OpenMP topic: Loop parallelism

Experimental html version of downloadable textbook, see http://www.tacc.utexas.edu/~eijkhout/istc/istc.html

17 OpenMP topic: Loop parallelism

17.1 Loop parallelism

Loop parallelism is a very common type of parallelism in scientific codes, so OpenMP has an easy mechanism for it. OpenMP parallel loops are a first example of OpenMP `worksharing' constructs (see section labelstring for the full list): constructs that 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
}
```

A more natural option is to use the `parallel for` pragma:

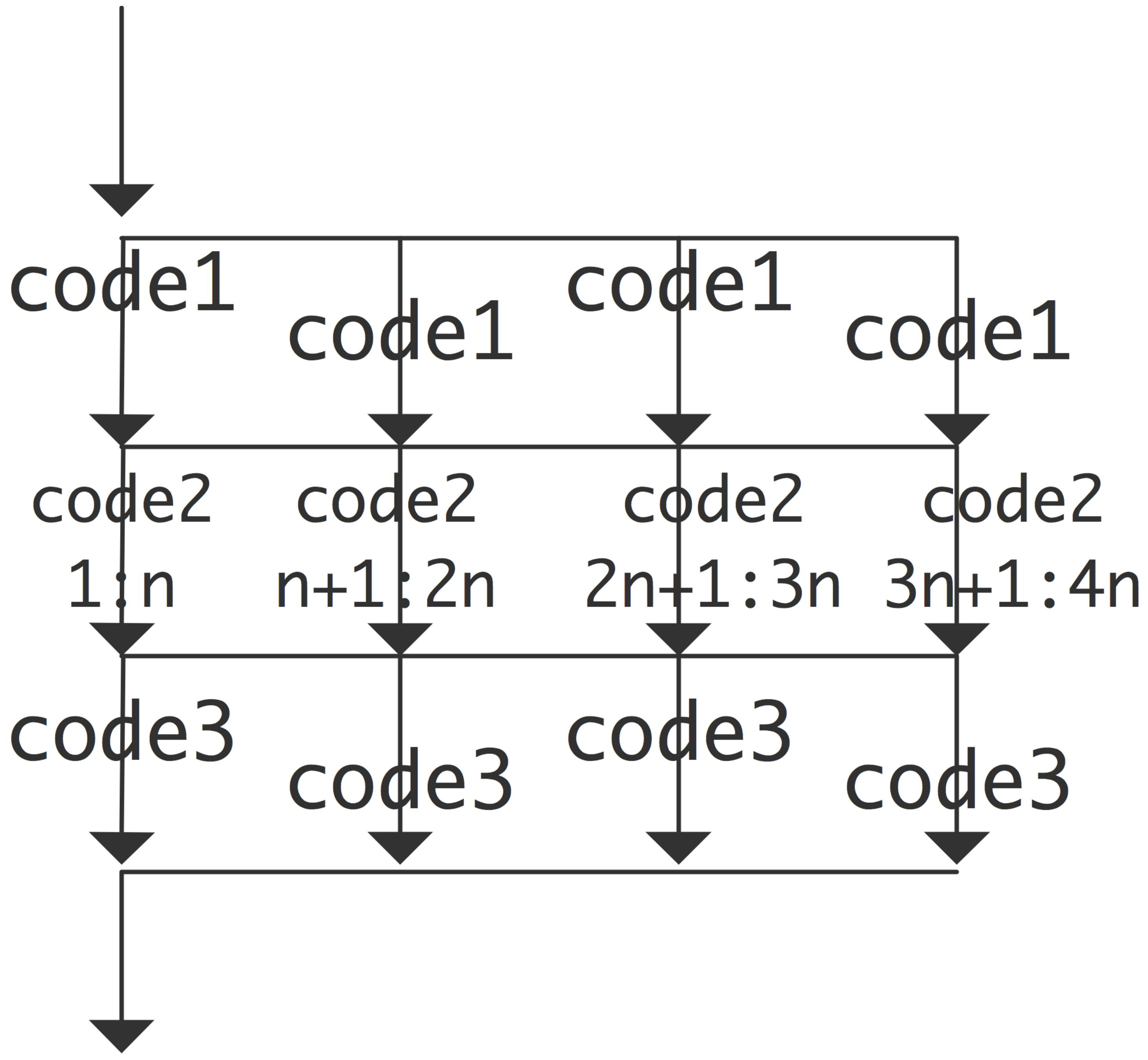
```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
  // do something with i
}
```

This has several advantages. For one, 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 (section 17.2).

Figure 1 shows the execution on four threads of

```
#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; the loop iterations are spread over the four threads.



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; .....
```

Exercise

Compute π by *numerical integration* . We use the fact that π is the area of the unit circle, and we approximate this by computing the area of a quarter circle using *Riemann sums* .

- Let $f(x) = \sqrt{1-x^2}$ be the function that describes the quarter circle for $x = 0 \ldots 1$;
- Then we compute

$$\pi/4 \approx \sum_{i=0}^{N-1} \Delta x f(x_i) \qquad \text{where } x_i = i\Delta x \text{ and } \Delta x = 1/N \tag{1}$$

Write a program for this, and parallelize it using OpenMP parallel for directives.

1. Put a `parallel` directive around your loop. Does it still compute the right result? Does the time go down with the number of threads? (The answers should be no and no.)
2. Change the `parallel` to `parallel for` (or \n{parallel do}). Now is the result correct? Does execution speed up? (The answers should now be no and yes.)
3. Put a `critical` directive in front of the update. (Yes and very much no.)
4. Remove the `critical` and add a clause `reduction(+:quarterpi)` to the `for` directive. Now it should be correct and efficient.

Use different numbers of cores and compute the speedup you attain over the sequential computation. Is there a performance difference between the OpenMP code with 1 thread and the sequential code?

Remark

In this exercise you may have seen the runtime go up a couple of times where you weren't expecting it. The issue here is \indexterm{false sharing}; see \HPSCref{sec:roundoff-parallel} for more explanation.

There are some restrictions on the loop: basically, OpenMP needs to be able to determine in advance how many iterations there will be.

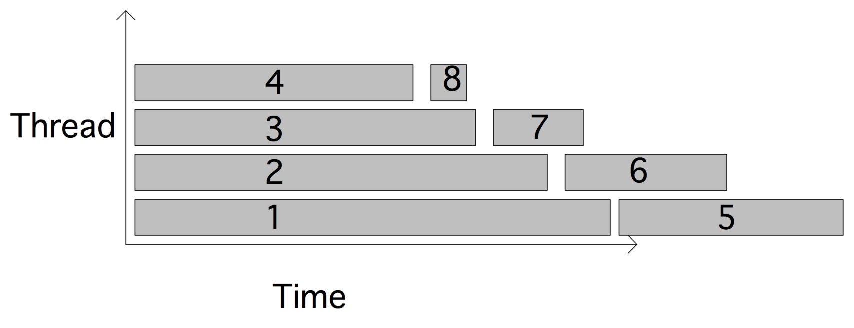
- The loop can not contains `break` , `return` , `exit` statements, or `goto` to a label outside the loop.
- The `continue` (C) or `cycle` (F) statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and not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

17.2 Loop schedules

Usually you will have many more iterations in a loop than there are threads. Thus, there are several ways you can assign your loop iterations to the threads. OpenMP lets you specify this with the \indexompshow{schedule} clause.

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

The first distinction we now have to make is between static and dynamic schedules. With static schedules, the iterations are assigned purely based on the number of iterations and the number of threads (and the `chunk` parameter; see later). In dynamic schedules, on the other hand, iterations are assigned to threads that are unoccupied. Dynamic schedules are a good idea if iterations take an unpredictable amount of time, so that *load balancing* is needed.



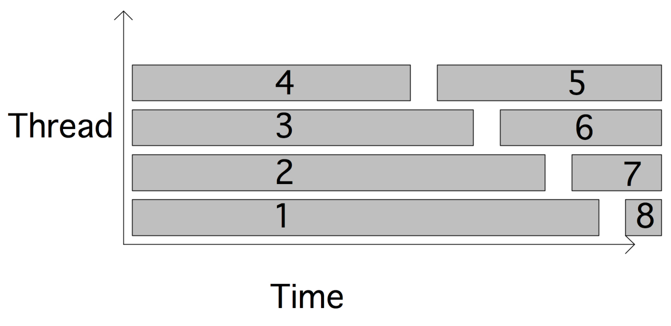
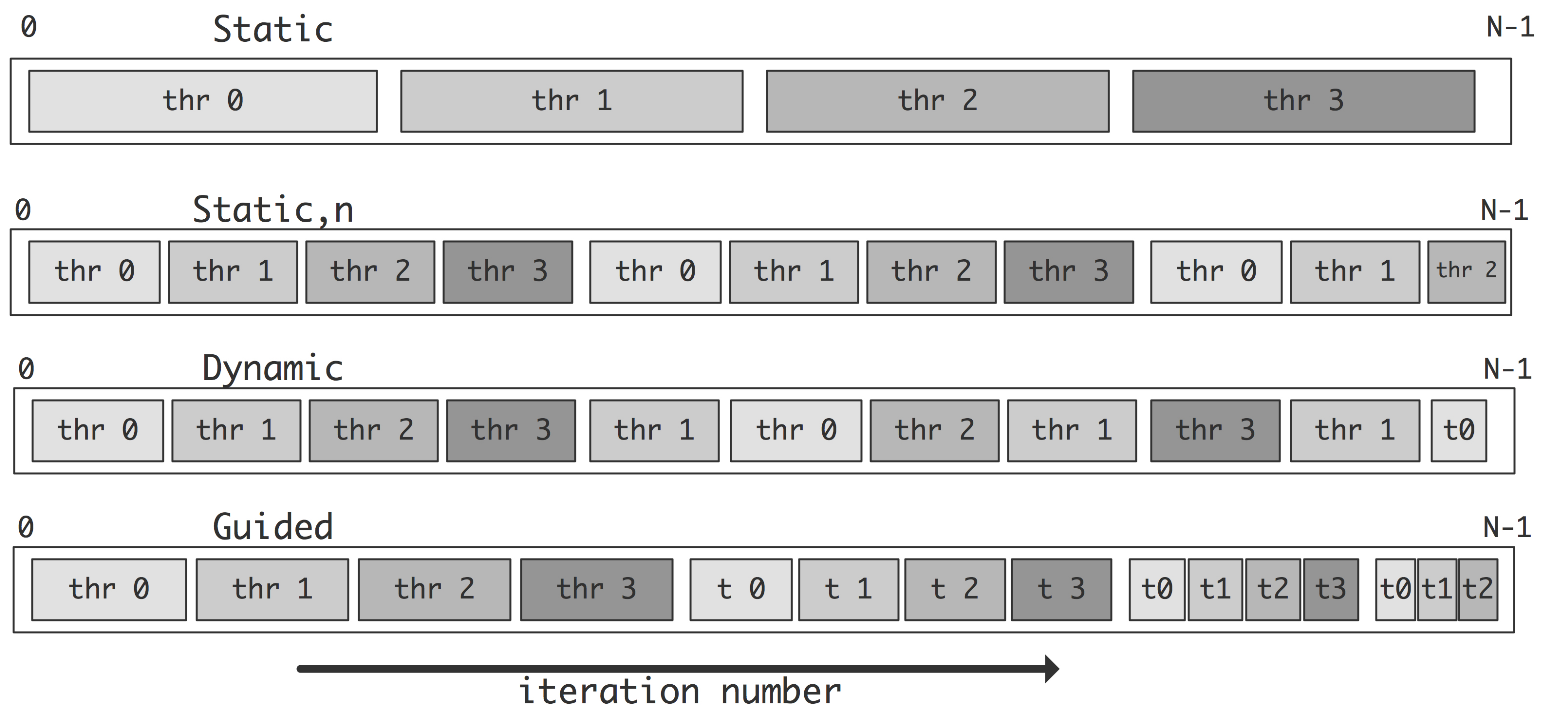


Figure 2 illustrates this: assume that each core gets assigned two (blocks of) iterations and these blocks take gradually less and less time. You see from the left picture that thread 1 gets two fairly long blocks, where as thread 4 gets two short blocks, thus finishing much earlier. (This phenomenon of threads having unequal amounts of work is known as \indexterm{load imbalance}.) On the other hand, in the right figure thread 4 gets block 5, since it finishes the first set of blocks early. The effect is a perfect load balancing.



The default static schedule is to assign one consecutive block of iterations to each thread. If you want different sized blocks you can defined a \indexclauseoption{schedule}{chunk} size:

```
#pragma omp for schedule(static[,chunk])
```

(where the square brackets indicate an optional argument). With static scheduling, the compiler will split up the loop iterations at compile time, so, provided the iterations take roughly the same amount of time, this is the most efficient at runtime.

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

Exercise Why is a chunk size of 1 typically a bad idea? (Hint: think about cache lines, and read \HPSCref{sec:falseshare}.)

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

```
#pragma omp for schedule(static[,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.

Finally, there is the \indexclauseoption{schedule}{guided} schedule, which gradually decreases the chunk size. The thinking here is that large chunks carry the least overhead, but smaller chunks are better for load balancing. The various schedules are illustrated in figure 3 .

If you don't want to decide on a schedule in your code, you can specify the \indexclauseoption{schedule}{runtime} schedule. The actual schedule will then at runtime be read from the \indexompshow{OMP_SCHEDULE} environment variable. You can even just leave it to the runtime library by specifying \indexclauseoption{schedule}{auto}

Exercise

We continue with exercise 17.1 . We add `adaptive integration': where needed, the program refines the step size\footnote{It doesn't actually do this in a mathematically sophisticated way, so this code is more for the sake of the example.}. This means that the iterations no longer take a predictable amount of time.

\small

```
for (i=0; i<nsteps; i++) {
 double
   x = i*h,x2 = (i+1)*h,
   y = sqrt(1-x*x),y2 = sqrt(1-x2*x2),
   slope = (y-y2)/h;
 if (slope>15) slope = 15;
 int
   samples = 1+(int)slope, is;
 for (is=0; is<samples; is++) {
   double
     hs = h/samples,
     xs = x+ is*hs,
     ys = sqrt(1-xs*xs);
   quarterpi += hs*ys;
   nsamples++;
 }
}
pi = 4*quarterpi;
```

1. Use the `omp parallel for` construct to parallelize the loop. As in the previous lab, you may at first see an incorrect result. Use the `reduction` clause to fix this.
2. Your code should now see a decent speedup, using up to 8 cores. However, it is possible to get completely linear speedup. For this you need to adjust the schedule.

   Start by using `schedule(static,`n`)` . Experiment with values for n. When can you get a better speedup? Explain this.
3. Since this code is somewhat dynamic, try `schedule(dynamic)` . This will actually give a fairly bad result. Why? Use `schedule(dynamic,`n`)` instead, and experiment with values for n.
4. Finally, use `schedule(guided)` , where OpenMP uses a heuristic. What results does that give?

Exercise

Program the *LU factorization* algorithm without pivoting.

```
for k=1,n:
  A[k,k] = 1./A[k,k]
  for i=k+1,n:
    A[i,k] = A[i,k]/A[k,k]
    for j=k+1,n:
      A[i,j] = A[i,j] - A[i,k]*A[k,j]
```

1. Argue that it is not possible to parallelize the outer loop.
2. Argue that it is possible to parallelize both the i and j loops.

3. Parallelize the algorithm by focusing on the i loop. Why is the algorithm as given here best for a matrix on row-storage? What would you do if the matrix was on column storage?
4. Argue that with the default schedule, if a row is updated by one thread in one iteration, it may very well be updated by another thread in another. Can you find a way to schedule loop iterations so that this does not happen? What practical reason is there for doing so?

The schedule can be declared explicitly, set at runtime through the \indexompshow{OMP_SCHEDULE} environment variable, or left up to the runtime system by specifying `auto` . Especially in the last two cases you may want to enquire what schedule is currently being used with \indexompshow{omp_get_schedule}.

```
int omp_get_schedule(omp_sched_t * kind, int * modifier );
```

Its mirror call is \indexompshow{omp_set_schedule}, which sets the value that is used when schedule value `runtime` is used. It is in effect equivalent to setting the environment variable `OMP_SCHEDULE` .

```
void omp_set_schedule (omp_sched_t kind, int modifier);
```

Type	environment variable	clause	modifier default
	{\tt OMP_SCHEDULE\char`\=}	{\tt schedule(...)}	
static	{static[,n]}	{static[,n]}	$N/nthreads$
dynamic	{dynamic[,n]}	{dynamic[,n]}	1
guided	{guided[,n]}	{guided[,n]}	

Here are the various schedules you can set with the `schedule` clause:

- [affinity] Set by using value \indexompshow{omp_sched_affinity}
- [auto] The schedule is left up to the implementation. Set by using value \indexompshow{omp_sched_auto}
- [dynamic] value:nbsp;2. The modifier parameter is the *chunk* size; defaultnbsp;1. Set by using value \indexompshow{omp_sched_dynamic}
- [guided] Value:nbsp;3. The modifier parameter is the `chunk` size. Set by using value \indexompshow{omp_sched_guided}
- [runtime] Use the value of the \indexompshow{OMP_SCHEDULE} environment variable. Set by using value \indexompshow{omp_sched_runtime}
- [static] value:nbsp;1. The modifier parameter is the *chunk* size. Set by using value \indexompshow{omp_sched_static}

17.3 Reductions

Top > Reductions

So far we have focused on loops with independent iterations. Reductions are a common type of loop with dependencies. There is an extended discussion of reductions in sectionnbsp; .

17.4 Collapsing nested loops

Top > Collapsing nested loops

In general, the more work there is to divide over a number of threads, the more efficient the parallelization will be. In the context of parallel loops, it is possible to increase the amount of work by parallelizing all levels of loops instead of just the outer one.

Example: in

```
for ( i=0; ilt;N; i++ )
for ( j=0; jlt;N; j++ )
A[i][j] = B[i][j] + C[i][j]
```

all N^2 iterations are independent, but a regular `omp for` directive will only parallelize one level. The \indexclause{collapse} clause will parallelize more than one level:

```
#pragma omp for collapse(2)
for ( i=0; ilt;N; i++ )
for ( j=0; jlt;N; j++ )
A[i][j] = B[i][j] + C[i][j]
```

It is only possible to collapse perfectly nested loops, that is, the loop body of the outer loop can consist only of the inner loop; there can be no statements before or after the inner loop in the loop body of the outer loop. That is, the two loops in

```
for (i=0; ilt;N; i++) {
y[i] = 0.;
for (j=0; jlt;N; j++)
y[i] + A[i][j] * x[j]
}
```

can not be collapsed.

Exercise

Can you rewrite the preceding code example so that it can be collapsed? Do timing tests to see if you can notice the improvement from collapsing.

17.5 Ordered iterations

Top > Ordered iterations

Iterations in a parallel loop that are execution in parallel do not execute in lockstep. That means that in

```
#pragma omp parallel for
for ( ... i ... ) {
... f(i) ...
printf("something with
}
```

it is not true that all function evaluations happen more or less at the same time, followed by all print statements. The print statements can really happen in any order. The \indexclause{ordered} clause coupled with the `ordered` directive can force execution in the right order:

```
#pragma omp parallel for ordered
for ( ... i ... ) {
... f(i) ...
#pragma omp ordered
printf("something with
}
```

Example code structure:

```
#pragma omp parallel for shared(y) ordered
for ( ... i ... ) {
int x = f(i)
#pragma omp ordered
y[i] += f(x)
z[i] = g(y[i])
}
```

There is a limitation: each iteration can encounter only one `ordered` directive.

17.6 \texttt{nowait}

Top > \texttt{nowait}

The implicit barrier at the end of a work sharing construct can be cancelled with a \indexclause{nowait} clause. This has the effect that threads that are finished can continue with the next code in the parallel region:

```
#pragma omp parallel
{
#pragma omp for nowait
for (i=0; ilt;N; i++) { ... }
// more parallel code
}
```

In the following example, threads that are finished with the first loop can start on the second. Note that this requires both loops to have the same schedule.

```
#pragma omp parallel
{
x = local_computation()
#pragma omp for nowait
for (i=0; ilt;N; i++) {
x[i] = ...
}
#pragma omp for
for (i=0; ilt;N; i++) {
y[i] = ... x[i] ...
}
}
```

17.7 While loops

Top > While loops

OpenMP can only handle `for' loops: *while loops* can not be parallelized. So you have to find a way around that. While loops are for instance used to search through data:

```
while ( a[i]!=0  ilt;imax ) {
i++; }
// now i is the first index for which
a[i]
is zero.
```

We replace the while loop by a for loop that examines all locations:

```
result = -1;
#pragma omp parallel for
for (i=0; ilt;imax; i++) {
if (a[i]!=0  resultlt;0) result = i;
}
```

Exercise Show that this code has a race condition.

You can fix the race condition by making the condition into a critical section; sectionnbsp; . In this particular example, with a very small amount of work per iteration, that is likely to be inefficient in this case (why?). Anbsp;more efficient solution uses the `lastprivate` pragma:

```
result = -1;
#pragma omp parallel for lastprivate(result)
for (i=0; ilt;imax; i++) {
if (a[i]!=0) result = i;
}
```

You have now solved a slightly different problem: the result variable contains the *last* location where `a[i]` is zero. contains the *last* location where `a[i]` is zero.

{|||||||}

Back to Table of Contents