Java Performance Analysis for Scientific Computing

Roldan Pozo
Leader, Mathematical Software Group
National Institute of Standards and Technology
USA

UKHEC: Java for High End Computing
Nov. 20th, 2000
Background: Where we are coming from...

- National Institute of Standards and Technology
  - US Department of Commerce
  - NIST (3,000 employees, mainly scientists and engineers)
- Middle to large-scale simulation modeling
- Fortran and C/C++ applications
- Cray C-90, IBM SP2, SGI, Alpha/PC clusters
NIST Applied and Computational Mathematics Division

- Algorithms for simulation and modeling
- Consultants to NIST scientist and engineers
- Computational linear algebra
- Numerical solution of PDEs
- Special functions
- Multigrid and hierarchical methods
- Numerical Optimization
- Monte Carlo simulations
Why Java?

- Portability of the Java Virtual Machine (JVM)
- Safe, minimize memory leaks and pointer errors
- Network-aware environment
- Parallel and Distributed computing
  - Threads
  - Remote Method Invocation (RMI)
- Integrated graphics
- Widely adopted
  - embedded systems, browsers, appliances
  - being adopted for teaching, development
Why *not* Java?

- **Performance**
  - bytecode interpreters too slow
  - poor optimizing compilers
  - virtual machine
- **lack of scientific software**
  - computational libraries
  - numerical interfaces
  - major effort to port from f77/C
Java Benchmarking: What are we really measuring?

- language vs. virtual machine (VM)
- Java -> bytecode translator
- bytecode execution (VM)
  - interpreted
  - just-in-time compilation (JIT)
  - adaptive compiler (HotSpot)
- underlying hardware
Making Java fast(er)

• Native methods (JNI)
• stand-alone compliers (.java -> .exe)
• modified JVMs
  – (fused mult-adds, bypass array bounds checking)
• aggressive bytecode optimization
  – JITs, flash compilers, HotSpot
• bytecode transformers
• concurrency
Computational Linear Algebra

- Time-consuming portion of PDE solvers and optimization problems
- Basic matrix/vector operations (BLAS) often comprise major portion of cycles
- Key: optimize BLAS
Matrix multiply
(100% Pure Java)

* Pentium II 1.500Mhz; java JDK 1.2 (Win98)
Optimizing Java linear algebra

- Use native Java arrays: A[][]
- algorithms in 100% Pure Java
- exploit
  - multi-level blocking
  - loop unrolling
  - indexing optimizations
  - maximize on-chip / in-cache operations
- can be done today with javac, jview, J++, etc.
Matrix Multiply: data blocking

- 1000x1000 matrices (out of cache)
- Java: 181 Mflops
- 2-level blocking:
  - 40x40 (cache)
  - 8x8 unrolled (chip)
- subtle trade-off between more temp variables and explicit indexing
- block size selection important: 64x64 yields only 143 Mflops

* Pentium III 500Mhz; Sun JDK 1.2 (Win98)
Matrix multiply optimized
(100% Pure Java)

Matrix size (NxN) vs. Mflops

(i,j,k) optimized

* Pentium II 1.500Mhz; java JDK 1.2 (Win98)
Sparse Matrix Computations

- unstructured pattern
- coordinate storage (CSR/CSC)
- array bounds check cannot be optimized away
## Sparse matrix/vector Multiplication

(Mflops)

<table>
<thead>
<tr>
<th>Matrix size (nnz)</th>
<th>C/C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>371</td>
<td>43.9</td>
<td>33.7</td>
</tr>
<tr>
<td>20,033</td>
<td>21.4</td>
<td>14.0</td>
</tr>
<tr>
<td>24,382</td>
<td>23.2</td>
<td>17.0</td>
</tr>
<tr>
<td>126,150</td>
<td>11.1</td>
<td>9.1</td>
</tr>
</tbody>
</table>

*266 MHz PII, Win95: Watcom C 10.6, Jview (SDK 2.0)*
Java Benchmarking Efforts

- Caffeine Mark
- SPECjvm98
- Java Linpack
- Java Grande Forum Benchmarks
- SciMark
- Image/J benchmark
- BenchBeans
- VolanoMark
- Plasma benchmark
- RMI benchmark
- JMark
- JavaWorld benchmark
- ...

...
SciMark Benchmark

• Numerical benchmark for Java, C/C++
• results for over 1,000 platforms
• composite results for five kernels:
  – FFT (complex, 1D)
  – Successive Over-relaxation
  – Monte Carlo integration
  – Sparse matrix multiply
  – dense LU factorization
SciMark 2.0 results

- Intel PIII (600 MHz), IBM 1.3, Linux
- AMD Athlon (750 MHz), IBM 1.1.8, OS/2
- Intel Celeron (464 MHz), MS 1.1.4, Win98
- Sun UltraSparc 60, Sun 1.1.3, Sol 2.x
- SGI MIPS (195 MHz) Sun 1.2, Unix
- Alpha EV6 (525 MHz), NE 1.1.5, Unix
SciMark: Java vs. C
(Sun UltraSPARC 60)

* Sun JDK 1.2, javac -0; Sun cc -0; Solaris 2.x
SciMark (large): Java vs. C
(Sun UltraSPARC 60)

* Sun JDK 1.2, javac -0; Sun cc -0; Solaris 2.x
SciMark: Java vs. C
(Intel PIII 500MHz, Win98)

FFT | SOR | MC | Sparse | LU
--- | --- | --- | --- | ---
C | Java

* Sun JDK 1.2, javac -0; Microsoft VC++ 5.0, cl -0; Win98
SciMark (large): Java vs. C
(Intel PIII 500MHz, Win98)

* Sun JDK 1.2, javac -0; Microsoft VC++ 5.0, cl -0; Win98
SciMark: Java vs. C
(Intel PIII 500MHz, Linux)

FFT  SOR  MC  Sparse  LU

C   Java

* RH Linux 6.2, gcc -O6, IBM JDK 1.3, javac -O
SciMark results
500 MHz PIII (Mflops)

Mflops

C (Borland 5.5) | C (VC++ 5.0) | Java (Sun 1.2) | Java (MS 1.1.4) | Java (IBM 1.1.8)

*500MHz PIII, Microsoft C/C++ 5.0 (cl -O2x -G6), Sun JDK 1.2, Microsoft JDK 1.1.4, IBM JRE 1.1.8
SciMark FFT results

Intel 500MHz PIII (Mflops)

Mflops

C (Borland 5.5)  C (VC++ 6)  Java (Sun)  Java (MS)  Java (IBM)

*500MHz PIII, Microsoft C/C++ 5.0 (cl -O2x -G6), Sun JDK 1.2, Microsoft JDK 1.1.4, IBM JRE 1.1.8
SciMark SOR results
(Mflops)

*500MHz PIII, Microsoft C/C++ 5.0 (cl -O2x -G6), Sun JDK 1.2, Microsoft JDK 1.1.4, IBM JRE 1.1.8
SciMark Monte Carlo results (Mflops)

*500MHz PIII, Microsoft C/C++ 5.0 (cl -O2x -G6), Sun JDK 1.2, Microsoft JDK 1.1.4, IBM JRE 1.1.8
SciMark Sparse-Matmult results
(Mflops)

*500MHz PIII, Microsoft C/C++ 5.0 (cl -O2x -G6), Sun JDK 1.2, Microsoft JDK 1.1.4, IBM JRE 1.1.8
SciMark LU results
(Mflops)

*C 500MHz PIII, Microsoft C/C++ 5.0 (cl -O2x -G6), Sun JDK 1.2, Microsoft JDK 1.1.4, IBM JRE 1.1.8*
C vs. Java

• Why C is faster than Java
  – direct mapping to hardware
  – more opportunities for aggressive optimization
  – no garbage collection

• Why Java is faster than C (?)
  – different compilers/optimizations
  – performance more a factor of economics than technology
  – C compilers could be further improved
Current JVMs are quite good...

- 1000x1000 matrix multiply over 180Mflops
  - 500 MHz Intel PIII, Win98, JDK 1.2

- Scimark high score: 163.0 Mflops
  - 933 MHz Intel PIII, Linux, IBM JDK 1.3
Another approach...

- Use an aggressive optimizing compiler
- code using Array classes which mimic Fortran storage
  - e.g. $A[i][j]$ becomes $A.get(i,j)$
  - ugly, but can be fixed with operator overloading extensions
- “cheat” a little to exploit hardware (FMAs)
- result: 80+% of Fortran on RS/6000
IBM High Performance Compiler

- Snir, Moreria, et. al
- native compiler (.java -> .exe)
- requires source code
- can’t embed in browser, but…
- produces very fast codes
Java vs. Fortran Performance

*IBM RS/6000 67MHz POWER2 (266 Mflops peak) AIX Fortran, HPJC
Conclusions

• Java numerics can be competitive with C
  – 50% “rule of thumb” for many instances
  – can achieve efficiency of optimized C/Fortran
• best Java performance on commodity platforms
• new compiling and execution environments are further improving these numbers.
Java Resources for Scientific Computing

- Java Numerics Group
  - http://math.nist.gov/javanumerics
- Java Grande Forum
  - http://www.javagrade.org
- SciMark Benchmark
  - http://math.nist.gov/scimark