Guide into OpenMP: Easy multithreading programming for C++

By Joel Yliluoma, September 2007; last update in December 2008

Abstract

This document attempts to give a quick introduction to OpenMP (as of version 3.0), 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 GCC compiler and the C++ language in particular. However, other compilers and languages supporting OpenMP also exist.

This document covers OpenMP 3.0, but because version 2.5 is more commonly supported than 3.0, we also place emphasis on version 2.5.

Table of contents [expand all] [collapse all]

- Abstract
- Preface: Importance of multithreading
  - Support in different compilers
- Introduction to OpenMP in C++
  - Example: Initializing a table in parallel
  - Example: Calculating the Mandelbrot fractal in parallel
  - Discussion
- The syntax
  - The `parallel` pragma
  - Loop directive: `for`
  - Sections
  - The `task` directive (OpenMP 3.0 only)
- Thread-safety (i.e. mutual exclusion)
  - Atomicity
  - The `critical` directive
  - Locks
  - The `flush` directive
- Controlling which data to share between threads
  - The `private`, `firstprivate` and `shared` clauses
  - The `lastprivate` clause
  - The `default` clause
  - The `reduction` clause
- Execution synchronization
  - The `barrier` directive and the `nowait` clause
  - The `single` and `master` directives
- Loop nesting
  - The problem
  - Restrictions

- Shortcomings
  - OpenMP and fork()
  - Cancelling of threads (and breaking out of loops)
- Some specific gotchas
- Other
- Further reading

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.

In this document, we concentrate on the GCC compiler and the C++ language in particular. However, other compilers and languages supporting OpenMP also exist.

Support in different compilers

- OpenMP 2.5 is supported by the GNU Compiler Collection (GCC) since version 4.2. Add the commandline option `-fopenmp` to enable it.
- OpenMP 3.0 is supported by the GNU Compiler Collection (GCC) since version 4.4. Add the commandline option `-fopenmp` to enable it.
- OpenMP 4.0 is supported by the GNU Compiler Collection (GCC) since version 4.9. Add the commandline option `-fopenmp` to enable it.
- OpenMP 2.5 is supported by Microsoft Visual C++ 2005 (cl). Add the commandline option `/openmp` to enable it.
- OpenMP 2.5 is supported by Intel C Compiler (icc) 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.
- OpenMP 3.0 is supported by Intel C Compiler (icc) since version 11.0. Add the commandline option `-openmp` to enable it. Add the `-openmp-stubs` option instead to enable the library without actual parallel execution.
- Clang++ still does not have OpenMP support as of version 3.5.

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

```
#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: Calculating the Mandelbrot fractal in parallel

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 = z + 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("|");
        }
    }
}
```

(There are a multitude of ways this program can be improved but that is beside the point.)

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 directives 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` pragma

The parallel pragma 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 "HeHlellolo", 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).

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 directive: `for`

The `for` directive 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` directive 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` directive, 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");
```

`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.)

OpenMP 4.0 also added a "runtime" option to scheduling, which just chooses one of the other scheduling options at runtime at the compiler library's discretion.

Ordering

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` directive must contain the `ordered` clause.

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 `task` directive (OpenMP 3.0 only)

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

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
            }
        }
    }
}
```

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.

Unfortunately, the atomicity setting can only be specified to simple expressions that usually can be compiled into a single processor opcode, such as increments, decrements, xors and the such. For example, it cannot include function calls, array indexing, overloaded operators, non-POD types, or multiple statements. Furthermore, it only specifies atomicity for the action relating to the lefthand side of the operation; it does not guarantee that the expression on the right side of the operator is evaluated atomically. If you need to atomicise more complex constructs, use either the `critical` directive or the locks, as explained below.

See also `reduction`.

The `critical` directive

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

The `critical` directive may optionally contain a global name that identifies the type of the `critical` directive. No two threads can execute a `critical` directive 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` directive.)

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 `#ifdef`s 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` directive), 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` directive), the last value is the value assigned in the last section denoted by the `section` directive. 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 will require that
// it will be explicitly specified 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

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 the 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 fac_private = 1; /* This value comes from the table shown
above */
    #pragma omp for nowait
    for(int n=2; n<=number; ++n)
      fac_private *= n;
    #pragma omp atomic
    fac *= fac_private;
  }
  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.

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` directives

The `single` directive 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` directive, 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. In GCC, the decision is hidden inside the mechanics of libgomp.

The `master` directive 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.

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` directive, 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` directive.

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` directive.

Workarounds in OpenMP 2.5

If you are stuck with a compiler that does not support OpenMP 3.0 (such as GCC version older than 4.4), you may need to use a workaround. Workarounds exist for OpenMP 2.5.

As was shown in the Mandelbrot example near the beginning of this document, the problem can be worked around by turning the nested loop into a non-nested loop:

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

However, rewriting code like this can be a nuisance.

An alternative is to enable nested threading:

```
omp_set_nested(1);
#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);
  }
}
```

In this example, the inner loop is also parallelised. However, instead of N threads, the user will find N*N threads running, because the nested `parallel` directive now starts a new team of N threads instead of starting a single-thread team. This might be highly undesirable, hence why the nesting flag is disabled by default.

So it might be wisest to write the code like this:

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

Now only the inner loop is run in parallel. In most cases, it is only slightly less optimal than the code that was rewritten to use only one loop.

Performance

One may be worried about the creation of new threads within the inner loop. Worry not, the libgomp in GCC is smart enough to actually only creates the threads once. Once the team has done its work, the threads are returned into a "dock", waiting for new work to do.

In other words, the number of times the `clone` system call is executed is exactly equal to the maximum number of concurrent threads.
The `parallel` directive is not the same as a combination of `pthread_create` and `pthread_join`.

There will be lots of locking/unlocking due to the implied barriers, though. I don't know if that can be reasonably avoided or whether it even should.

There is also overhead to calling the internal OpenMP functions involved in assigning the threads to handle the loop.

Restrictions

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

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()`.

Cancelling of threads (and breaking out of loops)

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 finder problems where N threads search for a solution and once a solution is found by any thread, all threads end their search, you will need to seek other resolutions:

- Use pthreads and `pthread_cancel()` instead of OpenMP.
- Poll an interrupt flag.

Example of polling an interrupt flag ("done").

```
 const char* FindAnyNeedle(const char* haystack, size_t size, char
needle)
 {
   const char* result = NULL;
   bool done = false;
   #pragma omp parallel for
   for(size_t p = 0; p < size; ++p)
   {
     #pragma omp flush(done)
     if(!done)
     {
       /* Do work only if no thread has found the needle yet. */
       if(haystack[p] == needle)
       {
         /* Inform the other threads that we found the needle. */
         done = true;
         #pragma omp flush(done)
         result = haystack+p;
       }
     }
   }
   return result;
 }
```

However, the above version has a performance shortcoming: if a thread detects that the search is over ("done" is true), they will still iterate through their respective chunks of the `for` loop. You could just as well ignore the whole "done" flag:

```
 const char* FindAnyNeedle(const char* haystack, size_t size, char
needle)
 {
   const char* result = NULL;
   #pragma omp parallel for
   for(size_t p = 0; p < size; ++p)
     if(haystack[p] == needle)
       result = haystack+p;
   return result;
 }
```

This does not incur any savings in memory access; each position in "haystack" is still searched. Furthermore, the value of "result" may be set multiple times, once for each match. If matches are abundant, it may be a lot slower than the non-parallel version was.

The only possibility to avoid the extra loops is to avoid the OpenMP `for` directive alltogether and write it manually:

```
 const char* FindAnyNeedle(const char* haystack, size_t size, char
needle)
 {
   const char* result = NULL;
```

```
   bool done = false;
   #pragma omp parallel
   {
     int this_thread = omp_get_thread_num(), num_threads =
omp_get_num_threads();
     size_t beginpos = (this_thread+0) * size / num_threads;
     size_t endpos   = (this_thread+1) * size / num_threads;
     // watch out for overflow in that multiplication.

     for(size_t p = beginpos; p < endpos; ++p)
     {
       /* End loop if another thread already found the needle. */
       #pragma omp flush(done)
       if(done) break;

       if(haystack[p] == needle)
       {
         /* Inform the other threads that we found the needle. */
         done = true;
         #pragma omp flush(done)
         result = haystack+p;
         break;
       }
     }
   }
   return result;
 }
```

In this version, no extra iterations are done, but there is the fact that the "done" variable is polled at each loop. If that bothers you, you can seek a middle ground by dividing the loop into shorter sections where you don't poll the interrupt flag, and only checking it in the between of them:

```
 const char* FindAnyNeedle(const char* haystack, size_t size, char
needle)
 {
   const char* result = NULL;
   bool done = false;
   size_t beginpos = 0, n_per_loop = 4096 * omp_get_max_threads();
   while(beginpos < size && !done)
   {
     size_t endpos = std::min(size, beginpos + n_per_loop);
     #pragma omp parallel for reduction(|:done)
     for(size_t p = beginpos; p < endpos; ++p)
       if(haystack[p] == needle)
       {
         /* Found it. */
         done = true;
         result = haystack+p;
       }
     beginpos = endpos;
   }
   return result;
 }
```

This however again does not guarantee that no extra comparisons are done; the value of "result" may be set multiple times in this function.

If this is also unacceptable, then OpenMP has no more solutions remaining. You can use pthreads and try `pthread_create` to create the team and `pthread_cancel` to topple the sibling threads. Such a solution is not portable and is out of the scope of this article. (It can however be downloaded here.)

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 directives. 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.

C

- C does not allow variables to be declared in the `for` syntax, which means that by default, variables used for `for` iteration become shared. You must use the `private` clause explictly for loop variables.

GCC

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

Other

For the data copying clauses such as `threadprivate`, `copyprivate` and `copyin`, see the official specification.

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 2.5 official specification
- The OpenMP 3.0 official specification
- Wikipedia article for OpenMP
- OpenMP crash course at ARSC — has especially good list of gotchas.

Last edited: 2014-10-12 12:18:41