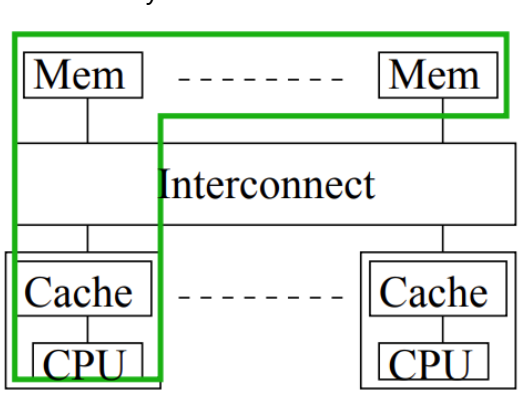
0

Parallel Programming Languages and Systems

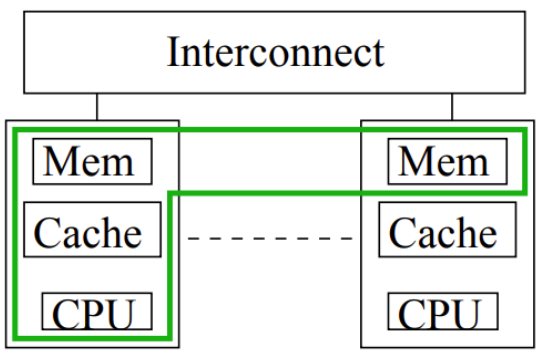
Content

Lecture1 Introduction

1. Programing mechanisms need to balance between:
    1. Conceptual simplicity: it should be "easy" to program correctly and
    2. Performance retention: if our algorithm and architecture are good, we shouldn't lost "too much" in the mapping between them.
2. Express parallelism (concurrency) concisely, correctly nad efficiently in several contexts:
    1. Performance computing: Parallelism is the means to reduce execution time of computationally demanding applications
    2. Distributed computing: Concurrency is the inherent in the nature of the system and we have to express and control it.
    3. Systems programming: It is conceptually simpler to think of a system as being composed of concurrent components, even though these will actuall be executed by time-sharing a single processor.
3. Two traditional parallel models: ***shared variable*** programming and ***message passing*** programming.
4. Two parallel architectures classes:
    1. Shared memory: Processors and physically address the whole memory, usually with support for cache coherency (For example multi-core machines)
    2. Multicomputer: Processors can only physically address their own memory (for example a networked cluster for PCs), which interact with messages across the network.
    3. Systems will spanboth classes (eg cluster of manycore, or network-on-chip manycores like the Intel SCC), and incorporate otehr specialized, constrained parallel hardware such as GPUs.
5. Shared memory architectures



UMA style



NUMA style

1. UMA style: Uniform Memory Access architectures have all memory "equidistant" from all CPUs.
2. NUMA style: Non-UMA performance varies with data location. It is also called ***Distributed Shared Memory*** because memory is physically distributed but logically shared.

6. Memory consistency

    1. Caches improve performance, but raise a memory consistency challenge: in what order should *updates* to memory made by one processor become *visible to others*.

```
) others!                [x==0, y==0]

   x = 2;                                LD R1,#2
                                         ST R1,x  [to cache]

   y = 1;                                LD R1,#1
                                         ST R1,y  [to cache]

   ....

                                         cache write-back of y -> memory

   ....
   if (x<=y)                             LD R1,x  [from cache]
                                         LD R2,y  [memory back to cache]
                                         BGT R1,R2,target

      print("Yes");
```

    . In this case, the processor sees x==2, y==1, however in the main memory x==0, y==1
    2. What and when it is permissible for each processor to see is defined by the ***consistency model***.

7. Consistency model: A effective contract between hardware and software, must be respected by *programmers, compilers and library writers. Different consistency models trade off conceptual simplicity against time/hardware complexity cost.

    1. Sequential consistency: every processor sees the same sequential interleaving of the basic reads and writes. It is intuitive but expensive to implement.
    2. Release consistency: writes are only guaranteed to be visible after program-specified synchronization points (triggered by special machine instructions)Even the ordering as written by one processor between such points may be seen differently by other processors. It is less intuitive but enables faster implementations.

8. Ping-pong effects: the unit of transfer between memory and cache is a cache-line or block, containing several words. False sharing occurs when two logically unconnected variables share the same cache-line. Updates to one cause remote copies of the line, including the other variable to be invalidated, creating very expensive, but semantically undetectable, "ping-pong" effects.

```
                shared int x, y;

          P0                              P1
for 1000000 iterations {        for 1000000 iterations {
   x = ....not touching y .....    y = ....not touching x .....
}                               }
```

    . If x and y are located side by side, they may share the same cache-line

9. Multiconputer Architectures

1. Eahc processor only accesses its own physical address space, so no consistency issues. information is shared by explicit, co-operative message passing.
2. Performance/correctness issues include the semantics of synchronization and constraints on message ordering.

```
P0                                  P1

while (whatever) {                  recv(y, P0);
  x = ...;                          while (y!=-1) {
  send (x, P1);                       ...= ....y....;
}                                     recv(y, P0);
send(-1, P1);                       }
```

In this case the order of messages might not be the same as sending. For example, sending 3, 7, 2, -1 may the received in the ordered of 3, 7, -1, 2, which cause P1 fail to receive 2.

Lecture2 Notation

1. Three well-known patterns:
    1. Bag of tasks
    2. Pipeline
    3. Interacting peers
2. Notation is not a real programming language, just a concise way of expressing what we need mechanisms to say in real languages and libraries.
3. CO notation
    1. Use a pair "co" and "oc" to enclose a block
    2. Indicates creation of a set of activities, for the duration of the enclosed block, with synchronization across all activities at the end of the block.
    3. Also called "fork-join" parallelism. `
    4. Use "//" to separate each statement.
4. Sequential memory consistence (SC)
    1. Ordering of atomic actions (particularly reads and writes to memory) from any one thread have to occur in normal program order.
    2. Atomic actions from different threads are interleaved arbitrarily (i.e. in an unpredictable sequential order, subject only to rule 1), with every thread seeing the same order.
    3. SC execution is like a random stitch, allowing processes to access memory one at a time.
5. Examples of SC

```
a=0;
co
   a=1; // a=2; // b=a+a; ## all at the same time  What is b?
oc
```

1. In this cases b can be 0, 1, 2, 3, 4
2. each single read or write is considered as an atomic action.
3. A possible order is: b reads a(0), a is set to 2, b reads a(2), ..., b=0+2=2

6. Atomic notation

    1. A pair of "<" and ">"
    2. Make the enclosed statements an atomic action
    3. For the above example, b can then be 0, 2, 4
    4. ```
       co
          count++; // count++;
       oc
       ```

       In this example, count++ consists of three steps (read, computer, write). What we actually mean is

       ```
       co
          <count++;> // <count++;>
       oc
       ```

    5. ```
       co [i = 0 to n-1]
          a[i] = a[n-i-1]; ##try to reverse a in parallel
       oc
       ```

       It can work in conditions:

        1. all reads are executed before writes.
        2. Every two elements to be swaped are equal. However, add an atomic notation makes it worse because the first condition won't work.

7. Await Notation

    1. < await (B) S >. The atomic notation is required. It means S must be delayed until B is true within the same atomic action as a successful check of B.
    2. S is not necessarily executed immediately B becomes true. Other threads can get executed, and B can be false again. In this case S should still be waiting.

```
a=0; b=0;
co
  {a=1; other stuff; a=0;}
//
  <await (a==1) b=a;>
oc
```

3.
    1. Can terminate: a=1, b=a, a=0.
    2. Can fail to terminate: a=1, a=0, then thread 2 will wait forever.