Parallel & Concurrent Programming: OpenMP

Emery Berger
CMPSCI 691W
Spring 2006
Outline

- Last time(s):
  - MPI – point-to-point & collective
    - Library calls
- Today:
  - OpenMP - parallel directives
    - Language extensions to Fortran/C/C++
Motivation

- Take vectors $\mathbf{a} & \mathbf{b}$ (100 ints)
- Distribute across all processors
- Each processor:
  - Compute sum of all $a[i] * b[i]$
  - Print overall sum

- MPI: Use **MPI_Scatter, MPI_Gather or MPI_Reduce**
  - MPI_Scatter/Gather
    (sendbuf, cnt, type, recvbuf, recvcnt, type, root, comm)
  - MPI_Reduce
    (sendbuf, recvbuf, cnt, type, op, root, comm)
MPI Solution

```c
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);

// Distribute a and b
MPI_Scatter (a, 100, MPI_INT, a1, 100 / size, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter (b, 100, MPI_INT, b1, 100 / size, MPI_INT, 0, MPI_COMM_WORLD);

// Multiply each chunk
for (int i = 0; i < 100/size; i++) {
    z += a[i] *b1[i];
}

// Reduce by summing
if (rank == 0) {z1 = new int[size]; }
MPI_Reduce (&z, &z, 1, MPI_INT, MPI_OP_PLUS, 0, MPI_COMM_WORLD);

// Output result
if (rank == 0) {
    cout << z << endl;
}
```
Ideal Solution

```cpp
int z = 0;
parallel for (i = 0; i < nProcessors; i++) {
    z += a[i] * b[i];
}
cout << z << endl;
```
OpenMP Solution

```c
int z = 0;
#pragma omp for
for (int i = 0; i < 100; i++) {
    z += a[i] * b1[i];
}
cout << z << endl;
```

- **OpenMP** `pragma` directives
  - Omit = sequential program
  - More declarative style
  - Add more pragmas for more efficiency
OpenMP Concepts

- Fork-join model
- One thread executes sequential code
- Upon reaching parallel directive:
  - Start new team of work-sharing threads
  - Wait until all done (usually barrier)
  - Can be nested!
- Apparent global shared memory but relaxed consistency model
Consistency

- Consistency = ordering of reads & writes
  - In same thread, across threads

- Most “intuitive” consistency model = sequential consistency (Lamport)
  - Behaves like some sequential execution
  - **BUT:** seriously limits parallelism
    - Must synchronize frequently
OpenMP Consistency

- OpenMP: consistency across flushes
  - Writes set of variables to memory
  - If two flushes have intersecting sets, flushes must be seen in some sequential order by all threads

```c
/* Announce that I am done with my work. The first flush
* ensures that my work is made visible before synch.
* The second flush ensures that synch is made visible.
*/

#pragma omp flush(work,synch)
synch[iam] = 1;
#pragma omp flush(synch)
```
Parallel Execution

- `#pragma omp parallel`  
  - Executes next chunk of code across all or some number of threads  
    - `num_threads(n)`  
  - Only “master thread” continues after parallel section completes
#include <omp.h>
int main()
{
    omp_set_dynamic(1);

    #pragma omp parallel num_threads(10)
    {
        /* do work here */
    }
    return 0;
}
Parallel + nowait

```c
void a8(int n, int m, float *a, float *b, float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

        #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```

- Implicit barrier unless `nowait`
- Barrier = flush operation
Parallel + Memory

Memory model:
- Heap objects shared
- Stack objects private
  - Includes loop iterators
- unless indicated otherwise...

```c
void a1(int n, float *a, float *b)
{
    int i;

    #pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```
Parallel Example

```c
void subdomain(float *x, int istart, int ipoints)
{
    int i;

    for (i = 0; i < ipoints; i++)
        x[istart+i] = 123.456;
}

void sub(float *x, int npoints)
{
    int iam, nt, ipoints, istart;

    #pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt = omp_get_num_threads();
        ipoints = npoints / nt;    /* size of partition */
        istart = iam * ipoints;    /* starting array index */
        if (iam == nt-1)           /* last thread may do more */
            ipoints = npoints - istart;
        subdomain(x, istart, ipoints);
    }
}
Data-Sharing Attributes

- shared
- private
  - Each thread gets own private copy
  - Undefined value
- firstprivate
  - Copies in original value
- lastprivate
  - Copies out private value
Lastprivate Example

```c
void a30 (int n, float *a, float *b)
{
    int i;

    #pragma omp parallel
    {
        #pragma omp for lastprivate(i)
        for (i=0; i<n-1; i++)
            a[i] = b[i] + b[i+1];
    }

    a[i]=b[i];     /* i == n-1 here */
}
```
Threadprivate Example

- Can also declare variables as **always** thread-private

```c
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return(counter);
}
```
Reduce

- reduction
  - private value per thread
  - initialized “appropriately”
    - uses predefined operators
  - copies out to original
- reduction(+:a)
  - initializes a = 0
- reduction(*:1)
  - initializes a = 1
OpenMP Solution

```c++
int z = 0;
#pragma omp for reduction(+:z)
for (int i = 0; i < 100; i++) {
    z += a[i] * b1[i];
}
cout << z << endl;
```

- **OpenMP pragma directives**
  - Omit = sequential program
  - More declarative style
  - **Add more pragmas for more efficiency**
void a31_1(float *x, int *y, int n)
{
    int i, b;
    float a;

    a = 0.0;
    b = 0;

    #pragma omp parallel for private(i) shared(x, y, n) \  
    reduction(+:a) reduction(^:b)
    for (i=0; i<n; i++) {
        a += x[i];
        b ^= y[i];
    }
}
```c
#include <stdio.h>
#include <omp.h>

int main()
{
    int x;

    x = 2;
    #pragma omp parallel num_threads(2) shared(x)
    {
        if (omp_get_thread_num() == 0) {
            x = 5;
        } else {
            /* Print 1: the following read of x has a race */
            printf("1: Thread% d: x = %d\n", omp_get_thread_num(), x);
        }

        #pragma omp barrier

        if (omp_get_thread_num() == 0) {
            /* Print 2 */
            printf("2: Thread% d: x = %d\n", omp_get_thread_num(), x);
        } else {
            /* Print 3 */
            printf("3: Thread% d: x = %d\n", omp_get_thread_num(), x);
        }
    }

    return 0;
}
```
Master & Synchronization

- **master**
  - Always run by master thread

- **critical**
  - Declares critical section (one thread at a time)
  - Can add *names* for greater concurrency

- **barrier**

- **atomic**
  - Updated atomically (a++, a--, etc.)

- **ordered**
  - Executes loop body sequentially
Atomic Example

```c
void a16(float *x, float *y, int *index, int n)
{
    int i;

    #pragma omp parallel for shared(x, y, index, n)
    for (i=0; i<n; i++) {
        #pragma omp atomic
        x[index[i]] += work1(i);
        y[i] += work2(i);
    }
}
```
The End
Single Example

```c
void work1() {}
void work2() {}

void a10()
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Beginning work1.\n");

        work1();

        #pragma omp single
        printf("Finishing work1.\n");

        #pragma omp single nowait
        printf("Finished work1 and beginning work2.\n");

        work2();
    }
}```
int main()
{
    int iam, neighbor;

    #pragma omp parallel private(iam,neighbor) shared(work,synch)
    {
        iam = omp_get_thread_num();
        synch[iam] = 0;

        #pragma omp barrier
        /*Do computation into my portion of work array */
        work[iam] = fnl(iam);

        /* Announce that I am done with my work. The first flush
         * ensures that my work is made visible before synch.
         * The second flush ensures that synch is made visible.
         */

        #pragma omp flush(work,synch)
        synch[iam] = 1;
        #pragma omp flush(synch)
#pragma omp parallel for ordered schedule(dynamic)
for (i=lb; i<ub; i+=stride)
    work(i);
Copypin Example

```c
#pragma omp threadprivate(work, size, tol)

void a32( float t, int n )
{
    tol = t;
    size = n;
    #pragma omp parallel copyin(tol, size)
    {
    build();
    }
}
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)

void init(float a, float b) {
    #pragma omp single copyprivate(a,b,x,y)
    {
        scanf("%f %f %f %f", &a, &b, &x, &y);
    }
}
The End

Next time:
- OpenMP