OpenMP: Guided Scheduling

- **Size of chunks in dynamic schedule**
  - too small → large overhead
  - too large → load imbalance
- **Guided scheduling: dynamically vary chunk size.**
  - Size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to chunk-size. (default: → 1)
- **Chunk size:**
  - means minimum chunk size (except perhaps final chunk)
  - default value is **1**

  - both dynamic and guided scheduling are useful for handling **poorly balanced and unpredictable** workloads.

OpenMP: Dynamic Scheduling Example

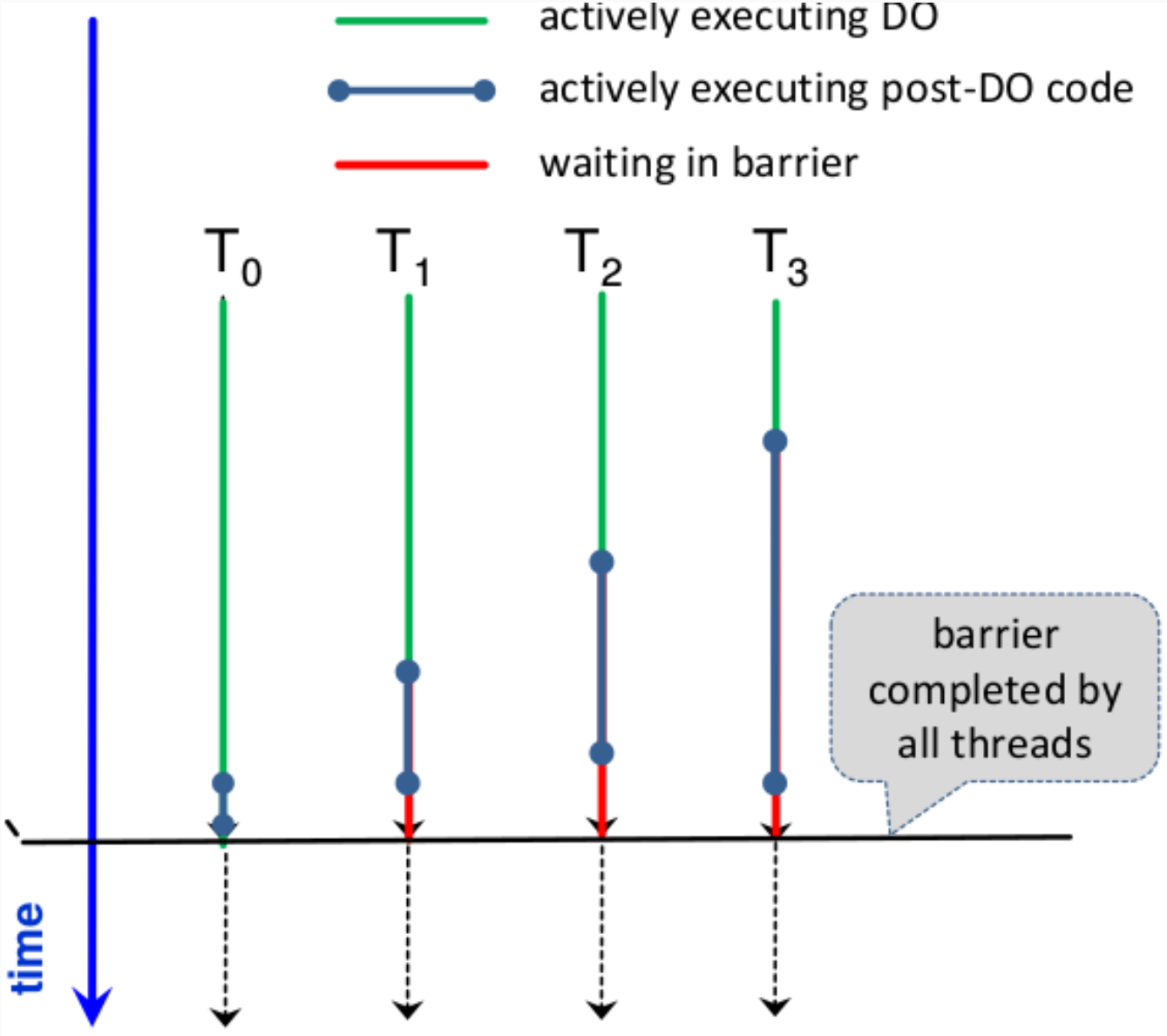
```
1   #include <omp.h>
2   #define N 1000
3   #define CHUNKSIZE 100
4
5   main(int argc, char *argv[]) {
6
7   int i, chunk;
8   float a[N], b[N], c[N];
9
10  /* Some initializations */
11  for (i=0; i < N; i++)
12    a[i] = b[i] = i * 1.0;
13  chunk = CHUNKSIZE;
14
15  #pragma omp parallel shared(a,b,c,chunk) private(i)
16    {
17
18    #pragma omp for schedule(dynamic,chunk) nowait
19    for (i=0; i < N; i++)
20      c[i] = a[i] + b[i];
21
22    }   /* end of parallel region */
23
24  }
```

OpenMP: Nowait clause

```
#pragma omp parallel
#pragma omp for reduction(+: tsum) nowait
for (k=1; k<kmax; k++)
{
   tsum = tsum + foo(a,b,c);
}
some_work();
#pragma omp barrier
```

- nowait clause is used to remove implicit barrier at the end of for loop
- if there is work to do immediately after for loop that does not depend on `tsum`, use of nowait can be useful

OpenMP: nowait clause



OpenMP: Collapse

```
#pragma omp parallel for collapse(2)
for (k=1; k<kmax; k++)
{
    for (j=1; j<jmax; j++)
    {
        some_work()
    }
}
```

- slicing is performed on the virtual index I_{coll}

I_{coll}	0	1	2	3	4	5
J	1	2	3	1	2	3
K	1	1	1	2	2	2

sequenced by serial execution order

OpenMP: Sections worksharing directive

```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {

  #pragma omp section   newline

     structured_block

  #pragma omp section   newline

     structured_block

  }
```

OpenMP: Sections

- The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team
- Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team. Different sections may be executed by different threads. It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such
- There is an implied barrier at the end of a SECTIONS directive

OpenMP: Sections worksharing directive

```
1   #include <omp.h>
2   #define N 1000
3
4   main(int argc, char *argv[]) {
5
6   int i;
7   float a[N], b[N], c[N], d[N];
8
9   /* Some initializations */
10  for (i=0; i < N; i++) {
11    a[i] = i * 1.5;
12    b[i] = i + 22.35;
13    }
14
15  #pragma omp parallel shared(a,b,c,d) private(i)
16    {
17
18    #pragma omp sections nowait
19      {
20
21      #pragma omp section
22      for (i=0; i < N; i++)
23        c[i] = a[i] + b[i];
24
25      #pragma omp section
26      for (i=0; i < N; i++)
27        d[i] = a[i] * b[i];
28
29      }  /* end of sections */
30
31    }  /* end of parallel region */
32
33  }
```

OpenMP: Single Directive

```
#pragma omp single [clause ...]  newline
                   private (list)
                   firstprivate (list)
                   nowait

     structured_block
```

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team
- May be useful when dealing with sections of code that are not thread safe
- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block
- It is illegal to branch into or out of a SINGLE block

OpenMP: Single directive

```
#include <stdio.h>

void work1() {}
void work2() {}

void single_example()
{
  #pragma omp parallel
  {
    #pragma omp single
      printf("Beginning work1.\n");

    work1();

    #pragma omp single
      printf("Finishing work1.\n");

    #pragma omp single nowait
      printf("Finished work1 and beginning work2.\n");

    work2();
  }
}
```

OpenMP: Master construct

C

```
#pragma omp master
    { block }
```

- **Only thread zero (from the current team) executes the enclosed code block**
  - there is **no implied barrier** either on entry to, or exit from, the master construct. Other threads continue **without synchronization**
- **Notes:**
  - Not all threads must reach the construct; if the master thread does not reach it, it will not be executed at all
  - this is not a work sharing construct, it only serves for execution control

OpenMP: firstprivate clause

```
double s;
s = ...
#pragma omp parallel firstprivate(s)
{
    ... = ... + s;
   s = ...
}
 ... = ... + s;
```

- Value of master copy is transferred to private variable
- cant be a pointer, should not have been declared as firstprivate already

OpenMP: lastprivate clause

```
double s;
s = ...
#pragma omp parallel
{
#pragma omp for lastprivate(s)
for(i=1; i<imax; i++)
   s = ...

... = ... + s;
}
```

- value from thread which executes last update in the serial code is transferred back to master copy

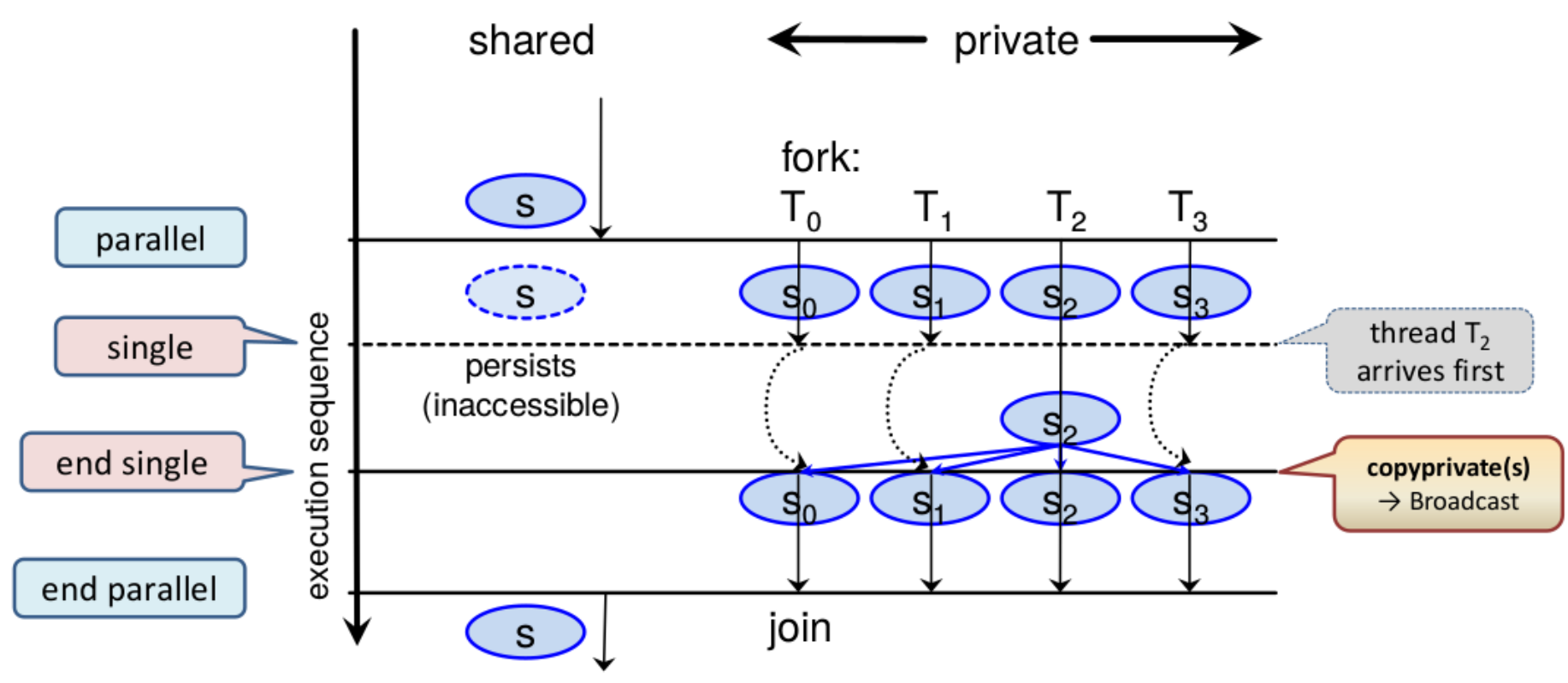
OpenMP: copyprivate clause

```c
float s;

  s = …;
#pragma omp parallel private(s)
 {
#pragma omp single \
        copyprivate(s)
 {
  …;
  s = …;
 }  // end single
… = … + s;
} // end parallel
```

- A thread enters the single region, does computations with s, then broadcasts value of s to other threads private copy of s

OpenMP: copyprivate clause



- A thread enters the single region, does computations with s, then broadcasts value of s to other threads private copy of s

OpenMP: Conditional Parallel Regions

```
#pragma omp parallel if (n > 800)
{
...
...
}
```

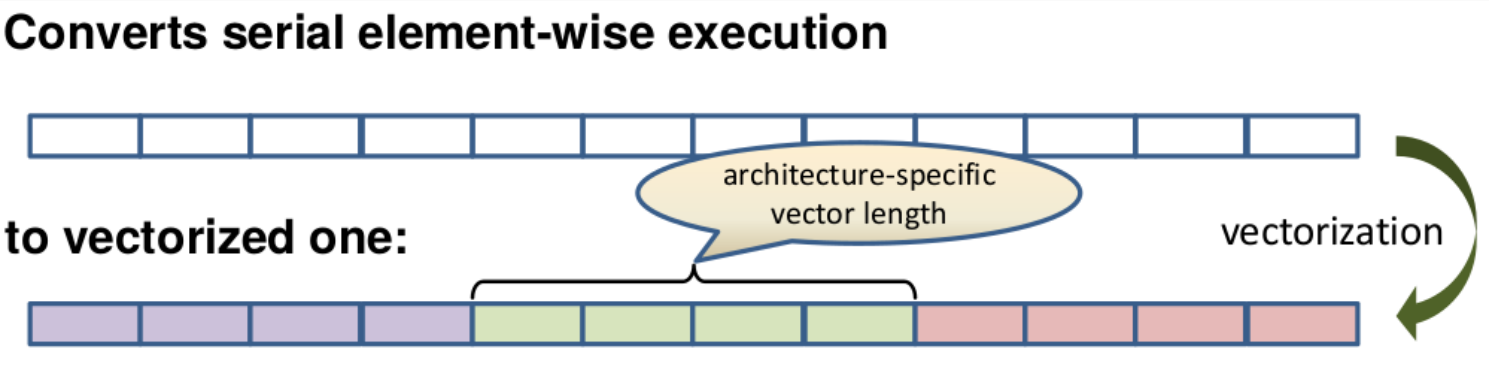
- If $n > 800$ then create the parallel region

```
#pragma omp parallel if ( ! omp_in_parallel() )
```

- If already not in parallel region, create parallel region

OpenMP: SIMD

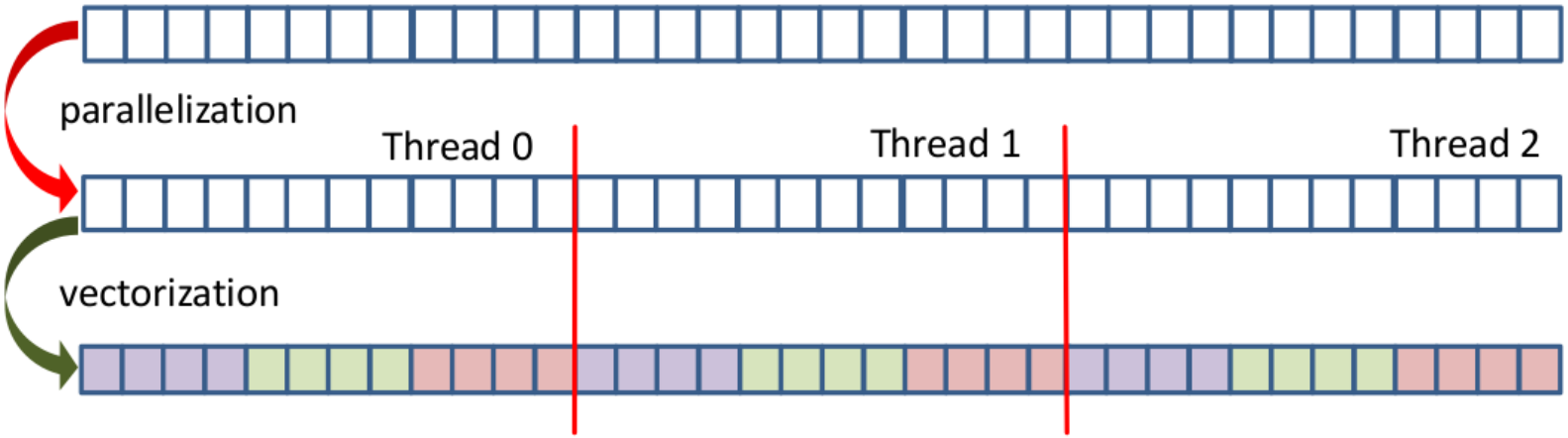
```c
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++) {
    sum += a[k] * b[k];
  }
```



OpenMP: Worksharing for loop + SIMD

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++) {
    sum += a[k] * b[k];
  }
```

assume invocation by all threads executing in a parallel region



OpenMP: Function Call inside SIMD

- **Function call inside SIMD region**

```
float min(float a, float b) {
  return a < b ? a : b;
}

float distsq(float x, float y) {
  return (x - y)*(x - y);
}

void example() {
#pragma omp for simd
  for (i=0; i<N; i++) {
    d[i] = min(
      distsq( a[i],b[i] ),c[i] );
  }
}
```

may fail if functions outside file scope

- **Therapy: explicitly declare for use in vectorized loops**
  - C/C++ syntax

```
#pragma omp declare simd
function def. or decl.
```

  - Fortran syntax

```
!$omp declare simd &
!$omp (proc-name-list)
```

  - clauses are also supported
  - causes generation of multi-version code by the compiler

OpenMP: Function Call inside SIMD

- **vectorized versions of generated functions are shown**

```
#pragma omp declare simd
float min(float a, float b) {
  return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
  return (x - y)*(x - y);
}


void example() {
#pragma omp for simd
  for (i=0; i<N; i++) {
    d[i] = min(
      distsq( a[i],b[i] ),c[i] );
  }
}
```

```
vec8 min_v(vec8 a, vec8 b) {
  return a < b ? a : b;
}
```

```
vec8 distsq_v(vec8 x, vec8 y) {
  return (x - y)*(x - y);
}
```

no SIMD directives permitted inside vectorized functions!

```
vd = min_v(
     distsq_v (va, vb), vc );
```