COMP90025 Parallel and Multicore Computing

OpenMP

Lachlan Andrew

School of Computing and Information Systems
The University of Melbourne

2019 Semester II

Overview

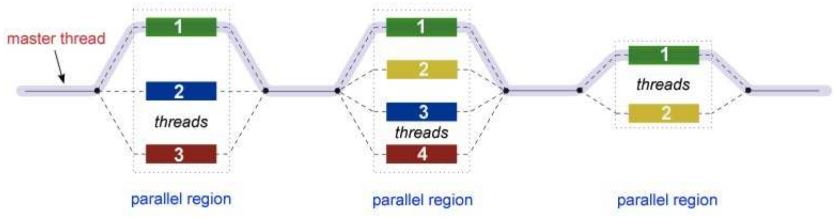
- What is OpenMP?
- Compiler Directives
- Run-time Library Routines
- Environment Variables
- C/C++ general code structure
- OpenMP Directives
    - SCHEDULE (Dynamic vs Static)
    - ORDERED clause
    - COLLAPSE clause
- Synchronization and sharing variables
- Memory Consistency and flush
- Data environment

What is OpenMP?

- The OpenMP is an Application Programming Interface (API) for parallel computing on multiprocessor *shared memory* machines.
- It *abstracts* the different thread implementations used in differing systems and *optimizes* thread usage according to the kinds of parallel operations that are required.
- The OpenMP API uses the *fork-join model* of parallel execution and is best suited to *large array-based applications*, althoughit can also be used for solving general parallel programs.
- One of three programming frameworks we will consider, in addition to the Message Passing Interface (MPI) and OpenCL for GPUs

OpenMP Programming Model

- Thread Based Parallelism
- Explicit Parallelism
- Fork-Join Model: All OpenMP programs begin as a single process: the master thread. The master thread then creates a team of parallel threads at parallel region. Then the team threads synchronize and terminate.



Isn't this model restrictive?

- POSIX threads (`pthreads`) are much more flexible
  - No need to keep joining
  - Ideal when each thread has a well-defined, different task
  - e.g., one running the GUI, one performing calculation, one loading / saving
  - (Ever noticed that you can't scroll with a document when you are choosing a filename to save as?)
- Fleibility requires the programmer to manage more details
  - Locks for accessing common data
  - Synchronization between cooperating threads
- Like the "rails" in Ruby on Rails
- If your program is suited to this model, the model helps.

OpenMP API Overview

- The OpenMP API is comprised of three distinct components. As of version 4.0:
  - 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

- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads
- Compiler directives have the following syntax for C/C++:

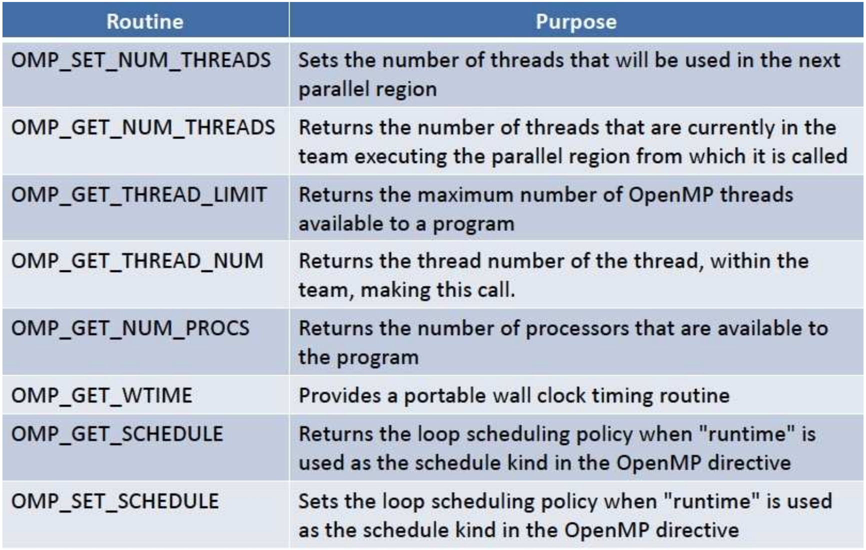
```
#pragma omp parallel default(shared) private(beta)
```

Run-time Library Routines

- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Setting and querying nested parallelism
  - Querying if in a parallel region, and at what nesting level
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution
- Note that for C/C++, you usually need to include the `<omp.h>` header file.

```
int omp_get_num_threads(void);
```

OpenMP Run-time Library Routines

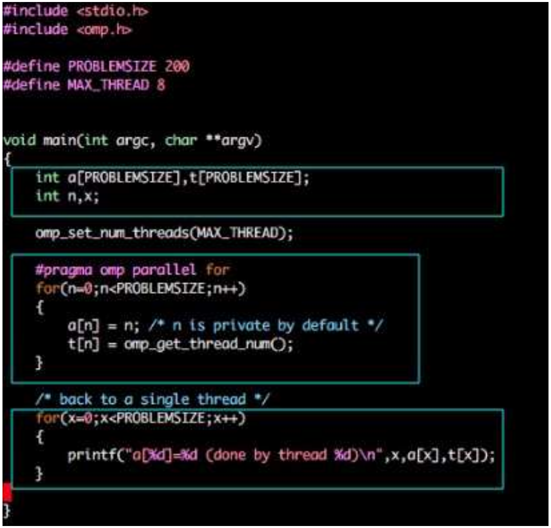


Routine	Purpose
OMP_SET_NUM_THREADS	Sets the number of threads that will be used in the next parallel region
OMP_GET_NUM_THREADS	Returns the number of threads that are currently in the team executing the parallel region from which it is called
OMP_GET_THREAD_LIMIT	Returns the maximum number of OpenMP threads available to a program
OMP_GET_THREAD_NUM	Returns the thread number of the thread, within the team, making this call.
OMP_GET_NUM_PROCS	Returns the number of processors that are available to the program
OMP_GET_WTIME	Provides a portable wall clock timing routine
OMP_GET_SCHEDULE	Returns the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_SET_SCHEDULE	Sets the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive

C/C++ general code structure

```
#include <omp.h>
main ()
{
int var1, var2, var3;
/* Serial code . . . */
/* Beginning of parallel section. Fork a team of th
/* Specify variable scoping */
#pragma omp parallel private(var1, var2) shared(var
{
        /* Parallel section executed by all threads
        /* Other OpenMP directives . */
        /* Run-time Library calls . */
        /* All threads join master thread and disba
}
/* Resume serial code . . . */
}
```

C/C++ general code structure



```
#include <stdio.h>
#include <omp.h>

#define PROBLEMSIZE 200
#define MAX_THREAD 8


void main(int argc, char **argv)
{
    int a[PROBLEMSIZE],t[PROBLEMSIZE];
    int n,x;

    omp_set_num_threads(MAX_THREAD);

    #pragma omp parallel for
    for(n=0;n<PROBLEMSIZE;n++)
    {
        a[n] = n; /* n is private by default */
        t[n] = omp_get_thread_num();
    }

    /* back to a single thread */
    for(x=0;x<PROBLEMSIZE;x++)
    {
        printf("a[%d]=%d (done by thread %d)\n",x,a[x],t[x]);
    }

}
```

A simple OpemMP program – example1.c

```c
#include <stdio.h>
#define SIZE 10

int main ()
{
    double value[SIZE];
    int i;

    #pragma omp parallel for num_threads (10)
    for (i = 0; i < SIZE; i++) {
        value[i] = i;
        printf ("%g\n", value[i]);
    }
}
```

Compiling and running the program

- `$ ssh` *your_login_id*`@spartan.hpc.unimelb.edu.au`
- `$ gcc -fopenmp example1.c -o example1`
- `$ sinteractive` # *brief shell on compute node for testing*
- `$ ./example1`
- Output: 1 0 7 6 8 5 9 4 3 2

Running in parallel on Spartan

- Create a SLURM file telling the system how to run your job, such as

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --time=0-12:00:00

# Load required modules
module load Python/3.5.2-intel-2016.u3

# Launch multiple process python code
echo "Searching␣for␣mentions"
time mpiexec -n 8 python3 twitter_search_541635.py -i
echo "Searching␣for␣topics"
time mpiexec -n 8 python3 twitter_search_541635.py -i
```

- `$ slurm` ⟨*your_slurm_file.slurm*⟩
- This schedules your job to execute, rather than running it now
- The wait can be frustratingly long (days if the system is busy)

(`https://dashboard.hpc.unimelb.edu.au/getting_started/`)

Environment variables

- Did you notice that example.c specified the number of threads?
- You may want to debug your code on your own computer, and run it on Spartan
- You can specify the number of threads on each platform without changing the code
- Remove `numthreads (10)` from the code
- Set the environment variable `export OMP_NUM_THREADS=10`
- = number of threads created when the master thread hits this spot
- Other environment variables:

OMP_DYNAMIC	Adjust number of thread dynamically
OMP_NESTED	Allow non-master threads to spawn their own
OMP_THREAD_LIMIT	Max number of simultaneous threads for the
OMP_STACKSIZE	Memory of each thread for local variables

Environment Variables

- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop interations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy
- Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which shell you use. E.g. for sh/bash:

```
export OMP_NUM_THREADS=8
```

OpenMP Directives

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

```
#pragma omp parallel [clause [clause ...]]
structured_block
```

where clause is one of

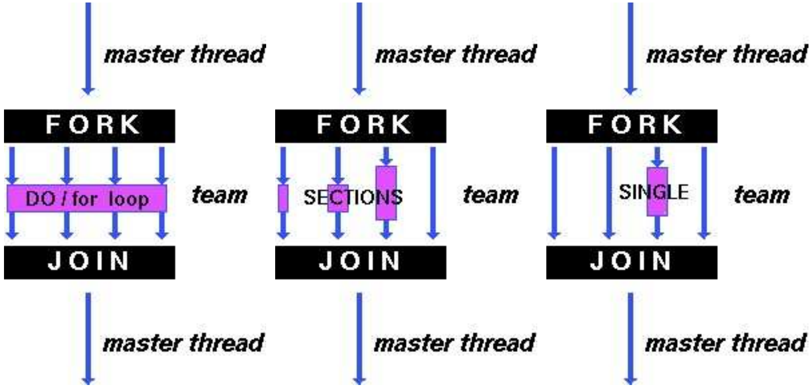
```
if (scalar_expression)
private (list) shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
num_threads (integer-expression)
```

OpenMP Directives

- 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 an *implied barrier* at the end of a work sharing construct, but no implied barrier upon entry to a work-sharing construct.
    - ★ No thread may process the instruction after the construct until all have finished the construct.
    - ★ Common synchronisation construct. (See also Rendezvous in Plan 9, mutual exclusion etc.)
- Do / for
  SECTIONS
  SINGLE

OpenMP Directives

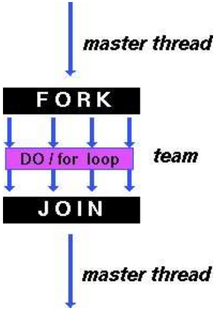
Work-Sharing Constructs



OpenMP Directives

Work-Sharing Constructs - 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 serially on a single processor.



```
#pragma omp for [clause ...]
```

OpenMP Directives

Work-Sharing Constructs - DO / for Clauses

- SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. Types are STATIC, DYNAMIC, GUIDED, RUNTIME or AUTO
- nowait: If specified, then threads do not synchronize at the end of the parallel loop.
- ORDERED: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- COLLAPSE: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause.

SCHEDULE

- 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.
- GUIDED: Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.
- RUNTIME: The scheduling decision is deferred until runtime by the environment variable `OMP_SCHEDULE`.
- AUTO: The scheduling decision is delegated to the compiler and/or runtime system.

Code Example and Exercise

Go to directory `.../openmp/`
check out examples `helloworld.c` `multiply.c` `s_example.c`

1. Use run-time library routine to print out runtime of the program. Using `omp_get_wtime()`
2. Change schedule construct on `s_example.c`. E.g. static, dynamic. Change chunk size
3. Compare the runtime of static, dynamic schedule.
4. Write your own parallel version of a serial program `pi_serial.c` (under the same directory)

Dynamic vs Static Example - Static

```
#define THREADS 4
#define N 16

int main ( ) {
  int i;
  #pragma omp parallel for schedule(static)
        num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* wait for i seconds */
    sleep(i);
    printf("Thread␣%d␣has␣completed␣iteration␣%d.\n
        omp_get_thread_num( ), i);
  }
  printf("All␣done!\n");
  return 0;
}
```

Dynamic vs Static Example - Dynamic

```
#define THREADS 4
#define N 16

int main ( ) {
  int i;
  #pragma omp parallel for schedule(dynamic)
        num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* wait for i seconds */
    sleep(i);
    printf("Thread␣%d␣has␣completed␣iteration␣%d.\n
        omp_get_thread_num( ), i);
  }
  printf("All␣done!\n");
  return 0;
}
```

Dynamic vs Static

- Dynamic scheduling is better when the iterations may take very different amounts of time.
- However, there is some overhead to dynamic scheduling.
- After each iteration, the threads must stop and receive a new value of the loop variable to use for its next iteration.

OpenMP Directives

Work-Sharing Constructs - DO / for Clauses

- SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. Types are STATIC, DYNAMIC, GUIDED, RUNTIME or AUTO
- nowait: If specified, then threads do not synchronize at the end of the parallel loop.
- ORDERED: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- COLLAPSE: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause.

ORDERED clause

The ordered construct is used with loops and is useful for printing results in order.

```
#pragma omp parallel for ordered
for (i=0;i<10;i++)
{ a[i]=func(i);
    #pragma omp ordered
        print("a[%d]␣is␣%d",i,a[i]);
}
```

In the example, the array elements will be printed in order, rather than arbitrarily according to which threads get to the print statement first.

COLLAPSE clause

Use the OpenMP collapse-clause to increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread.

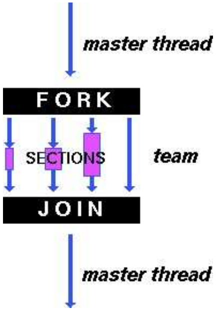
```
#pragma omp parallel for collapse(2)
    for (i = 0; i < imax; i++) {
        for (j = 0; j < jmax; j++) {
            a[ j + jmax*i] = 1.;
        }
  }
```

Especially useful when imax is smaller than number of threads.

OpenMP Directives

Work-Sharing Constructs - SECTIONS Directive

specifies that the enclosed section(s) of code are to be divided among the threads in the team.



```
#pragma omp sections [nowait]
    #pragma omp section
      structured_block
```

OpenMP Directives

Work-Sharing Constructs - SECTIONS Directive

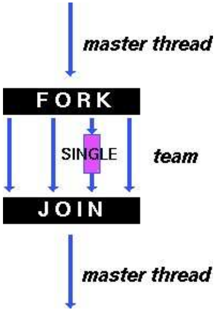
Parallel sections can be used to specify arbitrary work in different threads.

```
#pragma omp parallel sections
{
        #pragma omp section
        a[5]=a[5]*2;
        #pragma omp section
        a[1]=a[1]*2;
        #pragma omp section
        a[8]=a[8]*2;
}
```

OpenMP Directives

Work-Sharing Constructs - SINGLE Directive

specifies that the enclosed code is to be executed by only one thread in the team.



```
#pragma omp single [nowait]
      structured_block
```

OpenMP Directives

Work-Sharing Constructs - SINGLE Directive

```
#include <stdio.h>
#include <omp.h>

int main() {
   #pragma omp parallel num_threads(2)
   {
      char ch;
      #pragma omp single
      // Only a single thread can read the input.
      scanf("%c",&ch);

      // Multiple threads compute the results.
      printf("compute␣results\n");
    }
   return 0;
}
```

Synchronization and sharing variables

- Synchronization
  1. critical
  2. barrier
  3. atomic
  4. flush
  5. Example - parallel pi program
- Data environment
  1. stack and heap
  2. shared
  3. private
  4. firstprivate
  5. lastprivate
  6. reduction - parallel pi program
- Dijkstra Shortest-Path Algorithm

Synchronization:critical

A critical region applies to all current threads in the program, not just the threads in the current team. It restricts executions of the critical block to one thread at a time.

```
#pragma omp parallel shared(X,Y)
{
    #pragma omp critical
        Y=func(++X,Y);
}
```

Synchronization:barrier

The barrier directive has no statement associated with it. The barrier ensures that all threads in the team have reach the barrier before any thread is allowed to continue beyond the barrier.

```
#pragma omp parallel
{
        /* some work is done */
        #pragma omp barrier
        /* all previous work is done,
        now continue */
}
```

Synchronization:atomic

The atomic directive assures that updates to a variable are free from conflict. For example:

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

The kinds of updates are limited to a common arithmetic. A flush for the variable b is implicit on entry to and exit from the atomic region.

Memory Consistency and flush

- The compiler may store x in a register, and update the memory holding x only at certain points.
- In between such updates, since the memory location for x is not written to, the cache will be unaware of the new value, which thus will not be visible to other threads. This is important for shared variables.
- OpenMP takes a relaxed consistency approach, meaning that it forces updates to memory (flushes) at all synchronization points

Memory Consistency and flush

If two threads wish to communicate via a shared memory variable then the following must be ensured by the programmer:

- The value is written to the variable by the first thread.
- The variable is flushed by the first thread.
- The variable is flushed by the second thread.
- The value is read from the variable by the second thread.

Incorrect example of flush

```
Assuming a=b=0:
thread 1                    thread 2
b=1                         a=1
flush(b)                    flush(a)
flush(a)                    flush(b)
if (a==0) then              if (b==0) then
  critical section             critical section
```

The critical section, which should only be executed by one of the threads, is not safe. The compiler can reorder the flush(b) in thread 1 to be after the critical section since the variable b is not used in the condition (and assuming it is not used in the critical section itself). Similarly for thread 2.

Correct example of flush

```
Assuming a=b=0:
thread 1                         thread 2
b=1                              a=1
flush(a,b)                       flush(a,b)
if (a==0) then                   if (b==0) then
  critical section                 critical section
```

The flush statements cannot be reordered and only one thread could enter the critical region. In this example, neither thread may be able to enter the critical region.

Data environment: stack and heap

There are two types of memory to store your data, you can store your data in either stack or heap.

- Local variables are stored in stack. Global variables, static local variables, or memory allocated using malloc are in heap. The stack is always reserved in a LIFO (last in first out) order, stack allocated values are "deleted" once you leave the scope.
- The heap size is much larger than the size of stack, therefore it is good to storage large variables in heap, e.g. use char *a = malloc(1024) instead of char a[1024]
- In a multi-threaded situation each thread will have its own completely independent stack, so the threads will copy the stack variables but they will share the heap variables.

Data environment:shared and private

- Variables in shared context are visible to all threads running in associated parallel regions.
- Variables in private context are hidden from other threads. Each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads.

Data environment:shared and private

```
int E1;                       /* shared static      */
void main (argvc,...) {       /* argvc is shared    */
   int i;                     /* shared automatic   */
   void *p = malloc(...);     /*memory allocated by*/
          /* malloc is accessible by all threads*/
   #pragma omp parallel private (p)
   {
       int b;                 /* private automatic*/
       static int s;          /* shared static*/
       #pragma omp for
       for (i = 0;...)
       {         /* i is private here because it*/
                 /* is the iteration variable */
         b = 1;  /* b is still private here! */
       }
   }
 }
```

Data environment:Firstprivate Clause

Firstprivate is a special case of private.
It initializes each private copy with the corresponding value from the master thread

```
int main(void)
{
  int i = 0;
  int x;
  x=44;
  #pragma omp parallel for firstprivate(x)
  for (int i = 0; i < 10; i++) {
        x += i;
         printf("%d\n", x);
  }
  printf("x␣is␣%d\n", x);
}
```

Data environment:Lastprivate Clause

Lastprivate passes the value of a private from the last iteration to a global variable.

```
int main(void)
{
  int i = 0;
  int x;
  x=44;
  #pragma omp parallel for firstprivate(x)
        lastprivate(x)
  for (int i = 0; i < 10; i++) {
        x += i;
        printf("%d\n", x);
  }
  printf("x␣is␣%d\n", x);
}
```

Data environment:reduction

A private copy for each list variable is created and initialized 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.

```
main(int argc, char *argv[])  {
int    i, n, chunk;
float a[100], b[100], result;
/* Some initializations */

#pragma omp parallel for default(shared) \
  private(i) reduction(+:result)

  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
  printf("Final␣result=␣%f\n",result);
}
```

The Algorithm

Dijkstra algorithm is for finding the shortest paths from vertex 0 to the other vertices in an N-vertex undirected graph.

Pseudocode

```
Done = {0} # vertices checked so far
NewDone = None # currently checked vertex
NonDone = {1,2,...,N-1} # vertices not checked
for J = 0 to N-1 Dist[J] = G(0,J) # initialize

for Step = 1 to N-1
find J so Dist[J] is min among all J in NonDone
transfer J from NonDone to Done
NewDone = J
for K = 1 to N-1
 if K is in NonDone
  # check if there is a shorter path from
  # 0 to K through NewDone than our best so far
  Dist[K]=min(Dist[K],Dist[NewDone]+G[NewDone,K]
```

Pseudocode

At each iteration, the algorithm finds the closest vertex J to 0 among all those not yet processed, and then updates the list of minimum distances to each vertex from 0 by considering paths that go through J. Two obvious potential candidate part of the algorithm for parallelization are the find J and for K lines.

Some fine tuning

```
#pragma omp parallel
{ int startv,endv, // start, end vertices for my thread
      step, // whole procedure goes nv steps
      mymv, // vertex which attains the min value in my chunk
      me = omp_get_thread_num();
  unsigned mymd; // min value found by this thread
  #pragma omp single
  { nth = omp_get_num_threads();  chunk = nv/nth; }
   startv = me * chunk; endv = startv + chunk - 1;
   for (step = 0; step < nv; step++) {
      #pragma omp single
      { md = largeint; mv = 0; }
      findmymin(startv,endv,&mymd,&mymv);
      // update overall min if mine is smaller
      #pragma omp critical
      { if (mymd < md)
          { md = mymd; mv = mymv; } }
      #pragma omp barrier
      // mark new vertex as done
      #pragma omp single
      { notdone[mv] = 0; }
     // now update my section of mind
     updatemind(startv,endv);
    #pragma omp barrier
    }
    }
```

```
Int *mymins; //(mymd,mymv) for
             //each thread;
```

```
mymins = malloc(2*nth*sizeof(int));
findmymin(startv,endv,&mymd,&mymv);
mymins[2*me] = mymd;
mymins[2*me+1] = mymv;
#pragma omp barrier
// mark new vertex as done
#pragma omp single
{ notdone[mv] = 0;
 for (i = 1; i < nth; i++)
   if (mymins[2*i] < md) {
      md = mymins[2*i];
      mv = mymins[2*i+1];
   }
 }
// now update my section of mind
updatemind(startv,endv);
```