Plan

Last time

- Guassian elimination
- Intro to pthreads

Today

- Intro to OpenMP (Open Multi-Processing)

Based on:

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

Pthreads - `pthread_create`

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*thread_fun)(void *), void *arg);
```

`thread` is a handle for referncing the thread

`thread_fun` function executed by thread. It must return `void *` and take a single`( void *)` argument

thread terminates by calling `pthread-exit(NULL)`

Typical use

```
pthread_t thrs[NUM_THREADS];
err = pthread_create(&thrs[t], NULL, PrintHello, (void *) t);
pthread-exit(NULL);
```

Compile with the `-lpthread` flag.

Pthreads - `pthread_join`

`pthread join(thr_id,status)`

blocks the calling thread until thread `thr_id` terminates (synchronization with a single thread)

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

err = pthread_create(&thread[t], &attr, SomeWork, (void *)t);

pthread_attr_destroy(&attr);
rc = pthread_join(thread[t], &status);
pthread_exit(NULL);
```

Pthreads - `pthread_mutex_t`

Serialization of execution of a fragment of code.

```
pthread_mutex_t mutexsum;
pthread_mutex_init(&mutexsum, NULL);

pthread_mutex_lock (&mutexsum);
  global_sum += mysum;
pthread_mutex_unlock (&mutexsum);

pthread_mutex_destroy(&mutexsum);
```

Pthreads - pthread_barrier

Synchronization of multiple threads

```
int pthread_barrier_init(pthread_barrier_t *barrier,
           const pthread_barrierattr_t *attr, unsigned int count);

pthread_barrier_t my_barrier;
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

pthread_barrier_init(&my_barrier,NULL,2);

/* in ptheard_function */
pthread_barrier_wait(&my_barrier);

/* in main */
pthread_barrier_destroy(&my_barrier);
```

Pthreads -

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>

pthread_mutex_t mutex_norm = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t CurRow_norm = PTHREAD_MUTEX_INITIALIZER;
int  Norm = 0, row, col, **Mat, CurRow;

void * doMyWork(int myId) {
  int   cRow, cCol, rowSum = 0;
  while (1){
    pthread_mutex_lock(&CurRow_norm); {
      if (CurRow >= row) {
         pthread_mutex_unlock(&CurRow_norm);
         return(0);
      }
      cRow = CurRow; CurRow++;
    }
    pthread_mutex_unlock(&CurRow_norm);

    rowSum = 0;
```

```
    for (cCol = 0; cCol < col; cCol++)
      rowSum += abs(Mat[cRow][cCol]);
    pthread_mutex_lock(&mutex_norm);
      if (Norm < rowSum)
        Norm = rowSum;
    pthread_mutex_unlock(&mutex_norm);
  }
}

void *main(int argc, char *argv[]) {

  FILE  *fp;
  pthread_t  *threads;
  int  counter, cRow, cCol, NumThreads;

  if (argc != 2) {
    printf(" Missing Arguments: Number of Threads.\n");
    return(0);
  }
  fp = fopen("Norm_data.txt", "r");
```

```
fscanf(fp, "%d", &row); fscanf(fp, "%d", &col);
printf("\n Row:%d  Col:%d.", row, col);

NumThreads = atoi(argv[1]);
if (NumThreads < 1) {
  printf("\n Number of threads must be greater than zero. Aborting ...\n");
  return(0);
}
threads = (pthread_t *) malloc(sizeof(pthread_t) * NumThreads);

Mat = (int **) malloc(sizeof(int) * row);
for (counter = 0; counter < row; counter++)
  Mat[counter] = (int *) malloc(sizeof(int) * col);

for (cRow = 0; cRow < row; cRow++)
  for (cCol = 0; cCol < col; cCol++)
    fscanf(fp, "%d", &Mat[cRow][cCol]);

CurRow = 0;
for (counter = 0; counter < NumThreads; counter++)
```

```
    pthread_create(&threads[counter], NULL,  (void *) doMyWork, (void *) (counter))

  for (counter = 0; counter < NumThreads; counter++)
    pthread_join(threads[counter], NULL);
  printf("\n Row Wise partitioning. Infinity Norm: %d\n", Norm);
}
```

Pthreads -

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>

pthread_mutex_t *mutex_Res;
pthread_mutex_t mutex_col;
int Norm = 0;
int row, col, **Mat, *Res;
int CurCol=0;

void *doMyWork(int myId){
  int cRow, cCol;
  while (1){
    pthread_mutex_lock(&mutex_col); {
      cCol=CurCol;
      if (CurCol >= col){
        pthread_mutex_unlock(&mutex_col);
        return(0);
      }
    CurCol++;
    }
```

```
    pthread_mutex_unlock(&mutex_col);
    for(cRow=0; cRow<row; cRow++){
      pthread_mutex_lock(&mutex_Res[cRow]);
        Res[cRow] += abs(Mat[cRow][cCol]);
      pthread_mutex_unlock(&mutex_Res[cRow]);
    }
  }
} /* end of doMywork */

void main(int argc, char*argv[]){
  pthread_t *threads;
  int counter, cRow, cCol, NumThreads;

  FILE *fp = fopen("Norm_data.txt", "r");
    if (fp == NULL) {
      printf("Error opening file!\n");
      exit(1);
    }
    if (argc != 2 ){
      printf("\n Missing Arguments: Number of Threads.\n");
```

```
  return;
}
fscanf(fp, "%d", &row); fscanf(fp, "%d", &col);
printf("\n Row:%d  Col:%d.", row, col);

NumThreads = atoi(argv[1]);

threads = (pthread_t*)malloc(sizeof(pthread_t)*NumThreads);

Mat = (int**)malloc(sizeof(int)*row);
Res = (int*)malloc(sizeof(int)*row);
mutex_Res = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t)*row);

for (counter=0; counter<row; counter++){
  Mat[counter]=(int*)malloc(sizeof(int)*col);
  Res[counter]=0;
}
for (cRow=0; cRow<row; cRow++)
  for (cCol=0; cCol<col; cCol++)
    fscanf(fp, "%d", &Mat[cRow][cCol]);
```

```
    for (counter=0; counter<NumThreads; counter++)
      pthread_create(&threads[counter], NULL, (void*)doMyWork, (void *) counter);

    for (counter=0; counter<NumThreads; counter++)
      pthread_join(threads[counter], NULL);

    for (cRow=0; cRow<row; cRow++)
      if (Res[cRow]>Norm)
        Norm=Res[cRow];

     printf("\n Col Wise partitioning. Infinity Norm: %d.\n", Norm);
     return;
}
```

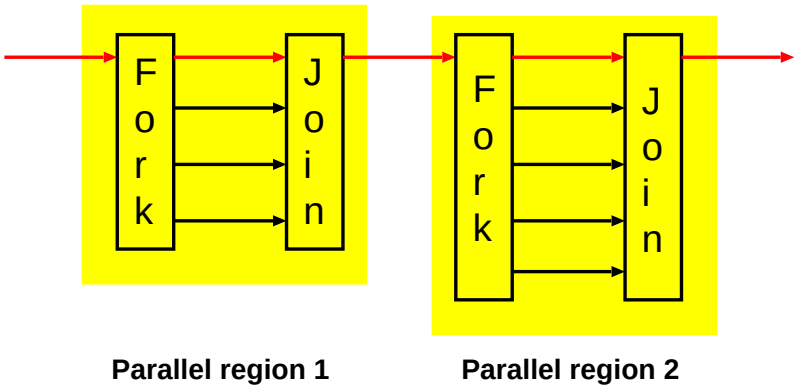
pthreads are cumbersome - difficult to maintain correctnes.

OpenMP is meant to provide relief from the difficulties present in pthreads

- a simple and limited set of directives

OpenMp combines serial and parallel code into a single sequential task graph where each task can be a concurrent task.

OpenMP



OpenMP programs begin as a single master thread.

1. master performs some sequential calculation, and next
2. forks a team of concurrent threads
3. when concurrent threads complete execution, they synchronize and terminate
4. master may start from (1) or terminates
5. concurrent threads can be placed inside other concurrent threads.

Team of concurrent threads executes either

- **parallel regions** - data parallelism, or
- parallel **serial regions** - task parallelism

OpenMP - parallel region

- The master thread executes sequentially until a **parallel region** construct is encountered.
- The master thread then creates a team of threads which execute the **same code**
- When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread
  - no need for explicit synchronization.

`#include<omp.h>` must be present

All OpenMP compiler directives start with

`#pragma omp`

The directive is followed by additional info which tells the compiler how to manage threads.

Compilation

`gcc filename.c -fopenmp` (plus other flags)

How many threads?

- `lscpu` will tell you how many (physical) CPUs are available. The system will use this as the number of threads.
- `omp_get_num_threads();` will tell how many threads will be used.
- `export setenv OMP_NUM_THREADS = 4` will set the number of threads to 4
- `omp_set_num_threads(8);` will overwrite and set the number of threads to 8
- threads are numbered from 0 (master) to `num_threads -1`

OpenMP - parallel region

```
#pragma omp parallel [some clauses ...]
```

When distributing work among concurrent threads it is assumed that there is no data dependency

It is programmer's responsibility to make sure that dependency does not exist in the current parallel region

OpenMP - parallel region

omp_fork.c

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

main ()  {
int nthreads;
/* Fork concurrent threads, each having a private tid */
  #pragma omp parallel {
    /* Obtain and print thread id */
    int tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    if (tid == 0) {      /* Only master thread does this */
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
}
```

Shared - All variables defined outside a parallel region are by default shared.

Private - All variables declared inside a parallel region are duplicated as many times as there are threads in the team.

- Their values outside the parallel region are not defined.

`parallel` clause list includes:

- `shared (variable list)` - variables are shared across all threads
- `private (variable list)` - variables local to each thread
- `default(shared)` - all variables are shared
- `default(none)` - signals error if not all variables are specied
- `num_threads(integer expression)` - # threads to create

and more..

Parallel region - clauses

```
/* parallel and sum are evaluated outside parallel block */

# pragma omp parallel if (parallel==1) num_threads(8) \
private (a) shared (b,sum) default(none)
```

- `if (parallel == 1) num_threads(8)`
- `private (a) shared (b,sum)`
- `default(none)` all variables must be specified as shared or private, (only `a, b, sum` are used in parallel region)

`default(shared)` all variables are shared

omp_error.c

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

main ()  {
int nthreads, tid;
#pragma omp parallel default(none) private(tid) num_threads(4)
  {
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
}
```

will generate error as the scope of **nthreads** is not specified

OpenMP - private

`omp_private.c`

```
void ex() {
/* 8 threads in parallel regions */
omp_set_num_threads(8);
int i = 0, N = 8, cout = 6;
#pragma omp parallel default(none) private(i) shared(N,cout)
/* default(none) means we must specify the scope for i and cout */
   for(i=0; i<N; i++) {
      cout = omp_get_thread_num(); }
   printf("After parallel region");
}
```

Each copy of `i` is internaly incremented from 0 to 7,
after the parallel region finishes the original `i = 0`,
`cout` is some number between 0 and 7

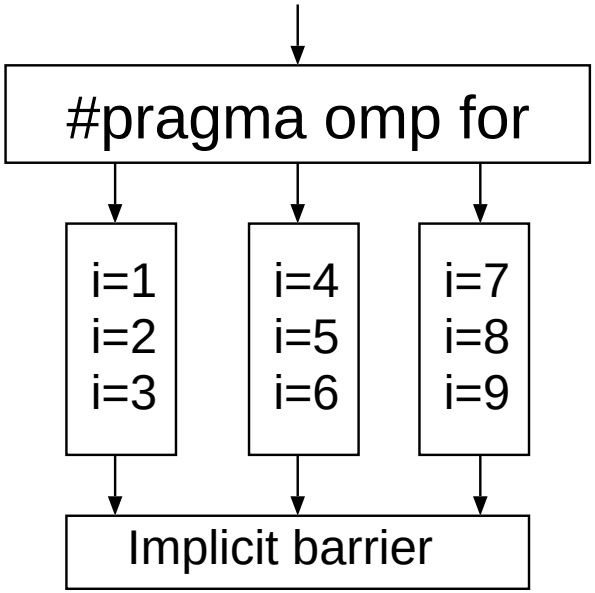
Work sharing constructs divide the execution of the parallel region among the members of the team

There are two main constructs

- `omp for` - data parallelism, sections of data assigned to different threads
- `omp section` - task parallelism

OpenMP - parallel region

```
#pragma omp for
for (i = 0; i < 9; i++) {
   c[i] = a[i] + b[i];  }
```



(`default` schedule).

OpenMP - parallel for

`omp for` loop distributes iterations across the thread team (data parallelism)

```
omp for [clause ...] \
         schedule (type [,chunk])
         private (list)
         firstprivate (list)
         lastprivate (list)
         shared (list)
         reduction (operator: list)
         nowait
```

Implicit barrier at end of for loop

OpenMP - firstprivate

`firstprivate(list of variables)`

- The `firstprivate` variable is initialized (once per thread) to the values of the original variables from the outside of parallel block
- for example `sum = 0.0`.

OpenMP - lastprivate

`lastprivate(list of variables)`

- in the `for` directive, value from the last iteration is copied back to the original variable (but we do not know by which thread in the team)
- in the `sections` construct - value from the last (lexicographical) section is copied back to the original variable (but we do not know by which thread in the team)

OpenMP - lastprivate

omp_lastprivate.c

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

int main(void){
  int i;
  int x;
  x = 1;
  #pragma omp parallel for firstprivate(x) num_threads(4)
  for(i=0;i<=10;i++){
    x = x+i;
    printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
  }
  printf("x is %d\n", x);
}
```

Iterations in `for` loop are divided among threads

- `static` - Iterations are divided into groups of size `chunk` and then statically assigned to threads in round robin distribution.

  If chunk is not specified, the iterations are evenly and contiguously divided among the threads.

  – Low overhead but may cause load imbalance if differengt iterations take different amount of time.

OpenMP - loop scheduling `static`

Assume 4 threads

```
#pragma omp parallel for schedule(static,16)
 for(i=; i<128; i++) {
   c[i] = a[i] + b[i]; }

thread0: i=0:15, 64:79
thread1: i=16:31, 80:95
thread2: i=32:47, 96:111
thread3: i=48:63, 112:127
```

OpenMP - loop scheduling `static`

omp_workshare_loop.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE  20
#define N  100

int main (int argc, char *argv[]) {
  int nthreads, tid, i, chunk;
  float a[N], b[N], c[N];

  for (i=0; i < N; i++) /* Some initializations */
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;

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

OpenMP - loop scheduling `static`

```
    }
    printf("Thread %d starting...\n",tid);

    #pragma omp for schedule(static,chunk)
      for (i=0; i<N; i++) {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
      }
  }  /* end of parallel section */
}
```

OpenMP - `nowait`

`omp for` ends with an implicit barrier. Synchronization can be skipped with `nowait` clause.

```
#pragma omp parallel {
   #pragma omp for nowait
      for (i = 0; i < nmax; i++)
         if (id == c[i]) work_current(id);
   #pragma omp for
      for (i = 0; i < mmax; i++)
         if (id == p[i]) work_new(id);
}
```

Any thread that finished the first `omp for` can begin the second `omp for` without waiting for other threads to finish first loop. The second `omp for` ends with an implicit barrier.

OpenMP - loop scheduling `dynamic`

`dynamic` - Iterations are divided into groups of size `chunk`, and dynamically assigned to threads (first come first served).

- When a thread finishes one chunk, it is assigned another chunk.
- Higher overhead than for static but can reduce load imbalance.

OpenMP - loop scheduling dynamic

```
#include ....
#define CHUNK_SIZE 10
#define N          1000

main ()  {
int i, chunk = CHUNK_SIZE;
double a[N], b[N], c[N];

/* Initialize data */
   for (i=0; i < N; i++)
     a[i] = drand48(); b[i] = drand48();
   #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];
   }
}
```

OpenMP - loop scheduling `guided`

- `guided` - Iterations are divided into chunks of sizes which decrease with each iteration.

  If `chunk = 1`, the size of the initial chunk is proportional to

  `number_of_iterations/number_of_threads`

  Subsequent chunks are proportional to

  `number_of_iterations_remaining/number_of_threads`

For `num_threads = 2` and `N = 9999` we get
Thread 0 gets 5000 iterations. Iteration remaining 4999
Thread 1 gets 2500 iterations. Iteration remaining 2499
If Thread 1 finishes before Thread 0 it gets 1250 iterations with 1249 remaining, etc.

If `chunk = k (k>1)` the chunks do not contain fewer than k iterations.

`omp sections` - breaks work into separate tasks that may be executed concurrently (Note **s** at the end.)

- Must be inside a parallel region (`#pragma omp parallel`).
- `omp section` - defines a separate tasks defined by `omp section` (Note **no s** at the end.)

Each task is executed by a thread.

This implements **task parallelism**

OpenMP - parallel sections

```
#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];
  }
```

OpenMP - sections

sections
serialized

sections
parallelized

OpenMP - sections

The `sections` directive is a non-iterative construct.

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.

It is possible for a thread to execute more than one section.

omp_workshare_section.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N    100000

int main (int argc, char *argv[])
{
int i, tid;
float a[N], b[N], c[N], d[N], sum = 0.0, prod = 1.0;

/* Some initializations */
for (i=0; i<N; i++) {
  a[i] = 1/(i+1) + 0.1; b[i] = 1/(i+1) + 0.15; c[i] = d[i] = 0.0;
  }

#pragma omp parallel shared(a,b,c,d,sum,prod) private(i,tid) num_threads(4)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
```

OpenMP - sections

```
    printf("Number of threads = %d\n", omp_get_num_threads());
  printf("Thread %d starting...\n",tid);

  #pragma omp sections nowait {
    #pragma omp section {
      printf("Thread %d doing section 1\n",tid);
      for (i=0; i<N; i++)
        sum =  sum + a[i] + b[i];
      printf("Thread %d: sum = %f\n",tid,sum);
    }
    #pragma omp section {
      printf("Thread %d doing section 2\n",tid);
      for (i=0; i<N; i++)
        prod =  -prod/2 + a[i] + b[i];
      printf("Thread %d: prod = %f\n",tid,prod);
    }
  }  /* end of sections */
  printf("Thread %d done.\n",tid);
  } /* end of pragma parallel */
}
```

OpenMP - `master`

The `master` directive specifies a region that is executed only by the master thread

All other threads in the `parallel` region skip this section of code.

```
#pragma omp master
{
   structured_block
}
```

OpenMP - master

omp_getNumThr.c

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

int main() {
  int thread_id, nthreads;
  omp_set_num_threads(8);
  printf("main outside pragma 111 sees %d threads\n", omp_get_num_threads( ));
  #pragma omp parallel
    #pragma omp master
      printf("master in pragma 222 sees %d threads\n", omp_get_num_threads( ));

    printf("main outside pragma 333 sees %d threads\n", omp_get_num_threads( ));

   #pragma omp parallel num_threads(4) {
     #pragma omp master {
        nthreads = omp_get_num_threads();
        printf("main in pragma 444 sees %d threads \n", nthreads);
      }
      thread_id = omp_get_thread_num();
```

```
    nthreads = omp_get_num_threads();
    printf("Hello from thread %d, out of %d threads\n", thread_id, nthreads);
  }
  printf("main outside pragma sees %d threads\n", omp_get_num_threads( ));
}
```

OpenMP - `critical`

The `critical` directive specifies a segment of code that must be executed by only one thread at a time.

```
#pragma omp critical ( name ) \
   /* block of work */
```

If a thread enters the critical region all other threads attempting to execute it are blocked until the first thread exits that critical region.

The optional `name` defines a different critical region.

Regions with the same name are treated as one.

Unnamed critical sections are treated as one.

OpenMP - `critical`

```
#pragma omp parallel for shared(sum1,sum2)
   for(i = 0; i < n; i++){
     value1 = f1(a[i]); value2 = f2(a[i]);
     #pragma omp critical(name)
     {
       sum1 = sum1 + value1;
       /* some other stuff */
     }
     #pragma omp critical(name)
     {
       sum2 = sum2 + value2;
       /* some other stuff */
     }
  }
```

OpenMP - `atomic`

The `atomic` directive specifies that a specific memory location must be updated atomically.

- a mini-`critical` section.

The directive applies only to a single, immediately following statement.

```
#pragma omp parallel for shared(sum1,sum2)
{
   for(i = 0; i < n; i++){
     value1 = f1(a[i]);
     #pragma omp atomic
       sum1 = sum1 + value1;
     }
  }
```

`#pragma omp 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.

OpenMP - `barrier`

```
void worker( ) {
  int id = omp_get_thread_num( );
  printf("Thread %d starting!\n", id);
  /* threads taking different amounts of time */
  sleep(id);
  printf("Thread %d is done its work!\n", id);
  #pragma omp barrier
  printf("Thread %d is past the barrier!\n", id);
}
int main ( ) {
  # pragma omp parallel num_threads(THREADS)
    worker( );
}
```

OpenMP - integrtation by trapezoidal rule

```
#include ...

double f (double x) {
   return 4.0/(1.0 + x*x);}
/*   return x;}*/
double trapezoidal_rule( double x0, double x1, int N,
                         double (*f) (double))
{
   double int_f = 0.0;        /* accumulate sum of f(x0+i*h) */
   double h = (x1 - x0)/N;    /* step size                   */
   double x_i = x0;           /* running point               */
   double int_f = (f(x0) + f(x1))/2;
   int i; double x;
   for (i = 1; i<N; i++) {
      x_i += h;                /* next point                  */
      int_f += f(x_i);
   }
   int_f *= h;
   return int_f;
}
```

Trapezoidal rule - serial code

```
int main(int argc, char *argv[]) {
  double x0, x1;
  int N;
  x0 = atof(argv[1]); x1 = atof(argv[2]); N  = atoi(argv[3]);

  double int_f = trapezoidal_rule(x0,x1,N,f);
  printf("x0 = %10.4f, x1 = %10.4f, N = %d\n",x0,x1,N);
  printf("Integral = %14.10f\n",int_f);
  printf("Integration error = %14.10g\n",(M_PI-int_f)/M_PI);
  return 0;
}
```

Trapezoidal rule - parallel code

```
#include ...

double f (double x) {
   return 4.0/(1.0 + x*x);}
double *local_int_f;
double trapezoidal_rule( double x0, double x1, int N,
                         double (*f) (double))
{
   double int_f = 0.0;        /* accumulate sum of f(x0+i*h) */
   double h = (x1 - x0)/N;    /* step size                   */
   double x_i = x0;           /* running point               */
   double f_ends = (f(x0) + f(x1))/2;
   int i; double x;
   for (i = 1; i<N; i++) {
      x_i += h;                /* next point                  */
      int_f += f(x_i);
   }
   int_f *= h;
   return int_f;
}
```

Trapezoidal rule - parallel code

```
int main(int argc, char *argv[]) {
   double x0, x1, h, int_f = 0.0;
   int N, i;
   x0 = atof(argv[1]); x1 = atof(argv[2]); N  = atoi(argv[3]);
/* get or set # threads  */
   int no_thr = omp_get_max_threads();
/* allocate memory for partial results */
   local_int_f = (double*)malloc(sizeof(double)*no_thr);
   N = (N/no_thr)*no_thr;
   h = (x1-x0)/N;
  printf("No of Threads= %d\n",no_thr);
#pragma omp parallel
{
     int local_N = N/no_thr;
     int thr_id = omp_get_thread_num();
     double local_x0 = x0 + thr_id*local_N*h;
     double local_x1 = local_x0 + h*local_N;
     local_int_f[thr_id] = trapezoidal_rule(local_x0,local_x1,local_N,f);
  }
/* reduction step */
```

Trapezoidal rule - parallel code

```
  for (i = 0; i< no_thr; i++){
     int_f += local_int_f[i]; }
  printf("x0 = %10.4f, x1 = %10.4f, N = %d\n",x0,x1,N);
  printf("Integral = %14.10f\n",int_f);
  printf("Integration error = %14.10g\n",(M_PI-int_f)/M_PI);
  return 0;
}
```

`reduction (operator: list)`

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

OpenMP - binary tree addition

omp_reduction.c

```
#include <omp.h>

main ()  {
int   i, n = 1000, chunk = 10;
double a[1000], b[1000], result = 0.0;

for (i=0; i < n; i++) {
  a[i] = i * 1.0; b[i] = 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]); /* local sum */
}
printf("Final result= %f\n",result); /* global sum */
}
```

Variables in the `reduction` list must be named scalar variables.

They can not be array or structure type variables.

They must also be declared `shared` in the enclosing context.

`+, *, -, /, &&, ||`

Trapezoidal rule - parallel code

```
#include ....
double f (double x) {
   return 4.0/(1.0 + x*x);}
double trapezoidal_rule( double x0, double x1, int N,
                         double (*f) (double))
{
   double h = (x1 - x0)/N;    /* step size                    */
   double x_i = x0;           /* running point                */
   double int_f = (f(x0) + f(x1))/2;
   int i;
#pragma omp parallel for private(x_i) reduction(+:int_f)
  for(i=1; i<N; i++) {
    x_i = x0 + i*h;                /* next point                    */
    int_f += f(x_i);
  }
   int_f *= h;
   return int_f;
}
```

```
int main(int argc, char *argv[]) {
   double x0, x1, h, x_i, int_f;
   int N, i;
   x0 = atof(argv[1]); x1 = atof(argv[2]); N  = atoi(argv[3]);
/* reduction step */
int_f = trapezoidal_rule(x0,x1,N,f);
  printf("x0 = %10.4f, x1 = %10.4f, N = %d\n",x0,x1,N);
  printf("Integral = %14.10g\n",int_f);
  printf("Integration error = %14.10g\n",(M_PI-int_f)/M_PI);
  return 0;
}
```

OpenMP matrix multiply

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

#define DIM_I 100
#define DIM_J 100
#define DIM_K 100

int main (int argc, char *argv[]) {
int     t_id, n_threads, i, j, k, chunk;
double  a[DIM_I][DIM_J],
        b[DIM_J][DIM_K],
        c[DIM_I][DIM_K];

chunk = 10;                     /* set chunk size */
```

OpenMP matrix multiply

```
#pragma omp parallel shared(a,b,c,n_threads,chunk) \
                     private(t_id,i,j,k)
  {
  t_id = omp_get_thread_num();
  if (t_id == 0)
    {
    n_threads = omp_get_num_threads();
    printf("There are %d threads\n",n_threads);
    }
  }
```

```
#pragma omp for schedule (static, chunk)
for (i=0; i<DIM_I; i++)
  for (j=0; j<DIM_J; j++)
    a[i][j]= drand48();
#pragma omp for schedule (static, chunk)
for (i=0; i<DIM_J; i++)
  for (j=0; j<DIM_K; j++)
    b[i][j]= drand48();
#pragma omp for schedule (static, chunk)
for (i=0; i<DIM_I; i++)
  for (j=0; j<DIM_K; j++)
    c[i][j]= 0;

/*** Do matrix multiply sharing iterations on outer loop ***/
#pragma omp for schedule (static, chunk)
for (i=0; i<DIM_I; i++)
  {
  printf("Thread=%d got row=%d\n",t_id,i);
  for(j=0; j<DIM_K; j++)
    for (k=0; k<DIM_J; k++)
```

```
        c[i][j] += a[i][k] * b[k][j];
    }
/*** End of parallel region ***/
```