COSC 7364
Advanced Parallel Computations

Process, Thread and Memory Affinity

Edgar Gabriel
Fall 2009

Multi-core processors: Example (I)

- Intel X7350 quad-core (Tigerton) multi-processor configuration

Socket 0: C0, C1, C8, C9 (L2)
Socket 1: C2, C3, C10, C11 (L2)
Socket 2: C4, C5, C12, C13 (L2)
Socket 3: C6, C7, C14, C15 (L2)

Memory Controller Hub (MCH)

Memory 8 GB/s Memory 8 GB/s Memory 8 GB/s Memory 8 GB/s
Multi-core processors: Example (II)

- AMD 8350 quad-core Opteron (Barcelona): multi-processor configuration
  - It’s a NUMA!

Memory Bandwidth

What happens with data in a cache on a write operation?

- **Write through**—The information is written to both the block in the cache and to the block in the lower-level memory.
- **Write back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - Is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no repeated writes to same location
- WT always combined with write buffers so that don’t wait for lower level memory

Cache Coherence

- Question: What happens if both caches hold a copy of the same data item and
  - CPU 1 modifies the data and CPU 2 tries to read it at the same time?
    - Don’t want a CPU to read ‘old’ values
  - Both CPU’s try to write the same item?
    - Don’t want both CPU’s to have a different view
    - Requires write serialization
Cache Coherence Protocols

Two basic schemes:
- Directory based: the status (shared/not shared) of a cache block is stored in a memory location
- Snooping: each cache block has a copy of the sharing status of the block.
  - Caches are kept on a bus
  - Caches ‘snoop’ the bus in order to determine whether a copy of block has been modified
  - Dominating in today’s PCs

Cache Coherence Protocols (II)

Two forms of implementing cache coherence using bus snooping
- Write invalidate: a write operation by a CPU invalidates the cache blocks of the same data items
  - Other CPU has to re-fetch the data item
- Write update: a write operation by a CPU broadcasts the value on the bus to all other caches
  - Have to track which data items are held by other caches in order to minimize data transfer
Write invalidate vs. Write update

- Multiple writes to the same word (without any read operation in-between) requires multiple broadcasts in the write update protocol, but only one invalidate operation in the write invalidate protocol.
- Multi-word write requires one invalidate operation per cache block for the write-invalidate protocol, requires however the broadcast of every word in the write-update protocol.
- A read after write by different processes might be more efficient by the write update protocol.

→ because of bus and memory bandwidth issues, write invalidate is the dominant protocol.

Cache Coherence Protocols (III)

- A bus connects the different caches.
- In order to perform an invalidate operation, processor needs to acquire bus access:
  - Automatically serializes multiple writes.
  - All writes are complete (e.g. it is not possible, that the first part of a multi-word write contains data from Proc. A and the second part from Proc. B if both write exactly the same memory locations).
- Extra bit on the cache tag indicates whether a cache block is shared.
  - A write operation changes the bit from shared to exclusive.
Consequences for multi-threaded applications

- Threads share the same address space
  - data items might be shared by different processors
  - multiple cores/CPUs have copies of the same data items
  - vulnerable for the cache coherence problem
  - > threads should be as close together as possible
- What happens with a multi-threaded application on a NUMA node?
  - address space is contiguous
  - cache coherence is required between the caches of the different processors, not just between the cores of one processor
  - location of data item very important for the performance of the application

Consequences for multi-threaded applications (II)

- Can multithreaded applications benefit from the increased memory bandwidth of NUMA applications?
  - assuming we are using first-touch policy only: who allocates the data?
    - main thread only: all memory accesses of threads located on a different processor will be remote access. bad.
    - each thread separately: all data access is local. good.
  - assuming a stripe size on how data is allocated on the memory modules (e.g. see libnuma):
    - if the stripe size matches the access pattern of the threads.
      - good.
    - else: bad.
Consequences for multi-process applications

- Each process has its own (virtual) address space
  - no two processors/cores share the same real memory address
  - no cache coherence problems!
  - maps well to NUMA architecture if processes run on different processors and can benefit from the increased memory bandwidth

Why not go with multi-process only?

- Memory footprint of multiple processes can be huge
- Consider a 2-D finite difference code with one row/column of ghostcells
Why not go with multi-process only?

- Estimate of memory overhead due to ghostcells for a 2-D problem
  - $N \times N$ grid points overall
  - $p$ processes
  - each process holds $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}} = \frac{N^2}{p}$
  - plus $4\frac{N}{\sqrt{p}}$ ghostcells
  - keep $N$ constant for a given problem but increase $p$
  - overall memory consumption due to ghostcells:
    $$p \times 4\frac{N}{\sqrt{p}} = 4N\sqrt{p}$$

Why not go with multi-process only?

- Estimate of memory overhead due to ghostcells for a 3-D problem
  - $N \times N \times N$ grid points overall
  - $p$ processes
  - each process holds $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}} = \frac{N^3}{p}$
  - plus ghostcells: $6\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}} = 6N^2\frac{N}{\sqrt{p}}$
  - keep $N$ constant for a given problem but increase $p$
  - overall memory consumption due to ghostcells:
    $$6 p \frac{N^2}{\sqrt{p}^2} = 6 N^2 \sqrt{\frac{N}{p}}$$
Why not go with multi-process only?

Memory overhead due to ghostcells for a 2D/3D matrix of 1 GB

Programming for multi-core

- Each thread/process has an affinity mask
  - Affinity mask specifies what cores the thread is allowed to run on
  - Different threads can have different masks
  - Affinities are inherited across fork()
- Example: 4-way multi-core, without SMT

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Process/thread is allowed to run on cores 0,2,3, but not on core 1

Process migration is costly

- **Default Affinities**
  - Default affinity mask is all 1s: all threads can run on all processors
  - OS scheduler decides what threads run on what core
  - OS scheduler detects skewed workloads, migrating threads to less busy processors
- **Soft affinity:**
  - Tendency of a scheduler to try to keep processes on the same CPU as long as possible
- **Hard affinity:**
  - Affinity information has been explicitly set by application
  - OS has to adhere to this setting

---

**Linux Kernel scheduler API**

Retrieve the current affinity mask of a process

```c
#include <sys/types.h>
#include <sched.h>
#include <unistd.h>
#include <errno.h>

unsigned int len = sizeof(cpu_set_t);
cpu_set_t mask;
pid_t pid = getpid(); /* get the process id of this app */

ret = sched_getaffinity(pid, len, &mask);
if ( ret != 0 )
    printf("Error in getaffinity %d (%s)\n",
        errno, strerror(errno);
for (i=0; i<NUMCPUS; i++) {
    if ( CPU_ISSET(i, &mask) )
        printf("Process could run on CPU %d\n", i);
```
Linux Kernel scheduler API (II)

Set the affinity mask of a process

```c
unsigned int len = sizeof(cpu_set_t);
cpu_set_t mask;
pid_t pid = getpid(); /* get the process id of this app */
/* clear the mask */
CPU_ZERO (&mask);
/* set the mask such that the process is only allowed to
execute on the desired CPU */
CPU_SET (cpu_id, &mask);
ret = sched_setaffinity (pid, len, &mask);
if ( ret != 0 ) {
    printf("Error in setaffinity %d (%s)\n",
           errno, strerror(errno);
}
```

Linux Kernel scheduler API (III)

- Setting thread-related affinity information
  - Use sched_setaffinity with a pid = 0
    - Changes the affinity settings for this thread only
  - Use libnuma functionality
    ```c
    numa_run_on_node();
    numa_run_on_node_mask();
    ```
  - Modifying affinity information based on CPU sockets, not on cores
  - Use pthread functions on most Linux systems
    ```c
    #define __USE_GNU
    pthread_setaffinity_np(thread_t t, len, mask);
    pthread_attr_setaffinity_np (thread_attr_t a, len, mask);
    ```
Windows APIs

- BOOL WINAPI GetProcessAffinityMask(HANDLE hProcess,
  PDWORD_PTR lpProcessAffinityMask,
  PDWORD_PTR lpSystemAffinityMask);
- BOOL WINAPI SetProcessAffinityMask(HANDLE hProcess,
  DWORD_PTR dwProcessAffinityMask);