Reading Assignment

- This lecture: Notes
- Next lecture: Notes
Outline

Multi-Core Processors

Message Passing Interface (MPI)
Multi-Core Processors

- Widely used nowadays for computer systems and consumer electronics.
  - Single-core processor performance is limited by power consumption and related heat dissipation issues.
  - More processor cores are integrated to improve overall system performance.
- Specification (programming) model: multi-threaded programs
  - Each thread is a sequential program.
  - Implicit data transfer: shared memory.
  - Explicit synchronization: via special instructions, e.g. Compare-And-Swap (CAS), that are supported by dedicated bus(es).
- Multiple high end multi-core processors may be interconnected to support the same specification model.
- For the discussion later, we generally ignore the complexity introduced by caching, virtual memory, and OS scheduling.
Multi-Core Processors

- Widely used nowadays for computer systems and consumer electronics.
  - Single-core processor performance is limited by power consumption and related heat dissipation issues.
  - More processor cores are integrated to improve overall system performance.
- Specification (programming) model: multi-threaded programs
  - Each thread is a sequential program.
  - Implicit data transfer: shared memory.
  - Explicit synchronization: via special instructions, e.g. Compare-And-Swap (CAS), that are supported by dedicated bus(es).
- Multiple high end multi-core processors may be interconnected to support the same specification model.
- For the discussion later, we generally ignore the complexity introduced by caching, virtual memory, and OS scheduling.
Multi-Core Processors

- Widely used nowadays for computer systems and consumer electronics.
  - Single-core processor performance is limited by power consumption and related heat dissipation issues.
  - More processor cores are integrated to improve overall system performance.
- Specification (programming) model: multi-threaded programs
  - Each thread is a sequential program.
  - Implicit data transfer: shared memory.
  - Explicit synchronization: via special instructions, e.g. Compare-And-Swap (CAS), that are supported by dedicated bus(es).
- Multiple high end multi-core processors may be interconnected to support the same specification model.
- For the discussion later, we generally ignore the complexity introduced by caching, virtual memory, and OS scheduling.
Multi-Core Processors

- Widely used nowadays for computer systems and consumer electronics.
  - Single-core processor performance is limited by power consumption and related heat dissipation issues.
  - More processor cores are integrated to improve overall system performance.

- Specification (programming) model: multi-threaded programs
  - Each thread is a sequential program.
  - Implicit data transfer: shared memory.
  - Explicit synchronization: via special instructions, e.g. Compare-And-Swap (CAS), that are supported by dedicated bus(es).

- Multiple high end multi-core processors may be interconnected to support the same specification model.

- For the discussion later, we generally ignore the complexity introduced by caching, virtual memory, and OS scheduling.
Consider a single core with the memory.

- State bits consisting of those in the sequential elements.
  - Micro-architecture-level registers, e.g. IR and those used by pipelining.
  - Instruction-level registers, e.g. PC and EAX
  - Memory words.

- Not all state bits are observable.
  - At instruction-level, micro-architecture-level registers cannot be seen.

- Not all state transitions are observable.
  - An instruction may take multiple cycles to execute.
State Space

- Consider a single core with the memory.
- State bits consisting of those in the sequential elements.
  - Micro-architecture-level registers, e.g. IR and those used by pipelining.
  - Instruction-level registers, e.g. PC and EAX
  - Memory words.
- Not all state bits are observable.
  - At instruction-level, micro-architecture-level registers cannot be seen.
- Not all state transitions are observable.
  - An instruction may take multiple cycles to execute.
Consider a single core with the memory.

State bits consisting of those in the sequential elements.
  - Micro-architecture-level registers, e.g. IR and those used by pipelining.
  - Instruction-level registers, e.g. PC and EAX
  - Memory words.

Not all state bits are observable.
  - At instruction-level, micro-architecture-level registers cannot be seen.

Not all state transitions are observable.
  - An instruction may take multiple cycles to execute.
State Space

- Consider a single core with the memory.
- State bits consisting of those in the sequential elements.
  - Micro-architecture-level registers, e.g. IR and those used by pipelining.
  - Instruction-level registers, e.g. PC and EAX
  - Memory words.
- Not all state bits are observable.
  - At instruction-level, micro-architecture-level registers cannot be seen.
- Not all state transitions are observable.
  - An instruction may take multiple cycles to execute.
Performance Optimizations: Pipelining

- Introduce micro-architecture-level registers to overlap state transitions for individual instructions.
  - Make full use of hardware resources within the chip.
- Give the illusion that an instruction can be completed by one cycle.
- Hazards arise when dependencies are presented.
  - The correct state transition depends on previous state transitions
  - Data dependencies can be resolved by forwarding results.
  - Control dependencies due to branch instructions are more difficult to handle.
Performance Optimizations: Pipelining

- Introduce micro-architecture-level registers to overlap state transitions for individual instructions.
  - Make full use of hardware resources within the chip.
- Give the illusion that an instruction can be completed by one cycle.
- Hazards arise when dependencies are presented.
  - The correct state transition depends on previous state transitions
  - Data dependencies can be resolved by forwarding results.
  - Control dependencies due to branch instructions are more difficult to handle.
Performance Optimizations: Pipelining

- Introduce micro-architecture-level registers to overlap state transitions for individual instructions.
  - Make full use of hardware resources within the chip.
- Give the illusion that an instruction can be completed by one cycle.
- Hazards arise when dependencies are presented.
  - The correct state transition depends on previous state transitions
  - Data dependencies can be resolved by forwarding results.
  - Control dependencies due to branch instructions are more difficult to handle.
Performance Optimizations: Out-Of-Order Execution

- When it takes much more cycles to read/write memory, pipelining becomes less effective.
  - A load/store instruction could block all following instructions in the pipeline.
  - Communication takes time!
- Out-Of-Order (OOO) execution
  - Allow later instructions to execute as long as there is no data/control dependencies.
  - A load/store buffer is usually necessary to buffer memory writes and to forward memory reads for better performance.
- Correctness
  - State transitions that are not observable could be replaced for better performance as long as observable state transitions are the same (by the processor core).
Performance Optimizations: Out-Of-Order Execution

- When it takes much more cycles to read/write memory, pipelining becomes less effective.
  - A load/store instruction could block all following instructions in the pipeline.
  - Communication takes time!
- Out-Of-Order (OOO) execution
  - Allow later instructions to execute as long as there is no data/control dependencies.
  - A load/store buffer is usually necessary to buffer memory writes and to forward memory reads for better performance.
- Correctness
  - State transitions that are not observable could be replaced for better performance as long as observable state transitions are the same (by the processor core).
Performance Optimizations: Out-Of-Order Execution

- When it takes much more cycles to read/write memory, pipelining becomes less effective.
  - A load/store instruction could block all following instructions in the pipeline.
  - Communication takes time!
- Out-Of-Order (OOO) execution
  - Allow later instructions to execute as long as there is no data/control dependencies.
  - A load/store buffer is usually necessary to buffer memory writes and to forward memory reads for better performance.
- Correctness
  - State transitions that are not observable could be replaced for better performance as long as observable state transitions are the same (by the processor core).
Out-Of-Order Memory Updates

- With OOO on a single core, the overlaped state bits of the memory and the load/store buffer can be safely replaced by the later for better performance.
  - When the load/store buffer *actually* updates the memory becomes irrelevant for correctness and may only depend on performance decisions.

Consider code below

```assembly
load R1, Mem[100]
load R2, Mem[104]
mul R1, R1, R3
store R1, Mem[200]
mul R2, R2, R4
store R2, Mem[300]
load R5, Mem[200]
load R6, Mem[300]
mul R7, R5, R6
store R7, Mem[204]
```

- If a memory transaction may take 8 bytes, it is possible for R7 to be written back to Mem[204] earlier (together with Mem[200]) than that R2 is written back to Mem[300].
Out-Of-Order Memory Updates

- With OOO on a single core, the overlaped state bits of the memory and the load/store buffer can be safely replaced by the later for better performance.
  - When the load/store buffer actually updates the memory becomes irrelevant for correctness and may only depends on performance decisions.

Consider code below

```assembly
load R1, Mem[100]
load R2, Mem[104]
mul R1, R1, R3
store R1, Mem[200]
mul R2, R2, R4
store R2, Mem[300]
load R5, Mem[200]
load R6, Mem[300]
mul R7, R5, R6
store R7, Mem[204]
```

- If a memory transaction may take 8 bytes, it is possible for R7 to be written back to Mem[204] earlier (together with Mem[200]) than that R2 is written back to Mem[300].
Implications

- Threads on different cores communicate via shared variable.
- Assume initially `ready=0` and `data=0`.
- The above code pieces may print 0 instead of 100.
  - There is no guarantee Core 1 will actually update `ready` before `data`.

<table>
<thead>
<tr>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>data = 100;</td>
<td></td>
</tr>
<tr>
<td>ready = 1;</td>
<td></td>
</tr>
<tr>
<td>while (ready == 0)</td>
<td></td>
</tr>
<tr>
<td>wait_a_while();</td>
<td></td>
</tr>
<tr>
<td>print(data);</td>
<td></td>
</tr>
</tbody>
</table>
Implications

- Threads on different cores communicate via shared variable.
- Assume initially \( \text{ready} = 0 \) and \( \text{data} = 0 \).
- The above code pieces may print 0 instead of 100.
  - There is no guarantee Core 1 will actually update \( \text{ready} \) before \( \text{data} \).

Core 1

```c
data = 100;
ready = 1;
```

Core 2

```c
while (ready == 0)
    wait_a_while();
print(data);
```
Implications

▶ Threads on different cores communicate via shared variable.
▶ Assume initially \( \text{ready}=0 \) and \( \text{data}=0 \).
▶ The above code pieces may print 0 instead of 100.
  ▶ There is no guarantee Core 1 will actually update \( \text{ready} \) before \( \text{data} \).

Core 1

\[
\begin{align*}
\text{data} &= 100; \\
\text{ready} &= 1;
\end{align*}
\]

Core 2

\[
\begin{align*}
\text{while} & \ (\text{ready} == 0) \\
& \quad \text{wait\_a\_while();} \\
& \quad \text{print(data);}
\end{align*}
\]
Multi-threaded programs can fully exploit the underlying multi-core processor(s) platform.
  ▶ No overhead for shared memory communication on and below presentation layer.
  ▶ Efficient!

Synchronization must be used to safely publish memory writes among multiple cores.
  ▶ Use locks: may hurt performance or even lead to deadlock.
  ▶ Use other synchronization primitives: not intuitive.
  ▶ Productivity is a concern.
Multi-threaded programs can fully exploit the underlying multi-core processor(s) platform.

- No overhead for shared memory communication on and below presentation layer.
- Efficient!

Synchronization must be used to *safely publish* memory writes among multiple cores.

- Use locks: may hurt performance or even lead to deadlock.
- Use other synchronization primitives: not intuitive.
- Productivity is a concern.
Multi-Core Processors

Message Passing Interface (MPI)
Message Passing Interface (MPI)

- Standardized during 90’s to support programming over parallel computers.
  - As a library that different languages can utilize, usually Fortran, C, and C++.
- Specification model
  - Explicit communication (data transfer and synchronization) via message passing.
  - Multiple processes, each being a sequential program.
  - All processes are based on the same binary program, though their actual behaviors may differ as each process knows where it maps to.
- May utilize multi-core processors by spawning multiple processes.
Message Passing Interface (MPI)

- Standardized during 90’s to support programming over parallel computers.
  - As a library that different languages can utilize, usually Fortran, C, and C++.
- Specification model
  - Explicit communication (data transfer and synchronization) via message passing.
  - Multiple processes, each being a sequential program.
  - All processes are based on the same binary program, though their actual behaviors may differ as each process knows where it maps to.
- May utilize multi-core processors by spawning multiple processes.
Message Passing Interface (MPI)

- Standardized during 90’s to support programming over parallel computers.
  - As a library that different languages can utilize, usually Fortran, C, and C++.
- Specification model
  - Explicit communication (data transfer and synchronization) via message passing.
  - Multiple processes, each being a sequential program.
  - All processes are based on the same binary program, though their actual behaviors may differ as each process knows where it maps to.
- May utilize multi-core processors by spawning multiple processes.
MPI defines standard ways to represent host data types.

- **Primitive types**, e.g.,
  - `int` -> `MPI_INT`
  - `double` -> `MPI_DOUBLE`
  - `char` -> `MPI_CHAR`

- **User defined types**, e.g. C struct
  ```c
  int MPI_Type_create_struct(
      int count,
      int array_of_blocklengths[],
      MPI_Aint array_of_displacements[],
      MPI_Datatype array_of_types[],
      MPI_Datatype *newtype
  );
  ```
MPI defines standard ways to represent host data types.

**Primitive types, e.g.,**

```
int    -> MPI_INT
double -> MPI_DOUBLE
char   -> MPI_CHAR
```

**User defined types, e.g. C struct**

```
int MPI_Type_create_struct(
    int count,
    int array_of_blocklengths[],
    MPI_Aint array_of_displacements[],
    MPI_Datatype array_of_types[],
    MPI_Datatype *newtype
);
```
MPI defines standard ways to represent host data types.

Primitive types, e.g.,

```c
int -> MPI_INT
double -> MPI_DOUBLE
char -> MPI_CHAR
```

User defined types, e.g. C struct

```c
int MPI_Type_create_struct(
    int count,
    int array_of_blocklengths[],
    MPI_Aint array_of_displacements[],
    MPI_Datatype array_of_types[],
    MPI_Datatype *newtype
);
```
struct Particle {char c; double d[6]; char b[7];} particle[1000];

MPI_Datatype elem_type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_CHAR};
int block_len[3] = {1, 6, 7};
MPI_Aint disp[3] = {
    &particle[0].c-&particle[0],
    &particle[0].d-&particle[0],
    &particle[0].b-&particle[0]};

MPI_Datatype ParticleType;
MPI_Type_create_struct(3, block_len, disp, elem_type, &ParticleType);
MPI_Type_commit(&ParticleType);

- ParticleType will then contain meta-data to encode/decode
  Particle to/from untyped byte messages.
Send and Recv

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype, 
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, 
             int source, int tag, MPI_Comm comm, MPI_Status *st);
```

- For communications from process source to process send.
  - Session layer: multiple channels among the same pair of processes are identified by tag.
  - Transport Layer: reliable and ordered message of arbitrary size.
    - Allow to use application types directly with proper meta-data.
    - As a comparison, TCP is reliable and ordered but only a stream of bytes.
Send and Recv

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *st);
```

▶ For communications from process source to process send.
▶ Session layer: multiple channels among the same pair of processes are identified by tag.
▶ Transport Layer: reliable and ordered message of arbitrary size.
  ▶ Allow to use application types directly with proper meta-data.
  ▶ As a comparison, TCP is reliable and ordered but only a stream of bytes.
Send and Recv

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *st);
```

» For communications from process source to process send.
» Session layer: multiple channels among the same pair of processes are identified by tag.
» Transport Layer: reliable and ordered message of arbitrary size.
  » Allow to use application types directly with proper meta-data.
  » As a comparison, TCP is reliable and ordered but only a stream of bytes.
Send and Recv (cont.)

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *st);

int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) {
    MPI_Send(particle, 1000, ParticleType, 1, 123, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Status st;
    MPI_Recv(particle, 1000, ParticleType, 0, 123, MPI_COMM_WORLD, &st);
}

- Note that the same program is used as both the sender and receiver.
  - There will be 2 MPI processes, one with myrank=0 and the other with myrank=1.
```
Synchronization

```c
int MPI_Barrier(MPI_Comm comm);
MPI_Barrier(MPI_COMM_WORLD);
```

- All processes are blocked at the barrier until every of them has reached it.
- MPI also supports many other synchronization and data transfer primitives.
int MPI_Barrier(MPI_Comm comm);

MPI_Barrier(MPI_COMM_WORLD);

▶ All processes are blocked at the barrier until every of them has reached it.

▶ MPI also supports many other synchronization and data transfer primitives.
MPI is mainly for building parallel applications.

- A distributed application usually relies on different programs running on different computers.
- It is generally not possible to guarantee an MPI program to be deadlock free.
- Difficult to debug, especially when a large number of processes are involved. Productivity is a concern.
- Explicit checkpointing is necessary to prevent loss of work due to any failure.
- A concern for any MoC that could potentially be scaled to a large number of computers.
Discussions

» MPI is mainly for building parallel applications.
  » A distributed application usually relies on different programs running on different computers.

» It is generally not possible to guarantee an MPI program to be deadlock free.
  » Difficult to debug, especially when a large number of processes are involved. Productivity is a concern.

» Explicit checkpointing is necessary to prevent loss of work due to any failure.
  » A concern for any MoC that could potentially be scaled to a large number of computers.
Discussions

- MPI is mainly for building parallel applications.
  - A distributed application usually relies on different programs running on different computers.
- It is generally not possible to guarantee an MPI program to be deadlock free.
  - Difficult to debug, especially when a large number of processes are involved. Productivity is a concern.
- Explicit checkpointing is necessary to prevent loss of work due to any failure.
  - A concern for any MoC that could potentially be scaled to a large number of computers.