HARVARD
Faculty of Arts and Sciences
RESEARCH COMPUTING

Introduction to OpenMP

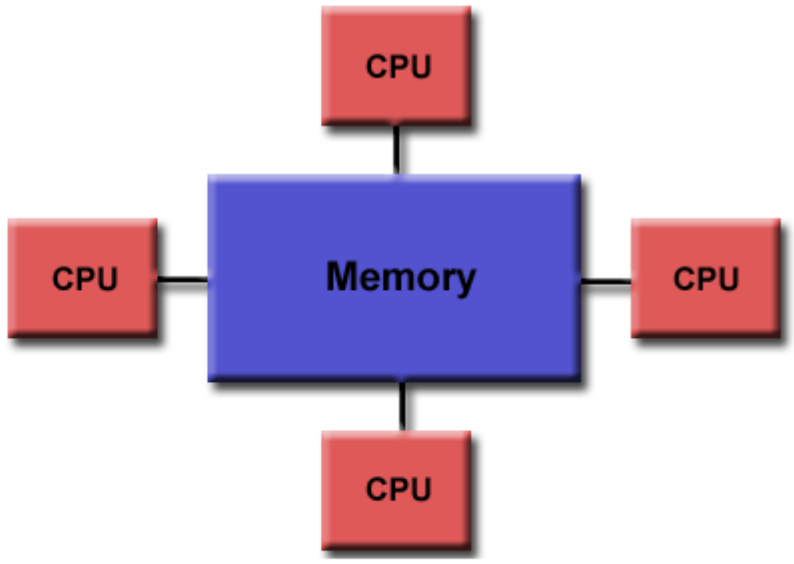
Paul Edmon
ITC Research Computing Associate

Overview

- Threaded Parallelism
- OpenMP Basics
- OpenMP Programming
- Benchmarking

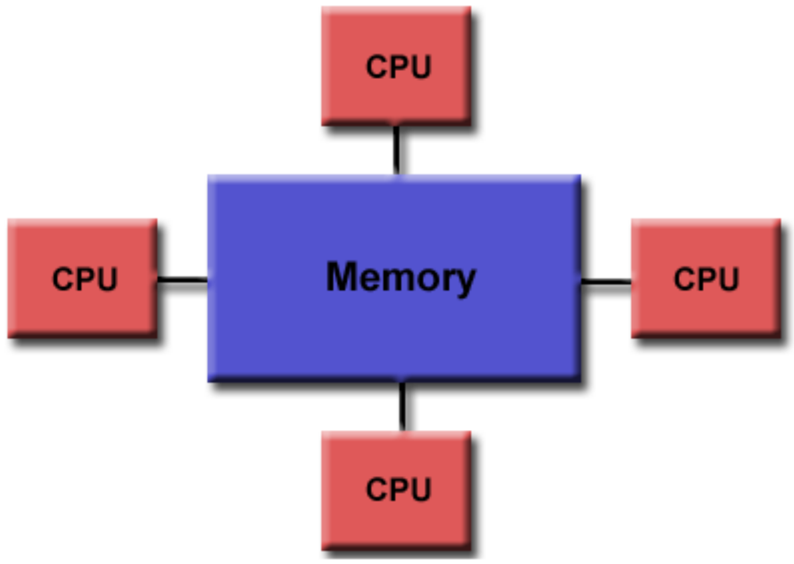
Threaded Parallelism

- Shared Memory
- Single Node
- Non-uniform Memory Access (NUMA)
- One thread per core



Threaded Languages

- PThreads
- Python
- Perl
- OpenCL/CUDA
- OpenACC
- OpenMP



OpenMP Basics

What is OpenMP?

- OpenMP (Open Multi-Processing)
  - Application Program Interface (API)
  - Governed by OpenMP Architecture Review Board

- OpenMP provides a portable, scalable model for developers of shared memory parallel applications

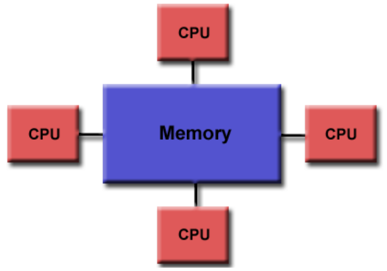
- The API supports C/C++ and Fortran on a wide variety of architectures

Goals of OpenMP

HARVARD
Faculty of Arts and Sciences
RESEARCH COMPUTING

- Standardization
  - Provide a standard among a variety shared memory architectures / platforms
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
- Lean and Mean
  - Establish a simple and limited set of directives for programming shared memory machines
  - Significant parallelism can be implemented by just a few directives
- Ease of Use
  - Provide the capability to incrementally parallelize a serial program
- Portability
  - Specified for C/C++ and Fortran
  - Most majors platforms have been implemented including Unix/Linux and Windows
  - Implemented for all major compilers

OpenMP Programming Model



Shared Memory Model: OpenMP is designed for multi-processor/core, shared memory machines

Thread Based Parallelism: OpenMP programs accomplish parallelism exclusively through the use of threads

Explicit Parallelism: OpenMP provides explicit (not automatic) parallelism, offering the programmer full control over parallelization

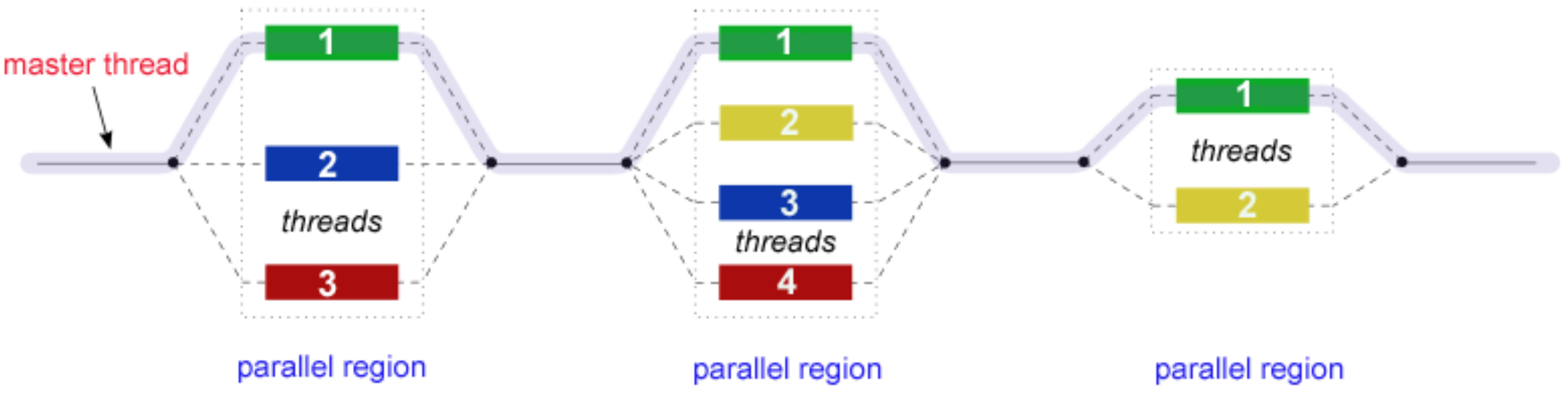
Compiler Directive Based: Parallelism is specified through the use of compiler directives embedded in the C/C++ or Fortran code

I/O: OpenMP specifies nothing about parallel I/O. It is up to the programmer to ensure that the I/O is conducted correctly in the context of the multi-threaded program

MPI: OpenMP can interoperated with MPI to create a hybrid model of parallelism

Fork-Join Model

HARVARD Faculty of Arts and Sciences RESEARCH COMPUTING



- All OpenMP programs begin as a single tread – the master thread. The master thread executes sequentially until the first parallel region is encountered
- **FORK:** The master thread then creates a team of parallel threads
- The statements in the program that are enclosed by the parallel region construct are executed in parallel among the team threads
- **JOIN:** When the team threads complete the statements in the parallel region, they synchronize and terminate leaving the master thread
- Note that starting the destroying threads is expensive in OpenMP so it is best to start the threads once and destroy them once.

Components of OpenMP

HARVARD
Faculty of Arts and Sciences
RESEARCH COMPUTING

The OpenMP API is comprised of three components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

The application developer decides how to employ these components. In the simplest case, only a few of them are needed.

Compiler Directives (Pragma)

- Compiler directives (aka pragmas) appear as comments in the source code and are completely ignored by compilers unless you tell them otherwise – usually by specifying appropriate compiler flags

- Compiler directives are used for various purposes, e.g.,
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations among threads
  - Synchronization of work among threads

- Compiler directives have the following syntax:
  - *sentinel directive-name [clause,…]*

```
Fortran: !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PHI)
C/C++: #pragma omp parallel default(shared) private(beta, phi)
```

Run-Time Library Routines

- OpenMP includes several run-time library routines

- These routines are used for various purposes such as:
  - Setting and querying the number of threads
  - Querying threads' unique identifier (thread ID)
  - Querying the thread pool size

```
FORTRAN: INTEGER FUNCTION GET_NUM_THREADS()

C/C++:      #include<omp.h>
            int omp_get_num_threads(void)
```

Environment Variables

- OpenMP provides several environment variables for controlling execution of parallel code at run-time

- Note that all of these variables can also be set in the code itself or via code inputs.

- These environment variables can be used for
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Enabling / disabling dynamic threads

- Setting OpenMP threads depends upon the shell you use:

```
csh/tcsh: setenv OMP_NUM_THREADS 8

sh/bash:  export OMP_NUM_THREADS=8
```

Compiling OpenMP Programs

Compiler/Platform	Compiler	Flag
Intel	icc icpc ifort	-openmp
GNU	gcc g++ g77 gfortran	-fopenmp

Intel: ifort –o omp_test.x omp_test.f90 –openmp

GNU: gfortran –o omp_test.x omp_test.f90 –fopenmp

Running OpenMP on Odyssey

(1) Compile your code, e.g.,

```
ifort –o omp_code.x omp_code.f90 –ompenmp –O2
```

(2) Prepare a batch-job submission script

```
#!/bin/bash
#SBATCH -J omp_job
#SBATCH -o slurm.out
#SBATCH -e slurm.err
#SBATCH -p general
#SBATCH --mem=1750
#SBATCH -c 8
#SBATCH -N 1
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun -c $SLURM_CPUS_PER_TASK ./omp_test.x
```

(3) Submit the job to the queue

```
sbatch omp_test.run
```

HARVARD
Faculty of Arts and Sciences
RESEARCH COMPUTING

OpenMP Programming

OpenMP Directives Fortran

Fortran Directives Format:

- All Fortran OpenMP directives must begin with a sentinel
- The accepted sentinels depend upon the type of Fortran source
- **!OMP, COMP, *$OMP**
- Comments can not appear on the same line as a directive
- Several Fortran OpenMP directives come in pairs

!$OMP *directive*

  [structured block of code]

!$OMP end *directive*

OpenMP Directives C/C++

HARVARD
Faculty of Arts and Sciences
RESEARCH COMPUTING

C/C++ Directive Format:

- All C/C++ directives begin with **#pragma omp**
- Case sensitive
- Directives follow conventions of the C/C++ standards for compiler directives
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block

```
#pragma omp directive
{

   [structured block of code]

}
```

Parallel Region Construct

HARVARD
Faculty of Arts and Sciences
RESEARCH COMPUTING

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct

Fortran

```
!$OMP PARALLEL [clause ...]
              IF (scalar_logical_expression)
              PRIVATE (list)
              SHARED (list)
              DEFAULT (PRIVATE |SHARED |
NONE)
              FIRSTPRIVATE (list)
              REDUCTION (operator: list)
              COPYIN (list)
              NUM_THREADS (scalar-integer-
expression)

   block

!$OMP END PARALLEL
```

C/C++

```
#pragma omp parallel [clause ...]
                    if (scalar_expression)
                    private (list)
                    shared (list)
                    default (shared | none)
                    firstprivate (list)
                    reduction (operator: list)
                    copyin (list)
                    num_threads (integer-
expression)


   structured_block
```

Data Scope Attribute Clauses

- Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables

- Because OpenMP is based upon the shared memory programming model, most variables are shared by default

- Global variables include:
  - **Fortran:** COMMON blocks, SAVE variables, MODULE variables
  - **C:** File scope variables, static

- Private variables include:
  - Loop index variables
  - Stack variables in subroutines called from parallel regions

Data Scope Attribute Clauses

- **PRIVATE** clause declares variables in its list to be private to each thread

  FORTRAN: PRIVATE (list)
  C/C++: private (list)

- **SHARED** clause declares variables in its list to be shared among all threads in the team

  FORTRAN: SHARED (list)
  C/C++: shared (list)

- **DEFAULT** clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region

  FORTRAN: DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
  C/C++: default (shared | none)

Important Runtime Routines

- OMP_SET_NUM_THREADS
  - Sets the number of threads for the application
- OMP_GET_NUM_THREADS
  - Polls the current setting for number of threads
- OMP_GET_THREAD_NUM
  - Tells you which thread number you are
- OMP_GET_WTIME
  - Timing routine

Example: Parallel Region Fortran

```
program hello
      implicit none
      integer(4):: nthreads
      integer(4):: tid
      integer(4):: omp_get_num_threads
      integer(4):: omp_get_thread_num
!$omp parallel private(tid)
      tid = omp_get_thread_num()
      write(6,*) "Hello World from thread =", tid
      if ( tid == 0 ) then
            nthreads = omp_get_num_threads()
            write(6,*) "Number of threads =", nthreads
      end if
!$omp end parallel
end program hello
```

Example: Parallel Region C/C++

```
#include <iostream>
#include <omp.h>
using namespace std;
int main () {
      int nthreads;
      int tid;
#pragma omp parallel private(tid)
      {
            tid = omp_get_thread_num();
            cout << "Hello World from thread = " << tid << endl;
            if (tid == 0){
                  nthreads = omp_get_num_threads();
                  cout << "Number of threads = " << nthreads << endl;
            }
      }
}
```

Work Sharing Constructs

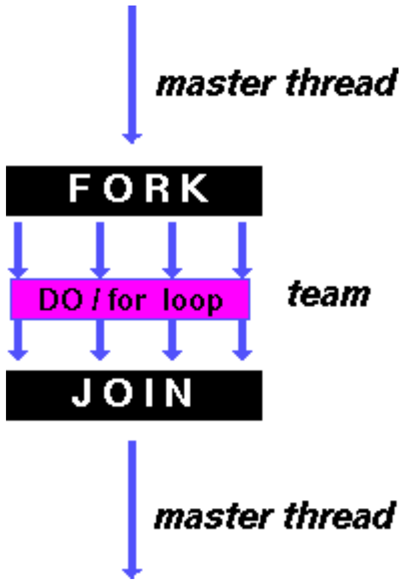
- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct

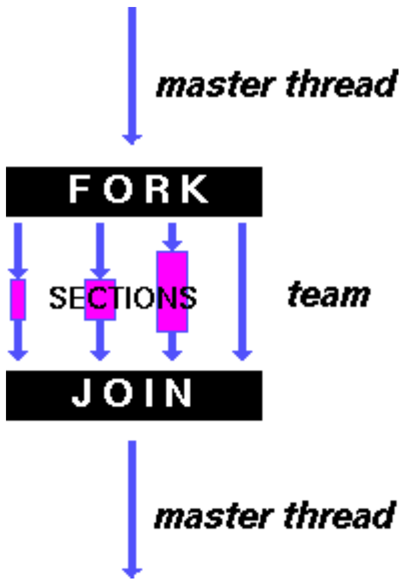
Types of Work Sharing Constructs

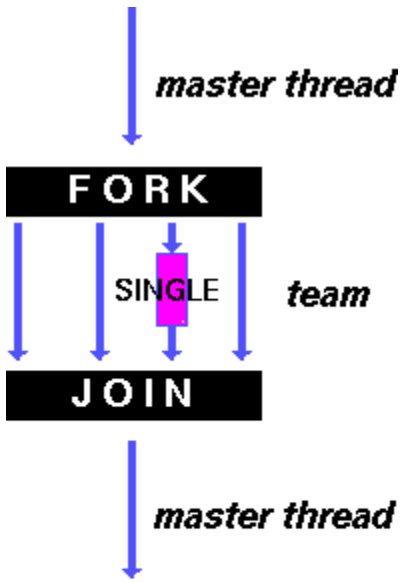
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism"

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







DO/FOR Directives

The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor

This is the easiest, fastest, and most efficient way to parallelize your code.

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 ...]
          schedule (type [,chunk])
          ordered
          private (list)
          firstprivate (list)
          lastprivate (list)
          shared (list)
          reduction (operator: list)
          collapse (n)
          nowait


  for_loop
```

Example: DO/FOR Directive Fortran

```
program vec_add_do
        implicit none
        integer(4) :: chunk, i
        integer(4), parameter :: n = 1000
        integer(4), parameter :: chunksize = 100
        real(4) :: a(n), b(n), c(n)
        do i = 1, n
                a(i) = i * 1.0
                b(i) = a(i)
        end do
        chunk = chunksize
!$omp parallel shared(a,b,c,chunk) private(i)
!$omp do schedule(dynamic,chunk)
        do i = 1, n
                c(i) = a(i) + b(i)
        end do
!$omp end do nowait
!$omp end parallel
end program vec_add_do
```

Example: DO/FOR Directive C/C++

```
#include <iostream>
#include <omp.h>
using namespace std;
#define CHUNKSIZE 100
#define N     1000
int main(){
        int i, chunk;
        float a[N], b[N], c[N];
        for ( i = 0; i < N; i++ )
                a[i] = b[i] = i * 1.0;
        chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
        {
#pragma omp for schedule(dynamic,chunk) nowait
                for ( i = 0; i < N; i++ )
                        c[i] = a[i] + b[i];
        }
        return 0;
}
```

Synchronization Constructs

- Since each thread is independent they can run at different speeds and thus threads may complete different sections at different times and get out of sync.
- OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads
- The BARRIER directive synchronizes all threads in the team
- When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier

```
FORTRAN: !$OMP BARRIER

C/C++:         #pragma  omp   barrier
```

HARVARD
Faculty of Arts and Sciences
RESEARCH COMPUTING

Benchmarking

Benchmarking

- Check top and see if your code is using the number of threads you set.
  - The process should be using number of threads x 100% of load
  - Underloaded applications are caused by thread contention or thread starvation.
- Run a scaling test
  - Take the same amount of work and divide it between 1, 2, 4, 8, etc. threads.
  - Ideal scaling would be that the amount of time it takes to do work will half every time you double the number of threads.
- After you complete your scaling test look at results and set thread count at the point where you still get appreciable performance gains due to parallelization.

HARVARD
Faculty of Arts and Sciences
RESEARCH COMPUTING

Research Computing Help

https://rc.fas.harvard.edu

Office Hours: Wednesdays noon-3pm

38 Oxford Street, 2nd Floor Conference Room

Sections Directive

- 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

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 ...]
              private (list)
              firstprivate (list)
              lastprivate (list)
              reduction (operator: list)
              nowait
 {
 #pragma omp section   newline

   structured_block

 #pragma omp section   newline

   structured_block

 }
```

Example: Sections Directive Fortran

```
PROGRAM VEC_ADD_SECTIONS
    INTEGER N, I
    PARAMETER (N=1000)
    REAL A(N), B(N), C(N), D(N)
    DO I = 1, N
     A(I) = I * 1.5
     B(I) = I + 22.35
    ENDDO
!$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)
!$OMP SECTIONS
!$OMP SECTION
    DO I = 1, N
      C(I) = A(I) + B(I)
    ENDDO
!$OMP SECTION
    DO I = 1, N
      D(I) = A(I) * B(I)
    ENDDO
!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL
    END
```

Example: Sections Directive C/C++

```
#include <omp.h>
#define N     1000
main ()
{
int i;
float a[N], b[N], c[N], d[N];
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];
    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];
    }  /* end of sections */
  }  /* end of parallel section */
}
```

Reduction Clause

- The REDUCTION clause performs a reduction on the variables that appear in its list
- 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

FORTRAN: REDUCTION *(operator | intrinsic: list)*

C/C++: reduction *(operator: list)*

Example: Reduction Clause Fortran

```
PROGRAM DOT_PRODUCT
      INTEGER N, CHUNKSIZE, CHUNK, I
      PARAMETER (N=100)
      PARAMETER (CHUNKSIZE=10)
      REAL A(N), B(N), RESULT
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = I * 2.0
      ENDDO
      RESULT= 0.0
      CHUNK = CHUNKSIZE
!$OMP  PARALLEL DO
!$OMP& DEFAULT(SHARED) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)
!$OMP& REDUCTION(+:RESULT)
      DO I = 1, N
        RESULT = RESULT + (A(I)  * B(I))
      ENDDO
!$OMP  END PARALLEL DO
      PRINT *, 'Final Result= ', RESULT
      END
```

Example: Reduction Clause C/C++

```
#include <omp.h>
main ()  {
int   i, n, chunk;
float a[100], b[100], result;
n = 100;
chunk = 10;
result = 0.0;
for ( i=0; i < n; i++ )
  {
  a[i] = i * 1.0;
  b[i] = i * 2.0;
  }
#pragma omp parallel for        \
  default(shared) private(i)    \
  schedule(static,chunk)        \
  reduction(+:result)
  for ( I = 0; i < n; i++ )
    result = result + (a[i] * b[i]);
printf("Final result= %f\n",result);
}
```