Concurrent Programming

Dr A Sahu
Dept of Computer Science & Engineering
IIT Guwahati

Outline

• Synchronization primitive: TAS, TTAS, BTAS
• Example of Parallel Programming
  – Shared memory: C/C++ Pthread, C++11 thread, OpenMP, Cilk
  – Distributed Memory: MPI
• Concurrent Objects
  – Concurrent Queue, List, stack, Tree, Priority Queue, Hash, SkipList
• Use of Concurrent objects

Review: Test-and-Set

import java.util.concurrent.atomic
public class AtomicBoolean {
  boolean value;
  public synchronized boolean getAndSet(
    boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
  }
}

Test-and-Set Locks

• Locking
  – Lock is free: value is false
  – Lock is taken: value is true
• Acquire lock by calling TAS
  – If result is false, you win
  – If result is true, you lose
• Release lock by writing false

Test-and-set Lock

class TASlock {
  AtomicBoolean state =
  new AtomicBoolean(false);
  void lock() {
    while (state.getAndSet(true)) {} // Keep trying until lock acquired
  }
  void unlock() {
    state.set(false);
  }
}

class TTASlock {
  AtomicBoolean state =
  new AtomicBoolean(false);
  void lock() {
    while (true) {
      while (state.get()) {} // Then try to acquire it
      if (!state.getAndSet(true)) {
        state.set(false);
        return;
      }
    }
  }
}

Test-and-test-and-set Lock

• Acquire lock by calling TTAS
  – If result is false, you win
  – If result is true, you lose
• Release lock by writing false
Parallel Programming

- Shared memory
  - Pthread, C++11 thread
  - Java
  - OpenMP
  - Cilk
- Distributed Memory
  - MPI

In C/C++/Java Thread

- We need to identify parallelism
  - How to do extract parallelism manually
  - Parallel Decomposition
- Code in threaded model
  - OS is responsible for running it efficiently
    - Less control over runtime

Parallel Decomposition

- Data Parallelism
- Function Parallelism
- Pipeline Parallelism
- Mixed Parallelism (D+F+P)

Parallel Decomposition

- Data Parallelism

Parallel Decomposition

- Function Parallelism

Parallel Decomposition

- Pipeline Parallelism
**OpenMP**

- Compiler directive: Automatic parallelization
- Auto generate thread and get synchronized

```c
#include <openmp.h>
main(){
    #pragma omp parallel
    #pragma omp for schedule(static)
    for (int i=0; i<N; i++) {
        a[i]=b[i]+c[i];
    }
}
```

**OpenMP: Parallelism**

<table>
<thead>
<tr>
<th>Sequential code</th>
<th>(Semi) manual parallel</th>
<th>Auto parallel of the for loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int i=0; i&lt;N; i++) { a[i]=b[i]+c[i]; }</td>
<td>#pragma omp parallel</td>
<td>#pragma omp parallel for schedule(static)</td>
</tr>
<tr>
<td>#pragma omp parallel</td>
<td>for (int i=0; i&lt;N; i++) { a[i]=b[i]+c[i]; }</td>
<td>( for (int i=0; i&lt;N; i++) { a[i]=b[i]+c[i]; } )</td>
</tr>
</tbody>
</table>

**Work-sharing: the for loop**

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

```c
#pragma omp parallel
#pragma omp for
for(i =1;i<13; i++)
c[i]=a[i]+b[i];
```

**OpenMP Fork-and-Join model**

- printf("program begin\n");
- N = 1000;
- #pragma omp parallel for
- for (i=0; i<N; i++)
- A[i] = B[i] + C[i];
- M = 500;
- #pragma omp parallel for
- for (j=0; j<M; j++)
- p[j] = q[j] – r[j];
- printf("program done\n");

**Critical Construct**

```c
sum = 0;
#pragma omp parallel private (lsum)
{
    lsum = 0;
    #pragma omp for
    for (i=0; i<N; i++)
    {
        lsum = lsum + A[i];
    }
    #pragma omp critical
    { sum += lsum; }
}
```

**Reduction Clause**

```c
sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++)
{
    sum = sum + A[i];
}
```
OpenMP Schedule

- Can help OpenMP decide how to handle parallelism
  
  schedule(type [,chunk])

- Schedule Types
  
  - Static – Iterations divided into size chunk, if specified, and statically assigned to threads
  - Dynamic – Iterations divided into size chunk, if specified, and dynamically scheduled among threads

Static Schedule with chunk

- A loop with 1000 iterations and 4 omp threads. Static Schedule with Chunk 10

```c
#pragma omp parallel for schedule (static, 10)
{  
  for (i=0; i<1000; i++)
    A[i] = B[i] + C[i];
}
```

Dynamic Schedule

- With a dynamic schedule new chunks are assigned to threads when they come available.
  
  SCHEDULE(DYNAMIC,n)
  
  -- Loop iterations are divided into pieces of size chunk. When a thread finishes one chunk, it is dynamically assigned another.

  SCHEDULE(GUIDED,n)
  
  -- Similar to DYNAMIC but chunk size is relative to number of iterations left.

  Although Dynamic scheduling might be the preferred choice to prevent load imbalance
  
  -- In some situations, there is a significant overhead involved compared to static scheduling.

Issues with Static schedule

- With static scheduling the number of iterations is evenly distributed among all openmp threads (i.e. Every thread will be assigned similar number of iterations).
  
  - This is not always the best way to partition. Why is This?

  Example: Sqrt timing is data dependent...

  This is called load imbalance. In this case threads 2,3, and 4 will be waiting very long for thread 1 to finish

More Examples on OpenMP

Cilk

- Developed by Leiserson at CSAIL, MIT
- Addition of 6 keyword to standard C
  - Easy to install in linux system: with gcc and pthread
- Biggest principle
  - Programmer should be responsible for exposing the parallelism, identifying elements that can safely be executed in parallel;
  - Work of run-time environment (scheduler) to decide during execution how to actually divide the work between processors
- Work Stealing Scheduler
  - Proved to be good scheduler
  - Now also in GCC, Intel CC, Intel acquire Cilk++

Fibonacci

int fib (int n) {
  if (n<2) return (n);
  else {
    int x, y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}

Cilk code

cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}

C elision

Cilk is a faithful extension of C. A Cilk program’s serial elision is always a legal implementation of Cilk semantics. Cilk provides no new data types.

Parallelizing Vector Addition

C

void vadd (real *A, real *B, int n) {
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}

Cilk

cilk void vadd (real *A, real *B, int n) {
  int i; for (i=0; i<n; i++) A[i]+=B[i];
  else {
    spawn vadd (A, B, n/2);
    spawn vadd (A+n/2, B+n/2, n-n/2);
    sync;
  }
}

Parallelization strategy:
1. Convert loops to recursion.
2. Insert Cilk keywords.

Parallelization strategy:
1. Convert loops to recursion.
2. Insert Cilk keywords.

Side benefit: D&C is generally good for caches!
**Compiling Cilk Program**

- Cilk source
- `cilk2c` translator
- C source
- `gcc` compiler
- Object code
- Linking loader
- Binary

`cilk2c` translates straight C code into identical C postsource.

```
cilk fib.cilk –o fib
./fib -proc 4
```

• Run using 4 threads.

**MPI**

- Message Passing Interface
- Distributed memory multiprocessor: cluster programming
- Scalable to Large Number of Processors
- Send(), Recv() construct
- It uses processes not thread
- Not part of this course

**MPI Example**

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(int argc, char *argv[]) {
    int myid, numprocs, tag, source, destination, count, buffer;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;
    source=0;
    destination=1;
    count=1;

    if(myid == source){
        buffer=5678;
        MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
        printf("processor %d sent %d\n",myid,buffer);
    }

    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
        printf("processor %d got %d\n",myid,buffer);
    }
    MPI_Finalize();
}
```

**MPI Examples**

```c
if(myid == source){
    buffer=5678;
    MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
    printf("processor %d sent %d\n",myid,buffer);
}
if(myid == destination){
    MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    printf("processor %d got %d\n",myid,buffer);
}
MPI_Finalize();
```

**Parallel Programming**

- Exploits section that can be parallelized easily
  - Cuda GPU (Very simplistic, completely SPMD)
  - OpenMP (A bit more complex SPMD with control of reduction, critical and scheduling)
- How to handle critical path efficiently
- Testing for Good Driver
  - Driving 120kmph in express high way or drive for 2 hour in Guwahati city in evening 4PM to 6PM near fancy bazar.
- How to overall speed up the application by trying to parallelize the not the easiest part
  - The data storing, adding, removing and retrieving, organizing.
  - This leads to design Concurrent Data Structure

**Concurrent Data Structure**

- Classical Data structure (simply DS)
  - Complexity of Add, Del, Search, Modify, Rearrange
- Concurrent Data Structure (CDS)
  - Many thread access to CDS
  - Consistency and Serialization
  - Performance of locking
  - Property should hold
    - No: live lock, deadlock, starvation
  - Wait free and lock free CDS
Available CDS library
• Java, C++11 and C# (Microsoft VC++)
  – Java thread safe/concurrent collections
    • /usr/share/javadoc/java-1.6.0-
      openjdk/api/java/util/concurrent/
  – C++11 : through Boost library or Libcds
  – C# The Parallel Patterns Library (PPL)
    • concurrent_vector_class, Concurrent_queue_class
• All have competitive memory model
• Use Java
  – Still I think Java is better in CDS...
  – Shavit: author of AMP Book used Java

Guideline for designing CDS
• Ensure no thread can see a state where invariant of the DS have been broken by the action of other thread.
• Take care to avoid race condition inherent in the interface to the DS by proving function for complete functions for the operation rather than for operation steps. Race condition (unexpected output)
• Pay attention to how the DS behaves in the presence of exceptions to ensure that the invariants are not broken.
• Minimized the opportunities for deadlock when using the DS by restricting the scope of locked and avoid nested locks where possible.

CDS
• List, Queue, Stack, Hash, Priority queue, skiplist
• Discussion on internal implementation of some of these : list, q, stack, pq, hash..
• Use this CDS to make parallel application faster

Lock free CDS
• More than one thread must be able to access the DS concurrently
  – Example: A Queue might allow one thread to push and others to pop but may not same push/pop.
• If one thread accessing DS and get suspended by OS in midway , other thread should be able to access without waiting for suspended thread
• Lock free may not be wait free: it may have starvation
  – 1TS 2T 1TS 2T, .....sequence lead to starvation by a RR scheduler

Enabling genuine concurrent access to CDS
• Can the scope of locks be restricted to allow some parts of an operation to be performed outside the lock?
• Can different part of the DS be protected by different locks?
• Do all operations require the same level of protection?
• Can a simple change to the DS improve the improve the opportunities for concurrency without affecting the operational semantics ?

Wait free data structure
• Lock free DS with property: every thread accessing the DS can complete its operation within bounded number of steps, regardless of behavior of other threads.
• Coding for wait free DS correctly is extremely hard
Coarse-Grained Synchronization

• Each method locks the object
  – Avoid contention using queue locks
  – Easy to reason about
    • In simple cases
• So, are we done?

• NOPE

Thanks