Scheduling in OpenMP

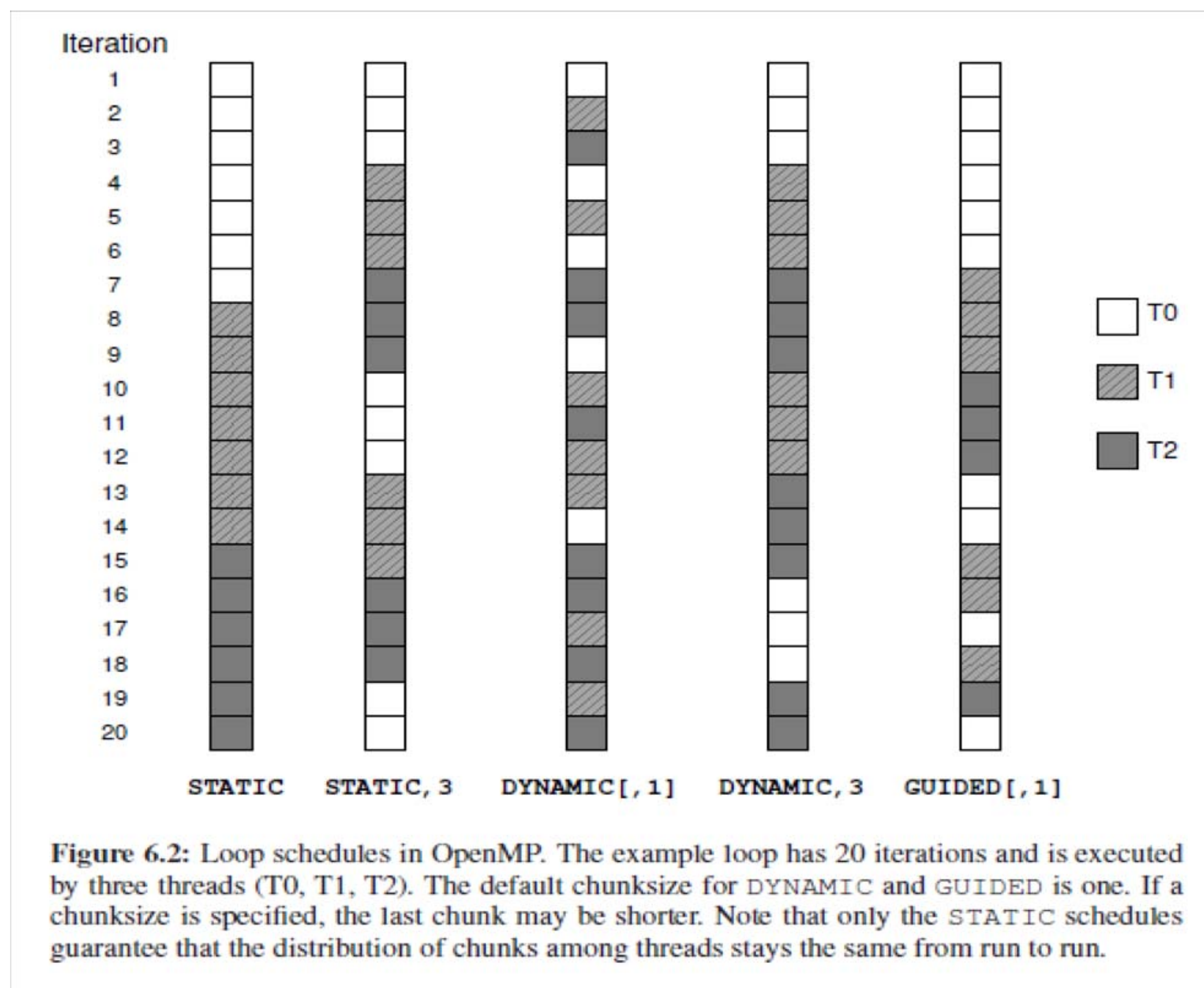


Figure 6.2: Loop schedules in OpenMP. The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for `DYNAMIC` and `GUIDED` is one. If a chunksize is specified, the last chunk may be shorter. Note that only the `STATIC` schedules guarantee that the distribution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

Tasking in OpenMP

```
1    integer i,N=1000000
2    type(object), dimension(N) :: p
3    double precision :: r
4    ...
5  !$OMP PARALLEL PRIVATE(r,i)
6  !$OMP SINGLE
7    do i=1,N
8      call RANDOM_NUMBER(r)
9      if(p(i)%weight > r) then
10 !$OMP TASK
11         ! i is automatically firstprivate
12         ! p() is shared
13         call do_work_with(p(i))
14 !$OMP END TASK
15     endif
16   enddo
17 !$OMP END SINGLE
18 !$OMP END PARALLEL
```

The Wavefront method

First consider a simple two dimensional Fortran 77 form of this method. That is, two dimensional loop nests with a single statement inside that assigns to a two dimensional array.

To illustrate this method we draw a graph of the iteration space of the loop. Each iteration will be a node in the graph. The graph will taked the form of a mesh with equal vertical and horizontal separation.

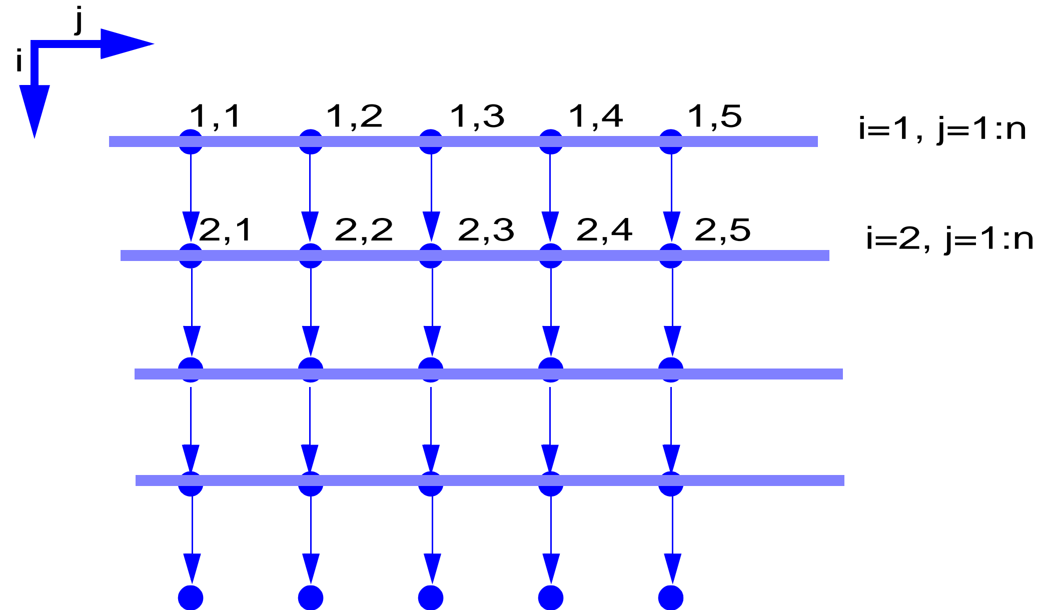
These nodes will be joined by three classes of arcs representing races (write-read, read-write, write-write). These arcs (which are called **dependences**) will always flow in the direction of execution in the original loop.

The idea is that a vector form can be obtained by finding a collection of parallel lines that are equidistant, are not parallel to any dependence arc, and pass through all the nodes in the graph.

For example, the loop

```
do i=1,n
   do j=1,n
      a(i,j)=a(i-1,j)+1
   end do
end do
```

can be represented by the following graph:

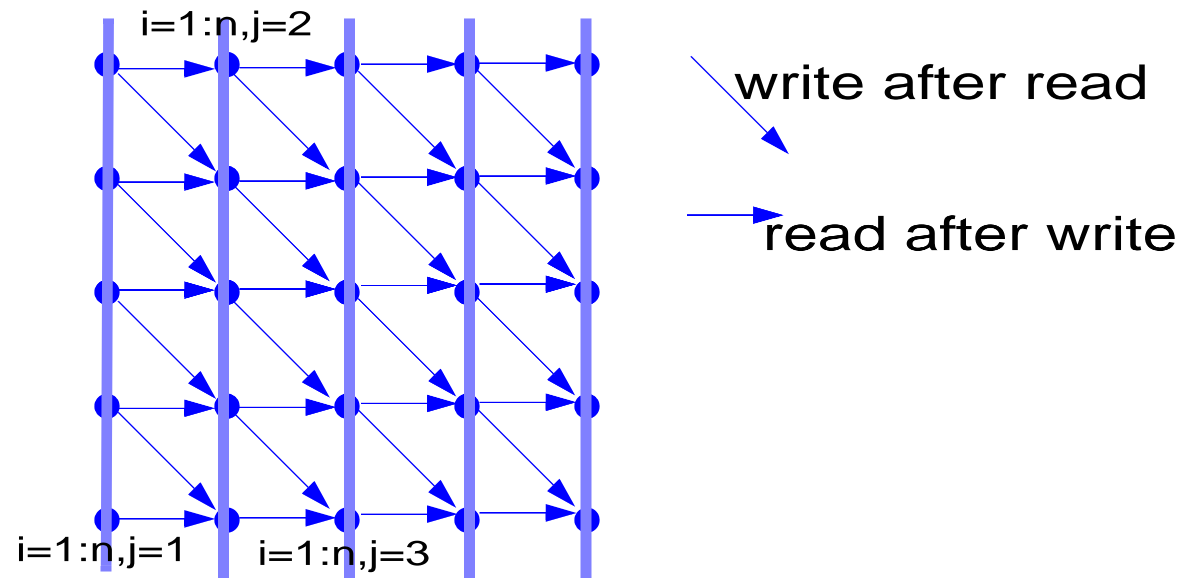


From the graph it is clear that for each i there is a vector operation in j.

```
do i=1,n
   a(i,1:n)=a(i-1,1:n)+1
end do
```

A second example:

```
do i=1,n
   do j=1,n
      a(i,j)=a(i,j-1)+a(i+1,j+1)+b(i)+c(j)
   end do
end do
```

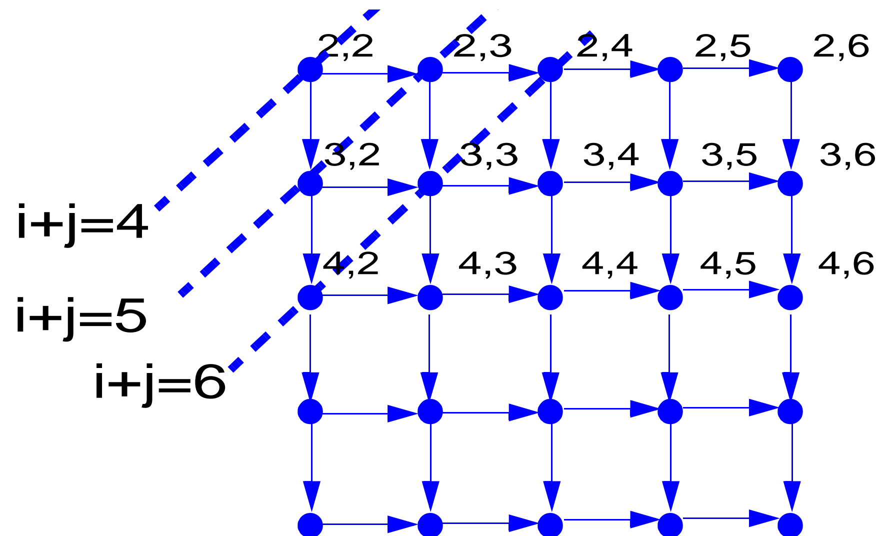


Now for each j there is a vector operation

```
do j=1,n
   a(1:n,j)=a(1:n,j-1)+a(2:n+1,j+1)+b(i)+c(1:n)
end do
```

A more complicated case:

```
do i=2,n
   do j=2,n
      a(i,j)=a(i,j-1)+a(i-1,j)
   end do
end do
```



From the equations:

$$2 \le i \le n$$
$$2 \le k - i \le n$$
$$k = 4, 5, \ldots, 2n$$

We conclude that:

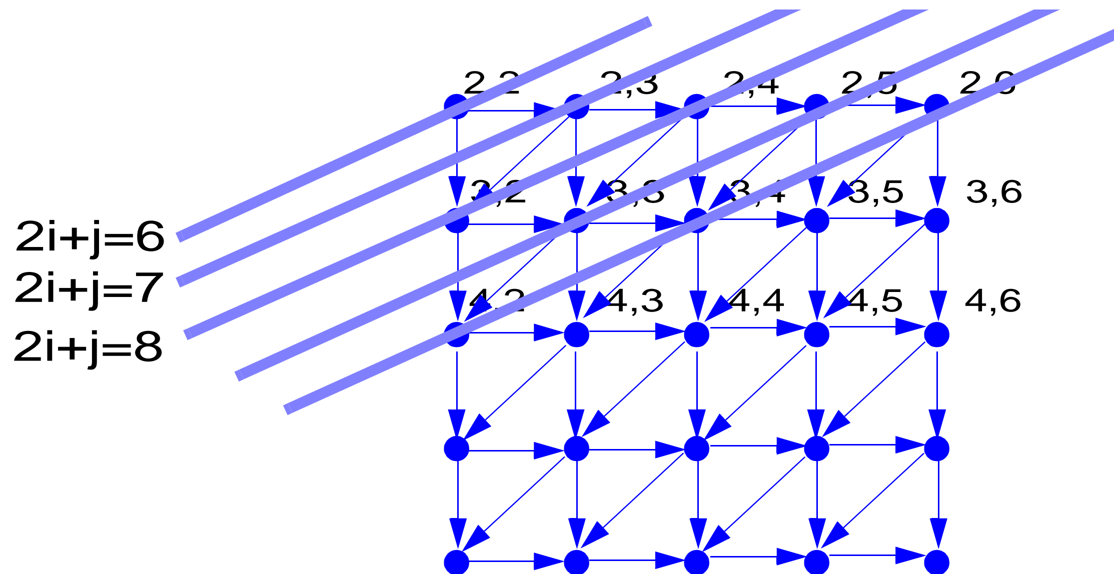
$$max(2, k-n) \le i \le min(n, k-2)$$

From where:

```
do k=4,2*n
   forall (i=max(2,k-n):min(n,k-2)) a(i,k-j)=...
end do
```

Another complex example:

```
do i=2,n
   do j=2,n
      a(i,j)=a(i+1,j-1)+a(i-1,j)+a(i,j-1)
   end do
end do
```



From the equations:

$$4 \le 2i \le 2n$$

$$2 \le k - 2i \le n$$

$$k = 6, 5, \ldots, 3n$$

We conclude that:

$$max\left(2, \left\lceil \frac{k-n}{2} \right\rceil\right) \le i \le min\left(n, \left\lfloor \frac{k-2}{2} \right\rfloor\right)$$

From where:

```
do k=6,3*n
   forall (i=max(2,(k-n+1)/2):min(n,(k-2)/2)) a(i,k-j)=...
end do
```

Listing 6.5: OpenMP implementation of the 2D Jacobi algorithm on an $N \times N$ lattice, with a convergence criterion added.

```
1    double precision, dimension(0:N+1,0:N+1,0:1) :: phi
2    double precision :: maxdelta,eps
3    integer :: t0,t1
4    eps = 1.d-14       ! convergence threshold
5    t0 = 0 ; t1 = 1
6    maxdelta = 2.d0*eps
7    do while(maxdelta.gt.eps)
8      maxdelta = 0.d0
9  !$OMP PARALLEL DO REDUCTION(max:maxdelta)
10     do k = 1,N
11       do i = 1,N
12          ! four flops, one store, four loads
13          phi(i,k,t1) = (  phi(i+1,k,t0) + phi(i-1,k,t0)
14                         + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
15          maxdelta = max(maxdelta,abs(phi(i,k,t1)-phi(i,k,t0)))
16       enddo
17     enddo
18 !$OMP END PARALLEL DO
19     ! swap arrays
20     i = t0 ; t0=t1 ; t1=i
21   enddo
```

Listing 6.6: A straightforward implementation of the Gauss–Seidel algorithm in three dimensions. The highlighted references cause loop-carried dependencies.

```
1  double precision, parameter :: osth=1/6.d0
2  do it=1,itmax   ! number of iterations (sweeps)
3    ! not parallelizable right away
4    do k=1,kmax
5      do j=1,jmax
6        do i=1,imax
7          phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
8                       + phi(i,j-1,k) + phi(i,j+1,k)
9                       + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
10       enddo
11     enddo
12   enddo
13 enddo
```

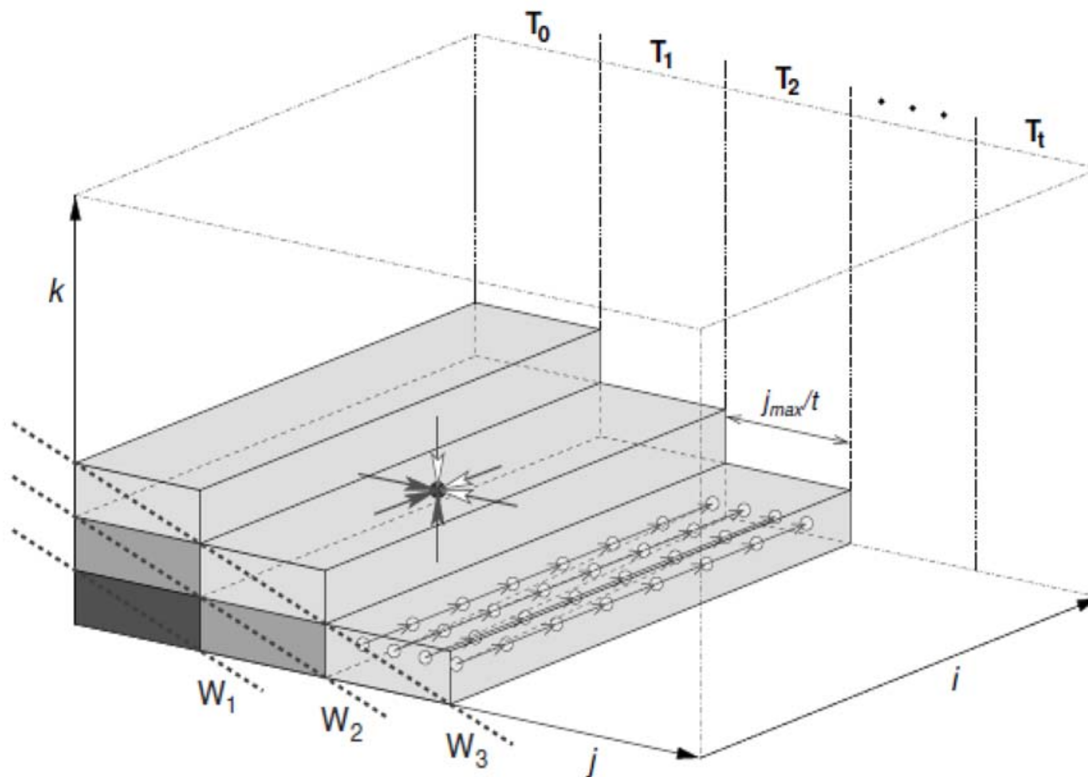


Figure 6.4: Pipeline parallel processing (PPP), a.k.a. wavefront parallelization, for the Gauss–Seidel algorithm in 3D (wind-up phase). In order to fulfill the dependency constraints of each stencil update, successive wavefronts ($W_1, W_2, \ldots, W_n$) must be performed consecutively, but multiple 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.

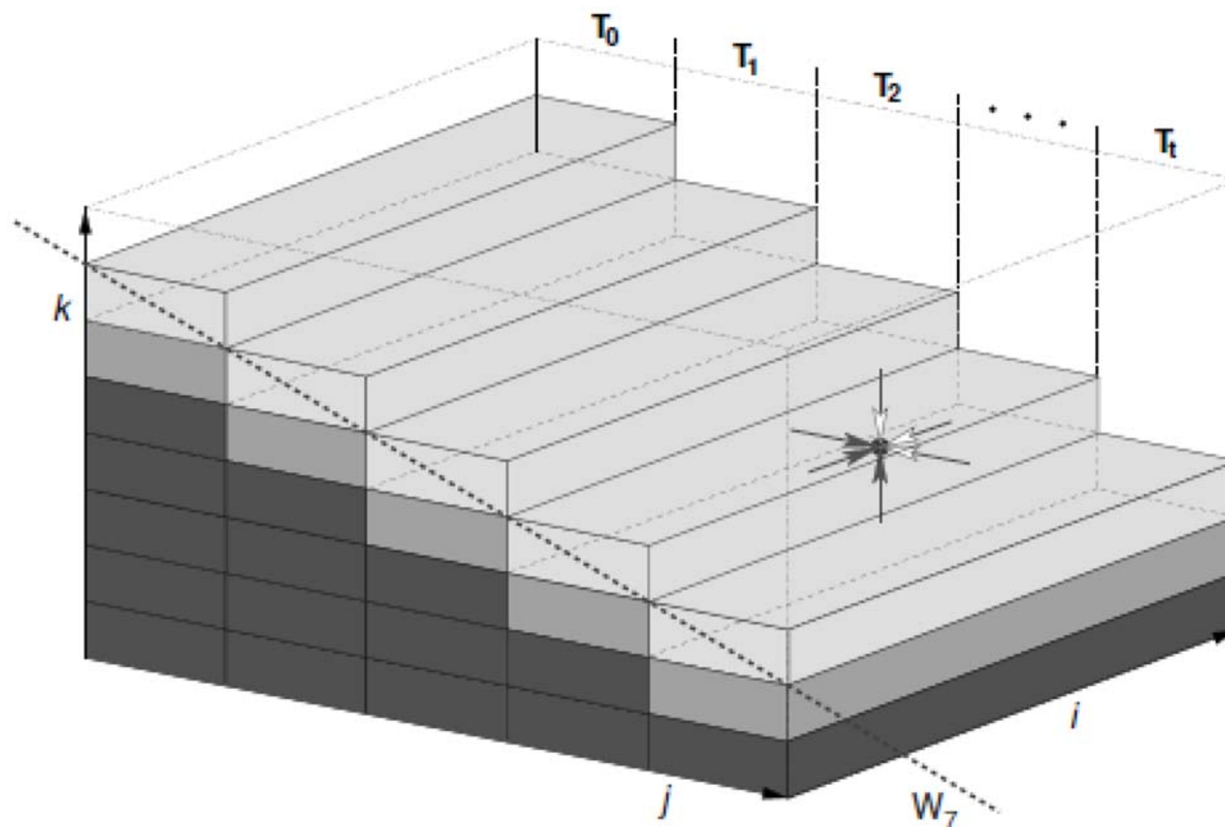


Figure 6.5: Wavefront parallelization for the Gauss–Seidel algorithm in 3D (full pipeline phase). All t threads participate. Wavefront W_7 is shown as an example.

Listing 6.7: The wavefront-parallel Gauss–Seidel algorithm in three dimensions. Loop-carried dependencies are still present, but threads can work in parallel.

```
1  !$OMP PARALLEL PRIVATE(k,j,i,jStart,jEnd,threadID)
2    threadID=OMP_GET_THREAD_NUM()
3  !$OMP SINGLE
4    numThreads=OMP_GET_NUM_THREADS()
5  !$OMP END SINGLE
6    jStart=jmax/numThreads*threadID
7    jEnd=jStart+jmax/numThreads ! jmax is amultiple of numThreads
8    do l=1,kmax+numThreads-1
9      k=l-threadID
10     if((k.ge.1).and.(k.le.kmax)) then
11       do j=jStart,jEnd        ! this is the actual parallel loop
12         do i=1,iMax
13           phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
14                        + phi(i,j-1,k) + phi(i,j+1,k)
15                        + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
16         enddo
17       enddo
18     endif
19 !$OMP BARRIER
20   enddo
21 !$OMP END PARALLEL
```