PROGRAMMING WITH SHARED MEMORY

Tulin Kaman

Department of Applied Mathematics and Statistics

Stony Brook/BNL New York Center for Computational Science

tkaman@ams.sunysb.edu

Aug 23, 2012
How to program machines?

- Distributed - Memory Machines
  - Each node in the computer has a locally addressable memory space
  - Parallel programs consists of cooperating processes, each with its own memory
  - Processes send data to one another as messages. Message Passing Interface (MPI) is a widely used standard for writing message-passing programs

- Shared - Memory Machines
  - Each core can access the entire data space
  - In shared memory multi-core architectures, OpenMP, Pthreads can be used to implement parallelism
Shared Memory Model

- All threads have access to the same, **global shared**, memory.
- Threads also have their own private memory.
- Shared data is accessible by all threads.
- Private data can be only accessed by the thread that owns it.
- Programmers are responsible for synchronizing access (protecting) globally shared data.
Labeling the data

As programmers we need to think about where in the memory we can put the data

- **Shared**
  - All threads can read and write the data simultaneously.
  - The changes are visible to all threads.

- **Private**
  - Thread has a copy of the data.
  - Other thread can not access to this data.
  - The changes are visible to only the thread that owns the data.
Common model for threaded program

Fork-Join Model

- Master threads runs from start to end.
- When the parallelism is specified, the main thread gets help from the threads that are called worker threads.
- At the end of the parallel portion of the work, the threads synchronize and terminate.
- Only the main thread leaves
Start with auto parallelization

• IBM XL thread safe compilers will automatically parallelize your code if
  • There is no branching into or out of the loop.
  • The increment expression is not within a critical section.
  • The order in which loop iterations start and end doesn’t affect the result.
  • The loop doesn’t contain I/O operations
  • The program is compiled with a thread-safe version of compiler. (_r suffix : mpixlc_r, mpixlcxx_r, mpixlf77_r,...)
Shared Memory Programming

Pthreads

POSIX threads

(Portable Operating System Interface) threads
Shared Memory Programming: Pthreads

- Hardware vendors have their own versions of threads. Difficult for programmers to develop portable threaded applications.

- For UNIX systems, a standardized programming interface specified by the IEEE POSIX 1003.1c standard. This standard are referred to as POSIX threads.
POSIX Threads - Pthreads

- Pthreads API contains around 100 subroutine
- `pthread.h` header file should be included
- POSIX standard for the C language. Fortran programmers can use wrappers around C function calls. *IBM XL provides a Fortran pthreads API.*

<table>
<thead>
<tr>
<th>Routine Prefix</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_</td>
<td>Threads themselves and miscellaneous subroutines</td>
</tr>
<tr>
<td>pthread_attr_</td>
<td>Thread attributes objects</td>
</tr>
<tr>
<td>pthread_mutex_</td>
<td>Mutexes</td>
</tr>
<tr>
<td>pthread_mutexattr_</td>
<td>Mutex attributes objects</td>
</tr>
<tr>
<td>pthread_cond_</td>
<td>Condition variables</td>
</tr>
<tr>
<td>pthread_condattr_</td>
<td>Condition attributes objects</td>
</tr>
<tr>
<td>pthread_key_</td>
<td>Thread-specific data keys</td>
</tr>
<tr>
<td>pthread_rwlock_</td>
<td>Read/write locks</td>
</tr>
<tr>
<td>pthread_barrier_</td>
<td>Synchronization barriers</td>
</tr>
</tbody>
</table>
## Compiling Threaded Programs

<table>
<thead>
<tr>
<th>Compiler / Platform</th>
<th>Compiler Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEL Linux</td>
<td>icc -pthread</td>
</tr>
<tr>
<td></td>
<td>icpc -pthread</td>
</tr>
<tr>
<td>PGI Linux</td>
<td>pgcc -lpthread</td>
</tr>
<tr>
<td></td>
<td>pgCC -lpthread</td>
</tr>
<tr>
<td>GNU Linux, Blue Gene</td>
<td>gcc -pthread</td>
</tr>
<tr>
<td></td>
<td>g++ -pthread</td>
</tr>
<tr>
<td>IBM Blue Gene</td>
<td>bgxlc_r / bgcc_r</td>
</tr>
<tr>
<td></td>
<td>bgxlC_r, bgxlc++_r</td>
</tr>
</tbody>
</table>

IBM Blue Gene P: If you want to use fewer than the number of available you need to set in your LoadLeveler batch job file

# @ arguments = -env XLSMPOPTS=PARTHDS=2 -exe …
Example: PI
3.14159265358979323846264338327950288419716

\[ \int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi \]

**NUMERICAL INTEGRATION**

\[ \sum_{i=0}^{N} F(x_i) \Delta x \approx \pi \]
Serial Code in C

```c
#include <stdio.h>
#define NUMSTEPS 1000000

int main ()
{
    int i;
    double x, Pi, step, sum = 0.0;

    step = 1.0/NUMSTEPS;

    for(i = 0; i <= NUMSTEPS; ++i)
    {
        x = (i+0.5)*step;
        sum += 4.0 / (1.0+x*x);
    }
    Pi = step * sum;

    return 0;
}
```
Creating and Terminating Threads

• Creating Threads:
  • A single, default thread in your main() program
  • Other threads must be explicitly created by the programmer.

```c
pthread_create(thread, attr, start_routine, arg)
```

  - identifier for the new thread returned by the subroutine
  - attribute object used to set thread attributes
  - C routine that the thread execute once it is created
  - argument that is passed to start_routine

• Terminating Threads: `pthread_exit()`
```c
#include <stdio.h>
#include <pthread.h>
#define NUMSTEPS 1000000
#define NUMTHRDS 4

double step = 0.0, Pi = 0.0;
pthread_mutex_t mutexsum;

void *Pi_Func(void *pArg)
{
    int i, myrank = *((int *)pArg);
    double x, mysum = 0.0;

    for (i = myrank; i < NUMSTEPS; i += NUMTHRDS)
    {
        x = (i + 0.5) * step;
        mysum += 4.0 / (1.0 + x*x);
    }

    pthread_mutex_lock(&mutexsum);
    printf("Thread %ld adding partial sum %f.\n", pArg, mysum);
    Pi += mysum * step;
    pthread_mutex_unlock(&mutexsum);

    return 0;
}
```
int main ()
{
    pthread_t   callThd[NUMTHRDS];
    int         i;
    step = 1.0/ NUMSTEPS;

    pthread_mutex_init(&mutexsum, NULL);

    for(i=0; i<NUMTHRDS; i++) {
        pthread_create( &callThd[i], NULL, Pi_Func, (void *)i);
    }

    for(i=0; i<NUMTHRDS; i++) {
        pthread_join( callThd[i], NULL);
    }

    printf ("Threaded version: Pi = %f \n", Pi);

    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
Shared Memory Programming
OpenMP

OpenMP
Open Multi-Processing
Shared Memory Programming: OpenMP

OpenMP = Open Multi-Processing

- The OpenMP Application Program Interface (API) for writing shared memory parallel programs.
- Supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures.
- Consists of
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- OpenMP program is portable. Compilers have OpenMP support.
- Requires little programming effort.
## Compiling Threaded Programs

<table>
<thead>
<tr>
<th>Compiler / Platform</th>
<th>Compiler</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Linux Opteron/Xeon</td>
<td>icc icpc ifort</td>
<td>-openmp</td>
</tr>
<tr>
<td>PGI Linux Opteron/Xeon</td>
<td>pgcc pgCC pgf77 pgf90</td>
<td>-mp</td>
</tr>
<tr>
<td>GNU Linux Opteron/Xeon</td>
<td>gcc g++ g77 gfortran</td>
<td>-fopenmp</td>
</tr>
<tr>
<td>IBM Blue Gene</td>
<td>bgxlc_r, bgcc_r bgxLC_r, bgxlc++_r bgxlc89_r bgxlc99_r bgxlf_r bgxlf90_r bgxlf95_r bgxlf2003_r</td>
<td>-qsmp=omp</td>
</tr>
</tbody>
</table>

IBM Blue Gene P: If you want to use fewer than the number of available you need to set in your LoadLeveler batch job file

```bash
# @ arguments = -env OMP_NUM_THREADS=2 -exe ...
```
OpenMP uses the fork-join model

**FORK:** the master thread creates a team of parallel threads
Labeling the data

- Most variables are shared by default

- Global variables: File scope variables, static

- Private variables:
  - Do/For Loop index variables
  - Stack variables in functions called from parallel regions
The OpenMP API
three distinct components

As of version 3.1

- Compiler Directives (20)
- Runtime Library Routines (32)
- Environment Variables (9)
Compiler Directives

- In source code **compiler directives** are used for
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads

<table>
<thead>
<tr>
<th>Fortran</th>
<th>!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(…)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>#pragma omp parallel default(shared) private(…)</td>
</tr>
</tbody>
</table>
Runtime Library Routines

• Include the <omp.h> header file
• These routines are used for:
  • Setting and querying
    • the number of threads
    • the dynamic threads feature
    • nested parallelism
    • thread's unique identifier (thread ID), the thread team size
    • wall clock time and resolution
  • initializing and terminating locks and nested locks
• Fortran routines are not case sensitive, C/C++ routines are

<table>
<thead>
<tr>
<th>Fortran</th>
<th>INTEGER FUNCTION OMP_GET_NUM_THREADS()</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>int omp_get_num_threads(void)</td>
</tr>
</tbody>
</table>
# Environment Variables

- control the execution of parallel code at run-time
- Set the same way set any other environment variables

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_SCHEDULE</td>
<td>how iterations of the loop are scheduled on processors</td>
</tr>
<tr>
<td>OMP_NUM_THREADS</td>
<td>Sets the maximum number of threads</td>
</tr>
<tr>
<td>OMP_DYNAMIC</td>
<td>Enables or disables dynamic adjustment of the threads</td>
</tr>
<tr>
<td>OMP_PROC_BIND</td>
<td>Enables or disables threads binding to processors</td>
</tr>
<tr>
<td>OMP_NESTED</td>
<td>Enables or disables nested parallelism</td>
</tr>
<tr>
<td>OMP_STACKSIZE</td>
<td>Controls the size of the stack for non-Master threads</td>
</tr>
<tr>
<td>OMP_WAIT_POLICY</td>
<td>ACTIVE / PASSIVE</td>
</tr>
<tr>
<td>OMP_MAX_ACTIVE_LEVELS</td>
<td>Controls the maximum number of nested active parallel regions</td>
</tr>
<tr>
<td>OMP_THREAD_LIMIT</td>
<td>Sets the number of threads to use for the whole program</td>
</tr>
</tbody>
</table>
Parallel Regions: Thread creation

```c
omp_set_num_threads(4);
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    ...
}

#pragma omp parallel num_threads(4)
{
    int tid = omp_get_thread_num();
    ...
}
```

- Sets the number of threads that will be used in the next parallel region.
- Returns the thread number of the thread, within the team.
- Clause to request a number of threads.

Each thread executes the section of the code in parallel region.
False sharing occurs when processors in a shared-memory parallel system make references to different data objects within the same coherence block (cache line or page), thereby inducing "unnecessary" coherence operations.

CRITICAL directive executed by only one thread at a time
Work-Sharing Constructs
DO / for Directive

**Fortran**

```
!$OMP DO [clause ...]
do_loop

!$OMP END DO [ NOWAIT ]
```

**C/C++**

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

```
#include <stdio.h>
#include <omp.h>
define NUMSTEPS 1000000
#define NUMTHRDS 4

int main ()
{
    int i, nthreads;
    double x, step, Pi=0.0, sum=0.0;
    step = 1.0/NUMSTEPS;

    omp_set_num_threads(NUMTHRDS);

    #pragma omp parallel for reduction (operator: list)
    default(shared) private(i,x)
    reduction(+:sum)
    for(i = 0; i <= NUMSTEPS; ++i)
    {
        x = (i+0.5)*step;
        sum += 4.0 / (1.0+x*x);
    }

    Pi += step * sum;
    return 0;
}
```
Work-Sharing Constructs
SECTIONS Directive

- Barrier at the end of a SECTIONS directive
- Use NOWAIT clause to turn off the barrier
- Each SECTION within a SECTIONS directive is executed once by a thread in the team.

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$OMP SECTIONS [clause ...]</td>
<td>#pragma omp sections [clause ...]</td>
</tr>
<tr>
<td>!$OMP SECTION block</td>
<td></td>
</tr>
<tr>
<td>!$OMP SECTION block</td>
<td></td>
</tr>
<tr>
<td>!$OMP END SECTIONS [NOWAIT ]</td>
<td></td>
</tr>
</tbody>
</table>
Work-Sharing Constructs
SINGLE Directive

- the code is to be executed by only one thread in the team.
- useful when the part of the code is not thread safe (I/O)

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$OMP SINGLE [clause …] block</td>
<td>#pragma omp single [clause …] structured_block</td>
</tr>
<tr>
<td>!$OMP END SINGLE [ NOWAIT]</td>
<td></td>
</tr>
</tbody>
</table>
#include <stdio.h>
#include <omp.h>
#define NUMSTEPS 1000000
#define NUMTHRDS 4

int main ()
{
    int i, nthreads;
    double x, step, Pi=0.0, sum=0.0;
    step = 1.0/NUMSTEPS;

   omp_set_num_threads(NUMTHRDS);

    #pragma omp parallel for   
    default(shared)   
    private(i,x,sum)   
    for(i = 0; i <= NUMSTEPS; ++i)
    {
        x = (i+0.5)*step;
        sum += 4.0 / (1.0+x*x);
    }
    printf("sum = %f\n",sum);
    return 0;
}
Firstprivate: variables are automatically initialized from the shared variables
Comparison

**PTHREADS**
- To make use of Pthreads, developers must write their code specifically for this API.

```c
void start_routine (void *pArg)
{
    Routines ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
    pthread_create(&tid[i],NULL,start_routine, (void *)i);

start_routine();
for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

**OpenMP**
- Easy to implement

```c
omp_set_num_threads(4);
#pragma omp parallel
{
    Routines();
}
```
References

http://www.openmp.org

An Overview of OpenMP Video, Ruud van der Pas

Parallel Programming in OpenMP by Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon

POSIX Threads Programming
   Blaise Barney, Lawrence Livermore National Laboratory
   https://computing.llnl.gov/tutorials/pthreads/

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