UNIL | Université de Lausanne

OpenMP II

Simon Scheidegger
simon.scheidegger@gmail.com
July 23th, 2019
Open Source Economics Laboratory – BFI/UChicago

Including adapted teaching material from books, lectures and presentations by
B. Barney, B. Cumming, G. Hager, R. Rabenseifner, O. Schenk, G. Wellein

Roadmap – fast forward (3):

Day 3, Tuesday – July 23th

1. OpenMP session II (8.00-9.00 – hands on).
2. MPI session I (9.30-10.30 – hands on).
3. MPI session II (10.45-11.45 – hands on).
4. Exercise sheets (OpenMP & MPI) related to the day's topics (11.50-12.00).

Outline

- Work sharing constructs
  - → Loops
  - → Sections
  - → Reductions (max, summation,...)

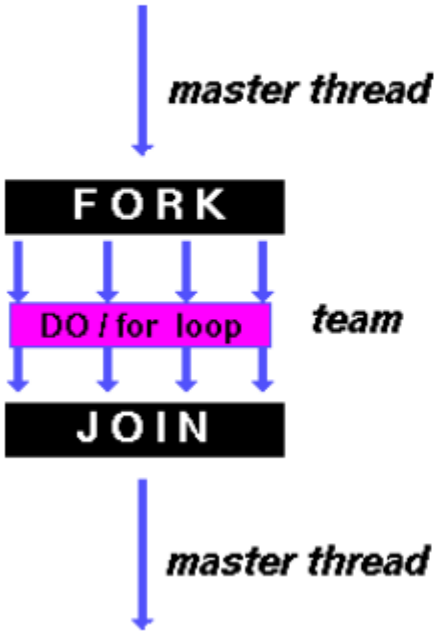
Components of OpenMP

Directives	Runtime library routines	Environment variables
Parallel regions	Number of threads	Number of threads
Work sharing	Thread ID	Scheduling type
Synchronization	Dynamic thread adjustment	Dynamic thread adjustment
Data scope attributes : • private • firstprivate • last private • shared • reduction	Nested Parallelism	Nested Parallelism
Orphaning	Timers	
	API for locking	

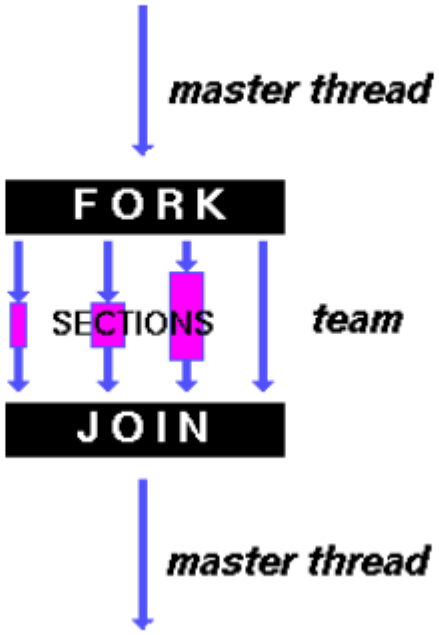
Worksharing constructs in OpenMP

See https://computing.llnl.gov/tutorials/openMP/

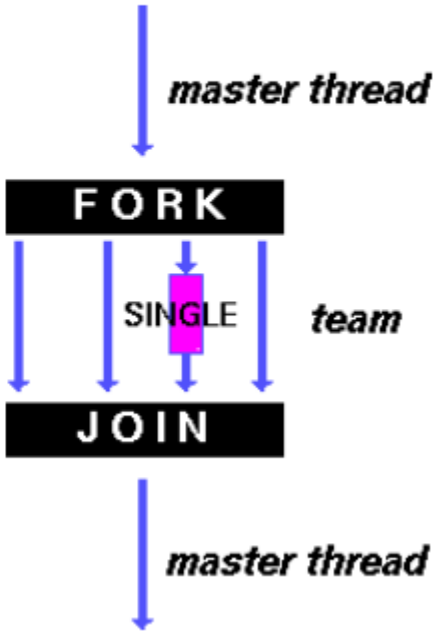
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



SINGLE - serializes a section of code



OpenMP worksharing “for/do loops”

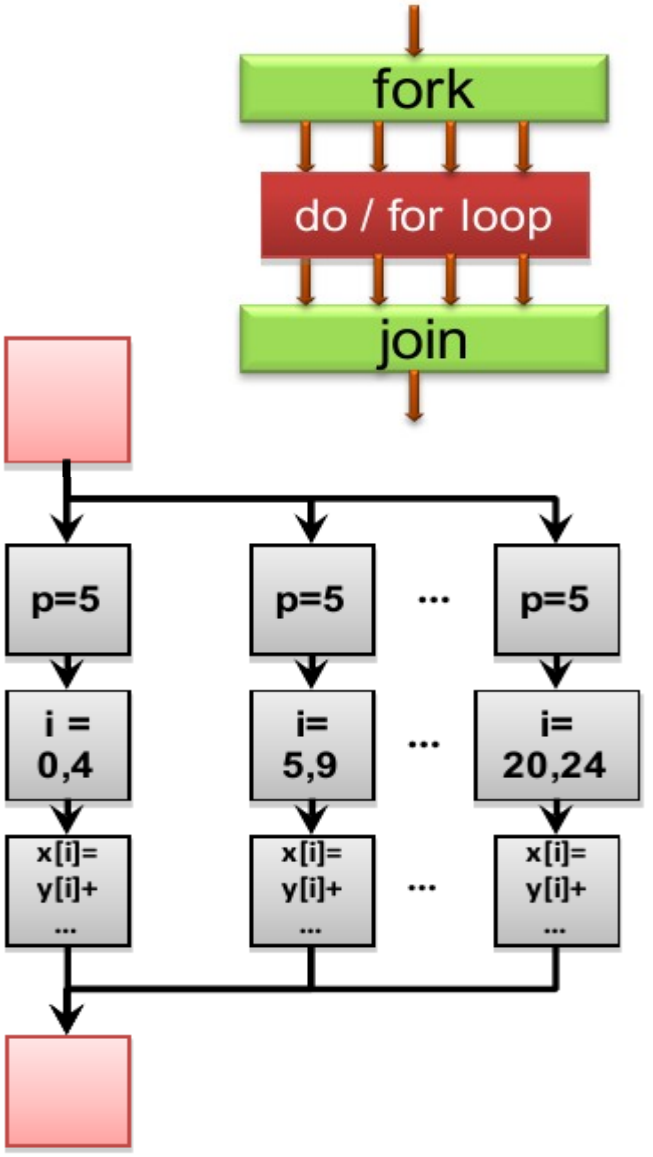
See “A beginner's guide to supercomputing” – Sterling & Anderson

- *do/for* directive helps share iterations of a loop between a group of threads.

- If *nowait* is specified then the threads do not wait for synchronization at the end of a parallel loop.

- The *schedule* clause describes how iterations of a loop are divided among the threads in the team.

```
#pragma omp parallel private(p)
{
  p=5;
  #pragma omp for
     for (i=0; i<25; i++)
         x[i]=y[i]+p*(i+3);
      …
  …
} /* omp end parallel */
```



Format (Fortran/C/C++)

See https://computing.llnl.gov/tutorials/openMP/

Fortran

```
!$OMP DO [clause ...]
         SCHEDULE (type [,chunk])
         ORDERED
         PRIVATE (list)
         FIRSTPRIVATE (list)
         LASTPRIVATE (list)
         SHARED (list)
         REDUCTION (operator | intrinsic : list)
         COLLAPSE (n)

   do_loop

!$OMP END DO   [ NOWAIT ]
```

C/C++

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

Example: “do loops”

Serial code

```cpp
double *x, *y, *z;
int n;
for(int i=0; i<n; ++i) {
  z[i] = x[i] + y[i];
}
```

C++

```fortran
real(kind=8) :: x(:), y(:), z(:)
integer      :: i, n
do i=1,n
  z(i) = x(i) + y(i)
end do
```

Fortran

Parallel code

→ **compiler handles loop bounds for you.**

→ **there is a compact single-line directive.**

→ **!$OMP DO (in Fortran)**

loop index variable i is private by default

```cpp
double *x, *y, *z;
int n, i;
#pragma omp parallel
{
  #pragma omp for
  for(i=0; i<n; ++i) {
    z[i] = x[i] + y[i];
  }
}
```

C++

```fortran
real(kind=8) :: x(:), y(:), z(:)
integer      :: i, n
!$omp parallel
!$omp do
do i=1,n
  z(i) = x(i) + y(i)
end do
!$omp end do
!$omp end parallel
```

Fortran

→ let's attempt to parallelize the integral $\pi = \int_0^1 dx \frac{4}{1+x^2}$

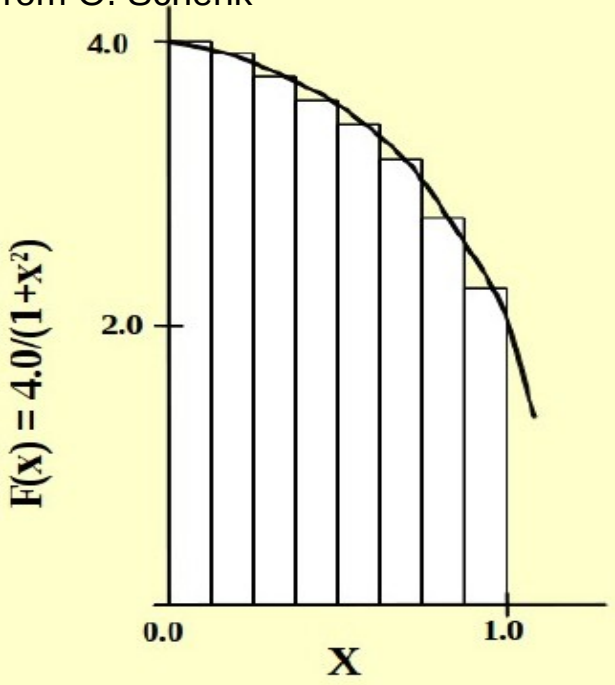
by using techniques learnt so far (“**summation the hard way**”).

→ **OSE2019/day3/code_day3/openmp/4.integration_pi.f90**

→ **OSE2019/day3/code_day3/openmp/4a.integration_pi.cpp**

Computing Pi – the hard way

Fig. from O. Schenk



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

```
#define _USE_MATH_DEFINES

const int num_steps = 500000000;

int main( void ){
    int i;
    double sum = 0.0;
    double pi  = 0.0;


    std::cout << "using " << omp_get_max_threads() << " OpenMP threads" << std::endl;

    const double w = 1.0/double(num_steps);

    double time = -omp_get_wtime();

    #pragma omp parallel firstprivate(sum)
    {
        #pragma omp for
            for (int i=0; i<num_steps; ++i)
            {
                double x = (i+0.5)*w;
                sum += 4.0/(1.0+x*x);
            }
        #pragma omp critical
            {
            pi= pi + w*sum;
            }
    }

    time += omp_get_wtime();

    std::cout << num_steps
              << " steps approximates pi as : "
              << pi
              << ", with relative error "
              << std::fabs(M_PI-pi)/M_PI
              << std::endl;
    std::cout << "the solution took " << time << " seconds" <<std::endl;
}
```

private(x) initializes a copy of the same variable for each thread but it could be anything

firstprivate(x) initializes a copy of the same variable for each thread with a specific value

Some clauses

SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent

STATIC
Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC
Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

AUTO
The scheduling decision is delegated to the compiler and/or runtime system.

NO WAIT / nowait: If specified, then threads do not synchronize at the end of the parallel loop.

Example – default work sharing

See OSE2019/day3/code_day3/openmp/openmp/4d.work.cpp

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
  {
    const int N = 100;
    int nthreads, threadid;
    int i;
    double a[N], b[N], c[N];

    //Initialize
    for (i = 0; i < N; i++){
     a[i] = 1.0*i;
     b[i] = 2.0*i;
    }

#pragma omp parallel shared(a,b,c,nthreads) private(i,threadid)
    {
    threadid = omp_get_thread_num();
    if (threadid ==0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("My threadid = %d\n", threadid);

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

    }  //join

  cout << "TEST c[99] = " << c[99] << endl;

  return 0;
  }
```

Example of loop work sharing

/code_day3/openmp$ ./4d.work-static.exec

```
My threadid = 2
My threadid = 1
My threadid = 3
Number of threads = 4
My threadid = 0
TEST c[99] = 297
```

Example – default work sharing

See OSE2019/day3/code_day3/openmp/4e.work-print.cpp

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
  {
    const int N = 20;
    int nthreads, threadid;
    int i;
    double a[N], b[N], c[N];

    //Initialize
    for (i = 0; i < N; i++){
     a[i] = 1.0*i;
     b[i] = 2.0*i;
    }

#pragma omp parallel shared(a,b,c,nthreads) private(i,threadid)
    {
    threadid = omp_get_thread_num();
    if (threadid ==0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("My threadid = %d\n", threadid);

#pragma omp for
    for (i = 0; i < N; i++){
        c[i] = a[i] + b[i];
        printf("Thread id = %d working on index %d\n", threadid,i);
    }

    }  //join

  cout << "TEST c[19] = " << c[19] << endl;

  return 0;
  }
```

Print out the thread ID and the index.
Use default scheduling.

```
./4e.work-static-print.exec
```

```
Number of threads = 4
My threadid = 0
Thread id = 0 working on index 0
Thread id = 0 working on index 1
Thread id = 0 working on index 2
Thread id = 0 working on index 3
Thread id = 0 working on index 4
My threadid = 2
Thread id = 2 working on index 10
Thread id = 2 working on index 11
Thread id = 2 working on index 12
Thread id = 2 working on index 13
Thread id = 2 working on index 14
My threadid = 3
Thread id = 3 working on index 15
Thread id = 3 working on index 16
Thread id = 3 working on index 17
Thread id = 3 working on index 18
Thread id = 3 working on index 19
My threadid = 1
Thread id = 1 working on index 5
Thread id = 1 working on index 6
Thread id = 1 working on index 7
Thread id = 1 working on index 8
Thread id = 1 working on index 9
TEST c[19] = 57
```

Example – static work sharing

See OSE2019/day3/code_day3/openmp/openmp/4f.work-static.cpp

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
  {
    const int N = 20;
    int nthreads, threadid;
    int i;
    double a[N], b[N], c[N];

    //Initialize
    for (i = 0; i < N; i++){
     a[i] = 1.0*i;
     b[i] = 2.0*i;
    }

    int chunk = 3;

#pragma omp parallel shared(a,b,c,nthreads) private(i,threadid)
    {
    threadid = omp_get_thread_num();
    if (threadid ==0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("My threadid = %d\n", threadid);

#pragma omp for schedule(static,chunk)
    for (i = 0; i < N; i++){
        c[i] = a[i] + b[i];
        printf("Thread id = %d working on index %d\n", threadid,i);
    }

    }  //join

  cout << "TEST c[19] = " << c[19] << endl;

  return 0;
  }
```

It assigns the first chunk to the first thread, the second to the second and so on. When it finishes the threads available it re-start from the beginning

Print out the thread id and the index.
Use static scheduling.

In this case you don't have a good work in term of balance

```
Number of threads = 4
My threadid = 0
Thread id = 0 working on index 0
Thread id = 0 working on index 1
Thread id = 0 working on index 2
Thread id = 0 working on index 12
Thread id = 0 working on index 13
Thread id = 0 working on index 14
My threadid = 2
Thread id = 2 working on index 6
Thread id = 2 working on index 7
Thread id = 2 working on index 8
Thread id = 2 working on index 18
Thread id = 2 working on index 19
My threadid = 1
Thread id = 1 working on index 3
Thread id = 1 working on index 4
Thread id = 1 working on index 5
Thread id = 1 working on index 15
Thread id = 1 working on index 16
Thread id = 1 working on index 17
My threadid = 3
Thread id = 3 working on index 9
Thread id = 3 working on index 10
Thread id = 3 working on index 11
TEST c[19] = 57
```

Example – dynamic work sharing

See OSE2019/day3/code_day3/openmp/4g.work-dynamic.cpp

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

int main()
  {
    const int N = 20;
    int nthreads, threadid;
    int i;
    double a[N], b[N], c[N];

    //Initialize
    for (i = 0; i < N; i++){
     a[i] = 1.0*i;
     b[i] = 2.0*i;
    }

    int chunk = 3;

#pragma omp parallel shared(a,b,c,nthreads) private(i,threadid)
    {
     threadid = omp_get_thread_num();
//      if (threadid ==0) {
//        nthreads = omp_get_num_threads();
//        printf("Number of threads = %d\n", nthreads);
//      }
//      printf("My threadid = %d\n", threadid);

#pragma omp for schedule(dynamic,chunk)
    for (i = 0; i < N; i++){
        c[i] = a[i] + b[i];
        printf("Thread id = %d working on index %d\n", threadid,i);
    }

    }  //join

  cout << "TEST c[19] = " << c[19] << endl;
```

Print out the thread id and the index.
Use dynamic scheduling.

Here it assigns the chunk to the first thread available. The thread in this case re-starts as soon as it finishes with the chunk before.

```
Thread id = 0 working on index 0
Thread id = 0 working on index 1
Thread id = 0 working on index 2
Thread id = 1 working on index 9
Thread id = 1 working on index 10
Thread id = 1 working on index 11
Thread id = 1 working on index 15
Thread id = 1 working on index 16
Thread id = 1 working on index 17
Thread id = 1 working on index 18
Thread id = 1 working on index 19
Thread id = 3 working on index 3
Thread id = 3 working on index 4
Thread id = 3 working on index 5
Thread id = 0 working on index 12
Thread id = 0 working on index 13
Thread id = 0 working on index 14
Thread id = 2 working on index 6
Thread id = 2 working on index 7
Thread id = 2 working on index 8
TEST c[19] = 57
```

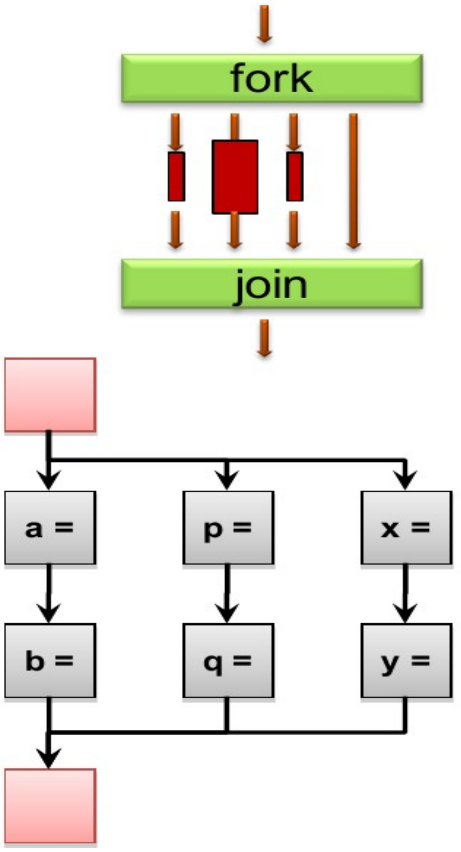
OpenMP worksharing “sections”

- **sections** directive is a non iterative work sharing construct.

- Independent section of code are nested within a **sections** directive.

- It specifies enclosed **section** of codes between different threads.

- Code enclosed within a **section** directive is executed by a thread within the pool of threads.




```
#pragma omp parallel private(p)
{
#pragma omp sections
{{  a=…;
   b=…;}
 #pragma omp section
 {  p=…;
   q=…;}
 #pragma omp section
 {  x=…;
   y=…;}
 } /* omp end sections */
} /* omp end parallel */
```

Fortran/C/C++ “sections”

Fortran

```
!$OMP SECTIONS [clause ...]
               PRIVATE (list)
               FIRSTPRIVATE (list)
               LASTPRIVATE (list)
               REDUCTION (operator | intrinsic : list)

!$OMP  SECTION

   block

!$OMP  SECTION

    block

!$OMP END SECTIONS  [ NOWAIT ]
```

C/C++

```
#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

  }
```

Example on “sections”

See OSE2019/day3/code_day3/openmp/5a.vec_add_sections.cpp

```
#include <iostream>
 #include <omp.h>
 #define N 1

 using namespace std;

int main ()
{
 int i;
 float a[N], b[N], c[N], d[N];

 /* Some initializations */
 for (i=0; i < N; i++) {
   a[i] = i * 1.5;
   b[i] = i + 22.35;
   }

 #pragma omp parallel shared(a,b,c,d) private(i)
   {

   #pragma omp sections nowait
     {

     #pragma omp section
     for (i=0; i < N; i++)
       {
       c[i] = a[i] + b[i];
       cout << "Section 1: hello from thread " << omp_get_thread_num() << " of " << omp_get_num_threads() << " index " << i << endl;

      }

     #pragma omp section
     for (i=0; i < N; i++)
      {
       d[i] = a[i] * b[i];
       cout << "Section 2: hello from thread " << omp_get_thread_num() << " of " << omp_get_num_threads() << " index " << i << endl;

      }

      #pragma omp section
      {
         cout << "Section 3: hello from thread " <<  omp_get_thread_num() << " of " << omp_get_num_threads()  << endl;
         cout << "This section 3 does nothing " << endl;
     }


     }  /* end of sections */

   }  /* end of parallel region */

 }
```

3 sections
>export OMP_NUM_THREADS = 2
./5.vec_add_sections.f90

>export OMP_NUM_THREADS = 3
>export OMP_NUM_THREADS = 4

→ **what do we observe?**

Reductions
→ e.g. summation the “easy way”

The **REDUCTION** clause performs a reduction on the variables that appear in the list.

→ **reduction(op:list)**

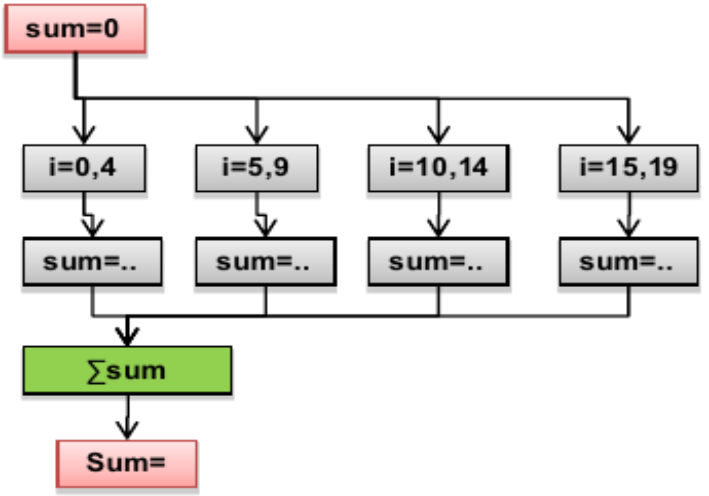
e.g. reduction(+:pi),
reduction(max:Maxval)

A private copy for each list variable is created for each thread.

At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

```
#pragma omp parallel for        \
     default(shared) private(i) \
     schedule(static,chunk)     \
     reduction(+:result)

    for (i=0; i < n; i++)
        result += (a[i] * b[i]);
```



Example: Reduction

> OSE2019/day3/code_day3/openmp/6a.integration_pi_reduction.cpp
(play with OMP_NUM_THREADS & timing)

```
#include <iostream>
#include <cmath>

#include <omp.h>

#define _USE_MATH_DEFINES

const int num_steps = 500000000;

int main( void ){
    int i;
    double sum = 0.0;
    double pi  = 0.0;

    std::cout << "using " << omp_get_max_threads() << " OpenMP threads" << std::endl;

    const double w = 1.0/double(num_steps);

    double time = -omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<num_steps; ++i) {
        double x = (i+0.5)*w;
        sum += 4.0/(1.0+x*x);
    }

    pi = sum*w;

    time += omp_get_wtime();

    std::cout << num_steps
              << " steps approximates pi as : "
              << pi
              << ", with relative error "
              << std::fabs(M_PI-pi)/M_PI
              << std::endl;
    std::cout << "the solution took " << time << " seconds" <<std::endl;
}
```

Questions?

1. Advice – RTFM
https://en.wikipedia.org/wiki/RTFM

2. Advice – http://lmgtfy.com/
http://lmgtfy.com/?q=open+mp