Multi-Threading Performance on Commodity Multi-Core Processors

Jie Chen and William Watson III
Scientific Computing Group
Jefferson Lab
12000 Jefferson Ave.
Newport News, VA 23606
Organization

- Introduction
  - Multi-Core Processors based SMP
- Hardware and Software Environment
  - QMT (QCD Multi-Threading)
- Memory System
  - Cache Coherence Protocol
- Barrier Algorithms
  - Centralized and Queue-based
- Analysis of the Barrier Algorithms
- Performance of the Barrier Algorithms
- OpenMP and QMT
Why Multi-core Processors

- Traditional Design
  - High clock frequency
    - Deep pipeline
    - Pipeline interruption is expensive
      - Cache misses, branch mis-predication
      - Larger gap between memory and CPU
  - ILP (Instruction Level Parallelism)
    - Challenge to find enough parallel instructions
      - Underutilized
    - Superscalar processors have multiple executing engines
      - Hardware look for parallelism in large instruction window (expensive and complex)
      - Compilers

- Most processors can’t find enough work
  - peak IPC is 6, average IPC is 1.5!
Multi-core Processors

- CMT (Chip Multi-Threading)
  - Exploit higher level of parallelism
  - Execute multiple threads on multi-cores within a single CPU
- Cache-coherence circuitry operates at higher clock speed
  - Signals stay inside the chip
- Each core is relatively simple
  - Smaller instruction window
    - Smaller die area
    - Consume less power
- Sequential programs may perform poorly
  - Cache contention/cache thrashing on shared cache systems
Motivation

- Linux clusters based on commodity multi-core SMP nodes
  - Multi-threading data parallel scientific applications
  - Fork-join style parallel programming paradigm
- Multi-threading Strategy
  - OpenMP
  - Hand written threading library (QMT)
    - Optimal barrier algorithm
- Barrier algorithm performance
OpenMP

- Portable, Shared Memory Multi-Processing API
  - Compiler Directives and Runtime Library
  - C/C++, Fortran 77/90
  - Unix/Linux, Windows
  - Intel c/c++, gcc-4.x
    - Implementation on top of native threads

- Fork-join Parallel Programming Model
OpenMP

- Compiler Directives (C/C++)
  
  ```c
  #pragma omp parallel
  {
    thread_exec (); /* all threads execute the code */
  } /* all threads join master thread */
  #pragma omp critical
  #pragma omp section
  #pragma omp barrier
  #pragma omp parallel reduction(+:result)
  ```

- Run time library
  - `omp_set_num_threads`, `omp_get_thread_num`
QMT (QCD Multi-Threading)

- QMT: Local developed multi-threading library
  - LQCD applications: data parallel
  - Fork-join programming model
  - Very light weight lock
  - Optimal barrier algorithm

```c
typedef void (*qmt_userfunc_t) (void *usrarg, int thid);
extern int qmt_init (void);
extern int qmt_finalize (void);
extern int qmt_pexec (qmt_userfunc_t func, void* arg);
extern int qmt_thread_id (void);
extern int qmt_num_threads(void);
```
Motivation

- Memory architecture of a SMP using multi-core processors
  - Shared BUS (Intel Xeons)
  - ccNUMA (AMD Opterons)
  - Cache Coherence Protocol

- Memory architecture impact on the performance of barrier algorithms
  - Memory Organization
  - Cache coherence protocol
Hardware and Software Environment

- **Dell Power Edge 1850**
  - Dual Intel Xeon 5150 2.66 GHz
  - Shared Bus Memory System

- **Dell Power Edge SC1435**
  - Dual AMD Opteron 2220SE 2.8 GHz
  - ccNUMA Memory System

- **Fedora Core 5 Linux x86_64**
  - Kernel 2.6.17 (with PAPI support)
  - gcc 4.1 and Intel icc 9.1
Multi-Core Architecture

L1 Cache
- 32 KB Data, 32 KB Instruction

L2 Cache
- 4 MB Shared among 2 cores
  - 256 bit width
  - 10.6 GB/s bandwidth to cores

FB-DDR2
- Increased Latency
- Memory disambiguation allows load ahead store instructions

Executions
- Pipeline length 14; 24 bytes
- Fetch width; 96 reorder buffers
- 3 128-bit SSE Units; One SSE instruction/cycle

Intel Woodcrest Xeon

L1 Cache
- 64 KB Data, 64 KB Instruction

L2 Cache
- 1 MB dedicated
  - 128 bit width
  - 6.4 GB/s bandwidth to cores

NUMA (DDR2)
- Increased latency to access the other memory
- Memory affinity is important

Executions
- Pipeline length 12; 16 bytes
- Fetch width; 72 reorder buffers
- 2 128-bit SSE Units; One SSE instruction = two 64-bit instructions.

AMD Opteron
## Configurations of Test Machines

<table>
<thead>
<tr>
<th></th>
<th>CPUs</th>
<th>L1</th>
<th>L2</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>Two 2.66 GHZ Dual-Core</td>
<td>32K Data 32K Instr</td>
<td>4MB Shared</td>
<td>4GB Shared Bus</td>
</tr>
<tr>
<td>AMD</td>
<td>Two 2.8 GHz Dual-Core</td>
<td>64K Data 64K Instr</td>
<td>1MB Private</td>
<td>4GB ccNUMA</td>
</tr>
</tbody>
</table>
Hardware and Software Environment

- **EPCC**
  - Micro-benchmark measuring thread synchronization overhead

- **PAPI (Performance Programming Interface API)**
  - Performance Event Counter Registers available on Intel Xeon and AMD Opteron Processors
  - Linux kernel modules (perfctr) can read performance event counts from the registers
  - Vast available performance metrics
    - PAPI_L2_TCM (L2 Cache Misses)
    - PAPI_FP_INS (Total Floating Point Ins)
  - Machine specific performance metrics
    - SI_PREFETCH_ATTEMPT (Opteron)
    - NB_MC_PAGE_MISS (Opteron)
    - SIMD_Int_128_Ret (Xeon)
Intel Xeon Memory Architecture

Processor 1
- Core 1 (CPU)
  - L1 Cache
- Core 2 (CPU)
  - L1 Cache
- L2 Cache

Processor 2
- Core 1 (CPU)
  - L1 Cache
- Core 2 (CPU)
  - L1 Cache
- L2 Cache

System Bus

System Memory
AMD Opteron Memory Architecture

- Core 1
  - L1 Cache
  - L2 Cache
  - SRQ

- Core 2
  - L1 Cache
  - L2 Cache
  - SRQ

- Cross Bar
- MCT
- Local Memory

- Core 1
  - L1 Cache
  - L2 Cache
  - SRQ

- Core 2
  - L1 Cache
  - L2 Cache
  - SRQ

- Cross Bar
- MCT
- Local Memory
Cache Coherence with Write Back
Caches

Core 1
- L1 Cache
- L2 Cache
  - \( a = 28 \)

Core 3
- L1 Cache
- L2 Cache
  - \( a = ? \)

Memory
- \( a = 7 \)
Intel utilizes MESI protocol:
- M: Modified
- E: Exclusive
- S: Shared
- I: Invalid

A write miss results in a read-exclusive bus transaction
- Write to a shared or an invalid block
- Invalidate other copies of the block
- Modified block has to be written back to memory when another cache reads the invalid block

AMD utilizes MOESI protocol:
- M: Modified
- E: Exclusive
- S: Shared
- I: Invalid
- O: Owner (Most Recent Copy of Data)

One cache can hold a block of data in the Owner state and the others are in shared state
- The owner of a cache block updates other caches reading the block
- The copy in the main memory can be stale
- Write modified cache back to memory can be avoided
MESI Cache Coherence Protocol

Accessing System Memory is Expensive
MOESI Cache Coherence Protocol

Cache to cache transfers are carried out by SRI or HPT
### Memory and Cache Access Latency

<table>
<thead>
<tr>
<th></th>
<th>L1 (ns)</th>
<th>L2 (ns)</th>
<th>Memory (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>1.13</td>
<td>5.29</td>
<td>150</td>
</tr>
<tr>
<td>AMD</td>
<td>1.07</td>
<td>4.3</td>
<td>173</td>
</tr>
</tbody>
</table>

#### Random Memory Access Latency

<table>
<thead>
<tr>
<th></th>
<th>Same CPU (ns)</th>
<th>Different CPU (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>55</td>
<td>187</td>
</tr>
<tr>
<td>AMD</td>
<td>83.5</td>
<td>119</td>
</tr>
</tbody>
</table>

#### Cache-to-cache Transfer Latency
Software Barrier

Threads

Barrier call

First comes

gather

release

Last comes

Last leaves

Barrier time
Centralized Barrier Algorithm

```c
int flag=atomic_get(&release);
int count=atomic_int_dec(&counter);
if (count == 0) {
    atomic_int_set (&counter,num_threads);
    atomic_int_inc (&release);
}
else spin_until (flag != release)
```

Memory contention
To the counter
typedef struct qmt_cflag {
    int volatile c_flag;
    int c_pad[CACHE_LINE_SIZE – 1];
} qmt_cflag_t;

typedef struct qmt_barrier {
    int volatile release;
    char br_pad[CACHE_LINE_SIZE – 1];
    qmt_flag_t flags[1];
} qmt_barrier_t;

/* Master Thread */
for (i = 1; i < num_threads; i++) {
    while (barrier->flags[i].cflag] == 0) ;
    barrier->flags[i].cflag=0;
}
atomic_int_inc(&barrier->release);

/* Thread i */
int rkey = barrier->release;
barrier->flags[i].cflag = -1;
While (rkey == barrier->release);

Eliminate some memory contention
Analysis of the Barrier Algorithms under Mesi Protocol

<table>
<thead>
<tr>
<th>Threads</th>
<th>Centralized</th>
<th>Queue-based</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>11</td>
<td>1.36</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>16</td>
<td>1.25</td>
</tr>
</tbody>
</table>
Analysis of the Barrier Algorithms under MOESI Protocol

<table>
<thead>
<tr>
<th>Threads</th>
<th>Centralized</th>
<th>Queue based</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>1.67</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>6</td>
<td>1.33</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>9</td>
<td>1.22</td>
</tr>
</tbody>
</table>
Performance of the Barrier Algorithms

Something else besides memory contention
Performance Monitoring using PAPI

- PAPI can be used to read performance event counter on both Xeon and Opteron processors
  - Intel
    - BUS_TRANS_MEM
      - Monitor all memory transactions
  - AMD
    - DC_COPYBACK_I
      - Equivalent to the number of cache transactions
    - NB_HT_BUS(x)_DATA
      - The number of HyperTransport data transactions
Memory/Cache Transactions of the Barrier Algorithms

- Software Overhead
- Memory Transaction

- Cache Invalidation
- HyperTransport Transaction

Centralized Barrier/Queue-based Barrier

Number of Threads

1 2 3 4 5
Multi-Threading Strategy

- OpenMP
  - Standard
  - C, C++, Fortran
  - Compiler Dependent

- Hand-written pthread Library
  - Portable
  - Complex
  - Can be a better performer
OpenMP Performance from Different Compilers
QMT and OpenMP
Conclusions

- Multi-core processor based SMP clusters present new challenges
- Memory architecture influences the performance of barrier algorithms
  - Memory contention
  - Cache coherence protocol
- OpenMP is getting better
  - Hand-written multi-threading libraries can perform even better