# OpenMP Overview (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

# Based on

https://en.wikipedia.org/wiki/OpenMP

# OpenMP

OpenMP (Open Multi-Processing) is
an application programming interface (API)
that supports multi-platform shared memory multiprocessing
programming in C, C++, and Fortran, on most platforms

An application built with the hybrid model
OpenMP is used for parallelism within a (multi-core) node
MPI is used for parallelism between nodes.

https://en.wikipedia.org/wiki/OpenMP

# Fork-Join Model

OpenMP is an implementation of multithreading,
a method of parallelizing whereby a master thread
(a series of instructions executed consecutively)
forks a specified number of slave threads
and the system divides a task among them.

The threads then run concurrently,
with the runtime environment allocating threads
to different processors.

https://en.wikipedia.org/wiki/OpenMP

# Thread ID

The section of code that is meant to run in parallel
is marked accordingly, with a **compiler directive**
that will cause the threads to form before the section is executed.

Each thread has an **id** attached to it
(obtained by **omp_get_thread_num()**).
The **thread id** is an integer, and the master thread has an id of **0**.

After the execution of the parallelized code,
the threads **join** back into the master thread,
which continues onward to the end of the program.

https://en.wikipedia.org/wiki/OpenMP

# Work Sharing Constructs

By default, each thread executes
the parallelized section of code independently.

**Work-sharing constructs** can be used
to divide a task among the threads

Both **task parallelism** and **data parallelism** can be achieved

https://en.wikipedia.org/wiki/OpenMP

# The runtime environment

The **runtime environment** allocates threads to processors
depending on usage, machine load and other factors.

The **runtime environment** or the **code**
can assign the **number** of threads based on environment variables,

The OpenMP functions are included
in a header file labelled **omp.h** in C/C++.

https://en.wikipedia.org/wiki/OpenMP

# The core elements

**thread** creation constructs

workload distribution (**work sharing**) constructs

**data-environment** management constructs

thread **synchronization** constructs

user-level **runtime routines** constructs

**environment variables** constructs

In C/C++, OpenMP uses **#pragmas**.

https://en.wikipedia.org/wiki/OpenMP

# Thread creation (1)

The **pragma omp parallel** is used to **fork** additional threads
to carry out the work enclosed in the construct in parallel.
The original thread will be denoted
as **master thread** with thread ID **0**.

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

int main(void) {
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

https://en.wikipedia.org/wiki/OpenMP

# Thread creation (2)

**$ gcc -fopenmp hello.c -o hello**

Output on a computer with two cores, and thus two threads:

Hello, world.
Hello, world.

However, the output may also be garbled because of the **race condition** caused from the two threads sharing the standard output.

Hello, wHello, woorld.
rld.

(in the case of using **C++ std::cout**, for example, the example is always true. **printf** can be or not thread-safe)

https://en.wikipedia.org/wiki/OpenMP

# Work-sharing creation (1)

to specify how to assign independent work
to one or all of the threads.

**omp for** or **omp do**: used to split up loop iterations
among the threads, also called loop constructs.
**sections**: assigning consecutive but independent code blocks
to different threads
**single**: specifying **a** code block that is executed
by only one thread, a **barrier** is implied in the end
**master**: similar to single, but the code block will be executed
by the **master thread** only and no barrier implied in the end.

https://en.wikipedia.org/wiki/OpenMP

# Work-sharing creation (2)

Example: initialize the value of a large array in parallel,
using each thread to do part of the work

```
int main(int argc, char **argv) {
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

https://en.wikipedia.org/wiki/OpenMP

# Work-sharing creation (3)

This example is embarrassingly parallel,
and depends only on the value of i.
The **parallel for** flag tells the OpenMP system
to split this task among its working threads.
The threads will each receive
a unique and private version of the **variable**.

For instance, with two worker threads,
one thread might be handed a version of i that runs from 0 to 49999
while the second gets a version running from 50000 to 99999.

https://en.wikipedia.org/wiki/OpenMP

# Clauses

Since OpenMP is a shared memory programming model,
most variables in OpenMP code are visible to all threads by default.

But sometimes **private** variables are necessary
to avoid **race conditions**
and there is a need to pass values
between the sequential part and the parallel region
(the code block executed in parallel),
so **data environment management** is introduced
as data sharing attribute clauses
by appending them to the OpenMP **directive**.

https://en.wikipedia.org/wiki/OpenMP

# Clause types

- Data sharing attribute clauses
- Synchronization clauses
- Scheduling clauses
- IF control
- Initialization
- Data copying
- Reduction
- Others

https://en.wikipedia.org/wiki/OpenMP

# Data sharing attribute clauses (0)

**shared**

**private**

**default**

**firstprivate**

**lastprivate**

**reduction**

https://en.wikipedia.org/wiki/OpenMP

# Data sharing attribute clauses (1)

**shared**:

the data within a parallel region is shared by all threads

visible and accessible by all threads simultaneously

by default, all variables in the work sharing region

are shared except the loop iteration counter.

**private**:

the data within a parallel region is private to each thread

each thread will have a **local copy** as a **temporary variable**.

not initialized and not maintained for use outside

by default, the loop iteration counters private.

https://en.wikipedia.org/wiki/OpenMP

# Data sharing attribute clauses (2)

**default**:

the default data scoping within a parallel region will be

either shared, or none for C/C++, or

shared, firstprivate, private, or none for Fortran.

the none option forces to declare each variable

in the parallel region using the data sharing attribute clauses.

**firstprivate**:

like **private** except initialized to original value.

**lastprivate**:

like **private** except original value is updated after construct.

**reduction**:

a safe way of joining work from all threads after construct.

https://en.wikipedia.org/wiki/OpenMP

# Synchronization clauses (0)

**Critical**

**atomic**

**ordered**

**barrier**

**nowait**

https://en.wikipedia.org/wiki/OpenMP

# Synchronization clauses (1)

**critical**:

executed by only one thread at a time

not simultaneously executed by multiple threads

to proctect shared data from race conditions.

**atomic**

the memory update (write, or read-modify-write)

in the next instruction will be performed atomically.

not make the entire statement atomic;

only the memory update is atomic.

might use a special hardware instructions

https://en.wikipedia.org/wiki/OpenMP

# Synchronization clauses (2)

**ordered**:
the structured block is executed in the order
in which iterations would be executed in a sequential loop

**barrier**:
each thread waits until all of the other threads of a team have
reached this point.
A work-sharing construct has
an implicit barrier synchronization at the end.

https://en.wikipedia.org/wiki/OpenMP

# Synchronization clauses (3)

**nowait**:

specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish.

In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

https://en.wikipedia.org/wiki/OpenMP

# Scheduling clauses (0)

**schedule(type, chunk)**: This is useful if the work sharing construct is a **do-loop** or **for-loop**. The **iteration**(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause. The three types of scheduling are:

- **Static**
- **dynamic**
- **guided**

https://en.wikipedia.org/wiki/OpenMP

# Scheduling clauses (1)

**static schedule(type, chunk)**:
all the threads are allocated iterations
before they execute the loop iterations.
The iterations are divided among threads equally by default.
However, specifying an integer for the **parameter chunk**
will allocate chunk number of contiguous iterations
to a particular thread.

https://en.wikipedia.org/wiki/OpenMP

# Scheduling clauses (2)

**dynamic schedule(type, chunk)**:

some of the iterations are allocated to a smaller number of threads.

Once a particular thread finishes its allocated iteration,

it returns to get another one from the iterations that are left.

The **parameter chunk** defines the number of contiguous iterations

that are allocated to a thread at a time.

https://en.wikipedia.org/wiki/OpenMP

# Scheduling clauses (2)

**guided schedule(type, chunk)**:
A **large chunk** of contiguous iterations are allocated
to each thread **dynamically** (as above).
The chunk size decreases exponentially
with each successive allocation to a minimum size
specified in the **parameter chunk**.

https://en.wikipedia.org/wiki/OpenMP

## References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf