High Performance Runtime for Next Generation Parallel Programming Languages

Vivek Kumar¹, Stephen M Blackburn¹, David Grove², Daniel Frampton¹, ³

¹ The Australian National University
² IBM T.J. Watson Research
³ Microsoft
Hardware and Software Today

“Hey, Joe. Would you eat another apple? I need to add one more core to this computer.”
The Challenge

• Productivity

• Performance

• Portability
Options?

• Productivity
  • Language based features to expose parallelism – X10, Cilk, Habanero-Java etc

• Performance
  • Work–stealing scheduling

• Portability
  • Managed runtime to hide the hardware complexities
Thesis Statement

High performance languages are using managed platforms for productivity and portability, but performance is inadequate. By exploiting and extending the underlying mechanisms of managed runtimes, implementation of these languages will be able to deliver scalability and performance at the levels necessary for widespread uptake.
Contributions
Contributions
Contributions
Contributions

High Performance  High Productivity
Contributions

High Performance

High Productivity

Highly Competitive
Understanding Work–Stealing
Understanding Work–Stealing
Understanding Work–Stealing
Methodology

• Hardware Platform
  – 2x8 cores Intel Xeon E5-2450

• Software Platform
  – Jikes RVM (3.1.3)

• Benchmarks
  – UTS, BarnessHut, FFT, Jacobi, LUDeomposition, JGF_SeriesTest, HeatDiffusion, PointCorrelation, NQueens, Matmul, CilkSort and Fibonacci
    • To evaluate performance
  – JMetal (sourceforge project with 327 Java files)
    • To evaluate the productivity of our system
Big...... But How Big ??
Motivating Analysis

Sequential Overhead

![Bar chart showing sequential overhead comparisons.](image)

- Habanero-Java: 3.7x
- ManagedX10: 2.5x
- Fork-Join: 1.6x

High Performance Runtime for Next Generation Parallel Programming Languages | Kumar
Steal to Task Ratio

![Graph showing the Steal to Task Ratio with different shades and lines for various thread counts from 2 to 16. The x-axis represents Threads ranging from 2 to 16, and the y-axis represents values from $10^{-1}$ to $10^{-8}$. Each line corresponds to a different thread count, with the values stabilizing at different points along the y-axis.](image-url)
Insights

- Move the overheads from common case to the rare case
- Re-use existing mechanisms inside modern managed runtimes
foo() {
    finish {
        async X = S1();
        Y = S2();
    }
}

- Yieldpoint mechanism
- On-stack replacement
- Java try/catch exceptions
- Dynamic code patching
Sequential Overhead

- Habanero-Java: 3.7x
- ManagedX10: 2.5x
- Fork-Join: 1.6x
- TryCatchWS: 7%

High Performance Runtime for Next Generation Parallel Programming Languages | Kumar
Steal Rate

Steals per milli-seconds

Threads

Motivating Analysis

High Performance Runtime for Next Generation Parallel Programming Languages | Kumar
Dynamic Overhead

![Graph showing dynamic overhead vs threads]

- X-axis represents the number of threads.
- Y-axis represents the dynamic overhead in percentage.

The graph illustrates the increasing dynamic overhead with an increase in the number of threads.
Insights

• Still the same
  – Re-use existing mechanisms inside modern managed runtimes
Return Barrier

Hijack a return and bridge to some other method
Dynamic Overhead

For threads=16

Dynamic overhead (%)

geomean

Old  New
Productivity in a Large Code Base

- Project with several hundred files
- Multiple dependencies (inheritance…)
- Achieving parallelism
  - Minimal changes
  - Track fields with atomic updates
  - Avoid deadlocks
Java Language Annotations

• Annotate and leave the rest on compiler
• Parallelism
  – syncsteal {...}
  – steal {...}
• Data centric concurrency control (Dolby et al. 2012)
  – @Atomicsets(X)
  – @Atomic(X)
  – @AliasAtomic(Y=this.X)
So Where Do We Stand …?
Work–Stealing Performance

![Graph showing speedup over sequential execution with different algorithms and thread counts. The graph plots speedup on the y-axis and thread count on the x-axis. The algorithms compared are Habanero-Java, Fork-Join, ManagedX10, and TryCatchWS. The graph demonstrates the performance improvements of work-stealing strategies in parallel computing tasks.](image-url)
Work–Stealing Performance

![Graph showing speedup over sequential execution for different work-stealing algorithms.](UTS)
Summary and Conclusion

- Work-stealing overheads – sequential and dynamic
- Reused existing mechanisms inside modern managed runtimes
  - Yieldpoint mechanism
  - On-stack replacement
  - Java try/catch exception handling
  - Dynamic code patching
  - Return barrier
- Effectively eliminated sequential overhead (only 7%)
- Halved the dynamic overhead
- Annotations in Java to generate work-stealing calls and synchronization blocks