CMOR 421/521
Shared memory parallelism

Jesse Chan

Some basic parallel theory

Computing trends

Slower processors but more of them

- Trend is towards slower CPUs
  - Higher CPU clock speeds generate more heat and consume more power
  - Supercomputers consume huge amounts of energy
  - Some of that energy is from computing, some is from cooling; industry is experimenting with hot server rooms, water cooling, etc because of this
- Easier to manufacture: fast CPUs are more sensitive to manufacturing defects, easier to fabricate lots of slower CPUs instead.

Challenge of parallelism

- Not all work is parallelizable; “9 women can’t make a baby in 1 month”
- If a small amount of work is serial / sequential, this limits scalability
- Suppose p, s are “parallelizable” and “serial” fractions of work ($s = 1 - p$).
  - Let n denote the number of processors.
- Total runtime of a program is $T(n) = T_s(n) + T_p(n)$
  - $T_s(n)$ is the time spent on the sequential portion
  - $T_p(n)$ is the time spent on the parallel portion

Some scaling estimates

- Total runtime of a program is $T(n) = T_s(n) + T_p(n)$
  - Assume sequential portion is independent of n, so $T_s(n) = sT(1)$
  - Assume parallel portion is perfectly parallelizable, so $T_p(n) = \frac{p}{n}T(1)$
- Total runtime is then $T(n) = \left(s + \frac{p}{n}\right) T(1).$
- Can analyze *scaling* behavior as the number of processors increases.

Amdahl's law: strong scaling

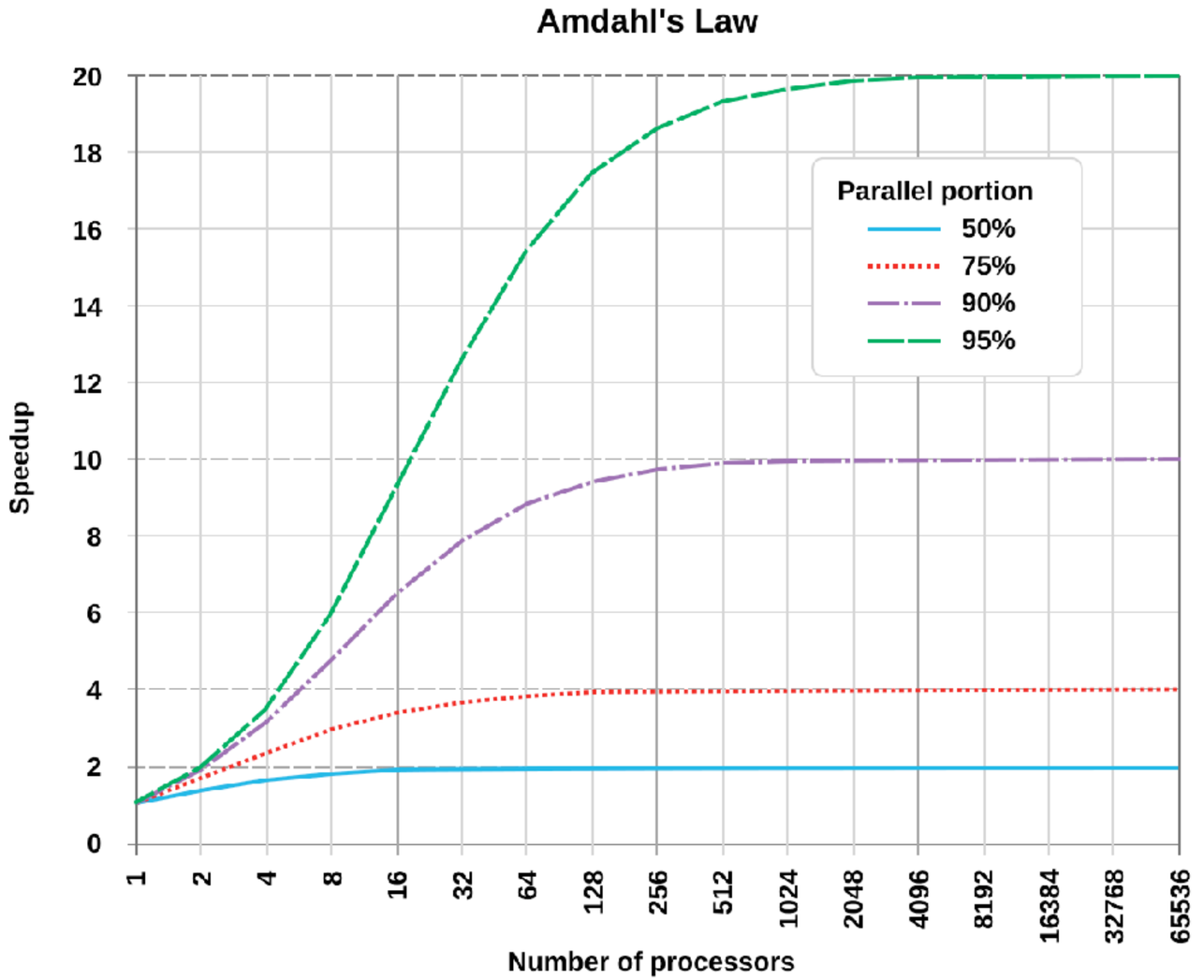
- There are different ways of measuring speed-up related to scaling
  - **Strong scaling**: amount of work is fixed
  - **Weak scaling**: amount of work scales with the number of processors
- For strong scaling, parallel speed-up $S(n) = \frac{T(1)}{T(n)}$ is the main metric
- Substitute in $T(n) = \left(s + \frac{p}{n}\right) T(1)$ to get Amdahl's law:
  - $S(n) = \frac{T(1)}{T(n)} = \frac{s+p}{s+p/n} = \frac{1}{s+p/n} < \lim_{n\to\infty} S(n) = \frac{1}{s}$

What does Amdahl's law mean?

- Parallel speed-up $S(n) = \frac{T(1)}{T(n)}$ is bounded by the sequential work
  - $S(n) = \frac{T(1)}{T(n)} = \frac{s+p}{s+p/n} = \frac{1}{s+p/n} < \lim_{n\to\infty} S(n) = \frac{1}{s}$
- Cannot expect arbitrarily large parallel speed-ups; eventually you will hit a sequential bottleneck.
  - Suppose only 50% of a program is parallelizable; maximum parallel speed-up is at most 2.
  - If 80% is parallelizable, then maximum parallel speed-up is 1 / .2 = 5.
  - Relatively modest speed-ups unless *everything* is parallelizable.

Amdahl’s law implies the sequential part matters

- Speeding up the sequential part asymptotically improves max speed-up by same factor
- Implies that the parallel *efficiency* $E(n) = \frac{S(n)}{n}$ must eventually decrease as n increases.
- Note: Amdahl assumes fixed parallel resources (potentially good), ignores overhead from synchronization (bad)



Weak scaling

- Strong scaling increases the number of processors n but keeps work fixed
- Weak scaling increases both number of processors and work together
  - Performance if work per processor is held constant
- Why weak scaling? For many programs, the sequential part does not increase as the problem size increases. Examples:
  - Monte Carlo simulations; minimal setup, “embarrassingly” parallel
  - Numerical simulations: often dominated by multiple matrix multiplies. Serial part typically negligible or scales very slowly with problem size.

Relating weak and strong scaling

- Weak scaling changes the assumptions of strong scaling. Suppose the total work depends on the problem size N
  - Total work: $W(N) = W_s(N) + W_p(N)$.
  - Recover serial/parallel parts: $1 = \frac{W_s(N)}{W(N)} + \frac{W_p(N)}{W(N)} = s_N + p_N$
- Strong scaling assumes that N is fixed; weak scaling assumes the serial part of the work $W_s(N)$ does not change with N.
- Only way this can happen is if s_N decreases and p_N increases with N.

Weak scaling: Gustafson's law

- Suppose the problem size is proportional to number of processors n
  - This implies that $T(1) = (s + np)T(n)$, e.g., cost on one processor is serial part plus n times the parallel part.
- Assume without loss of generality $T(n) = 1$.
  - Then parallel speed-up is $S(n) = \dfrac{T(1)}{T(n)} = \dfrac{s + np}{s + p} = s + np$
- In other words, if we increase the problem size with n, parallel speed-up increases *linearly* with respect to the number of processors.

Caveats of Gustafson's law

- Gustafson's law: under weak scaling, parallel speed-up is $S(n) = s + np$
  - Speed-up can be arbitrarily large as long as we grow the problem size!
  - Weak speed-up measures how much time it would take for one processor to do the work of n processors.
- Assumes that the serial part is independent of problem size
- Assumes that the cost scales *linearly* with respect to number of processors. This may not be true for algorithms which are not perfectly scalable or have nonlinear complexity.

Strong and weak scaling

- Both strong and weak scaling are useful measures of parallel efficiency
  - Strong vs weak scaling: “9 women can’t make a baby in 1 month, but could possibly make 81 babies in 9 months?”
  - Good weak scaling is usually easier to achieve.
  - Strong scaling is still important! Shows how large your problem size must be on each individual worker/processor.
- Both measures can be gamed!
  - Imagine if the parallelizable part of a program is scalable but implemented in a highly inefficient manner; strong / weak scaling limits may not be observed until the number of processors is very large.

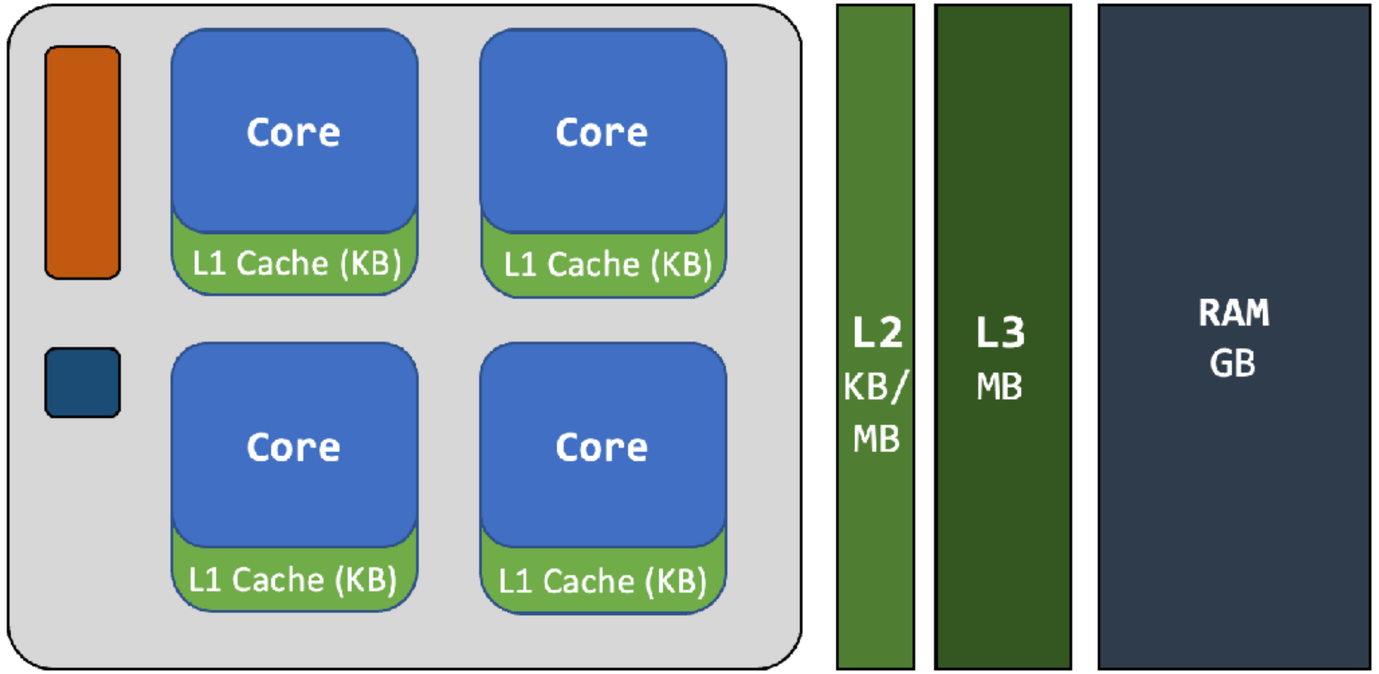
Parallelism and communication

- Parallel work can be split into **computation** and **communication**
- Parallel computation is usually pretty straightforward: divide work up (as evenly as possible) among multiple workers.
- However, there are several ways to implement communication among workers, each of which corresponds to a different parallel “paradigm”.
- We’ll focus on **shared memory** and **distributed memory** parallelism.
  - **Shared memory:** multiple workers communicate through centralized storage. Appropriate for multi-core CPUs (OpenMP) and GPUs.
  - **Distributed memory:** workers communicate by explicit *message-passing.* Appropriate for clusters, uses MPI: message passing interface

Shared memory parallelism and OpenMP

Shared memory programming: threads vs cores

- Cores/threads have shared memory that they all read from or write to.
- Cores are related to hardware, threads are related to software
  - One thread is similar to an independent subroutine
  - Threads get created and joined / destroyed when the program runs
  - Threads are independent from each other and communicate through shared variables and synchronization.



Pthreads: low level thread management

- **P**osix (Portable Operating System Interface) **threads**
- Requires manual thread management (creation/ joining of threads)
- Can avoid unnecessary thread creation, which has significant computational overhead.

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Compile using gcc –lpthread

Example from Aydin Buluc

OpenMP

- **Open** specification for **M**ulti-**P**rocessing.
  - Specification is determined by the OpenMP Architecture Review Board (ARB): a nonprofit specifically for OpenMP.
  - Actively managed and updated; OpenMP 6.0 released Nov. 2024.
  - See openmp.org for talks, examples, forums, etc.
- Motivation: capture common multi-threaded programming patterns and simplify implementation compared to POSIX-threads.
  - OpenMP syntax additionally aims to be “light” and relatively non-intrusive.

How does OpenMP work?

- Because it's a *specification*, OpenMP requires compiler-side implementation.
  - OpenMP typically converts OpenMP code to lower level Pthreads code.
  - Behavior laid out in the specification should be consistent across compilers.
- Main idea: rather than fork-joining individual threads, OpenMP introduced the idea of *serial and parallel regions.*
  - OpenMP hides thread management from the user and provides convenience methods for common thread synchronization/communication patterns.
- OpenMP *does not* guarantee automatic parallel speedup or memory safety issues (e.g., data race conditions).

An OpenMP “Hello world”

- #pragma (e.g., a directive or “pragmatic”) specifies an OpenMP parallel *region.*
  - Everything inside the parallel region is run on every single thread.
  - Threads execute in any order.
- The #pragma is a preprocessor directive; if a compiler doesn’t support OpenMP, it gets ignored.
- Can query for total (active) number of threads and local thread id number.

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

using namespace std;

int main()
{
#pragma omp parallel
  {
    cout << "Hello world " << endl;
  }
  return 0;
}
```

An OpenMP “Hello world”

- #pragma (e.g., a directive or “pragmatic”) specifies an OpenMP parallel *region.*
  - Everything inside the parallel region is run on every single thread.
  - Threads execute in any order.
- The #pragma is a preprocessor directive; if a compiler doesn’t support OpenMP, it gets ignored.
- Can query for total (active) number of threads and local thread id number.

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

using namespace std;

int main()
{
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    cout << "Hello world from thread " <<
      tid << " / " << omp_get_num_threads()
      << " threads" << endl;
  }
```

How to build an OpenMP program

- On NOTSx, compile OpenMP via “g++ **-fopenmp** hello_omp.cpp”
  - Need “include <omp.h>” to define OpenMP directives/functions.
  - “-fopenmp” links OpenMP headers/library but also includes compiler symbols necessary to process parallel directives.
- On your own machine, installing gcc/g++ should also install OpenMP
  - On Macs, can install gcc/g++ via Xcode command line tools or Homebrew.
  - I use Homebrew’s “g++-14” on my laptop; Apple’s “g++” redirects to a version of the “llvm” compiler which doesn’t support -fopenmp.
- Demo…

How to configure OpenMP threads

- How to configure the number of threads? Three (four?) ways:
  - Within an OpenMP program:
    - “omp_set_num_threads(4)” sets the number of threads globally
    - “#pragma omp parallel num_threads(4)” sets the number of threads for a specific parallel region
  - Outside of an OpenMP program using environment variables: “export OMP_NUM_THREADS = …”
- On NOTSx:

```
srun --pty --partition=scavenge --reservation=cmor421 –
ntasks=1 –cpus-per-task=4 –mem=1G --time=00:30:00 $SHELL
```

Example: setting the number of threads

- When setting the number of threads in multiple places, there is a priority.
  - The “num_threads(…)” clause takes precedence
  - Next is the number of threads set by “omp_set_num_threads(…)”
  - Last is the “OMP_NUM_THREADS” environment variable.

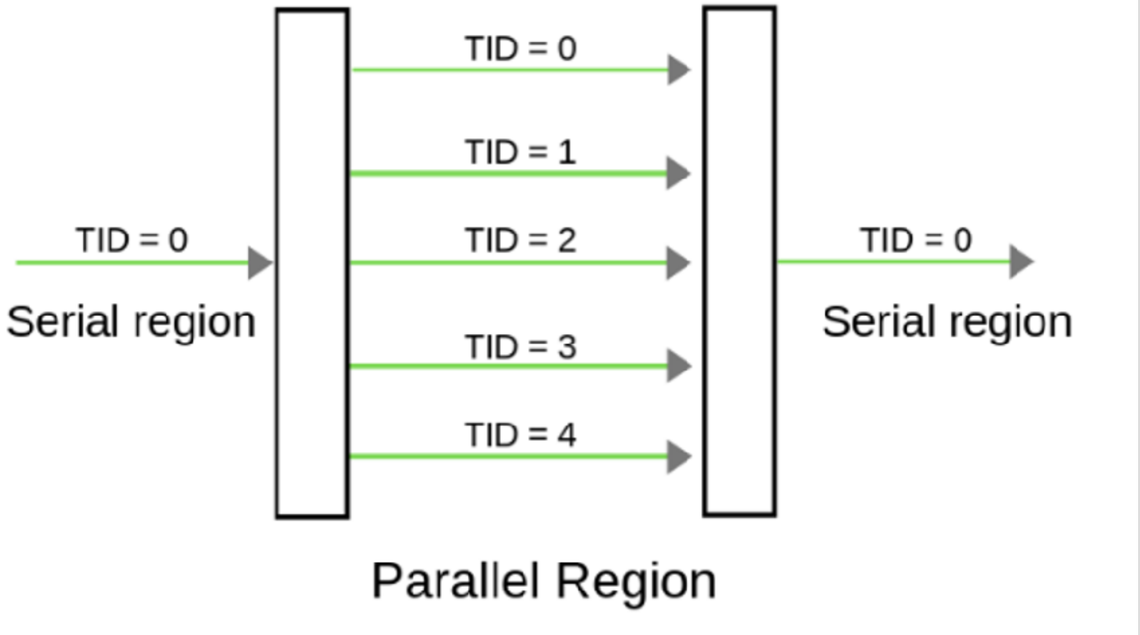
```
int main()
{
  omp_set_num_threads(1);

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    cout << "Hello world from thread " <<
      tid << " / " << omp_get_num_threads()
      << " threads" << endl;
  }

#pragma omp parallel num_threads(2)
  {
    int tid = omp_get_thread_num();
    cout << "Hello world again from thread " <<
      tid << " / " << omp_get_num_threads()
      << " threads" << endl;
  }
  return 0;
```

“How many threads” vs “how many cores"?

- Recall cores are hardware units and threads are software units. However, you can have fewer threads than cores, or more threads than cores.
  - Fewer threads than cores: cores sit idle
  - More threads than cores: cores process some threads threads, then switch to processing other threads.
- Many more threads than cores can require switching processes often, resulting in higher overhead.
- Rule of thumb: aim for more threads than cores, with the number of threads up to 2x the number of cores.

OpenMP's model of parallelism

- Specifying a parallel *region* instead of individual threads (like Pthreads)
- The thread with ThreadId = 0 is referred to as a "master" thread, and persists in both the serial and parallel regions
- Instead of assigning work to specific threads, OpenMP assumes the same code is executed on every thread.



https://www.geeksforgeeks.org/openmp-hello-world-program/

Fork-join parallelism

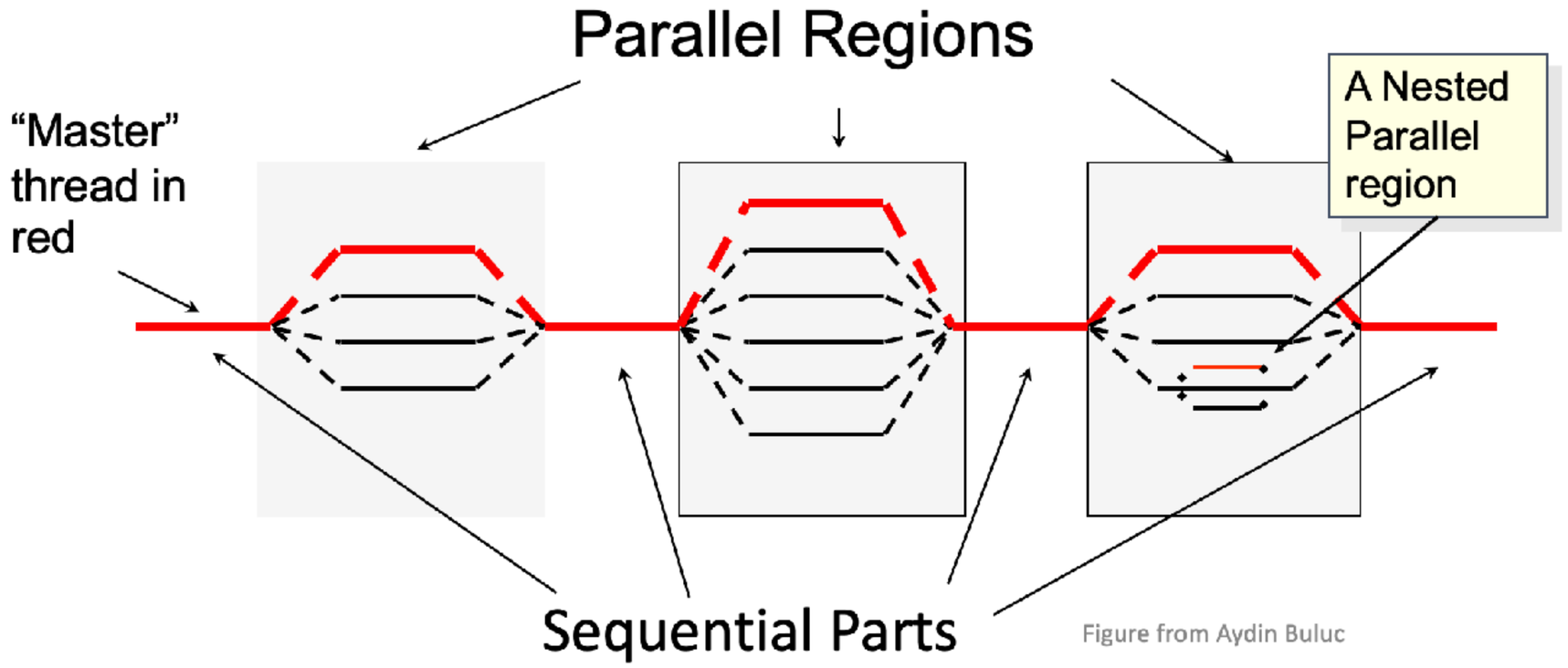
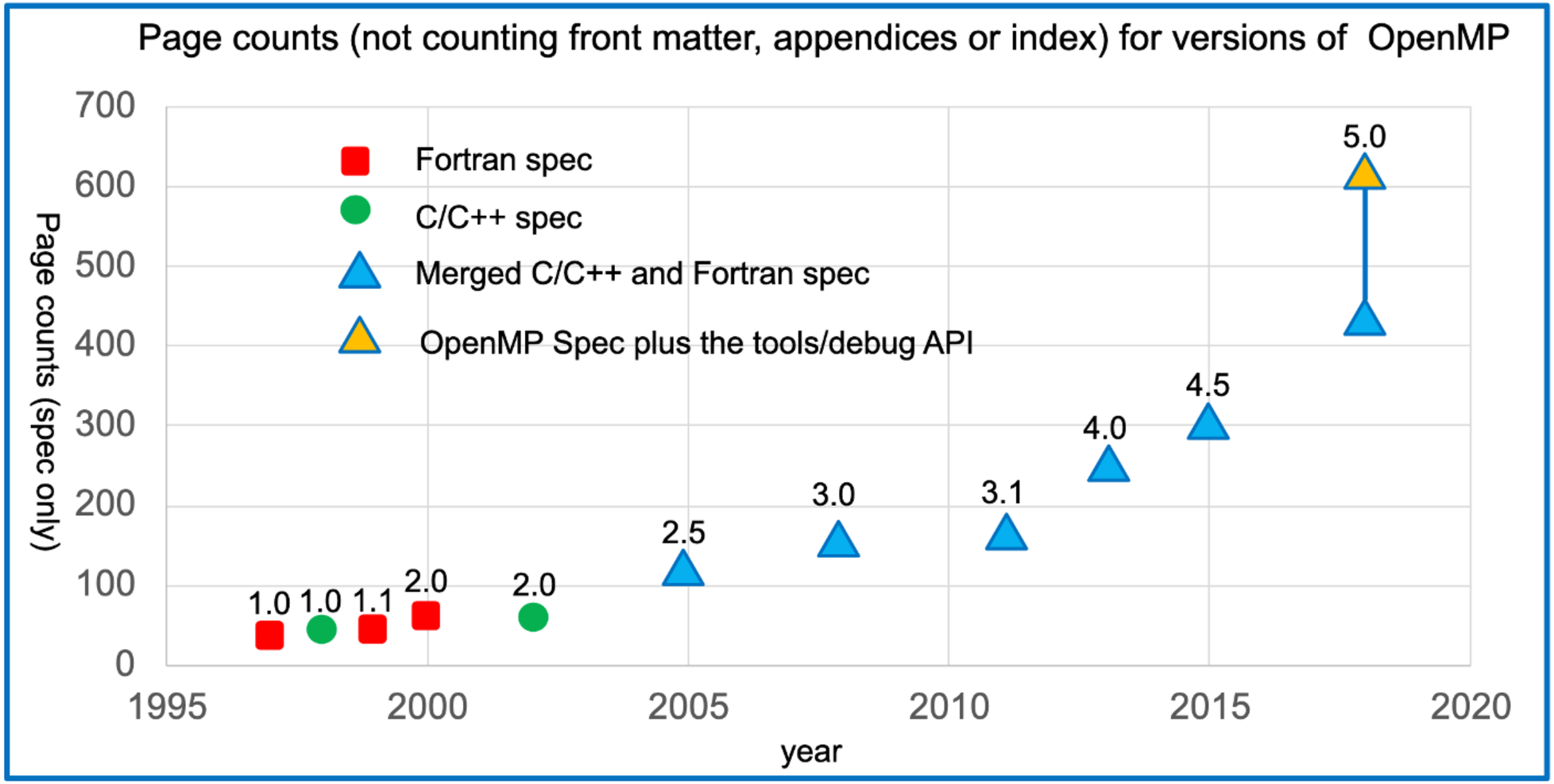


Figure from Aydin Buluc

- OpenMP is flexible with respect to constructing parallel regions.
- Different numbers of threads per parallel region, nested parallel regions, dynamic thread allocation (not shown here) are all possible.

What else can we do? Quite a lot

OpenMP is huge! However, not all the functionality is necessary



Most OpenMP programs use ~20 directives

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.