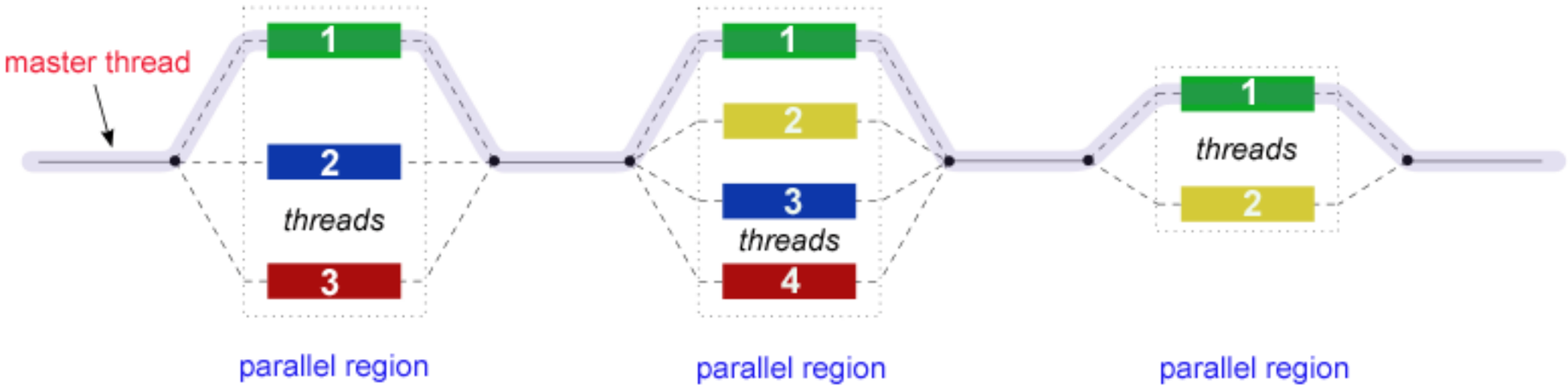
Discussion 5

Basic OpenMP



Pthreads vs OpenMP

- Low-level model
- Not really portable
  - Different versions/ library for each language
- More programming effort
- Fine grained control (ex: for Data)

- High-level API
- Good for loop-level parallelism, and even more now
- Multi-platform/Portable
  - Supports C/C++, Fortran
- Less programming effort for similar speedup
- Easier work sharing (ex: *omp for* inside *parallel* region)
- Can be used in conjunction with MPI

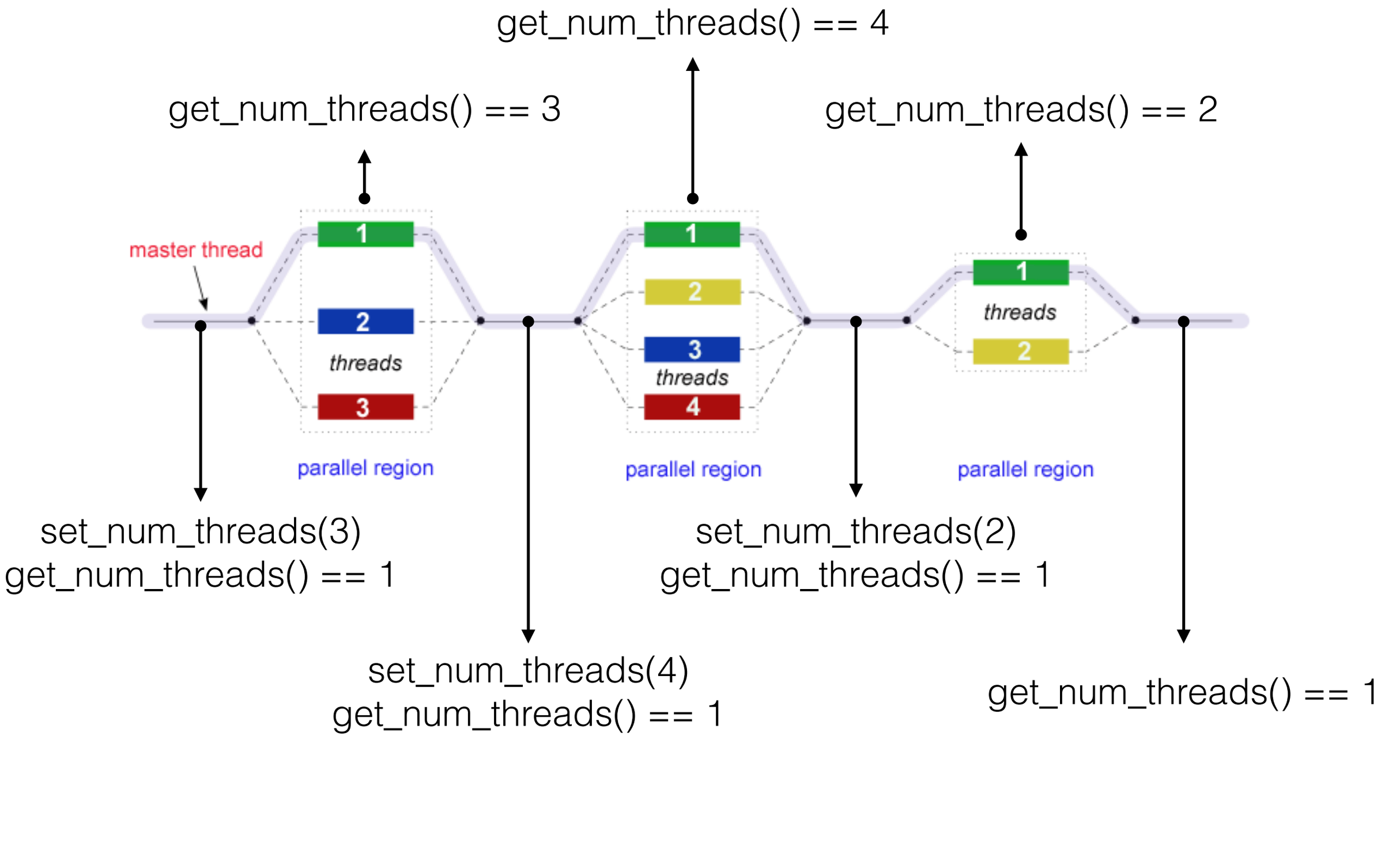
Essential OpenMP Directives and Functions

- omp_get_num_threads()
- omp_get_thread_num()
- omp_set_num_threads(numThreads) - *Don't use this for the assignment*
- #pragma omp parallel
- #pragma omp parallel for
    - Static
    - Dynamic
- #pragma omp critical
- #pragma omp task
- #pragma omp taskwait

Set/Get Num Threads

- Set environment variable using CL (Recommended):
  - export/set **OMP_NUM_THREADS**=<number of threads to use>
- **omp_get_num_threads**():
  - Returns the number of threads executing in the current region (parallel or not)
    - If not in a parallel region it will return 1 (i.e. only the master thread is running)
- **omp_set_num_threads**(numThreads)
  - Overrides the above env variable.
  - Declared outside OMP clauses.
  - Sets the number of threads
  - numThreads must a int larger than 0

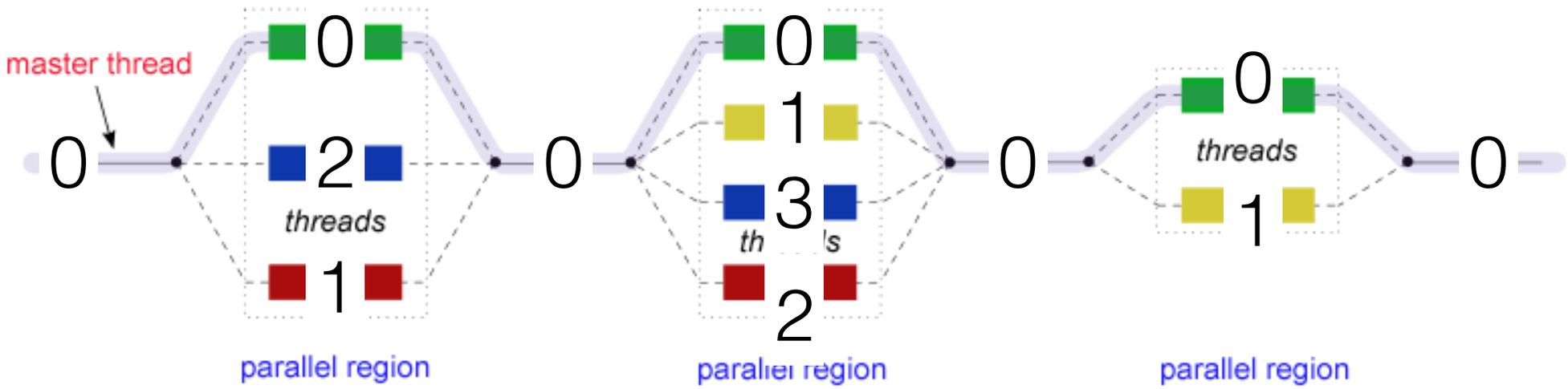
Threads


get_num_threads() == 4
get_num_threads() == 3
get_num_threads() == 2
master thread
1
2
threads
3
parallel region
1
2
3
threads
4
parallel region
1
threads
2
parallel region
set_num_threads(3)
get_num_threads() == 1
set_num_threads(4)
get_num_threads() == 1
set_num_threads(2)
get_num_threads() == 1
get_num_threads() == 1

omp_get_thread_num()

- Returns an Id for the thread that is executing
- Id is unique and always the same in parallel region
- It is between 0 and (omp_get_num_threads() - 1)
- Master Thread always has Id 0

Thread Id’s


master thread
0
0
2
threads
1
parallel region
0
0
1
3
2
parallel region
0
0
threads
1
parallel region

#pragma omp parallel

- Defines a parallel region whose code will be executed by multiple threads in parallel

```
#pragma omp parallel [clauses]
{
   code_block
}
```

- Implicit barrier/join at the end of parallel region
- Clauses are all optional, all have default values

#pragma omp parallel Clauses

- `num_threads` —> how many threads execute parallel region
  - If not defined it is the default value.
    - Default value is:
      - the one set by set_num_threads or
      - the environment variable $OMP_NUM_THREAD
- Order of clauses doesn't matter
- Several other clauses.
- See https://msdn.microsoft.com/en-us/library/68ah4xc7.aspx

#pragma omp parallel Clauses

- `private` —> list of variables that are private to the threads inside of the parallel region
  - Variables are copied into a local version for each thread
  - Changes to these variable are not shared between threads
  - Changes to these variable will not be reflected after the parallel region
  - If a variable is not private it is shared between all of the threads in the parallel region and changes to it will be reflected outside of the region
  - There are other similar options: **firstPrivate** (initialize with a value), also lastPrivate
  - Good example to understand clauses:
    - https://msdn.microsoft.com/en-us/library/c3dabskb.aspx
    - Default clause (Shared or Private): https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbcpx01/cuppvars.htm

Example

```
#pragma omp parallel num_threads(4)
{
   int i = omp_get_thread_num()
   printf_s("Hello from thread %d\n", i);
}
```

=

```
set_num_threads(4);
#pragma omp parallel
{
   int i = omp_get_thread_num();
   printf_s("Hello from thread %d\n", i);
}
```

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

- Notice that thread id's go from 0 to 3
- Order of print statements may change

#pragma omp parallel for

- Defines a parallel for whose iterations will be divided among several threads

```
#pragma omp [parallel] for [clauses]
{
   code_block
}
```

- If in a parallel region then don't put the parallel…

```
#pragma omp parallel for [clauses]
{
   code_block
}
```

=

```
#pragma omp parallel [clauses]
#pragma omp for [clauses]
{
   code_block
}
```

OMP Parallel For

General Syntax:

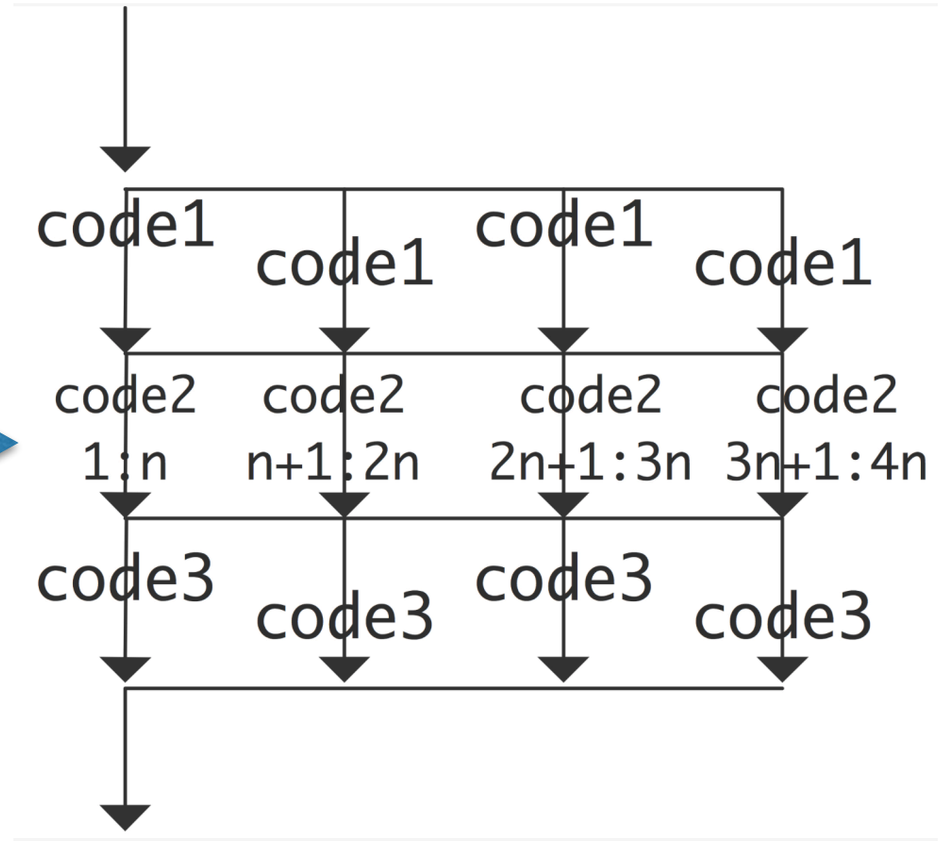
```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
  // do something with i
}
```

Or

```
#pragma omp parallel for
  for (i=0; .....
```

Reason you may want a separate parallel code segment:

```
#pragma omp parallel
{
  code1();
#pragma omp for
  for (i=1; i<=4*N; i++) {
    code2();
  }
  code3();
}
```



Execution of parallel code inside and outside a loop

Source: http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

OpenMP *for* scheduling

```
#pragma omp parallel for schedule(kind [,chunkSize])
```

- A parallel for divides the iterations between the executing threads
- How is the dividing done?
  - It depends on the schedule kind clause(static, dynamic, guided)
- There are several kinds of scheduling, each has it's advantages and disadvantages
  - We will use: Static and Dynamic
- See https://msdn.microsoft.com/en-us/library/x5aw0hdf.aspx

Static and Dynamic Scheduling

- **Static** scheduling divides iterations evenly between threads
  - Default chunk size: *num_Iterations/num_Threads*
- **Dynamic** scheduling uses an internal work queue to dispatch chunkSized blocks to each thread. When a thread finishes a block, it gets the next block
  - Default chunk size is 1
  - Think Part B Lab 1

Static vs Dynamic Scheduling

This order will change on every execution for dynamic

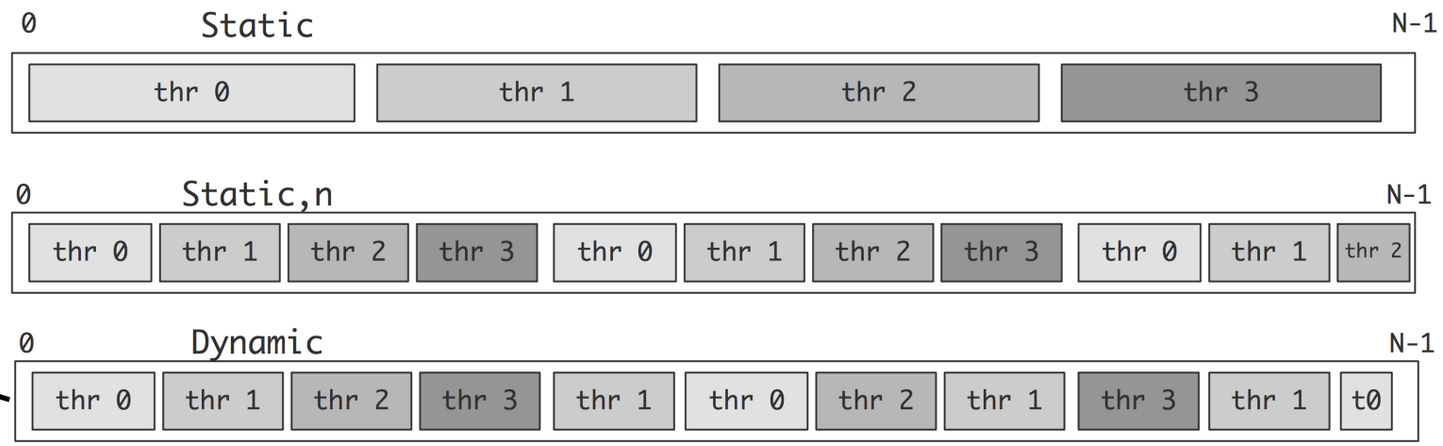
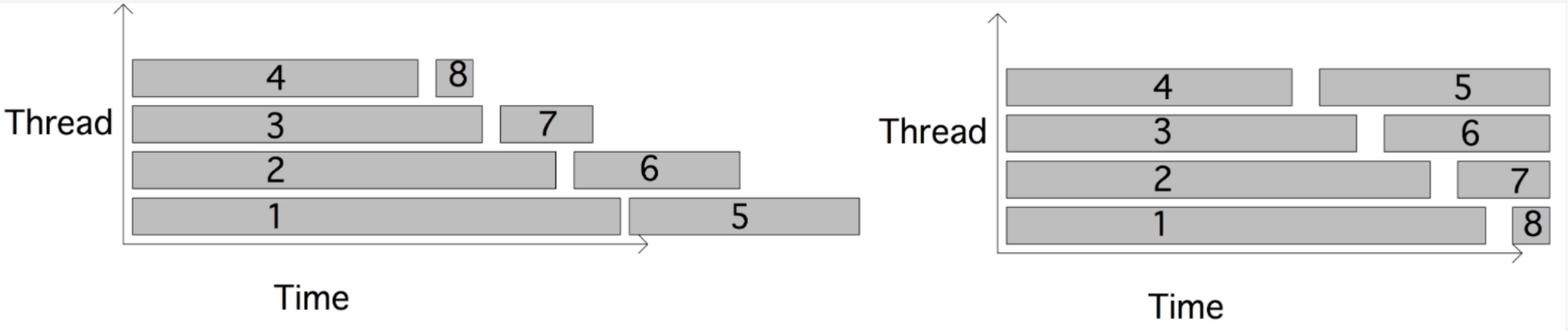


Illustration static round-robin scheduling versus dynamic



Source: http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

#pragma omp task

- When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time.
- If deferred, the task in placed in a conceptual pool of tasks associated with the current parallel region.
- All team threads will take tasks out of the pool and execute them until the pool is empty.
- A thread that executes a task might be different from the thread that originally encountered it.

#pragma omp task

Clauses:

- depend (list)
- if (expression)
- final (expression)
- untied
- mergeable
- default (shared | firstprivate | none)
- private (list)
- firstprivate (list)
- shared (list)
- priority (value)

```
int main(int argc, char *argv[]) {
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf(“A “);
      #pragma omp task {printf("race ");}
      #pragma omp task {printf("car ");}
    }
  }//End omp parallel - taskwait not needed
  printf("\n");
  return(0);
}

OUTPUT: A race car / A car race
```

#pragma omp task

```c
float sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return *a;
    }

    // recursive case
    size_t half = n / 2;
    return sum(a, half) + sum(a + half, n - half);
}
```

```c
float sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }

    // recursive case
    size_t half = n / 2;
    float x, y;

    #pragma omp parallel
    #pragma omp single nowait
    {
        #pragma omp task shared(x)
        x = sum(a, half);
        #pragma omp task shared(y)
        y = sum(a + half, n - half);
        #pragma omp taskwait
        x += y;
    }
    return x;
}
```

#pragma omp task

```c
float sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }

    // recursive case
    size_t half = n / 2;
    float x, y;

    #pragma omp parallel
    #pragma omp single nowait
    {
        #pragma omp task shared(x)
        x = sum(a, half);
        #pragma omp task shared(y)
        y = sum(a + half, n - half);
        #pragma omp taskwait
        x += y;
    }
    return x;
}
```

```c
#define CUTOFF 100  // arbitrary

static float parallel_sum(const float *, size_t);
static float serial_sum(const float *, size_t);

float sum(const float *a, size_t n)
{
    float r;

    #pragma omp parallel
    #pragma omp single nowait
    r = parallel_sum(a, n);
    return r;
}

static float parallel_sum(const float *a, size_t n)
{
    // base case
    if (n <= CUTOFF) {
        return serial_sum(a, n);
    }

    // recursive case
    float x, y;
    size_t half = n / 2;

    #pragma omp task shared(x)
    x = parallel_sum(a, half);
    #pragma omp task shared(y)
    y = parallel_sum(a + half, n - half);
    #pragma omp taskwait
    x += y;

    return x;
}

static float serial_sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0.;
    }
    else if (n == 1) {
        return a[0];
    }

    // recursive case
    size_t half = n / 2;
    return serial_sum(a, half) + serial_sum(a + half, n - half);
}
```

#pragma omp critical

- Defines a critical region where only one thread can execute at a time. Good for output and debugging

```
#pragma omp critical [(name)]
{
   // code_block
}
```

- (name) is used to identify the region, two critical regions with the same name are considered the same region
- (name) must be between parentheses
- See https://msdn.microsoft.com/en-us/library/b38674ky.aspx
- Similar option for single assignment statement: #pragma omp atomic

#pragma omp atomic

- Similar to *critical*, this directive allows access of a specific memory location atomically.
- Advantage: Fewer locks/mutexes, limited but more efficient (Similar to Compare-and-Exchange/Swap)
- Atomic Clauses:
  - Update (default) -> x++, x--, ++x, --x, binary ops (+, *, -, /, &, ^, |, <<, >>)
  - Read -> y = x
  - Write -> x = y
  - Capture -> y = x++, x--, ++x, --x, binary ops

```
#pragma omp atomic [atomic-clause]
{
   // atomic operation
}
```

```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic read
  temp[i] = x[f(i)];

  #pragma omp atomic write
  x[i] = temp[i]*2;

  #pragma omp atomic update
  x[i]++;
}
```

- See https://www.ibm.com/docs/en/zos/2.2.0?topic=SSLTBW_2.2.0/com.ibm.zos.v2r2.cbclx01/prag_omp_atomic.htm
- Also: https://www.openmp.org/spec-html/5.0/openmpsu95.html

Important things!

- Be clear on what variables needs to be PRIVATE, FirstPrivate and SHARED

- Make sure you print Threads and the starting of their chunks (after output image is created).

- Reference for OMP Tasks:

  - http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-task.html