CS 4230: Parallel Programming

Lecture 4: OpenMP

Open Multi-Processing

January 23, 2017
Outline

• OpenMP – another approach for thread parallel programming
• Fork-Join execution model
• OpenMP constructs – syntax and semantics
  – Work sharing
  – Thread scheduling
  – Data sharing
  – Reduction
  – Synchronization
• ‘count_primes’ hands-on!
OpenMP: Common Thread-Level Programming Approach in HPC

• Portable across shared-memory architectures
• Incremental parallelization
  – Parallelize individual computations in a program while leaving the rest of the program sequential
• Compiler based
  – Compiler generates thread program and synchronization
• Extensions to existing programming languages (Fortran, C and C++)
  – mainly by directives
  – a few library routines

See http://www.openmp.org
Fork-Join Model
OpenMP HelloWorld

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

int main (int argc, char *argv[]) {
    #pragma omp parallel {
        printf("Hello World from Thread %d!\n", omp_get_thread_num());
    }
    return 0;
}

Compiling for OpenMP
    gcc: -fopenmp, icc: -openmp, pgcc: -mp, ...
Number of threads

- if clause
- NUM_THREADS clause
- omp_set_num_threads()
- OMP_NUM_THREADS
- Default
OpenMP constructs

- Compiler directives (44)
  
  ```
  #pragma omp parallel [clause]
  ```

- Runtime library routines (35)
  
  ```
  #include <omp.h>
  int omp_get_num_threads(void)
  int omp_get_thread_num(void)
  ```

- Environment variable (13)
  
  ```
  export OMP_NUM_THREADS=x
  ```
Work sharing

• divides the execution of the enclosed code region among multiple threads
  – for shares iterations of a loop across the team of threads

#pragma omp parallel for [clause]

– Also sections and single (see [1])
Work sharing - for

#include <omp.h>

int main (int argc, char *argv[]) {
    int i, n=10;

    #pragma omp parallel for {
        for(i=0;i<n;i++)
            printf("Hello World!\n");
    }
    return 0;
}
Thread scheduling

• Static: Loop iterations are divided into pieces of size \textit{chunk} and then statically assigned to threads.
  – \texttt{schedule(static [,chunk])}

• Dynamic: Loop iterations are divided into pieces of size \textit{chunk}, and dynamically scheduled among the threads
  – \texttt{schedule(dynamic [,chunk])}

• More options,
  – guided, runtime, auto
Data sharing/ Data scope

- shared variables are shared among threads
- private variables are private to a thread
- Default is shared

- Loop index is private, nested loops

`#pragma omp parallel for private(list) shared(list)`
- Can be used with any work sharing clause
- Also firstprivate, lastprivate, default, copyin, ... (see [1])
Reduction

• 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.

reduction (operator: list)
Reduction

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

int main (int argc, char *argv[])
{
    int i,n=1000;
    float a[1000], b[1000], sum;

    for (i=0; i<n; i++) a[i] = b[i] = i * 1.0;
    sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (i=0; i<n; i++) sum = sum + (a[i] * b[i]);
    printf("Sum = %f\n",sum);
}
```

Source: [http://computing.llnl.gov/tutorials/openMP/samples/C/omp_reduction.c](http://computing.llnl.gov/tutorials/openMP/samples/C/omp_reduction.c)
OpenMP Synchronization

• Recall ‘barrier’ from pthreads
  – int pthread_barrier_wait(pthread_barrier_t *barrier);

• Implicit barrier
  – At the end of parallel regions
  – Barrier can be removed with nowait clause

#pragma omp parallel for nowait

• Explicit synchronization
  – single, critical, atomic, ordered, flush
Exercise

• See `prime_sequential.c`
• How to improve? Write a thread parallel version using what we discussed
• Observe scalability with the #of threads

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

• What’s good?
  – Small changes are required to produce a parallel program from sequential (parallel formulation)
  – Avoid having to express low-level mapping details
  – Portable and scalable, correct on 1 processor

• What is missing?
  – Not completely natural if want to write a parallel code from scratch
  – Not always possible to express certain common parallel constructs
  – Locality management
  – Control of performance
References

[1] Blaise Barney, Lawrence Livermore National Laboratory
https://computing.llnl.gov/tutorials/openMP

[2] XSEDE HPC Workshop: OpenMP