OpenMP Parallelism

Notes and Samples on OpenMP Protocol

20th August, 2021

Ed Chen

1 Overview

1.1 Introduction

Motivation behind parallel computing stems from hardware focused performance improvements are running up against a power wall, since as normalized performance goes up power goes up nearly quadratically. Splitting computations into multiple processes reduces this power load through $P = CV^2F$ and gives us tremendous power savings. Hence there is a trend of putting more cores on pieces of silicon and the burden of improving performance is shifting to the software side, which is accomplished by parallel computing.

Concurrency is when tasks are *logically* active simultaneously, parallelism is where the tasks are *actually* executed simultaneously. Concurrent applications are applications which are inherently designed to be concurrent, take web servers. Parallel applications are applications where the computation is parallelized to run faster, but can still be considered in a sequential sense. The latter is what OpenMP is designed for. The bulk of the algorithm design work is figuring out where algorithms can be split to run in parallel, the end portion is just actually implementing it an API for multithreaded applications, like OpenMP.

Basic syntax of OpenMP in C is dictated by compiler directives of the form `#pragma omp construct [clause[clause]...]`, for instance some four-thread parallelism `#pragma omp parallel num_threads(4)`. Most constructs related to a structured block, which has one entry and exit point, compartmentalizing code in a sense.

1.2 OpenMP Compiler

On Linux with gcc, execute something with OpenMP using `gcc -fopenmp foo.c` and dictate the number of threads within, for instance in bash shell this is `export OMP_NUM_THREADS = 4` and run `./a.out` for the code itself. To check if this works, make a simple "Hello World" program that executes print statements on parallel threads.

```
#include <omp.h>
int main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello(%d)", ID);
        printf(" world(%d)", ID);
    }
}
```

Compiling and executing this gives accordingly a sequence of print statements executed on a default number of threads, whatever that is on your system.

```
hello(0) world(0)hello(1) world(1)hello(3) world(3)hello(2) world(2)
```

1.3 Memory and Synchronization

In shared memory machines, there are two classes that are mainly differentiated by their memory architectures. The *SMP*, Symmetric Multiprocessor, has a shared address space with equal time access for each processor, in other words, each processor has access to the shared memory with the same efficiency. On the other hand, *NUMA*, Non Uniform Address Space Multiprocessor, provides different access costs for different memory regions, in other words there are "closer" and "farther" memory regions.

Most current systems are not true SMPs, so we should keep in mind which memory regions we are accessing in algorithms. Figuring out how to structure algorithms to constrain memory accesses to certain caches is an important part to parallelizing, to see these available cache sizes, use `lscpu` on Linux and `system_profiler SPHardwareDataType` for the L2/L3 cache and `sysctl -a | grep hw.l1` for the L1 cache. Unlike main memory which is expandable, caches are fixed with the L1 and L2 caches on the chips itself, while the L3 cache is shared between all cores.

```
iMac:~ Ed$ system_profiler SPHardwareDataType;
Hardware:

    Hardware Overview:
      Model Name: iMac
      Model Identifier: iMac18,3
      Processor Name: Intel Core i5
      Processor Speed: 3.4 GHz
      Number of Processors: 1
      Total Number of Cores: 4
      L2 Cache (per Core): 256 KB
      L3 Cache: 6 MB
      Memory: 8 GB
      Boot ROM Version: 175.0.0.0.0
      SMC Version (system): 2.41f1

iMac:~ Ed$ sysctl -a | grep hw.l1
hw.l1icachesize: 32768
hw.l1dcachesize: 32768
```

For instance, in a dense matrix matrix multiplication algorithm $C \leftarrow C + A * B$ we can do cache blocking that splits the larger matrix into working sets where we optimize for the number of fused-multiply-add operations that can be executed within those working sets to minimize memory movement. So with a cache size of Z, say we can execute some maximum number of M operations with this cache, requiring $2Z$ memory movements to and from cache and main memory. As we know, $C \leftarrow C + A * B$ requires $2n^3$ operations, so we'd need to do this $2n^3/M$ times. Hence this gives us a total of $(2n^3/M)2Z$ memory movements. To express an upper bound on M we have $M = \#$ of operations $\leq \sqrt{|S_A| \cdot |S_B| \cdot |S_C|}$ where the subsets of matrices A, B, C are denoted by S_A, S_B, S_C. At most, we can have Z entries of any of these matrices within the cache, this gives us $M \leq \sqrt{Z^3}$. Plugging in, we have $4n^3/\sqrt{Z}$ memory movements, or a model of $\Omega(n^3/\sqrt{Z})$.

In shared memory programming, we have some number of threads that are executing in a shared address space, which interleave in terms of which things are executing before others. In essence, this is why the aforementioned hello world program yields the "hello" and "world" texts in a different way every execution. Unlike distributed programming, threads can directly communicated here by referring to variables within the shared address space. Hence, *synchronization* is the process of controlling conditions to prevent data conflicts between different threads, in turn preventing *race conditions* which is when multithreaded code yields different results upon different thread schedules. Synchronization can be thought as analogous to blocking in MPI, in short too much synchronization is going to result in excessive downtime, and low speedup.

2 OpenMP Constructs

2.1 Threads

OpenMP is built on fork-join parallelism, where your algorithm starts as a single thread, which then happens upon a point where multiple threads will help with efficiency, ie a loop. The master thread then splits off to ID 0 and the other threads are numbered accordingly. This collection is called a *team of threads* which work in parallel. When they are finished executing, they join back to the single master thread, which then continues until we find another place where parallelism would be beneficial and the process repeats. For instance, in a relatively standard unstructured mesh finite element model, I'd pinpoint three main places where parallelism would be beneficial - in the mesh determination, then the constituent matrix formulation/stamping, and the final matrix computations. Furthermore, we can use nested threads where a parallel forked thread can create its own team of threads. Fundamentally, this is what we are using in OpenMP. The **only way to create threads** in OpenMP is to use the parallel construct.

Say we have an array `double A[100];`, and we request 4 threads with `omp_set_num_threads(4);`. Then, each thread gets its id with `omp_get_thread_num();`, which can then be used to specify which portion the specific thread is dealing with. For instance, this is immediately obvious in bitonic sort where each thread would use its id to refer to which portion its dealing with. Logically, if I allocate data outside the structured block it is visible to all the threads, whereas data inside the structured block are private/local to the thread. Pragma deals with the low-level details of this in the background, and just creates the number of requested threads and joins them upon completion. The master thread creates all the offshoot threads before executing as a thread itself.

For instance, a example would be using parallelism to get an approximation on pi, which can be obtained by solving the integral $\int_0^1 \frac{40}{1-x^2}dx$. Using a strategy called *SPMD*, Single Program Multiple Data, we can use a single program on multiple threads, which all execute it based on the ID thread. There are many ways of splitting this loop between the threads, though two mentioned are splitting it into blocks or in a cyclic fashion, think $\{1,2,3,4,5\}$ and $\{5,6,7,8,9,10\}$ as opposed to $\{1,3,5,7,9\}$ and $\{2,4,6,8,10\}$.

It is also important to keep in mind the scope of the variables, differentiating between the shared variables and the local variables for each thread. For instance, we have to declare the iterator i again in the local scope for each thread so they are not constantly overriding the global iterator which is shared across all threads. This can similarly be said for the thread ID, number of threads, etc. Note that we lose these variables after the parallel region, so we need to draw out whatever useful results we need into a shared variable before ending the parallel region. A common solution for this would be to create an array of sums with a size corresponding to the number of threads and simply adding it up after the parallel region.

Also, we should always refer to the number of threads with the `omp_get_num_threads();` command instead of the shared variable, the reason for this being we may not always get the number of threads we want. If I request 4 threads, the computer, may for a variety of reasons - ie availability or allocation, not allot that many. At this stage, the scalability of this program is still quite bad though, since the shared sum array variable is on the same cache line and is getting requested and overwritten by each of the threads even though there is no sharing, each thread has its own separate index it is writing to. This is known as cache sloshing and false sharing which can lead to peformance hits. A very ugly way of solving this is to make a 2d array where the second dimension is simply the cache line size so we can guarantee each element of this sum array is sitting on a different cache line.

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

static long num_steps = 1000000;
double step;

void main()
{
    int NUM_THREADS = 5;
    int i; double x, parallel_pi, serial_pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);

    // parallel pi calculation
    #pragma omp parallel
    {
        int i;
        int steps_per_thread = floor(num_steps/omp_get_num_threads());
        int ID = omp_get_thread_num();
        for (i=ID*steps_per_thread+1; i < (ID+1)*steps_per_thread; i++){
            x = (i+0.5)*step;
            sum[ID] += 4.0/(1.0+x*x);
        }
    }
    for (i=0; i<NUM_THREADS; i++){
        parallel_pi += step*sum[i];
    }

    // serial pi calculation
    for (i=1; i<num_steps; i++) {
        x = (i+0.5)*step;
        serial_pi += 4.0/(1.0+x*x);
    }

    printf("Serial Pi approximation yields %f\n", serial_pi*step);
    printf("Parallel Pi approximation yields %f\n", parallel_pi);
}
```

2.2 Synchronization

Another way, and the preferred way, of dealing with this issue is known as synchronization, one of the key constructs of OpenMP defined earlier. There are two basic ways of synchronization in OpenMP, *Barrier Synchronization* and *Mutual Exclusion Synchronization*. For Barrier Synchronization, we essentially introduce a barrier saying all threads must complete some step before progressing past a certain point, think similar to MPI blocking communication. This is implemented with the *barrier* construct `#pragma omp barrier` which inserts a barrier that makes sure no thread goes past this point until all threads are completed with the prior portion. Note that this has the possibility to introduce lots of overhead if the work distribution on each thread is too uneven, so use consciously. So, for instance

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    A[id] = big_calculation_1(id);
    #pragma omp barrier
    B[id] = big_calculation_2(id);
}
```

For Mutual Exclusion, the easiest way to deal with this is the *critical* construct, `#pragma omp critical`. In mutual exclusion, when a thread gets to the critical portion it will "hold" the section and other threads will wait until it gets released, meaning only one thread at a time will execute that code. Again, there's a lot of overhead here so make sure it's absolutely necessary. Below, we can see that the big calculation is executed before we update the result, which only one thread will access at a time, otherwise we will run into issues with shared variable overrides. Note this can also take a structured block, `#pragma omp critical{...}` and will apply accordingly to everything within. Basically this is just serializing a portion of your parallel program.

```
#pragma omp parallel
{
    float B; int i, id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for (i = id; i < niters; i += nthrds) {
        B = big_job(i);
        #pragma omp critical
        res += consume(B);
    }
}
```

For the atomic construct, the basic version operates on the concept that there are certain constructs within hardware for doing quick memory updates since low-level things need really efficient mutual exclusion. Atomic is saying if these constructs are available to use them, otherwise just use the mutual exclusion. For instance, read-available-write or an iterator. Critical is more general, but simple updates are fine for atomic.

```
#pragma omp parallel
{
    double tmp, B;
    B = doing_something();
    tmp = big_ugly(B);
    #pragma omp atomic
    X += tmp;
}
```

These are the three major high-level synchronization constructs that are most commonly used in OpenMP. Note that for our aforementioned pi program, we can then not need to use the additional shared sum array and filling cache lines, instead introducing the critical barrier for the sum, which is needed to avoid multiple threads trying to change sum at the same time. The advantage of this as opposed to the cache blocking since its portable across different systems. Importantly, note that if we had put the critical barrier within the for loop we would've nearly serialized the program, make sure to put synchronizations where it will not remove as much parallelism.

```
for (i=ID*steps_per_thread+1; i < (ID+1)*steps_per_thread; i++){
    x = (i+0.5)*step;
    local_sum += 4.0/(1.0+x*x);
}
#pragma omp critical
    parallel_pi += step*local_sum;
```

2.3 Worksharing

Most common model that is used in scientific computing is still SPMD (Single Program, Multiple Data). Sometimes we want to share a single construct across all threads. There are loop, section, single, and task constructs for worksharing. We'll first consider the loop construct. Essentially, using the `#pragma omp for` (do in Fortran), we are just asking OpenMP to do all the thread ID-ing and splitting of the for loop for us.

```
#pragma omp parallel
{
    #pragma omp for
    for (I = 0; I < N; I++) {
        neat_stuff(I);
    }
}
```

Initially when OpenMP was being created, there were a portion of the developers who wanted to leave it to the above mentioned constructs in order to keep it as a very lightweight library. Through introducing these worksharing constructs however, we are able to significantly simplify the rather mundane portions of parallelizing loops, however notice we lose ability to dictate which threads will get to execute which portions of the loop. Hence, there is a *schedule* clause to the loop worksharing construct. We have five main schedule clause options:

`schedule(static [,chunk])`	deals out blocks of iterations of size chunk, default 1 if not provided, to each thread. dealt with in *compile time*, use for predictable scheduling logic
`schedule(dynamic [,chunk])`	depending on how much load each thread experiences, each progressively grabs iterations of size chunk off a queue until everything finishes. dealt with in *runtime*, use for complex scheduling logic
`schedule(guided, [,chunk])`	starts from a larger dynamic chunk size and starts progressively shrinking each chunk as calculation proceeds, pretty uncommon nowadays
`schedule(runtime)`	gets schedule and chunk size from the `OMP_SCHEDULE` environment variable so user can change on the fly
`schedule(auto)`	whatever the runtime thinks is best (does not have to be one of the aforementioned)

Also, sometimes we just need to use these two constructs with OpenMP, so we can simplify the `#pragma omp parallel{#pragma omp for ...}` constructs into `#pragma omp parallel for`, which is pretty commonly used in practice.

2.4 Parallel Loops

So to recap, what does the typical process in parallelizing a loop look like? First identify the compute-intensive loops, identify if there is concurrency, modify the concurrency, and make it parallel. For instance, the concurrency in

```
int i, j, A[MAX];
j = 5
for (i = 0; i < MAX; i++) {
    j += 2
    A[i] = big(j)
}
```

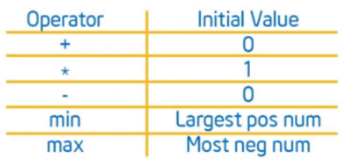
there is concurrency in that j is not dependent off of the iteration, so we can move it to be determined within the loop, computed from the value of the iteration. This is called a *loop carry dependency.*

```
int i, j, A[MAX];
#pragma omp parallel for
for (i = 0; i < MAX; i++) {
    int j = 5+2*(i+1);
    A[i] = big(j)
}
```

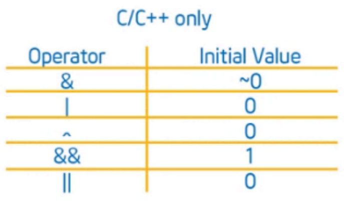
In a typical averaging loop, we then have a loop carry dependency. When we are summing the values of a list with a $+=$ the value of the average variable is dependent on which iteration of the loop we are in. For this OpenMP has built in reductions. Reduction clauses are added to parallel or for constructs and are structured as `#pragma omp parallel [for] reduction(op:list_of_variables)`. Compiler creates a local copy of that variable and initializes it to the identity of whatever operator is identified, for addition this is 0, for multiplication this is 1, etc. Inside the local copy it does whatever, and when its done it gets combined into a global copy automatically.

```
double avg = 0.0, A[MAX]; int i;
#pragma omp parallel for reduction(+:ave)
{
    for (i = 0; i < MAX; i++) {
        avg += A[i];
    }
}
avg = avg/MAX
```

Below is a graphic showing all the possible operators and their respective initial values. So going back to the serial pi program, we only need to add the parallel and reduction pragmas to parallelize in light of these further constructs. One thing of note is that the schedules between two `#pragma omp for`s guarantees that however you map one schedule static to another will be the same, in other words two sequential of these parallel for loops will have each thread scheduled for the same block iterations in between the two regions. To try things out with the aforementioned `schedule(runtime)`, use `omp_set_schedule()` and `omp_get_schedule()`, see https://www.rookiehpc.com/openmp/docs/omp_set_schedule.php

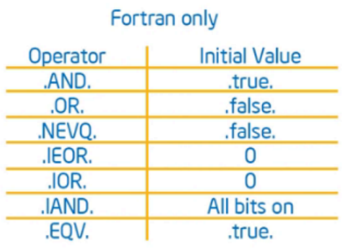


Operator	Initial Value
+	0
*	1
-	0
min	Largest pos num
max	Most neg num



C/C++ only

Operator	Initial Value
&	~0
|	0
^	0
&&	1
||	0



Fortran only

Operator	Initial Value
.AND.	.true.
.OR.	.false.
.NEVQ.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

```
#pragma omp parallel
{
    double x;
    #pragma omp for reduction(+:serial_pi)
    for (i=1; i<num_steps; i++) {
        x = (i+0.5)*step;
        serial_pi += 4.0/(1.0+x*x);
    }
}
```

If the loops to be parallelized are perfectly rectangular, we can collapse multiple loops with the collapse clause, counting from outside region (first loop counts as 1) in the format `collapse(num_loops)`. This will form a single loop of length $N \times M$ and parallelize that. This is useful if the outer loop depends on the number of threads so parallelizing it makes load balancing difficult.

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        ...
    }
}
```

3 OpenMP Workflow

3.1 Implied Barriers

Coming back to barriers, there are some places where OpenMP automatically puts a barrier as it would make sense. For instance, at the end of a parallel block there's a barrier for all the threads until moving on, as logically the split between parallel and serial portions of the program should be kept well-defined. After worksharing constructs like `#pragma omp for` statements there is also a implied barrier, naturally we usually parallelize for loops and use the result of that to extrapolate some other thing. However, if for whatever reason the next statement executed is not dependent on this we can use the `nowait` clause. Be extremely careful with using this as it is fully dependent on what your algorithm needs at various stages, don't worry about the first line for now, it's a data clause

```
#pragma omp parallel shared(A, B, C) private(id)
{
    id = omp_get_thread_num();
    A[id] = big_calc(id);
    #pragma omp barrier
    #pragma omp for
        for (i = 0; i < N; i++) {C[i] = big_calc2(i, A);}
    #pragma omp for nowait
        for (i = 0; i < N; i++) {B[i] = big_calc3(C, i);}
    A[id] = big_calc4(id); // notice in this line we are not using B from prior statement
}
```

There are times where we may want to complete something only on one thread within our parallel block of code. Here, using the `#pragma omp master` construct we can have the master thread go off on a tangential structured block which is ignored by all the other threads. However, there is no synchronization implied here so if we want the other threads to have access to this information before proceeding, we'd need to put a barrier synchronization after the fact.

```
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    {
        do_one_thing();
    }
    #pragma omp barrier
    do_many_other_things();
}
```

Very similarly, we have a worksharing construct that does something similar but for just the first thread which happens upon the structured block, `#pragma omp single`. As with all worksharing constructs, there'd be an implied barrier at the end of this construct here. Likewise, we can use `nowait` here aswell. If we want one thread to do multiple sections of code with different instructions, we can use the sections construction, `#pragma omp sections{#pragma omp section...}` though it is not used as commonly. Not sure, but believe this is in essence a very rudimentary MISD (Multiple Instruction, Single Data) model.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            x_calc();
        #pragma omp section
            y_calc();
        #pragma omp section
            z_calc();
    }
}
```

3.2 Lock Synchronization

So the forms of synchronization we have discussed are barrier synchronizations, critical sections, and basic atomic. However, there are times where we need something even more low level with greater control, where we can use locks - the lowest level of mutual exclusion synchronization in concurrent programming. Conceptually it works like this: If you hold the lock, you're happy and can go do what you want. If you don't hold the lock wait until you can. This is a function API, so there are very granular things we can do with it. Quite straightforwardly, we have `omp_init_lock()` to create a lock, `omp_set_lock()` to take the lock, `omp_unset_lock()` to release the lock, `omp_destroy_lock()` to destroy the lock. Note that we may want to test if the lock is available, if we are trying to get a lock that is already taken up then that thread may be idling and docking our performance. This is accomplished by `omp_test_lock()` which immediately returns whether a lock is taken/available.

So where is this useful? Well, think that we are calculating a histogram, where we have to commonly update some entries on a data structure with multiple threads. If we introduce the critical section for a mutual exclusion synchronization here then we have essentially serialized our program. However, if we think of it the chances of two threads needing to update the same bin in a histogram if we have a sufficiently large number of bins is quite low. Thus, using locks we can check if another thread is already doing so and only update accordingly if we can.

```
#pragma omp parallel for
for(i = 0; i < NBUCKETS; i++) {
    omp_init_lock(&hist_locks[i]); hist[i] = 0;
}

#pragma omp parallel for
for(i = 0; i < NVALS; i++) {
    ival = (int) sample (arr[i]);
    omp_set_lock(&hist_locks[ival]);
    hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
}

for (i = 0; i < NBUCKETS; i++) {
    omp_destroy_lock(&hist_locks[i]);
}
```

So on the `omp_set_lock` line here, if that lock is already set, we will get the lock when it is released. Note at the end we should always still destroy the locks as they do take up memory at scale.

3.3 Dynamic Runtime Routines

There are certain things that we cannot deal with in compile time and can only happen in runtime, which are things we can deal with within runtime libraries. We've already seen some of these runtime routines, such as `omp_set_num_threads()`. An useful new one would be `omp_get_max_threads()` which will give the max threads that we can be given. Remember that the processor might not give us as many threads as we requested, so we can use this max thread count to work with memory allocation. There's also a runtime routine that can detect whether we are in a parallel region: imagine I have a function that I may use both inside and outside a parallel region of which certain parts may need to be executed differently in the two situations. `omp_in_parallel()` does just this, returning a boolean accordingly.

OpenMP is smart, it will try to get you threads based on a multitude of factors, ie what else is running on the processor. So the number of threads we have in one parallel region might not be equal to the number of threads we get in another parallel region. This is called dynamic mode. For this, `omp_set_dynamic()` tells OpenMP to set dynamic mode while `omp_get_dynamic()` tells you whether OpenMP is set up for dynamic mode at its current state. We also may want to know how many cores are on the system we are currently running this on, which is given by `omp_get_num_procs()`.

So, lets put this all together: say we have for whatever reason designed our algorithm so we specifically want some number of threads, and want one thread on each processor. So we first say since we want this consistent number of threads, to turn off dynamic mode passing it false. Then we set number of threads to the number of processors from the aforementioned `omp_get_num_procs()` routine. Then, inside the parallel region we can get each threads ID, and further within a single block can detect the number of threads we have in this parallel region. On the extreme side, we can then check saying if we do not have the appropriate amount of threads, then to exit altogether. Something important to note here is that `omp_get_num_threads()` only works within a parallel region, a common mistake is directly calling this after setting the number of threads; since we are in a serial region, there will only be one thread.

```
#include <omp.h>
void main()
{
    int num_threads;
    omp_set_dynamic(0);
    omp_set_num_threads(omp_num_procs());
    omp_get_num_threads(); // common error! this gives 1
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        #pragma omp single
            num_threads = omp_get_num_threads();
        do_lots_of_stuff(id);
    }
}
```

3.4 Environment Variables

So in production settings when people write parallel code they don't want to adjust it a lot and just want to keep on running it over and over again. Hence, there are some things within parallelization settings that we should typically leave up to environment variables. Simplest of these is of course `OMP_NUM_THREADS`, where we can run for a number of threads expressed as a literal integer. Another environment variable that may be useful is the stacksize, as in some uses we will need to have bigger caches than are provided to us by default which can easily crash your script if not set properly, for this we use `OMP_STACKSIZE`.

Another useful one deals with the wait policy for idling threads in the different synchronization methods that we covered earlier, by using `OMP_WAIT_POLICY` set to either `ACTIVE` or `PASSIVE` we can either have each thread be spinning waiting in active, or be completely shut down and rewoken again for passive. Note that while waiting continuously takes overhead, shut down and being rewoken takes even more: hence if we have an algorithm where we'd expect threads to be idling only for small portions of time as with the histogram example, we'd use active, whereas for larger expected idle periods we'd go with passive.

Recall that although we describe many processors as symmetric multiprocessors (SMP), most of them are typically non-uniform multiprocessor architectures (NUMA). What this means is that as different processes get kicked up on your processor, the system may opt to move your threads around from core to core. Sometimes, if we have an algorithm specifically designed to prestage data in specific portions of caches, we may want to bind threads to processors, disallowing them from being moved around; this can be accomplished with `OMP_PROC_BIND` set to true.

3.5 Data Environment

The set of data and their respective scopes (shared, local) makes up the data environment we are discussing here. OpenMP is a shared memory programming model, in other words most variables are shared by default. Global variables are shared among threads, in C file scope variables are shared. Not everything is shared however, if it is *on the stack* it is private to a thread, for instance, if I have a parallel region and a variable within that region that stays on the stack. Automatic variables created within statement blocks are thus private. So in general, heap is shared, stack is private. A quick C refresher: static variables within functions are kept on memory when the program is running, as opposed to normal variables which are destroyed once the function call is over.

```
double A[10];
int main() {
    int index[10];
    #pragma omp parallel
        work(index);
    printf("%d\n" , index[0]);
}

extern double A[10];
void work(int*index) {
    double temp[10];
    static int count;
    ...
}
```

In this example, when we are initially going into the parallel region we have three variables on the heap: A (filescope), index (declared in main, technically stack strictly speaking since it's inside main), count (static from function). Inside each thread, we have the separate temp variable that each thread can work their own on. There are times where we might want to change the scope of a variable once we are in a parallel region. To change storage attributes, we have the following clauses: *shared*, *private*, *lastprivate*, and *firstprivate*. We can change the default to private, shared, or none using `default(option)`, note that shared is the default. Default private is not available in C, but in Fortran it can be useful. None forces you to explicitly declare the scope of each variable and can be especially helpful for debugging.

```
void wrong() {
    int tmp = 0;
    #pragma omp parallel for private(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

For instance, using private variables the above code would not work as intended since the version of tmp that is piped into the parallel region is not declared with an initial value. In any case, the print statement outside the parallel region will revert back to the previous version. If we want to give it an initial value, we can use firstprivate, which defaults it to the value of the global copy.

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
    if ((i%2) == 0) incr++;
    A[i] = incr;
}
```

So here, going inside the loop i inherits the 0 value from the global version. On the other hand, for lastprivate we are essentially telling the parallel region to draw the variable's value on the last run iteration and draw it back out to the global scope.

```
void sq2(int n, double *lastterm)
{
    double x; int i;
    #pragma omp parallel for lastprivate(x)
    for (i = 0; i < n; i++) {
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
    }
    *lastterm = x
}
```

Here, the value of x at the last scope, when the loop is run for iteration $n-1$ is then copied out to the global scope and set on the lastterm pointer onto x. Keep in mind that we cannot declare something both shared and private since that makes no sense. However, we can declare something both firstprivate and lastprivate since we may want the global value and also update it with the last of the loop. Given the simple example then,

```
int A = 1, B = 1, C = 1;
#pragma omp parallel private(B) firstprivate(C)
```

So in this case, A is shared, B is local to each thread being undefined, and C is local to each thread but receives the initial shared value of 1 as an initialization. At the end of the parallel region, A will be whatever it was modified to be, while B and C will revert back to their global values before the parallel region. We have the Mandelbrot code to calculate the area within the Mandelbrot set:

```
# define NPOINTS 1000
# define MAXITER 1000

struct d_complex{
   double r;
   double i;
};

void testpoint(struct d_complex);

struct d_complex c;
int numoutside = 0;

int main(){
   int i, j;
   double area, error, eps = 1.0e-5;


//   Loop over grid of points in the complex plane which contains the Mandelbrot set,
//   testing each point to see whether it is inside or outside the set.

#pragma omp parallel for default(shared) private(c, j) firstprivate(eps) // eps needs to be initialized
   for (i=0; i<NPOINTS; i++) { // i will be private as default
     for (j=0; j<NPOINTS; j++) { // j needs to be manually made private
       c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
       c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
       testpoint(c);
     }
   }
...

void testpoint(struct d_complex c){

// Does the iteration z=z*z+c, until |z| > 2 when point is known to be outside set
// If loop count reaches MAXITER, point is considered to be inside the set
        ...
        for (iter=0; iter<MAXITER; iter++){
         ...
         if ((z.r*z.r+z.i*z.i)>4.0) {
           #pragma omp atomic
           numoutside++;
           break;
        }
      }
}
```

The changes we needed to make to get this working were mainly concerning the variable scopes within the program, portions that were correct have been replaced with ellipses for brevity. Firstly, for the testpoint function it was initially accepting a void input and using the shared c to calculate whether it was outside. This is problematic as we call this function inside a parallel region and a race condition would arise from the multiple threads repeatedly calling this. Secondly, the parallel region inside main needs to have j declared as private: note that by default *only the first loop* has its iterator automatically made private, all nested loop iterators must be explicitly defined as private. eps also needs to be made firstprivate here to inherit the value defined in the global value (also works just as shared in this case). Lastly, we need to insert a `#pragma omp atomic` inside the iterator on testpoint which counts the number of points outside as again, this function is run in parallel and `numoutside` is a shared variable. Using `default(none)` clause to force individual scope declarations is a nice way to manually debug data environments.

```
#include <omp.h>
static long num_steps = 100000;
double step;
void main()
{
    int i; double x, pi, sum = 0.0;
    #pragma omp parallel for private(x) reduction(+:sum)
    step = 1.0/(double) num_steps;
    for (i=1; i<num_steps; i++) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = step*sum
}
```

For the last time, coming back to the pi calculation program we have the x variable made private as its being changed throughout the local scopes of each thread, and using reduction adds all instances of sum that is calculated in each thread and adds it to the global sum value by reducing all the local copies into a single value and then operating it onto the global value. Hence, we can see to make this serial we only need one pragma statement. As a reminder, a compiler without OpenMP, **skips a pragma if it does not know it**. Hence, this is a fully portable program between parallel and serial with only two additional lines, pretty elegant!

4 OpenMP Advanced

4.1 Tasks

When OpenMP was initially written, it was very much focused on scientific computing in Fortran, so there weren't as many concerns with more complicated structures along the lines of linked lists or recursions.

```
// count list length
while (p != NULL) {
    p = p->next;
    count++;
}
```

```
// save pointer to each element in list
p = head;
for (i = 0; i < count; i++) {
    parr[i] = p;
    p = p -> next;
}

// use list of pointers to go to every element
#pragma omp for schedule(static, 1)
for (i = 0; i < count; i++) {
    do_work(parr[i]);
}
```

This is a very ugly way and unsustainable way to parallelize linked list traversal. Data movement nowadays is much more costly than computation, however, and this requires three traversals of the linked list, far too many to be practical. Tasks are much better ways to do this within OpenMP. Side note, this was a pain in the ass for them to implement it. Tasks in the most basic sense are independent units of work, and consist of *code* to execute, *data* in the environment, and *internal control variables*. Internal Control Variables (ICV) are the running environment controls that dictate the OpenMP settings, whether they are left to default or changed within compile time or runtime, for instance, `omp_set_num_threads()` and related routines all query these ICVs. With tasks, these ICVs are bound **not to the thread but instead the task**. Runtime system will decide when to run these tasks accordingly.

```
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

For instance, here the threads will complete the foo task before running up against the barrier, and then the single thread will create the bar task and complete it. Note that there isn't a `nowait` on the single, so everyone will block until the task is done to continue. Taking a rather inefficient recursive Fibonacci solver, we have

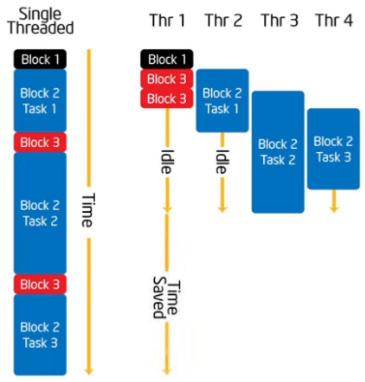
```
int fib(int n)
{
    int x,y;
    if (n < 2) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

So here, the `taskwait` clause will wait till all prior tasks are completed and then execute the structured block attached to it, in this case $x + y$. For linked lists, we then need to use the `firstprivate` clause to capture the pointer or the threads will be running into each other trying to access the shared list. This is an extremely common issue so by default, private variables will be made firstprivate.

```
List ml;
Element *e;
#pragma omp parallel
#pragma omp single
{
    for (e = ml - > first;e;e=e->next)
    #pragma omp task firstprivate(e)
    process(e);
}
```

In light of this, let's revisit our linked list traversal code. We essentially want to isolate one thread to run through the list and create tasks attached to every pointer within this linked list. It will then allocate these tasks to the rest of the threads, which do not need to deal with the linked list traversal at all. The thread which is dealing with this scheduling is basically making a task queue.

```
#pragma omp parallel
{
    #pragma omp single
    {
        node * p = head;
        while (p) {
            #pragma omp task firstprivate(p)
                process(p);
            p = p->next;
        }
    }
}
```



The above figure demonstrates where our time savings comes from, and where the tasks are allocated to be run on the other threads. As an example where this has been used in numerical linear algebra, consider the rank k update given by $C \leftarrow AA^T$ with the other dimension of A given by k. This can be parallelized with three pragmas, and yields great performance even relative to the theoretical peak. For further details see https://www.cs.utexas.edu/users/flame/pubs/openflame_syrk.ps.

4.2 Memory Models

Recall that the formal memory model for OpenMP is shared memory programming, in that it has a shared memory and each processor with a separate cache. If we have a variable in the shared memory, say a, and another instance of a in Cache 3, when processor 1 requests a, how do we know which version to give it? We have to know this to be able to write a well-controled program.

In any code, there will be a lot of reads, writes, and operations, but all we need to track is the state of the variable, which will be only the reads and writes. A compiler will reorder these reads and writes in a way conscious of loads and stores, moving things around so things can be loaded in cache if needed soon and whatnot. Compilers are really good at optimizing the order of reads and writes. The order of these reads and writes is called the program order in source and the code order in the executable.

Now when I run this on a machine in parallel, I will have threads swapping in and out to manage other tasks. So this order is switched up once more in what actually executes and called the commit order. The point of the memory model is to control this, to dictate precise rules and controls to deal with these orders. So what we're concerned with is the order of reads, writes, and synchronization operations (R, W, S) respectively. What a memory model does is tell us where the compiler can and cannot relax these orders, for instance, R »> W must be honored.

Sequential Consistency is when the multi-processor operations remain in the program order when run on each processor and all threads should see the same order. This is naturally the logical way of thinking, but if we did this in practice we would be putting a ton of constraints for memory movement and would be restricting the performance terribly. So, most systems use *relaxed consistency*, where we are able to start removing some of the strict constraints for memory operations. In OpenMP, consistency is defined as a variant of weak consistency. It requires that we cannot reorder synchronization operations with read or write operations on the same thread, in other words {S » W, S » R, R » S, W » S, S » S}. So when we are depending on these reads and writes to be consistent we need to keep in mind to put synchronization constraints accordingly.

In OpenMP, we have a *Flush* which defines a sequence point at which we can guarantee that a thread has a consistent view of memory with respect to the "flush set". This flush set is thus describes by the construct `flush(list)` with list as a list of variables. If we leave off any arguments to the construct, it defaults to all thread visible variables without a list argument. In short, the flush guarantees that: all read and write operations before the flush happen before the flush executes, all read and write operations after the flush happen after the flush executes, and flushes cannot be reordered if they have overlapping flush sets. A flush is basically a fence.

```
double A;
A = compute();
#pragma omp flush(A)
```

Here, we flush to make *A* available to all the threads within the parallel region. Typically, since determining where to use flushes can become convoluted, we can rely on the implicit flushes built into OpenMP. Upon the entry and exit of parallel regions there is an implicit total flush, in other words everything back to DRAM will be visible. The same holds for any implicit and explicit barrier. Again, the flush guarantees that the view *of the thread doing the flush* will be consistent to everyone else, but is not a global synchronization. The flush is also implied by the entry/exit of critical sections and whenever a lock is set or unset. Compilers can still reorganize other variables around a flush set, if it deduces that the variables in question don't overlap with the flush set. It is very finicky to use the flush without a list, typically just flush everything in one go.

Consider an example where we want pairwise synchronization, having a distinct pair of threads that will need to coordinate what they do, in this case we use the population and summation of a random matrix. Note that anytime we have pairwise synchronization there is no serial reading of these codes. We define a flag variable and spin lock the consumer thread until the producer indicates that the flag has been resolved.

```
int main()
{
    double *A, sum, runtime; int numthreads, flag = 0;
    A = (double*) malloc (N*sizeof(double));
    #pragma omp parallel sections
    {
        #pragma omp section // producer
        {
            fill_rand(N, A);
            #pragma omp flush // make sure everyone who wants A can see it
            flag = 1; // flag everyone to let them know
            #pragma omp flush (flag) // reset flag
        }
        #pragma omp section
        {
            #pragma omp flush (flag)
            while (flag == 0) { // spin lock
                #pragma omp flush (flag) // check if flag is updated in memory hierarchy, not register
            }
            #pragma omp flush // flush everything and pick up A
            sum = Sum_array(N, A);
        }
    }
}
```

Note the point of this is to demonstrate how to split workload between threads and update the memory space in between them, but we are still running the population and summation of the array in serial here. So for the above script, there is no parallelism.

This is not best practice though. Note that we do not know upon checking flag what its being updated to, only that it is not 0. Technically, this would be a race condition due to the fact we are not protecting the updates and reading of flag. If we have a trashed or incomplete value for flag, the consumer will still drop out of that spin lock here. We can use atomics with the syntax `#pragma omp atomic [read|write|update|capture]`. Using atomic read dictates that we either complete the read/load or not do it at all, protecting the read. Same deal with write. So how would we fix the above code? Just add the `#pragma omp atomic [read|write]` in front of our read/writes of the flag integer.

4.3 Private Threads

Thread private is another type of scope we can do. What if I want data that is global to a collection of functions and private to a thread, in Fortran this is commonly implemented in common blocks. This is helpful in modifying existing serial code. In C, this happens with file scope and static variables, static class members.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

So here, we have a `counter` initialized as 0. The counter will have a value locally per thread, but global to all the functions that have access to this filescope variable. In other words, each thread has its own copy of the variable but the function can access all threads' versions of this variable. Consider the Monte Carlo problem of finding an estimation of pi by randomly throwing darts at the square and record the ratio of darts that fall within the circle and within the square - this should give roughly $\pi/4$. Using the provided code, we can parallelize by

```
#include <stdio.h>
#include <omp.h>
#include "random.h"

static long num_trials = 10000;

int main ()
{
   long i;  long Ncirc = 0;
   double pi, x, y, r = 1.0;
   seed(0, -r, r);
   #pragma omp parallel for private(x,y,test) reduction(+:Ncirc)
   for(i=0;i<num_trials; i++)
   {
      x = drandom();
      y = drandom();
      if (x*x + y*y <= r*r) Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %ld trials, pi is %lf \n",num_trials, pi);
}
```

This basic program paralellizes the randomization of throwing the dart and counting whether it falls in the circle, being reduced with a summation. Our random number generator we use is the Linear Congruiential Generator (LCG) which operates through `random_next = (MULTIPLIER * random_last + ADDEND)%PMOD` and then `random_last = random_next`. There are complicated ways to pick these parameters, but using a multiplier of 1366, addend of 150889, and pmod of 714025 works for our purposes. See the current implementation of LCG below.

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
double random_low, random_hi;

double drandom()
{
    long random_next;
    double ret_val;
    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

    ret_val = ((double)random_next/(double)PMOD)*(random_hi-random_low)+random_low;
    return ret_val;
}
```

If we actually run this and compare the number of samples and errors we notice that using a higher number of threads our error is increasing more and more. This is because the LCG random number generator we describe is not thread-safe. Since our random number generator is off of the last random number we have threads picking up each other's random last values - a race condition.

In other words, the random library routine was not thread-safe. What I need is to have each thread maintain its own copy of random last, which I could quite straightforwardly do by modifying `drandom`. However, in practice, since we can't change these library routines we have to then rely on private threads.

```
#pragma omp threadprivate(random_last)
```

Through now using the threadprivate clause we can then dictate that each thread had its own last random value and there is no longer the cross pollution as before. However, notice that the generator spits out a sequence of pseudo-random numbers. If we think of this as a line, when we consider the multi-threaded situation; there is a good chance these pseudo-random sequences will have overlap, oversampling a section of the dartboard when we are testing the different points. Intel's Math Kernel Library (mkl) is a good solution for this. However, we can use the leapfrog method to solve this issue; essentially, by interleaving the random numbers that we feed to each thread so thread i starts at the ith number of the sequence, and each successive number skips over a stride length equal to the number of threads. This is known as the *leapfrog method* which we can execute on one thread to dictate the random numbers to the other threads. Within the seeding method we add

```
#pragma omp single
{
    nthreads = omp_get_num_threads();
    iseed = PMOD/MULTIPLIER;
    pseed[0] = iseed;
    mult_n = MULTIPLIER;
    for (i = 1; i < nthreads; ++i) {
        iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
        pseed[i] = iseed;
        mult_n = (mult_n * MULTIPLIER) % PMOD;
    }
}
random_last = (unsigned long long) pseed[id];
```

This is essentially a cyclic way of generating the random numbers. All in all, running this shows that we then get the answers converging to the same results with all possible numbers of threads. Keep this in mind, as many times scientific scripts will make the error of not taking into account how random number generators will skew their probability distribution and not properly seed their parallelized versions.

4.4 Conclusions

The above covers most of the essentials of OpenMP from version 4.0. Nonetheless, there have been 7 years since so many new features have been gradually added on from there. In terms of parallel algorithm patterns, SPMD is highly used in both MPI and OpenMP. The other common pattern is loop parallelism within OpenMP, collecting tasks in iterations of loops which are then divided between a collection of processing elements to compute in parallel. The Divide and Conquer pattern on the other hand divides a problem into subproblems which we can later combine into a global solution. The key steps here are to define a split operation, recursively split it until we can solve directly, then combine to get a solution to the original global problem. We saw this in the Fibonacci implementation mentioned above. These are pretty standard across all parallelization APIs, be it OpenMP or Cilk, etcetera. This intuition behind the paralell design patterns is important to keep in mind as to how they can be expressed within these tools.

Future resources for OpenMP can be accessed through the OpenMP Architecture Review Board, the organization which focuses on the ongoing development of OpenMP at openmp.org. There is also a yearly conference called OpenMP User's Group (cOMPunity) which has interesting discussions on the future of OpenMP. Three good books on OpenMP: Using OpenMP by B. Chapman; Patterns for Parallel Programming by T. Mattson, B. Sanders, B. Massingill (discussed in OpenMP, MPI, Java); Concurrency in Programming Languages by C. RAsmussen, M. Sottile, T. Mattson; and The Art of Concurrency by C. Breshears.

5 GPU Parallelism

5.1 Nvidia Compilers

Nvidia has a few compilers for high-performance computing: NVC++, NVC, NVFortran, NVCC (Cuda Compiler). For those that use OpenMP, use the `-mp` tag to target multicore, use `-mp=gpu` to target GPU and multicore, use `-gpu = ccXX` to set GPU target (default is fine aswell), use `-Minfo = mp` to view compiler diagnostics for OpenMP and set the environment variable `set NVCOMPILER_ACC_NOTIFY 1/2/3` to get varying degrees of runtime logs.

5.2 OpenMP GPU

CPU cores are big but few cores. GPU has many but small cores. Think of CPUs like a car they only transport a few people but it can take you almost everywhere (fast and agile for sequential programming). GPUs are like planes in that they transport many passengers many fast but only from direction to direction, you cannot go to the small roads. But GPUs are designed inherently to be parallel, keep that in mind while looking at OpenMP. For memory space, they all have their own memories and are local to the device. If you want to use your CPU memory, it is your responsibility to move it around.

Existing OpenMP code uses the fork-join model, namely `#pragma omp parallel for`, which cannot be ported to GPU, it does not enable the parallelism possible in GPU. We need to enable more parallelism

- `collapse(N)` clause to increase parallelism
- parallelize inner loops
- reorder loops to enable SIMD/SIMT in outermost loops
- replace critical sections with atomics
- remove I/O statements
- remove memory allocation

Note that is not a silver bullet solution, in that it's not a consistently recompile and run deal. Use the compiler feedback and design around that accordingly.

5.3 Nvidia OpenMP Execution Model

In order to use OpenMP constructs on Nvidia GPUs the following mapping is used. So in a GPU parallelization structure, we need target to start offload, teams to take care of first-level parallelism, and parallel to take advantage of second-level parallelism. In the CPU side, we use parallel and simd because we can only use the different threads as parallel. The question becomes how we maintain portability between these two systems.

`#pragma omp target`	Starts Offload
`#pragma omp teams`	[GPU] CUDA Thread Blocks in grid [CPU] num_teams(1)
`#pragma omp parallel`	[GPU] CUDA Threads within thread block [CPU] CPU threads
`#pragma omp simd`	[GPU] simdlen(1) [CPU] Provides hints for vectorizations

Using the construct `#pragma omp target teams num_teams(X) thread_limit(Y)` we can then initiate offload and enable parallelism. The `target` clause starts the offload while the `teams` clause creates the teams X is the number of CUDA thread blocks, which does not affect the CPU. Y is the number of CUDA threads per thread block, corresponding to the number of CPU threads on that end. To compile use `nvc test.c -mp=gpu =Minfo=mp`. Using this we can see the NVidia compiler generate both Tesla (GPU) and Multicore (CPU) code. In the GPU implementation each team has a master thread, and here the compute region is run by these master threads *sequentially*; in other words, we have offloaded our code but we have not enabled parallelism yet.

5.4 Prescriptive Model

There are two ways to enable parallelism in OpenMP. There are two main models being the prescriptive and the descriptive model. Starting with the prescriptive model, this means that every action taken needs to be specified by the script through means of the directives covered above, like `#pragma omp distribute` and `#pragma omp for/do`. Our goal is to enable as much parallelism as we can to occupy the GPU fully. For instance,

```
#pragma omp target teams distribute parallel for reduction(max:error)
for (int j = 1; j < n-1; j++) {
    for (int i = 1; i < m-1; i++) {
        Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j+1][i]);
        error = fmaxf(error, fabsf(Anew[j][i] - A[j][i]));
    }
}
```

This would successfully parallelize the first loop, using `target` to start the offload, `teams` to create the teams, `distribute` to workshare among the teams, `parallel` to create the threads, and `for` to workshare among the threads. However, think of the case where the thread count of the first loop is small and does not fully occupy the GPU, how can we enable more parallelism. Simplest way here for strictly nested threads is just to add the `collapse(2)` clause so the inner loop is also parallelized, instead of being portioned as an unit. In the case where we cannot simply use `collapse` due to interleaving code between the two loops, we can write out another construct, dividing the `parallel for` directives strictly on the inner loop. In this case the first loop is then parallelized across the teams using a static schedule, and the inner loop is parallelized across threads. However, we are losing parallelism on the outer loop for CPU, instead moving it to the inner loop which introduces costly fork join within the loop and introduces a lot of overhead. Hence, this is not a good solution with portability between CPUs and GPUs, but is a good solution for GPUs only.

```
#pragma omp target teams distribute reduction(max:error)
for (int j = 1; j < n-1; j++) {
    #pragma omp parallel for reduction(max:error)
    for (int i = 1; i < m-1; i++) {
        Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j+1][i]);
        error = fmaxf(error, fabsf(Anew[j][i] - A[j][i]));
    }
}
```

The third way of course would be to use preprocessor macros to direct CPUs for coarse-grained parallelism and GPUs for fine-grained parallelism, the downside being this introduces divergence in code and can be hard to maintain. We would also need to compile different binary in each device, ie `-mp = multicore` and `-mp = gpu`

```
#ifdef TARGET_GPU
    #pragma omp target teams distribute reduction(max:error)
#else
    #pragma omp parallel for reduction(max:error)
#endif
// some loop
{
    #ifdef TARGET_GPU
        #pragma omp parallel for reduction(max:error)
        // some other loop
}
```

5.5 Descriptive Model

The difference with the descriptive model is that we only need to specify which loops we want to parallelize and not every action in our parallelization. Naturally, we use different directives for this. Important to this is the `#pragma omp loop` construct which parallelizes across both teams, threads, and vector instructions. This unlike the `#pragma omp for` construct which only parallelizes with threads, with loop we don't need to explicitly write omp parallel. This is a very simple way to parallelize loops.

`#pragma omp target teams loop`	Recommended way, can use `num_teams` and `thread_limit` clauses
`#pragma omp target loop`	Fully automatic way
`#pragma omp target parallel loop`	Uses only threads, might be useful for light kernels

So, lets revisit the previous algorithm in light of this and use these new methods. Using the `#pragma omp target teams loop` recommendation, and parallelize inner loops with `#pragma omp loop`. This way we can use the outer loop to parallelize teams and inner loop to parallelize threads on the GPU, which is great. Notice it is parallelizing across threads without the `parallel` construct, it is built into the `loop` construct. On the CPU side, it still parallelizes across threads with the outer loop as intended due to the bigger caches.

```
#pragma omp target teams loop reduction(max:error)
for (int j = 1; j < n-1; j++) {
    #pragma omp loop reduction(max:error)
    for (int i = 1; i < m-1; i++) {
        Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j+1][i]);
        error = fmaxf(error, fabsf(Anew[j][i] - A[j][i]));
    }
}
```

This is what we are trying to get towards. It is challenging to do similar portability with the descriptive model. The compiler assigns parallelism here, but we're free to tune to our wishes using the `bind` clause.

`#pragma loop bind(teams)`	[GPU] CUDA Thread Blocks and Threads [CPU] Threads
`#pragma loop bind(parallel)`	[GPU] CUDA Thread [CPU] Threads
`#pragma loop bind(thread)`	[GPU] Single Thread [CPU] Single Thread, Vectorization hints

Let's give an example of tuning a loop, done in Fortran. As mentioned above, if we use `omp target teams loop` only on a multiple loop block

```
!$omp target teams loop
do i = 1, SZ1
    do j = 1, SZ2
        do k = 1, SZ3
            do l = 1, SZ4
                do m = 1, SZ5
                    ! expensive computation codes !
                enddo
            enddo
        enddo
    enddo
enddo
```

Here for GPU, we see that the outer loop is parallelized across teams as expected, but that the innermost loop is also assigned to thread parallelism - this is an automatic thing enabled by compiler detecting better memory access practices. In the multicore part outer loop is still parallelized across threads as expected. To tune this, similar to what was done previously we can add a `collapse(5)` clause, linearizing all the loops and enabling parallelism for them. If we wanted to change the scheduling, we could modify the loop to first collapse 3 loops across teams and then 2 loops across threads.

```
!$omp target teams loop collapse(3)
do i = 1, SZ1
    do j = 1, SZ2
        do k = 1, SZ3
        !$omp loop collapse(2)
            do l = 1, SZ4
                do m = 1, SZ5
                    ! expensive computation codes !
                enddo
            enddo
        enddo
    enddo
enddo
```

In another case where we want to run the inner two loops serially we can use the `bind(threads)` clause to further dictate that. As shown in the compiler message, the inner loops will then be run sequentially. As a reason why this might be helpful - perhaps the thread count here would be very small and it wouldn't make sense to parallelize there. All the while, CPU parallelism is still across threads as intended. The most stringent downside of using the descriptive model is that we **cannot use OpenMP directives** (synchronization, single, barrier, etc.) but this is often not scalable and its recommended we move towards not using them anyhow. Prescriptive model execution model is not fully mappable on GPU and also might introduce some overhead, descriptive model is full SPMD on the other hand.

5.6 Procedures

We can call procedures to utilize functions in other files using the `declare target(function_name)` on both the declaration of the function and the utilization of the function (if it is on a separate file). There is an easier way however, if they are in the same file you can call it directly and the compiler will figure out the rest itself. You can also use the different strategies to parallelize loops, `omp loop` covered above within functions, but note you will have to explicitly state a `bind` clause.