FREE eBook

LEARNING openmp

Free unaffiliated eBook created from
Stack Overflow contributors.

#openmp

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: openmp

It is an unofficial and free openmp ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official openmp.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with openmp

Remarks

OpenMP (Open MultiProcessing) is a parallel programming model based on compiler directives which allows application developers to incrementally add parallelism to their application codes.

OpenMP API specification for parallel programming provides an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

Since OpenMP focuses on the parallelism within a node (shared memory multiprocessing) it can be combined with message-passing programming models, such as MPI, to execute on multiple nodes.

Versions

Version	Language	Release date
4.5	C/C++/Fortran	2015-11-01
4.0	C/C++/Fortran	2013-07-01
3.1	C/C++/Fortran	2011-07-01
3.0	C/C++/Fortran	2008-05-01
2.5	C/C++/Fortran	2005-05-01
2.0c	C/C++	2002-03-01
2.0f	Fortran	2000-11-01
1.0c	C/C++	1998-10-01
1.0f	Fortran	1997-10-01

Examples

Compilation

There are many compilers that support different versions of the OpenMP specification. OpenMP maintains a list here with the compiler that support it and the supported version. In general, to compile (and link) an application with OpenMP support you need only to add a compile flag and if

you use the OpenMP API you need to include the OpenMP header (omp.h). While the header file has a fixed name, the compile flag depends on the compiler. The following is a non-exhaustive list of compilers and the flag that enables OpenMP.

- GCC (including gcc, g++ and gfortran) : `-fopenmp`
- LLVM: -fopenmp
- Intel compiler-suite (including icc, icpc and ifort) : `-qopenmp` (and `-fopenmp` for compatibility with GCC/LLVM)
- IBM XL compiler-suite (including xlc, xlC and xlf) : `-xlsmp=omp`
- PGI compiler-suite (including pgcc pgc++ pgfortran) : '-mp'

Parallel hello world using OpenMP

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

int main (int argc, char *argv[])
{
   #pragma omp parallel
   {
     printf ("Hello world! I'm thread %d out of %d threads.\n",
              omp_get_thread_num(), omp_get_num_threads());
   }
   return 0;
}
```

This code simply creates a team of threads (according to the environment variable `OMP_NUM_THREADS` - and if not defined will create one per logical core on the system) and each thread will identify itself besides printing the typical Hello world message.

Work Sharing construct - Example of For loop

```
double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

The for loop will be executed in parallel. huge() is some method which can take too long to get execute. OpenMP supports a shortcut to write the above code as :

```
double  res[MAX];  int i;
#pragma omp parallel for
for (i=0;i< MAX; i++) {
    res[i] = huge();
}
```

We can also have a schedule clause which effects how loop iterations are mapped to threads. For example:

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;I<N;i++) {
    a[i] = a[i] + b[i];
}
```

Different styles of scheduling are:

schedule(static [,chunk])
Deal-out blocks of iterations of size “chunk” to each thread.
If not specified: allocate as evenly as possible to the available threads

schedule(dynamic[,chunk])
Each thread grabs “chunk” iterations off a queue until all iterations have been handled.

schedule(guided[,chunk])
Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.

schedule(runtime)
Schedule and chunk size taken from the OMP_SCHEDULE environment variable.

Reduction Example

```
#include <omp.h>
void main ()
{
    int i;
    double ZZ, func(), res=0.0;

    #pragma omp parallel for reduction(+:res) private(ZZ)
    for (i=0; i< 1000; i++){
        ZZ = func(I);
        res = res + ZZ;
    }
}
```

In the last line: Actually added to a private copy, then combined after the loop. Compiler takes care of the details.

Read Getting started with openmp online: https://riptutorial.com/openmp/topic/4354/getting-started-with-openmp

Chapter 2: Conditional parallel execution

Examples

Conditional clauses in OpenMP parallel regions

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

int main (void)
{
  int t = (0 == 0); // true value
  int f = (1 == 0); // false value

  #pragma omp parallel if (f)
  { printf ("FALSE: I am thread %d\n", omp_get_thread_num()); }

  #pragma omp parallel if (t)
  { printf ("TRUE : I am thread %d\n", omp_get_thread_num()); }

  return 0;
}
```

Its output is:

```
$ OMP_NUM_THREADS=4 ./test
FALSE: I am thread 0
TRUE : I am thread 0
TRUE : I am thread 1
TRUE : I am thread 3
TRUE : I am thread 2
```

Read Conditional parallel execution online: https://riptutorial.com/openmp/topic/6928/conditional-parallel-execution

Chapter 3: Irregular OpenMP parallelism

Remarks

A common pitfall is to believe that all threads of a parallel region should instantiate (create) tasks but this is not typically the case unless you want to create as many tasks as the number of threads times the number of elements to process. Therefore, in OpenMP task codes you'll find something similar to

```
#pragma omp parallel
#pragma omp single
...
   #pragma omp task
   { code for a given task; }
...
```

Examples

Parallel processing of a c++ list container using OpenMP tasks

```
#include <omp.h>
#include <unistd.h>
#include <iostream>
#include <list>

static void processElement (unsigned n)
{
    // Tell who am I. The #pragma omp critical ensures that
    // only one thread sends data to std::cout
    #pragma omp critical
    std::cout <<
      "Thread " << omp_get_thread_num() << " processing element " << n
      << std::endl;

    // Simulate some work
    usleep (n*1000);
}

int main (void)
{
    std::list<unsigned> lst;

    // Fill the list
    for (unsigned u = 0; u < 16; ++u)
            lst.push_back (1+u);

    // Now process each element of the list in parallel

    #pragma omp parallel  // Create a parallel region
    #pragma omp single    // Only one thread will instantiate tasks
    {
            for (auto element : lst)
            {
```

```
                    #pragma omp task firstprivate (element)
                    processElement (element);
            }

            // Wait for all tasks to be finished
            #pragma omp taskwait
    }

    return 0;
}
```

This example simulates the processing of a STL list (named `lst` in the code) in parallel through the OpenMP task constructs (using the `#pragma omp task` directive). The example creates/instantiates one OpenMP task for each element in `lst` and the OpenMP threads execute the tasks as soon as they're ready to run.

```
$ OMP_NUM_THREADS=4 ./a.out
Thread 0 processing element 16
Thread 3 processing element 3
Thread 2 processing element 1
Thread 1 processing element 2
Thread 2 processing element 4
Thread 1 processing element 5
Thread 3 processing element 6
Thread 2 processing element 7
Thread 1 processing element 8
Thread 3 processing element 9
Thread 2 processing element 10
Thread 1 processing element 11
Thread 0 processing element 15
Thread 3 processing element 12
Thread 2 processing element 13
Thread 1 processing element 14
```

Recursive calculation for pi using OpenMP tasks

The code below calculates the value of PI using a recursive approach. Modify the `MAX_PARALLEL_RECURSIVE_LEVEL` value to determine at which recursion depth stop creating tasks. With this approach to create parallelism out of recursive applications: the more tasks you create, the more parallel tasks created but also the lesser work per task. So it is convenient to experiment with the application to understand at which level it creating further tasks do not benefit in terms of performance.

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

double pi_r (double h, unsigned depth, unsigned maxdepth, unsigned long long begin, unsigned
long long niters)
{
    if (depth < maxdepth)
    {
        double area1, area2;

        // Process first half
        #pragma omp task shared(area1)
```

```
        area1 = pi_r (h, depth+1, maxdepth, begin, niters/2-1);

        // Process second half
        #pragma omp task shared(area2)
        area2 = pi_r (h, depth+1, maxdepth, begin+niters/2, niters/2);

        #pragma omp taskwait

        return area1+area2;
    }
    else
    {

        unsigned long long i;
        double area = 0.0;

        for (i = begin; i <= begin+niters; i++)
        {
            double x = h * (i - 0.5);
            area += (4.0 / (1.0 + x*x));
        }

        return area;
    }
}

double pi (unsigned long long niters)
{
    double res;
    double h = 1.0 / (double) niters;

    #pragma omp parallel shared(res)
    {
#define MAX_PARALLEL_RECURSIVE_LEVEL 4

        #pragma omp single
        res = pi_r (h, 0, MAX_PARALLEL_RECURSIVE_LEVEL, 1, niters);
    }
    return res * h;
}


int main (int argc, char *argv[])
{
#define NITERS (100*1000*1000ULL)

    printf ("PI (w/%d iters) is %lf\n", NITERS, pi(NITERS));

    return 0;
}
```

Read Irregular OpenMP parallelism online: https://riptutorial.com/openmp/topic/6930/irregular-openmp-parallelism

Chapter 4: Loop parallelism in OpenMP

Parameters

Clause	Parameter
`private`	Comma-separated list of private variables
`firstprivate`	Like `private`, but initialized to the value of the variable before entering the loop
`lastprivate`	Like `private`, but the variable will get the value corresponding to the last iteration of the loop upon exit
`reduction`	reduction operator : comma-separated list of corresponding reduction variables
`schedule`	`static`, `dynamic`, `guided`, `auto` or `runtime` with an optional chunk size after a coma for the 3 former
`collapse`	Number of perfectly nested loops to collapse and parallelize together
`ordered`	Tells that some parts of the loop will need to be kept in-order (these parts will be specifically identified with some `ordered` clauses inside the loop body)
`nowait`	Remove the implicit barrier existing by default at the end of the loop construct

Remarks

The meaning of the `schedule` clause is as follows:

- `static[,chunk]`: Distribute statically (meaning that the distribution is done before entering the loop) the loop iterations in batched of `chunk` size in a round-robin fashion. If `chunk` isn't specified, then the chunks are as even as possible and each thread gets at most one of them.
- `dynamic[,chunk]`: Distribute the loop iterations among the threads by batches of `chunk` size with a first-come-first-served policy, until no batch remains. If not specified, `chunk` is set to 1
- `guided[,chunk]`: Like `dynamic` but with batches which sizes get smaller and smaller, down to 1
- `auto`: Let the compiler and/or run time library decide what is best suited
- `runtime`: Deffer the decision at run time by mean of the `OMP_SCHEDULE` environment variable. If at run time the environment variable is not defined, the default scheduling will be used

The default for `schedule` is **implementation define**. On many environments it is `static`, but can also be `dynamic` or could very well be `auto`. Therefore, be careful that your implementation doesn't implicitly rely on it without explicitly setting it.

In the above examples, we used the fused form `parallel for` or `parallel do`. However, the loop

construct can be used without fusing it with the `parallel` directive, in the form of a `#pragma omp for [...]` or `!$omp do [...]` standalone directive within a `parallel` region.

For the Fortran version only, the loop index variable(s) of the parallized loop(s) is (are) always `private` by default. There is therefore no need of explicitly declaring them `private` (although doing so isn't a error).
For the C and C++ version, the loop indexes are just like any other variables. Therefore, if their scope extends outside of the parallelized loop(s) (meaning if they are not declared like `for ( int i = ...)` but rather like `int i; ... for ( i = ... )` then they **have to** be declared `private`.

Examples

Typical example in C

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

#define N 1000000

int main() {
    double sum = 0;

    double tbegin = omp_get_wtime();
    #pragma omp parallel for reduction( +: sum )
    for ( int i = 0; i < N; i++ ) {
        sum += cos( i );
    }
    double wtime = omp_get_wtime() - tbegin;

    printf( "Computing %d cosines and summing them with %d threads took %fs\n",
            N, omp_get_max_threads(), wtime );

    return sum;
}
```

In this example, we just compute 1 million cosines and sum their values in parallel. We also time the execution to see whether the parallelization has any effect on the performance. Finally, since we do measure the time, we have to make sure that the compiler won't optimize away the work we've done, so we pretend using the result by just returning it.

Same example in Fortran

```
program typical_loop
    use omp_lib
    implicit none
    integer, parameter :: N = 1000000, kd = kind( 1.d0 )
    real( kind = kd ) :: sum, tbegin, wtime
    integer :: i

    sum = 0

    tbegin = omp_get_wtime()
```

```
    !$omp parallel do reduction( +: sum )
    do i = 1, N
        sum = sum + cos( 1.d0 * i )
    end do
    !$omp end parallel do
    wtime = omp_get_wtime() - tbegin

    print "( 'Computing ', i7, ' cosines and summing them with ', i2, &
        & ' threads took ', f6.4,'s' )", N, omp_get_max_threads(), wtime

    if ( sum > N ) then
        print *, "we only pretend using sum"
    end if
end program typical_loop
```

Here again we compute and accumulate 1 million cosines. We time the loop and to avoid unwanted compiler optimization-away of it, we pretend using the result.

Compiling and running the examples

On a 8 cores Linux machine using GCC version 4.4, the C codes can be compiled and run the following way:

```
$ gcc -std=c99 -O3 -fopenmp loop.c -o loopc -lm
$ OMP_NUM_THREADS=1 ./loopc
Computing 1000000 cosines and summing them with 1 threads took 0.095832s
$ OMP_NUM_THREADS=2 ./loopc
Computing 1000000 cosines and summing them with 2 threads took 0.047637s
$ OMP_NUM_THREADS=4 ./loopc
Computing 1000000 cosines and summing them with 4 threads took 0.024498s
$ OMP_NUM_THREADS=8 ./loopc
Computing 1000000 cosines and summing them with 8 threads took 0.011785s
```

For the Fortran version, it gives:

```
$ gfortran -O3 -fopenmp loop.f90 -o loopf
$ OMP_NUM_THREADS=1 ./loopf
Computing 1000000 cosines and summing them with  1 threads took 0.0915s
$ OMP_NUM_THREADS=2 ./loopf
Computing 1000000 cosines and summing them with  2 threads took 0.0472s
$ OMP_NUM_THREADS=4 ./loopf
Computing 1000000 cosines and summing them with  4 threads took 0.0236s
$ OMP_NUM_THREADS=8 ./loopf
Computing 1000000 cosines and summing them with  8 threads took 0.0118s
```

Addition of two vectors using OpenMP parallel for construct

```
void parallelAddition (unsigned N, const double *A, const double *B, double *C)
{
    unsigned i;

    #pragma omp parallel for shared (A,B,C,N) private(i) schedule(static)
    for (i = 0; i < N; ++i)
    {
        C[i] = A[i] + B[i];
```

```
    }
}
```

This example adds two vector (`A` and `B` into `C`) by spawning a team of threads (specified by the `OMP_NUM_THREADS` environtment variable, for instance) and assigning each thread a chunk of work (in this example, assigned statically through the `schedule(static)` expression).

See remarks section with respect to the `private(i)` optionality.

Read Loop parallelism in OpenMP online: https://riptutorial.com/openmp/topic/5657/loop-parallelism-in-openmp

Chapter 5: OpenMP reductions

Remarks

All 4 version are valid, but they exemplify different aspects of a reduction.

By default, the first construct using the `reduction` clause **must be preferred**. This is only if some issues are explicitly identified that any of the 3 alternatives might be explored.

Examples

Approximation of PI hand-crafting the #pragma omp reduction

```
int i;
int n = 1000000;
double area = 0;
double h = 1.0 / n;

#pragma omp parallel shared(n, h)
{
  double thread_area = 0;                       // Private / local variable

  #pragma omp for
  for (i = 1; i <= n; i++)
  {
    double x = h * (i - 0.5);
    thread_area += (4.0 / (1.0 + x*x));
  }

  #pragma omp atomic                          // Applies the reduction manually
  area += thread_area;                        // All threads aggregate into area
}
double pi = h * area;
```

The threads are spawned in the #pragma omp parallel. Each thread will have an independent/private thread_area that stores its partial addition. The following loop is distributed among threads using #pragma omp for. In this loop, each thread calculates its own thread_area and after this loop, the code sequentially aggregates the area atomically through

Approximation of PI using reductions based on #pragma atomic

```
double area;
double h = 1.0 / n;
#pragma omp parallel for shared(n, h, area)
for (i = 1; i <= n; i++)
{
  double x = h * (i - 0.5);
  #pragma atomic
  area += (4.0 / (1.0 + x*x));
}
pi = h * area;
```

In this example, each threads execute a subset of the iteration count and they accumulate atomically into the shared variable area, which ensures that there are no lost updates. We can use the #pragma atomic in here because the given operation (+=) can be done atomically, which simplifies the readability compared to the usage of the #pragma omp critical.

Approximation of PI using reductions based on #pragma omp critical

```
double area;
double h = 1.0 / n;
#pragma omp parallel for shared(n, h, area)
for (i = 1; i <= n; i++)
{
  double x = h * (i - 0.5);
  #pragma omp critical
  {
    area += (4.0 / (1.0 + x*x));
  }
}
double pi = h * area;
```

In this example, each threads execute a subset of the iteration count and they accumulate atomically into the shared variable area, which ensures that there are no lost updates.

Approximation of PI using #pragma omp reduction clause

```
int i;
int n = 1000000;
double area = 0;
double h = 1.0 / n;
#pragma omp parallel for shared(n, h) reduction(+:area)
for (i = 1; i <= n; i++)
{
  double x = h * (i - 0.5);
  area += (4.0 / (1.0 + x*x));
}
pi = h * area;
```

In this example, each threads execute a subset of the iteration count. Each thread has its local private copy of area and at the end of the parallel region they all apply the addition operation (+) so as to generate the final value for area.

Read OpenMP reductions online: https://riptutorial.com/openmp/topic/5653/openmp-reductions

Chapter 6: OpenMP reductions

Examples

Approximation of PI using #pragma omp reduction clause

```
h = 1.0 / n;
#pragma omp parallel for private(x) shared(n, h) reduction(+:area)
for (i = 1; i <= n; i++)
{
  x = h * (i - 0.5);
  area += (4.0 / (1.0 + x*x));
}
pi = h * area;
```

In this example, each threads execute a subset of the iteration count. Each thread has its local private copy of `area` and at the end of the parallel region they all apply the addition operation (`+`) so as to generate the final value for `area`.

Approximation of PI using reductions based on #pragma omp critical

```
h = 1.0 / n;
#pragma omp parallel for private(x) shared(n, h, area)
for (i = 1; i <= n; i++)
{
  x = h * (i - 0.5);
  #pragma omp critical
  {
    area += (4.0 / (1.0 + x*x));
  }
}
pi = h * area;
```

In this example, each threads execute a subset of the iteration count and they accumulate atomically into the shared variable `area`, which ensures that there are no lost updates.

Approximation of PI using reductions based on #pragma atomic

```
h = 1.0 / n;
#pragma omp parallel for private(x) shared(n, h, area)
for (i = 1; i <= n; i++)
{
  x = h * (i - 0.5);
  #pragma atomic
  area += (4.0 / (1.0 + x*x));
}
pi = h * area;
```

In this example, each threads execute a subset of the iteration count and they accumulate atomically into the shared variable `area`, which ensures that there are no lost updates. We can use

the `#pragma atomic` in here because the given operation (`+=`) can be done atomically, which simplifies the readability compared to the usage of the `#pragma omp critical`.

Approximation of PI hand-crafting the #pragma omp reduction

```
h = 1.0 / n;

#pragma omp parallel private(x) shared(n, h)
{
  double thread_area = 0;                       // Private / local variable

  #pragma omp for
  for (i = 1; i <= n; i++)
  {
    x = h * (i - 0.5);
    thread_area += (4.0 / (1.0 + x*x));
  }

  #pragma omp atomic                          // Applies the reduction manually
  area += thread_area;                        // All threads aggregate into area
}

pi = h * area;
```

The threads are spawned in the `#pragma omp parallel`. Each thread will have an independent/private `thread_area` that stores its partial addition. The following loop is distributed among threads using `#pragma omp for`. In this loop, each thread calculates its own `thread_area` and after this loop, the code sequentially aggregates the area atomically through `#pragma omp atomic`.

Read OpenMP reductions online: https://riptutorial.com/openmp/topic/5967/openmp-reductions

Chapter 7: Simple parallel example

Syntax

- `#pragma omp parallel` indicates that the following block shall be executed by all the threads.
- `int omp_get_num_threads (void)` : returns the number of the threads working on the parallel region (aka team of threads).
- `int omp_get_thread_num (void)` : returns the identifier of the calling thread (ranges from 0 to N-1 where N is bounded to `omp_get_num_threads()`).

Remarks

You can use the `OMP_NUM_THREADS` environment variable or the `num_threads` directive within the `#pragma parallel` to indicate the number of executing threads for the whole application or for the specified region, respectively.

Examples

Parallel hello world using OpenMP

The following C code uses the OpenMP parallel programming model to write the thread ID and number of threads to `stdout` using multiple threads.

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

int main ()
{
    #pragma omp parallel
    {
        // ID of the thread in the current team
        int thread_id = omp_get_thread_num();
        // Number of threads in the current team
        int nthreads = omp_get_num_threads();

        printf("I'm thread %d out of %d threads.\n", thread_id, nthreads);
    }
    return 0;
}
```

In Fortran 90+ the equivalent program looks like:

```
program Hello
  use omp_lib, only: omp_get_thread_num, omp_get_num_threads

  implicit none
```

```
  integer :: thread_id
  integer :: nthreads

  !$omp parallel private( thread_id, nthreads )

  ! ID of the thread in the current team
  thread_id = omp_get_thread_num()
  ! Number of threads in the current team
  nthreads = omp_get_num_threads()

  print *, "I'm thread", thread_id, "out of", nthreads, "threads."
  !$omp end parallel
end program Hello
```

Read Simple parallel example online: https://riptutorial.com/openmp/topic/4959/simple-parallel-example

Credits

S. No	Chapters	Contributors
1	Getting started with openmp	Ani Menon, Community, Gilles, Harald, M. Chinoune, Massimiliano, takirala
2	Conditional parallel execution	Harald
3	Irregular OpenMP parallelism	Harald
4	Loop parallelism in OpenMP	Gilles, Harald, NoseKnowsAll
5	OpenMP reductions	Gilles, Harald, meJustAndrew
6	Simple parallel example	Gilles, Harald, Massimiliano, NoseKnowsAll, Vladimir F