Shared Memory Programming: pThreads and OpenMP

IBM
July, 2006
Agenda

• Shared Memory
  • Computer Architecture
  • Programming Paradigm
• pThreads
• OpenMP Compilers
  • Implementation
  • Compiler Internals
• Automatic Parallelization
• Work Distribution
• Performance
• Problems and Concerns
Compiling and Running an OpenMP Program

```bash
xlf90_r -O3 -qsmp -c openmp_prog.c
xlf90_r -qsmp -o a.out openmp_prog.o
export OMP_NUM_THREADS=4
./a.out &
ps -m -o THREAD
```

_r = reentrant code
Use –qsmp both for compiling and for linking
Shared Memory Programming

**pThreads**
- POSIX standard
- Library functions
- Explicit fork and join
- Explicit synchronization
- Explicit locks
- Often used in “operational” environment
- Often used for M:N applications = many short tasks for a few processors

**OpenMP**
- Industry Standard
- Compiler assist:
  - Directives (Fortran)
  - Pragmas (C, C++)
- Explicit “fork” and “join”
- Implicit synchronization
- Implicit locks
- Often used in data analysis
- Often used for 1:1 applications = one task per processor
Shared Memory Architecture

- One address space
  - Multiple processor access
- Software synchronization
- Hardware (and software) cache coherence
Shared Memory Architecture Variation: NUMA

- Single address space
- Non-uniform memory latency
- Non-uniform memory bandwidth

Shared Memory Definition: Single address space
pThreads

• Standard for shared memory programming
• POSIX standard
• ALL modern computers
• Reentrant code:
  • Does not hold static data over successive call.
  • Important for f77 (static storage) programs
pThreads Thread Scope

- **AIXTHREAD_SCOPE={S|P}**
  - **S**: Thread is bound to a kernel thread
    - Scheduled by the kernel.
  - **P**: Thread is subject to the user scheduler
    - Does not have a dedicated kernel thread
    - Sleeps in user mode
    - Placed on the user run queue when waiting for a processor
    - Subjected to time slicing by the user scheduler
    - Priority of thread is controlled by user
pThreads Thread Scope

• Typically, user is concerned with process threads
• Typically, operating system uses system threads
  • Process to system thread mapping (pThreads default):
    • 8 to 1
      – 8 process threads map onto 1 system thread
• User can choose system or process threads
  • pthread_attr_setscope(&attr,PTHREAD_SCOPE_SYSTEM);
  • export AIXTHREAD_SCOPE=S
  • export AIXTHREAD_MNRATIO=m:n
Void main()
{
... 
for (i=0;i<nthreads;i++)
  pthread_create(....)
....
}
OpenMP Thread Scope

- 1:1 model
- Each user (process) thread is mapped to one system thread

```c
Void main()
{
    ...
    #pragma OMP parallel
    for (i=0;i<nthreads;i++)
        work();
    ....
}
```

System Threads

-qsmpe
OpenMP and pThreads

- OpenMP uses a shared memory model similar to pThreads
  - Fork
  - Join
  - Barriers (mutexes)
- NOT strictly built on top of pThreads
//"Master" thread forks slave threads

main()
{
...
for (i = 0; i < num_threads; i++)
    pthread_create ( &tid[i], ...
for (i = 0; i < num_threads; i++)
    pthread_join( &tid[i], ... );
}

void do_work(void *it_num)
{
for (i=start;i<ending;i++)
    A[i]=func(B);
    ...
}

Void sub(n,A,B)
{
    #pragma omp parallel
    {
        #pragma omp for
        for (i=0;i<n;i++)
            {
                A[i] = func(B);
                ...  
            
    }
          ....
}
**OpenMP Example**

```
Void user_sub()
{
  ...
  #pragma omp parallel
  {
    ...
    #pragma omp for
    for (i=0;i<;i++)
      {A[i] = ....
       B[i] = ..
      }
  }
  ...
}

Return (0)
```

**Fork**

**Locks**

**Locks**

**Join**
Parallel Regions and Work Sharing

Void user_sub()
{
    ...
    #pragma omp parallel
    {
        ...
        #pragma omp for
        for (i=0;i<;i++){
            A[i] = ....
            B[i] = ....
        }
    }
    ...
}
Return (0)

• Parallel Region
  • Master thread forks slave threads
  • Slave threads enter “re-entrant” code

• Work Sharing
  • Slave threads collective divide work
  • Various scheduling schemes
    • User choice
Parallelization Example

```c
Void sub(n,A,B)
{
    #pragma omp parallel
    {
        #pragma omp for
        for (i=0;i<n;i++)
            A[i] = func(B);
        ....
    }
}
```
void sub(n,A,B)  
{  
    ...  
    #pragma omp parallel  
    sub1(A,B);  
    ...  
}  

void sub1(A,B)  
{  
    ...  
    #pragma omp for  
    sub2(ns,ne,A,B)  
    ....  
}  

void sub2(ns,ne,A,B)  
{  
    for (i=ns;i<ne;i++)  
        A[i] = func(B[i])  
}
OpenMP Work Distribution

• Scheduling determined by the user
  • Directives
  • Environment variables
• STATIC: small overhead
  • Distribution done at compile time
• DYNAMIC: better for load balancing
  • Distribution done during execution
• GUIDED
Scheduling

- Environment variable:
  - OMP_SCHEDULE={static, dynamic, guided}

-Pragma:
  - #pragma omp for schedule({static, dynamic, guided})

-Full syntax:
  - OMP_SCHEDULE={s...,d...,g...}[(chunk_size)]
  - #pragma ... schedule({s...,d...,g...},[chunk_size])
Scheduling: Static

- Default (xlf and xlc)
- Number of iterations per thread determined before loop execution

Compile

```c
for (i=0; i<n; i++)
{
    ...
}
```

Execute

```c
for (i=0; i<n/p; i++)
{
    ...
}
```

for (i=n/p+1; i<2*n/p; i++)
{
    ...
}

for (i=n/p+1; i<2*n/p; i++)
{
    ...
}

for (i=n/p+1; i<2*n/p; i++)
{
    ...
}
Static Scheduling Example

n=100 iterations

4 threads

Static Schedule

0 1 2 3

25 25 25 25
Scheduling: Dynamic

• Next thread takes next slab

Compile

for (i=0; i<n; i++)
{
    ...
}

Execute

for (i=0; i<n/p; i++)
{
    ...
}
Dynamic Scheduling Example

n=100 iterations
4 threads

Dynamic Schedule
Chunk_size=5
Scheduling: Guided

- Next thread takes reduced size slab

```c
for (i=0; i<n; i++)
{
    ...
}
```

```c
for (i=0; i<n/p; i++)
{
    ...
}
```
Guided Scheduling Example

n=100 iterations

4 threads
Performance Concerns

• False Sharing
  • Cache coherency thrashing
• Load balance
  • Uneven number distribution
  • Uneven work units
  • Triangles
• Barriers
  • Synchronization is expensive
False Sharing

- Multiple processors (threads) write to same cache line
  - Valid shared memory operation but causes severe performance penalty
  - Common in older Cray parallel/vector codes
- Dangerous programming practice
  - Difficult to detect

```c
float sum[8];
#pragma omp parallel
p = ...my thread number...
#pragma omp for
for (i=1; i<n; i++)
  sum[p] = sum[p] + func(i,n);
```
Corrected False Sharing

• Each processor (thread) writes to own cache line
  • Wastes a tiny bit of memory
  • Cache line is 128 bytes = thirty-two 4-byte words

```c
float sum[8*32];
#pragma omp parallel
p = ...my thread number...
#pragma omp for
for (i=1;i<n;i++)
  sum[p*32] = sum[p*32] + func(i,n);
```
False Sharing

- Multiple threads accessing SAME line cause contention

```c
#pragma omp for
for (i=1;i<n;i++) {
    k = index[i];
    A[i] = B[k] + i;
}
```
Effect of False Sharing

![Graph showing the effect of false sharing on overhead with varying thread counts. The x-axis represents the length, and the y-axis represents the percentage overhead. Different thread counts (2, 4, 8, 16, 24, 32) are represented by different markers and line styles.](image-url)
Effect of False Sharing: Two threads on Same Chip

% Overhead

Length

1 Chip
2 Chips

0 128 256 384 512 640 768 896 1024

0 50 100 150 200 250
False Sharing Summary

• Effect is worse with smaller features
• Effect is worse with more threads
Summary

• Shared memory programming
  • Only works on single address space node
• Dynamic constructs
  • Good for load balancing
• Concerns:
  • False sharing
  • Critical regions