OpenMP Basics: Directives and Runtime

Piotr Luszczek

Restrictions on OpenMP Loops

```
#pragma omp for
for (index = <START> ; index { < | <= | >= | > } <END>; { index++ | ++index | index-- | --index | index += inc | index -= inc | index = index + inc | index = inc + index | index = index - inc } )
```

Allowed (does not change iteration count)

```
continue
```

```
break
exit()
goto
return
```

Not allowed

Changes iteration count

Working Around Restrictions

- The restrictions are in place so that the OpenMP runtime can compute the right schedule for all threads
  - Complicated loops require complicated math or cannot be computed at all
  - STL containers often cannot easily know their size to compute a balanced schedule for threads
- Example: go over powers of two
  - Bit shifting loop:

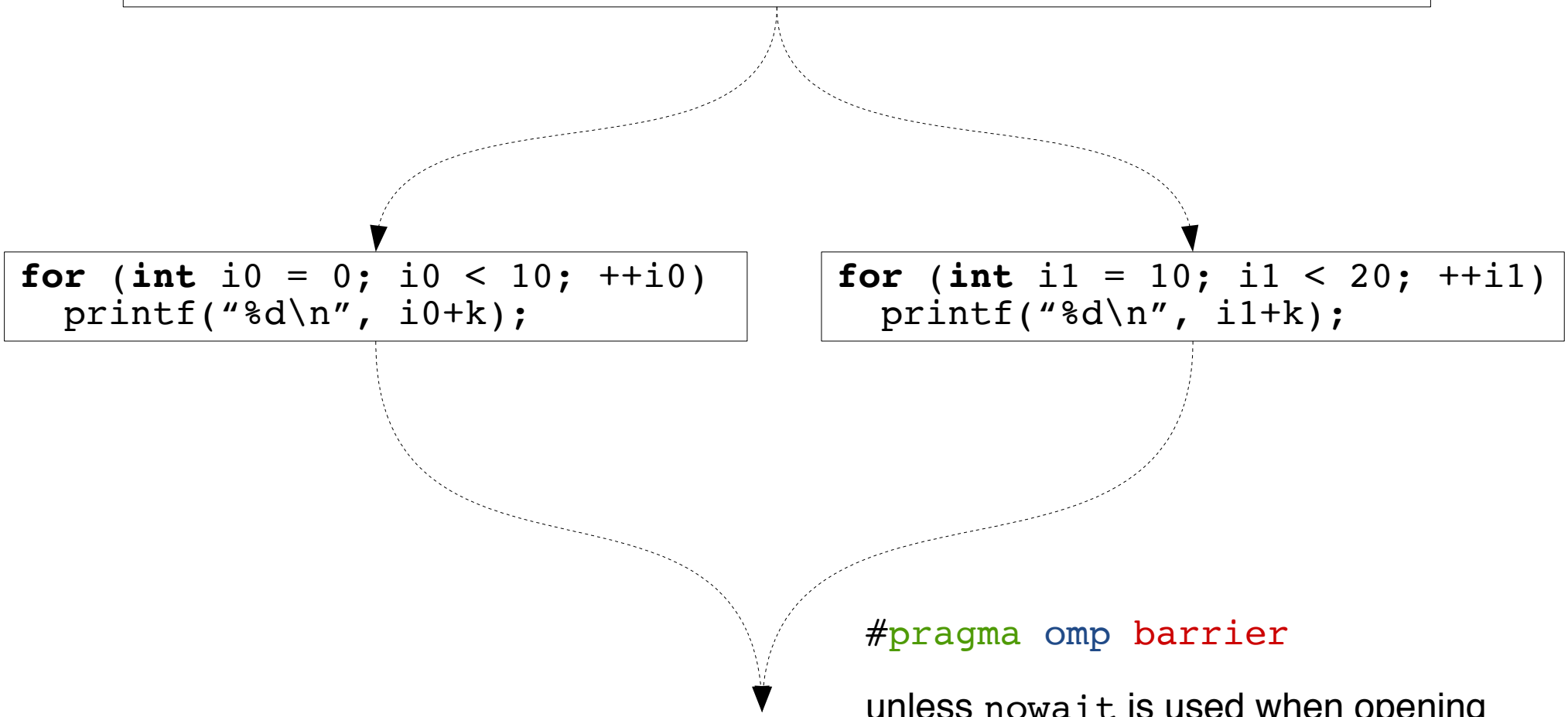
```
for (k = 1 << 31 ; k != 0 ; k >>= 1) {
}
```

replace with:

```
// compiler "sees" that there are 31 iterations
for (kk = 31 ; kk > 0 ; --kk) {
  k = 1 << kk;
}
```

Variable Scope: shared and private

```
#pragma omp parallel for shared(k) private(i)
for (int i = 0; i < 20; ++i)
  printf("%d\n", i+k);
```



```
for (int i0 = 0; i0 < 10; ++i0)
  printf("%d\n", i0+k);
```

```
for (int i1 = 10; i1 < 20; ++i1)
  printf("%d\n", i1+k);
```

```
#pragma omp barrier
```

unless `nowait` is used when opening a region

Variable Scope: shared and private

```
j = 13;
#pragma omp parallel for private(i) firstprivate(j)
for (int i = 0; i < 20; ++i)
  printf("%d\n", i+j);
```

```
j0 = j; // copy from master thread
for (int i0 = 0; i0 < 10; ++i0)
  printf("%d\n", i0+j0);
```

```
j1 = j; // copy from master thread
for (int i1 = 10; i1 < 20; ++i1)
  printf("%d\n", i1+j1);
```

`lastprivate` can be used for copying private variable out of the "last" thread into the master thread.

Runtime Functions

```
#ifdef _OPENMP
#include <omp.h>
#endif


omp_set_num_threads(13);

printf("%d\n", omp_in_parallel()); // in parallel region? NO

#pragma omp parallel
{

printf("%f\n", omp_get_wtime()); // wall clock time

printf("%d\n", omp_in_parallel()); // in parallel region? YES

printf("%d\n", omp_get_num_threads());// number of active threads

printf("%d\n", omp_get_thread_num());// thread number (starting at 0)

printf("%d\n", omp_get_num_procs());// number of processors

printf("%f\n", omp_get_wtime()); // wall clock time

}
```

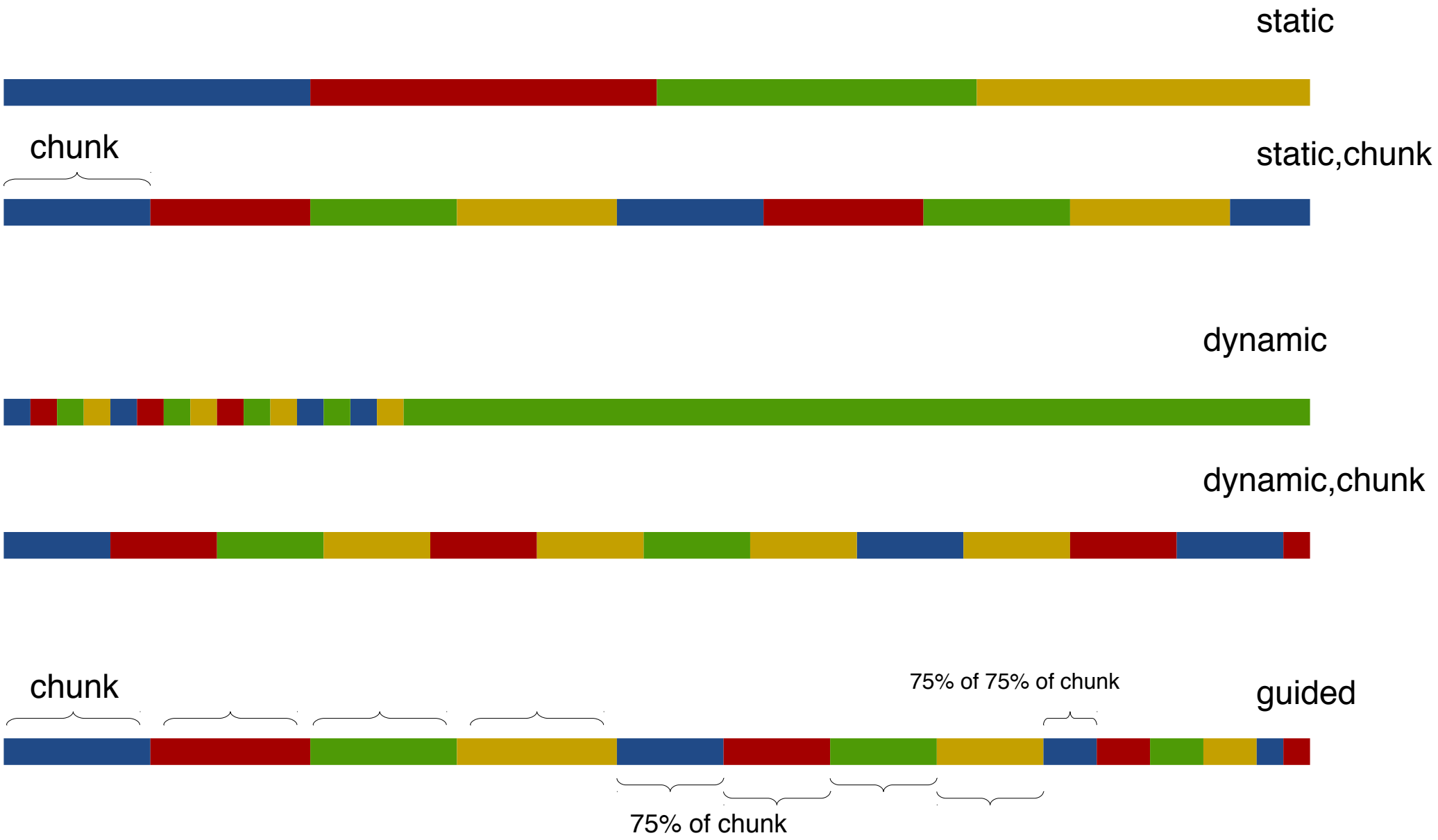
Mutual Exclusion: Directives and Functions

```
#pragma omp parallel for
for (int i=0; i<N; ++i) {
  if (omp_get_thread_num() == 7)
    printf( "%d %d %d\n", __LINE__, i, omp_get_thread_num() );

  #omp critical
  printf( "%d %d %d\n", __LINE__, i, omp_get_thread_num() );

  #omp single
  printf( "%d %d %d\n", __LINE__, i, omp_get_thread_num() );

  #omp master
  printf( "%d %d %d\n", __LINE__, i, omp_get_thread_num() );
}

omp_lock_t L;
omp_init_lock( &L );

#pragma omp parallel for
{
  omp_set_lock( &L );

  omp_unset_lock( &L );

  omp_test_lock( &L );
}

omp_destroy_lock( &L );
```

Scheduling for Loops

- #pragma omp parallel **for** schedule(<TYPE> [, <CHUNK>])
- Not all loops benefit from the same type parallelism and/or are load balanced: for (N=2; N<100; ++N) matmatmul(N, a, b, c)
- schedule(static) – each thread gets #iters / THREADS
- schedule(static, C) – first thread gets C iterations, second thread gets the next C iterations, ...
- schedule(dynamic) – first thread gets an iteration and then gets another available iteration when its finished
- schedule(dynamic, C) – first thread gets C iterations, ...
- schedule(guided) – chunks are exponentially decreasing to 1
- schedule(guided, C) – chunks are exponentially decreasing to C
- schedule(runtime)
  - export/setenv OMP_SCHEDULE “static,1”

OpenMP Schedules' Details



Collapsing Nested Loops

- Multiple loop nests are supported in OpenMP
  - Compiler rearranges the code behind the scenes into single loop
  - The standard schedule types work on the reorganized loop
    - More opportunities for parallelism
- Consider matrix multiplication:

```
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

- With OpenMP:

```
#pragma omp parallel for collapse(3)
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j)
    for (int k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j];
```