High Performance Computing - Session 2

Shared Memory Programming

Jayanti Prasad
http://www.iucaa.ernet.in/ jayanti/

Inter-University Centre for Astronomy & Astrophysics
Pune, India (411007)

November 15, 2011

Plan of the Talk

- Introduction
  - What is `OpenMP` ?
  - The Hello World program !
  - Function calls
  - Parallel for
- Private and Shared variables
  - Private and Shared variables
  - First private and last private
- Sheduling
- Synchronization
  - Barrier
  - Critical
- Examples
- Problems

What is OpenMP ?

- OpenMP is a shared memory application programming interface (API).
- OpenMP is not a new programming language and OpenMP programs are written just other C or FORTRAN programs, with some added features.
- Since OpenMP is based on threading programming model so it is useful on shared memory systems in which we have more than one computing elements (cores) sharing the same main memory.
- There is very less communication overhead in data sharing between different threads so OpenMP is very fast.
- OpenMP provides tools, in the form of function calls, pragmas and clauses for data sharing and thread coordination.

The Hello World Program !

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<math.h>
4   #include<omp.h>
5
6   #define NUMTHREADS 4
7
8   int main(int argc, char *argv[]){
9     int myid, numthreads;
10
11    omp_set_num_threads(NUMTHREADS);
12
13  #pragma omp parallel
14    {
15
16      myid = omp_get_thread_num();
17
18      numthreads = omp_get_num_threads();
19
20      printf("Hello World from  %d of %d threads ! \n",myid,numthreads);
21
22    }
23    return(0);
24  }
```

Compilation

gcc demo1.c -fopenmp

Things you must do !

- Include the header file (line 4)
- Specify the number of threads (line 6)
- Set the number of threads (line 11)
- Use `OpenMP` pragma (line 13)
- Have a parallel block (between line 14 and 22)

Three important functions

omp_set_num_threads();
omp_get_get_thread_num();
omp_get_num_threads();

OpenMP Parallel For

```
1   #pragma omp parallel
2
3    for(i=0; i < n; i++) private(i,myid)
4     {
5
6       myid = omp_get_thread_num();
7
8       printf("Hello World from  %d of %d threads ! \n",myid,numthreads);
9
10    }
```

Note that 'go to' and 'exit' type of statements are not allowed inside a parallel loop.

OpenMP Clauses : private and shared

- Variables which are modified by every thread inside the for loop must be declared `private` .
- Every thread has it own copy of `private` variables.
- Loop controlling variable is always `private` and its value is undefined after the parallel loop has ended.
- Variable which have common values for all the threads are declared `shared` .
- By default all variables are `shared` variable.
- By declaring a variable `firstprivate` the private copies of the variables are initialized from the original variables existing before the construct.
- By declaring a variable `lastprivate` the value of the variable is set to the value which the last iteration of the loop.

firstprivate

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<math.h>
4   #include<omp.h>
5
6   int main(int argc, char *argv[]){
7     int numthreads,i,indx,myid;
8     if (argc < 2){
9       fprintf(stderr,"./a.out <numthreads>\n");
10      return(-1);
11    }
12
13    numthreads = atoi(argv[1]);
14    omp_set_num_threads(numthreads);
15    indx  = 3;
16  #pragma omp parallel firstprivate(indx) private(myid)
17  //#pragma omp parallel private(myid,indx)   // this is wrong !
18    {
19      myid  = omp_get_thread_num();
20      indx += myid;
21      printf("my id=%d indx=%d\n",myid,indx);
22    }
23    printf("After the parallel region: indx=%d\n",indx);
24    return(0);
25  }
```

lastprivate

```
1   include<stdio.h>
2   #include<stdlib.h>
3   #include<math.h>
4   #include<omp.h>
5
6   int main(int argc, char *argv[]){
7     int numthreads,i,n,indx,myid;
8     if (argc < 3){
9       fprintf(stderr,"./a.out <numthreads> <num points> \n");
10      return(-1);
11    }
12
13    numthreads = atoi(argv[1]);
14    omp_set_num_threads(numthreads);
15    n = atoi(argv[2]);
16
17  #pragma omp parallel for private (i) lastprivate(indx)
18  //#pragma omp parallel for private (i) private(indx) // this is wrong !
19
20  for(i=0; i < n; i++){
21     indx  = i;
22  }
23  printf("After the parallel region: indx=%d\n",indx);
24   return(0);
25  }
```

Scheduling

- The `schedule` clause is used to specify how the loop iterations are distributed amoung threads.
- In general the work distribution is done using "chunks" which are defined as contiguous, nonempty subsets of the iteration space.
- In case of `static` schedule (default) the chunks are assigned to the threads statically in a round-robin manner, in the order of the thread number. The last chunk to be assigned may have a smaller number of iterations. `You know which thread will get what`.
- In case of `dynamic` schedule the iterations are assigned to threads as the threads request them. The thread executes the chunk of iterations then requests another chunk until there are no more chunks to work on. `You do not know which thread will get what`.

Scheduling

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<math.h>
4   #include<omp.h>
5   int main(int argc, char *argv[]){
6     int numthreads,i,n,myid;
7
8     if (argc < 3){
9       fprintf(stderr,"./a.out <numthreads> <num points> \n");
10      return(-1);
11    }
12
13    numthreads = atoi(argv[1]);
14    omp_set_num_threads(numthreads);
15    n = atoi(argv[2]);
16
17  //#pragma omp parallel for schedule (static) private(i,myid)
18  #pragma omp parallel for schedule (dynamic) private(i,myid)
19    for(i=0; i < n; i++){
20
21      myid = omp_get_thread_num();
22
23      printf("I am thread %d and doing %d\n",myid,i);
24
25    }
26   return(0);
```

Example : N Body kernal

```
1   #pragma omp parallel for private (i,j,dr2,l) shared (npart,ndim,r,f)  schedule (static)
2   for(i=0; i < npart; i++){
3      for(j=0; j < npart; j++){
4        if (i !=j){
5         dr2 = eps2;
6         for(l=0; l < ndim; l++){
7           dr[l] = r[l+ndim*j]-r[l+ndim*i];
8           dr2   += dr[l] * dr[l];
9         }
10        for(l=0; l < ndim; l++){
11          f[l+ndim*i] += MG * dr[l]/(dr2*sqrt(dr2));
12        }
13       }
14      }
15    }
```

Syncronization

- **Barrier :** This makes sure that no thread moves ahead unless all threads have done the job i.e., reached a point. In most cases it is unnecessary.
- **Critical :** This provides means to ensure that multiple threads do not attempt to update the same shared data simultaneously. By declaring a block critical we can make sure that only thread executes it at a time.

Example : OpenMP Barrier

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<math.h>
4   #include<omp.h>
5   #include<sys/time.h>
6
7   int main(int argc, char *argv[]){
8     int numthreads,myid;
9
10    if (argc < 2){
11      fprintf(stderr,"./a.out <numthreads>\n");
12      return(-1);
13    }
14
15    numthreads = atoi(argv[1]);
16    omp_set_num_threads(numthreads);
17
18  #pragma omp parallel private(myid)
19    {
20      myid = omp_get_thread_num();
21      if (myid < omp_get_num_threads()/2 )
22        system("sleep 3");
23      printf("thread %d: before\n",myid);
24
25  #pragma omp barrier
26      printf("thread  %d: after\n",myid);
27
28    }
29    return(0);
30  }
```

Example : OpenMP Critical

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<math.h>
4   #include<omp.h>
5   int main(int argc, char *argv[]){
6     int i, num_threads,n,TID, sumLocal,sum,*a;
7     if (argc < 2){
8       fprintf(stderr,"./timer_omp <num threads> <num_points> \n");
9       return(-1);
10     }
11
12    num_threads = atoi(argv[1]);
13    n = atoi(argv[2]);
14    a = (int *)malloc(n*sizeof(int));
15
16    for(i=0; i < n; i++)
17      a[i] = i;
18
19    omp_set_num_threads(num_threads);
20
21    sum = 0;
22
23  #pragma omp parallel shared(n,a,sum) private(TID,sumLocal)
24    {
25      TID = omp_get_thread_num();
26      sumLocal = 0;
27  #pragma omp for
28      for (i=0; i<n; i++)
29        sumLocal += a[i];
30  #pragma omp critical (update_sum)
31      {
32        sum += sumLocal;
```

Examples

1. `hello_world_omp.c`
2. `scalar_product_omp.c`
3. `matrix_vector_omp.c`
4. `compute_pi_omp.c`
5. `prime_number_omp.c`
6. `nbody_kernal_omp.c`
7. `barrier_omp.c`
8. `critical_omp.c`
9. `first_private_omp.c`
10. `last_private_omp.c`
11. `schedule_omp.c`

Problems

1. Gravitaional force acting on a system of N bodies is given by:

$$\mathbf{f_i} = -\sum_{j=1;j\neq i}^{N} \frac{G(\mathbf{r_j} - \mathbf{r_i})}{(\epsilon^2 + |\mathbf{r_j} - \mathbf{r_i}|^2)^{3/2}} \tag{1}$$

   write a parallel program using `OpenMP` which can compute the force in parallel and compare the time taken by the serial and parallel code.

2. Show that for a power law model with $P(k) \propto k^n$ the mass varaince is given by $\sigma^2(r) \propto r^m$ and compute the value of m by doing the following neumerical integration using `OpenMP` :

$$\sigma^2(r) = \int_{k_{min}}^{k_{max}} \frac{dk}{k} \frac{k^3 P(k)}{2\pi^2} \left[3\left(\frac{\sin kr - kr\cos kr}{k^3 r^3}\right)\right]^2 \tag{2}$$

   try for different values of n in the range $(-3, 1)$.

Thank You !