Paralelní a distribuované výpočty (B4B36PDV)

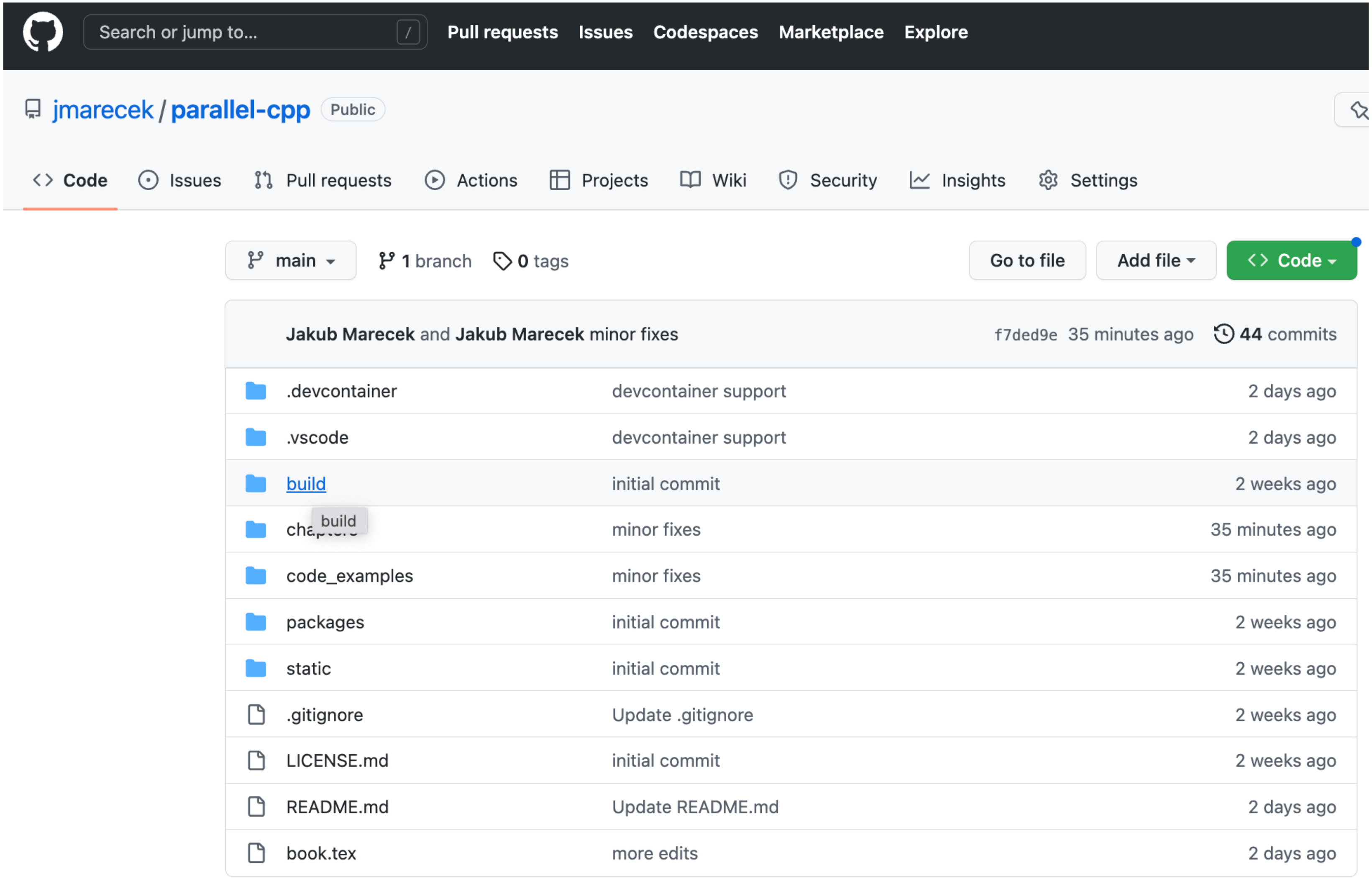
Jakub Mareček
jakub.marecek@fel.cvut.cz

Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

What comes next?

What comes next?

https://github.com/jmarecek/parallel-cpp/blob/main/static/2024March7.pdf



Jakub Marecek and Jakub Marecek minor fixes f7ded9e 35 minutes ago 44 commits

.devcontainer	devcontainer support	2 days ago
.vscode	devcontainer support	2 days ago
build	initial commit	2 weeks ago
chapters	minor fixes	35 minutes ago
code_examples	minor fixes	35 minutes ago
packages	initial commit	2 weeks ago
static	initial commit	2 weeks ago
.gitignore	Update .gitignore	2 weeks ago
LICENSE.md	initial commit	2 weeks ago
README.md	Update README.md	2 days ago
book.tex	more edits	2 days ago

What is OpenMP?

A Specification

OpenMP®
Enabling HPC since 1997

2015: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

2020: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf

2022: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

2023: TR 12 introduces a preview of OpenMP 6 with support for C++23: https://www.openmp.org/wp-content/uploads/openmp-TR12.pdf

What is OpenMP?

A Specification

- OpenMP is a specification for parallel programming of shared-memory systems in Fortran, C, and C++.
- The current version of the specification can be downloaded from https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf with many examples at https://github.com/OpenMP/Examples
- The generally does not provide any guarantees as to how a particular directive or function is implemented.
- This also means that running on a different hardware may result in different order of execution of floating-point operations and vastly different performance.
- There is a long history of the evolution of OpenMP since 1997, from a 50-page long specification of OpenMP 1.0 to 795 pages in TR 12.

What is OpenMP?

A Specification

Prime implementations include

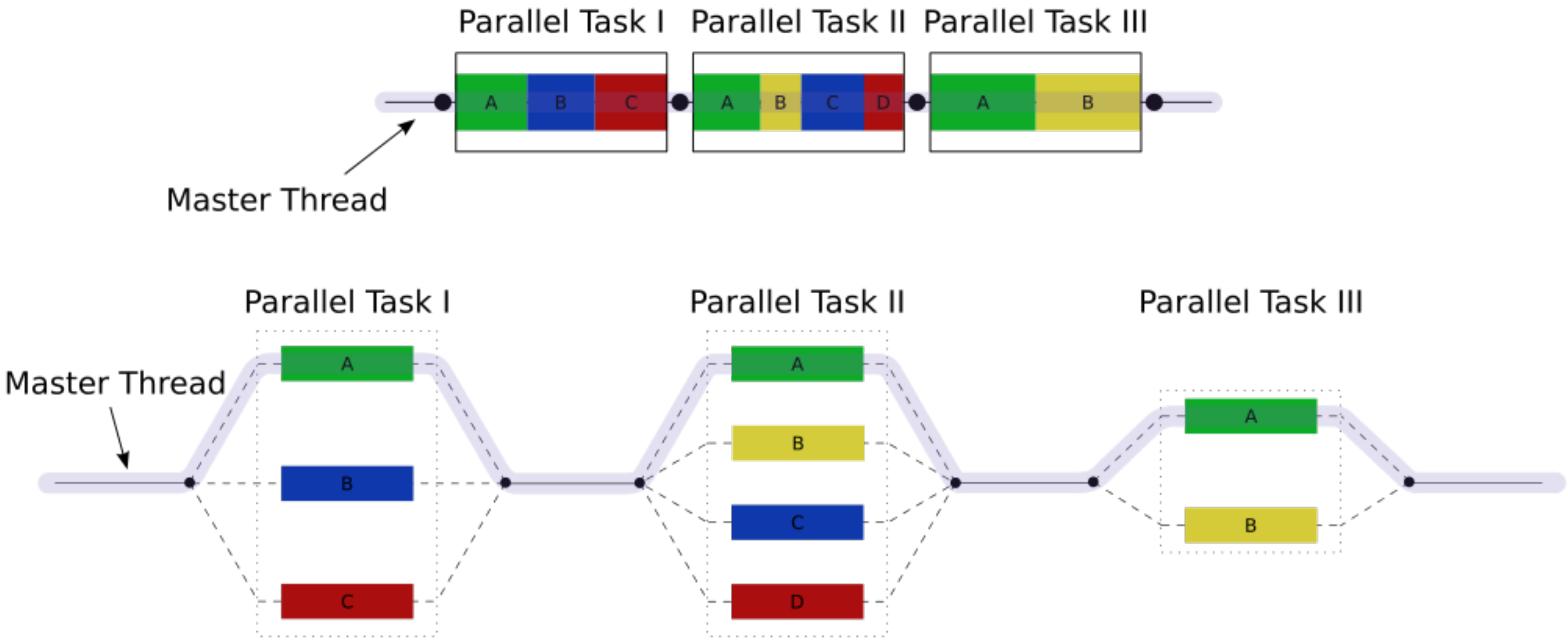
- libgomp (GOMP) for GCC,
- libomp for clang, and
- liomp5 (IOMP) for ICC/Clang.

There are also two “subprime” implementation available in Microsoft Visual Studio, which has its own OpenMP runtime (/openmp) and an experimental support (/openmp:llvm) for the clang/llvm OpenMP runtime, in both cases at version 2.0. If you wish to use Microsoft Visual Studio, notice that Intel oneAPI release 2023.0 oficially supports Microsoft Visual Studio 2022.

What is OpenMP?

A Specification

- The specification is built on top of the fork-join model of parallel execution (sériově-paralelní model uspořádání vláken).



By Wikipedia user A1 - w:en:File:Fork_join.svg, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=32004077

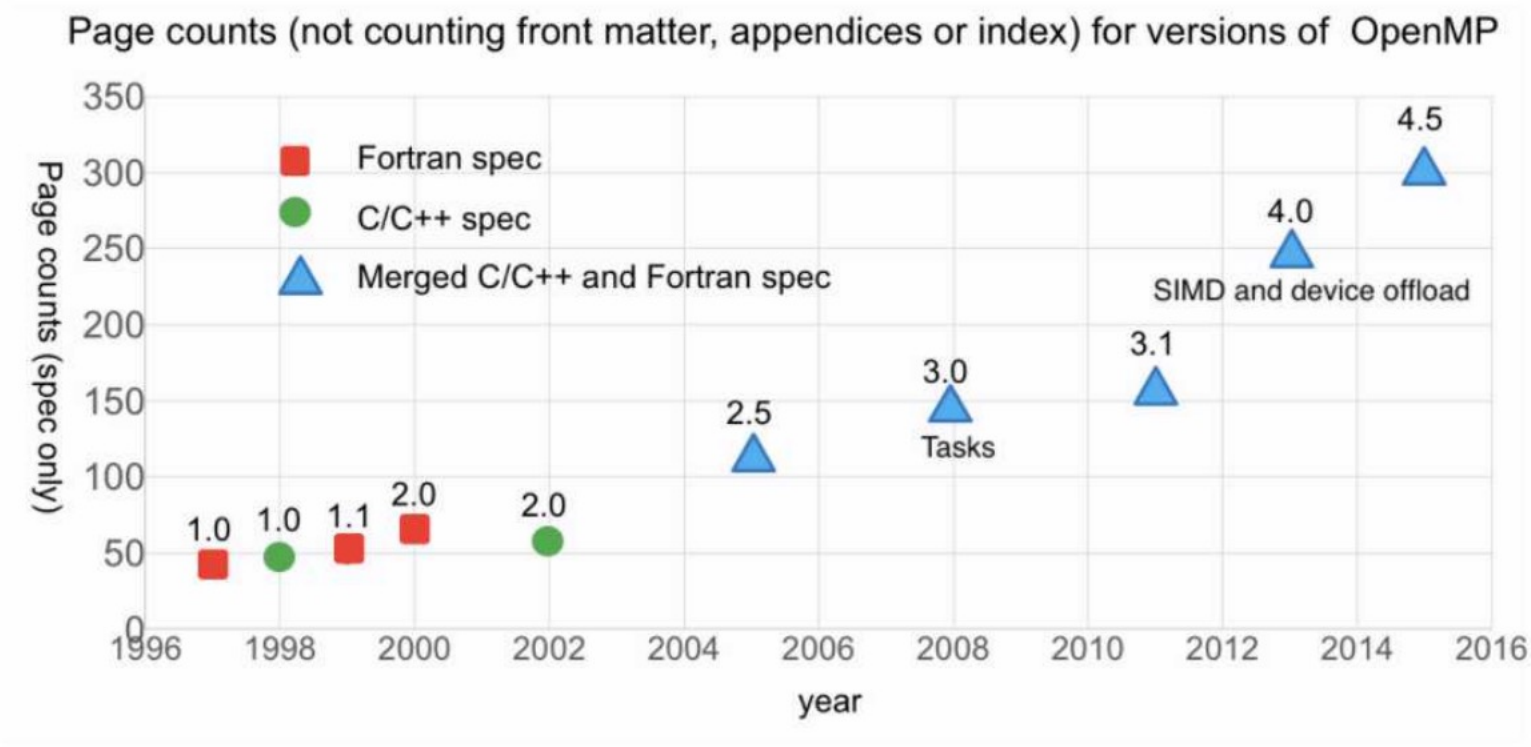
What is OpenMP?

A Specification

- The specification is built on top of the fork-join model of parallel execution (sériově-paralelní model uspořádání vláken).
- Traditional implementations of OpenMP have been rather closely built on top of Pthreads, which results in the lack of fine-grained scheduling, memory management, network management, signaling, etc.
- The lack of fine-grained scheduling notably meant the lack of user-level threads (co-routines) and the lack of queries as to the number of hardware threads utilized by other processes, which often results in high overhead when the number of threads (across all processes!) increases above the number of hardware threads supported.
- Since OpenMP 5.0, the distinction between threads and tasks has been erased and thread teams are also cast into tasks. There are now also OpenMP implementations over lightweight threads, notably (BOLT is OpenMP over Lightweight Threads, https://www.bolt-omp.org/).

What is OpenMP?

A Specification



“The Ongoing Evolution of OpenMP,” https://doi.org/10.1109/JPROC.2018.2853600

What is OpenMP?

A Specification

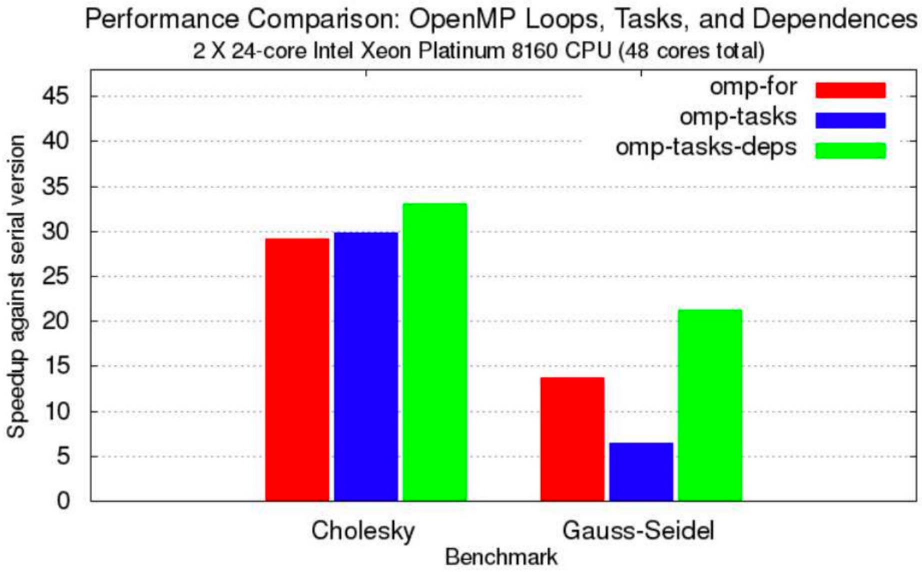
- OpenMP specifies preprocessor directives (pragmas) and a library of functions exported via **omp.h**.
- Many OpenMP programs use only the preprocessor directives, which makes it possible to compile them as serial code even without OpenMP-aware compiler.
- Ideally, one and the same program written with OpenMP should be possible to run as serial code, or with any number of threads.

What comes next?

- Structuring code
  - OpenMP Task Region
  - Threads and their Sizing
  - Section
  - Task
  - Kernels, Teams, and Targets

- Synchronization primitives
  - Atomic variables
  - Mutexes and locks
  - Critical sections
  - Barrier
  - Fences and Flushes

- For each



“The Ongoing Evolution of OpenMP,” https://doi.org/10.1109/JPROC.2018.2853600

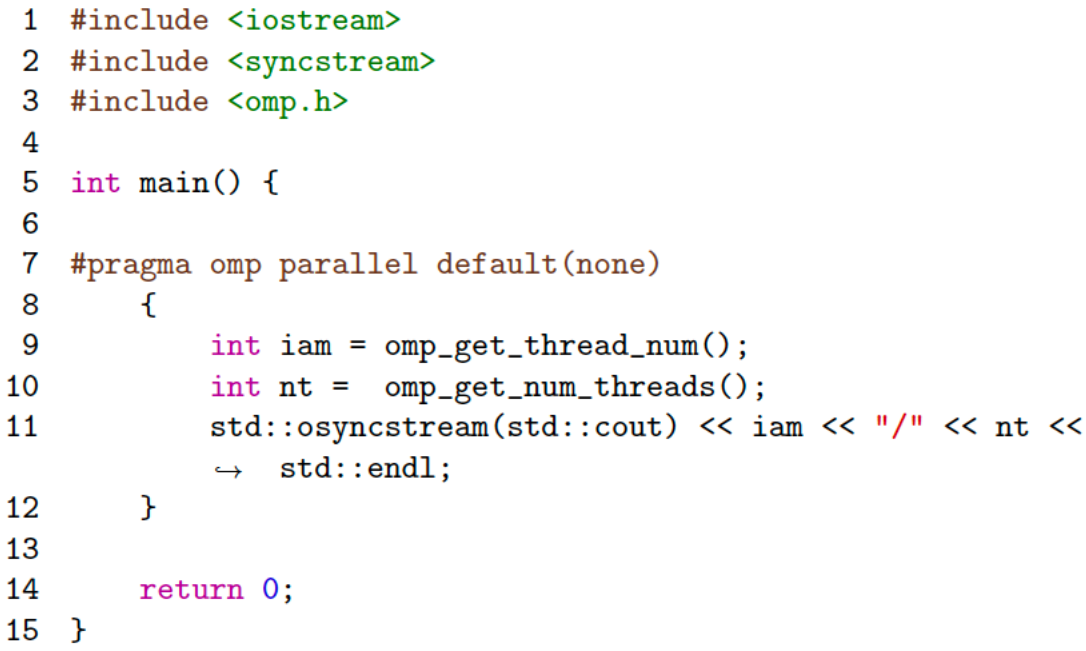
Structuring code

Task Region

- Initially, OpenMP application starts with a single thread (initial/master thread). This can spawn parallel regions, typically with multiple threads in a thread pool (``thread team'') in the fork-join manner.
- This means that in any thread, one can either wait for all the ``sibling'' threads to finish (``join them'') or spawn a further, nested parallel region.
- A key construct in OpenMP is thus **#pragma omp parallel**, which delineates a parallel region and opens a ``team of OpenMP threads'', which could be seen as a thread pool of threads or user-level threads.
- By default, the number of threads is set based on the available hardware threads, but this can be affected by the environment variables (OMP_NUM_THREADS) and modifiers of the pragma (num_threads(2)) and function calls (omp_set_num_threads()).

Structuring code

Task Region



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

int main() {

#pragma omp parallel default(none)
    {
        int iam = omp_get_thread_num();
        int nt =  omp_get_num_threads();
        std::osyncstream(std::cout) << iam << "/" << nt << std::endl;
    }

    return 0;
}
```

Structuring code

Task Region

The **parallel** construct can take a number of modifiers, including:

- **num_threads**: number of threads to use in the team
- **private** (list of variables): those variables will be private to each thread
- **firstprivate** (list of variables): those variables will be private to each thread, but initially their value will be copied from the master thread using the default copy constructor.
- **lastprivate** (list of variables): those variables will be private to each thread, but at the end, their value will be copied to the master thread using the default copy constructor.
- **shared** (list of variables): these variables will be shared between the master thread and all threads in the new team. It is the programmer's responsibility to keep the variables constructed as long as the parallel region is running
- **default** : values of **private, firstprivate, shared, none** suggest what should be the default behaviour for variables not listed above. The default is shared , which is suboptimal from both performance and thread-safety perspective. It is wise to issue default(none) .

Structuring code

Task Region

The **parallel** construct can take a number of modifiers, including:

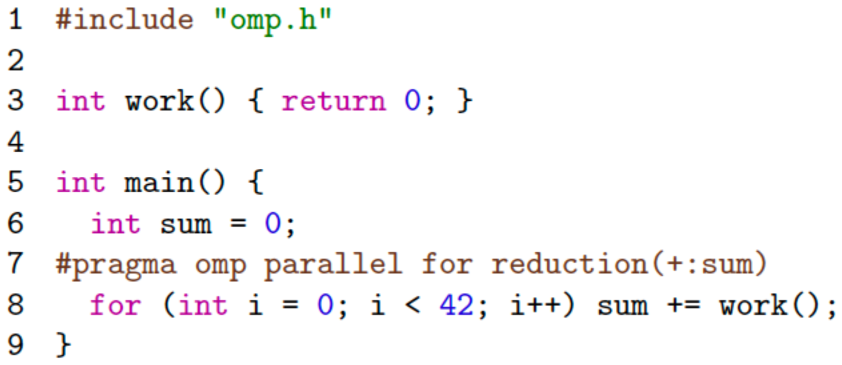
- **reduction** (reduction-identifier : list) suggests that variables in list should be treated as shared, when they are used by the function reduction-identifier , which could also take the special values of +, -, *, \&, |, ^, ||, **max, min**. The list can include array elements and, when reduction-identifier is a static function of a class, accessible data objects of the object.
- **proc_bind** : values of master and close and spread suggest how far from the master thread should be executed the new threads (same core, close in non-uniform architectures, as far as possible in non-uniform architectures).

Structuring code

Task Region

The **parallel** construct can take a number of modifiers, including:

- **reduction** (reduction-identifier : list) suggests that variables in list should be treated as shared, when they are used by the function reduction-identifier , which could also take the special values of +, -, *, \&, |, ^, ||, **max, min.** The list can include array elements and, when reduction-identifier is a static function of a class, accessible data objects of the object.
- **proc_bind** : values of master and close and spread suggest how far from the master thread should be executed the new threads (same core, close in non-uniform architectures, as far as possible in non-uniform architectures).



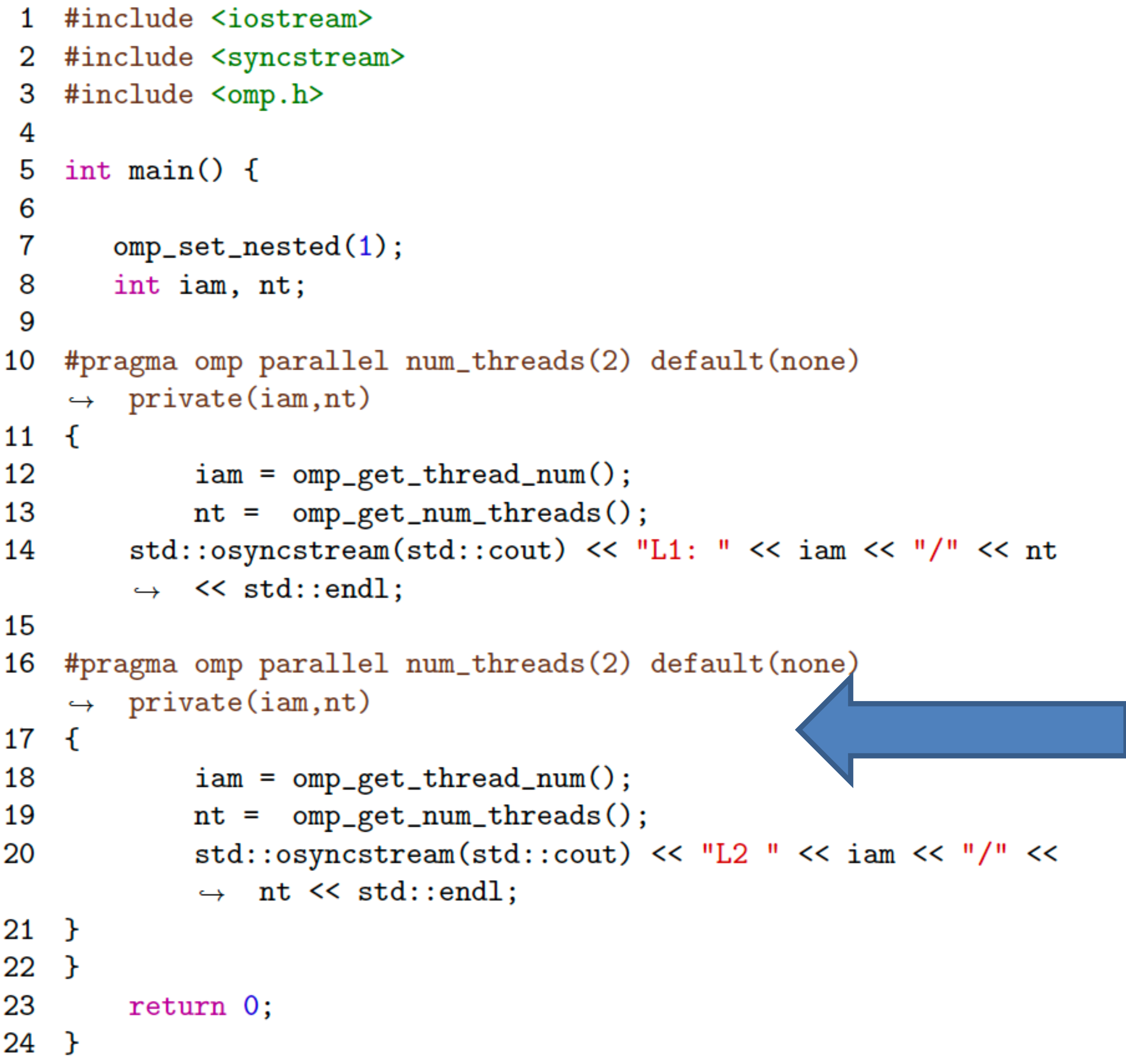
```c
#include "omp.h"

int work() { return 0; }

int main() {
  int sum = 0;
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < 42; i++) sum += work();
}
```

The reduction produces a single value from an associative operations such as addition, multiplication, taking of the minimum, maximum, or custom associative functions. The goal is for each thread to run the reduction with a private copy and then to produce the final result with the same reduction, perhaps in a hierarchical fashion.

Structuring code

Task Region

Whether the nested parallel regions create their own thread teams or use the existing thread teams depends on settings that we can affect through **omp_set_nested**, or environment variables OMP_NESTED (which can be true or false) and OMP_MAX_ACTIVE_LEVELS , which controls the maximum number of nested active parallel regions.



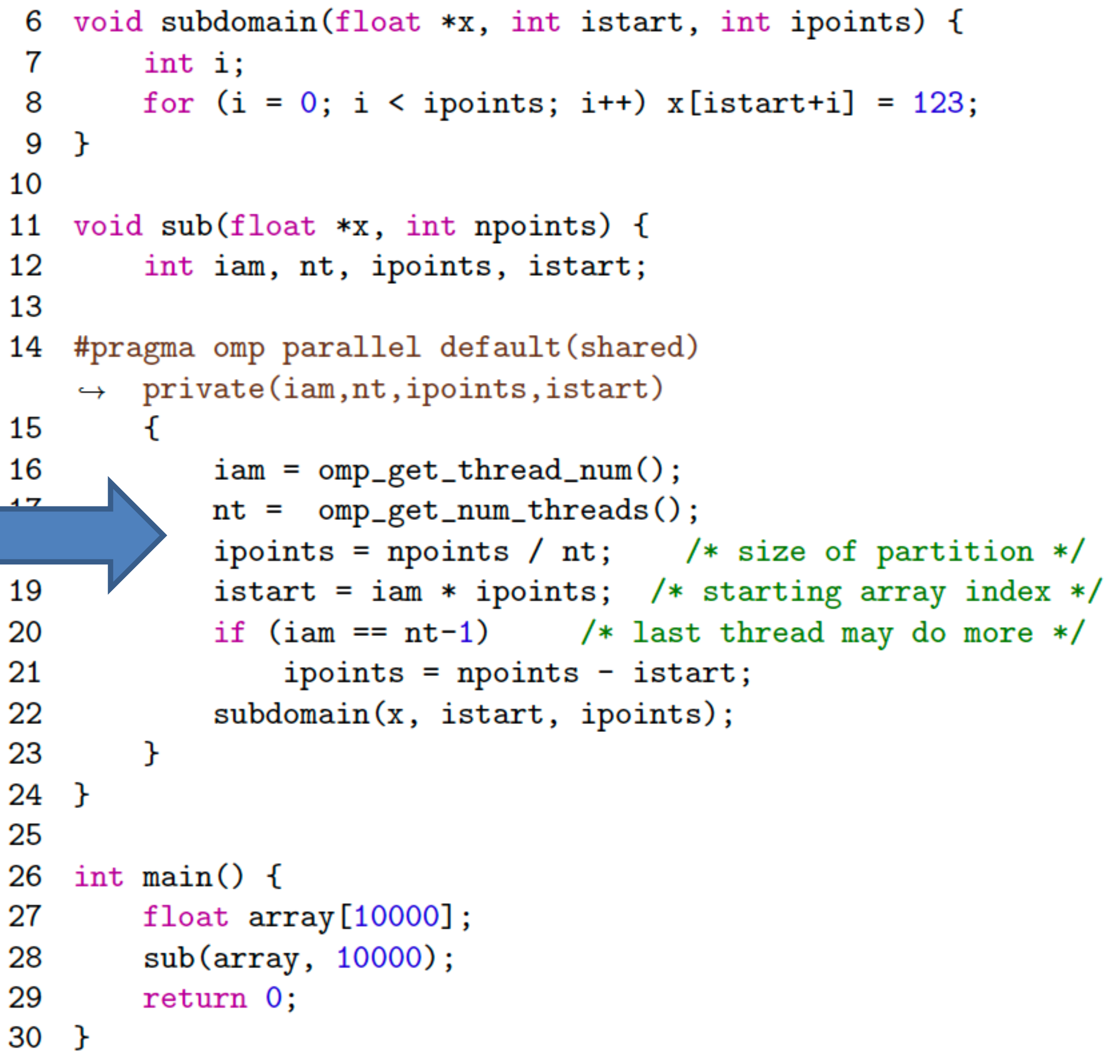
```
#include <iostream>
#include <syncstream>
#include <omp.h>

int main() {

   omp_set_nested(1);
   int iam, nt;

#pragma omp parallel num_threads(2) default(none) private(iam,nt)
{
        iam = omp_get_thread_num();
        nt =  omp_get_num_threads();
    std::osyncstream(std::cout) << "L1: " << iam << "/" << nt << std::endl;

#pragma omp parallel num_threads(2) default(none) private(iam,nt)
{
        iam = omp_get_thread_num();
        nt =  omp_get_num_threads();
        std::osyncstream(std::cout) << "L2 " << iam << "/" << nt << std::endl;
}
}

    return 0;
}
```

Structuring code

Threads and their Sizing

As has been mentioned above, ideally one and the same program written with OpenMP should be possible to run as serial code, or with any number of threads.

This requires sizing the work in each thread depending on the number of threads:



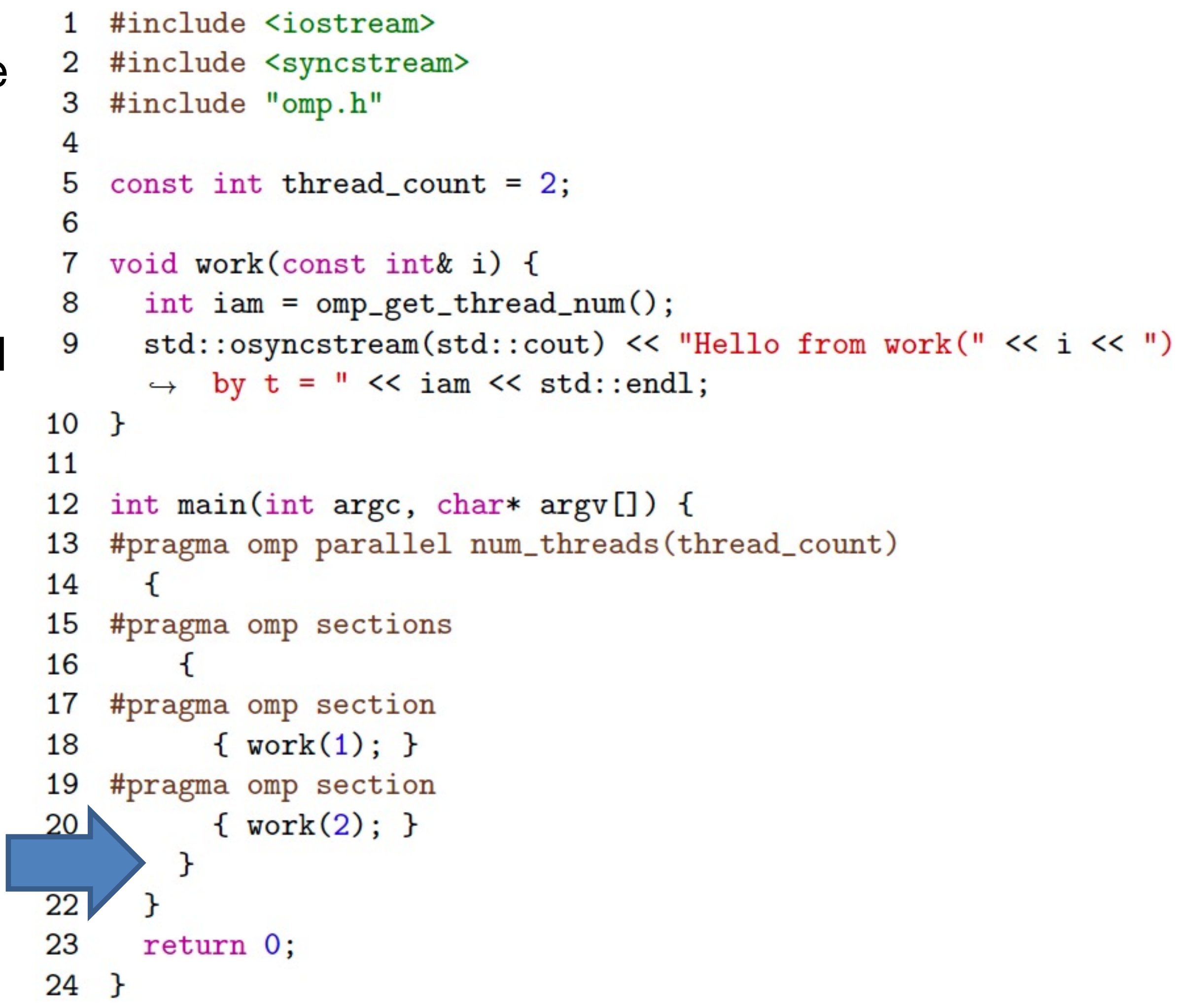
```c
void subdomain(float *x, int istart, int ipoints) {
    int i;
    for (i = 0; i < ipoints; i++) x[istart+i] = 123;
}

void sub(float *x, int npoints) {
    int iam, nt, ipoints, istart;

#pragma omp parallel default(shared)
 ↪  private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt =  omp_get_num_threads();
        ipoints = npoints / nt;    /* size of partition */
        istart = iam * ipoints;  /* starting array index */
        if (iam == nt-1)     /* last thread may do more */
            ipoints = npoints - istart;
        subdomain(x, istart, ipoints);
    }
}

int main() {
    float array[10000];
    sub(array, 10000);
    return 0;
}
```

https://github.com/OpenMP/Examples/blob/main/sources/Example_parallel.1.c

Structuring code

Sections

An alternative, non-iterative structuring of the code is possible with sections. Each section is a block of code executed by one thread of the current thread team (corresponding to the innermost enclosing parallel region). One can use **private, firstprivate, lastprivate**, and **reduction** modifiers, similar to the parallel construct. There is an implied barrier of the **sections** region, unless eliminated by a **nowait** clause.

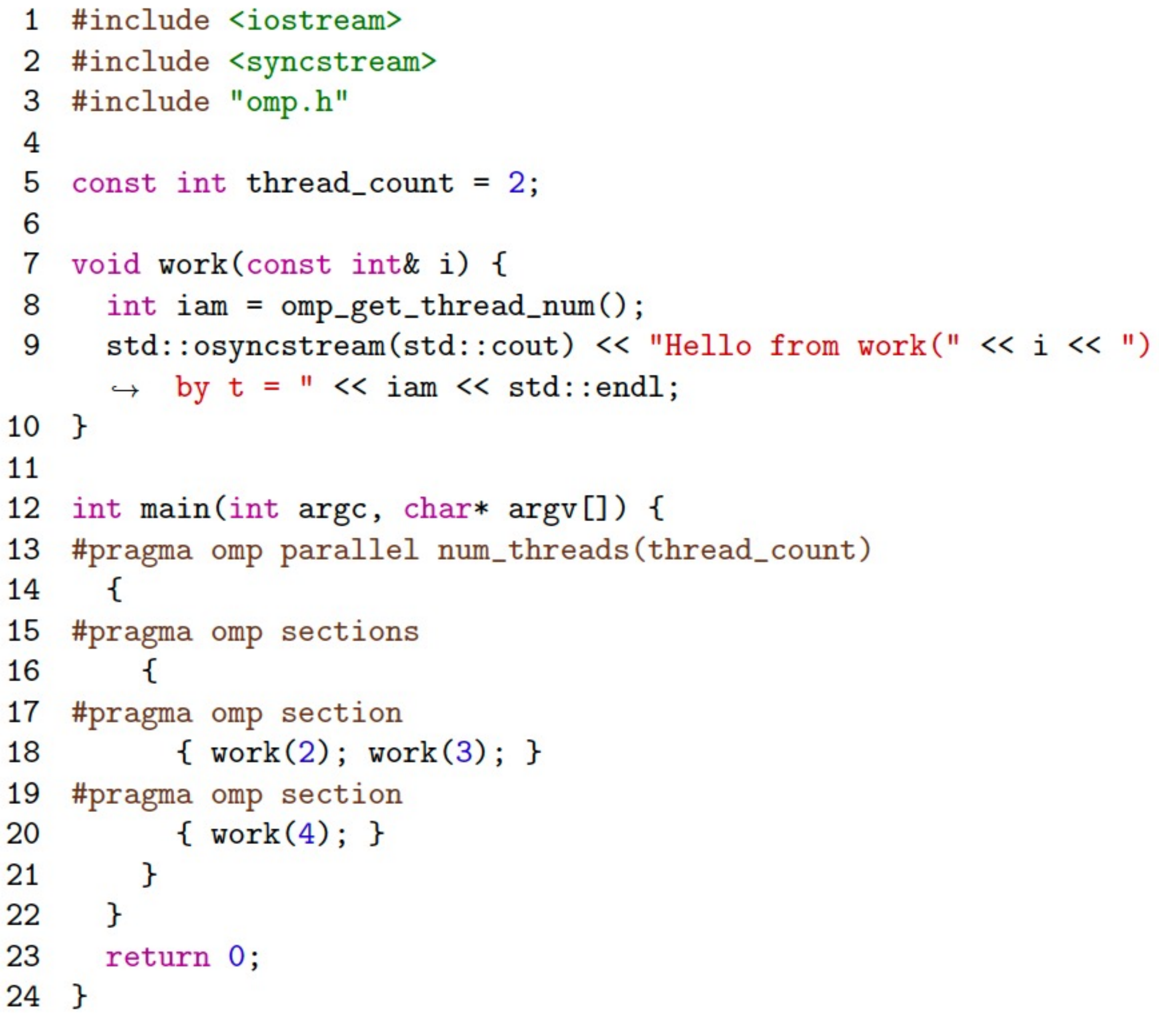


```
#include <iostream>
#include <syncstream>
#include "omp.h"

const int thread_count = 2;

void work(const int& i) {
  int iam = omp_get_thread_num();
  std::osyncstream(std::cout) << "Hello from work(" << i << ") by t = " << iam << std::endl;
}

int main(int argc, char* argv[]) {
#pragma omp parallel num_threads(thread_count)
  {
#pragma omp sections
    {
#pragma omp section
        { work(1); }
#pragma omp section
        { work(2); }
    }
  }
  return 0;
}
```

Structuring code

Sections

An alternative, non-iterative structuring of the code is possible with sections. Each section is a block of code executed by one thread of the current thread team (corresponding to the innermost enclosing parallel region). One can use **private, firstprivate, lastprivate**, and **reduction** modifiers, similar to the parallel construct. There is an implied barrier of the **sections** region, unless eliminated by a **nowait** clause.

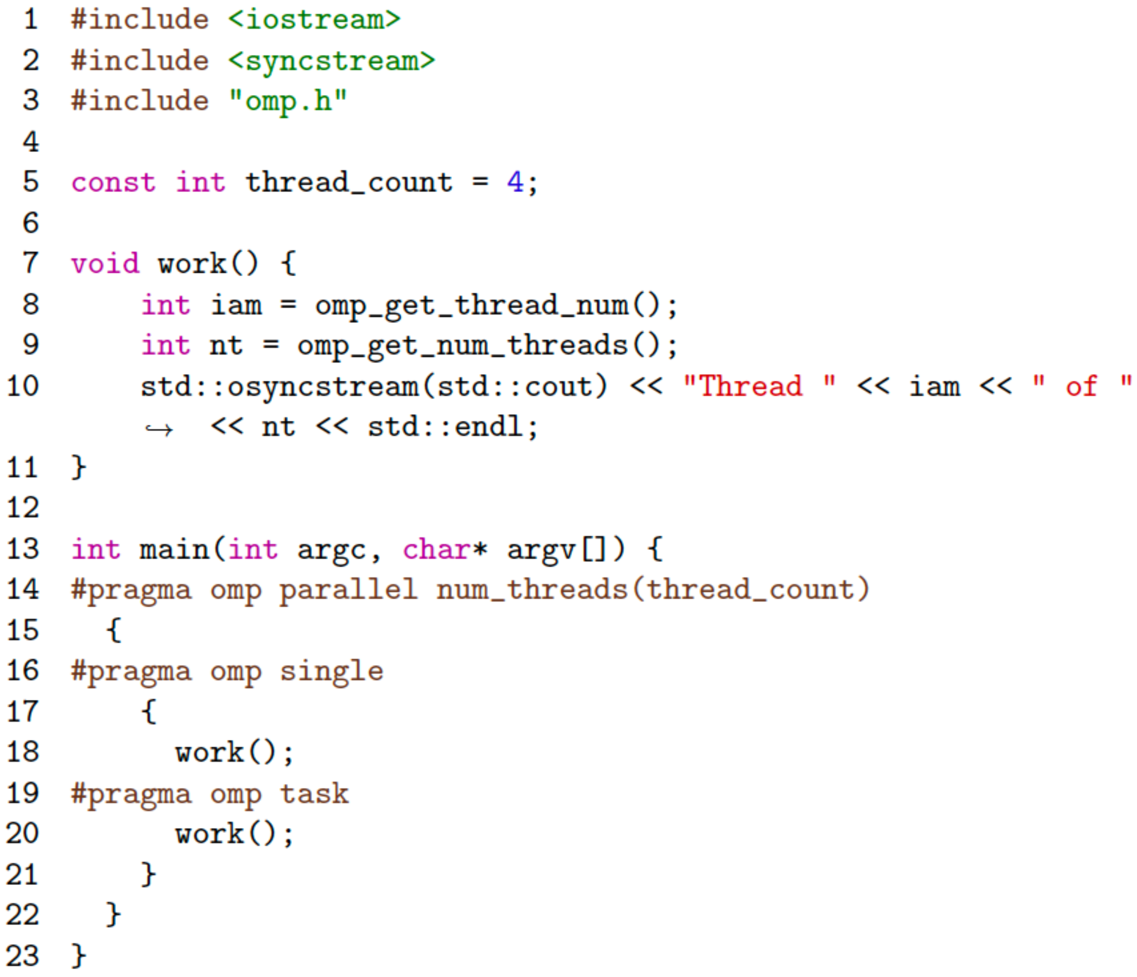


```
#include <iostream>
#include <syncstream>
#include "omp.h"

const int thread_count = 2;

void work(const int& i) {
  int iam = omp_get_thread_num();
  std::osyncstream(std::cout) << "Hello from work(" << i << ") by t = " << iam << std::endl;
}

int main(int argc, char* argv[]) {
#pragma omp parallel num_threads(thread_count)
  {
#pragma omp sections
    {
#pragma omp section
      { work(2); work(3); }
#pragma omp section
      { work(4); }
    }
  }
  return 0;
}
```

Structuring code

Tasks

- The closest to a coroutine in OpenMP is the concept of a task.
- While it does not come with a promise of an implementation with a user-level thread library (cf. Argobots, Converse threads, Qthreads, MassiveThreads, Nanos++, Maestro, GnuPth, StackThreads/MP, Protothreads, Capriccio, StateThreads, TiNy-threads, etc), it often is implemented thus.

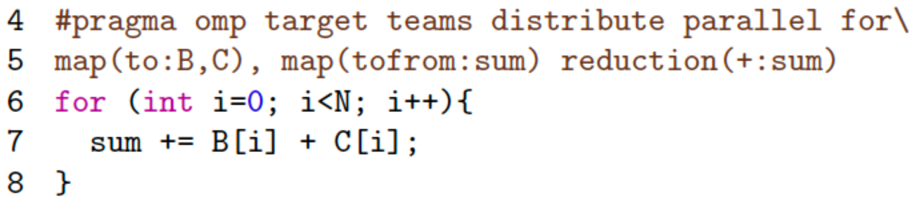


```cpp
#include <iostream>
#include <syncstream>
#include "omp.h"

const int thread_count = 4;

void work() {
    int iam = omp_get_thread_num();
    int nt = omp_get_num_threads();
    std::osyncstream(std::cout) << "Thread " << iam << " of " << nt << std::endl;
}

int main(int argc, char* argv[]) {
#pragma omp parallel num_threads(thread_count)
  {
#pragma omp single
    {
      work();
#pragma omp task
      work();
    }
  }
}
```

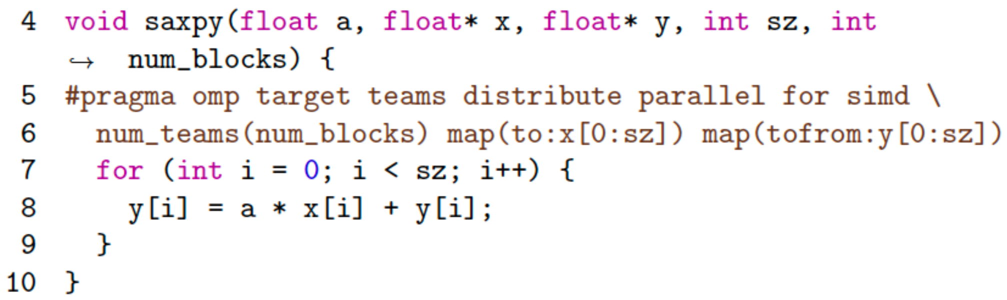
Structuring code

Tasks

- There is a nascent support for the use of GPGPUs via target construct.
- While the block following the target construct can be arbitrary, in principle, most GPUs are not able to support arbitrary code.
- Specifically, there limitations on the use of synchronization primitives and coherence among L1 caches.
- Ideally, one should combine the offloading of the code to the GPGU (via the target), across multiple partitions (via teams construct) etc. This can get quite non-trivial:



```
#pragma omp target teams distribute parallel for\
map(to:B,C), map(tofrom:sum) reduction(+:sum)
for (int i=0; i<N; i++){
  sum += B[i] + C[i];
}
```

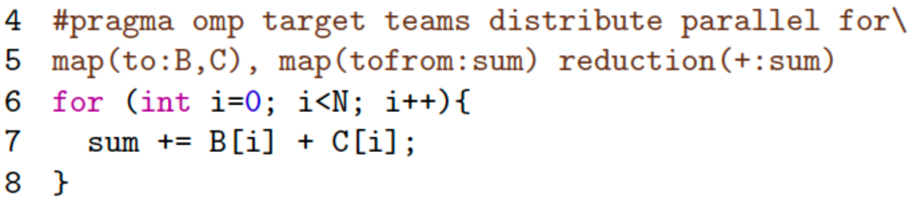


```
void saxpy(float a, float* x, float* y, int sz, int num_blocks) {
#pragma omp target teams distribute parallel for simd \
  num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])
  for (int i = 0; i < sz; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```

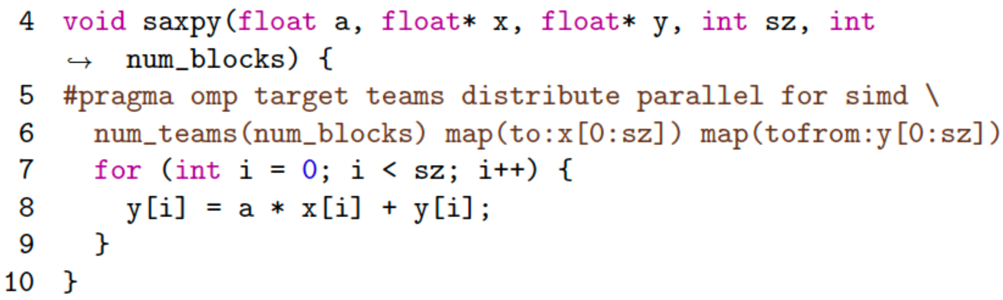
Structuring code

Teams and Targets

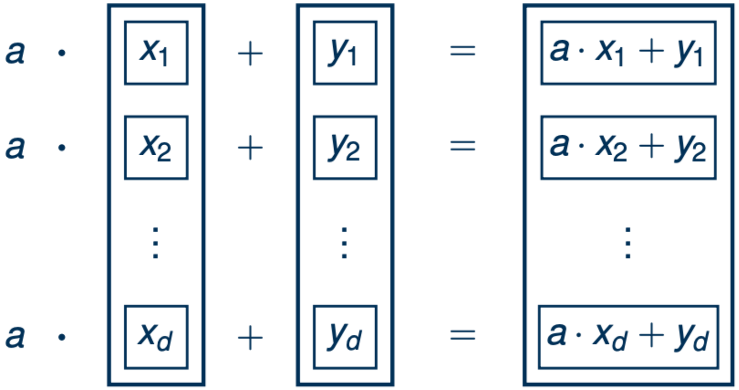
- There is a nascent support for the use of GPGPUs via target construct.
- While the block following the target construct can be arbitrary, in principle, most GPUs are not able to support arbitrary code.
- Ideally, one should combine the offloading of the code to the GPGU (via the target), across multiple partitions (via teams construct.
- We could also use nowait as in other OpenMP constructs, but that does not match the ``lockstep'' execution on the GPGPUs.



```
#pragma omp target teams distribute parallel for\
map(to:B,C), map(tofrom:sum) reduction(+:sum)
for (int i=0; i<N; i++){
  sum += B[i] + C[i];
}
```



```
void saxpy(float a, float* x, float* y, int sz, int
↪  num_blocks) {
#pragma omp target teams distribute parallel for simd \
  num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])
  for (int i = 0; i < sz; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```

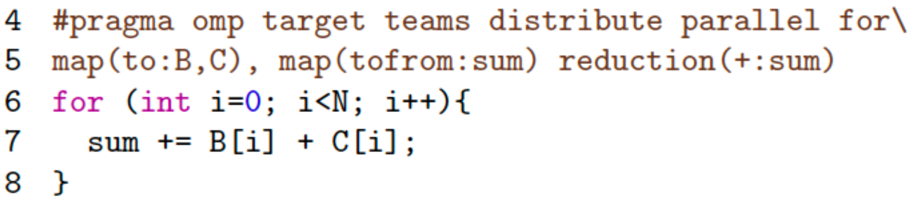


https://hpc-wiki.info/mediawiki/hpc_images/a/ab/GPU_tutorial_saxpy_cuda_c.pdf

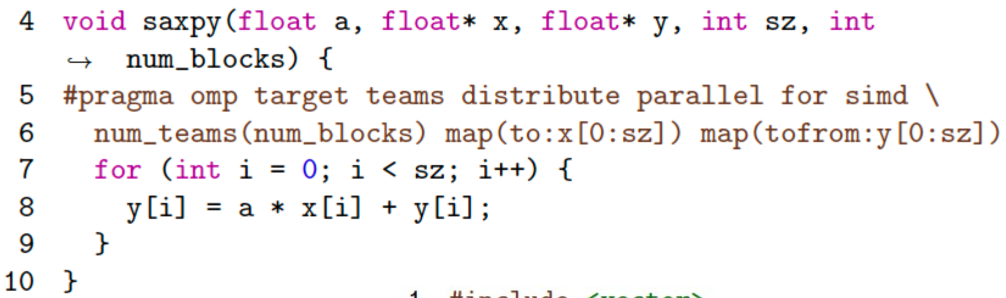
Structuring code

Teams and Targets

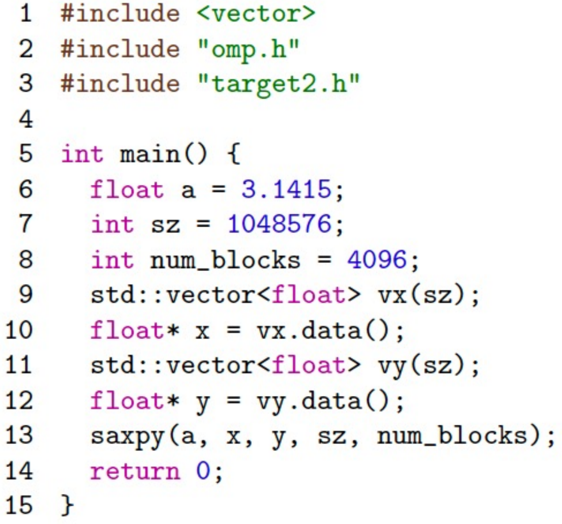
- There is a nascent support for the use of GPGPUs via target construct.
- While the block following the target construct can be arbitrary, in principle, most GPUs are not able to support arbitrary code.
- Ideally, one should combine the offloading of the code to the GPGU (via the target), across multiple partitions (via teams construct.
- We could also use nowait as in other OpenMP constructs, but that does not match the ``lockstep'' execution on the GPGPUs.



```
#pragma omp target teams distribute parallel for\
map(to:B,C), map(tofrom:sum) reduction(+:sum)
for (int i=0; i<N; i++){
  sum += B[i] + C[i];
}
```



```
void saxpy(float a, float* x, float* y, int sz, int
↪  num_blocks) {
#pragma omp target teams distribute parallel for simd \
  num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])
  for (int i = 0; i < sz; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```



```
#include <vector>
#include "omp.h"
#include "target2.h"

int main() {
  float a = 3.1415;
  int sz = 1048576;
  int num_blocks = 4096;
  std::vector<float> vx(sz);
  float* x = vx.data();
  std::vector<float> vy(sz);
  float* y = vy.data();
  saxpy(a, x, y, sz, num_blocks);
  return 0;
}
```

Building the code

Teams and Targets

Compiling OpenMP with offloading is non-trivial. Most likely, you have used **gcc -fopenmp**, so far. Depending on what GPGPUs you wish to target, you may need to switch:

- NVIDIA CUDA Compiler Driver NVCC may be the easiest to use with NVIDIA hardware. For A100 of the RCI cluster, use cc80: **nvcc –mp=gpu -gpu=cc80**
- clang/LLVM compilers uses **clang++ -fopenmp -fopenmp-targets=<target triple>**, where the triple is documented at https://llvm.org/doxygen/Triple_8h_source.html
- AMD ROC is built on top of Clang, so starting with **clang -fopenmp -offload-arch=gfx908** is a good idea.
- Intel Compiler Collection is, since 2021, also built on top of Clang, except still uses the Intel OpenMP library, so you start with **icx -fiopenmp -fopenmp-targets=<target triple>**.

What comes next?

- Structuring code
  - OpenMP Task Region
  - Threads and their Sizing
  - Section
  - Task
  - Kernels, Teams, and Targets

- Synchronization primitives
  - Atomic variables
  - Mutexes and locks
  - Critical sections
  - Barrier
  - Fences and Flushes

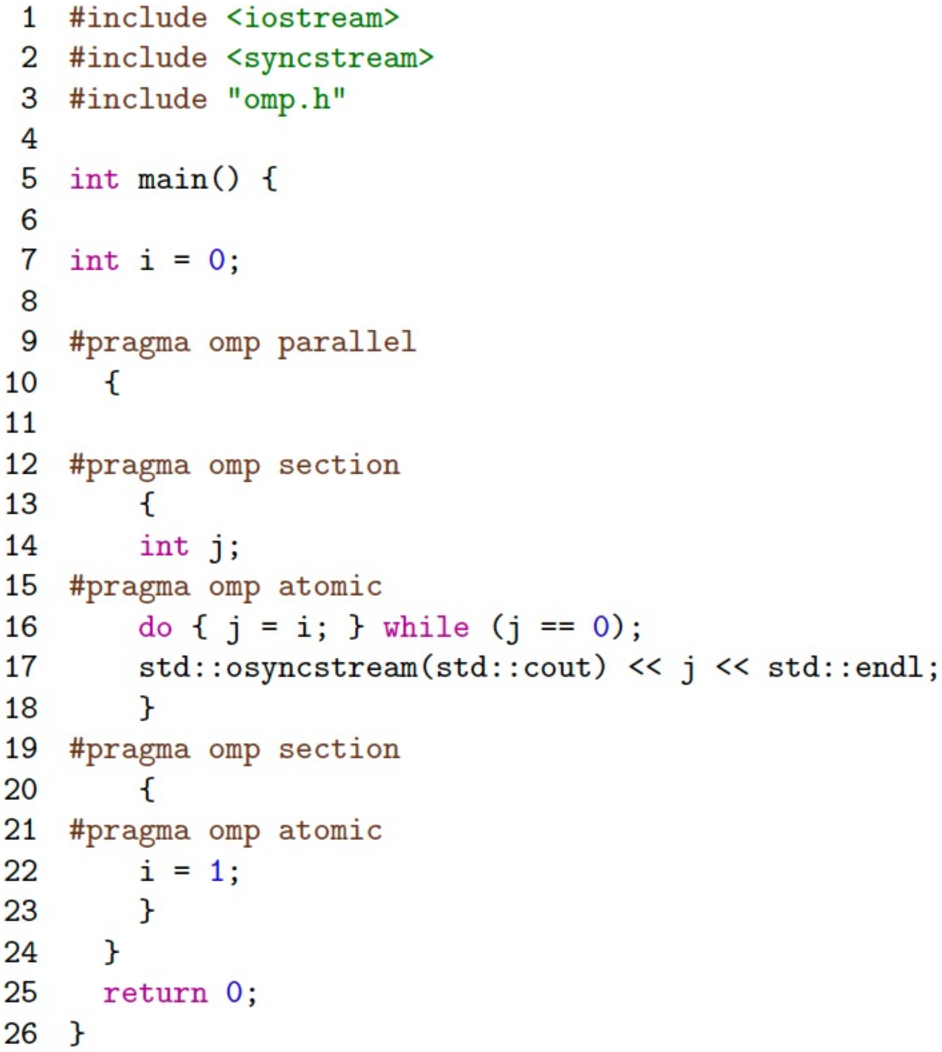
- For each

Synchronisation Primitives

Atomic Variables

OpenMP has a rich support for atomic variables. One can specify:

- operations for which the atomicity is enforced, out of **read, write, update, capture**, out of which update is the default. Capture makes it possible to use operators such as += , e.g., {v = x; x binop= expr; .
- memory order, out of **seq_cst, acq_rel, release, acquire, relaxed** as discussed in two weeks ago. The default is relaxed-consistency shared memory model.
- N.B. use {} after atomic!



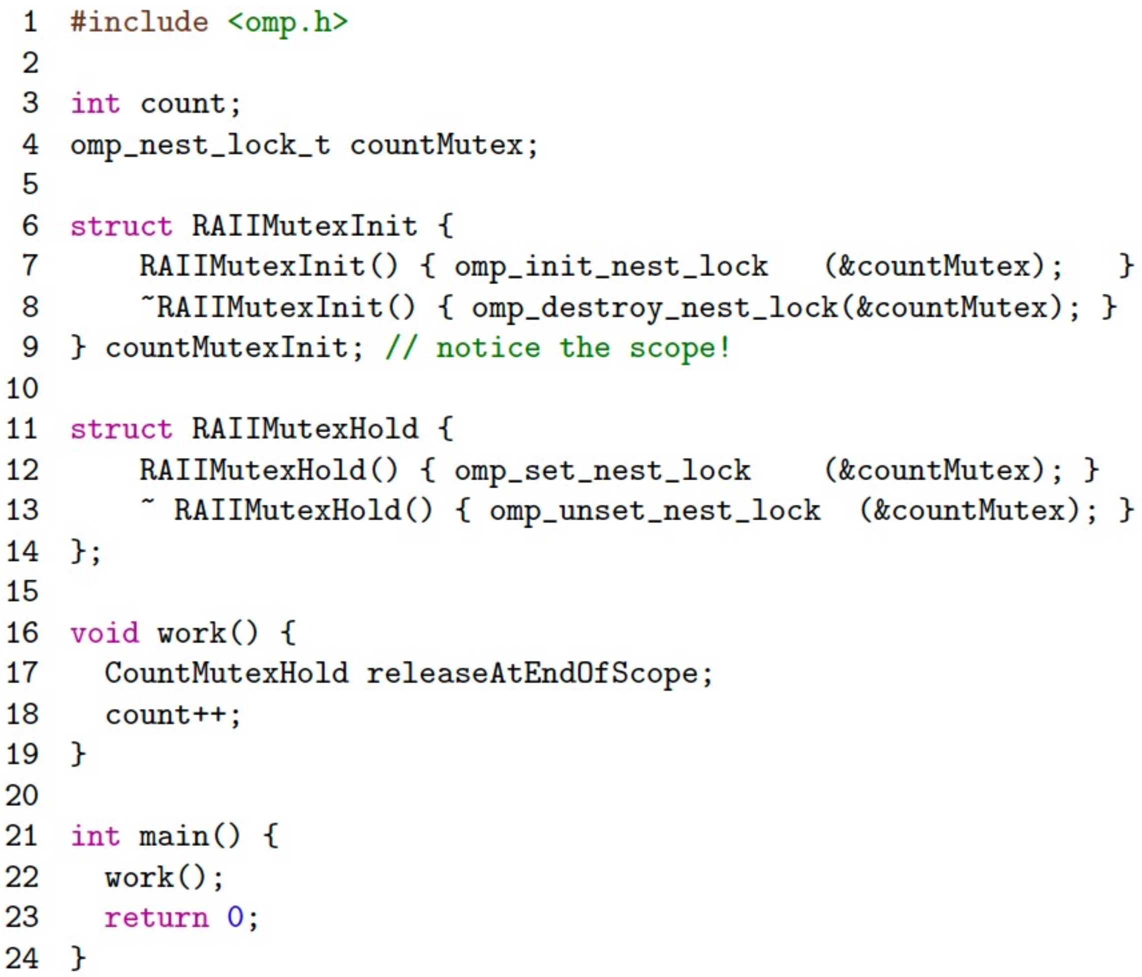
```
#include <iostream>
#include <syncstream>
#include "omp.h"

int main() {

int i = 0;

#pragma omp parallel
  {

#pragma omp section
    {
    int j;
#pragma omp atomic
    do { j = i; } while (j == 0);
    std::osyncstream(std::cout) << j << std::endl;
    }
#pragma omp section
    {
#pragma omp atomic
    i = 1;
    }
  }
  return 0;
}
```

Synchronisation Primitives

Mutexes

OpenMP has only a limited support for mutexes, as in does not support any “resource acquisition is initialization” (RAII) variant,
which makes them hard to use correctly.

One can, however, construct one own's RAII variant.



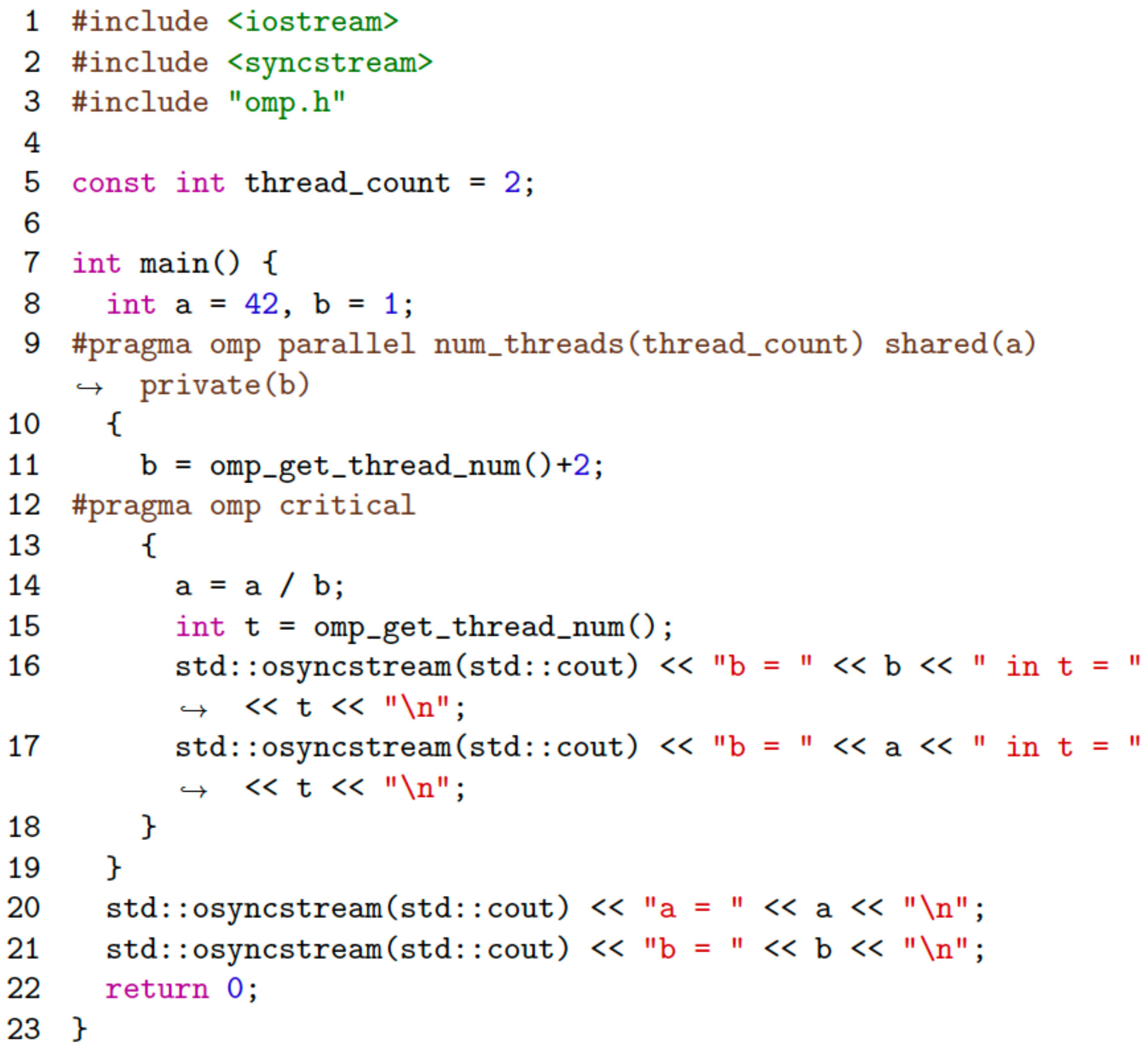
```
#include <omp.h>

int count;
omp_nest_lock_t countMutex;

struct RAIIMutexInit {
    RAIIMutexInit() { omp_init_nest_lock   (&countMutex);   }
    ˜RAIIMutexInit() { omp_destroy_nest_lock(&countMutex); }
} countMutexInit; // notice the scope!

struct RAIIMutexHold {
    RAIIMutexHold() { omp_set_nest_lock    (&countMutex); }
    ˜ RAIIMutexHold() { omp_unset_nest_lock  (&countMutex); }
};

void work() {
  CountMutexHold releaseAtEndOfScope;
  count++;
}

int main() {
  work();
  return 0;
}
```

https://www.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/model-threading-designs/add-parallelism-to-your-program/replace-annotations-with-openmp-code/openmp-locks.html?wapkw=openmp%20locks

Synchronisation Primitives

Critical Sections

- Instead of mutexes, one can safely use critical sections in OpenMP.
- A critical section is a block of code which can be executed by at most one thread at one time.
- This can be used to protect non-trivial non-associative update operations, for which we cannot use reductions.



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

const int thread_count = 2;

int main() {
  int a = 42, b = 1;
#pragma omp parallel num_threads(thread_count) shared(a) private(b)
  {
    b = omp_get_thread_num()+2;
#pragma omp critical
    {
      a = a / b;
      int t = omp_get_thread_num();
      std::osyncstream(std::cout) << "b = " << b << " in t = " << t << "\n";
      std::osyncstream(std::cout) << "b = " << a << " in t = " << t << "\n";
    }
  }
  std::osyncstream(std::cout) << "a = " << a << "\n";
  std::osyncstream(std::cout) << "b = " << b << "\n";
  return 0;
}
```

Synchronisation Primitives

Barrier

- OpenMP provides a straightforward, explicit barrier construct.
- Especially with nested parallel regions, the behaviour can be quite non-trivial. All threads of the current team must complete execution of all tasks bound to the same parallel region prior to continuing past the barrier. What is the current team, however, e.g., whether it is created by the innermost enclosing parallel region, depends on the settings of the nesting.
- Barrier is also implied by the entry and exit in **parallel** regions. There is also an implicit barrier at the end of a **for, sections, single, scope**, and **workshare** constructs, unless one explicitly adds a nowait clause.

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

void work() {
    std::osyncstream(std::cout) << "a";
#pragma omp barrier
    std::osyncstream(std::cout) << "A";
}

int main() {
#pragma omp parallel num_threads(5)
    work();
}
```

Synchronisation Primitives

Barrier

- OpenMP provides a straightforward, explicit barrier construct.
- Especially with nested parallel regions, the behaviour can be quite non-trivial. All threads of the current team must complete execution of all tasks bound to the same parallel region prior to continuing past the barrier. What is the current team, however, e.g., whether it is created by the innermost enclosing parallel region, depends on the settings of the nesting.
- Barrier is also implied by the entry and exit in **parallel** regions. There is also an implicit barrier at the end of a **for, sections, single, scope**, and **workshare** constructs, unless one explicitly adds a nowait clause.

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

void work(int n) { std::osyncstream(std::cout) << n; }
void sub3(int n)
{
    work(n);
#pragma omp barrier
    work(n);
}
void sub2(int k)
{
#pragma omp parallel shared(k)
    sub3(k);
}
void sub1(int n)
{
    int i;
#pragma omp parallel private(i) shared(n)
    {
#pragma omp for
        for (i=0; i<n; i++)
            sub2(i);
    }
}
int main()
{
    sub1(2);
    sub2(2);
    sub3(2);
    return 0;
}
```

Synchronisation Primitives

Fences and Flushes

As we have suggested above, the default memory ordering is relaxed.

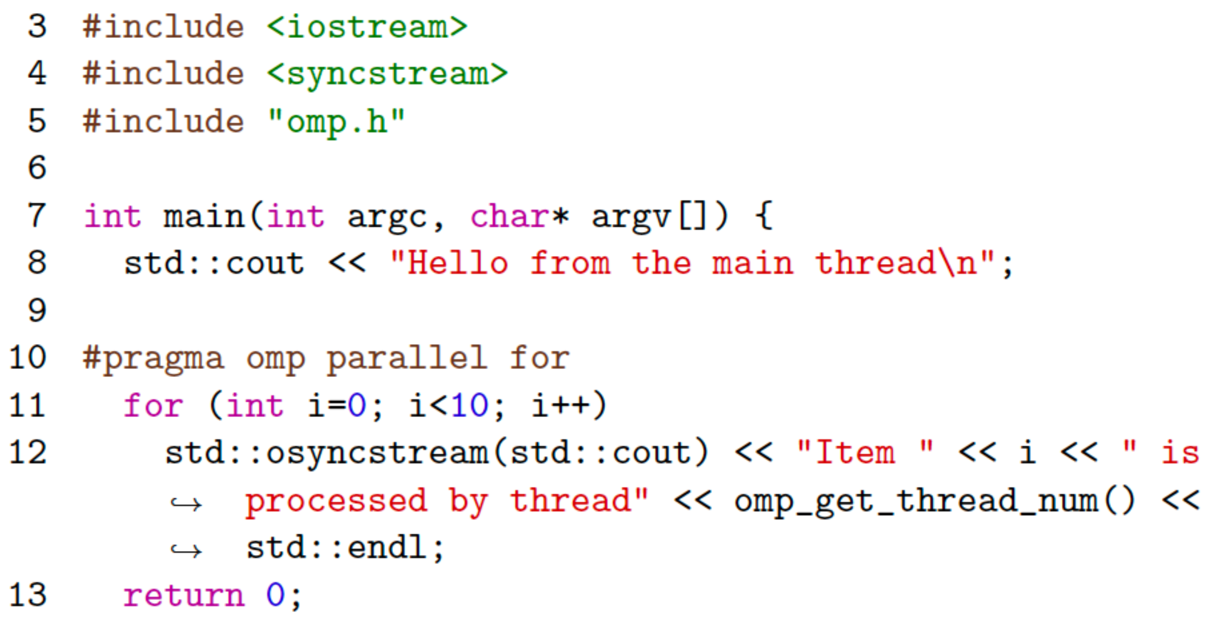
Similarly to a memory fence, the **flush** construct provides point at which a thread is guaranteed to have a consistent view of memory.

The flush is also implied by the entry and exit in parallel regions, critical regions, operations with locks etc.

Algorithms

Last but not least, the most frequent “idiom”

While OpenMP does not really implement any algorithms, some of the data-parallel constructs are similar to algorithms in the STL library. Notably, OpenMP makes it possible to use “for each”.

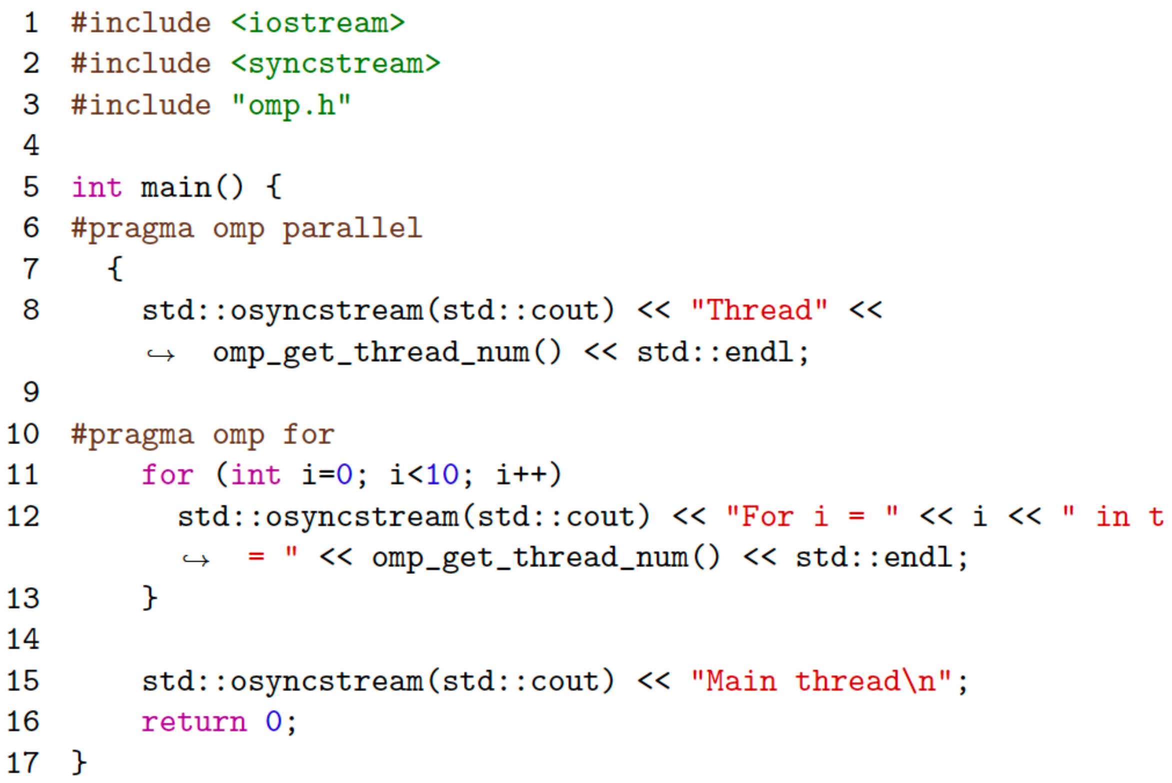


```
#include <iostream>
#include <syncstream>
#include "omp.h"

int main(int argc, char* argv[]) {
  std::cout << "Hello from the main thread\n";

#pragma omp parallel for
  for (int i=0; i<10; i++)
    std::osyncstream(std::cout) << "Item " << i << " is processed by thread" << omp_get_thread_num() << std::endl;
  return 0;
```

Algorithms

Last but not least, the most frequent “idiom”

While OpenMP does not really implement any algorithms, some of the data-parallel constructs are similar to algorithms in the STL library. Notably, OpenMP makes it possible to use “for each”.



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

int main() {
#pragma omp parallel
  {
    std::osyncstream(std::cout) << "Thread" <<
    ↪  omp_get_thread_num() << std::endl;

#pragma omp for
    for (int i=0; i<10; i++)
      std::osyncstream(std::cout) << "For i = " << i << " in t
      ↪  = " << omp_get_thread_num() << std::endl;
    }

    std::osyncstream(std::cout) << "Main thread\n";
    return 0;
}
```

Exercises

A simple exercise is to go through the examples at:
https://github.com/OpenMP/Examples
to understand the syntax in full.

A more demanding exercise may be to consider a simple ray tracer
https://github.com/ssloy/tinyraytracer/blob/master/tinyraytracer.cpp
with basic OpenMP parallelization and to try to target a GPGPU.

Backup Slides

Let us briefly consider a particular example of the NVIDIA Ampere architecture of GeForce RTX 3080 or NVIDIA A100:

- There are seven Graphics Processing Clusters (GPCs), sharing up to 40 MB of L2 cache and up to 40 GB of high-speed HBM2 memory
- Within each GPC, there are 12 Streaming Multiprocessors (SMs),
- Within each SM, there are 128 cores working with single-precision floating-point (FP32) precision and two double-precision (FP64) units. There is also 128 KB of L1/Shared Memory, shared across the 128 cores.
- Each SM is partitioned into four processing blocks (or partitions), each with a few kilobytes of L0 instruction cache and one warp scheduler.

Altogether, A100 has 10752 cores, but their use is rather constrained. Each warp can have at most 32 threads and runs them in lock-step on one processing blocks. Each streaming multiprocessors can run at most 64 warps of 32 threadblocks (i.e., 2048 threads per SM). Further constraints are due to the register use: each thread can use at most 255 registers, but there are only 65536 32-bit registers for the SM (yielding a limit of 257 threads per SM, at full register utilization). Further constraints are due to the use of memory hierarchy (esp. the 128 KB of shared memory, shared across the 128 cores).

Backup Slides

Similar to the CPU, the GPU hence has a memory hierarchy:

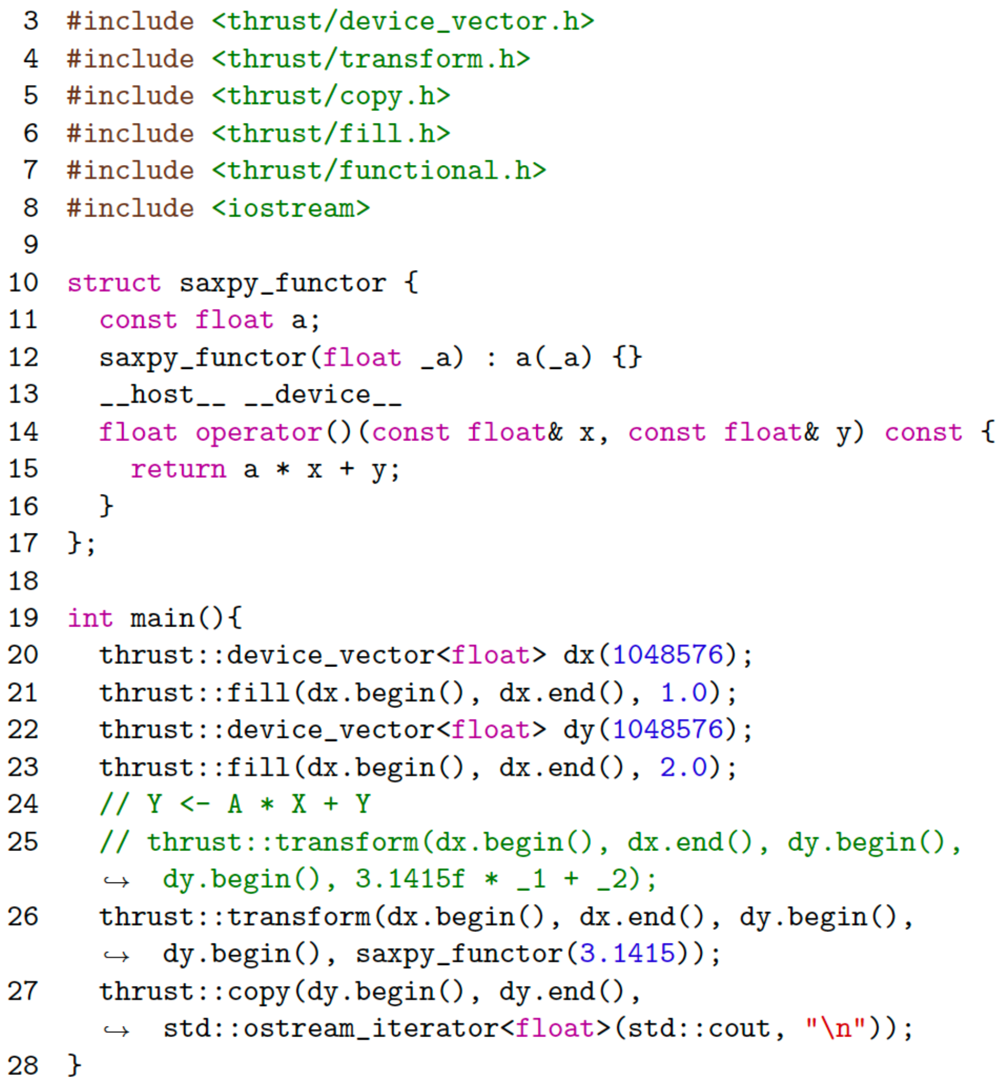
- L1 cache with 33 cycle latency and shared memory with even lower latency, based on microbenchmarking
- L2 cache with up to 2080 GB/s read bandwidth (200 cycle latency), based on microbenchmarking
- on-board HBM2 memory with 1555 GB/sec bandwidth (290 cycle latency)
- intra-board NVLink with 50 Gb/sec per signal pair bandwidth
- access to RAM via PCI Express Gen 4 (PCIe Gen 4) at 31.5 GB/sec
- optionally, intra-node communication at 200 Gbit/sec using InfiniBand.

The interaction of the GPGPU memory hierarchy and CPU memory hierarchy is non-trivial, but summarized by the suggestion to reduce the number and volume of transfers between the host and the GPGPU, even at the price of increasing the volume of computation substantially. (Compare the numbers above to M.2 PCIe Gen4 SSDs with 7 GB/sec bandwidth.)

Backup Slides

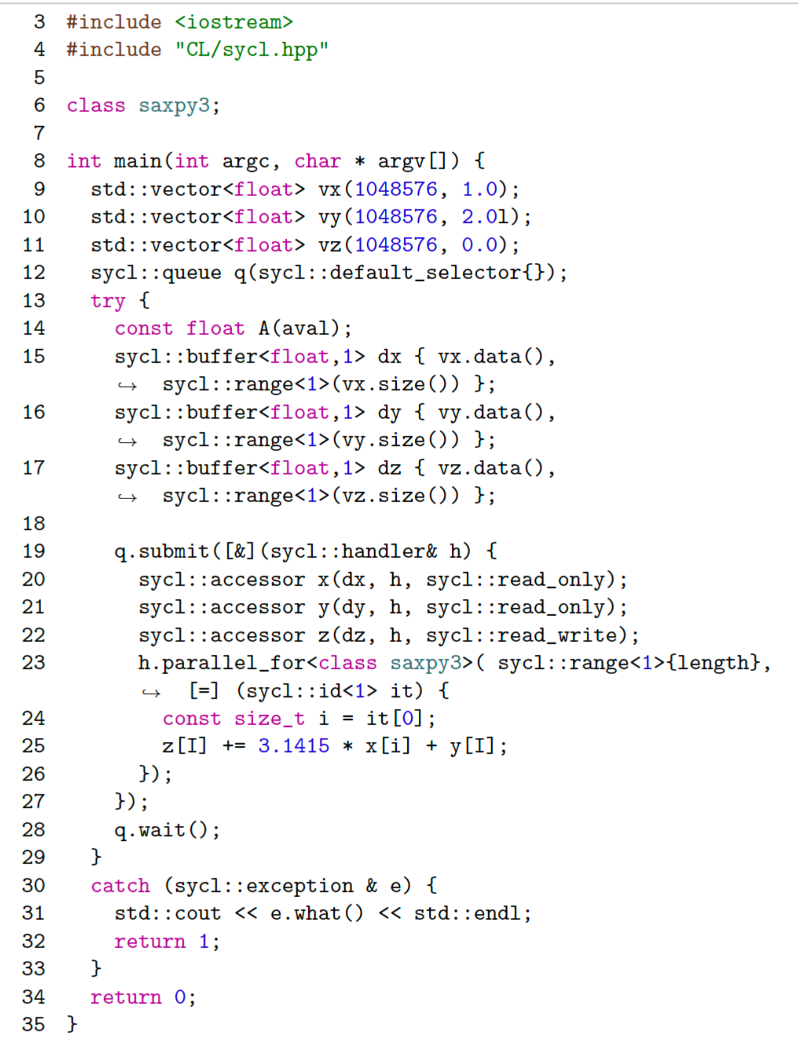
```
#include <vector>
#include <iostream>
#include <cuda.h>
#define N 1048576

__global__ void saxpy_kernel(float a, float* x, float* y, float* z){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(){
  std::vector<float> vx(N);
  float* x = vx.data();
  std::vector<float> vy(N);
  float* y = vy.data();
  std::vector<float> vz(N);
  float* z = vz.data();
  float *dx, *dy, *dz;
  cudaMalloc(&dx, N*sizeof(float));
  cudaMalloc(&dy, N*sizeof(float));
  cudaMalloc(&dz, N*sizeof(float));
  cudaMemcpy(dx, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(dy, y, N*sizeof(float), cudaMemcpyHostToDevice);
  int nblocks = (n + 255) / 256;
  saxpy_kernel<<<nblocks, 256>>>(3.1415, dx, dy, dz);
  cudaMemcpy(z, dz, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(dx);
  cudaFree(dy);
  cudaFree(dz);
  // we do not need free(x), free(y), free(z)
}
```

Backup Slides



```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/functional.h>
#include <iostream>

struct saxpy_functor {
  const float a;
  saxpy_functor(float _a) : a(_a) {}
  __host__ __device__
  float operator()(const float& x, const float& y) const {
    return a * x + y;
  }
};

int main(){
  thrust::device_vector<float> dx(1048576);
  thrust::fill(dx.begin(), dx.end(), 1.0);
  thrust::device_vector<float> dy(1048576);
  thrust::fill(dx.begin(), dx.end(), 2.0);
  // Y <- A * X + Y
  // thrust::transform(dx.begin(), dx.end(), dy.begin(), dy.begin(), 3.1415f * _1 + _2);
  thrust::transform(dx.begin(), dx.end(), dy.begin(), dy.begin(), saxpy_functor(3.1415));
  thrust::copy(dy.begin(), dy.end(), std::ostream_iterator<float>(std::cout, "\n"));
}
```

Backup Slides



```
#include <iostream>
#include "CL/sycl.hpp"

class saxpy3;

int main(int argc, char * argv[]) {
  std::vector<float> vx(1048576, 1.0);
  std::vector<float> vy(1048576, 2.0l);
  std::vector<float> vz(1048576, 0.0);
  sycl::queue q(sycl::default_selector{});
  try {
    const float A(aval);
    sycl::buffer<float,1> dx { vx.data(),
    ↪  sycl::range<1>(vx.size()) };
    sycl::buffer<float,1> dy { vy.data(),
    ↪  sycl::range<1>(vy.size()) };
    sycl::buffer<float,1> dz { vz.data(),
    ↪  sycl::range<1>(vz.size()) };

    q.submit([&](sycl::handler& h) {
      sycl::accessor x(dx, h, sycl::read_only);
      sycl::accessor y(dy, h, sycl::read_only);
      sycl::accessor z(dz, h, sycl::read_write);
      h.parallel_for<class saxpy3>( sycl::range<1>{length},
      ↪  [=] (sycl::id<1> it) {
        const size_t i = it[0];
        z[I] += 3.1415 * x[i] + y[I];
      });
    });
    q.wait();
  }
  catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
  }
  return 0;
}
```