Guide into OpenMP: Easy multithreading programming for C++

By Joel Yliluoma, September 2007; last update in June 2016 for OpenMP 4.5

Abstract

This document attempts to give a quick introduction to OpenMP (as of version 4.5), a simple C/C++/Fortran compiler extension that allows to add parallelism into existing source code without significantly having to rewrite it.

In this document, we concentrate on the C++ language in particular, and use GCC to compile the examples.

Table of contents [expand all] [collapse all]

Preface: Importance of multithreading

As CPU speeds no longer improve as significantly as they did before, multicore systems are becoming more popular.

To harness that power, it is becoming important for programmers to be knowledgeable in parallel programming — making a program execute multiple things simultaneously.

This document attempts to give a quick introduction to OpenMP, a simple C/C++/Fortran compiler extension that allows to add parallelism into existing source code without significantly having to entirely rewrite it.

Support in different compilers

- **GCC** (GNU Compiler Collection) supports OpenMP **4.5** since version 6.1, OpenMP 4.0 since version 4.9, OpenMP 3.1 since version 4.7, OpenMP 3.0 since version 4.4, and OpenMP 2.5 since version 4.2. Add the commandline

option `-fopenmp` to enable it. OpenMP *offloading* is supported for Intel MIC targets only (Intel Xeon Phi KNL + emulation) since version 5.1, and to NVidia (NVPTX) targets since version 7 or so.
- **Clang++** supports OpenMP **4.5** since version 3.9 (without offloading), OpenMP 4.0 since version 3.8 (for some parts), and OpenMP 3.1 since version 3.7. Add the commandline option `-fopenmp` to enable it.
- **Solaris Studio** supports OpenMP **4.0** since version 12.4, and OpenMP 3.1 since version 12.3. Add the commandline option `-xopenmp` to enable it.
- **Intel C Compiler** (icc) supports Openmp **4.5** since version 17.0, OpenMP 4.0 since version 15.0, OpenMP 3.1 since version 12.1, OpenMP 3.0 since version 11.0, and OpenMP 2.5 since version 10.1. Add the commandline option `-openmp` to enable it. Add the `-openmp-stubs` option instead to enable the library without actual parallel execution.
- **Microsoft Visual C++** (cl) supports OpenMP **2.0** since version 2005. Add the commandline option `/openmp` to enable it.

Note: If your GCC complains that "-fopenmp" is valid for D but not for C++ when you try to use it, or does not recognize the option at all, your GCC version is too old. If your linker complains about missing GOMP functions, you forgot to specify "-fopenmp" in the linking.

More information: http://openmp.org/wp/openmp-compilers/

Introduction to OpenMP in C++

OpenMP consists of a set of compiler `#pragma`s that control how the program works. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.

Here are two simple example programs demonstrating OpenMP.

You can compile them like this:

```
g++ tmp.cpp -fopenmp
```

Example: Initializing a table in parallel (multiple threads)

This code divides the table initialization into multiple threads, which are run simultaneously. Each thread initializes a portion of the table.

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

Example: Initializing a table in parallel (single thread, SIMD)

This version requires compiler support for at least OpenMP 4.0, and the use of a parallel floating point library such as AMD ACML or Intel SVML (which can be used in GCC with e.g. -mveclibabi=svml).

```
#include <cmath>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp simd
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

Example: Initializing a table in parallel (multiple threads on another device)

OpenMP 4.0 added support for offloading code to different devices, such as a GPU. Therefore there can be three layers of parallelism in a single program: Single thread processing multiple data; multiple threads running simultaneously; and multiple devices running same program simultaneously.

```
#include <cmath>
int main()
{
```

```
    const int size = 256;
    double sinTable[size];

    #pragma omp target teams distribute parallel for map(from:sinTable[0:256])
    for(int n=0; n<size; ++n)
      sinTable[n] = std::sin(2 * M_PI * n / size);

    // the table is now initialized
  }
```

Example: Calculating the Mandelbrot fractal in parallel (host computer)

This program calculates the classic Mandelbrot fractal at a low resolution and renders it with ASCII characters, calculating multiple pixels in parallel.

```
#include <complex>
#include <cstdio>

typedef std::complex<double> complex;

int MandelbrotCalculate(complex c, int maxiter)
{
    // iterates z = z²² + c until |z| >= 2 or maxiter is reached,
    // returns the number of iterations.
    complex z = c;
    int n=0;
    for(; n<maxiter; ++n)
    {
        if( std::abs(z) >= 2.0) break;
        z = z*z + c;
    }
    return n;
}
int main()
{
    const int width = 78, height = 44, num_pixels = width*height;

    const complex center(-.7, 0), span(2.7, -(4/3.0)*2.7*height/width);
    const complex begin = center-span/2.0;//, end = center+span/2.0;
    const int maxiter = 100000;

  #pragma omp parallel for ordered schedule(dynamic)
    for(int pix=0; pix<num_pixels; ++pix)
    {
        const int x = pix%width, y = pix/width;

        complex c = begin + complex(x * span.real() / (width +1.0),
                                    y * span.imag() / (height+1.0));

        int n = MandelbrotCalculate(c, maxiter);
        if(n == maxiter) n = 0;

      #pragma omp ordered
        {
          char c = ' ';
          if(n > 0)
          {
              static const char charset[] = ".,c8M@jawrpogOQEPGJ";
              c = charset[n % (sizeof(charset)-1)];
          }
          std::putchar(c);
          if(x+1 == width) std::puts("|");
        }
    }
}
```

This program can be improved in many different ways, but it is left simple for the sake of an introductory example.

Discussion

As you can see, there is very little in the program that indicates that it runs in parallel. If you remove the `#pragma` lines, the result is still a valid C++ program that runs and does the expected thing.

Only when the compiler interprets those `#pragma` lines, it becomes a parallel program. It really does calculate N values simultaneously where N is the number of threads. In GCC, libgomp determines that from the number of processors.

By C and C++ standards, if the compiler encounters a `#pragma` that it does not support, it will ignore it. So adding the OMP statements can be done safely[1] without breaking compatibility with legacy compilers.

There is also a runtime library that can be accessed through `omp.h`, but it is less often needed. If you need it, you can check the `#define _OPENMP` for conditional compilation in case of compilers that don't support OpenMP.

[1]: Within the usual parallel programming issues (concurrency, mutual exclusion) of course.

The syntax

All OpenMP constructs in C and C++ are indicated with a `#pragma omp` followed by parameters, ending in a newline. The pragma usually applies only into the statement immediately following it, except for the `barrier` and `flush` commands, which do not have associated statements.

The `parallel` construct

The parallel construct starts a parallel block. It creates a *team* of N threads (where N is determined at runtime, usually from the number of CPU cores, but may be affected by a few things), all of which execute the next statement (or the next block, if the statement is a {...} -enclosure). After the statement, the threads join back into one.

```
#pragma omp parallel
{
  // Code inside this region runs in parallel.
  printf("Hello!\n");
}
```

This code creates a team of threads, and each thread executes the same code. It prints the text "Hello!" followed by a newline, as many times as there are threads in the team created. For a dual-core system, it will output the text twice. (Note: It may also output something like "HeHllellolo", depending on system, because the printing happens in parallel.) At the }, the threads are joined back into one, as if in non-threaded program.

Internally, GCC implements this by creating a magic function and moving the associated code into that function, so that all the variables declared within that block become local variables of that function (and thus, locals to each thread). ICC, on the other hand, uses a mechanism resembling `fork()`, and does not create a magic function. Both implementations are, of course, valid, and semantically identical.

Variables shared from the context are handled transparently, sometimes by passing a reference and sometimes by using register variables which are flushed at the end of the parallel block (or whenever a `flush` is executed).

Parallelism conditionality clause: `if`

The parallelism can be made *conditional* by including a `if` clause in the parallel command, such as:

```
extern int parallelism_enabled;
#pragma omp parallel for if(parallelism_enabled)
for(int c=0; c<n; ++c)
  handle(c);
```

In this case, if `parallelism_enabled` evaluates to a zero value, the number of threads in the team that processes the `for` loop will always be exactly one.

Loop construct: `for`

The `for` construct splits the for-loop so that each thread in the current team handles a different portion of the loop.

```
#pragma omp for
for(int n=0; n<10; ++n)
{
  printf(" %d", n);
}
printf(".\n");
```

This loop will output each number from 0...9 once. However, it may do it in arbitrary order. It may output, for example:

```
0 5 6 7 1 8 2 3 4 9.
```

Internally, the above loop becomes into code equivalent to this:

```
int this_thread = omp_get_thread_num(), num_threads = omp_get_num_threads();
int my_start = (this_thread  ) * 10 / num_threads;
int my_end   = (this_thread+1) * 10 / num_threads;
for(int n=my_start; n<my_end; ++n)
  printf(" %d", n);
```

So each thread gets a different section of the loop, and they execute their own sections in parallel.

Note: `#pragma omp for` only delegates portions of the loop for different threads in the *current team*. A *team* is the group of threads executing the program. At program start, the team consists only of a single member: the master thread that runs the program.

To create a new team of threads, you need to specify the `parallel` keyword. It can be specified in the surrounding context:

```
#pragma omp parallel
{
 #pragma omp for
 for(int n=0; n<10; ++n) printf(" %d", n);
}
printf(".\n");
```

Equivalent shorthand is to specify it in the pragma itself, as `#pragma omp parallel for`:

```
#pragma omp parallel for
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

You can explicitly specify the number of threads to be created in the team, using the `num_threads` attribute:

```
#pragma omp parallel num_threads(3)
{
  // This code will be executed by three threads.

  // Chunks of this loop will be divided amongst
  // the (three) threads of the current team.
  #pragma omp for
  for(int n=0; n<10; ++n) printf(" %d", n);
}
```

Note that OpenMP also works for C. However, in C, you need to set explicitly the loop variable as `private`, because C does not allow declaring it in the loop body:

```
int n;
#pragma omp for private(n)
for(n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

See the "private and shared clauses" section for details.

In OpenMP 2.5, the iteration variable in `for` must be a signed integer variable type. In OpenMP 3.0, it may also be an unsigned integer variable type, a pointer type or a constant-time random access iterator type. In the latter case, `std::distance()` will be used to determine the number of loop iterations.

What are: `parallel`, `for` and a team

The difference between `parallel`, `parallel for` and `for` is as follows:

- A team is the group of threads that execute currently.
  - At the program beginning, the team consists of a single thread.
  - A `parallel` construct splits the current thread into *a new team* of threads for the duration of the next block/statement, after which the team merges back into one.
- `for` divides the work of the for-loop among the threads of the *current team*. It does not create threads, it only divides the work amongst the threads of the currently executing team.
- `parallel for` is a shorthand for two commands at once: `parallel` and `for`. Parallel creates a new team, and `for` splits that team to handle different portions of the loop.

If your program never contains a `parallel` construct, there is never more than one thread; the master thread that starts the program and runs it, as in non-threading programs.

Scheduling

The scheduling algorithm for the for-loop can explicitly controlled.

```
#pragma omp for schedule(static)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

There are five scheduling types: `static`, `dynamic`, `guided`, `auto`, and (since OpenMP 4.0) `runtime`. In addition, there are three scheduling modifiers (since OpenMP 4.5): `monotonic`, `nonmonotonic`, and `simd`.

`static` is the default schedule as shown above. Upon entering the loop, each thread independently decides which chunk of the loop they will process.

There is also the `dynamic` schedule:

```
#pragma omp for schedule(dynamic)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

In the dynamic schedule, there is no predictable order in which the loop items are assigned to different threads. Each thread asks the OpenMP runtime library for an iteration number, then handles it, then asks for next, and so on. This is most useful when used in conjunction with the `ordered` clause, or when the different iterations in the loop may take different time to execute.

The chunk size can also be specified to lessen the number of calls to the runtime library:

```
#pragma omp for schedule(dynamic, 3)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

In this example, each thread asks for an iteration number, executes 3 iterations of the loop, then asks for another, and so on. The last chunk may be smaller than 3, though.

Internally, the loop above becomes into code equivalent to this (illustration only, do not write code like this):

```
int a,b;
if(GOMP_loop_dynamic_start(0,10,1, 3, &a,&b))
{
  do {
    for(int n=a; n<b; ++n) printf(" %d", n);
  } while(GOMP_loop_dynamic_next(&a,&b));
}
```

The `guided` schedule appears to have behavior of `static` with the shortcomings of `static` fixed with `dynamic`-like traits. It is difficult to explain — this example program maybe explains it better than words do. (Requires libSDL to compile.)

The "runtime" option means the runtime library chooses one of the scheduling options at runtime at the compiler library's discretion.

A scheduling modifier can be added to the clause, e.g.: `#pragma omp for schedule(nonmonotonic:dynamic` The modifiers are:

- `monotonic`: Each thread executes chunks in an increasing iteration order.
- `nonmonotonic`: Each thread executes chunks in an unspecified order.
- `simd`: If the loop is a `simd` loop, this controls the chunk size for scheduling in a manner that is optimal for the hardware limitations according to how the compiler decides. This modifier is ignored for non-SIMD loops.

The `ordered` clause

The order in which the loop iterations are executed is unspecified, and depends on runtime conditions.

However, it is possible to force that certain events within the loop happen in a predicted order, using the `ordered` clause.

```
#pragma omp for ordered schedule(dynamic)
for(int n=0; n<100; ++n)
{
  files[n].compress();

  #pragma omp ordered
  send(files[n]);
}
```

This loop "compresses" 100 files with some files being compressed in parallel, but ensures that the files are "sent" in a strictly sequential order.

If the thread assigned to compress file 7 is done but the file 6 has not yet been sent, the thread will wait before sending, and before starting to compress another file. The `ordered` clause in the loop guarantees that there always exists one thread that is handling the lowest-numbered unhandled task.

Each file is compressed and sent exactly once, but the compression may happen in parallel.

There may only be one `ordered` block per an ordered loop, no less and no more. In addition, the enclosing `for` construct must contain the `ordered` clause.

OpenMP 4.5 added some modifiers and clauses to the `ordered` construct.

- `#pragma omp ordered threads` means the same as `#pragma omp ordered`. It means the threads executing the loop execute the `ordered` regions sequentially in the order of loop iterations.

- `#pragma omp ordered simd` can only be used in a `for simd` loop.
- `#pragma omp ordered depend(source)` and `#pragma omp ordered depend(vectorvariable)` also exist.

The `collapse` clause

When you have nested loops, you can use the `collapse` clause to apply the threading to multiple nested iterations.

Example:

```
#pragma omp parallel for collapse(2)
for(int y=0; y<25; ++y)
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
```

The `reduction` clause

The `reduction` clause is a special directive that instructs the compiler to generate code that accumulates values from different loop iterations together in a certain manner. It is discussed in a separate chapter later in this article. Example:

```
int sum=0;
#pragma omp parallel for reduction(+:sum)
for(int n=0; n<1000; ++n) sum += table[n];
```

Sections

Sometimes it is handy to indicate that "this and this can run in parallel". The `sections` setting is just for that.

```
#pragma omp sections
{
  { work1(); }
  #pragma omp section
  { work2();
    work3(); }
  #pragma omp section
  { work4(); }
}
```

This code indicates that any of the tasks `work1`, `work2 + work3` and `work4` may run in parallel, but that `work2` and `work3` must be run in sequence. Each work is done exactly once.

As usual, if the compiler ignores the pragmas, the result is still a correctly running program.

Internally, GCC implements this as a combination of the parallel `for` and a switch-case construct. Other compilers may implement it differently.

Note: `#pragma omp sections` only delegates the sections for different threads in the current team. To create a team, you need to specify the `parallel` keyword either in the surrounding context or in the pragma, as `#pragma omp parallel sections`.
Example:

```
#pragma omp parallel sections // starts a new team
{
  { work1(); }
  #pragma omp section
  { work2();
    work3(); }
  #pragma omp section
  { work4(); }
}
```

or

```
#pragma omp parallel // starts a new team
{
  //Work0(); // this function would be run by all threads.

  #pragma omp sections // divides the team into sections
  {
    // everything herein is run only once.
    { work1(); }
    #pragma omp section
    { work2();
      work3(); }
    #pragma omp section
```

```
    { work4(); }
  }

  //Work5(); // this function would be run by all threads.
}
```

The `simd` construct (OpenMP 4.0+)

OpenMP 4.0 added explicit SIMD parallelism (Single-Instruction, Multiple-Data). SIMD means that multiple calculations will be performed simultaneously by the processor, using special instructions that perform the same calculation to multiple values at once. This is often more efficient than regular instructions that operate on single data values. This is also sometimes called *vector parallelism* or vector operations (and is in fact the preferred term in *OpenACC*).

There are two use cases for the `simd` construct.

Firstly, `#pragma omp simd` can be used to declare that a loop will be utilizing SIMD.

```
float a[8], b[8];
...
#pragma omp simd
for(int n=0; n<8; ++n) a[n] += b[n];
```

Secondly, `#pragma omp declare simd` can be used to indicate a function or procedure that is explicitly designed to take advantage of SIMD parallelism. The compiler may create multiple versions of the same function that use different parameter passing conventions for different CPU capabilities for SIMD processing.

```
#pragma omp declare simd aligned(a,b:16)
void add_arrays(float *__restrict__ a, float *__restrict__ b)
{
  #pragma omp simd aligned(a,b:16)
  for(int n=0; n<8; ++n) a[n] += b[n];
}
```

Without the pragma, the function will use the default non-SIMD-aware ABI, even though the function itself may do calculation using SIMD.

Since compilers of today attempt to do SIMD regardless of OpenMP `simd` directives, the `simd` directive can be thought essentially as a directive to the compiler, saying: “Try harder”.

The `collapse` clause

The `collapse` clause can be added to bind the SIMDness into multiple nested loops. The example code below will direct the compiler into attempting to generate instructions that calculate 16 values simultaneously, if at all possible.

```
#pragma omp simd collapse(2)
for(int i=0; i<4; ++i)
  for(int j=0; j<4; ++j)
    a[j*4+i] += b[i*4+j];
```

The `reduction` clause

The `reduction` clause can be used with SIMD just like with parallel loops.

```
int sum=0;
#pragma omp simd reduction(+:sum)
for(int n=0; n<1000; ++n) sum += table[n];
```

The `aligned` clause

The `aligned` attribute hints the compiler that each element listed is aligned to the given number of bytes. Use this attribute if you are sure that the alignment is guaranteed, and it will increase the performance of the code and make it shorter.

The attribute can be used in both the function declaration, and in the individual SIMD statements.

```
#pragma omp declare simd aligned(a,b:16)
void add_arrays(float *__restrict__ a, float *__restrict__ b)
{
  #pragma omp simd aligned(a,b:16)
  for(int n=0; n<8; ++n) a[n] += b[n];
}
```

The `safelen` clause

While the `restrict` keyword in C tells the compiler that it can assume that two pointers will not address the same data (and thus it is safe to change the ordering of reads and writes), the `safelen` clause in OpenMP provides much fine-grained control over pointer aliasing.

In the example code below, the compiler is informed that `a[x]` and `b[y]` are independent *as long as* the difference between x and y is smaller than 4. In reality, the clause controls the upper limit of concurrent loop iterations. It means that only 4 items can be processed concurrently at most. The actual concurrency may be smaller, and depends on the compiler implementation and hardware limits.

```
#pragma omp declare simd
void add_arrays(float* a, float* b)
{
  #pragma omp simd aligned(a,b:16) safelen(4)
  for(int n=0; n<8; ++n) a[n] += b[n];
}
```

The `simdlen` clause (OpenMP 4.5+)

The `simdlen` clause can be added to a `declare simd` construct to limit how many elements of an array are passed in SIMD registers instead of using the normal parameter passing convention.

The `uniform` clause

The `uniform` clause declares one or more arguments to have an invariant value for all concurrent invocations of the function in the execution of a single SIMD loop.

The `linear` clause (OpenMP 4.5+)

The `linear` clause is similar to the `firstprivate` clause discussed later in this article.

Consider this example code:

```
#include <stdio.h>

int b = 10;
int main()
{
  int array[8];

  #pragma omp simd linear(b:2)
  for(int n=0; n<8; ++n) array[n] = b;

  for(int n=0; n<8; ++n) printf("%d\n", array[n]);
}
```

What does this code print? If we ignore the SIMD constructs, we can see it should print the sequence 10,10,10,10,10,10,10,10.

But, if we enable the OpenMP SIMD construct, the program should now print 10,12,14,16,18,20,22,24. This is because the `linear` clause tells the compiler, that the value of b inside each iteration of the loop should be a *copy* of the original value of b before the SIMD construct, plus the loop iteration number, times the linear scale, which is 2 in this case.

In essence, it should be equivalent to the following code:

```
int b_original = b;
for(int n=0; n<8; ++n) array[n] = b_original + n*2;
```

However, as of GCC version 6.1.0, the `linear` clause does not seem to be implemented correctly, at least according to my understanding of the specification, so I cannot do more experimentation.

The `inbranch` and `notinbranch` clauses

The `inbranch` clause specifies that the function will always be called from inside a conditional statement of a SIMD loop. The `notinbranch` clause specifies that the function will never be called from inside a conditional statement of a SIMD loop.

The compiler may use this knowledge to optimize the code.

The `for simd` construct (OpenMP 4.0+)

The `for` and `simd` constructs can be combined, to divide the execution of a loop into multiple threads, and then execute those loop slices in parallel using SIMD.

```
float sum(float* table)
{
  float result=0;
  #pragma omp parallel for simd reduction(+:result)
  for(int n=0; n<1000; ++n) result += table[n];
  return result;
}
```

The `task` construct (OpenMP 3.0+)

When `for` and `sections` are too cumbersome, the `task` construct can be used. This is only supported in OpenMP 3.0 and later.

These examples are from the OpenMP 3.0 manual:

```
struct node { node *left, *right; };
extern void process(node* );
void traverse(node* p)
{
    if (p->left)
        #pragma omp task // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        traverse(p->right);
    process(p);
}
```

In the next example, we force a postorder traversal of the tree by adding a `taskwait` directive. Now, we can safely assume that the left and right sons have been executed before we process the current node.

```
struct node { node *left, *right; };
extern void process(node* );
void postorder_traverse(node* p)
{
    if (p->left)
        #pragma omp task // p is firstprivate by default
        postorder_traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

The following example demonstrates how to use the task construct to process elements of a linked list in parallel. The pointer p is firstprivate by default on the task construct so it is not necessary to specify it in a firstprivate clause.

```
struct node { int data; node* next; };
extern void process(node* );
void increment_list_items(node* head)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            for(node* p = head; p; p = p->next)
            {
                #pragma omp task
                process(p); // p is firstprivate by default
            }
        }
    }
}
```

Offloading support

Offloading means that parts of the program can be executed not only on the CPU of the computer itself, but also in other hardware attached to it, such as on the graphics card.

The `declare target` and `end declare target` directives

The `declare target` and `end declare target` directives delimit a section of the source code wherein all declarations, whether they are variables or functions/subroutines, are compiled for a device.

Example:

```
#pragma omp declare target
int x;
void murmur() { x+=5; }
#pragma omp end declare target
```

This creates one or more versions of "x" and "murmur". A set that exists on the host computer, and also a separate set that exists and can be run on a device.

These two functions and variables are separate, and may contain values separate from each others.

Variables declared in this manner can be accessed by the device code without separate `map` clauses.

OpenACC differences

In OpenACC, device-functions are declared by prefixing each function with `#pragma acc routine`. Its data model is more complicated and has no direct translation from/to OpenMP.

The `target`, `target data` constructs

The `target data` construct creates a device data environment.

The `target` construct executes the construct on a device (and also has `target data` features).

These two constructs are identical in effect:

```
#pragma omp target // device()... map()... if()...
{
  <<statements...>>
}
```

And:

```
#pragma omp target data // device()... map()... if()...
{
  #pragma omp target
  {
    <<statements...>>
  }
}
```

IMPORTANT: The `target` construct does not add any parallelism to the program by itself. It only transfers the execution into another device, and executes the code there in a single thread.

To utilize parallelism on device, you have to engage a `teams` construct inside the `target` construct. Example:

```
#include <stdio.h>

long long r = 1;

int main(void)
{
  r=10;
  #pragma omp target teams distribute parallel for reduction(+:r) map(tofrom:r)
  for(unsigned long long n=0; n<0x800000000ull; ++n)
      r += n;
  printf("r=%llX\n", r);
  return 0;
}
```

See the `teams` keyword below for details.

The `if` clause

If an `if` clause is added to the `target` region, the attached expression is evaluated. If the expression returns `false`, the code is only executed on the host. Otherwise, or if the `if` clause is not used, the code is executed on the device, and the task will wait until the device is done with the processing.

Example:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv)
{
  int r=0;

  #pragma omp target if(atoi(argv[1])) map(tofrom:r)
```

```
    r += 4;

    printf("r=%d\n", r);
}
```

The `device` clause

Specifices the particular device that is to execute the code.

```
int device_number = ...;
#pragma omp target device(device_number)
{
  //...
}
```

You can acquire device numbers by using the <omp.h> library functions, such as `omp_set_default_device`, `omp_get_default_device`, `omp_get_num_devices`, and `omp_is_initial_device`.

If the `device` clause is not used, the code is executed on the default device. The default device number is controlled by the `omp_set_default_device` function, or the `OMP_DEFAULT_DEVICE` environment variable.

The `map` clause

The `map` clause controls how data is between the host and the device.

There are four different types of mappings:

- `map(alloc:variables)` specifies that at entry to the block, the specified variables have uninitialized values.
- `map(from:variables)` specifies that at entry to the block, the specified variables have copies of their original values on the host.
- `map(to:variables)` specifies that at exit from the block, the values of these variables will be copied back to the host.
- `map(tofrom:variables)` is a combination of `from` and `to`. This is the default mapping.

Variables are initialized and assigned through bitwise copy, i.e. constructors / operators are not called.

The mapping items can be entire variables or array sections.

Array sections (OpenMP 4.0+)

The variables in `map` and `depend` can also specify array sections. The array subsections are defined using one of the following syntax:

- `[lowerbound:length])`
- `[lowerbound:])`
- `[:length])`
- `[:])`

Array sections can only be specified in the `map`, and `depend` clauses. They are invalid in e.g `private`.

An example of a valid array subscript mapping:

```
void foo (int *p)
{
  int i;
  #pragma omp parallel
  #pragma omp single
  #pragma omp target teams distribute parallel for map(p[0:24])
  for (i = 0; i < 24; i++)
    p[i] = p[i] + 1;
}
```

The `target enter data` and `target exit data` constructs (OpenMP 4.5+)

While the `map` clauses within a `target data` construct can be used to allocate data in the device memory and automatically deallocate it in the end of the construct, the `target enter data` and `target exit data` constructs can be used to store data in the memory in a more persistent manner.

Examples:

- `#pragma omp target enter data map(from:var)`
- `#pragma omp target exit data map(to:var)`

The `target update` construct

The `target update` construct can be used to synchronize data between the device memory and the host memory without deallocating it.

- `#pragma omp target update from(c)`

Teams

While the `parallel` construct creates a *team* of *threads*, the `teams` construct creates a *league* of *teams*.

This directive can be only used directly inside a `target` construct. The optional attribute `num_teams` can be used to specify the maximum number of teams created. The actual number of teams may be smaller than this number. The *master* thread of each team will execute the code inside that team.

The example code below *may* print the message multiple times.

```
#include <stdio.h>

int main(void)
{
  #pragma omp target teams
  {
    printf("test\n");
  }
  return 0;
}
```

OpenACC differences

OpenACC calls teams and threads *gangs* and *workers* respectively. In OpenACC, a set of new teams is launched on the device with `#pragma acc parallel`, with the optional attribute `num_gangs(n)`. This combines the behavior of `#pragma omp target` and `#pragma omp teams`.

The `distribute` construct

The `distribute` construct can be used to distribute a `for` loop across the *master* threads of all teams of the current `teams` region.

For example, if there are 20 teams, the loop will be distributed across 20 *master* threads.

```
#include <stdio.h>

int main(void)
{
  int r=0;

  #pragma omp target teams distribute reduction(+:r)
  for(int n=0; n<10000; ++n)
    r += n;

  printf("r=%d\n", r);
  return 0;
}
```

OpenACC differences

In OpenACC this behavior is achieved by adding the word `gang` to existing worksharing constructs like `#pragma acc parallel` and `#pragma acc kernels`.

The `distribute simd` construct

Adding the `simd` clause into the `distribute` construct will combine the effects of `simd` and `distribute`, meaning that the loop will be divided across the *master* threads of all teams of the current `teams` region, and therein divided according to the same principles that are in effect in `#pragma omp simd` constructs.

The `dist_schedule` clause

Much like with the `schedule` clause used with `for` scheduling, the scheduling in `distribute` can be controlled with the `dist_schedule` clause. Currently the only possible value for `dist_schedule` is `static`.

The `distribute parallel for` construct

The `distribute parallel for` construct can be used to distribute a `for` loop across *all* threads of all teams of the current `teams` region.

For example, if there are 20 teams, and each team consists of 256 threads, the loop will be distributed across 5120 threads.

```
#include <stdio.h>

int main(void)
{
  int r=0;

  #pragma omp target teams distribute parallel for reduction(+:r)
  for(int n=0; n<10000; ++n)
    r += n;

  printf("r=%d\n", r);
  return 0;
}
```

The number of threads created in each team is implementation defined, but can be explicitly defined with the `num_threads` attribute.

The `simd` clause can be added once again to the loop to add SIMD execution, if possible.

OpenACC differences

In OpenACC this behavior is achieved by adding the word `worker` to existing worksharing constructs like `#pragma acc parallel` and `#pragma acc kernels`. Additionally the word `vector` can be added to achieve SIMD parallelism as well.

Thread-safety (i.e. mutual exclusion)

There are a wide array of concurrency and mutual exclusion problems related to multithreading programs. I won't explain them here in detail; there are many good books dealing with the issue. (For example, *Multithreaded, Parallel, and Distributed Programming* by Gregory R. Andrews.)

Instead, I will explain the tools that OpenMP provides to handle mutual exclusion correctly.

Atomicity

Atomicity means that something is inseparable; an event either happens completely or it does not happen at all, and another thread cannot intervene during the execution of the event.

```
#pragma omp atomic
counter += value;
```

The `atomic` keyword in OpenMP specifies that the denoted action happens atomically. It is commonly used to update counters and other simple variables that are accessed by multiple threads simultaneously.

See also `reduction`.

There are four different types of atomic expressions (since OpenMP 3.1):

Atomic read expressions

```
#pragma omp atomic read
var = x;
```

Here the reading of `x` is guaranteed to happen atomically, but nothing is guaranteed about `var`. Note that `var` may not access the memory location designated for `x`.

Atomic write expressions

```
#pragma omp atomic write
x = expr;
```

Here the writing of `x` is guaranteed to happen atomically, but nothing is guaranteed about `expr`. Note that `expr` may not access the memory location designated for `x`.

Atomic update expressions

```
#pragma omp atomic update // The word "update" is optional
// One of these:
++x; --x; x++; x--;
x += expr;  x -= expr;  x *= expr;   x /= expr;  x &= expr;
x = x+expr; x = x-expr; x = x*expr;  x = x/expr; x = x&expr;
x = expr+x; x = expr-x; x = expr*x;  x = expr/x; x = expr&x;
x |= expr;  x ^= expr;  x <<= expr;  x >>= expr;
x = x|expr; x = x^expr; x = x<<expr; x = x>>expr;
x = expr|x; x = expr^x; x = expr<<x; x = expr>>x;
```

Here the updating of `x` is guaranteed to happen atomically, but nothing is guaranteed about `expr`. Note that `expr` may not access the memory location designated for `x`.

Atomic capture expressions

Capture expressions combine the `read` and `update` features.

```
#pragma omp atomic capture
// One of these:
var = x++;  /* Or any other of the update expressions listed above */
{ var = x; x++; /* Or any other of of the update expressions listed above */ }
{ x++; /* Or any other of of the update expressions listed above */; var = x; }
{ var = x; x = expr; }
```

Note that neither `var` nor `expr` may not access the memory location designated for `x`.

The `critical` construct

The `critical` construct restricts the execution of the associated statement / block to a single thread at time.

The `critical` construct may optionally contain a global name that identifies the type of the `critical` construct. No two threads can execute a `critical` construct of the same name at the same time.

If the name is omitted, a default name is assumed.

```
#pragma omp critical(dataupdate)
{
  datastructure.reorganize();
}
...
#pragma omp critical(dataupdate)
{
  datastructure.reorganize_again();
}
```

In this example, only one of the critical sections named "dataupdate" may be executed at any given time, and only one thread may be executing it at that time. I.e. the functions "reorganize" and "reorganize_again" cannot be invoked at the same time, and two calls to the function cannot be active at the same time. (Except if other calls exist elsewhere, unprotected by the `critical` construct.)

Note: The critical section names are global to the entire program (regardless of module boundaries). So if you have a critical section by the same name in multiple modules, not two of them can be executed at the same time.

If you need something like a local mutex, see below.

Locks

The OpenMP runtime library provides a lock type, `omp_lock_t` in its include file, `omp.h`.

The lock type has five manipulator functions:

- `omp_init_lock` initializes the lock. After the call, the lock is unset.
- `omp_destroy_lock` destroys the lock. The lock must be unset before this call.
- `omp_set_lock` attempts to set the lock. If the lock is already set by another thread, it will wait until the lock is no longer set, and then sets it.
- `omp_unset_lock` unsets the lock. It should only be called by the same thread that set the lock; the consequences of doing otherwise are undefined.
- `omp_test_lock` attempts to set the lock. If the lock is already set by another thread, it returns 0; if it managed to set the lock, it returns 1.

Here is an example of a wrapper around `std::set<>` that provides per-instance mutual exclusion while still working even if the compiler does not support OpenMP.

You can maintain backward compability with non-OpenMP-supporting compilers by enclosing the library references in `#ifdef _OPENMP...#endif` blocks.

```
#ifdef _OPENMP
# include <omp.h>
#endif
#include <set>

class data
{
private:
  std::set<int> flags;
#ifdef _OPENMP
  omp_lock_t lock;
#endif
public:
  data() : flags()
  {
#ifdef _OPENMP
    omp_init_lock(&lock);
#endif
  }
  ~data()
  {
#ifdef _OPENMP
    omp_destroy_lock(&lock);
#endif
  }

  bool set_get(int c)
  {
  #ifdef _OPENMP
    omp_set_lock(&lock);
  #endif
    bool found = flags.find(c) != flags.end();
    if(!found) flags.insert(c);
  #ifdef _OPENMP
    omp_unset_lock(&lock);
  #endif
    return found;
  }
};
```

Of course, you would really rather wrap the lock into a custom container to avoid littering the code with `#ifdefs` and also for providing exception-safety:

```
#ifdef _OPENMP
# include <omp.h>
struct MutexType
{
  MutexType() { omp_init_lock(&lock); }
  ~MutexType() { omp_destroy_lock(&lock); }
  void Lock() { omp_set_lock(&lock); }
  void Unlock() { omp_unset_lock(&lock); }

  MutexType(const MutexType& ) { omp_init_lock(&lock); }
  MutexType& operator= (const MutexType& ) { return *this; }
public:
  omp_lock_t lock;
};
#else
/* A dummy mutex that doesn't actually exclude anything,
 * but as there is no parallelism either, no worries. */
struct MutexType
{
  void Lock() {}
  void Unlock() {}
};
#endif

/* An exception-safe scoped lock-keeper. */
struct ScopedLock
{
  explicit ScopedLock(MutexType& m) : mut(m), locked(true) { mut.Lock(); }
  ~ScopedLock() { Unlock(); }
  void Unlock() { if(!locked) return; locked=false; mut.Unlock(); }
  void LockAgain() { if(locked) return; mut.Lock(); locked=true; }
private:
  MutexType& mut;
  bool locked;
private: // prevent copying the scoped lock.
  void operator=(const ScopedLock&);
  ScopedLock(const ScopedLock&);
};
```

This way, the example above becomes a lot simpler, and also exception-safe:

```
#include <set>

class data
{
private:
  std::set<int> flags;
  MutexType lock;
public:
  bool set_get(int c)
  {
    ScopedLock lck(lock); // locks the mutex

    if(flags.find(c) != flags.end()) return true; // was found
    flags.insert(c);
    return false; // was not found
  } // automatically releases the lock when lck goes out of scope.
};
```

There is also a lock type that supports nesting, `omp_nest_lock_t`. I will not cover it here.

The `flush` directive

Even when variables used by threads are supposed to be shared, the compiler may take liberties and optimize them as register variables. This can skew concurrent observations of the variable. The `flush` directive can be used to ensure that the value observed in one thread is also the value observed by other threads.

This example comes from the OpenMP specification.

```
        /* presumption: int a = 0, b = 0; */

  /* First thread */              /* Second thread */
  b = 1;                          a = 1;
  #pragma omp flush(a,b)          #pragma omp flush(a,b)
  if(a == 0)                      if(b == 0)
  {                               {
    /* Critical section */          /* Critical section */
  }                               }
```

In this example, it is enforced that at the time either of a or b is accessed, the other is also up-to-date, practically ensuring that not both of the two threads enter the critical section. (Note: It is still possible that neither of them can enter it.)

You need the `flush` directive when you have writes to and reads from the same data in different threads.

If the program appears to work correctly without the `flush` directive, it does not mean that the `flush` directive is not required. It just may be that your compiler is not utilizing all the freedoms the standard allows it to do. You *need* the `flush` directive whenever you access shared data in multiple threads: After a write, before a read.

However, I do not know these:

- Is `flush` needed if the shared variable is declared `volatile`?
- Is `flush` needed if all access to the shared variable is `atomic` or restricted by `critical` sections?

Controlling which data to share between threads

In the parallel section, it is possible to specify which variables are shared between the different threads and which are not. By default, all variables are shared except those declared within the parallel block.

The `private`, `firstprivate` and `shared` clauses

```
int a, b=0;
#pragma omp parallel for private(a) shared(b)
for(a=0; a<50; ++a)
{
  #pragma omp atomic
  b += a;
}
```

This example explicitly specifies that a is private (each thread has their own copy of it) and that b is shared (each thread accesses the same variable).

The difference between `private` and `firstprivate`

Note that a private copy is an uninitialized variable by the same name and same type as the original variable; it does *not* copy the value of the variable that was in the surrounding context.

Example:

```
#include <string>
#include <iostream>

int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    #pragma omp parallel private(a,c) shared(b) num_threads(2)
    {
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << "), b is (" << b << ")\n";
    }
}
```

This will output the string "k", not "xk". At the entrance of the block, a becomes a new instance of `std::string`, that is initialized with the default constructor; it is not initialized with the copy constructor.

Internally, the program becomes like this:

```
int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    OpenMP_thread_fork(2);
    {                   // Start new scope
        std::string a; // Note: It is a new local variable.
        int c;         // This too.
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << "), b is (" << b << ")\n";
    }                   // End of scope for the local variables
    OpenMP_join();
}
```

In the case of primitive (POD) datatypes (`int`, `float`, `char*` etc.), the private variable is uninitialized, just like any declared but not initialized local variable. It does not contain the value of the variable from the surrounding context. Therefore, the increment of `c` is moot here; the value of the variable is still undefined. (If you are using GCC version earlier than 4.4, you do not even get a warning about the use of uninitialized value in situations like this.)

If you actually need a *copy* of the original value, use the `firstprivate` clause instead.

```
#include <string>
#include <iostream>

int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    #pragma omp parallel firstprivate(a,c) shared(b) num_threads(2)
    {
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << "), b is (" << b << ")\n";
    }
}
```

Now the output becomes "A becomes (xk), b is (y)".

The `lastprivate` clause

The `lastprivate` clause defines a variable private as in `firstprivate` or `private`, but causes the value from the last task to be copied back to the original value after the end of the loop/sections construct.

- In a loop construct (`for` construct), the last value is the value assigned by the thread that handles the last iteration of the loop. Values assigned during other iterations are ignored.
- In a sections construct (`sections` construct), the last value is the value assigned in the last section denoted by the `section` construct. Values assigned in other sections are ignored.

Example:

```
#include <stdio.h>
int main()
{
   int done = 4, done2 = 5;

    #pragma omp parallel for lastprivate(done, done2) num_threads(2) schedule(static)
    for(int a=0; a<8; ++a)
    {
      if(a==2) done=done2=0;
      if(a==3) done=done2=1;
    }
    printf("%d,%d\n", done,done2);
}
```

This program outputs "4196224,-348582208", because internally, this program became like this:

```
#include <stdio.h>
int main()
{
   int done = 4, done2 = 5;
   OpenMP_thread_fork(2);
   {
       int this_thread = omp_get_thread_num(), num_threads = 2;
       int my_start = (this_thread  ) * 8 / num_threads;
       int my_end   = (this_thread+1) * 8 / num_threads;

       int priv_done, priv_done2; // not initialized, because firstprivate was not used

       for(int a=my_start; a<my_end; ++a)
       {
           if(a==2) priv_done=priv_done2=0;
           if(a==3) priv_done=priv_done2=1;
       }
       if(my_end == 8)
       {
          // assign the values back, because this was the last iteration
          done  = priv_done;
          done2 = priv_done2;
       }
   }
   OpenMP_join();
}
```

As one can observe, the values of priv_done and priv_done2 are not assigned even once during the course of the loop that iterates through 4...7. As such, the values that are assigned back are completely bogus.

Therefore, `lastprivate` cannot be used to e.g. fetch the value of a flag assigned randomly during a loop. Use `reduction` for that, instead.

Where this behavior *can* be utilized though, is in situations like this (from OpenMP manual):

```
void loop()
{
  int i;
  #pragma omp for lastprivate(i)
  for(i=0; i<get_loop_count(); ++i) // note: get_loop_count() must be a pure function.
      { ... }

  printf("%d\n", i); // this shows the number of loop iterations done.
}
```

The `default` clause

The most useful purpose on the `default` clause is to check whether you have remembered to consider all variables for the private/shared question, using the `default(none)` setting.

```
int a, b=0;
// This code won't compile: It requires explicitly
// specifying whether a is shared or private.
#pragma omp parallel default(none) shared(b)
{
  b += a;
}
```

The `default` clause can also be used to set that all variables are `shared` by default (`default(shared)`).

> *Note: Because different compilers have different ideas about which variables are implicitly private or shared, and for which it is an error to explicitly state the private/shared status, it is recommended to use the* `default(none)` *setting only during development, and drop it in production/distribution code.*

The `reduction` clause

The `reduction` clause is a mix between the `private`, `shared`, and `atomic` clauses.
It allows to accumulate a shared variable without the `atomic` clause, but the type of accumulation must be specified. It will often produce faster executing code than by using the `atomic` clause.

This example calculates factorial using threads:

```
int factorial(int number)
{
  int fac = 1;
  #pragma omp parallel for reduction(*:fac)
  for(int n=2; n<=number; ++n)
    fac *= n;
  return fac;
}
```

- At the beginning of the parallel block, a private copy is made of the variable and preinitialized to a certain value .
- At the end of the parallel block, the private copy is atomically merged into the shared variable using the defined operator.

(The private copy is actually just a new local variable by the same name and type; the original variable is not accessed to create the copy.)

The syntax of the clause is:

```
reduction(operator:list)
```

where *list* is the list of variables where the operator will be applied to, and *operator* is one of these:

Operator	Initialization value
+, -, |, ^, ||	0
*, &&	1
&	~0
`min`	largest representable number
`max`	smallest representable number

To write the factorial function (shown above) without `reduction`, it probably would look like this:

```
int factorial(int number)
{
  int fac = 1;
  #pragma omp parallel for
  for(int n=2; n<=number; ++n)
  {
    #pragma omp atomic
    fac *= n;
  }
  return fac;
}
```

However, this code would be less optimal than the one with `reduction`: it misses the opportunity to use a local (possible register) variable for the cumulation, and needlessly places load/synchronization demands on the shared memory variable. In fact, due to the bottleneck of that atomic variable (only one thread may access it simultaneously), it would completely nullify any gains of parallelism in that loop.

The version with `reduction` is equivalent to this code (illustration only):

```
int factorial(int number)
{
  int fac = 1;
  #pragma omp parallel
  {
    int omp_priv = 1; /* This value comes from the table shown above */
    #pragma omp for nowait
    for(int n=2; n<=number; ++n)
      omp_priv *= n;
    #pragma omp atomic
    fac *= omp_priv;
  }
  return fac;
}
```

Note how it moves the atomic operation out from the loop.

The restrictions in `reduction` and `atomic` are very similar: both can only be done on POD types; neither allows overloaded operators, and both have the same set of supported operators.

As an example of how the `reduction` clause can be used to produce semantically different code when OpenMP is enabled and when it is disabled, this example prints the number of threads that executed the parallel block:

```
int a = 0;
#pragma omp parallel reduction (+:a)
{
  a = 1; // Assigns a value to the private copy.
  // Each thread increments the value by 1.
}
printf("%d\n", a);
```

If you preinitialized "a" to 4, it would print a number >= 5 if OpenMP was enabled, and 1 if OpenMP was disabled.
Note: If you really need to detect whether OpenMP is enabled, use the `_OPENMP` *#define instead. To get the number of threads, use* `omp_get_num_threads()` *instead.*

The `declare reduction` directive (OpenMP 4.0+)

The `declare reduction` directive generalizes the reductions to include user-defined reductions.

The syntax of the declaration is one of these two:

```
#pragma omp declare reduction(name:type:expression)
#pragma omp declare reduction(name:type:expression) initializer(expression)
```

- The *name* is the name you want to give to the reduction method.
- The *type* is the type of your reduction result.
- Within the `reduction` expression, the special variables `omp_in` and `omp_out` are implicitly declared, and they stand for the input and output expressions respectively.
- Within the `initializer` expression, the special variable `omp_priv` is implicitly declared and stands for the initial value of the reduction result.

An example use case is when you are running a data compressor with different parameters, and you want to find the set of parameters that results in best compression. Below is an example of such code:

```
#include <cstdio>

int compress(int param1, int param2)
{
  return (param1+13)^param2; // Placeholder for a compression algorithm
}

int main(int argc, char** argv)
{
  struct BestInfo { unsigned size, param1, param2; };

  #pragma omp declare reduction(isbetter:BestInfo: \
                                omp_in.size<omp_out.size ? omp_out=omp_in : omp_out \
                              ) initializer(omp_priv = BestInfo{~0u,~0u,~0u})

  BestInfo result{~0u,~0u,~0u};
  #pragma omp parallel for collapse(2) reduction(isbetter:result)
  for(unsigned p1=0; p1<10; ++p1)
  for(unsigned p2=0; p2<10; ++p2)
  {
    unsigned size = compress(p1,p2);
    if(size < result.size) result = BestInfo{size,p1,p2};
  }
  std::printf("Best compression (%u bytes) with params %u,%u\n",
    result.size, result.param1, result.param2);
}
```

Thread affinity (`proc_bind`)

The thread affinity of the `parallel` construct can be controlled with a `proc_bind` clause. It takes one of the following three forms:

- `#pragma omp parallel proc_bind(master)`
- `#pragma omp parallel proc_bind(close)`
- `#pragma omp parallel proc_bind(spread)`

For more information, read the OpenMP specification.

Execution synchronization

The `barrier` directive and the `nowait` clause

The `barrier` directive causes threads encountering the barrier to wait until all the other threads in the same team have encountered the barrier.

```
#pragma omp parallel
{
  /* All threads execute this. */
  SomeCode();

  #pragma omp barrier

  /* All threads execute this, but not before
   * all threads have finished executing SomeCode().
   */
  SomeMoreCode();
}
```

Note: There is an implicit barrier at the end of each parallel block, and at the end of each `sections`, `for` and `single` statement, unless the `nowait` directive is used.

Example:

```
#pragma omp parallel
{
  #pragma omp for
  for(int n=0; n<10; ++n) Work();

  // This line is not reached before the for-loop is completely finished
  SomeMoreCode();
}

// This line is reached only after all threads from
// the previous parallel block are finished.
CodeContinues();

#pragma omp parallel
{
  #pragma omp for nowait
  for(int n=0; n<10; ++n) Work();

  // This line may be reached while some threads are still executing the for-loop.
  SomeMoreCode();
}

// This line is reached only after all threads from
// the previous parallel block are finished.
CodeContinues();
```

The `nowait` directive can only be attached to `sections`, `for` and `single`. It cannot be attached to the within-loop `ordered` clause, for example.

The `single` and `master` constructs

The `single` construct specifies that the given statement/block is executed by only one thread. It is unspecified which thread. Other threads skip the statement/block and wait at an implicit barrier at the end of the construct.

```
#pragma omp parallel
{
  Work1();
  #pragma omp single
  {
    Work2();
  }
  Work3();
}
```

In a 2-cpu system, this will run Work1() twice, Work2() once and Work3() twice. There is an implied barrier at the end of the `single` construct, but not at the beginning of it.

Note: Do not assume that the `single` block is executed by whichever thread gets there first. According to the standard, the decision of which thread executes the block is implementation-defined, and therefore making assumptions on it is non-conforming.

The `master` construct is similar, except that the statement/block is run by the *master* thread, and there is no implied barrier; other threads skip the construct without waiting.

```
#pragma omp parallel
{
  work1();

  // This...
  #pragma omp master
  {
    work2();
  }

  // ...is practically identical to this:
  if(omp_get_thread_num() == 0)
  {
    work2();
  }

  work3();
}
```

Unless you use the `threadprivate` clause, the only important difference between `single nowait` and `master` is that if you have multiple `master` blocks in a `parallel` section, you are guaranteed that they are executed by the same thread every time, and hence, the values of `private` (thread-local) variables are the same.

Thread cancellation (OpenMP 4.0+)

Suppose that we want to optimize this function with parallel processing:

```
/* Returns any position from the haystack where the needle can
 * be found, or NULL if no such position exists. It is not guaranteed
 * to find the first matching position; it only guarantees to find
 * _a_ matching position if one exists.
 */
const char* FindAnyNeedle(const char* haystack, size_t size, char needle)
{
  for(size_t p = 0; p < size; ++p)
    if(haystack[p] == needle)
    {
      /* This breaks out of the loop. */
      return haystack+p;
    }
  return NULL;
}
```

Our first attempt might be to simply tack a `#pragma parallel for` before the `for` loop, but that doesn't work: OpenMP requires that a loop construct processes each iteration. Breaking out of the loop (using `return`, `goto`, `break`, `throw` or other means) is not allowed.

To solve this problem, OpenMP 4.0 added a mechanism called `cancellation points`, and a `cancel` construct. Cancellation points are implicitly inserted at the following positions:

- Implicit barriers
- `barrier` regions
- `cancel` regions
- `cancellation point` regions

It can be used to solve finder problems where N threads search for a solution and once a solution is found by any thread, all threads end their search.

Because there is a performance overhead in checking for cancellations, it is only enabled if the library-internal global variable OMP_CANCELLATION is set. The value of this variable can be checked with the `omp_get_cancellation()` function, but there is no way modify it from inside the program. It can only be set from the environment when the program is launched.

In this example program, once a thread finds the "needle", it signals cancellation for all threads of the current team processing the innermost `for` loop. Threads check the cancellation only at every loop iteration. It also checks whether OMP_CANCELLATION is set, and if not, sets it and reruns the program.

```
#include <stdio.h>  // For printf
#include <string.h> // For strlen
#include <stdlib.h> // For putenv
#include <unistd.h> // For execv
#include <omp.h>    // For omp_get_cancellation, omp_get_thread_num()

static const char* FindAnyNeedle(const char* haystack, size_t size, char needle)
{
  const char* result = haystack+size;
  #pragma omp parallel
  {
```

```
    unsigned num_iterations=0;
    #pragma omp for
    for(size_t p = 0; p < size; ++p)
    {
      ++num_iterations;
      if(haystack[p] == needle)
      {
        #pragma omp atomic write
        result = haystack+p;
        // Signal cancellation.
        #pragma omp cancel for
      }
      // Check for cancellations signalled by other threads:
      #pragma omp cancellation point for
    }
    // All threads reach here eventually; sooner if the cancellation was signalled.
    printf("Thread %u: %u iterations completed\n", omp_get_thread_num(), num_iterations);
  }
  return result;
}

int main(int argc, char** argv)
{
  if(!omp_get_cancellation())
  {
    printf("Cancellations were not enabled, enabling cancellation and rerunning program\n");
    putenv("OMP_CANCELLATION=true");
    execv(argv[0], argv);
  }
  printf("%s\n%*s\n", argv[1], FindAnyNeedle(argv[1],strlen(argv[1]),argv[2][0])-argv[1]+1, "^");
}
```

Example output:

```
./a.out "OpenMP cancellations can only be performed synchronously at cancellation points." "l"
Cancellations were not enabled, enabling cancellation and rerunning program
Thread 0: 10 iterations completed
Thread 1: 3 iterations completed
Thread 7: 10 iterations completed
Thread 3: 10 iterations completed
Thread 4: 10 iterations completed
Thread 2: 8 iterations completed
Thread 5: 5 iterations completed
Thread 6: 6 iterations completed
OpenMP cancellations can only be performed synchronously at cancellation points.
                          ^
```

The keyword in the end of the `#pragma omp cancellation point` construct is the name of the most closely nested OpenMP construct that you want to cancel. In the example code above, it is the `for` construct, and this is why the line says `#pragma omp cancellation point for`.

OpenMP cancellations can only be performed synchronously at cancellation points. GNU pthreads also permits asynchronous cancellations. This is rarely used, and requires special setup, because there are several resource leak risks involved in it. An example of such code can be found here: http://bisqwit.iki.fi/jutut/kuvat/openmphowto/pthread_cancel_demo.cpp

Loop nesting

The problem

A beginner at OpenMP will quickly find out that this code will not do the expected thing:

```
#pragma omp parallel for
for(int y=0; y<25; ++y)
{
  #pragma omp parallel for
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
}
```

The beginner expects there to be N tick() calls active at the same time (where N = number of processors). Although that is true, the inner loop is not actually parallelised. Only the outer loop is. The inner loop runs in a pure sequence, as if the whole inner `#pragma` was omitted.

At the entrance of the inner `parallel` construct, the OpenMP runtime library (libgomp in case of GCC) detects that there already exists a team, and instead of a new team of N threads, it will create a team consisting of only the calling thread.

Rewriting the code like this won't work:

```
#pragma omp parallel for
for(int y=0; y<25; ++y)
{
  #pragma omp for // ERROR, nesting like this is not allowed.
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
}
```

This code is erroneous and will cause the program to malfunction. See the restrictions chapter below for details.

Solution in OpenMP 3.0

In OpenMP 3.0, the loop nesting problem can be solved by using the `collapse` clause in the `for` construct.

Example:

```
#pragma omp parallel for collapse(2)
for(int y=0; y<25; ++y)
  for(int x=0; x<80; ++x)
  {
    tick(x,y);
  }
```

The number specified in the `collapse` clauses is the number of nested loops that are subject to the work-sharing semantics of the OpenMP `for` construct.

Restrictions

There are restrictions to which clauses can be nested under which constructs. The restrictions are listed in the OpenMP official specification.

Performance

Compared to a naive use of C++11 threads, OpenMP threads are often more efficient. This is because many implementations of OpenMP use a *thread pool*. A thread pool means that new operating system threads are only created once. When the threads are done with their work, they return to a “dock” waiting for new work to do.

Shortcomings

OpenMP and fork()

It is worth mentioning that using OpenMP in a program that calls `fork()` requires special consideration.

This problem only affects GCC; ICC is not affected.

If your program intends to become a background process using `daemonize()` or other similar means, you must not use the OpenMP features *before* the fork. After OpenMP features are utilized, a fork is only allowed if the child process does not use OpenMP features, or it does so as a completely new process (such as after `exec()`).

This is an example of an erroneous program:

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

void a()
{
  #pragma omp parallel num_threads(2)
  {
    puts("para_a"); // output twice
  }
  puts("a ended"); // output once
}
void b()
{
  #pragma omp parallel num_threads(2)
  {
    puts("para_b");
  }
  puts("b ended");
}

int main() {
 a();   // Invokes OpenMP features (parent process)
```

```
  int p = fork();
  if(!p)
  {
    b(); // ERROR: Uses OpenMP again, but in child process
    _exit(0);
  }
  wait(NULL);
  return 0;
}
```

When run, this program hangs, never reaching the line that outputs "b ended".

There is currently no workaround; the libgomp API does not specify functions that can be used to prepare for a call to `fork()`.

Missing in this article

- The `depend` clause (added in OpenMP 4.0)
- The `nowait` clause in `target` construct (added in OpenMP 4.5)
- The `taskgroup` construct (added in OpenMP 4.0)
- The `taskyield` construct (added in OpenMP 3.1)
- The `final`, `mergeable`, and `priority` clauses in `task` (added in OpenMP 3.1 through 4.5)
- The `threadprivate`, `copyprivate` and `copyin` clauses
- The `ref`, `val`, and `uval` modifiers in `linear` clause (added in OpenMP 4.5)
- The `hint` clause in `critical` construct (added in OpenMP 4.5)
- The `defaultmap` clause (added in OpenMP 4.5)

Some specific gotchas

C++

- STL is not thread-safe. If you use STL containers in a parallel context, you *must* exclude concurrent access using locks or other mechanisms. Const-access is usually fine, as long as non-const access does not occur at the same time.
- Exceptions may not be thrown and caught across omp constructs. That is, if a code inside an `omp for` throws an exception, the exception must be caught before the end of the loop iteration; and an exception thrown inside a `parallel` section must be caught by the same thread before the end of the `parallel` section.

GCC

- `fork()` is troublematic when used together with OpenMP. See the chapter "OpenMP and fork()" above for details.

Further reading

The official specification of OpenMP is the document that dictates what a conforming compiler should do. If you have any question regarding OpenMP, the official specification answers the questions. If it is not there, it is undefined.

- The OpenMP 4.5 official specification
- Wikipedia article for OpenMP
- OpenMP crash course at ARSC — has especially good list of gotchas.
- The OpenMP 4.0 official specification
- The OpenMP 3.0 official specification
- The OpenMP 2.5 official specification
- Wikipedia article for Xeon Phi — for more information about Intel MIC

Last edited at: 2018-02-10T19:10:44+00:00