OpenMP Loop Parallelism (2A)

- Loop
-

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

This document was produced by using OpenOffice and Octave.

Parallel region

The simplest way to create parallelism in OpenMP
is to use the **parallel** pragma.

A block preceded by the omp **parallel** pragma
is called a **parallel region** ;

it is executed by a newly created team of **threads**.

This is an instance of the **SPMD** model:
all **threads** execute the same segment of code.

```
#pragma omp parallel
{
  // this is executed by a team of threads
}
```

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-parallel.html

Parallel region

It would be pointless to have
the block be executed identically by all threads.

One way to get a meaningful parallel code is
to use the function **omp_get_thread_num** ,
to find out which thread you are,
and execute work that is individual to that thread.

There is also a function **omp_get_num_threads**
to find out the total number of threads.

Both these functions give a number relative to the current team;
recall from figure 15.3 that new teams can be created recursively.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-parallel.html

Parallel region

Immediately preceding the **parallel block**,
one **thread** will be executing the code.

In the **main** program this is the **initial thread**.

At the start of the block, a new team of **threads** is created,
and the **thread** that was active before the block becomes
the **master thread** of that team.

After the block, only the **master thread** is active.

Inside the block there is **team** of **threads**:

each **thread** in the **team** executes the body of the block,
and it will have access to all variables of the surrounding environment.

How many threads there are can be determined in a number of ways;

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-parallel.html

Parallel region

the **threads** that are forked are
all **copies** of the **master thread** :
they have access to all that was computed so far;
this is their **shared data**

Of course, if the **threads** were completely identical
the parallelism would be pointless,
so they also have **private data**,
and they can identify themselves:
they know their **thread number**.

this allows you to do meaningful
parallel computations with threads.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-parallel.html

Parallel region

work sharing constructs
In a **team** of **threads**,
initially there will be replicated execution;
a **work sharing construct** divides
available parallelism over the threads.

OpenMP uses **teams** of **threads**,
and inside a **parallel region**
the **work** is distributed over the **threads**
with a **work sharing construct**.
threads can access **shared data**,
and they have some **private data**.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-parallel.html

Parallel region

An important difference between **OpenMP** and **MPI** is
that parallelism in **OpenMP** is **dynamically activated**
by a **thread spawning** a **team** of **threads**.

Furthermore, the **number** of **threads** used
can differ between **parallel regions**,
and **threads** can create **threads** recursively.

This is known as as **dynamic mode** .

By contrast, in an **MPI** program
the **number** of running processes is
(mostly) constant throughout the run,
and determined by factors external to the program.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-parallel.html

Loop parallelism

OpenMP parallel loops are an example of
OpenMP `worksharing' constructs

take an amount of work and
distribute it over the available threads
in a parallel region.

The parallel execution of a loop
can be handled a number of different ways.

For instance, you can create
a parallel region around the loop,
and adjust the **loop bounds**:

```
#pragma omp parallel
{
  int threadnum = omp_get_thread_num(),
      numthreads = omp_get_num_threads();

  int low = N*threadnum/numthreads,
      high = N*(threadnum+1)/numthreads;

  for (i=low; i<high; i++)
    // do something with i
}
```

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop parallelism

use the **parallel for** pragma:

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
  // do something with i
}
```

you don't have to calculate
the loop bounds for the threads yourself,

but you can also tell OpenMP
to assign the loop iterations
according to different schedules

```
#pragma omp parallel
{
   code1();
   #pragma omp for
   for (i=1; i<=4*N; i++) {
      code2();
   }
   code3();
}
```

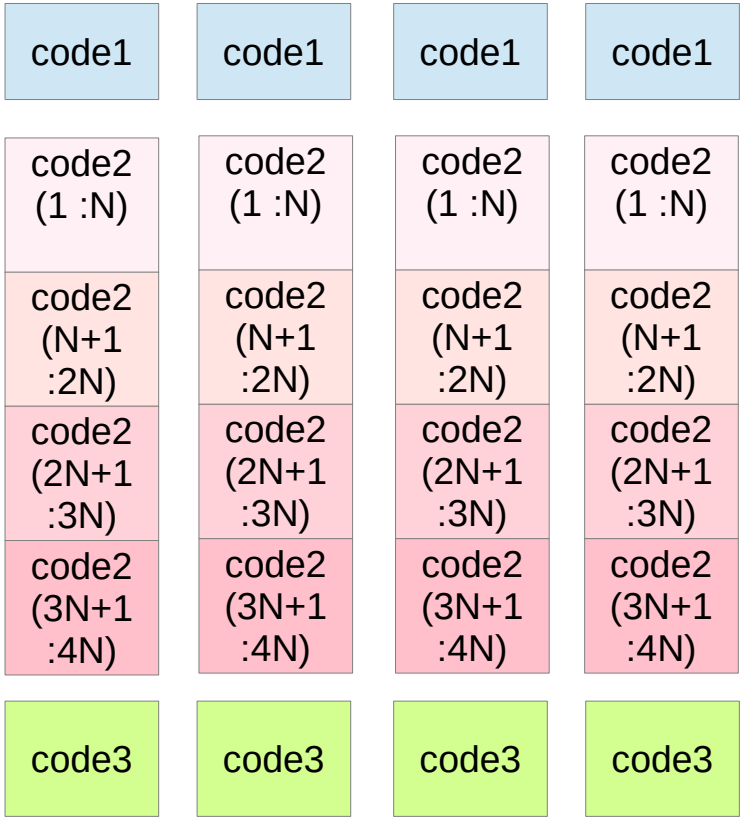
The code before and after the loop is
executed identically in each thread; t

he loop iterations are spread
over the four threads.

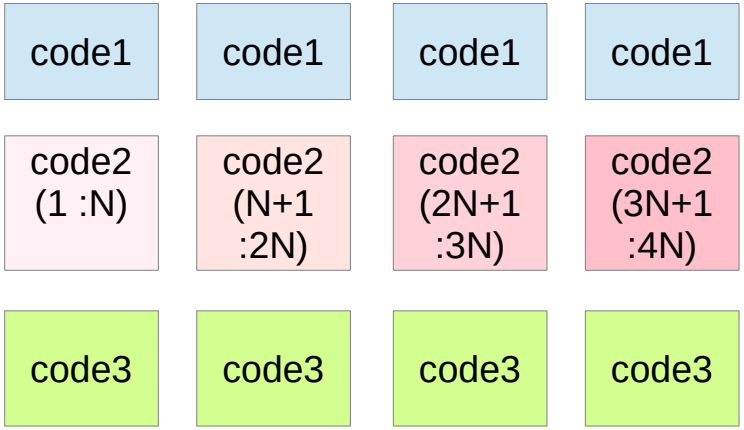
https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop parallelism

Without #pragma omp **for**



With #pragma omp **for**



https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop parallelism

Note that the **parallel do** and **parallel for** pragmas
do not create a team of threads:
they take the team of threads that is active,
and divide the loop iterations over them.

This means that the **omp for** or **omp do** directive
needs to be inside a **parallel region**.
It is also possible to have a combined
omp parallel for or **omp parallel do** directive.

If your parallel region only contains a loop,
you can combine the pragmas for the parallel region
and distribution of the loop iterations:

```
#pragma omp parallel for
  for (i=0; .....
```

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop parallelism

Note that the **parallel do** and **parallel for** pragmas
do not create a team of threads:
they take the team of threads that is active,
and divide the loop iterations over them.

This means that the **omp for** or **omp do** directive
needs to be inside a **parallel region**.
It is also possible to have a combined
omp parallel for or **omp parallel do** directive.

If your parallel region only contains a loop,
you can combine the pragmas for the parallel region
and distribution of the loop iterations:

```
#pragma omp parallel for
  for (i=0; .....
```

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop Schedules (1)

more iterations in a loop than **threads**
several ways to assign loop iterations to the **threads**
OpenMP lets you specify this with the **schedule clause**.

#pragma omp **for schedule**(....)

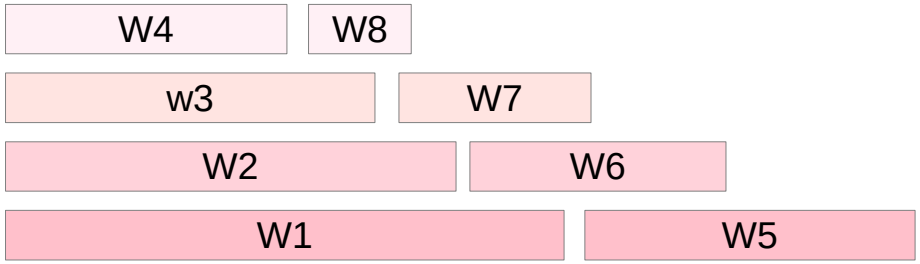
Static schedules
the iterations are assigned purely
based on the number of iterations
and the number of threads
(and the **chunk parameter**; see later).

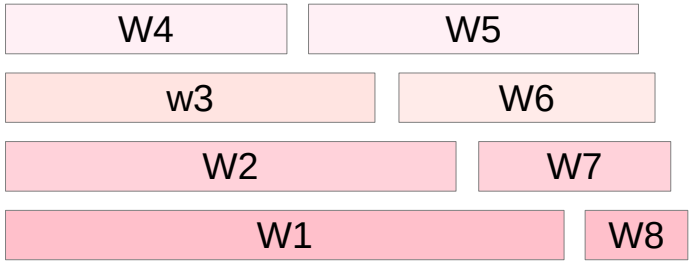
Dynamic schedules
iterations are assigned
to threads that are unoccupied.

when iterations take an unpredictable amount of time,
so **load balancing** is needed.

https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

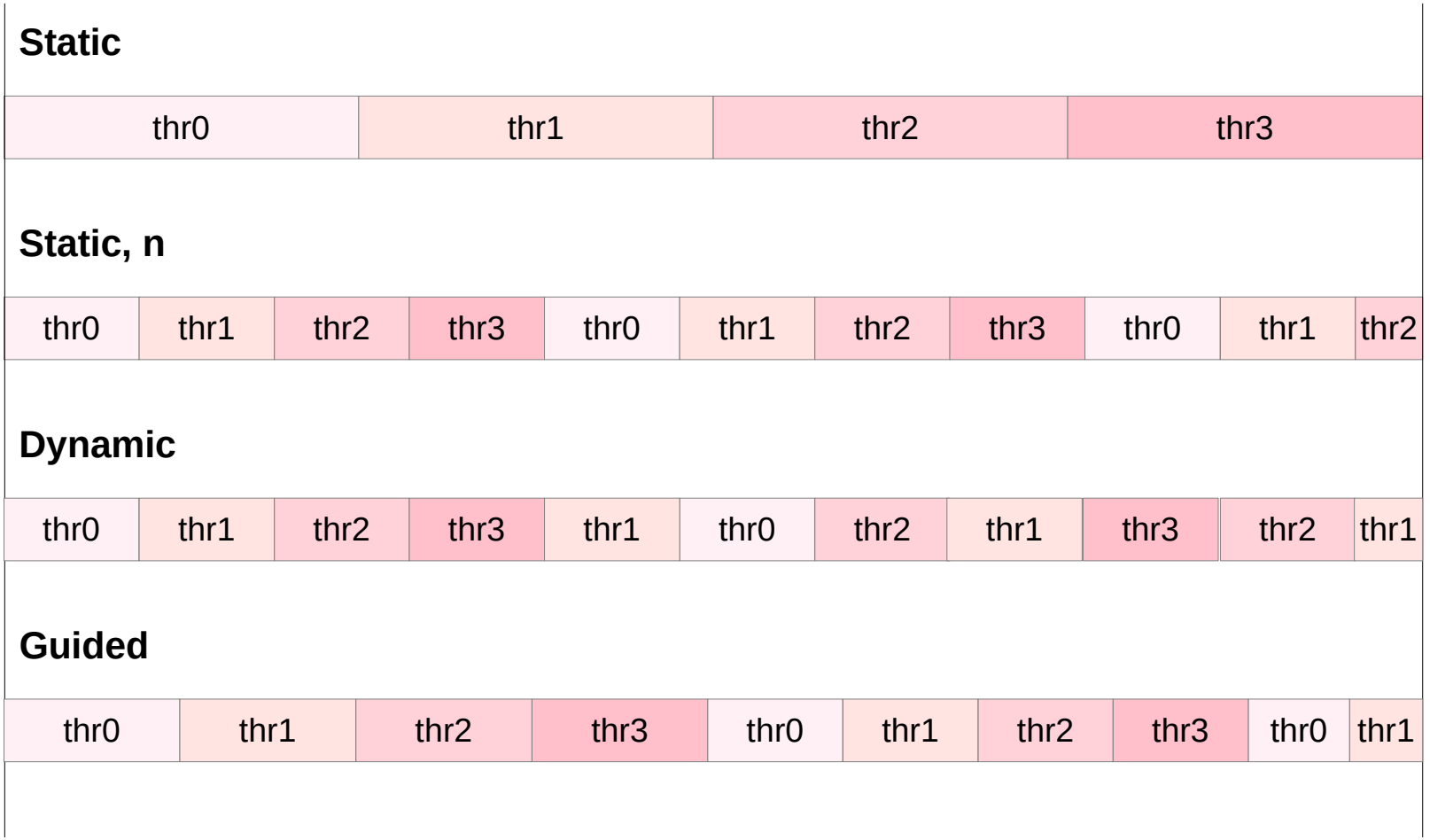
Loop Schedules (2)





https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop Schedules (3)



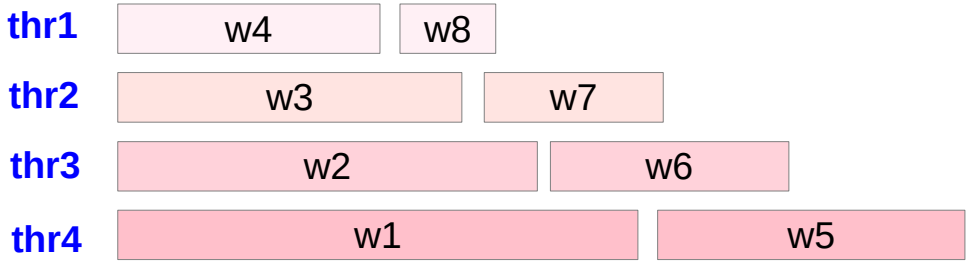
https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop Schedules (4)

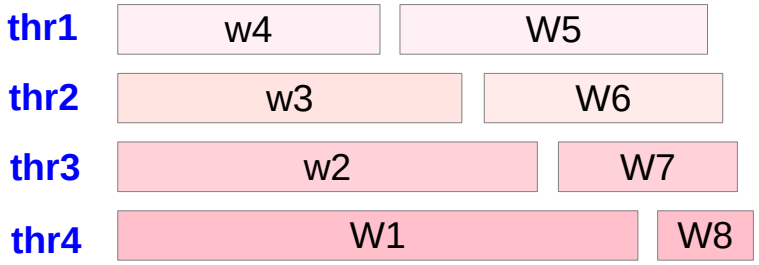
assume that each core gets assigned
two (blocks of) iterations and
these blocks take gradually
less and less time.

thread 1 gets two fairly long blocks,
whereas thread 4 gets two short blocks,

Thread 1 finishes much earlier.
Imbalance : unequal amounts of work



load balancing
thread 4 gets block 5,
since it finishes the first set of blocks early.
The effect is a perfect



https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop Schedule – Static (1)

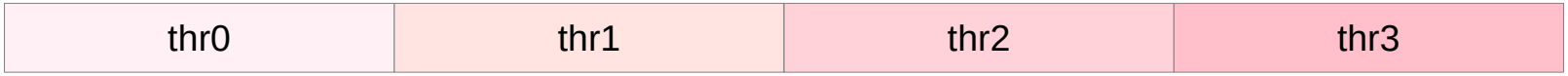
The **default static schedule** is
to assign one consecutive **block**
of iterations to each **thread**.

#pragma omp **for schedule(static)**
#pragma omp **for schedule(static, chunk)**

With **static scheduling**, the compiler
will split up the loop iterations at compile time,

When the iterations take roughly
the same amount of time,
this is the most efficient at runtime.

Static



https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

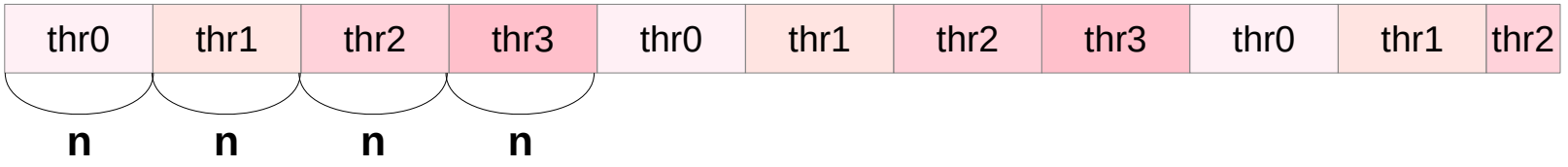
Loop Schedule – Static (2)

#pragma omp **for schedule(static,chunk)**

If you want different sized blocks
you can defined a **chunk size chunk**

The choice of a **chunk size** is often
a balance between the **low overhead**
of having only a few chunks, (big chunks)
versus the **load balancing effect**
of having smaller chunks. (many chunks)

Static, n



https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop Schedule – Static (3)

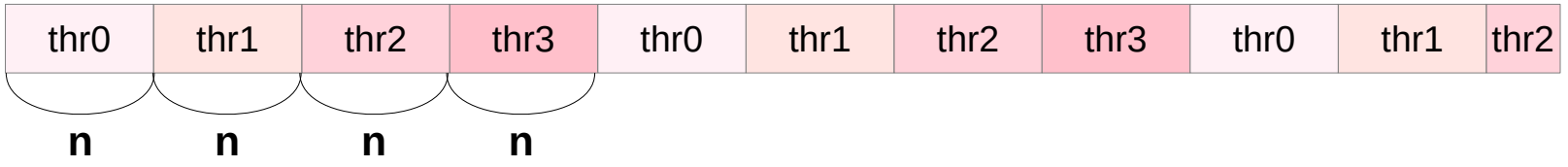
#pragma omp **for schedule(static,chunk)**

OpenMP divides the iterations
into chunks of size chunk-size

distributes the chunks to **threads**
in a circular order.

When no chunk-size is specified,
OpenMP divides iterations into chunks
that are approximately equal in size and
distributes at most one **chunk** to each **thread**.

Static, n



http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Static (4)

Static scheduling is used
when you know that each thread will do
the approximately same amount of work
at the **compile time**.

the following code can be parallelized
using OMP. (assume only 4 threads)

```
float A[100][100];

for(int i = 0; i < 100; i++)
{
   for(int j = 0; j < 100; j++)
   {
      A[i][j] = 1.0f;
   }
}
```

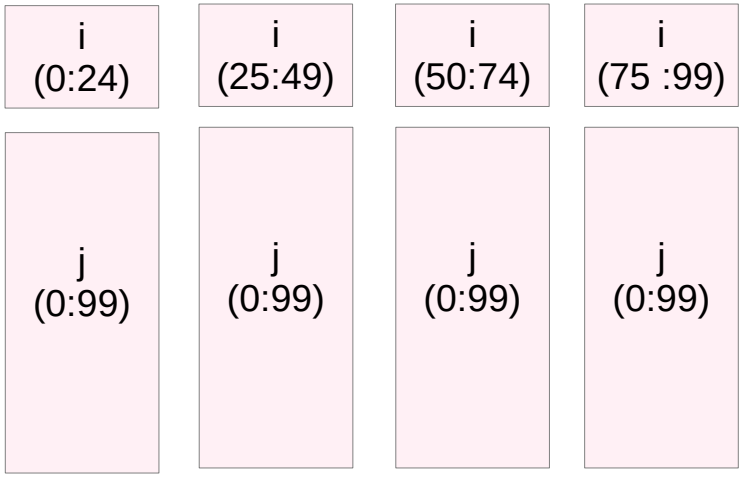
100 * 100 iterations
10000 / 4 iterations / thread

https://stackoverflow.com/questions/15508128/using-omp-schedule-with-pragma-omp-for-parallel-scheduleruntime

Loop Schedule – Static (5)

```
float A[100][100];

#pragma omp for schedule(static)
for(int i = 0; i < 100; i++)
{
   for(int j = 0; j < 100; j++)
   {
      A[i][j] = 1.0f;
   }
}
```



to use the default static scheduling,
place pragma on the outer for loop,

then each thread will do
25% of the outer loop (i) work
and eqaul amount of inner loop (j) work

Hence, the total amount of work done
by each thread is same.

Hence, we could simply stick
with the default static scheduling
to give optimal load balancing.

https://stackoverflow.com/questions/15508128/using-omp-schedule-with-pragma-omp-for-parallel-scheduleruntime

Loop Schedule – Static (6)

a for loop with 64 iterations
4 threads
each row represents a thread.

schedule(static) has 16 iterations in the first row.
64 iterations and 4 threads → 64 / 4 = 16
the first tread executes iterations 1, 2, 3, …, 15 and 16.
the second thread executes iterations 17, 18, 19, …, 31, 32.
Similar applies to the threads three and four.

For **schedule(static)**,
OpenMP divides iterations into four chunks of size 16
and it distributes them to four threads.

For **schedule(static, 4)** and **schedule(static, 8)**
OpenMP divides iterations into chunks of size 4 and 8, respectively.

The static scheduling type is appropriate
when all iterations have the same computational cost.

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Static (6)

schedule**(static, 4)**

0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51
4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55
8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59
12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63

schedule**(static, 8)**

0	1	2	3	4	5	6	7	32	33	34	35	36	37	38	39
8	9	10	11	12	13	14	15	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	48	49	50	51	52	53	54	55
24	25	26	27	28	29	30	31	56	57	58	59	60	61	62	63

schedule**(static)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

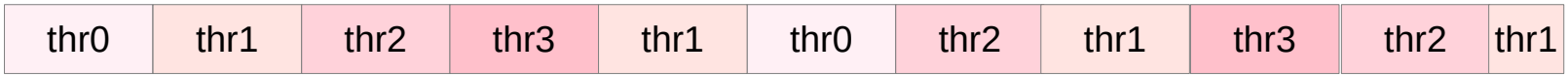
Loop Schedule – Dynamic (1)

In **dynamic scheduling** OpenMP will
put **blocks** of iterations in a **task queue**,
(the **default chunk size** is **1**)
and the **threads** take one of these tasks
whenever they are finished with the previous.

#pragma omp **for schedule(dynamic[,chunk])**

While this schedule may give **good load balancing**
if the iterations take very differing amounts of time to execute,
it does carry **runtime overhead**
for managing the **queue** of iteration tasks.

Dynamic



https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

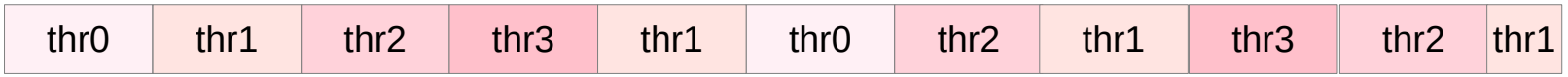
Loop Schedule – Dynamic (2)

large **chunks** carry the least **overhead**,
but smaller chunks are better for **load balancing**.

If you don't want to decide on a schedule in your code,
you can specify the schedule will then **at runtime**
be read from the OMP_SCHEDULE environment variable.

You can even just leave it to the runtime library by specifying
omp_set_schedule.

Dynamic



https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

Loop Schedule – Dynamic (3)

for schedule(dynamic, chunk-size)
the dynamic scheduling type

OpenMP divides the iterations
into chunks of size chunk-size.

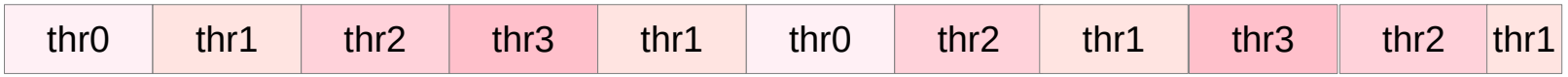
Each thread *executes*
a chunk of iterations and
then *requests* another chunk
until there are no more chunks available.

There is no particular order
in which the chunks are distributed
to the threads.

The order changes each time
when we execute the for loop.

If we do not specify **chunk-size**,
it defaults to one.

Dynamic



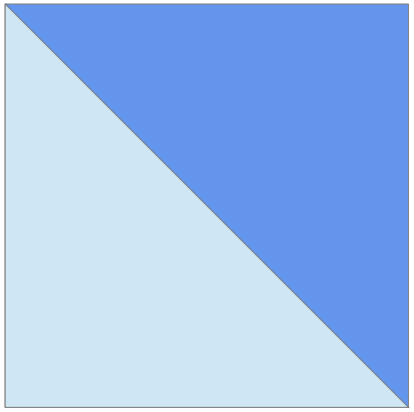
http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Dynamic (4)

Dynamic scheduling is used
when you know that each thread
will not do same amount of work
by using **static scheduling**.

```
float A[100][100];

for(int i = 0; i < 100; i++)
{
   for(int j = 0; j < i; j++)
   {
      A[i][j] = 1.0f;
   }
}
```

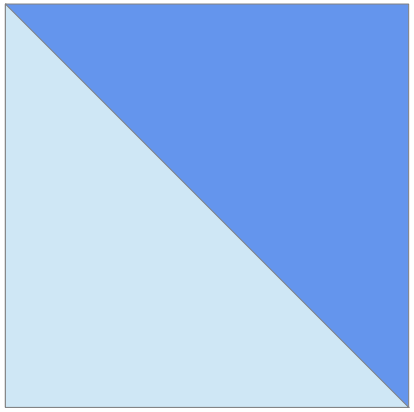


https://stackoverflow.com/questions/15508128/using-omp-schedule-with-pragma-omp-for-parallel-scheduleruntime

Loop Schedule – Dynamic (5)

```
float A[100][100];

for(int i = 0; i < 100; i++)
{
   for(int j = 0; j < i; j++)
   {
      A[i][j] = 1.0f;
   }
}
```



The inner loop variable **j**
is dependent on the **i**.

the **default static scheduling**,

- the outer loop (i) work might be divided equally between the 4 threads,
- but the inner loop (j) work will be large for some threads.

- not equal amount of work
- not optimal **load balancing**

https://stackoverflow.com/questions/15508128/using-omp-schedule-with-pragma-omp-for-parallel-scheduleruntime

Loop Schedule – Dynamic (6)

the dynamic scheduling

this scheduling is done at the run time
can make sure optimal **load balance**.

Note:
you can also specify the chunk_size for
scheduling. It depends on the **loop size**.

https://stackoverflow.com/questions/15508128/using-omp-schedule-with-pragma-omp-for-parallel-scheduleruntime

Loop Schedule – Dynamic (7)

for **schedule(dynamic)** and **schedule(dynamic, 1)**
OpenMP determines similar scheduling.
the size of chunks is equal to **1** in both instances.
the distribution of chunks between the threads is arbitrary.

For **schedule(dynamic, 4)** and **schedule(dynamic, 8)**
OpenMP divides iterations into chunks of size **4** and **8**, respectively.
the distribution of chunks to the threads has no pattern (abitrary)

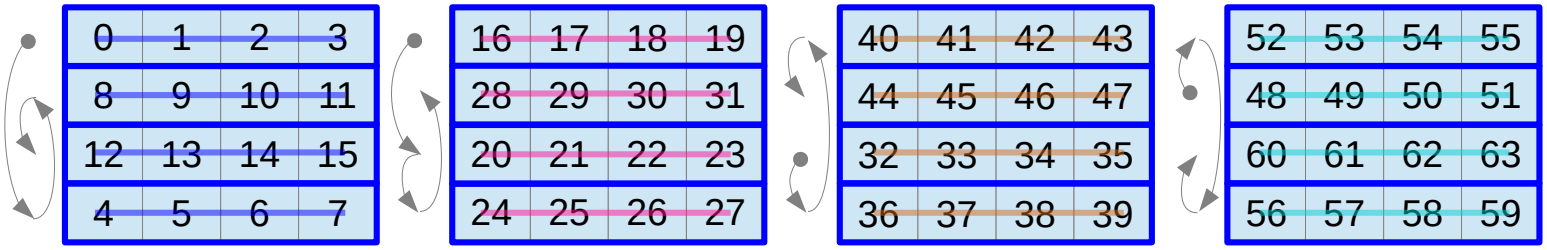
The dynamic scheduling type is appropriate
when the iterations require different computational costs.
I.e, when the iterations are poorly balanced between each other.

the dynamic scheduling type has higher **overhead**
than the static scheduling type
because it dynamically distributes the iterations during the runtime.

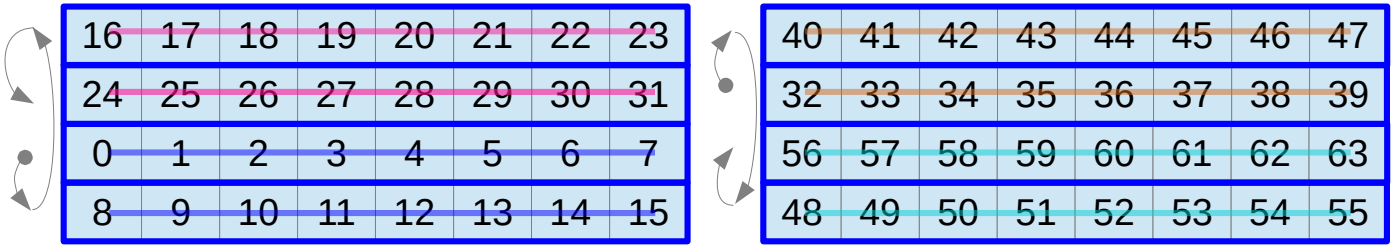
http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Dynamic (6)

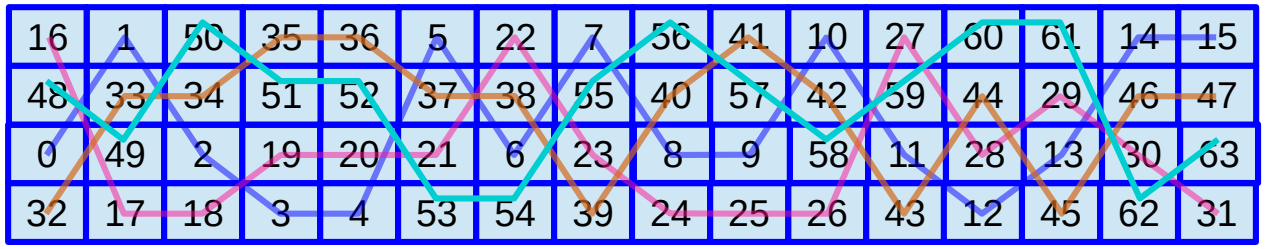
schedule**(dynamic, 4)**



schedule**(dynamic, 8)**



schedule**(dynamic)**



http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Guided (1)

The **guided** scheduling type is
similar to the **dynamic** scheduling type.

OpenMP again divides
the iterations into chunks.

Each thread executes a chunk of iterations
and then requests another chunk
until there are no more chunks available.

The difference is in **the size of chunks**.

The size of a chunk is proportional
to the **number of unassigned iterations**
divided by the **number of the threads**.

Therefore **the size of the chunks**
decreases as the execution goes on

Guided

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Guided (2)

The **minimum size of a chunk**
is set by '**chunk-size'**
in the scheduling clause:

for schedule(guided, chunk-size).

the chunk which contains the last iterations
may have smaller size than chunk-size.

If we do not specify **chunk-size**,
it defaults to **one**.

Guided

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Guided (3)

the **size** of the **chunks** is decreasing.
first chunk has always 16 iterations.
64 iterations and 4 threads → 64 / 4 = 16

the **minimum chunk size** is determined in the schedule clause.
The only exception is the **last chunk**.
Its size might be lower then the prescribed minimum size.

The guided scheduling type is appropriate
when the iterations are poorly balanced between each other.

The initial chunks are larger, because they reduce overhead.
The smaller chunks fills the schedule towards the end of the computation
and improve load balancing.
This scheduling type is especially appropriate
when poor load balancing occurs toward the end of the computation.

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Guided (3)

schedule(guided)

64/4 = 16	64
(64-16) / 4 = 12	48
(48-12) / 4 = 9	36
(36-9) / 4 = 7	27
(27-7) / 4 = 5	20
(20-5) / 4 = 4	15
(15-4) / 4 = 3	11
(11-3) / 4 = 2	8
(8-2) / 4 = 2	6
(6-2) / 4 = 1	4
(4-1) / 4 = 1	3
(3-1) / 4 = 1	2
(2-1) / 4 = 1	1

schedule(guided, 4)

64/4 = 16	64
(64-16) / 4 = 12	48
(48-12) / 4 = 9	36
(36-9) / 4 = 7	27
(27-7) / 4 = 5	20
(20-5) / 4 = 4	15
(11-4) / 4 = 4	7
(7-4) 4	3
3	

schedule(guided, 8)

64/4 = 16		64
(64-16) / 4 = 12		48
(48-12) / 4 = 9		36
(28-8)	8	20
(20-8)	8	12
(12-8)	8	4
4		

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Static (6)

schedule(**static)**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

schedule(guided)

64/4 = 16 64
(64-16) / 4 = 12 48
(48-12) / 4 = 9 36
(36-9) / 4 = 7 27
(27-7) / 4 = 5 20
(20-5) / 4 = 4 15
(15-4) / 4 = 3 11
(11-3) / 4 = 2 8
(6-2) / 4 = 1 4
(4-1) / 4 = 1 3
(3-1) / 4 = 1 2
(2-1) / 4 = 1 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	44	45	46	47	48	58	59	63
16	17	18	19	20	21	22	23	24	25	26	27	49	50	51	52	60							
28	29	30	31	32	33	34	35	36	53	54	55	61											
37	38	39	40	41	42	43	56	57	62														

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Auto

The auto scheduling type delegates
the decision of the scheduling
to the **compiler** and/or **runtime** system.

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedule – Runtime

The schedule(runtime) clause tells it
to set the schedule using the environment variable.

The environment variable can be set
to any other scheduling type.

It can be set by

setenv OMP_SCHEDULE “dynamic,5”

https://stackoverflow.com/questions/15508128/using-omp-schedule-with-pragma-omp-for-parallel-scheduleruntime

Loop Schedules – Runtime

The **runtime** scheduling type defers
the decision about the scheduling
until the **runtime**.

different ways of specifying
the scheduling type in this case.

One option is with the environment variable
OMP_SCHEDULE

and the other option is
with the function
omp_set_schedule.

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Loop Schedules – Runtime

If the scheduling-type (in the schedule clause of the loop construct) is equal to runtime then OpenMP determines the scheduling by the internal control variable run-sched-var. We can set this variable by setting the environment variable OMP_SCHEDULE to the desired scheduling type. For example, in bash-like terminals, we can do

$ export OMP_SCHEDULE=sheduling-type

Another way to specify run-sched-var is to set it with omp_set_schedule function.

```
...
omp_set_schedule(sheduling-type);
...
```

http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Nested Parallelism (1)

```
void fun1()
{
   for (int i=0; i<80; i++)
      ...
}

main()
{
  #pragma omp parallel
  {
    #pragma omp for
    for (int i=0; i<100; i++)
       ...

    #pragma omp for
    for (int i=0; i<10; i++)
       fun1();
  }
}
```

the 2nd loop in **main**
can only be distributed to 10 threads

80 loop iterations in **fun1**
which will be called 10 times in **main** loop.

total **800** iterations in **fun1** and the **main** loop

This gives much more parallelism potential
if parallelism can be added in both levels.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Nested Parallelism (2)

```
void fun1()
{
   #pragma omp parallel for
   for (int i=0; i<80; i++)
      ...
}

main
{
   #Pragma omp parallel
   {
      #pragma omp for
      for (int i=0; i<100; i++)
         …

      #pragma omp for
      for (int i=0; i<10; i++)
         fun1();
   }
}
```

may either have insufficient threads for the 1st main loop
as it has larger loop count, or

create exploded number of threads for the 2nd main loop
when OMP_NESTED=TRUE.

The simple solution is to split the parallel region in main and
create separate ones for each loop
with a distinct thread number specified.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Nested Parallelism (3)

```
void fun1()
{
   #pragma omp taskloop
   for (int I = 0; i<80; i++)
      ...
}

main
{
   #pragma omp parallel
   {
      #pragma omp for
      for (int i=0; i<100; i++)
         ...

      #pragma omp for
      for (int i=0; i<10; i++)
         fun1();
   }
}
```

don't have to worry about the thread number changes
in 1st and 2nd main loops.

Even though you still have a small amount of (10) threads
allocated for 2nd main loop,
the rest available threads will be able
to be distributed through omp **taskloop** in fun1.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Nested Parallelism (4)

nested parallel regions is
a way to distribute **tasks** by creating / forking more **threads**.

parallel region is the only construct
determines **execution thread number**
and controls **thread affinity**

Using **nested parallel regions** means
each **thread** in **parent region**
will yield multiple **threads** in enclosed regions,
which in turn create a product of **thread number**.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Nested Parallelism (5)

omp tasking shows another way
to explore parallelism by adding
more **tasks**, instead of **threads**.

though the **thread number** is unchanged
as specified at the entry of the **parallel region**,
the increased **tasks** from the **nested tasking** constructs
can be distributed and executed by any available/idle **threads**
in the current team of the same parallel region.

This gives opportunities to fully use all threads' capability,
and improve balance of workloads automatically.

https://software.intel.com/content/www/us/en/develop/articles/exploit-nested-parallelism-with-openmp-tasking-model.html

Implicit task (1)

In addition to **explicit tasks** specified using the **task** directive,
the OpenMP specification version 3.0 introduces
the notion of **implicit tasks**.

An **implicit task** is a task generated
- by the **implicit parallel region**,
- when a **parallel construct** is encountered during execution.

The **code** for each **implicit task** is
the code inside the **parallel construct**.

Each **implicit task** is
- assigned to a different **thread** in the **team** and is **tied**;
- always executed from beginning to end
by the **thread** to which it is initially assigned.

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Implicit task (2)

All **implicit tasks** generated
when a **parallel construct** is encountered
are guaranteed to be complete
when the **master thread** exits the **implicit barrier**
at the end of the **parallel region**.

all **explicit tasks** generated within a **parallel region**
are guaranteed to be complete
on exit from the next **implicit** or **explicit barrier**
within the **parallel region**.

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Implicit task (3)

When an **if clause** is present on a **task construct**
and the value of the scalar-expression evaluates to false,
the thread that encounters the task must immediately execute the task.

The **if clause** can be used to avoid the **overhead** of
generating many **finely grained tasks** and
placing them in the **conceptual pool**.

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

Implicit barrier

Implicit BarriersSeveral OpenMP* constructs have implicit barriers

- parallel
- for
- single

Unnecessary barriers hurt performance

- Waiting threads accomplish no work!

Waiting threads accomplish no work!
Suppress implicit barriers, when safe, with the nowait

https://www.intel.com/content/dam/www/public/apac/xa/en/pdfs/ssg/Programming_with_OpenMP-Linux.pdf

References

[1] en.wikipedia.org
[2] M Harris, http://beowulf.lcs.mit.edu/18.337-2008/lectslides/scan.pdf