How can we do this in OpenMP?

```
double t[4] = {0.0, 0.0, 0.0, 0.0}
int omp_set_num_threads(4);
#pragma omp parallel for
for (i=0; i < n; i++) {
   t[omp_get_thread_num( )] += a[i];
}
avg = 0;
for (i=0; i < 4; i++) }
   avg += t[i];
}
avg = avg / n;
```

This is getting messy and we still are using a O(#threads) summation of the partial sums.

parallel

serial

OpenMP function

OpenMP provides a better way

- Reductions are common enough that OpenMP provides support for them
- reduction clause for omp parallel pragma
- specify variable and operation
- OpenMP takes care of creating temporaries, computing partial sums, and computing the final sum

Dot product example

```
t=0;
for (i=0; i < n; i++) {
  t = t + a[i]*c[i];
}
```

OpenMP makes *t* private, puts the partial sums for each thread into *t*, and then forms the full sum of *t* as shown earlier

```
t=0;
#pragma omp parallel for reduction(+:t)
for (i=0; i < n; i++) {
     t = t + (a[i]*c[i]);
}
```

Restrictions on Reductions

Operations on the reduction variable must be of the form

> ***x*** *= x op expr*
>
> x = expr op x (except subtraction)
>
> x binop = expr
>
> x++
>
> ++x
>
> x--
>
> --x

- ***x*** is a scalar variable in the list
- ***expr*** is a scalar expression that does not reference ***x***
- ***op*** is not overloaded, and is one of +, *, -, /, &, ^, |, &&, ||
- ***binop*** is not overloaded, and is one of +, *, -, /, &, ^, |

Why the restrictions on where *t* can appear?

```
#pragma omp parallel for reduction(+:t)
// each element of a[i] = 1
for (i=0; i<n; i++) {
    b[i] = t;
     t += a[i];
}
```

- In the sequential loop, at the end of iteration *i*, $t = i+1$.
- Let s_t be the starting iteration for the thread *t*, then

  $$st = (t-1)*ceil(n/\#threads)+i+1$$
- If executed as a recurrence using a static distribution of iterations, at the end of iteration *i*, $t=i-s_t$,
- Thus, if $n = 100$, thread *3* executes iterations *50...74*, and in iteration *60*, $i = 11$
- This means $b[61] = 11$, not *61*
- making it work right is, in general, hard to do efficiently.
- Thus the OpenMP restriction on where *t* can appear

Improving performance of parallel loops

```
#pragma omp parallel for reduction(+:t)
for (i=0; i < n; i++) {
    t = t + (a[i]*c[i]);
}
```

- Parallel loop startup and teardown has a cost
- Parallel loops with few iterations can lead to slowdowns -- if clause allows us to avoid this
- This overhead is one reason to try and parallelize outermost loops.

```
#pragma omp parallel for reduction(+:t) if (n>1000)
for (i=0; i < n; i++) {
    t = t + (a[i]*c[i]);
}
```

Assigning iterations to threads (thread schduling)

- The schedule clause can guide how iterations of a loop are assigned to threads
- Two kinds of schedules:
    - static: iterations are assigned to threads at the start of the loop. Low overhead but possible load balance issues.
    - dynamic: some iterations are assigned at the start of the loop, others as the loop progresses. Higher overheads but better load balance.
- A *chunk* is a contiguous set of iterations

The schedule clause - static

- schedule(*type[, chunk])* where "*[ ]*" indicates optional
- *(type [,chunk])* is
  - (static): chunks of ~ *n/t* iterations per thread, no chunk specified. The default.
  - (static, chunk): chunks of size *chunk* distributed round-robin. No *chunk* specified means *chunk = 1*

The schedule clause - dynamic

- schedule(*type[, chunk])* where "*[ ]*" indicates optional
- *(type [,chunk])* is
  - (dynamic): chunks of size of *1* iteration distributed dynamically
  - (dynamic, chunk): chunks of size *chunk* distributed dynamically

Static

	thread 0	thread 1	thread 2
Chunk = 1	0, 3, 6, 9, 12	1, 4, 7, 10, 13	2, 5, 8, 11, 14

	thread 0	thread 1	thread 2
Chunk = 2	0, 1, 6, 7, 12, 13	2, 3, 8, 9, 14, 15	4, 5, 10, 11, 16, 17

With no chunk size specified, the iterations are divided as evenly as possible among processors, with one chunk per processor

With *dynamic* chunks go to processors as work needed.

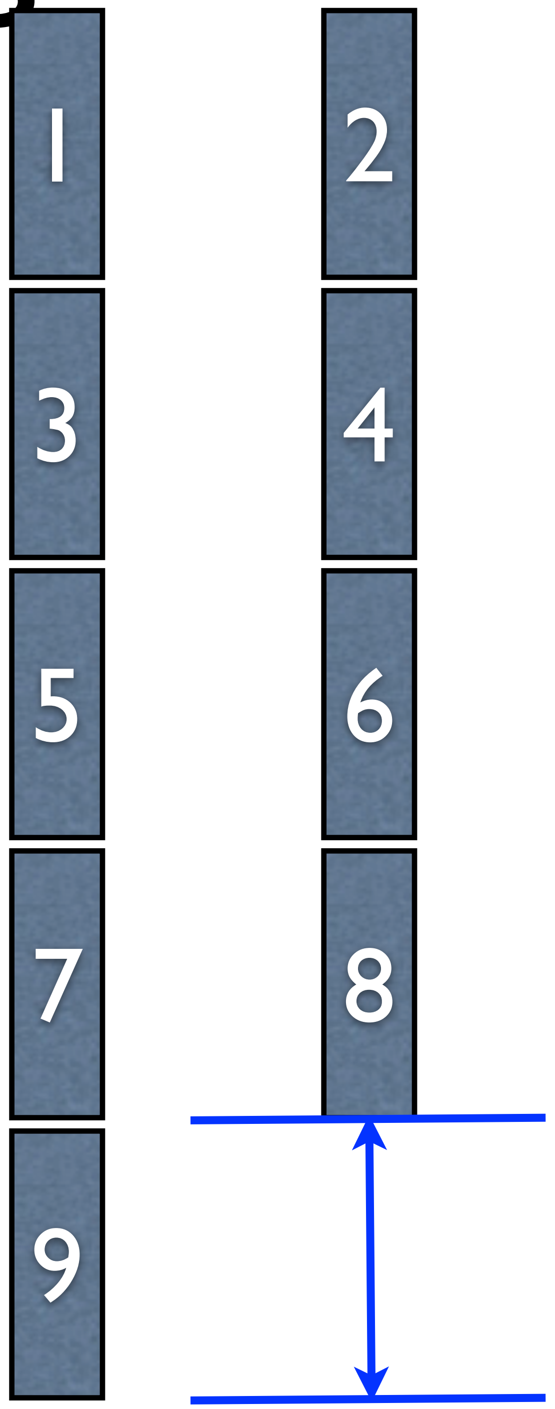
The schedule clause

- schedule(*type[, chunk])* (type [,chunk]) is
  - (guided,chunk): uses *guided self scheduling* heuristic. Starts with big chunks and decreases to a minimum chunk size of *chunk*
  - runtime - type depends on value of OMP_SCHEDULE environment variable, e.g. setenv OMP_SCHEDULE=”static,1”

Guided with two threads example

Dynamic schedule with large blocks

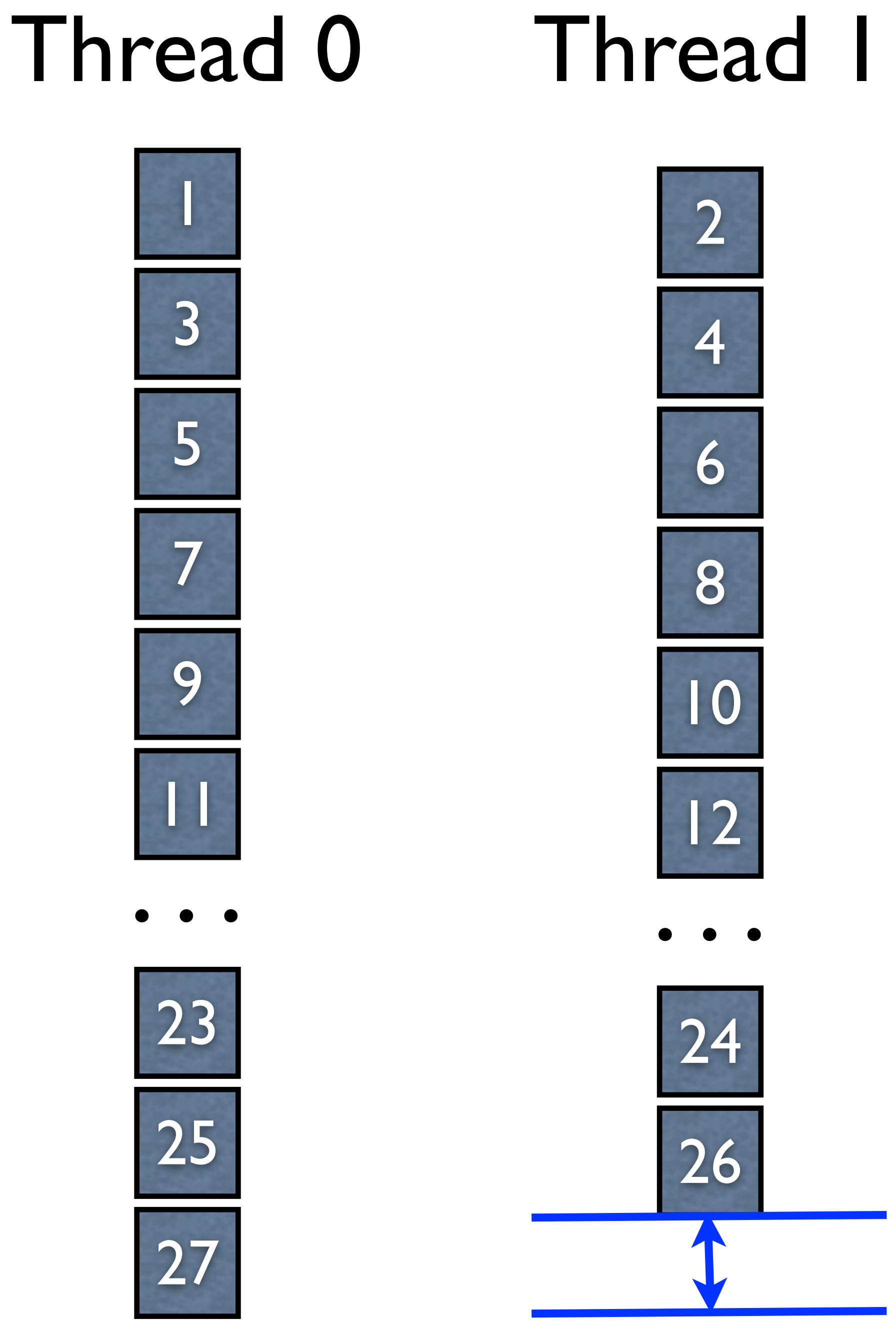
Large blocks reduce scheduling costs, but lead to large load imbalance



Dynamic schedule with smals blocks

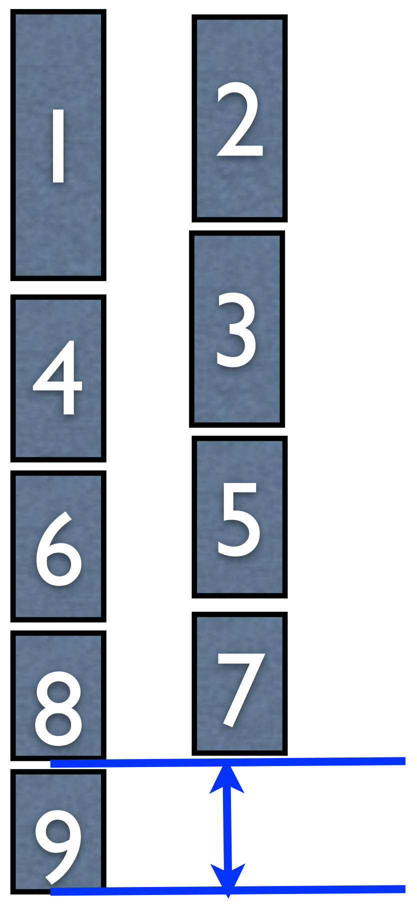
Small blocks have a smaller load imbalance, but with higher scheduling costs.

Would like the best of both methods.



Guided with two threads

By starting out with larger blocks, and then ending with small ones, scheduling overhead and load imbalance can both be minimized.



Books online for relatively cheap

- http://www.abebooks.com/ for about $28.00
- I have used them, but they aggregate different sellers, and cannot vouch for the seller with the best prices.

Program Translation for Microtasking Scheme

```
   Subroutine x
   ...
C$OMP PARALLEL DO
   DO j=1,n
     a(j)=b(j)
   ENDDO
   …
   END
```

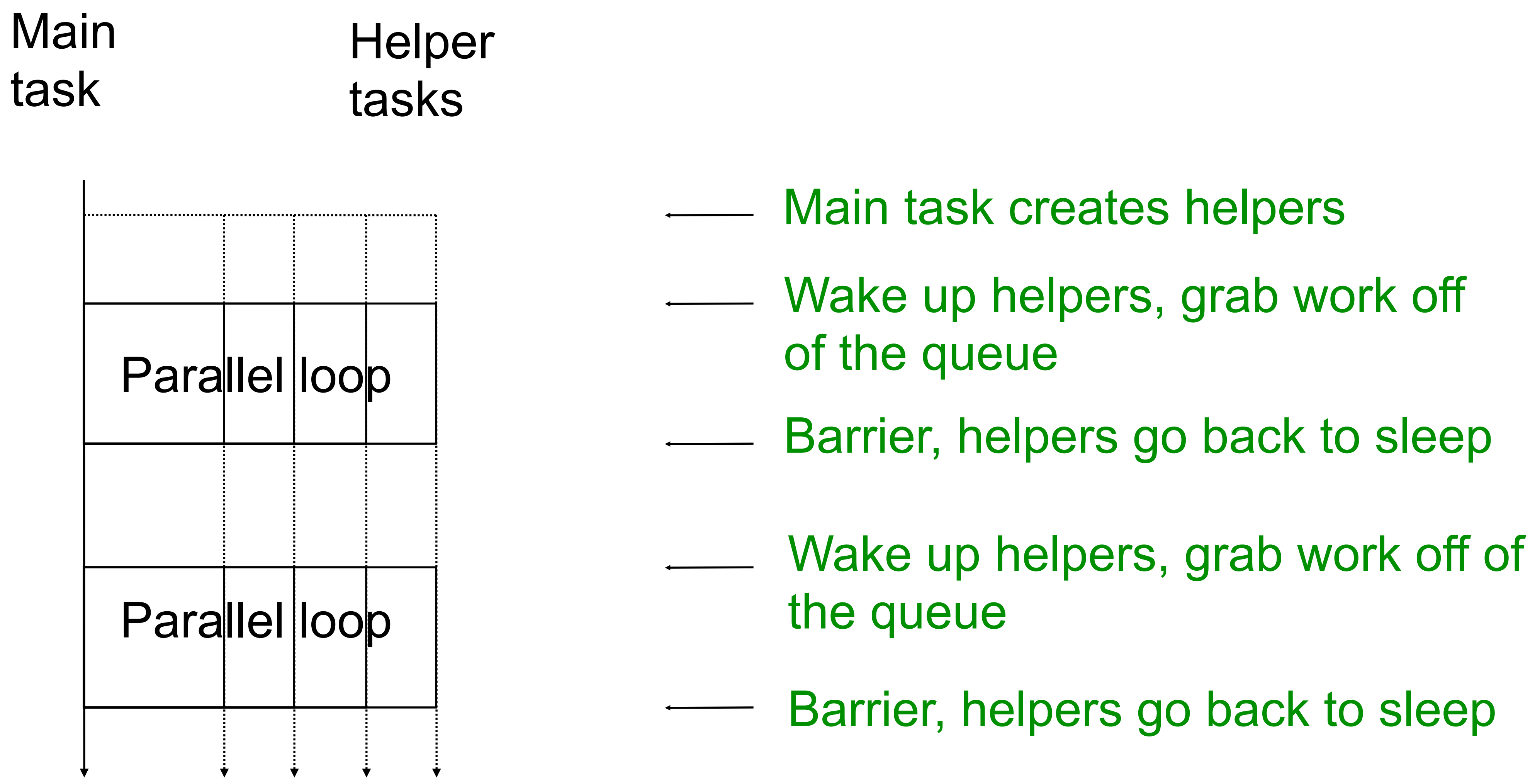
$\rightarrow$

```
Subroutine x
…
call scheduler(1,n,a,b,loopsub)
…
END
```

```
Subroutine loopsub(lb,ub,a,b)
integer lb,ub
DO jj=lb,ub
   a(jj)=b(jj)
ENDDO
END
```

Parallel ExecutionScheme

- Most widely used: Microtasking scheme

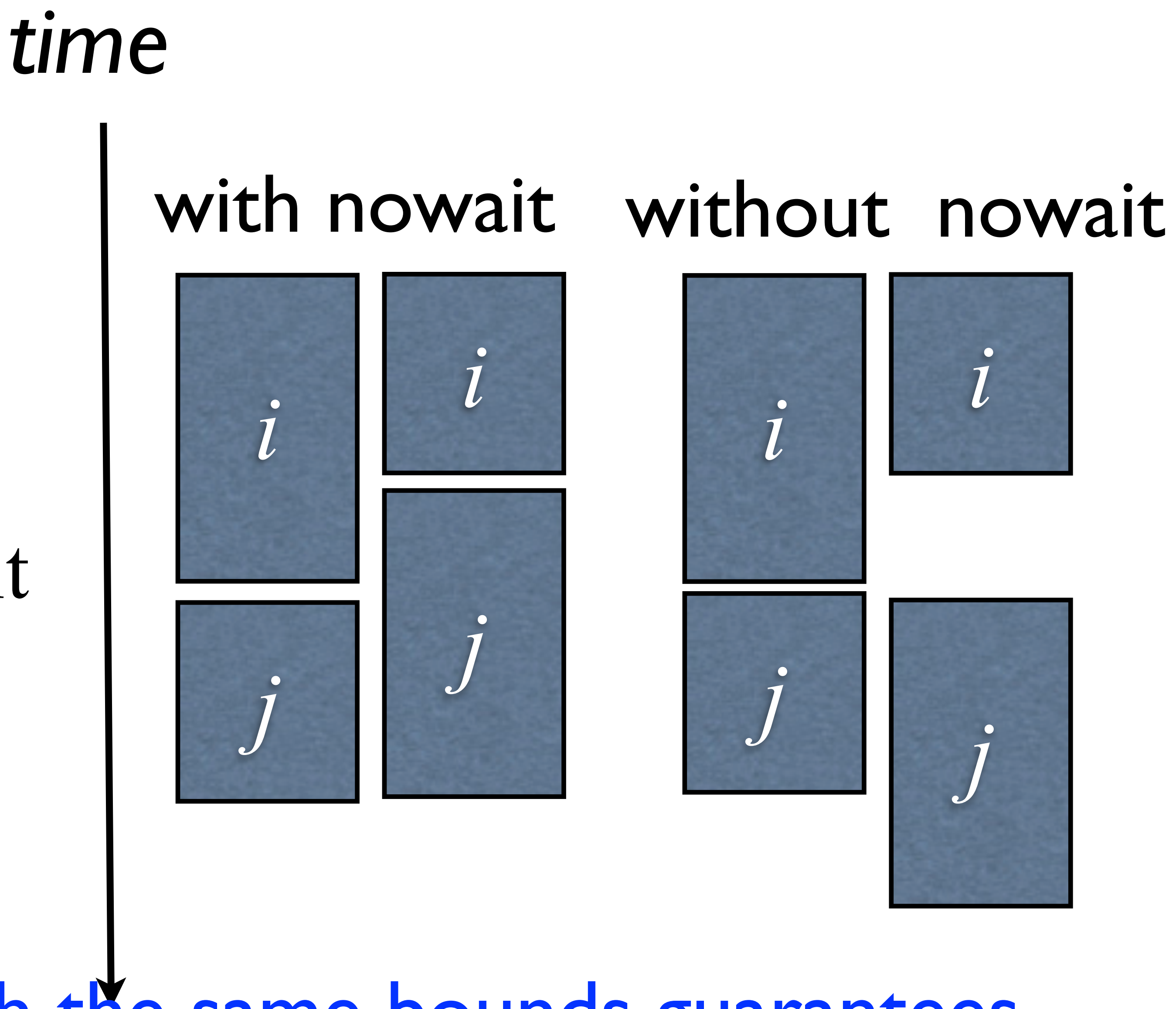

Main task
Helper tasks
Parallel loop
Parallel loop
Main task creates helpers
Wake up helpers, grab work off of the queue
Barrier, helpers go back to sleep
Wake up helpers, grab work off of the queue
Barrier, helpers go back to sleep

The nowait clause

```
#pragma omp parallel for
for (i=0; i < n; i++) {
   if (a[i] > 0) a[i] += b[i];
}
```

barrier here by default

```
#pragma omp parallel for nowait
for (i=0; i < n; i++) {
   if (a[i] < 0) a[i] -= b[i];
}
```

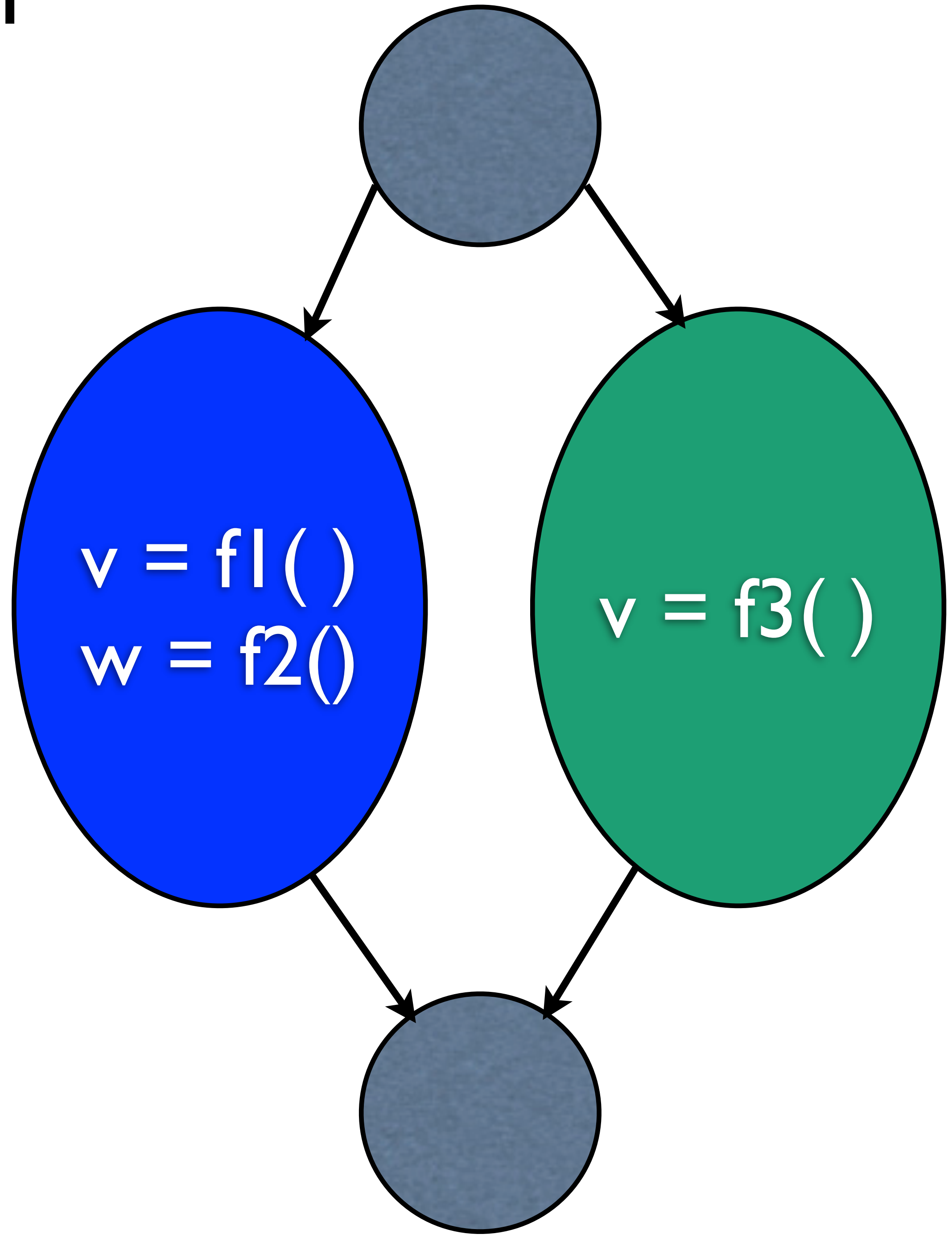


Only the static distribution with the same bounds guarantees the same thread will execute the same iterations from both loops.

The sections pragma

Used to specify *task* parallelism

```
#pragma omp parallel sections
   {
#pragma omp section /* optional */
   {
   v = f1( )
   w = f2( )
   }
#pragma omp section
  v = f3( )
}
```



The parallel pragma

```
#pragma omp parallel private(w)
{
  w = getWork (q);
  while (w != NULL) {
    doWork(w);
    w = getWork(q);
  }
}
```

- every processor executes the statement following the *parallel* pragma
- Parallelism of useful work in the example because independent and different work pulled of of *q*
- *q* needs to be thread safe

The parallel pragma

```
#pragma omp parallel private(w)
{
  w = getWork (q);
  while (w != NULL) {
    doWork(w);
#pragma omp critical
    w = getWork(q);
  }
}
```

- If data structure pointed to by *q* is not thread safe, need to synchronize it in your code
- One way is to use a *critical* clause

single and *master* clauses also exist.

Does OpenMP provide a way to specify:

- what parts of the program execute in parallel with one another
- how the work is distributed across different cores
- the order that reads and writes to memory will take place
- that a sequence of accesses to a variable will occur *atomically* or without interference from other threads.
- **And,** ideally, it will do this while giving *good performance* and allowing *maintainable programs* to be written.

What executes in parallel?

```
c = 57.0;
for (i=0; i < n; i++) {
  a[i] = c + a[i]*b[i]
}
```

```
c = 57.0
#pragma omp parallel for
for (i=0; i < n; i++) {
  a[i] = + c + a[i]*b[i]
}
```

- *pragma* appears like a comment to a non-OpenMP compiler
- pragma requests parallel code to be produced for the following for loop

The order that reads and writes to memory occur

```
c = 57.0
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c + a[i]*b[i]
}
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c + a[i]*b[i]
}
```

- Within an iteration, access to data appears in-order
- Across iterations, no order is implied. *Races* lead to undefined programs
- Across loops, an implicit *barrier* prevents a loop from starting execution until all iterations and writes (stores) to memory in the previous loop are finished
- Parallel constructs execute after preceding sequential constructs finish

Relaxing the order that reads and writes to memory occur

```
c = 57.0
#pragma omp parallel for schedule(static) nowait
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
```

no barrier

```
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
```

The *nowait* clause allows a thread to begin executing its part of the code after the nowait loop as soon as it finishes its part of the nowait loop

Accessing variables without interference from other threads

```
#pragma omp parallel for
for (i=0; i < n; i++) {
    a = a + b[i]
}
```

Dangerous -- all iterations are updating *a* at the same time -- a *race* (or *data race*).

```
#pragma omp parallel for
for (i=0; i < n; i++) {
#pragma omp critical
    a = a + b[i];
}
```

Stupid but correct -- *critical* pragma allows only one thread to execute the next statement at a time. *Very inefficient -- but ok if you have enough work to make it worthwhile.*