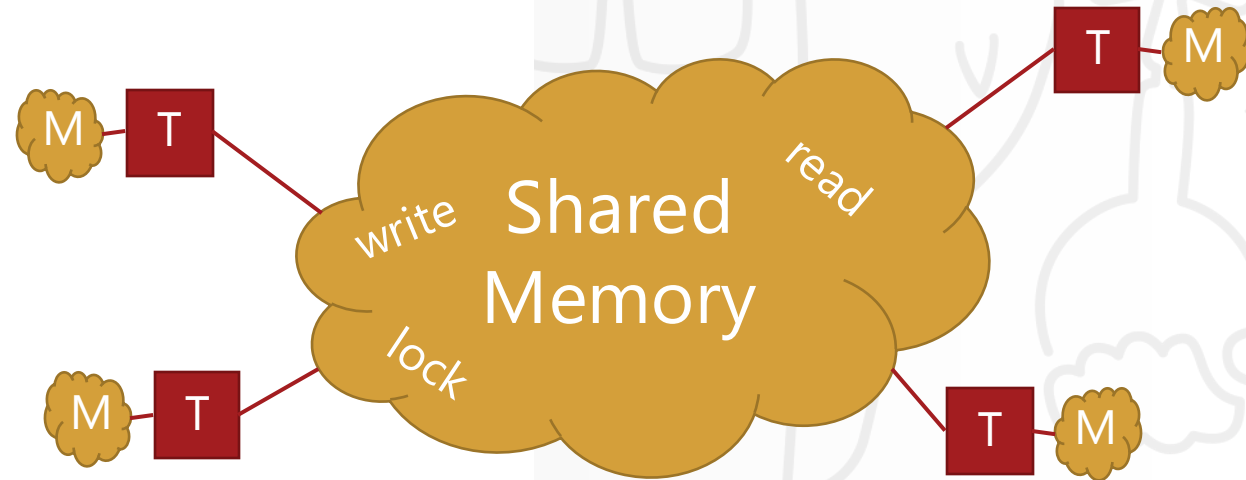


Introduction to OpenMP

Troels Haugbølle
haugboel@nbi.ku.dk

February 26, 2025



Overview

Today:

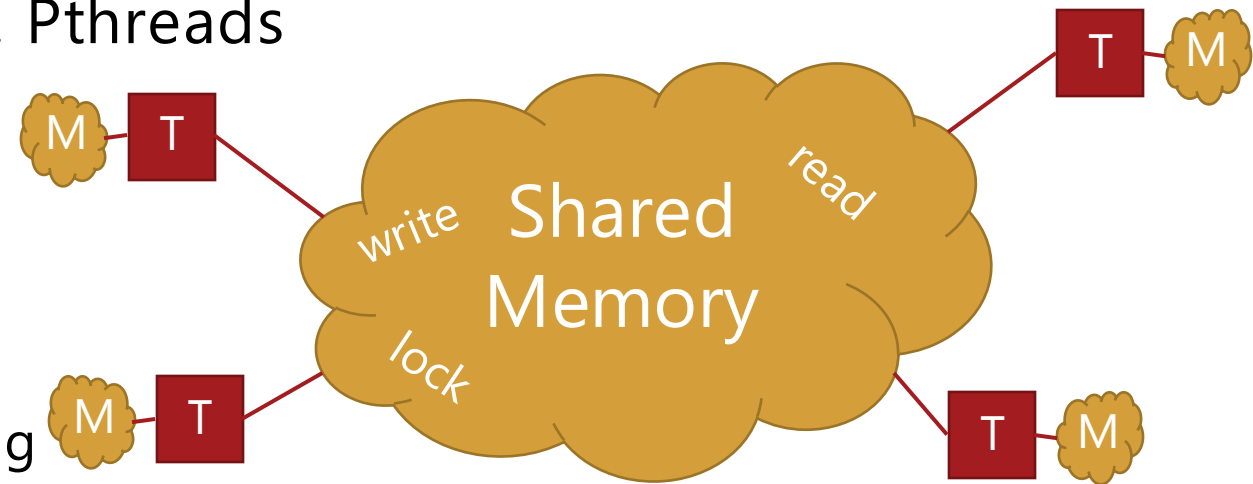
- Introduction to OpenMP, Synchronization
- Parallel Loops, Reduction
- Memory model: critical regions, atomic operations
- Tasks and Dynamic Parallelism
- Loop reordering and combining with omp simd
- Tools and references

Learning Objectives for this module:

- UMA, ccNUMA, and MP shared memory, limitations of shared memory
- Threads: different thread packages, working with threads, and coordination
- Hands-on experience with OpenMP

What is OpenMP?

- A set of compiler directives, library routines, and environment variables for shared memory programming in C, C++, and Fortran
- Simplifies writing multi-threaded programs
- Open standard defined in 1997 to merge a plethora of proprietary options
- Has grown to cover SMPs, vectorization and accelerators
- Handles the boilerplate code of e.g. Pthreads
 - Fork and joining of threads
 - Thread Pool, distribution of work
- Inconveniences:
 - Can (will!) lead to race conditions
 - Hard to get good performance & scaling
 - OpenMP parallelize the loops you tell it to indiscriminately



What is OpenMP?

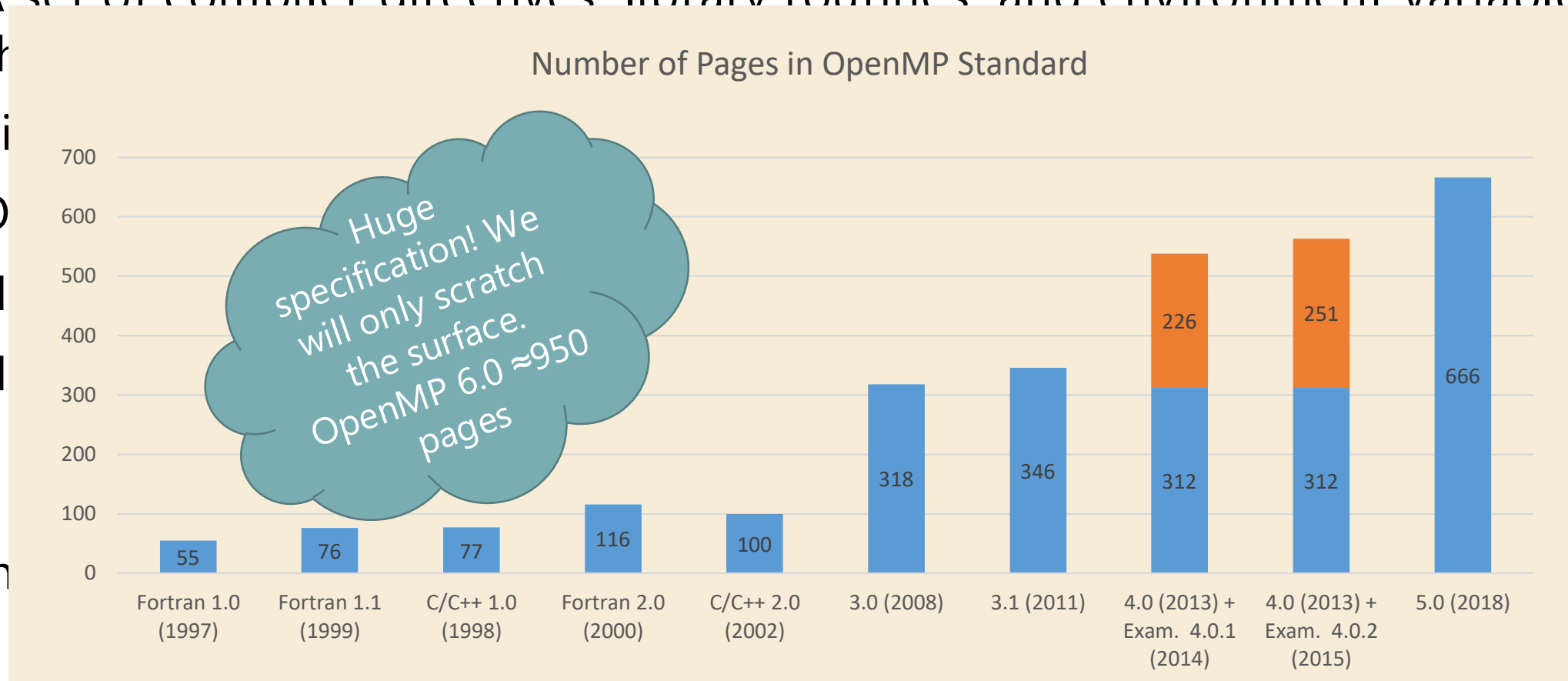
- A set of compiler directives, library routines, and environment variables for sharing

- Similar to MPI
- OpenMP is a standard

- Has a huge specification
- Hard to get good performance & scaling

- Hard to get good performance & scaling
- OpenMP parallelize the loops you tell it to indiscriminately

- In OpenMP 6.0, the specification is ~950 pages

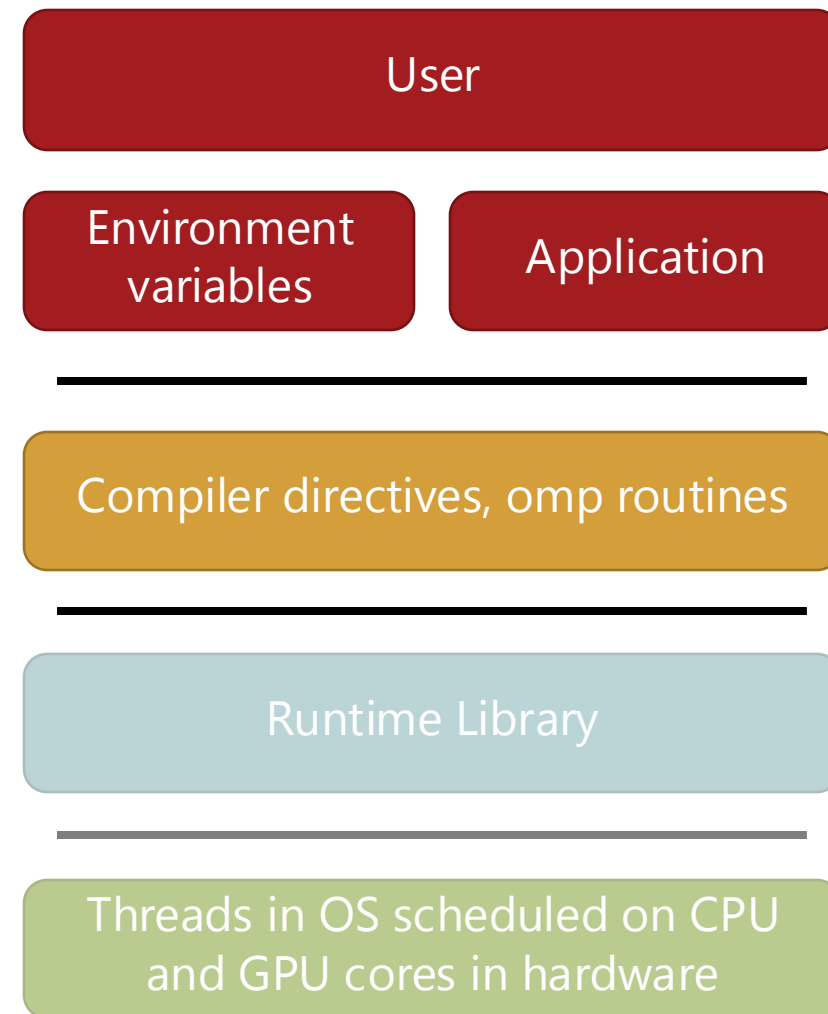


- Hard to get good performance & scaling

- OpenMP parallelize the loops you tell it to indiscriminately

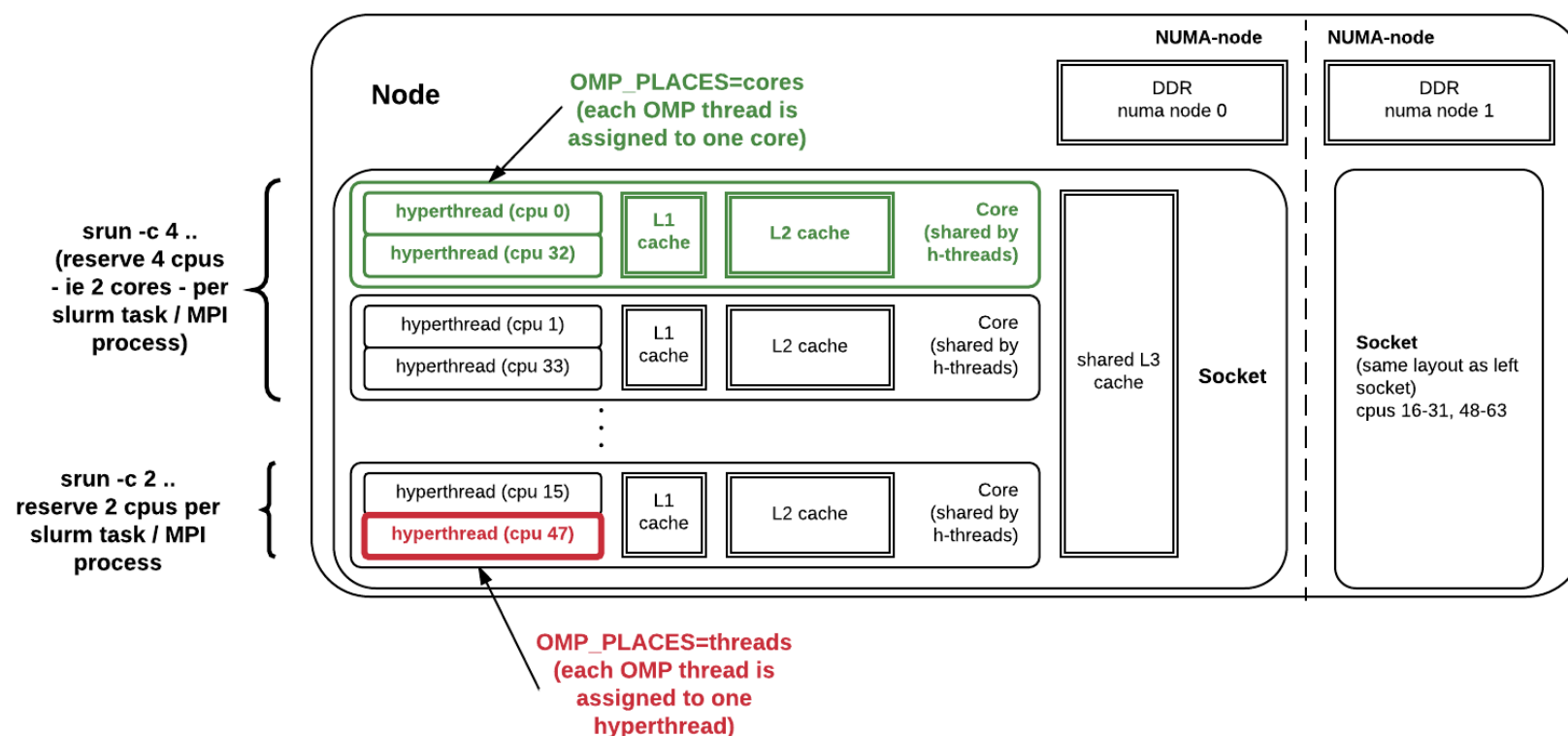
OpenMP Stack Overview

- User view: Environment variables set
 - How many threads to use: `OMP_NUM_THREADS=2`
 - Private memory per thread: `OMP_STACKSIZE=2m`
 - Affinity between threads and cores: `OMP_PLACES=cores`
 - How to bind to cores: `OMP_PROC_BIND=close`
 - Display information: `OMP_DISPLAY_AFFINITY=true`
- Developer view:
 - Directives: comments in code to instruct parallelism
 - `#pragma omp parallel, critical, atomic, barrier, ...`
 - Library routines for settings and info:
 - `omp_get_num_threads()`
 - `omp_get_thread_num()`
 - ...
- Low-level view:
 - OpenMP Runtime library: Schedule threads to do parallel work



OpenMP Stack Overview

- User view: Environment variables set
 - How many threads to use: OMP_NUM_THREADS=2
 - Private memory per thread: OMP_STACKSIZE=2m
 - Affinity between threads and cores: OMP_PLACES=cores
 - How to bind to cores: OMP_PROC_BIND=close



User

Environment
variables

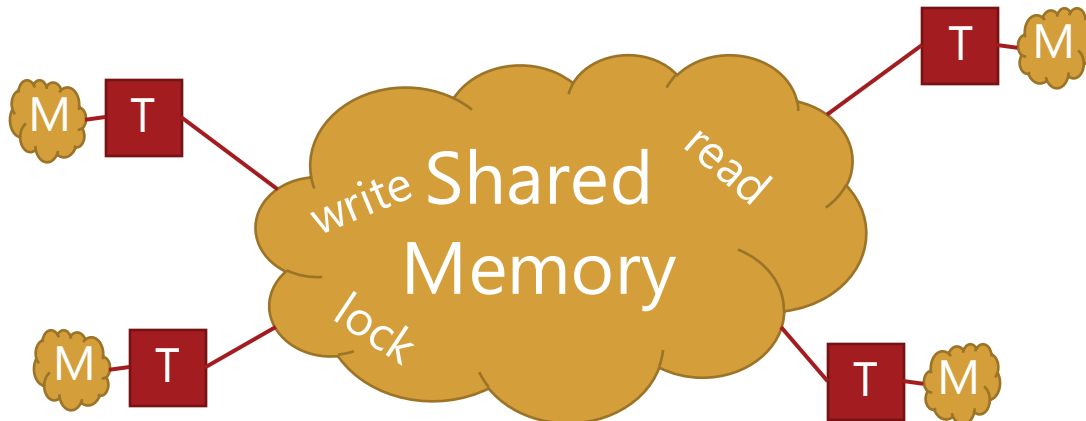
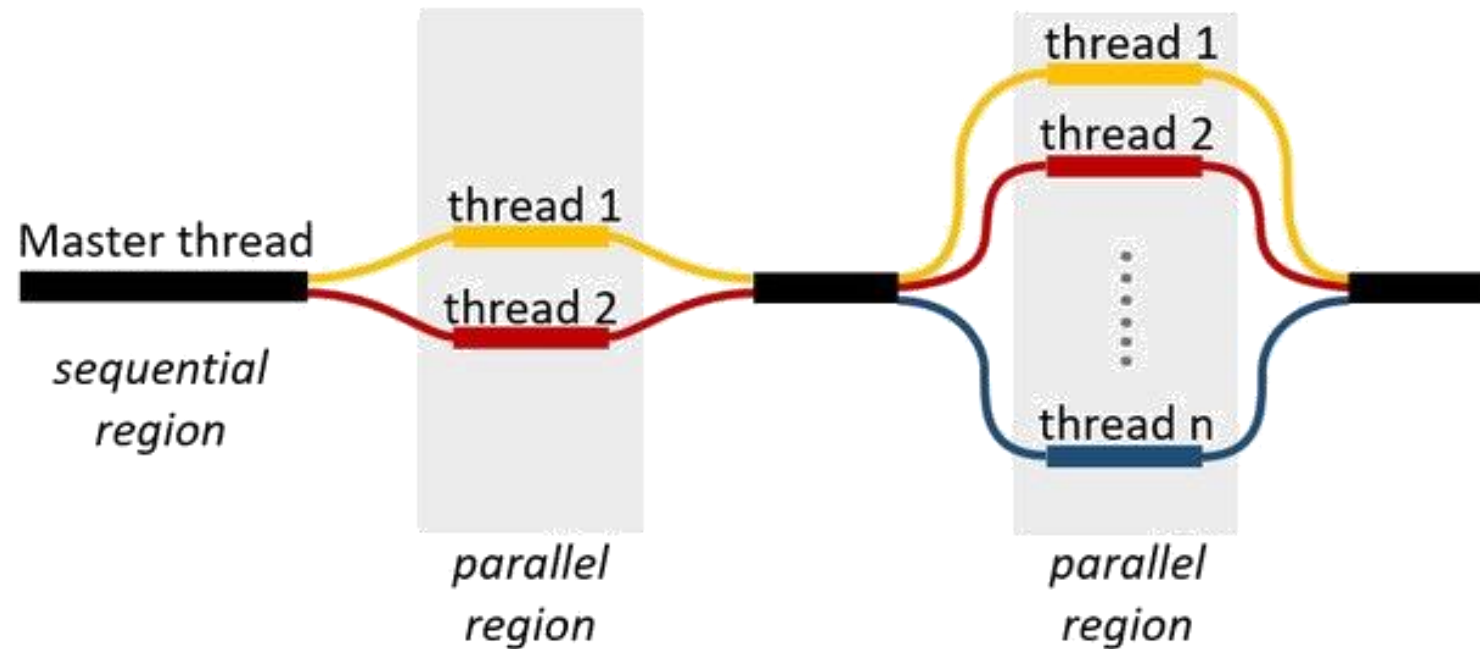
Application

mpiler directives, omp routines

Runtime Library

reads in OS scheduled on CPU
and GPU cores in hardware

Memory and fork-join execution model



- Program start: only master thread runs
- Parallel region: generate team of threads ("fork")
- Threads synchronize and leave parallel region ("join")
- Only master executes sequential region
- Directives
 - Task & data distribution
 - Synchronization
 - Data dependencies

parallel for
critical / atomic
reductions

OpenMP basic syntax

- Directives (C / C++):
`#pragma omp construct [clause ...]`
- Access to functions:
`#include <omp.h>`
- Most OpenMP constructs apply to a "structured block":
 - a block of one or more statements with one point of entry at the top and one point of exit at the bottom
- Compilation:
gcc: `g++ -fopenmp`
Nvidia: `nvc++ -mp`
Intel: `icpx -fopenmp`
LLVM: `clang++ -fopenmp`

Parallel Hello World

```
#include <iostream>
#include <omp.h>
int main()
{
    #pragma omp parallel {
        std::cout << " Hello ";
        std::cout << " World " << std::endl;
    }
}
```

Thread creation: parallel region

- Threads are created in OpenMP with the parallel construct
- Each thread executes a copy of the code within the structured block marked by the parallel construct
- Example shows how to use runtime functions to
 - request 3 threads
 - print thread id

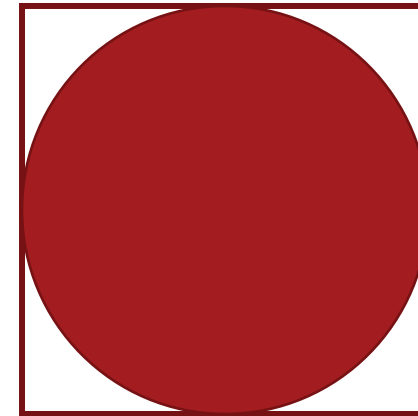
Thread creation

```
#include <iostream>
#include <omp.h>
int main()
{
    omp_set_num_threads(3);
    #pragma omp parallel {
        int tid = omp_get_thread_num();
        std::cout << tid << std::endl;
    }
}
```

Example: Monte-Carlo parallel integration of Pi

```
int main(int argc, char *argv[]) {  
    long n = 10000000;  
    long hits = 0;  
    for(long i=0; i<n; ++i) {  
        double x = randFloat()  
        double y = randFloat()  
        if (x*x + y*y <= 1) {  
            ++hits;  
        }  
    }  
    double pi = (hits / (double) n) * 4;  
}
```

randFloat
threadsafe by
construction



Area square: 4
Area circle: π

```
float randFloat() {  
    thread_local static std::random_device rd;  
    thread_local static std::mt19937 rng(rd());  
    thread_local std::uniform_real_distribution<float> urd;  
    return urd(rng, decltype(urd)::param_type(0,1));  
}
```

OpenMP: Monte-Carlo parallel integration of Pi

```
int main(int argc, char *argv[]) {  
    long n = 10000000;  
    long hits = 0;  
    → #pragma omp parallel for  
    for(long i=0; i<n; ++i) {  
        double x = randFloat()  
        double y = randFloat()  
        if (x*x + y*y <= 1) {  
            ++hits;  
        }  
    }  
    double pi = (hits / (double) n) * 4;  
}
```

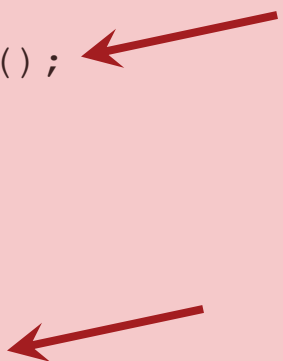
Q: is this
correct?

Running on 48 cores

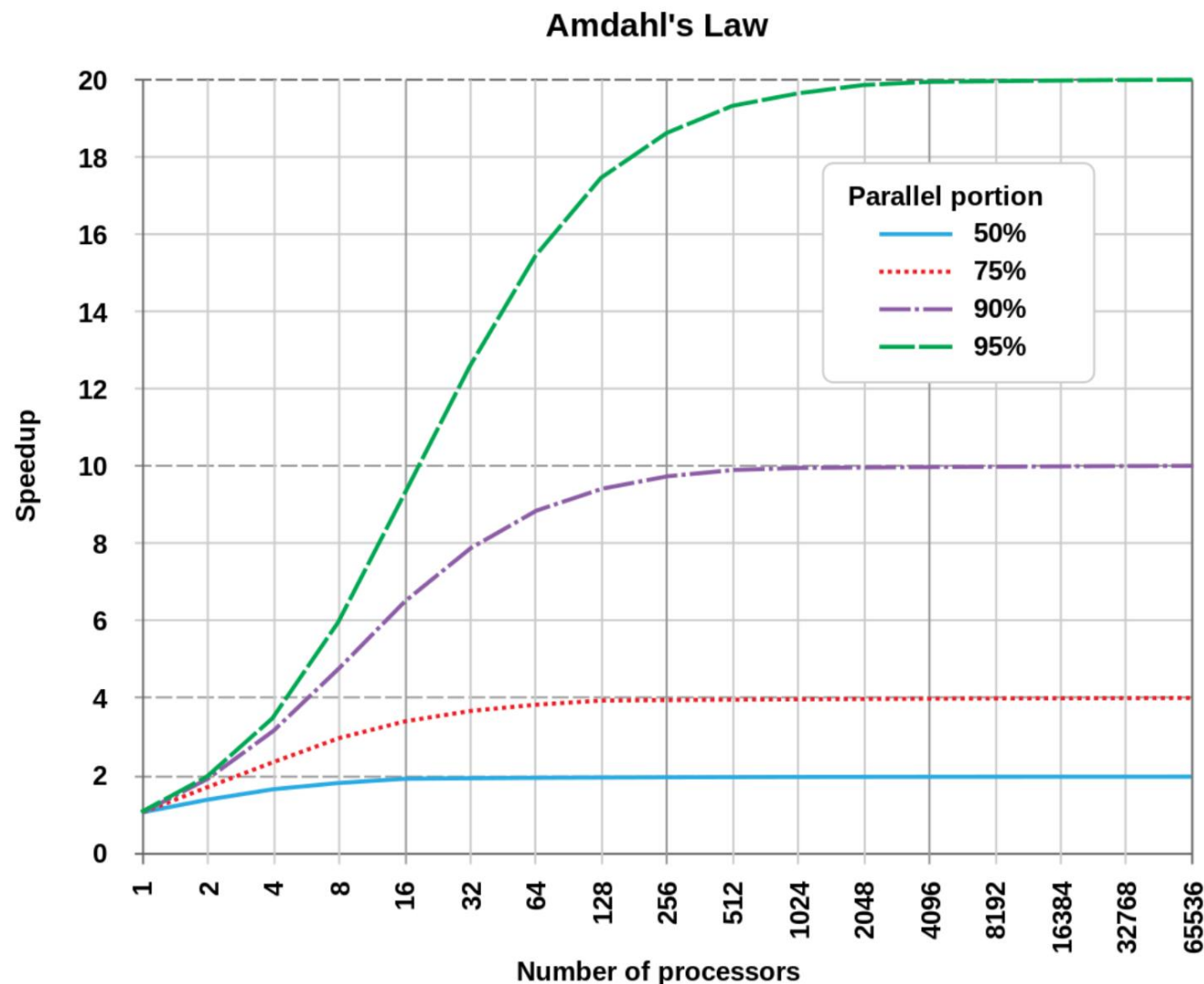
Value of pi 2.82052
Value of pi 3.0675
Value of pi 2.89714
Value of pi 3.00631
Value of pi 2.89751
Value of pi 3.03009

Protect access to hits with critical region (mutex / locks in pthreads)

```
int main(int argc, char *argv[]) {
    long n = 100000000;
    long hits = 0;
    double time = omp_get_wtime();
    #pragma omp parallel for
    for(long i=0; i<n; ++i) {
        double x = randFloat()
        double y = randFloat()
        if (x*x + y*y <= 1) {
            #pragma omp critical {
                ++hits;
            }
        }
    }
    time = omp_get_wtime() - time;
    double pi = (hits / (double) n) * 4;
    std::cout << "Time to integrate pi " << time << std::endl;
}
```



...alas, not super scalable



env OMP_NUM_THREADS=1 OMP_PLACES=cores ./pi
Time to integrate pi 1.62036

env OMP_NUM_THREADS=2 OMP_PLACES=cores ./pi
Time to integrate pi 5.76748

env OMP_NUM_THREADS=4 OMP_PLACES=cores ./pi
Time to integrate pi 6.02308

env OMP_NUM_THREADS=8 OMP_PLACES=cores ./pi
Time to integrate pi 5.87121

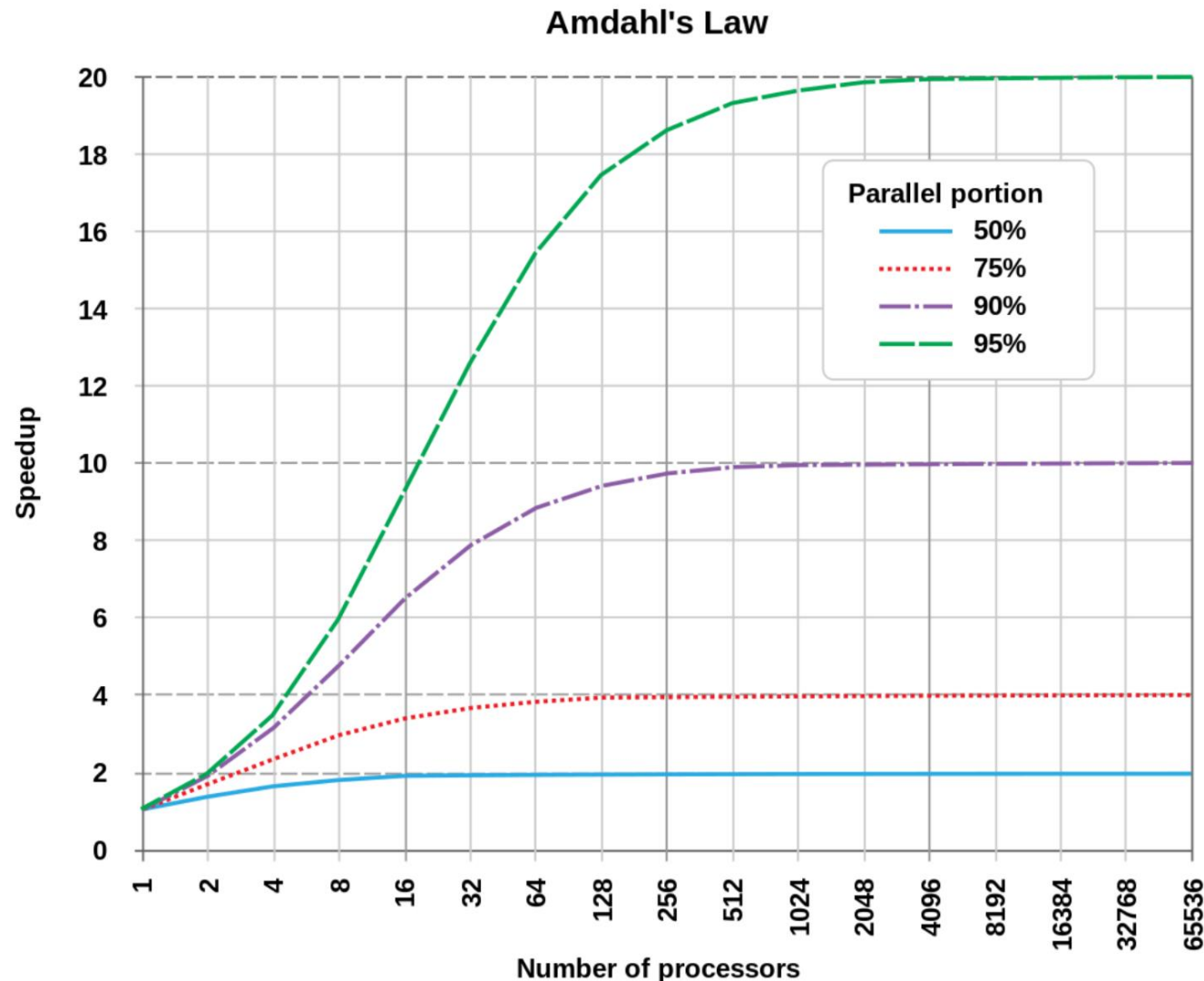
env OMP_NUM_THREADS=16 OMP_PLACES=cores ./pi
Time to integrate pi 6.10231

Speedup at 16 cores: 0.2x
Lock and memory contention!

Protect access to hits with hardware atomic instruction

```
int main(int argc, char *argv[]) {
    long n = 10000000;
    long hits = 0;
    double time = omp_get_wtime();
    #pragma omp parallel for
    for(long i=0; i<n; ++i) {
        double x = randFloat()
        double y = randFloat()
        if (x*x + y*y <= 1) {
            #pragma omp atomic ←
            ++hits;
        }
    }
    time = omp_get_wtime() - time;
    double pi = (hits / (double) n) * 4;
    std::cout << "Time to integrate pi " << time << std::endl;
}
```

...at least not decreasing performance for increasing #cores...



env OMP_NUM_THREADS=1 OMP_PLACES=cores ./pi
Time to integrate pi 1.47039

env OMP_NUM_THREADS=2 OMP_PLACES=cores ./pi
Time to integrate pi 1.45582

env OMP_NUM_THREADS=4 OMP_PLACES=cores ./pi
Time to integrate pi 1.2096

env OMP_NUM_THREADS=8 OMP_PLACES=cores ./pi
Time to integrate pi 1.13725

env OMP_NUM_THREADS=16 OMP_PLACES=cores ./pi
Time to integrate pi 1.0917

Speedup at 16 cores: 1.3x
Memory contention!

Protect access to hits with hardware atomic instruction

atomic [15.8.4] [2.19.7]

Ensures a specific storage location is accessed atomically.

C/C++

```
#pragma omp atomic [clause [ [, ] clause ] ... ]
statement
```

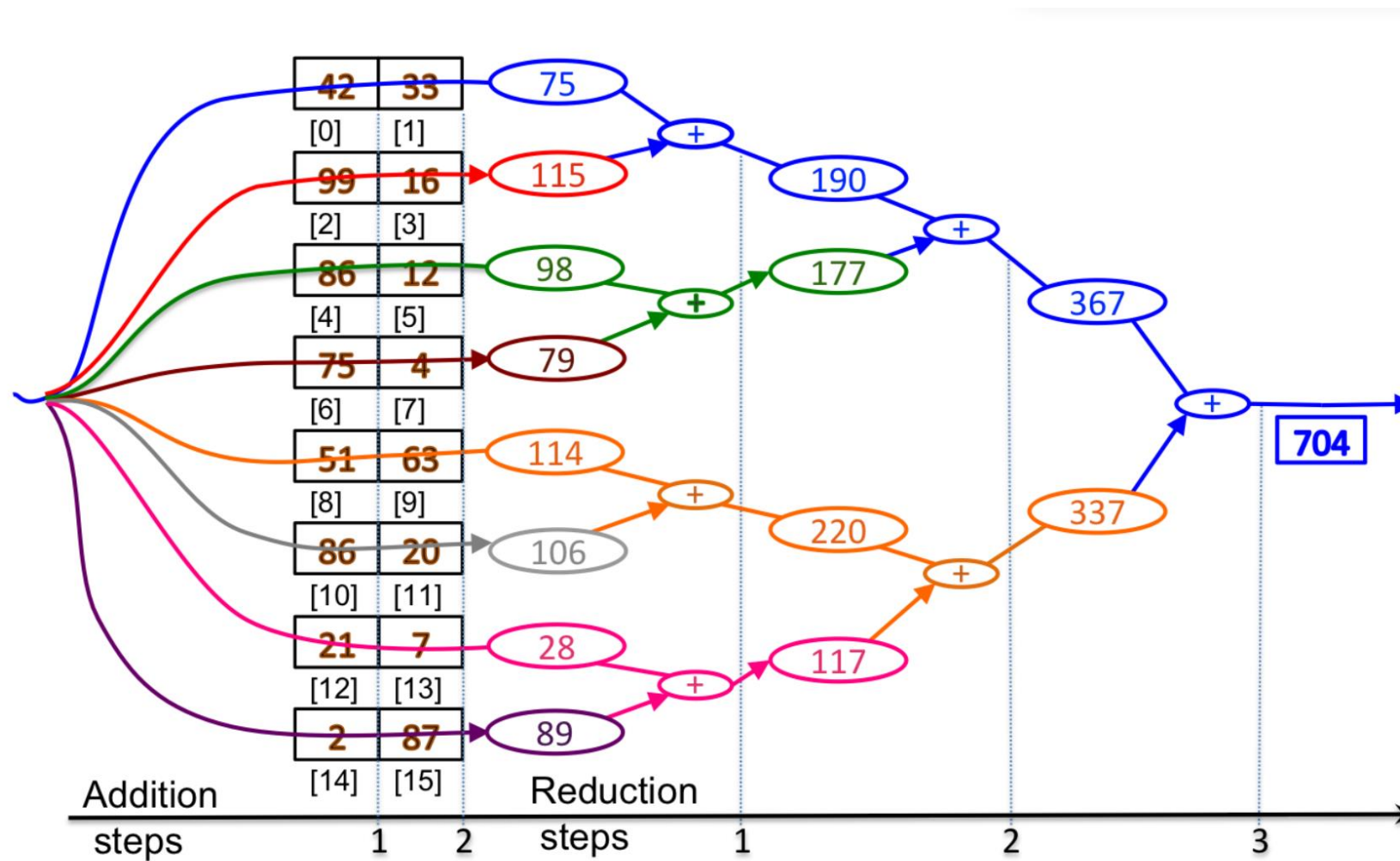
```
#pragma omp shared(counter)
#pragma omp private(private_cnt)
...
#pragma omp atomic capture
{
    private_cnt = counter;
    counter++;
}
```

C/C++ statement:

if atomic clause is... statement:


| | |
|--------------------|---|
| read | $v = x;$ |
| write | $x = \text{expr};$ |
| update | $x++; \quad x--; \quad ++x; \quad --x;$ $x \text{ binop} = \text{expr}; \quad x = x \text{ binop} \text{ expr};$ $x = \text{expr binop} x;$ |
| compare is present | <i>cond-expr-stmt:</i> $x = \text{expr} \text{ ordop} x ? \text{expr} : x;$ $x = x \text{ ordop} \text{expr} ? \text{expr} : x;$ $x = x == e ? d : x;$ <i>cond-update-stmt:</i> $\text{if}(\text{expr} \text{ ordop} x) \{ x = \text{expr}; \}$ $\text{if}(x \text{ ordop} \text{expr}) \{ x = \text{expr}; \}$ $\text{if}(x == e) \{ x = d; \}$ |
| capture is present | $v = \text{expr-stmt}$ $\{ v = x; \text{expr-stmt} \}$ $\{ \text{expr-stmt} v = x; \}$ <i>(expr-stmt: write-expr-stmt, update-expr-stmt, or cond-expr-stmt.)</i> |

Reducing memory contention with "reduction" clause

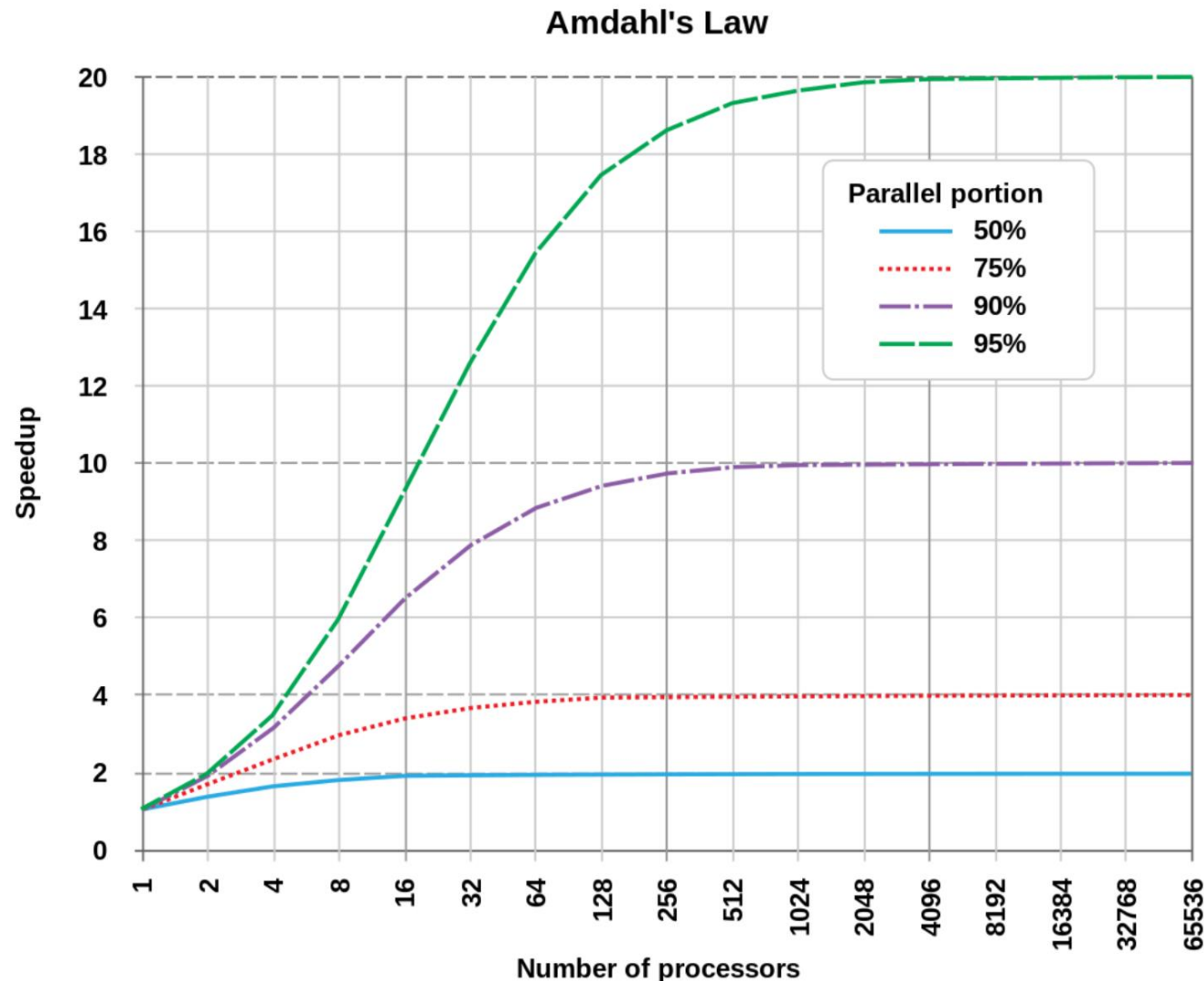


Reducing memory contention with "reduction" clause

```
int main(int argc, char *argv[]) {
    long n = 10000000;
    long hits = 0;
    double time = omp_get_wtime();
    #pragma omp parallel {
        lhits = 0
        #pragma omp for
        for(long i=0; i<n; ++i) {
            double x = randFloat()
            double y = randFloat()
            if (x*x + y*y <= 1) {
                ++lhits;
            }
        }
        #pragma omp atomic
        hits = hits + lhits;
    }
    time = omp_get_wtime() - time;
    double pi = (hits / (double) n) * 4;
    std::cout << "Time to integrate pi " << time << std::endl;
}
```



...reduction clause: finally success!



env OMP_NUM_THREADS=1 OMP_PLACES=cores ./pi
Time to integrate pi 1.46118

env OMP_NUM_THREADS=2 OMP_PLACES=cores ./pi
Time to integrate pi 0.737362

env OMP_NUM_THREADS=4 OMP_PLACES=cores ./pi
Time to integrate pi 0.36903

env OMP_NUM_THREADS=8 OMP_PLACES=cores ./pi
Time to integrate pi 0.186208

env OMP_NUM_THREADS=16 OMP_PLACES=cores ./pi
Time to integrate pi 0.0956299

Speedup at 16 cores: 15.3x

Reduction Operators

```
#pragma omp for reduction(reduction-identifier: variable list)
```

C/C++


Table 2.11 lists each *reduction-identifier* that is implicitly declared at every scope for arithmetic types and its semantic initializer value. The actual initializer value is that value as expressed in the data type of the reduction list item.

Table 2.11: Implicitly Declared C/C++ *reduction-identifiers*

| Identifier | Initializer | Combiner |
|------------|--|---|
| + | omp_priv = 0 | omp_out += omp_in |
| - | omp_priv = 0 | omp_out += omp_in |
| * | omp_priv = 1 | omp_out *= omp_in |
| & | omp_priv = ~ 0 | omp_out &= omp_in |
| | omp_priv = 0 | omp_out = omp_in |
| ^ | omp_priv = 0 | omp_out ^= omp_in |
| && | omp_priv = 1 | omp_out = omp_in && omp_out |
| | omp_priv = 0 | omp_out = omp_in omp_out |
| max | omp_priv = <i>Least representable number in the reduction list item type</i> | omp_out = omp_in > omp_out ? omp_in : omp_out |
| min | omp_priv = <i>Largest representable number in the reduction list item type</i> | omp_out = omp_in < omp_out ? omp_in : omp_out |

Going further: schedule and collapse clauses

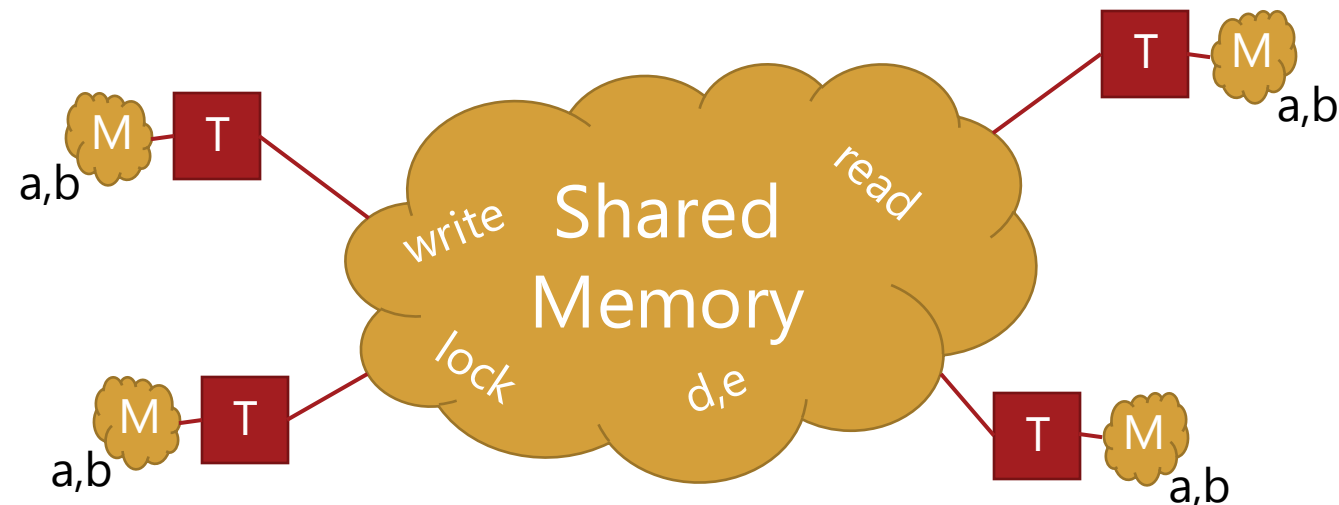
```
int main(int argc, char *argv[]) {
    long n = 3000;
    long hits = 0;
    double time = omp_get_wtime();
    #pragma omp parallel for schedule(static) collapse(2)
    for(long i=0; i<n; ++i) {
        for(long j=0; j<n; ++j) {
            double x = randFloat()
            double y = randFloat()
            if (x*x + y*y <= 1) {
                ++hits;
            }
        }
    }
    time = omp_get_wtime() - time;
    double pi = (hits / (double) n) * 4;
    std::cout << "Time to integrate pi " << time << std::endl;
}
```



- `#..for schedule()`
 - Indicates how to distribute loop iterations
 - `static`: equal chunk per thread. Low overhead
 - `dynamic`: assign chunks dynamically. High overhead
 - `guided`: start with larger chunks and decrease size in geometric progression. Medium overhead
 - `OMP_SCHEDULE` sets default value
- `#..for collapse(n)`
 - Apply parallel distribution across `n` nested loops

Data Model to share or not to share?

```
#pragma omp parallel for private(a,b) shared(d,e)
```



Memory scope: shared or private

- Variables declared before a parallel block are shared or private
- Shared variables are shared among all threads
- Private variables vary independently within threads
- On entry, values of private variables are undefined
- On exit, values of private variables are undefined
- By default, all variables declared outside a parallel block are shared
- Variables declared in a parallel block are always private
- Variables in a function called from within a parallel region are private, *except for static variables*
- `"omp parallel default(none)"` forces explicit specification of data scope for all variables in parallel region
- `"firstprivate(var-list)"` initialises private variables with shared value

Use (first)private variable and atomic to “simulate” reduction

```
int main(int argc, char *argv[]) {
    long n = 10000000;
    long hits = 0; long hits_priv = 0;
    double time = omp_get_wtime();
    #pragma omp parallel firstprivate(hits_priv)
    {
        #pragma omp for
        for(long i=0; i<n; ++i) {
            double x = randFloat(); double y = randFloat()
            if (x*x + y*y <= 1) {
                ++hits_priv;
            }
        }
        #pragma omp atomic
        hits += hits_priv;
    }
    time = omp_get_wtime() - time;
    double pi = (hits / (double) n) * 4;
    std::cout << "Time to integrate pi " << time << std::endl;
}
```

env OMP_NUM_THREADS=1 OMP_PLACES=cores ./pi
Time to integrate pi 1.43762

env OMP_NUM_THREADS=2 OMP_PLACES=cores ./pi
Time to integrate pi 0.730802

env OMP_NUM_THREADS=4 OMP_PLACES=cores ./pi
Time to integrate pi 0.36566

env OMP_NUM_THREADS=8 OMP_PLACES=cores ./pi
Time to integrate pi 0.184005

env OMP_NUM_THREADS=16 OMP_PLACES=cores ./pi
Time to integrate pi 0.0939641

Speedup at 16 cores: 15.3x

Use of `omp single`, `omp barrier`, and `nowait`

```
int main(int argc, char *argv[]) {
    long n = 10000000;
    long hits = 0; long hits_priv = 0;
    double time = omp_get_wtime();
    #pragma omp parallel firstprivate(hits_priv)
    {
        #pragma omp for nowait
        for(long i=0; i<n; ++i) {
            double x = randFloat(); double y = randFloat()
            if (x*x + y*y <= 1) {
                ++hits_priv;
            }
        }
        #pragma omp atomic
        hits += hits_priv;
        #pragma omp barrier
        #pragma omp single nowait
        {
            time = omp_get_wtime() - time;
            double pi = (hits / (double) n) * 4;
            std::cout << "Time to integrate pi " << time << std::endl;
        }
    }
}
```

- `#pragma omp single`
 - Region executed by a single thread
 - Useful for collection of data, I/O, initialization
- `#pragma omp barrier`
 - Signals synchronization point. All threads wait at this point
 - Implied barriers many places. Most importantly:
 - End of parallel region
 - End of `omp for` construct
 - End of `omp single` construct
 - Can be avoided with "nowait" clause

OpenMP Tasking

Dynamical Scheduling

Irregular Parallelism

Task Parallelism

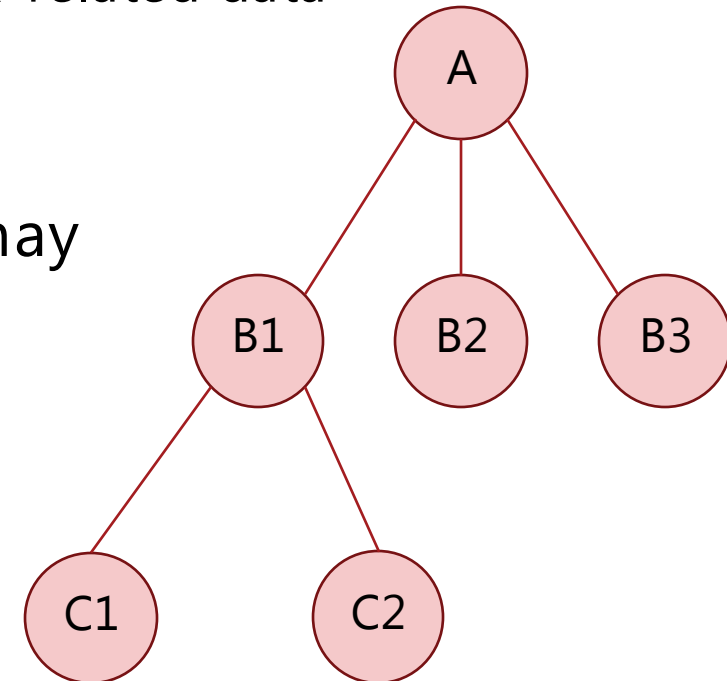
- Problems that are relevant for task parallelism have
 - data structures that are sparse
 - control structures that are not basic for-loops
- Classic example: list traversal

```
p = listhead;  
while (p) {  
    compute(p);  
    p=p->next;  
}
```

- Difficult to parallelise with "omp parallel for"

Task Parallelism

- The idea with task parallelism is to have independent work-units
- A task consists of data and instructions (much like an object!)
- A task can be scheduled on an arbitrary thread
 - Tasks can have dependencies
 - Tasks have a corresponding data scope for the task-related data
 - Tasks may be scheduled in any order
- A thread that encounters the task construct may
 - execute the task immediately
 - defer execution until later
 - let another thread execute the task



Task construct example: traversing a linked list

```
p = listhead;
while (p) {
    compute(p);
    p=p->next;
}
```



```
#pragma omp parallel
{
    #pragma omp single
    {
        p = listhead;
        while (p) {
            #pragma omp task firstprivate(p)
            {
                compute(p);
            }
            p=p->next;
            // dependency with next item
            if (p->flag) {
                #pragma omp taskwait
            }
        }
    }
}
```

Data scope of variables for tasks

- Variables can be shared, private or firstprivate with respect to task
- Scoping is relative to enclosing region:
- A shared variable on a task construct references to the storage with that name at the point where the task was encountered.
- A private variable on a task construct is private for this task region only.
- A firstprivate variable on a task construct, obtains the value of the existing storage of that name when the task is encountered. *Often variables should be declared firstprivate.*

```
#pragma omp parallel
{
    #pragma omp single
    {
        p = listhead;
        while (p) {
            #pragma omp task firstprivate(p)
            {
                compute(p);
            }
            p=p->next;
            // dependency with next item
            if (p->flag) {
                #pragma omp taskwait
            }
        }
    }
}
```

Calculating the Fibonacci sequence with task parallelism

- Serial algorithm for difference equation: $f_n = f_{n-1} + f_{n-2}$, $f_0=0$, $f_1=1$

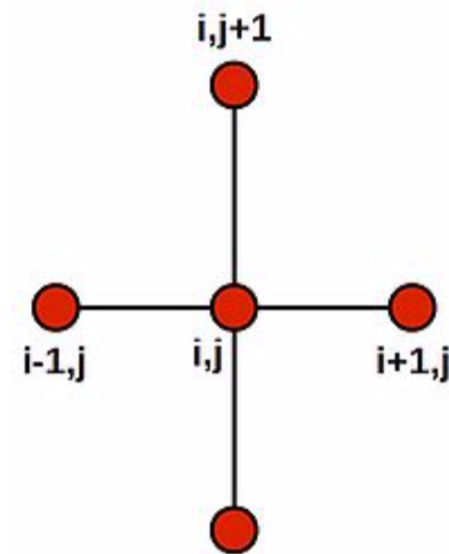
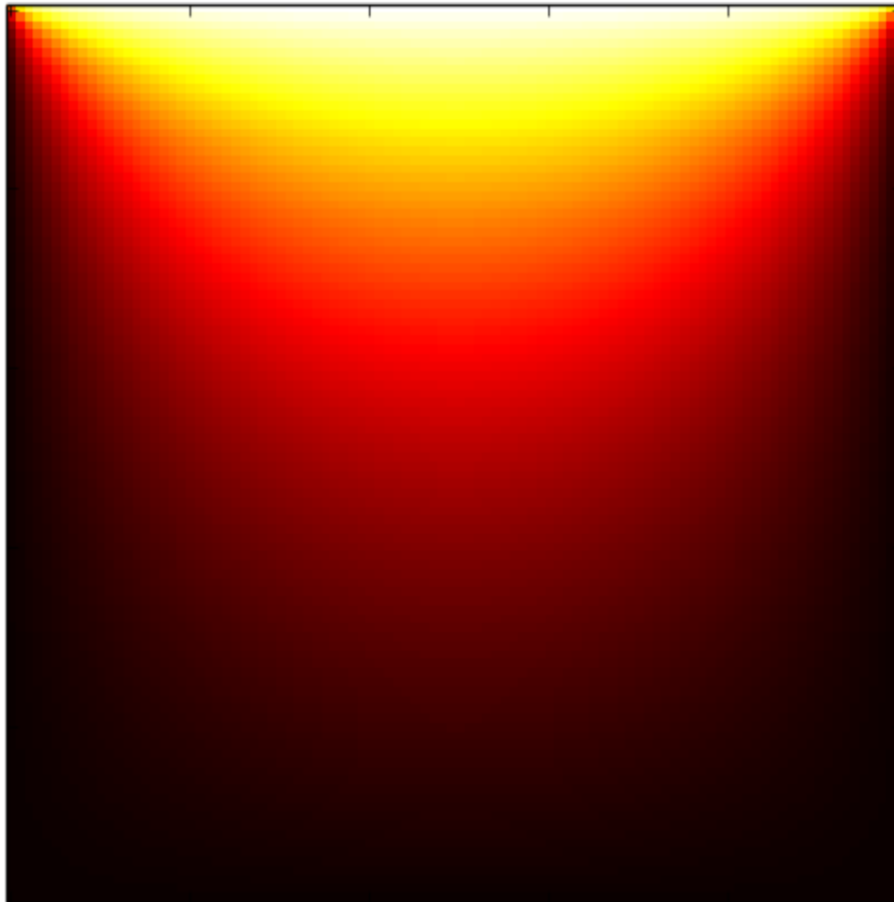
```
int fib (int n) {  
    int fn1,fn2;  
    if (n < 2) return n;  
    fn1 = fib(n-1);  
    fn2 = fib(n-2);  
    return (fn1+fn2);  
}  
  
int main() {  
    int Fmax = 1024;  
    #pragma omp parallel  
    fib(Fmax);  
}
```

Discuss in groups:
How can we use task constructs to parallelise the Fibonacci sequence?

Loop carried dependencies omp simd

Example 1: Where to parallelize Jacobi?

5-point stencil



Ex 1: Where to parallelize Jacobi?

This? ⇒ `while(delta > tol) {`
 `delta = 0.0;`
This? ⇒ `for(int j = 1; j < n-1; j++) { // Row idx`
This? ⇒ `for(int i = 1; i < m-1; i++) { // Col idx`
 `T[j][i] = A[j][i]; // Center`
 `T[j][i] += A[j][i+1]; // Right`
 `T[j][i] += A[j][i-1]; // Left`
 `T[j][i] += A[j+1][i]; // Up`
 `T[j][i] += A[j-1][i]; // Down`
 `T[j][i] *= 0.2; // Mean`
 `delta += fabs(T[j][i] - A[j][i]);`
 `}`
`}`

Ex 1: The outer loop

```
while(delta > tol) {  
    delta = 0.0;  
    #pragma omp parallel for reduction(+:delta)  
    for(int j = 1; j < n-1; j++) { // Row idx  
        for(int i = 1; i < m-1; i++ ) { // Col idx  
            T[j][i] = A[j][i]; // Center  
            T[j][i] += A[j][i+1]; // Right  
            T[j][i] += A[j][i-1]; // Left  
            T[j][i] += A[j+1][i]; // Up  
            T[j][i] += A[j-1][i]; // Down  
            T[j][i] *= 0.2; // Mean  
            delta += fabs(T[j][i] - A[j][i]);  
        }  
    }  
}
```

Ex 1: Vectorising the inner loop

```
while(delta > tol) {  
    delta = 0.0;  
    #pragma omp parallel for reduction(+:delta)  
    for(int j = 1; j < n-1; j++) { // Row idx  
        #pragma omp simd reduction(+:delta)  
        for(int i = 1; i < m-1; i++ ) { // Col idx  
            T[j][i] = A[j][i]; // Center  
            T[j][i] += A[j][i+1]; // Right  
            T[j][i] += A[j][i-1]; // Left  
            T[j][i] += A[j+1][i]; // Up  
            T[j][i] += A[j-1][i]; // Down  
            T[j][i] *= 0.2; // Mean  
            delta += fabs(T[j][i] - A[j][i]);  
        }  
    }  
}
```

Example 2: What loop to parallelise here ?

```
for(int i = 1; i < m; i++) {  
    for(int j = 1; j < n; j++) {  
        for(int k = 1; j < l; k++) {  
            A[i][j][k] = A[i-1][j][k];  
        }  
    }  
}
```

Ex 2: Optimal?

```
for(int i = 1; i < m; i++ ) {  
    #pragma omp parallel for  
    for(int j = 1; j < n; j++) {  
        #pragma omp simd  
        for(int k = 1; j < l; k++) {  
            A[i][j][k] = A[i-1][j][k];  
        }  
    }  
}
```

Ex 2: Reorder to lower number of barriers and parallel regions

```
for(int i = 1; i < m; i++ )  
    #pragma omp parallel for  
    for(int j = 1; j < n; j++)  
        #pragma omp simd  
        for(int k = 1; k < l; k++)  
            A[i][j][k] = A[i-1][j][k];
```



```
#pragma omp parallel for  
for(int j = 1; j < n; j++)  
    for(int i = 1; i < m; i++ )  
        #pragma omp simd  
        for(int k = 1; k < l; k++)  
            A[i][j][k] = A[i-1][j][k];
```


Example 3: Loop skewing

```
for (int i = 1; i < n; i++) {  
    b[i] = b[i] + a[i-1];  
    a[i] = a[i] + c[i];  
}
```

Example 3: Loop skewing

```
for (int i = 1; i < n; i++) {  
    b[i] = b[i] + a[i-1];  
    a[i] = a[i] + c[i];  
}
```



```
b[1] = b[1] - a[0];  
#pragma omp parallel for  
for (int i = 1; i < n; i++) {  
    a[i] = a[i] + c[i];  
    b[i+1] = b[i+1] + a[i];  
}  
a[n-1] = a[n-1] + c[n-1];
```

OpenMP tools for testing correctness

Thread Sanitizer: part of LLVM, gcc, Intel compilers, nvhpc, etc

See also PHPC chapter 17.5 and 17.6 – we will use Archer / Tsan discussed in 17.6.2

Tutorials / documentation:

<https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>

<https://www.intel.com/content/www/us/en/developer/articles/technical/find-bugs-quickly-using-sanitizers-with-oneapi-compiler.html>

<https://docs.nersc.gov/tools/debug/sanitizers/>

Alternatives exist:

- helgrind, DRD both part of Valgrind
- **thread sanitizer**, part of LLVM and gcc

Intel inspector is by far the best alternative with a nice GUI but Intel has sadly discontinued it by January 2025, and will concentrate on TSan

Runtime options:

<https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags>

<https://github.com/google/sanitizers/wiki/SanitizerCommonFlags>

Debug flag suggestions:

Threading errors: `-fsanitize=thread,undefined,bounds -fPIE -pie -Wall`

Memory leak errors: `-fsanitize=leak,address,undefined,bounds`

Summary – recipe for success

1. Start by profiling the code and identify where time is spent without OpenMP
2. Make sure to use OMP_PROC_BIND and OMP_PLACES for running code.
3. Parallelise individual loops where it makes sense. Verify! Test scaling. ***Loop-level OpenMP***
4. Start fusing parallel regions, by inserting *single pragmas*. Verify. Test scaling. Progress towards ***High-level OpenMP***
5. Where can *nowait* be used? Remove *implied barriers*. Verify! Test scaling
6. Consider reordering loops and statements, look at how memory is allocated. Add schedule clauses for consistent memory access. Verify. Test scaling.
7. Are there opportunities for irregular parallelism with *task pragmas*? Verify, remember taskwait, test scaling.

Be systematic. Write timings for each test into a spreadsheet or a piece of python code and keep track. Save each code version. Run correctness checker for all versions.

Summary

- OpenMP: pain-free parallelisation
- Standard is huge!
- Most can be done with constructs in table
- SEGFAULT: increase OMP_STACKSIZE
- `omp simd` a useful addition (week 2)
- Great tutorials here:

<https://www.iwomp.org/iwomp-2023/>

| OpenMP pragma, function, or clause | Concepts |
|---|--|
| <code>#pragma omp parallel</code> | Parallel region, teams of threads, structured block, interleaved execution across threads. |
| <code>void omp_set_thread_num()</code> <code>int omp_get_thread_num()</code> <code>int omp_get_num_threads()</code> | Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID. |
| <code>double omp_get_wtime()</code> | Speedup and Amdahl's law. False sharing and other performance issues. |
| <code>setenv OMP_NUM_THREADS N</code> | Setting the internal control variable for the default number of threads with an environment variable |
| <code>#pragma omp barrier</code> <code>#pragma omp critical</code> | Synchronization and race conditions. Revisit interleaved execution. |
| <code>#pragma omp for</code> <code>#pragma omp parallel for</code> | Worksharing, parallel loops, loop carried dependencies. |
| <code>reduction(op:list)</code> | Reductions of values across a team of threads. |
| <code>schedule (static [,chunk])</code> <code>schedule(dynamic [,chunk])</code> | Loop schedules, loop overheads, and load balance. |
| <code>shared(list), private(list), firstprivate(list)</code> | Data environment. |
| <code>default(none)</code> | Force explicit definition of each variable's storage attribute |
| <code>nowait</code> | Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive). |
| <code>#pragma omp single</code> | Workshare with a single thread. |
| <code>#pragma omp task</code> <code>#pragma omp taskwait</code> | Tasks including the data environment for tasks. |