CME 213, ME 339—Spring 2021

Eric Darve, ICME, Stanford

"Controlling complexity is the essence of computer programming." (Brian Kernigan)


OpenMP
Enabling HPC since 1997

C++ threads are great for low-level multicore programming

But too general and complicated for engineering applications

Two common scenarios

1. `For` loop: partition the loop into chunks
   Have each thread process one chunk.
2. Hand-off a block of code to a separate thread

OpenMP simplifies the programming significantly.

In many cases, adding one line is sufficient to make it run in parallel.

OpenMP is the standard approach in scientific computing for multicore processors

What is OpenMP?

Application Programming Interface (API)

Jointly defined by a group of major computer hardware and software vendors

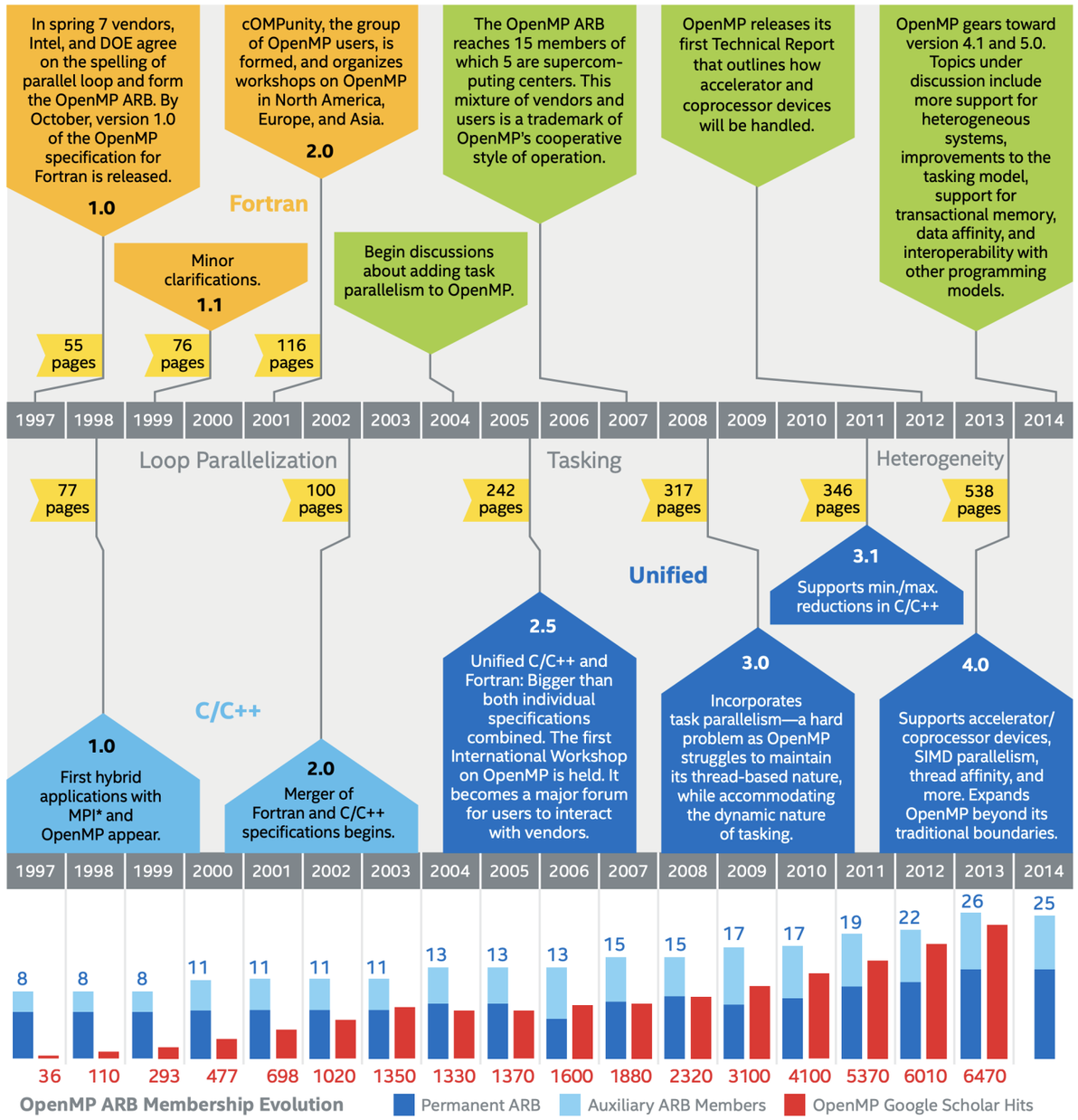
Portable, scalable model for developers of shared memory parallel applications

Supports C/C++ and Fortran on a wide variety of computers

OpenMP website
https://openmp.org

Wikipedia
https://en.wikipedia.org/wiki/OpenMP

LLNL tutorial
https://computing.llnl.gov/tutorials/openMP/


In spring 7 vendors, Intel, and DOE agree on the spelling of parallel loop and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released.
1.0
cOMPunity, the group of OpenMP users, is formed, and organizes workshops on OpenMP in North America, Europe, and Asia.
2.0
The OpenMP ARB reaches 15 members of which 5 are supercomputing centers. This mixture of vendors and users is a trademark of OpenMP's cooperative style of operation.
OpenMP releases its first Technical Report that outlines how accelerator and coprocessor devices will be handled.
OpenMP gears toward version 4.1 and 5.0. Topics under discussion include more support for heterogeneous systems, improvements to the tasking model, support for transactional memory, data affinity, and interoperability with other programming models.
Fortran
Minor clarifications.
1.1
Begin discussions about adding task parallelism to OpenMP.
55 pages
76 pages
116 pages
1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014
Loop Parallelization
Tasking
Heterogeneity
77 pages
100 pages
242 pages
317 pages
346 pages
538 pages
Unified
3.1
Supports min./max. reductions in C/C++
2.5
Unified C/C++ and Fortran: Bigger than both individual specifications combined. The first International Workshop on OpenMP is held. It becomes a major forum for users to interact with vendors.
3.0
Incorporates task parallelism—a hard problem as OpenMP struggles to maintain its thread-based nature, while accommodating the dynamic nature of tasking.
4.0
Supports accelerator/coprocessor devices, SIMD parallelism, thread affinity, and more. Expands OpenMP beyond its traditional boundaries.
C/C++
1.0
First hybrid applications with MPI* and OpenMP appear.
2.0
Merger of Fortran and C/C++ specifications begins.
1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014
8 8 8 11 11 11 11 13 13 13 15 15 17 17 19 22 26 25
36 110 293 477 698 1020 1350 1330 1370 1600 1880 2320 3100 4100 5370 6010 6470
OpenMP ARB Membership Evolution
Permanent ARB
Auxiliary ARB Members
OpenMP Google Scholar Hits

Parallel Universe Magazine Issue 18

Download PDF of paper

Compiling your code

Header file:

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

Compiler	Flag
gcc g++ g77 gfortran	`-fopenmp`
icc icpc ifort	`-openmp`

Installation on macOS

Option 1: `libomp`

```
$ brew install libomp
```

Use the system compiler `/usr/bin/g++`

Compile with options

```
-I/usr/local/include -Xpreprocessor -fopenmp
-L/usr/local/lib -lomp
```

Option 2: `gcc`

```
$ brew install gcc
```

Compiler: `/usr/local/bin/g++-9`

Flag: `-fopenmp`

Which version of openMP do you have?

This determines the set of features available

```
$ echo | cpp -I/usr/local/include -Xpreprocessor -fopenmp -dM | grep OPENMP
#define _OPENMP 201511
```

Mapping

Year	OpenMP version
200505	2.5
200805	3.0
201107	3.1
201307	4.0
201511	4.5
201811	5.0

Parallel regions

`#pragma omp parallel`

This is the basic building block

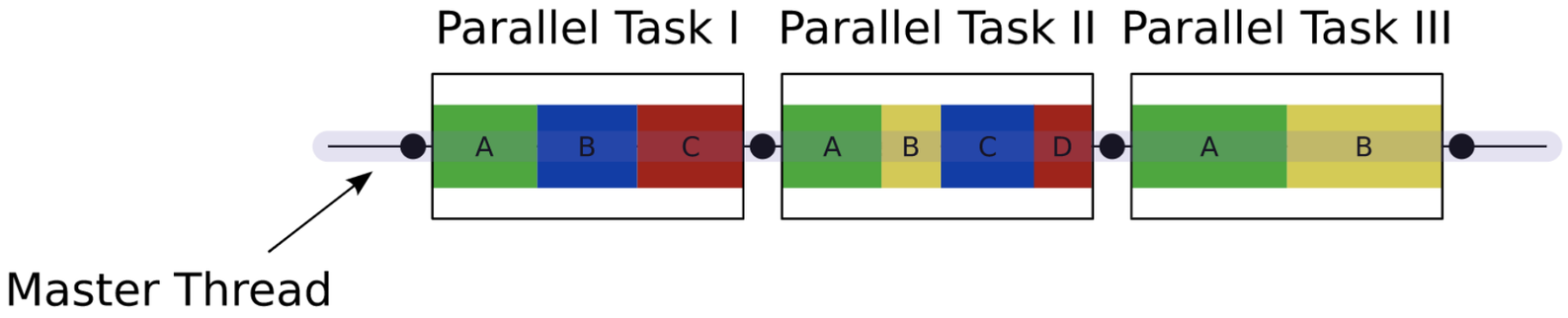
This is not how OpenMP is used in most cases

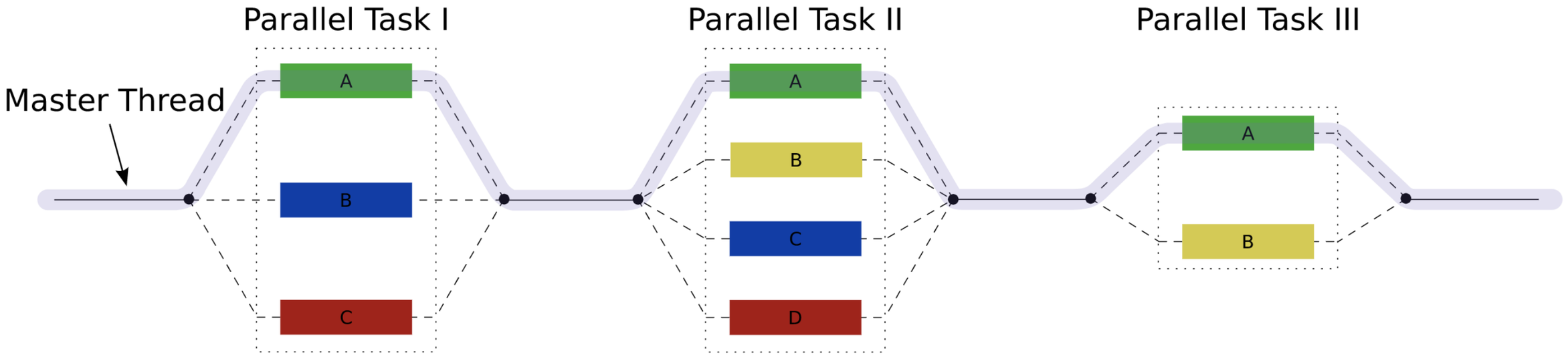
Serves simply as an introduction

```
#pragma omp parallel
```

The block of code that follows is executed

by all threads in the team

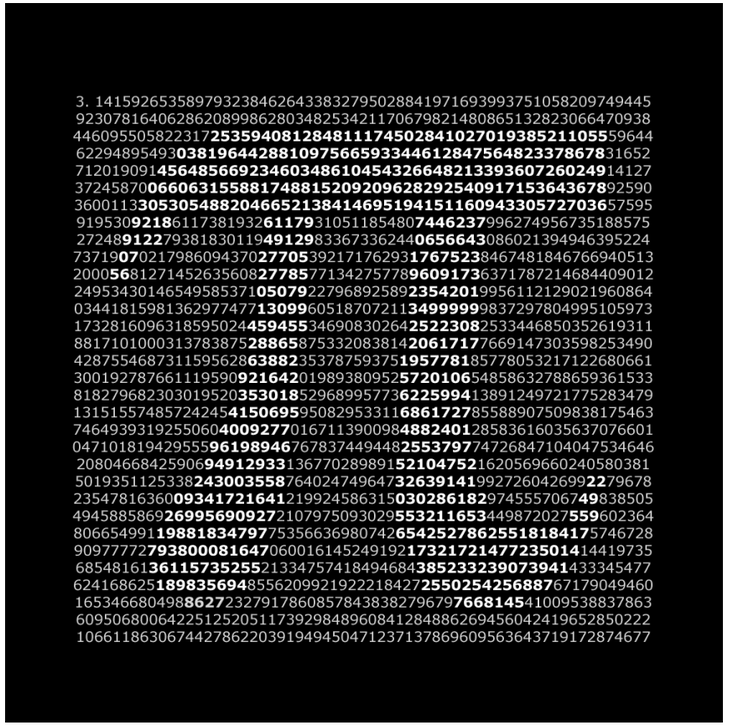

Parallel Task I
Parallel Task II
Parallel Task III
A
B
C
A
B
C
D
A
B
Master Thread


Parallel Task I
Parallel Task II
Parallel Task III
Master Thread
A
B
C
A
B
C
D
A
B

This is called the

fork-join model

Computing π



[hello_world_openmp.cpp]

```c
#pragma omp parallel num_threads(nthreads)
{
    long tid = omp_get_thread_num();
    // Only thread 0 does this
    if (tid == 0)
    {
        int n_threads = omp_get_num_threads();
        printf("[info] Number of threads = %d\n", n_threads);
    }
    // Print the thread ID
    printf("Hello World from thread = %ld\n", tid);

    // Compute digits of pi
    DoWork(tid, ndigits[tid], etime[tid]);
}
// All threads join the master thread and terminate
```

Choose your compiler in Makefile

```
$ make
```

```
$ ./hello_world_openmp
```

Sample output

```
Let's compute pi =              3.14159265358979323846264338327950288841...
[info] Number of threads = 8
Hello World from thread = 0
Thread  0 approximated Pi as     3141592653589793
Hello World from thread = 1
Thread  1 approximated Pi as     3141592653589793238462643
...
Thread  7 approximated Pi as     3141592653589793238462643383279502884l...
Thread 0 computed 16 digits of pi in    67 musecs (   4.188 musec per digit)
Thread 1 computed 24 digits of pi in    16 musecs (   0.667 musec per digit)
...
Thread 7 computed 72 digits of pi in    68 musecs (   0.944 musec per digit)
```

Formula used to approximate π

$$\frac{\pi}{2} = 1 + \frac{1}{3}\left(1 + \frac{2}{5}\left(1 + \frac{3}{7}\left(1 + \frac{4}{9}\left(1 + \cdots\right)\right)\right)\right)$$

Common use case: `for` loop

This example cover 99% of the needs for scientific computing

for_loop_openmp.cpp

```
#pragma omp parallel for
    for (int i = 0; i < n; ++i)
        z[i] = x[i] + y[i];
```

Exercise

matrix_prod_openmp.cpp

Parallelize the matrix-matrix product

Experiment with different options

`./matrix_prod_openmp -p PROC`

PROC number of threads to use

`for` loops can be scheduled in different ways by the library

```
#pragma omp for schedule(kind,chunk_size)
```

kind: `static`, `dynamic`, `guided`

`static`

chunk size is fixed (`chunk_size`)

round-robin assignment

0, 1, 2, 3, 0, 1, 2, 3, ...

Pro: low overhead

Con: assumes that running time is the same for all chunks

`dynamic`

Each thread executes a chunk

Then, requests another chunk until none remain

Pro: low overhead, adapts to threads that run at different speeds

Con: last thread may terminate long after the others

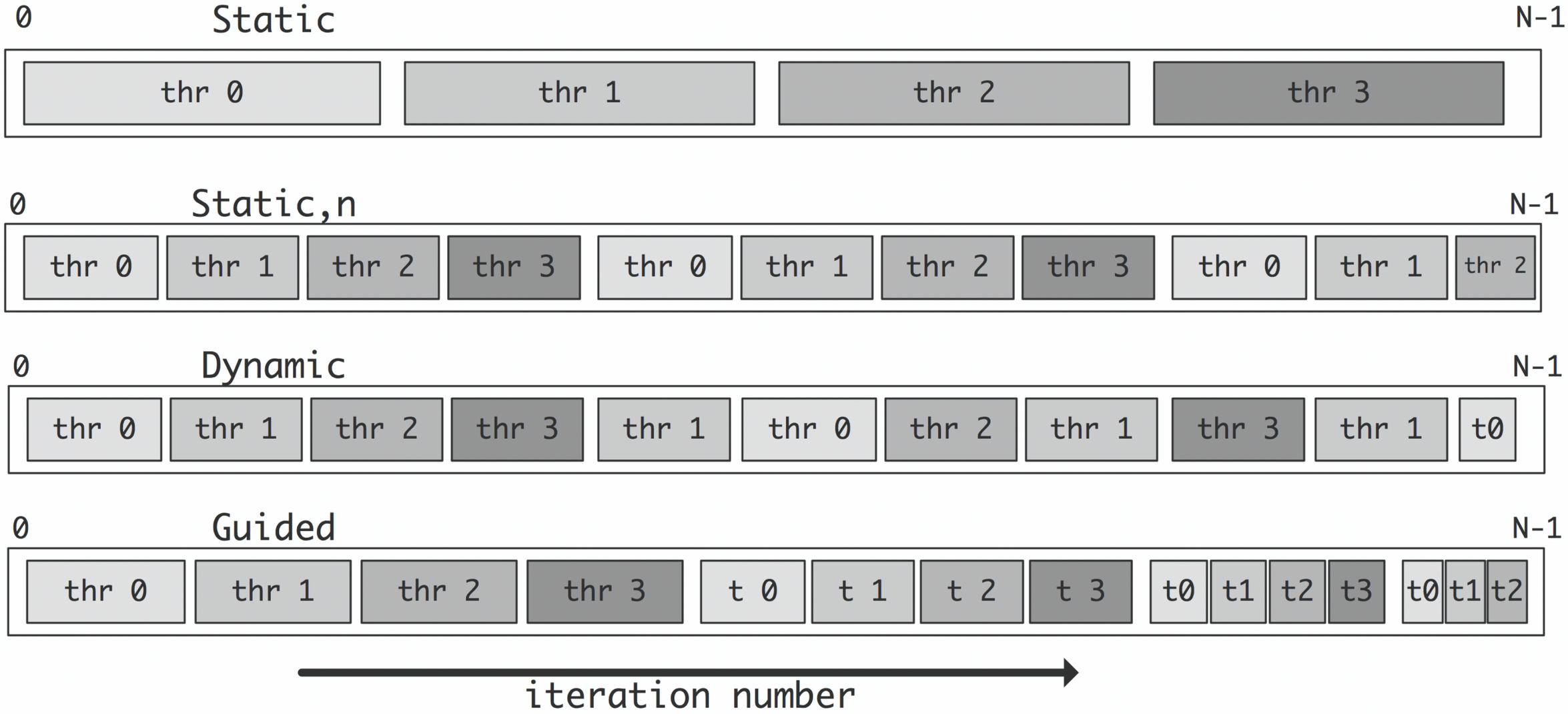
`guided`

Chunk size is different for each chunk

Each successive chunk is smaller than the last

Pro: all threads tend to finish at the same time

Con: high overhead for scheduling


0
Static
N-1
thr 0
thr 1
thr 2
thr 3
0
Static,n
N-1
thr 0
thr 1
thr 2
thr 3
thr 0
thr 1
thr 2
thr 3
thr 0
thr 1
thr 2
0
Dynamic
N-1
thr 0
thr 1
thr 2
thr 3
thr 1
thr 0
thr 2
thr 1
thr 3
thr 1
t0
0
Guided
N-1
thr 0
thr 1
thr 2
thr 3
t 0
t 1
t 2
t 3
t0
t1
t2
t3
t0
t1
t2
iteration number

nowait

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for nowait
    for (i=0; i<m; i++)
        y[i] = sqrt(z[i]);
}
```

collapse

```
#pragma omp for collapse(2) private(i, k, j)
    for (k=kl; k<=ku; k+=ks)
        for (j=jl; j<=ju; j+=js)
            for (i=il; i<=iu; i+=is)
                bar(a,i,j,k);
```

OpenMP clause

Recall in C++ threads:

Variables passed as argument to a thread are **shared**

Variables **inside the function** that a thread is executing are **private** to that thread

OpenMP makes some reasonable default choices

But they can be changed using `shared` and `private`

shared_private_openmp.cpp

Variables declared before the block are `shared`

```
int shared_int = -1;
#pragma omp parallel
{
    printf("Thread ID %2d | shared_int = %d\n", omp_get_thread_num(),
            shared_int);
}
```

`private` clause

```
int is_private = -2;

#pragma omp parallel private(is_private)
{
    const int rand_tid = rand();
    is_private = rand_tid;
    printf("Thread ID %2d | is_private = %d\n", omp_get_thread_num(),
           is_private);
    assert(is_private == rand_tid);
}
```

Data sharing attribute clause

Most common:

- `shared(list)`
- `private(list)`

Less common: `firstprivate`, `lastprivate`, `linear`

`firstprivate`: private variable; initialized using value when construct is encountered

`lastprivate`: set equal to the private version of whichever thread executes the final iteration

`linear`: see p. 25 of specifications; variable is private and has a linear relationship with respect to the iteration space of a loop associated with the construct

https://www.openmp.org/spec-html/5.1/openmp.html

PDF