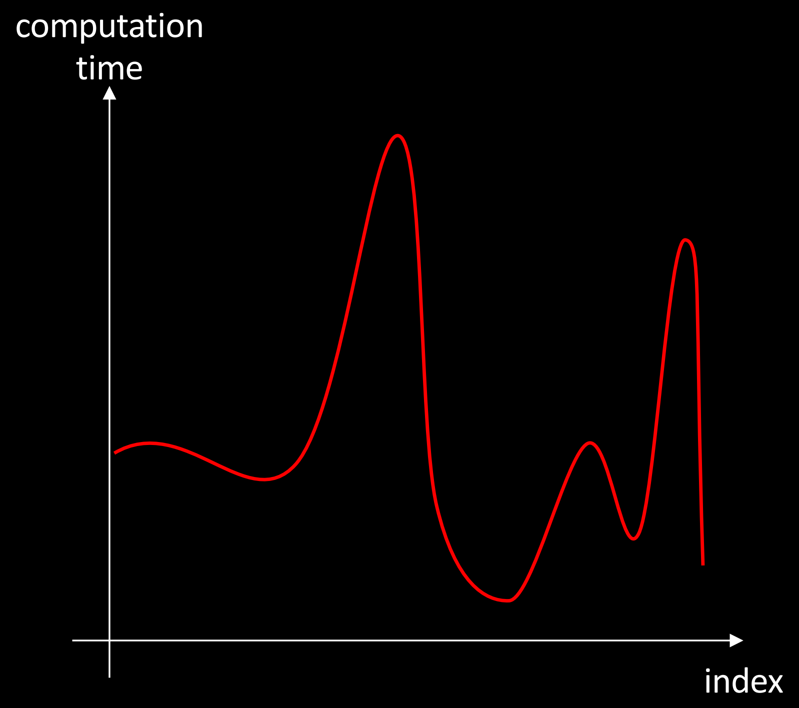
- What if the computation time is unpredictable?
  - Even the cyclic strategy may fail



Parallelizing computations

- What if the computation time is unpredictable?
  - Dynamic strategy
    - Distribute indexes in a greedy manner

```
#pragma omp for schedule (dynamic)
  for (int i = 0; i < 10; i++)
    f (i);
```



Fixing loop scheduling at run time

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for schedule (runtime)
    for (int i = 0; i < 10; i++)
      printf("f(%d) computed by %d\n",
             i, omp_get_thread_num());
  }
  return EXIT_SUCCESS;
}
```

```
[my-machine] OMP_SCHEDULE=dynamic ./loop
f(0) computed by 0
f(2) computed by 1
f(3) computed by 1
f(4) computed by 1
f(5) computed by 1
f(6) computed by 1
f(7) computed by 1
f(8) computed by 1
f(1) computed by 0
f(9) computed by 2
```

Collapsing nested loops

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 4; j++)
        f (i, j);
  }
  return EXIT_SUCCESS;
}
```

- Problem
  - We only distribute 3 i-values to threads
    - Then each threads executed the j-loop sequentially

Collapsing nested loops

```
int main ()
{
#pragma omp parallel
  {
    for (int i = 0; i < 3; i++)
#pragma omp for
      for (int j = 0; j < 4; j++)
        f (i, j);
  }
  return EXIT_SUCCESS;
}
```

- Problem
  - We only distribute 3 i-values to threads
    - Then each threads executed the j-loop sequentially
  - Moving `#pragma omp for` between i-loop and j-loop doesn't help that much

Collapsing nested loops

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 4; j++)
        f (i, j);
  }
  return EXIT_SUCCESS;
}
```

- Ideally, we'd like to perform all the f() calls in parallel on a 12-core machine

Collapsing nested loops

```
int main ()
{
#pragma omp parallel
  {
#pragma omp for collapse (2)
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 4; j++)
        f (i, j);
  }
  return EXIT_SUCCESS;
}
```

Merge two loops

- Ideally, we'd like to perform all the f() calls in parallel on a 12-core machine
- The collapse clause distributes all possible (i, j) pairs to threads
  - Can be used in conjunction with schedule (*policy*)