Parallel Software Design

Hwansoo Han
Level of Parallelism

- Task Level Parallelism
- Data Level Parallelism
- Instruction Level Parallelism
- Parallel Loop
- Multi-threading
- Out-of-order Execution
Instruction Level Parallelism

- Mostly compiled for sequential processors
  - An executable binary contains a sequence of instructions
  - Expect to execute one machine instruction by another in the order specified in the given executable binary
- Can execute instructions in any order
  - But the result should be the same as the sequential execution
    - Superscalar
    - Branch prediction
    - Out-of-order execution

```
1: mul  r1 = r2, r3
2: add  r2 = r4, r5
3: sub  r3 = r5, r6
```
Data Level Parallelism

- Process large data with the same operations
  - No dependences among different data
  - Typically, done in loops but the order of iterations can be shuffled (associativity)

```c
for (i=0; i<N; i++)
a[i] = (b[i] + c[i])/2;
```

```c
parallel_for (i=0; i<N; i++)
a[i] = (b[i] + c[i])/2;
```
Task Level Parallelism

- High level tasks are parallel from the design
  - Multiple requests to `inetd` are parallel each other
  - Web server handles multiple web requests in parallel

```c
while(1) {
    a = get_request();
    process(a);
}
```

```c
while(1) {
    a = get_request();
    create_thread(process, &a);
}
```
Parallelism in Applications

- Implicit parallel programs
  - Hardware support
  - Compiler support (automatic parallelization)

- Explicit parallel programs
  - Programmers are responsible

Diagram:
- Parallelism in Program
  - Implicit
    - Hardware
      - Superscalar
      - Out-of-order Processors
  - Explicit
    - Compiler
      - Parallel Architectures
Automatic Parallelization

- Fully automatic parallelization
  - Compiler analyzes the source code and finds opportunities
  - Loops are frequent targets for parallelization
    - e.g. Intel compiler (icc –parallel), SUIF compiler, …
  - Find loops which have no loop-carried dependences
    - e.g. GCD test, Fourier-Motzkin elimination, Omega test, …

- Programmer directed parallelization
  - “Compiler directives” explicitly tell compiler how to parallelize
    - e.g. OpenMP compiler, UPC, compiler, HPF compiler, …
  - No check for loop-carried dependences
Automatic Parallelization

- **Pros**
  - Start with an existing serial code
  - Easy to try out

- **Limitations**
  - Wrong results may be produced (if not correctly directed)
  - Performance may actually degrade
  - Limited to a subset of code (mostly loops)
  - May not parallelize if code is too complex to analyze
  - Cannot handle pointers very well
Design Parallel Programs

- Understand the problem
  - Is the nature of the code parallel?
    - Non-parallel algorithm such as Fibonacci series, \( F(n) = F(n-1) + F(n-2) \).

- Identify the hotspots
  - Need to improve the most time-consuming part for overall performance
    - Profiler (e.g. GNU gprof, instrumentation tools – PIN, valgrind, …)
    - Performance analysis tools (Intel VTune, AMD CodeAnalyst, …)

- Identify the bottlenecks (inhibitors to parallelism)
  - Disproportionally slow parts
  - Restructuring / elimination / different algorithm
Decomposition for Concurrency

- **Task decomposition**
  - How can a problem be decomposed into tasks that can execute concurrently?
  - Stream of instructions is broken into sequences (tasks)
    - Functional decomposition (divide-and-conquer, pipeline, event-based coordination, …)
    - Loop-splitting (parallel loops)
**Decomposition for Concurrency**

- **Data decomposition**
  - How can a program’s data be decomposed into units that can be operated on relatively independently?
  - Data required by tasks are decomposed into distinct chunks
    - Array-based computation
    - Recursive data structures (e.g. list, tree, kd-tree, quad-/oct-tree, …)
Program Structures

- **SPMD (single program, multiple data)**
  - Operations on each processor are similar
  - Each processor works on different data by obtaining proc ID

- **Master/worker**
  - Workloads associated with tasks are highly variable
  - Parallel work is not a simple loop to handle with OpenMP schedule (dynamic or guided)

- **Parallel loops**
  - Computationally intensive loops

- **Dynamic tasks**
  - Recursive algorithms
    - divide-and-conquer with recursive data structures
Debugging

- Make sure applications
  - Have no logical bugs in the design stage
  - Are fully debugged in a sequential mode

- Sources of parallelization bugs
  - Missing synchronizations (e.g. mutual exclusion, barrier)
  - Improper locking order
  - Improper message passing order (in MPI programs)
Atomicity

- Certain sequence of statements should be executed as an atomic unit
  - lock/unlock shared data is not enough
  - Somewhat similar to “synchronized” in Java

```java
Deposit(int money)
{
    t = shared_account;
    t += money;
    shared_account = t;
}
```

```java
Deposit(int money)
{
    lock(X);
    t = shared_account;
    unlock(X);
    t += money;
    lock(X);
    shared_account = t;
    unlock(X);
}
```

```java
atomic
Deposit(int money)
{
    t = shared_account;
    t += money;
    shared_account = t;
}
```
Atomicity Bugs

- Should’ve locked the whole things
  - Correlated operations
  - Correlated variables

Deposit(int money)
{
  lock(X);
  t = shared_account;
  unlock(X);
  t += money;
  lock(X);
  shared_account = t;
  unlock(X);
}

Empty()
{
  lock(T);
  memset(tab.entry, 0, N);
  unlock(T);

  lock(E);
  tab.empty = True;
  unlock(E);
}

Fill()
{
  lock(T);
  tab.entry[idx] = item;
  unlock(T);

  lock(E);
  tab.empty = False;
  unlock(E);
}
Data Race

- Not properly synchronized
  - Non-deterministic, Timing dependent
  - Cause data corruption, malfunction, crash
  - Difficult to detect, reproduce, & eliminate

- Many program actually have races
  - Caused due to programming errors
  - Fail to observe locking discipline

- Dynamic detection tools (based on binary instrumentation)
  - Lockset algorithm (ERASER)
  - Virtual clock (Intel Thread Checker)
  - Multi-variable correlation (MUVI)
Lockset Algorithm

- Lockset algorithm in ERASER
  - Dynamic lock discipline checker
    - The same lock(s) should be held when shared data is accessed
    - Need to infer which locks protect which data items

```java
foreach v, init C(v) to set of all locks

foreach access to v by thread t
    C(v) = C(v) ∩ locks_held(t)
    if (C(v) = {}) issue warning

lock(mu1); v = v + 1; unlock(mu1);
lock(mu2); v = v + 1; unlock(mu2);
```

[Need special treat]
- Stack variables (thread-local)
- Distinguish reads and writes
- Single writer, multiple readers
- Barriers
Virtual Clock

- Virtual clock
  - N threads create a vector of N local clocks
- Happens-before relation
  - Partial ordering across thread segments

```
T1
v = v + 1
lock(s)
[2,0]
unlock(t)
[1,6]
unlock(t)
[0,4]
[1,5]
lock(t)
T2
[1,0]
[0,0]

T1
v = v + 1
lock(s)
[0,0]
[0,4]
unlock(s)
[1,5]
lock(t)
[1,0]
unlock(t)
[0,4]
[0,5]
unlock(t)
[0,6]

T2
```

Data race
Deadlock

- Cycle in locking graph may lead to deadlock
  - Thread 1
    ```
    {  
      lock(X);
      lock(Y);
    }
    ```
  - Thread 2
    ```
    {  
      lock(Y);
      lock(Z);
    }
    ```
  - Thread 3
    ```
    {  
      lock(Z);
      lock(X);
    }
    ```

- Enforce canonical order of lock $\Rightarrow$ deadlock free
  - Acquire in increasing order
  - Release in decreasing order

- Not always easy to enforce this rule
Intel Thread Checker

- Dynamically check
  - Data races
  - Dead locks
- Exploits both
  - Virtual clock
  - Lockset
- Detection w/ limited history
- Tradeoff for speed and space
- Cannot detect all races
Summary

- Parallel SW
  - Automatic parallelization
  - Explicit parallel SW

- Design of parallel SW
  - Task decomposition
  - Data decomposition

- Debugging parallel SW
  - Data race
  - Dead lock