Pitfalls and advanced OpenMP

Lukas Einkemmer
Department of Mathematics
University of Innsbruck

Shared memory parallelization with OpenMP – Day 2.
Link to slides: `http://www.einkemmer.net/training.html`

With special thanks to Rolf Rabenseifner (HLRS) on whose original slide set parts of this course are based.

How to write correct OpenMP programs

Pitfalls

OpenMP is easy to write, but it is also easy to get wrong.

Our goal is to discuss **common pitfalls** and **best practice** to avoid errors in OpenMP code.

Synchronization

A **synchronization point** in a parallel program coordinates the work of two or more threads.

Types of synchronization points:

- **Barrier:** execution of the program can not continue until all threads have reached the barrier
- **Critical (and atomic):** Only one thread can execute the critical region at the same time.
- **Lock functions:** fine grained control over synchronization.

Example of a barrier in OpenMP

```
// code
#pragma omp barrier
// no thread can execute this code until all threads
// have reached the barrier
```

OpenMP memory model

WRONG!

```
bool wait = false;
#pragma omp parallel for
for(int i=0;i<n;i++) {
    // busy wait
    while(wait)
        ;

    wait = true;
    // do some work
    wait = false;
}
```

The code tries to emulate a critical region.

The program is wrong because we have a **race condition**.

- Each thread reads and writes to the shared variable wait.

OpenMP memory model

The program, most likely, stops to make any progress.

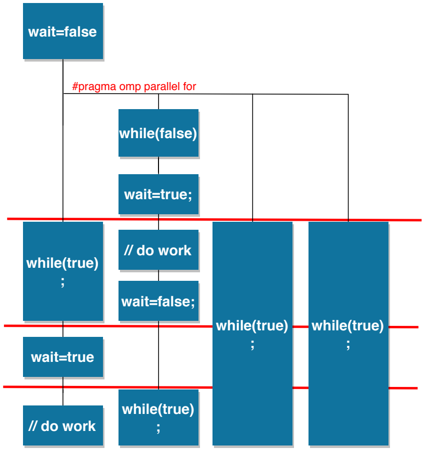
- This is called a **deadlock**.

Naive way to think about this program:

WRONG!

```
bool wait = false;
#pragma omp parallel for
for(int i=0;i<n;i++) {
    // busy wait
    while(wait)
        ;

    wait = true;
    // do some work
    wait = false;
}
```



OpenMP memory model

The **naive analysis** is **not correct**. The code

```
while(wait)
    ;
```

compiles to

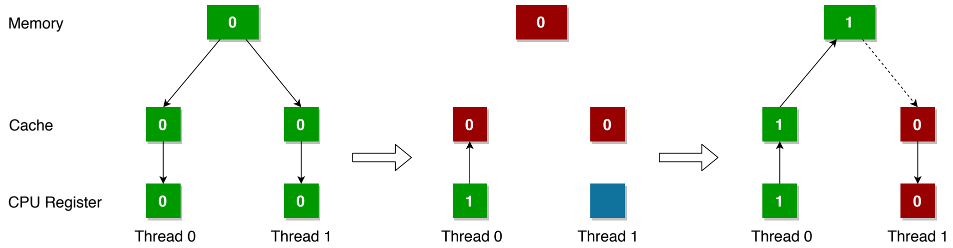
```
.L4:
    jmp .L4
```

The result of the compilation is an infinite loop.

Compiler Explorer: `https://godbolt.org`.
Full example: `https://godbolt.org/z/5xvBcC`.

OpenMP memory model

Accessing a shared variable from memory



OpenMP assumes that a thread can operate as if it were executed sequentially.

In a sequential program

```
wait = true;
while(wait) ;
```

is equivalent to

```
while(true) ;
```

From a performance perspective, this is the only choice.

OpenMP memory model

At some point in a program a consistent view of memory is required.

- This is called a **flush**.

A flush can be done explicitly by the

```
#pragma omp flush
```

directive. **Explicit flushes are almost never necessary.**

A **flush** is **very expensive**.

- All data in registers and caches have to be transferred back to main memory.
- Frequent flushes thus remove the performance benefit of the memory hierarchy.

OpenMP memory model

A flush is implied at

- barrier
- beginning and end of critical
- beginning and end of a parallel region
- end of a worksharing construct (for, do, sections, single, workshare)
- immediately before and after a task scheduling point

No flush is implied at

- beginning of a worksharing construct (for, do, sections, single, workshare)
- **beginning and end of master**

Recommendation: Use **OpenMP directives** (such as critical regions) for **synchronization**. Avoid lock functions.

Race condition

A **race condition** occurs when multiple threads are allowed to access the same memory location and at least one access is a write.

WRONG!

```
#pragma omp parallel
{
    #pragma omp for reduction(+:s) nowait
    for(int i=0;i<n;i++)
        s += v[i];

    int id = omp_get_thread_num();
    a[id] = f(s, id);
}
```

The nowait clause can be used to remove a flush.

Recommendation: be careful, this might introduce a race condition.

Race condition

Recommendation: declare variables where they are used.

Bad!

```
double x;
#pragma omp parallel for \
    private(x)
{
    // code
}
```

Good!

```
#pragma omp parallel for
{
    double x;
    // code
}
```

Recommendation: force the explicit declaration of all variables.

```
!$OMP PARALLEL DEFAULT(NONE) SHARED(...) PRIVATE(...)
// code
!$OMP END PARALLEL
```

Recommendation: use unit tests with different number of threads and multiple runs to test your code.

Recommendation: use tools that can detect race conditions (such as Intel Inspector).

Library functions

Race conditions can hide **inside library function**.

WRONG!

```
#pragma omp parallel
{
    time_t t;
    time(&t);
    tm* ptm = gmtime(&t);
}
```

From `http://www.cplusplus.com/reference/ctime/gmtime/`

> *A pointer to a tm structure with its members filled with the values that correspond to the UTC time representation of timer.*
> *The returned value points to an internal object whose validity or value may be altered by any subsequent call to gmtime or localtime.*

Library functions

Internally `gmtime` might look like

```
tm* gmtime(const time_t* timer) {
    static tm t;
    // code that populates t
    return &t;
}
```

`gmtime_r` is a thread safe alternative to `gmtime`, but `gmtime_r` is not part of the C++ standard.

Recommendation: make sure that library functions which are called inside OpenMP parallel regions are thread safe.

Recommendation: avoid side effects/internal state in functions that are called inside OpenMP parallel regions.

Implementation defined behavior

Certain behavior of the OpenMP runtime is not specified by the OpenMP standard:

- default number of threads;
- default schedule;
- size of the first chunck in schedule(guided);
- default schedule for schedule(runtime);
- default for dynamic thread adjustment;
- number of levels for nested parallelism.

Recommendation: do not rely on undefined behavior.

Recommendation: write OpenMP code that does not assume a certain number of threads, schedule, chunk size, etc.

How to write efficient OpenMP programs

Overhead of OpenMP

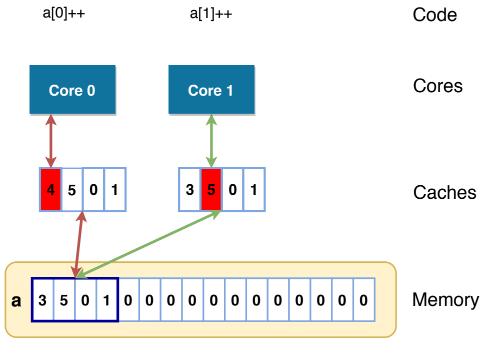
As a rule of thumb we pay the following penalty (in clock cycles)

Operation	cost in cycles	scaling
arithmetics	1	
L1 hit	1-10	
function call	10-20	
thread ID	**10-50**	**impl. dependent**
L3 hit	40	
sin/cos	100	
Static for, no barrier	**100-200**	**constant**
memory	200	
barrier	**200-500**	**log, linear**
parallel	**500-1000**	**linear**
dynamic for, no barrier	**10^3**	**problem dependent**
disk	10^5	

Exact numbers depend on the specific architecture.

False sharing

Several threads access the **same cache line**.



L1 and L2 caches are (usually) **distinct for each core**.

- **Cache coherence protocol** moves the cache line continuously between threads/cores.

This is associated with a large overhead.

Heat equation

Heat equation

Our goal is to solve the **heat equation**

$$\partial_t u(t,x,y) = \partial_{xx} u(t,x,y) + \partial_{yy} u(t,x,y)$$

with boundary conditions $u(t,x,0) = x$, $u(t,x,1) = x$, $u(t,0,y) = 0$, $u(t,1,y) = 1$ and initial condition $u(0,x,y) = 0$.

Solution is approximated by values on a grid u_{ij}^n.

Time discretization: $(\partial_t u)_{ij}^n \approx \frac{u_{ij}^{n+1} - u_{ij}^n}{\Delta t}$.

Space discretization: $(\partial_{xx} u)_{ij}^n \approx \frac{u_{i+1,j}^n - 2u_{ij}^n + u_{i-1,j}^n}{\Delta x^2}$

Time step

$$u_{ij}^{n+1} = u_{ij}^n + \frac{\Delta t}{\Delta x^2}\left(u_{i+1,j}^n - 2u_{ij}^n + u_{i-1,j}^n\right) + \frac{\Delta t}{\Delta y^2}\left(u_{i,j+1}^n - 2u_{ij}^n + u_{i,j-1}^n\right).$$

Heat equation

Goals:

- Parallelization of a more realistic application.
- Understand the performance of parallel programs.

Sequential program is provided

- **C/C++:** heat.c
- **Fortran:** heat.F

Compile flags to set the number of grid points

```
g++ -Dimax=250 -Dkmax=250 -O3 heat.c -o heat
```

Exercise 4a

Parallelize the program using the reduction clause.

Compile and run with 80×80 **grid points.**

Expected result (timings might be different):

- 0.4 s (sequential), 0.5 sec (1 thread), 2.8 sec (2 threads)

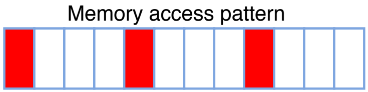
Why is the parallel implementation significantly slower than the sequential implementation?

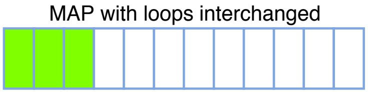
Solution 4a

The problem is in the sequential program

```
for(int k=0;k<kmax;k++)
    for(int i=0;i<imax;i++)
        dphi = (phi[i+1][k]+phi[i-1][k]-2.0*phi[i][k])*dy2i
             + (phi[i][k+1]+phi[i][k-1]-2.0*phi[i][k])*dx2i;
```

Memory access pattern:





Order of the two loops is important.

- Compiler might be smart enough to interchange the loops.
- Not possible if the outer loop is parallelized by OpenMP.

Exercise 4b

Tasks:

- Interchange nested loops.
- Investigate performance as a function of the problem size.

Expected results:

- No speedup for 80×80.
- Significant speedup for 250×250.
- Super-linear speedup for 1000×1000.

Why can we observe more than a speedup of 4 with `OMP_NUM_THREADS=4` (super-linear speedup)?

Solution 4b

Memory requirements: $2 \cdot \text{sizeof(double)} \cdot (10^3)^2 = 16\text{MB}$.

Problem does not fit into the cache of a single core anymore.

- **By increasing the number of cores** the amount of available **cache increases**.

Super-linear speedup is typical observed for relatively small problems.

Exercise 4c

Further optimize the code by **moving the parallel region outside of the time loop**.

Time the numerical computation and the abort statement.

- Why does the abort statement require almost the same time as the numerical computation?
- Use this knowledge to further optimize the program.

Solution 4c

```
#pragma omp parallel
for(it=1;it<=itmax;it++) {
    #pragma omp barrier
    #pragma omp single
    dphimax=0.;

    #pragma omp for reduction(max:dphimax)
    for(k=1;k<kmax;k++)
        for(i=1;i<imax;i++) {
            ...
        }
    #pragma omp for
    for(k=1;k<kmax;k++)
        for(i=1;i<imax;i++)
            phi[i][k] = phin[i][k];

    if(dphimax < eps)
        break;
}
```

Solution 4c

Do the abort condition only every 20th iteration.

Vectorization with OpenMP

Vectorization by the compiler

```
void vector_add(double* a, double* b) {
    a[0] += b[0]; a[1] += b[1];
    a[2] += b[2]; a[3] += b[3];
}
```

compiles to four different add instructions – **no vectorization!**.

```
void vector_add(double* __restrict a,
                double* __restrict b) {
    a[0] += b[0]; a[1] += b[1];
    a[2] += b[2]; a[3] += b[3];
}
```

compiles to

```
vmovupd ymm0, YMMWORD PTR [rsi]          # loads 4 doubles
vaddpd  ymm0, ymm0, YMMWORD PTR [rdi]    # adds 4 doubles
vmovupd YMMWORD PTR [rdi], ymm0          # write 4 doubles
```

Full examples: `https://godbolt.org/z/1IEVSj`, `https://godbolt.org/z/9JB6T2`.

Vectorization by the compiler

The function

```
void vector_add(double* a, double* b) {
    a[0] += b[0]; a[1] += b[1];
    a[2] += b[2]; a[3] += b[3];
}
```

can not be vectorized since the following call is completely legal

```
double* p;
vector_add(p, p+1);
```

which results in

```
p[0] = p[0] + p[1];
p[1] = p[1] + p[2]; // not independent of previous line
```

Automatic vectorization is a difficult problem for the compiler!

Keyword `__restrict` tells the compiler that all memory accesses that change a are done explicitly through a – makes it much easier for the compiler to reason about the code.

Vectorization using OpenMP

The **simd directive** is used to tell the compiler that the **loop iterations are independent**.

```
#pragma omp simd
for(int i=0;i<n;i++)
    a[i] += b[i];
```

Is used in the same way as the **for/do directives**.

Programmer takes responsibility that loop iterations can be parallelized.

- Responsibility to *proof* correctness is transferred to a human.

The clauses private, lastprivate, reduction, and collapse can be used exactly as for a parallel for loops.

Full example: `https://godbolt.org/z/AuNwOU`.

Vectorization using OpenMP

WRONG!

```
#pragma omp simd
for(int i=5;i<n;i++)
    a[i] = a[i-5]*b[i];
```

Correct.

```
#pragma omp simd safelen(4)
for(int i=5;i<n;i++)
    a[i] = a[i-5]*b[i];
```

safelen(m) clause specifies that a maximum of $m+1$ elements (index 0 to m) of the loop can be together in a vector.

Vectorization using OpenMP

Functions can be used in an omp simd directive.

```
#pragma omp declare simd notinbranch
double dist(double x1, double y1, double x2, double y2) {
    return sqrt(pow(x1-x2,2) + pow(y1-y2,2));
}

#pragma omp simd
for(int i=0;i<n;i++)
    d[i] = dist(x1[i], y1[i], x2[i], y2[i]);
```

Vectorization using OpenMP

Modern CPUs can also vectorize branches

```
#pragma omp declare simd inbranch
double dist(double x1, double y1, double x2, double y2) {
    return sqrt(pow(x1-x2,2) + pow(y1-y2,2));
}

#pragma omp simd
for(int i=0;i<n;i++)
    if(x1[i] > x2[i])
        d[i] = dist(x1[i], y1[i], x2[i], y2[i]);
    else
        e[i] = dist(x1[i], y1[i], x2[i], y2[i]);
```

Whether such a statement is actually vectorized depends on the compiler and the available instruction set.

Vectorization using OpenMP

Core based parallelism (MIMD) and vectorization (SIMD) can be combined.

```
#pragma omp parallel for simd
for(int i=0;i<n;i++)
    a[i] += b[i];
```

Array of struct vs struct of arrays

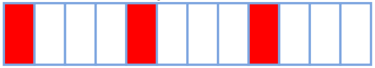
Array of struct (AoS)

```
struct state {
    double density;
    double momentum;
    // ...
};
vector<state> v_aos;
```

No vectorization

```
#pragma omp simd
for(int i=0;i<n;i++)
    v_aos[i].density
      = f(v_aos[i].density);
```

Memory access AoS



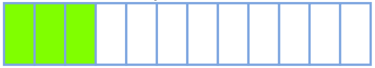
Struct of arrays (SoA)

```
struct states {
    vector<double> density;
    vector<double> momentum;
    // ...
};
states v_soa;
```

Vectorization

```
#pragma omp simd
for(int i=0;i<n;i++)
    v_soa.density[i]
      = f(v_soa.density[i]);
```

Memory access SoA



Full example: `https://godbolt.org/z/kik_VP`.

Thread affinity in OpenMP

Thread affinity

In order to run a OpenMP program **threads** have to be **mapped to cores**.

- By default, threads can be moved from one core to another.

On modern systems **moving threads can reduce performance**.

- Core specific caches have to be invalidated.
- First touch principle is only beneficial if threads are fixed to the same NUMA domain.

Disable thread movement:

```
export OMP_PROC_BIND=true
```

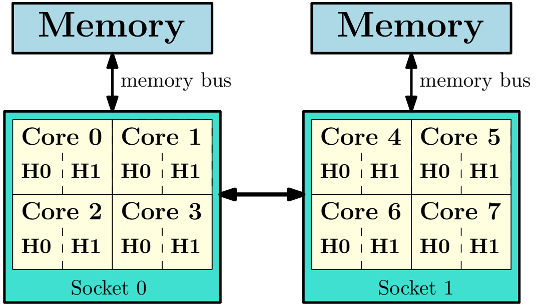
Support for mapping threads to the underlying hardware has been added in OpenMP 4.0.

- Previously, a patchwork of different tools could be used to accomplish this.

Thread affinity

```
cat /proc/cpuinfo
processors: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

processor   : 0                      processor   : 4
physical id : 0                      physical id : 0
core id     : 0                      core id     : 0
```



16 processors = 2 CPUs $\times$ 4 cores $\times$ 2 hyperthreads

OpenMP places and proc_bind

Place partition:

`OMP_PLACES = threads or cores or sockets`

Threads can freely migrate within a place.

Placement options:

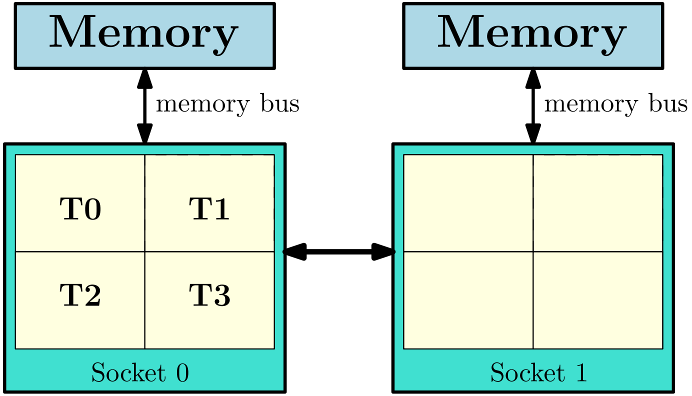
`OMP_PROC_BIND = spread or close or master`

- **close:** place threads as close together as possible.
- **spread:** place threads as far apart as possible.
- **master:** place threads on the same place partition.

Thread placement

Place all threads on the same NUMA node, one thread per core.

```
OMP_NUM_THREADS=4
OMP_PLACES=cores
OMP_PROC_BIND=close
```

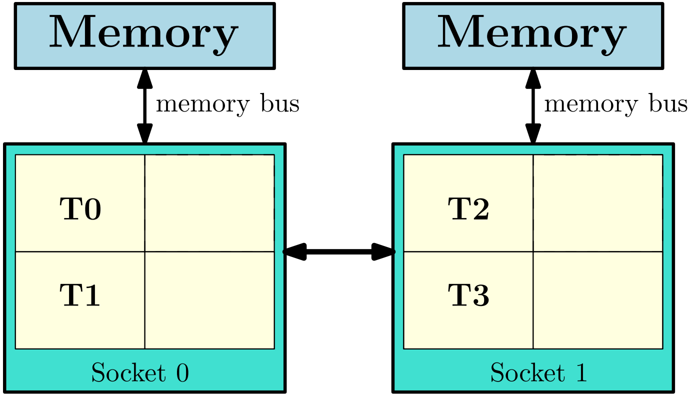


Threads can be moved between hyperthreads.

Thread placement

Spread threads equally among the two NUMA nodes, one thread per core.

```
OMP_NUM_THREADS=4
OMP_PLACES=cores
OMP_PROC_BIND=spread
```

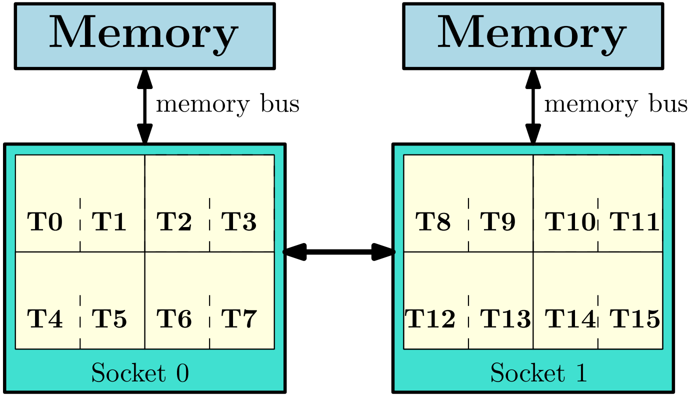


Threads can be moved between hyperthreads.

Thread placement

One-to-one placement between threads and hyperthreads.

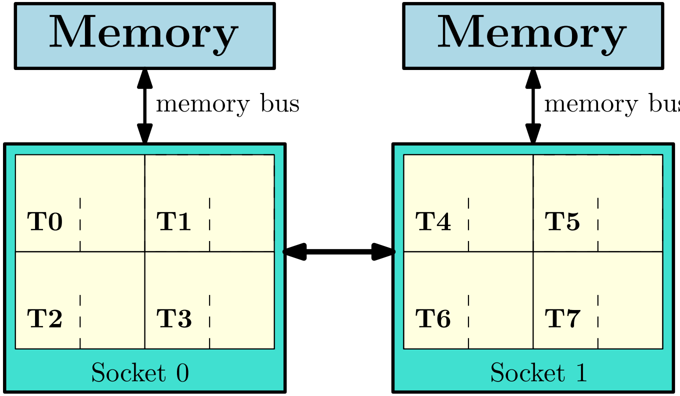
```
OMP_NUM_THREADS=16
OMP_PLACES=threads
OMP_PROC_BIND=close
```



Thread placement

One thread per core.

```
OMP_NUM_THREADS=8
OMP_PLACES=threads
OMP_PROC_BIND=spread
```



Threads are fixed to a single hyperthread.

Thread placement

Recommendation: number of threads $\leq$ number of cores. One thread per core.

Recommendation: for memory bound problems spread threads across all NUMA domains to make full use of the available memory bandwidth (requires first touch).

Recommendation: Hybrid MPI+OpenMP. One MPI process per socket and one thread per core.

```
OMP_NUM_THREADS=4
OMP_PLACES=cores
OMP_PROC_BIND=close
```

Each MPI process runs on a single NUMA domain.

Thread placement for nested parallelism

OpenMP environment variables can specify different values for **nested parallel regions**.

```
OMP_NUM_THREADS=2,4,2
OMP_PLACES=threads
OMP_PROC_BIND=spread,spread,close
```

The code

```
#pragma omp parallel         // creates one thread/socket
    #pragma omp parallel     // creates one thread/core
        #pragma omp parallel // creates one thread/hyperthread
            //code
```

creates a total of 16 threads.

The taskloop directive

Remember tasks

```
struct node {
    node *left, *right;
};
void traverse(node* p) {
    if(p->left)
        #pragma omp task
        traverse(p->left); // this is created as a task
    if(p->right)
        #pragma omp task
        traverse(p->right); // this is created as a task
    process(p);
}
int main() {
    node tree;
    #pragma omp parallel // create a team of threads
    #pragma omp single
    traverse(&tree); // executed sequentially
}
```

Taskloop

Taskloop works like a parallel for loop and is used like a task construct.

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop
    for(int i=0;i<n;i++)
        a[i] = b[i] + i;
```

We can control the number of tasks by setting either

- **num_tasks:** number of tasks that are generated; or
- **grainsize:** how many loop iterations should be assinged to a single task.

private, collapse, etc. can be used as in a parallel for loop.

- reduction clause for taskloop has been added in OpenMP 5.0.

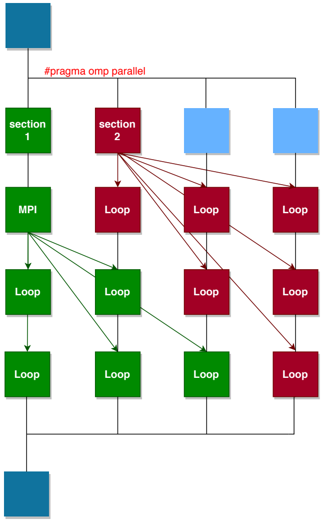
Many more tasks can be generated than threads are available.

- **Load balancing** similar to the **dynamic scheduling** strategy.

Taskloop

Main application of taskloop is to combine task based and loop based parallelism.

```
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    {
        // MPI communication
        #pragma omp taskloop
        for(int i=0;i<n_b;i++)
            a[i] = ...;
    }
    #pragma omp section
    {
        #pragma omp taskloop
        for(int i=n_b;i<n;i++)
            a[i] = ...;
    }
}
```



Exercise 5

Goal:

- usage of taskloop construct.

Sequential program is provided in

- **C/C++:** pi_taskloop.c and pi_taskloop2.c
- **Fortran:** pi_taskloop.f90 and pi_taskloop2.f90

Use taskloop to parallelize pi_taskloop.[c|f90].
Use sections+2×taskloop to parallelize pi_taskloop2.[c|f90].