IN3200/IN4200: Chapter 6
Shared-memory parallel programming with OpenMP

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Objectives

- A very brief introduction to **OpenMP**
  - An application programming interface (API) based mostly on a set of compiler directives
  - Today's most widely used approach for shared-memory parallel programming
- Basic OpenMP programming through examples

(More advanced OpenMP programming will be taught in later chapters.)

First things first

The applicable hardware context: **shared memory**

- All processors can directly access all data in a shared memory, no need for explicit communication between the processors
- OpenMP: A parallel programming standard for shared-memory parallel computers
  - A set of compiler directives (with additional clauses)
  - A small number of library functions
  - A few environment variables

Advantages of compiler directives

- An OpenMP compiler directive is a "parallelization hint" intended for a compatible compiler to automatically create parallel code, will be ignored by non-OpenMP compilers otherwise.

  `#pragma omp ...`
- It is **possible** to maintain a same code for both the serial implementation and the parallel OpenMP implementation.
- Allows incremental parallelization—simplifies coding effort—easier to debug

Threads in OpenMP

- The central execution entities in an OpenMP program are **threads**—lightweight processes.
- The OpenMP threads share a common address space and mutually access data.
- Spawning a thread is much less costly than forking a new process, because threads share everything except the instruction pointer, stack pointer and register state.
  - If wanted, each thread can have a few "private variables" (by means of the local stack pointer).

"Fork-join" model

- In any OpenMP program, a single thread, called **master thread**, runs immediately after startup.
- The master thread can spawn (also called **fork**) a number of additional threads when entering a so-called **parallel region**.
- Inside a parallel region, the master thread and the spawned threads execute instruction streams **concurrently**.
  - Each thread has a unique ID.
  - Different threads work on different parts of the shared data or carry out different tasks.
  - OpenMP has compiler directives for dividing the work among the threads.
- At the end of a parallel region, the threads are **joined**: all the threads are terminated but the master thread.
- There can be multiple parallel regions in an OpenMP program.

OpenMP's parallel execution model

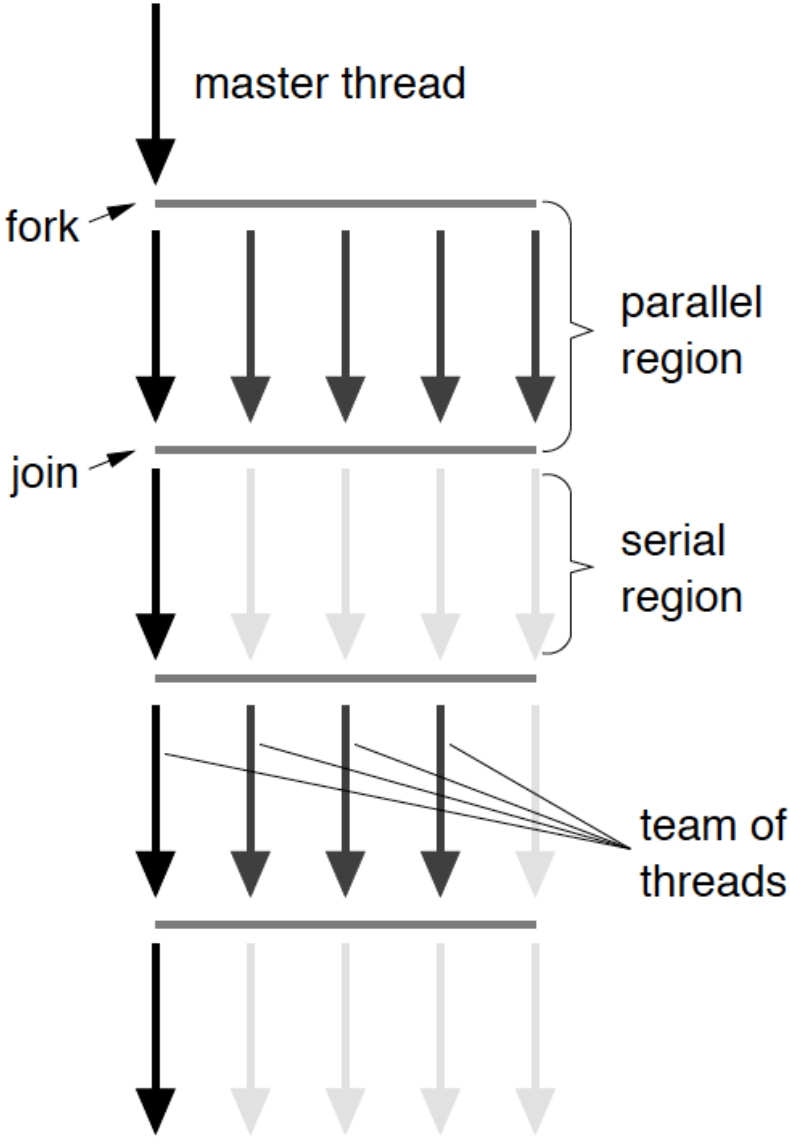


Figure 6.1: Model for OpenMP thread operations: The master thread "forks" team of threads, which work on shared memory in a parallel region. After the parallel region, the threads are "joined," i.e., terminated or put to sleep, until the next parallel region starts. The number of running threads may vary among parallel regions.

Hello world in OpenMP

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

int main (int nargs, char **args) {

#pragma omp parallel
{
  printf("Hello world!\n");
}

  return 0;
}
```

- Example of compilation: `gcc -fopenmp hello_world.c`
- Parallel execution: `./a.out`
- What do you get?

Control the number of threads at runtime

Example on a Linux system:

```
gcc -fopenmp hello_world.c
export OMP_NUM_THREADS=6
./a.out
export OMP_NUM_THREADS=8
./a.out
```

`OMP_NUM_THREADS` is an environment variable (understood by OpenMP)

It is also possible to hard-code the number of threads inside an OpenMP program:

```
#pragma omp parallel num_threads(6)
{
  // ......
}
```

But this approach is normally **not** recommended!

Hello world in OpenMP (a bit more interesting example)

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

int main (int nargs, char **args) {

  printf("I'm the master thread, I'm alone.\n");

#pragma omp parallel
{
  int num_threads, thread_id;
  num_threads = omp_get_num_threads();
  thread_id = omp_get_thread_num();
  printf("Hello world! I'm thread No.%d out of %d threads.\n",
         thread_id,num_threads);
}

  return 0;
}
```

“Manual” loop parallelization

Now, let’s try to “manually” parallelize a for-loop, that is, divide the iterations evenly among the threads.

Example:

```
for (i=0; i<N; i++)
  a[i] = b[i] + c[i];
```

Important observation: Assuming that the arrays `a`, `b` and `c` do not overlap (that is, no aliasing), then the iterations of this for-loop are independent of each other, thus safe to be executed by multiple threads concurrently.

"Manual" loop parallelization (2)

How to divide the iterations `evenly` among the threads?

Given `num_threads` as the total number of threads, **one** way to divide N iterations for thread with ID `thread_id` is as follows:

```
int blen, bstart;

blen = N/num_threads;

if (thread_id < (N%num_threads)) {
  blen = blen + 1;
  bstart = blen * thread_id;
}
else {
  bstart = blen * thread_id + (N%num_threads);
}
```

Why is this a fair division?

“Manual” loop parallelization (3)

OpenMP coding

```
#pragma omp parallel
{
  int num_threads, thread_id;
  int blen, bstart, bend, i;

  num_threads = omp_get_num_threads();
  thread_id = omp_get_thread_num();

  blen = N/num_threads;
  if (thread_id < (N%num_threads)) {
    blen = blen + 1;
    bstart = blen * thread_id;
  }
  else {
    bstart = blen * thread_id + (N%num_threads);
  }
  bend = bstart + blen;

  for (i=bstart; i<bend; i++)
    a[i] = b[i] + c[i];

} // end of parallel region
```

Data scoping

Any variables that existed before a parallel region still exist inside the parallel region, and are by default **shared** between all threads.

Often it will be necessary for the threads to have some *private* variables.

- Each thread can either declare new local variables inside the parallel region, these variables are private "by birth";
- Or, each thread can "privatize" some of the shared variables that already existed before a parallel region (using the `private` clause):

```
int blen, bstart, bend;
#pragma omp parallel private(blen, bstart, bend)
{
  // ...
}
```

- Each "privatized" variable has one (*uninitialized*) instance per thread;
- The private variables' scope is until the end of the parallel region.

Actually, parallelizing a for-loop is easy in OpenMP

- Parallelizing for-loops is OpenMP's main work-sharing mechanism.
- OpenMP has several built-in strategies for dividing the iterations among the threads.
  - No need to manually calculate each thread's loop bounds

```
#pragma omp parallel
{
#pragma omp for
  for (i=0; i<N; i++)
    a[i] = b[i] + c[i];
} // end of parallel region
```

or simply

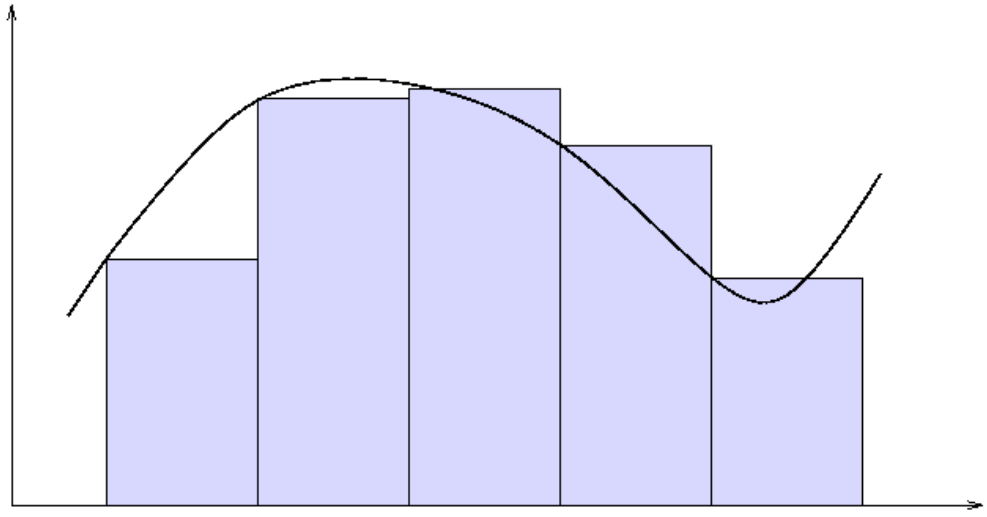
```
#pragma omp parallel for
  for (i=0; i<N; i++)
    a[i] = b[i] + c[i];
```

More remarks

- The loop can not contain `break`, `return`, `exit` statements.
- The `continue` statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and changes to it inside the loop are not allowed.

Numerical integration

How to numerically calculate $\int_{x_0}^{x_1} f(x)dx$?



$$w = \frac{x_1 - x_0}{N}$$

$$\int_{x_0}^{x_1} f(x)dx \approx w\left(f(x_0 + \frac{w}{2}) + f(x_0 + w + \frac{w}{2}) + \ldots \right.$$
$$\left. \ldots + f(x_0 + (N-1)w + \frac{w}{2})\right)$$

Numerical integration for calculating π

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

Serial implementation:

```
int N, i;
double w = 1.0/N, x, approx_pi;
double sum = 0.;

for (i=1; i<=N; i++) {
  x = w*(i-0.5);
  sum = sum + 4.0/(1.0+x*x);
}

approx_pi = w*sim;
```

A naive OpenMP implementation

```
int N, i;
double w = 1.0/N, x, approx_pi = 0.;
double sum = 0.;

#pragma omp parallel private(x) firstprivate(sum)
{

#pragma omp for
for (i=1; i<=N; i++) {
  x = w*(i-0.5);
  sum = sum + 4.0/(1.0+x*x);
}

#pragma omp critial
{
approx_pi = approx_pi + w*sim;
}

} // end of the parallel region
```

OpenMP critical regions

- Concurrent write accesses to a shared variable must be avoided by all means to circumvent **race conditions**.
- An OpenMP `critical` code block is executed by **one thread at a time**. This is one way to avoid race conditions. (There are other ways.)
- The variable `approx_pi` in the above example is a shared variable, to which all the threads will write. Thus, "protection" is provided by a `critical` code block.
- Use of the `critical` directive will incur overhead.
- Improper use of the `critical` directive may lead to **deadlock**.

Use of OpenMP's `reduction` clause

Actually, using `critical` to prevent concurrent writes to the variable `approx_pi` is an "over-kill". The `reduction` clause of OpenMP is designed for this particular purpose:

```
int N, i;
double sum = 0.;
double w = 1.0/N, x, approx_pi;

#pragma omp parallel for private(x) reduction(+:sum)
for (i=1; i<=N; i++) {
  x = w*(i-0.5);
  sum = sum + 4.0/(1.0+x*x);
}

approx_pi = w*sim;
```

Another example of using the `reduction` clause

$$s = a_0^2 + a_1^2 + \ldots + a_{N-1}^2$$

```
#pragma omp parallel for reduction(+:s)
for (i=0; i<N; i++)
  s = s + a[i]*a[i];
```

Loop scheduling

```
#pragma omp parallel for
for (i=0; i<N; i++)
  a[i] = b[i] + c[i];
```

How are the loop iterations exactly divided among the threads?

- Mapping of loop iterations to threads is configurable in OpenMP.
- The "secret" is the `schedule` clause:

  ```
  #pragma omp parallel for schedule(static|dynamic|guided [,chunk])
  ```

- Default scheduling is `static` (no need to specify), which divides the iterations into contiguous chunks of (roughly) equal size.
- Other alternatives of scheduling: `dynamic` and `guided`
- It is possible to prescribe a chunksize in the `schedule` clause.

  ```
  #pragma omp parallel for schedule(dynamic,3)
  ```

Examples of different schedulers

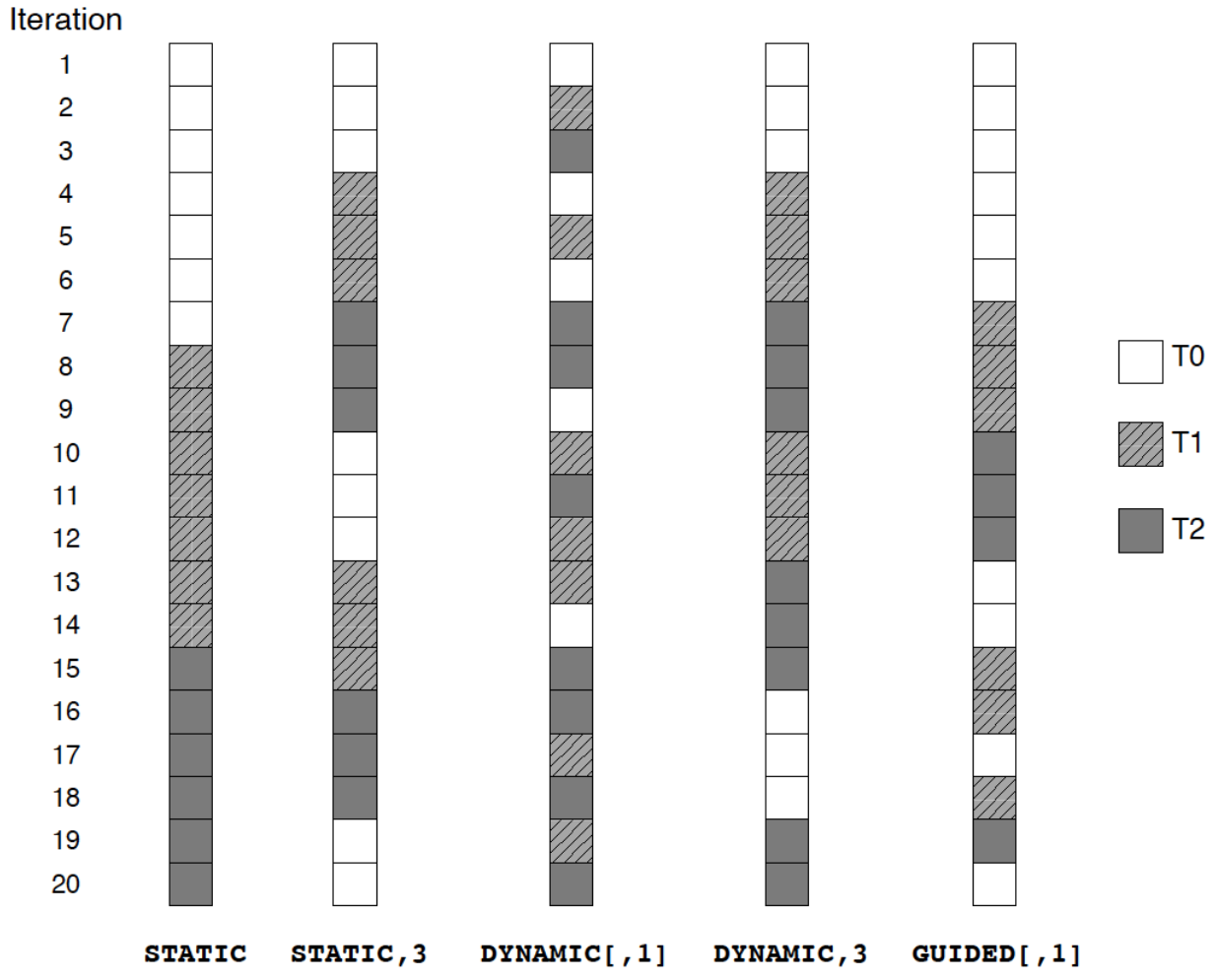


Figure 6.2: Loop schedules in OpenMP. The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for `DYNAMIC` and `GUIDED` is one. If a chunksize is specified, the last chunk may be shorter. Note that only the `STATIC` schedules guarantee that the distribution of chunks among threads stays the same from run to run.

Tasking

- A task can be defined by OpenMP's `task` directive, containing the code to be executed.
- When a thread encounters a task construct, it may execute it right away or set up the appropriate data environment and defer its execution. The task is then ready to be executed later by any thread of the team.

An example of OpenMP tasks

```
#pragma omp parallel private(r, i)
{

#pragma single
{
  for (i=0; i<N; i++) {
    r = rand();  // a randomly generated number
    if (p[i] > r) {
      #pragma task
      {
      do_some_work (p[i]);
      }
    } // enf of if-test
  } // end of for-loop
} // end of the single directive

} // end of the parallel region
```

The actual number of calls to `do_some_work` is unknown, so tasking is a natural choice for work division.

`single` and `master`

A "`single`" code block in OpenMP will be entered by one thread only, namely the therad that reaches the `single` directive first. All others skip the code and wait at the end of the `single` block due to an implicit barrier.

A "`master`" code block is only entered by the master thread, all the other threads skip over without waiting for the master thread to finish.

OpenMP has a separate "`barrier`" directive for explicit synchronization among the threads. (Use with care!!!)

Yet another example

```
int myid, numthreads;

#pragma omp parallel private(myid)
{
  my_id = omp_get_thread_num();

#pragma omp single
{
  numthreads = omp_get_num_threads();
}

#pragma omp critical   // not strictly necessary
{
  printf("This is thread No.%d out of %d threads\n",
         my_id, numthreads);
}

} // end of the parallel region
```

Jacobi algorithm

Serial C implementation (slightly different from that of Chapter 3):

```
double maxdelta = 1.0, eps = 1.0e-14;

while (maxdelta > eps) {
  maxdelta = 0.;

  for (k=1; k<kmax-1; k++)
    for (i=1; i<imax-1; i++) {
      phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]
                      +phi[k][i+1]+phi[k+1][i])*0.25;
      maxdelta = max(maxdelta, abs(phi_new[k][i]-phi[k][i]));
  }

  /* pointer swapping */
  temp_ptr = phi_new;
  phi_new = phi;
  phi = temp_ptr;
}
```

OpenMP-parallel Jacobi algorithm

```
double maxdelta = 1.0, eps = 1.0e-14;

while (maxdelta > eps) {
  maxdelta = 0.;

  #pragma omp parallel for reduction(max: maxdelta) private(i)
  {
  for (k=1; k<kmax-1; k++)
    for (i=1; i<imax-1; i++) {
      phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]
                      +phi[k][i+1]+phi[k+1][i])*0.25;
      maxdelta = max(maxdelta, abs(phi_new[k][i]-phi[k][i]));
    }
  } // end of the parallel region

  /* pointer swapping */
  temp_ptr = phi_new;
  phi_new = phi;
  phi = temp_ptr;
}
```

OpenMP-parallel Jacobi algorithm (version 2)

```
double maxdelta = 1.0, eps = 1.0e-14;

#pragma omp parallel
{
while (maxdelta > eps) {
#pragma omp barrier  // preventing maxdelta from being zeroed too earlier
#pragma omp single
{
  maxdelta = 0.;
}

#pragma omp for reduction(max: maxdelta) private(i)
{
  for (k=1; k<kmax-1; k++)
    for (i=1; i<imax-1; i++) {
      phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]
                      +phi[k][i+1]+phi[k+1][i])*0.25;
      maxdelta = max(maxdelta, abs(phi_new[k][i]-phi[k][i]));
  }
}

#pragma omp master
{
  /* pointer swapping */
  temp_ptr = phi_new;
  phi_new = phi;
  phi = temp_ptr;
}
} // end of while loop

} // end of the entire parallel region
```

Challenge: Parallelizing 3D Gauss-Seidel algorithm

What if the iterations of a triple loop nest are not entirely independent? (There are loop-carried independences.)

Example: 3D Gauss-Seidel algorithm (computational core)

```
for (k=1; k<kmax-1; k++)
  for (j=1; j<jmax-1; j++)
    for (i=1; i<imax-1; i++)
      phi[k][j][i] = (phi[k-1][j][i] + phi[k][j-1][i]
                     +phi[k][j][i-1] + phi[k][j][i+1]
                     +phi[k][j+1][i] + phi[k+1][j][i])/6.0;
```

We cannot just add `#pragma omp parallel for` before the `k`-indexed loop.

Note: The upper limits of `k`, `j` and `i` are different from those given in Chapter 6 of the textbook.

Wavefront parallelization

Although not as simple as the Jacobi algorithm, it is still possible to parallelize the Gauss-Seidel algorithm with OpenMP.

The key idea is to find a way of traversing the 3D lattice that fulfills the dependency constraints imposed by the stencil update.

A *wavefront* travels in the k direction. The dimension along which to parallelize is j. Each of the threads, T_0, T_1, ..., T_{t-1}, is assinged a consecutive chuck of the `j` indices.

Wavefront parallelization (2)

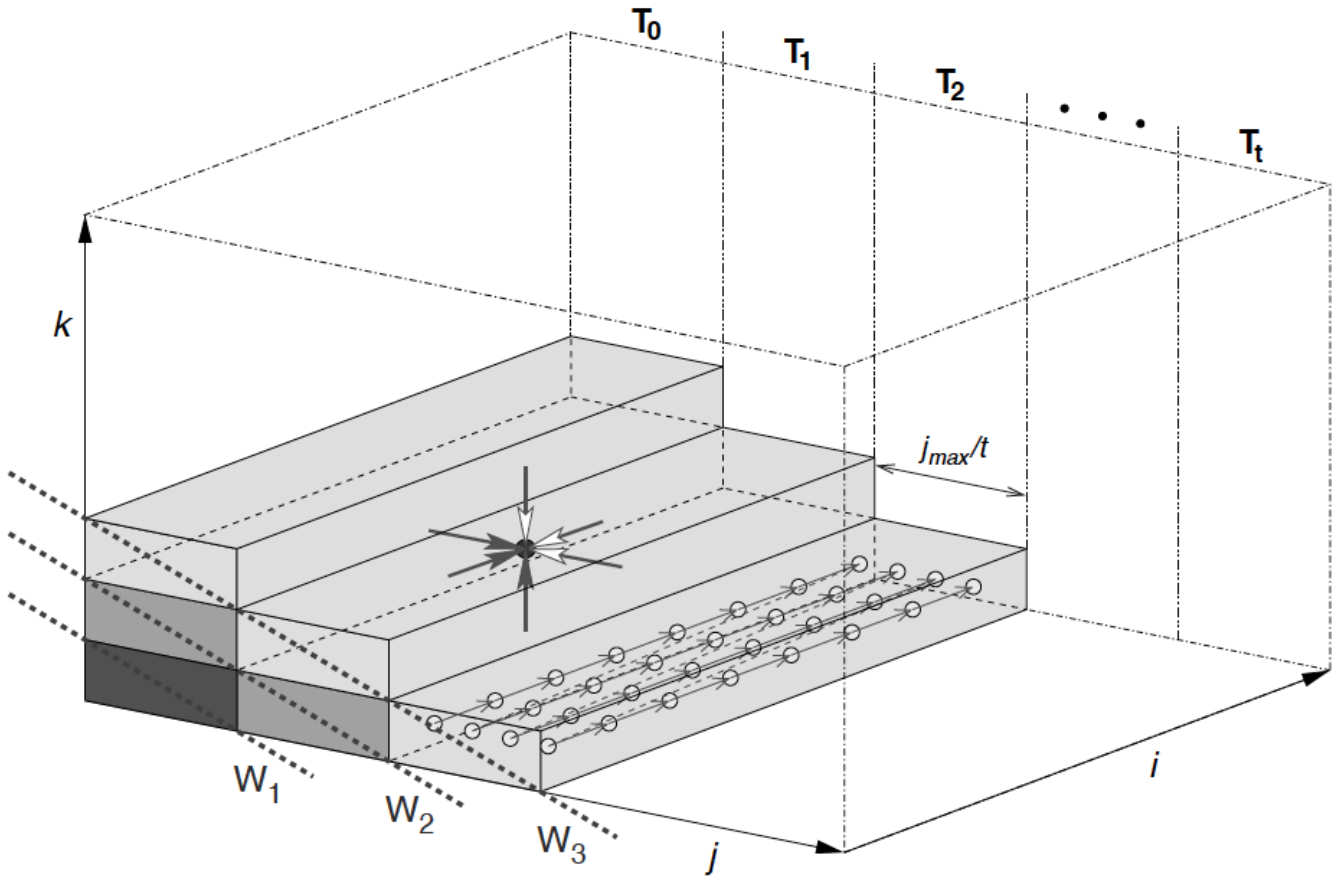


Figure 6.4: Pipeline parallel processing (PPP), a.k.a. wavefront parallelization, for the Gauss–Seidel algorithm in 3D (wind-up phase). In order to fulfill the dependency constraints of each stencil update, successive wavefronts (W_1,W_2,...,W_n) must be performed consecutively, but multiple threads can work in parallel on each individual wavefront. Up until the end of the wind-up phase, only a subset of all t threads can participate.

Wavefront parallelization (3)

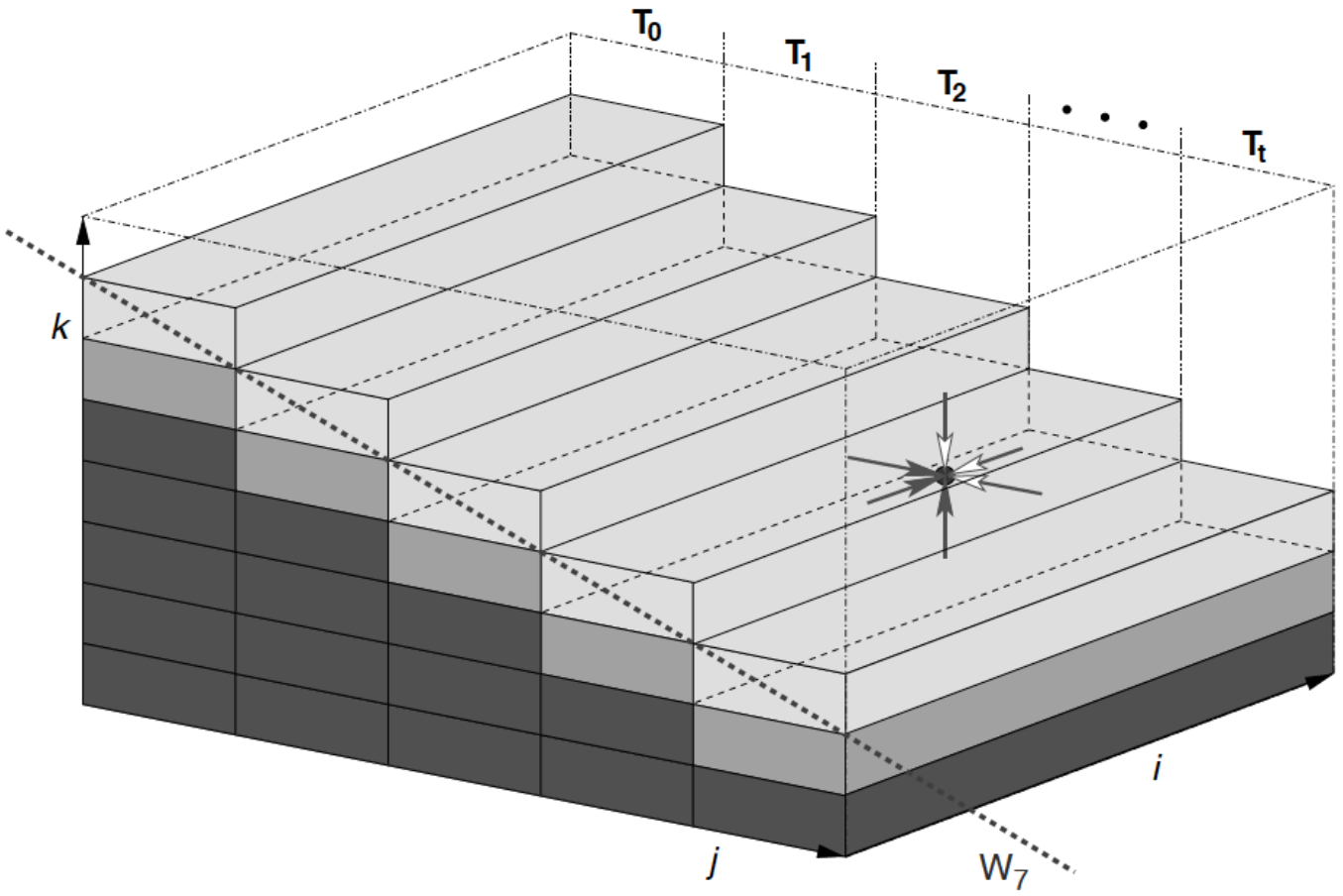


Figure 6.5: Wavefront parallelization for the Gauss–Seidel algorithm in 3D (full pipeline phase). All t threads participate. Wavefront W_7 is shown as an example.

Important observations

- The `k` index goes between `1` and `kmax-2`.
- All the `j` indices `1,2,...jmax-2` are divided evenly into consecutive chucks: $J_0, J_1, \ldots, J_{t-1}$ (one chunk per thread).
- Total number of wavefronts: `(kmax-2)`$+t-1$, for computing through the entire 3D lattice
  - Wavefront W_1 has only one block (`k`=1, J_0)
  - Wavefront W_2 has two concurrent blocks (`k`=1, J_1) and (`k`=2, J_0)
  - Wavefront W_3 has three concurrent blocks (`k`=1, J_2), (`k`=2, J_1) and (`k`=3, J_0)
  - $\ldots$
  - For wavefronts $W_t, W_{t+1}, \ldots, W_{\texttt{kmax-2}}$, each has t concurrent blocks
  - Wavefronts $W_{\texttt{kmax-1}}, \ldots, W_{\texttt{kmax-2}+t-1}$ have fewer and fewer concurrent blocks (the wind-down phase).

OpenMP wavefront parallelization

```
#pragma omp parallel private(k,j,i)
{
  int numthreads, threadID, jstart, jend, m;

  numthreads = omp_get_num_threads();
  threadID = omp_get_thread_num();
  jstart = ((jmax-2)*threadID)/numthreads + 1;
  jend = ((jmax-2)*(threadID+1))/numthreads;

  for (m=1; m<=kmax+numthreads-3; m++) {  // the wavefronts
    k = m - threadID;
    if (k>=1 && k<=kmax-2) {
      for (j=jstart; j<=jend; j++)
        for (i=1; i<imax-1; i++)
          phi[k][j][i] = (phi[k-1][j][i] + phi[k][j-1][i]
                         +phi[k][j][i-1] + phi[k][j][i+1]
                         +phi[k][j+1][i] + phi[k+1][j][i])/6.0;
    }
    #pragma omp barrier
  }
} // end of the parallel region
```