Run-Time Support for the Automatic Parallelization of Java Programs

Bryan Chan and Tarek S. Abdelrahman

The Edward S. Rogers Sr.
Department of Electrical and Computer Engineering
University of Toronto

August 21, 2001

{chanb, tsa}@eecg.toronto.edu
Overview

- The zJava system performs automatic parallelization of Java programs.

- Combine compile-time analysis with run-time support to automatically extract and exploit parallelism.

- Experimental results show that this approach is viable.
Existing Parallelizing Compilers

DO I=1,100
  DO J=1,100
    A(I,J) = ...
  END DO
END DO

Focus on loop-level parallelism:
- static arrays
- compile-time known bounds

But modern programs are object-oriented (C++, Java):
- pointers and objects
- dynamic data structures
- recursion

Existing techniques are insufficient:
- pointer/alias analysis [Deutsch, 1994; Ghiya and Hendren, 1998]
- escape analysis [Choi et al., 1999; Whaley, 1999]
The zJava System

- Exploit parallelism at the method level.

```java
float m, b, x, y;
void m1() {
    ...
    m2();
    ...
    y = m * x + b;
}
void m2() {
    ...
    m3();
    ...
}
void m3() {
    ...
    x = ...;
    ...
}
```

Must synchronize threads to maintain sequential execution semantics!
A Novel Approach

- A combined compile-time/run-time approach:
  - a compiler captures data accesses using a concise notation
  - a run-time system creates parallel threads for method calls and synchronizes resulting threads according to data dependence

- A trade-off between performance and aggressiveness
Outline of the Talk

- Overview of the system
- Representation of data accesses
- Run-time discovery and exploitation of parallelism
- Preliminary experimental results
- Conclusions and future work
Representation of Data Accesses

• Symbolic access paths

```java
class Point {
    static Point result;
    float x, y;
    void midPoint(Point p) {
        result.x = (this.x + p.x)/2;
        result.y = (this.y + p.y)/2;
    }
}
```

• The data access summary of a method:
  - lists shared variables read and/or written
  - resolved at run-time to find actual shared objects
• Support for arrays and recursive data structures
**The Registry**

- Associates each actual shared variable with a region node:
  - a lock that allows multiple concurrent reads but exclusive writes
  - thread list

- Updated dynamically as variables get (de-)allocated and threads fork and terminate

```c
float m, b, x, y;
void m1() {
    m2();
    ...
    y = m * x + b;
}
void m2() {
    m3();
    ...
}
void m3() {
    x = ...;
}
```
Ordering Threads in the Registry

- Prior to a method call, the caller creates and prepares a new thread.
  - The new thread is queued into the thread lists of regions accessed by the child thread.
  - Thread lists are kept in sequential execution order.
  - New threads are inserted into the thread lists right in front of their parents, after any earlier siblings.
Synchronization Using the Registry

- Compiler inserts calls to lock acquisition and release routines.
- Before the first use of a shared variable, acquire the lock:
  - readers at the head of the thread list can proceed simultaneously
  - a writer must wait until it is at the head of the thread list
- After the last use of a shared variable, release the lock:
  - signals the next waiting thread
Implementation & Preliminary Results

- Implemented the run-time system in Java:
  - 40 classes, 9857 lines of code

- Experimented on a 4-CPU SUN Ultra-4 multiprocessor:
  - MatrixMultiply: a object-oriented matrix multiplication program
  - 15-Puzzle: a puzzle solver using IDA*
Performance of Matrix Multiply

- One method call (thread) to compute each row of the result.

- Performance increases linearly as processors are added.
Performance of 15-Puzzle

- Adapted from a benchmark obtained from D. Szafron [Hui et al., 2000]
  - reverted to a sequential program, then automatically parallelized
- Iterative Deepening A* algorithm solves the 15-puzzle problem:
  - search tree expanded sequentially to a threshold level, then each sub-tree rooted at that level is expanded independently by a thread

- Reported results in [Hui et al., 2000]: 4.4 @ 8 CPUs (vs. 2.95 @ 4 CPUs)
Conclusions

• Explored a new approach to automatic parallelization:
  – combined compile/run-time approach
  – exploits method-level parallelism

• Our system creates threads for method calls and synchronizes those threads to maintain sequential execution semantics.

• Implemented the run-time components of the system.

• Experimental results show that the approach gives performance comparable to manual parallelization, despite the run-time overhead.
Future Work

• Complete the implementation

• Evaluation of the run-time system with large applications

• Explore optimization techniques:
  – Selective parallelization
  – Method versioning
  – Proxy synchronization
  – Automatic granularity control
Automatic Parallelization

- Advantages:
  - ease of use
  - correctness
  - portability
  - efficiency

Sequential source code

Parallel/multithreaded executable program
Representation of Data Accesses

class Point {
    static int nextId = 0;
    int id;
    float x, y;
    Point midPoint(Point p) {
        Point r = new Point();
        r.x = (this.x + p.x) / 2;
        r.y = (this.y + p.y) / 2;
        r.id = nextId++;
        return r;
    }
}

• **Symbolic access paths** address shared data:
  – global variables
  – parameters
  – (recursive) field variables
  – array slices

• The **data access summary** of a method includes:
  – a list of variables it shares with other methods, an whether each is read or written
  – summaries of methods it calls

($0.x$, read)
($0.y$, read)
($1.x$, read)
($1.y$, read)
(Point.nextId, write)
Recursive/Array Data Accesses

class LinkedList {
    int data;
    LinkedList next;
    void incrementAll() {
        this.data++;
        this.next.incrementAll();
    }
}

class Array {
    int data[];
    void incrementAll() {
        for (int i = 0; i < this.data.length; i++)
            this.data[i]++;
    }
}
Resolution and Expansion

\[ p \text{.midPoint}(q); \]

\[ q.x \]

resolution

\[ q.x \]

this.incrementAll();

\[ \text{this}.next*.data \]

resolution

\[ \text{this}.next*.data \]

expansion

\[ \text{this}.data \]

\[ \text{this}.next.data \]

\[ \text{this}.next.next.data \]
Performance of ReadTest

- Parallel threads perform independent reads of the same data.
- Measures the upper bound on speedup.

- Coarse-grain parallelism yields the most benefit.
- Too many concurrent threads lead to performance degradation.
Performance of Matrix Multiply

- One method call (thread) to compute each row of the result.

- Performance increases linearly as processors are added.

- Larger matrices require more coarse-grain threads, leading to better speedup.
The 15-Puzzle Benchmark

- Adapted from a benchmark used in a thread granularity control application [Hui et al., 2000]

- Search algorithm used: Iterative Deepening A*

- Fork N threads at the first frontier