Towards Easy-to-use PGAS Parallel Programming – The Distributed JVM Approach

Cho-Li Wang (王卓立)
The University of Hong Kong (香港大學)

CSO’10, Huangsan, China
Outline

1. Era of Petaflop Computing
2. PGAS Programming Language
3. Distributed Java Virtual Machine
4. Profile-guided locality management
5. Performance Evaluation
## Era of PetaFlop Computing

### Top500 Supercomputer List (Nov/2009)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>Computer/Year Vendor</th>
<th># of cores</th>
<th>Linpack ($R_{\text{max}}$)</th>
<th>Peak ($R_{\text{peak}}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Oak Ridge National Laboratory (USA)</td>
<td>Jaguar – Cray 2009, Cray Inc.</td>
<td>224162</td>
<td>1759.00</td>
<td>2331.00</td>
</tr>
<tr>
<td>2</td>
<td>DOE/NNSA/LANL, USA</td>
<td>Roadrunner, 2009 IBM</td>
<td>122400</td>
<td>1042.00</td>
<td>1375.78</td>
</tr>
<tr>
<td>3</td>
<td>National Institute for Computational Sciences/USA</td>
<td>Kraken XT5 2009, Cray Inc.</td>
<td>98928</td>
<td>831.70</td>
<td>1028.85</td>
</tr>
<tr>
<td>4</td>
<td>Forschungszentrum Juelich (FZJ) Germany</td>
<td>JUGENE - Blue Gene/P 2009 IBM</td>
<td>294912</td>
<td>825.50</td>
<td>1002.70</td>
</tr>
<tr>
<td>5</td>
<td>National SuperComputer Center in Tianjin/NUDT, China</td>
<td>Tianhe-1 天河一号, 2009 NUDT</td>
<td>71680</td>
<td>563.10</td>
<td>1206.19</td>
</tr>
<tr>
<td>6</td>
<td>NASA/USA</td>
<td>Pleiades - SGI Altix ICE 6200EX, 2009 SGI</td>
<td>56320</td>
<td>544.30</td>
<td>673.26</td>
</tr>
<tr>
<td>7</td>
<td>DOE/NNSA/LLNL (USA)</td>
<td>BlueGene/L 2007 IBM</td>
<td>212992</td>
<td>478.20</td>
<td>596.38</td>
</tr>
<tr>
<td>8</td>
<td>Argonne National Laboratory, USA</td>
<td>Blue Gene/P Solution / 2007 IBM</td>
<td>163840</td>
<td>458.61</td>
<td>557.06</td>
</tr>
<tr>
<td>9</td>
<td>Texas Advanced Computing Center, USA</td>
<td>Ranger - SunBlade x6420, 2008, Sun Microsystems</td>
<td>62976</td>
<td>433.20</td>
<td>579.38</td>
</tr>
<tr>
<td>10</td>
<td>Sandia National Laboratories, USA</td>
<td>Red Sky - Sun Blade x6275, 2009 Sun Microsystems</td>
<td>41616</td>
<td>423.90</td>
<td>487.74</td>
</tr>
</tbody>
</table>

Top 5 machines achieved PetaFlop computing power
China’s Tianhe-1 Petaflop Computers

**Hybrid structure:** 6,144 Intel Xeon E5540 CPUs + 5,120 GPUs (ATI Radeon HD4870)

5th in TOP500
Peak performance: 1.2 PetaFLOPS
LINPACK score: 563.1 TeraFLOPS

#8 at Top500 Green List

512 Operation Nodes In 20 cabinets
2560 Compute Nodes In 80 cabinets

Source: Institute of Computer, NUDT
Petaflop Supercomputers with >1M cores

100 Petaflops system most likely in the year 2016

2010: Dawning6000
2011: IBM Blue Waters

2009: Jaguar (Cray), Kraken XT5, JUGENE, Tianhe-1

06/2008: Roadrunner break the petaflop barrier (1.026 petaflop/s)


SUM from top500.org
A petascale Blue Gene/Q supercomputer: **1.6 million processor cores** divided into 98,304 nodes placed within 96 Racks, record the amount of memory installed, equivalent to 1.6 petabytes.
Dawning 6000 consists of two parts,

- **Dawning Nebulae (星云) GPU cluster:** 5000 blades, each contains two six-core INTEL 6-core X5650 2.66GHz processors and one NVIDIA C2050 Fermi GPU card. QDR Infiniband. **Peak:** 3.5 Petaflops. **Linpack 1.27 Petaflops. (2nd in TOP500, May 30, 2010)**

- **Loongson (龙芯) cluster:** about 5000 blades w/ 8000 to 10,000 8-core Godson-3B processor (under development)

- Located at National Supercomputing Shenzhen Center

- **Total investment:** 800M RMB (8亿元)
New Landscape of Parallel Computer Architecture

- **Multi-core Architectures**
  - Conventional multicore approach (2, 4, 8 cores) → manycore technology (hundreds or even thousands of cores)
  - Employs simpler cores running at modestly lower clock frequencies

- **Hardware accelerators**
  - FPGA (Cray XD1, SGI RASC), GPU (Tianhe-1, Dawning6000, TSUBAME), Cell, ClearSpeed (TSUBAME) and vector processors, LINPACK?

- **Networking**:
  - RDMA: A one-sided put/get message can be handled directly by a network interface with RDMA support
  - TCP Offload Engine (TOE)
  - Most systems use either a 4X 10 Gbit/s (SDR), 20 Gbit/s (DDR) or 40 Gbit/s (QDR) connection.
  - End-to-end MPI latency: 1.07 microseconds
  - 10 Gigabit Ethernet go mainstream (fallen to $500 per port)
<table>
<thead>
<tr>
<th>Micro-architecture</th>
<th>Clock Rate (GHz)</th>
<th>Cores</th>
<th>Threads Per Core</th>
<th>Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Power 7</td>
<td>3.00 - 3.14</td>
<td>4-8</td>
<td>4</td>
<td>32KB+32KB Private L1 256KB Private L2 4MB Shared L3</td>
</tr>
<tr>
<td>Sun/Oracle Niagara2</td>
<td>1.2-1.6</td>
<td>4-8</td>
<td>8</td>
<td>8KB+8KB Private L1 4MB Shared L2</td>
</tr>
<tr>
<td>Intel Westmere</td>
<td>1.86 - 2.66</td>
<td>4-8</td>
<td>2</td>
<td>32KB+32KB Private L1 256KB Private L2 12-24 MB Shared L3</td>
</tr>
<tr>
<td>Intel Harpertown</td>
<td>2.00 - 3.40</td>
<td>4</td>
<td>2</td>
<td>32KB+32KB Private L1 2x6MB L2 Cache</td>
</tr>
<tr>
<td>AMD Magny-Cours</td>
<td>1.7 - 2.3</td>
<td>12 or 16</td>
<td>1</td>
<td>64KB+64KB Private L1 512KB Private L2 2x6 MB Shared L3</td>
</tr>
<tr>
<td>Intel Single-Chip Cloud</td>
<td>1.0</td>
<td>48</td>
<td>1</td>
<td>16KB L1 Cache 256KB Private L2 Cache 16KB Msg Buffer per Tile</td>
</tr>
<tr>
<td>Intel Terascale</td>
<td>~ 4</td>
<td>80</td>
<td>1 ?</td>
<td>3KB Instruction + 2KB Data on each Core</td>
</tr>
<tr>
<td>Tilera Tile-GX</td>
<td>1.5</td>
<td>100</td>
<td>1 ?</td>
<td>32KB+32KB Private L1 256KB L2 Private L2 26MB Distributed L3</td>
</tr>
</tbody>
</table>
Outline

1. Era of Petaflop Computing
2. PGAS Programming Language
3. Distributed Java Virtual Machine
4. Profile-guided Locality Management
5. Performance Evaluation
Parallelism will explode
- Number of cores will double every 12-24 months
- Petaflop (million processor) machines will be common in HPC by 2015

Performance will become a software problem
- Parallelism and locality are key
- Concurrency is the next major revolution in how we write software

A new programming model will emerge for petaflop computing

Do we put enough emphasis on software?

Berkeley's Dr. Kathy Yelick (director of NERSC):

No. Unfortunately, the race for each major performance milestone, has resulted in a de-emphasis on software.

Source: The Software Challenges of Petascale Computing
Most parallel programs are written using:

- **Message passing**
  - Examples: CM5’s CMMD, PVM, IBM’s MPL,
  - Current standard: MPI (MPICH-1, MPICH-2, LAM/MPI..)
  - Usually used for scientific applications with C++/Fortran, or Java (JavaMPI, G-JavaMPI)
  - **Scales easily**: user controlled data layout
  - **Hard to use**: send/receive matching, message packing/unpacking

- **Shared memory**
  - Examples: OpenMP, pthreads, Java
  - Usually for non-scientific applications
  - **Easier to program**: direct reads and writes to shared data
  - **Hard to scale**: (mostly) limited to SMPs, no concept of locality
Optimizing is Hard!

- **Tianhe-1 Experience**: Scaling LINPACK performance from 20% to 70% of each CPU-GPU pair

Source: Dr. Chunyuan Zhang, National University of Defense Technology
Parallel Programming environments since the 90’s
Do you like to design another ONE?

Let me add one more?

Source: John Urbanic, Pittsburgh Supercomputing Center
The Software challenges of Petaflop computing

- New algorithmic approaches to increase the levels of concurrency on the order of $10^8$
- Developing effective methodologies for assessing and exploiting data locality (high cache hit rates) in the deep memory hierarchies
- **Hide latency** by utilizing low-level parallelism (e.g., prefetch queues and multithreading)
- Design algorithms and implementations that permit easy recovery from system failures
- Performance monitoring facilities (accurate timers and operation counters, out-of-cache loads and stores) and dynamic load balancing
- Accuracy and stability of numerical methods: formal methods to certify the correctness of petaflops algorithms and hardware logic designs
- New languages and constructs (alternatives to HPF, OpenMP, MPI,..)
Programmability in HPC

- Relevant research area in the last years
  - Growing interest on easier programming
- HPCS project (DARPA)
  - High-performance High-Productivity Programming
  - New languages that focus on programmability (IBM X10, Cray CHAPEL, Sun Fortress)
- PGAS (Partitioned Global Address Space):
  - Target global address space, multithreading platforms
  - Aim for high levels of scalability
  - Research languages:
    - Co-Array Fortran (CAF)
    - Unified Parallel C (UPC)
    - Titanium (Java)
Features of PGAS Languages

- **Explicitly-parallel programming model** with SPMD parallelism
  - Fixed at program start-up, typically 1 thread per processor
- **Global address space model of memory**
  - Allows programmer to directly represent distributed data structures
  - Can access local and remote data with same mechanisms
- **Address space is logically partitioned**
  - Local vs. remote memory (two-level hierarchy) – handled by users
- **Programmer control over performance critical decisions** (** burden to users **)
  - Data layout and communication
- **Base languages differ**: Co-Array Fortran (CAF), Unified Parallel C (UPC), Titanium (Java)

Source: Yelick’s (UCB) CS267 Lecture
Global Address Space Eases Programming

- The languages share the global address space abstraction
  - Shared memory is partitioned by processors
  - Remote memory may stay remote: **no automatic caching implied**
  - **One-sided communication** through reads/writes of shared variables
  - Both individual and bulk memory copies
- Differ on details
  - Some models have a separate private memory area
  - Distributed array generality and how they are constructed

Source: Yelick’s (UCB) CS267 Lecture
Languages (or language technologies) that solve real problems can succeed [Todd A. Proebsting, Microsoft Research, 2002]:
- Even if slow
- Even with simple types
- Even without academic significance (no papers?)
- Even without rocket science
- If useful

Programmer Productivity:
- Write programs correctly (50% of crashes caused by 1% of bugs)
- Write programs quickly
- Write programs easily

Why?
- Decreases support cost
- Decreases development cost
- Decreases time to market/solution
- Increases satisfaction
But ... “New Language Fear”

- **Long-Live Language Needed:**
  - Large-scale codes: *portability* is top priority.
  - Large-scale codes lifetimes: 10 to 30 years.
  - High-performance computers: 3-5 years between generations.
  - They can't risk spending 5-10 years writing their code in a new language only to find that the new language didn't gain general acceptance and support.

- **Fear of learning new language:**
  - Some people say that “if there's a lot of pain involved, they won't switch to a new programming language.”

- **How can you motivate people to migrate to a more efficient new language? Or do they have to?**
Why Java for HPC?

- **Good programmability for potential HPC**
  - Expressive grammar: simplified C++
  - Concurrent language: **multithreading** support at language level (Portable way of parallel programming)
  - Platform independence: **bytecode** (write once, run everywhere!)
  - Runtime: **GC, safety checking**, etc.
  - Libraries: a huge increasing list
  - Deliver **65%-90%** of performance of the best Fortran programs; compete with C++:
    - Java-based next-gen languages: X10 (IBM), Titanium, Fortress (Sun)

- **Easy to learn.**
  - Write Java programs quickly
  - Write Java programs easily
  - Less bugs (?)
Our Approach

Distributed Java Virtual Machine

Single system image (SSI) illusion to threads of a Java program
class worker extends Thread {
    private long n;
    public worker(long N) { n = N; }
    public void run() {
        long sum = 0;
        for (long i = 0; i < n; i++) sum += i;
        System.out.println("Sum = " + sum);
    }
}

public class test {
    static final int N = 100;

    public static void main(String args[]) {
        worker[] w = new worker[N];
        Random r = new Random();
        for (int i = 0; i < N; i++)
            w[i] = new worker(r.nextLong());
        for (int i = 0; i < N; i++) w[i].start();
        try{
            for (int i = 0; i < N; i++) w[i].join();
        } catch (Exception e){}
    }
}

Multithreaded Java application

DJVM hides the physical boundaries between machines
Support thread migration
History and Roadmap of JESSICA

- **JESSICA V1.0** *(1996-1999)*
  - Execution mode: Interpreter Mode
  - JVM kernel modification (Kaffe JVM)
  - Global Heap: built on top of TreadMarks (Lazy Release Consistency + homeless)

- **JESSICA V2.0** *(2000-2006)*
  - Execution mode: JIT-Compiler Mode (full speed)
  - JVM kernel modification (Kaffe JVM)
  - Lazy Release Consistency + migrating-home protocol

- **JESSICA V3.0** *(2008~2010?)*
  - Built above JVM (JVMTI)
  - Support Large Object Space
  - For any JVM. Run @ full speed of the underlying JVM.

- **JESSICA v.4** *(2009~)*
  - Software transactional memory model
  - Multicore/GPU cluster
JESSICA Distributed Java VM

- A cluster-wide JVM with
  - Dynamic thread mobility in JIT mode
  - Global Object Space (GOS)
Problem 1: Memory Consistency

When a write becomes visible to another thread? How?
Solution: Global Object Space (GOS)

- Per-object granularity, no false sharing
- **Home-based Lazy Release Consistency** (HLRC)
  - Home-based variant of LRC: always fetch latest object/page from its home
  - No traffic if object unchanged
  - **Object home migration**: better locality
- Connectivity-based object prefetching: more accurate
Problem 2: Thread migration under JITC Mode

**JAVA SOURCE CODE**

```java
public class Test {
    boolean multiple = true;
    int a = 1000;
    int b = 5;

    int compute() {
        if (multiple)
            return a * b;
        else
            return a / b;
    }

    static void main(String[] args) {
        System.out.println(compute());
    }
}
```

**NATIVE CODE**

```assembly
.LCOMPUTE:
.L0:
    MOVL EBP[0] EAX
    MOVL [EAX] EBX
    XORL EBX 0x1
    IFEQ L1

.L2:
    MOVL EBP[0] EAX
    MOVL [EAX, 1] EBX
    MOVL [EAX, 2] ECX
    MUL EBX ECX
    MOVL EBX EAX
    RET

.L1:
    MOVL EBP[0] EAX
    MOVL [EAX, 1] EBX
    MOVL [EAX, 2] ECX
    DIV EBX ECX
    MOVL EBX EAX
    RET
```

**JIT Compiler mode execution makes things complex**

Native code has no clear bytecode boundary
How to deal with machine registers?
How to organize the stack frames?
How to make extracted thread states recognizable by the remote JVM?
Thread Migration in JIT Compiler Mode

1. Alert
   - Thread Scheduler
   - JVM
   - Source node
   - Load Monitor

2. Stack analysis
   - Stack capturing
   - Method Area

3. Frame parsing
   - Restore execution
   - Migration Manager
   - Destination node
   - Method Area
   - (4a) Object Access
   - (4b) Load method from NFS

4. On-stack scanning
   - Java frame
   - C frame
   - Native thread stack
Thread Migration in JIT Compiler Mode

- **Dynamic Native Code Instrumentation**

  - **Migration points selection**
    - Delayed to the head of loop basic block or method

  - **Register context handler**
    - Spill dirty registers at migration point without invalidation so that native codes can continue the use of registers
    - Use register recovering stub at restoring phase

  - **Variable type deduction**
    - Spill type in stacks using compression

  - **Java frames linking**
    - Discover consecutive Java frames
Problem 3: Improve Locality

- Remote memory access is the scalability killer!
- Remote >> local latency (assume in 50-60ns)
  - Infiniband cluster (1-2μs): 20 x slower!
  - Ethernet cluster (100μs): 2,000 x slower!!
  - Grid/Internet (av. 500ms): 10,000,000 x slower!!

- "To speed up" ≈ "Reduce as much remote access as possible"
- The key is to improve locality
Solution: Profile-Guided PGAS (PG\textsuperscript{2}AS)

- **Profile-Guided PGAS (PG\textsuperscript{2}AS)**
  - A built-in runtime profiler instead of humans for digging out the locality hints

- **Profile-guided adaptive locality management**
  - Thread migration
  - Object home migration
  - Object prefetching

- **Challenges:**
  - How does the runtime know which threads to migrate can make the most locality benefit?
  - Difficult to decide if no global inter-thread sharing information

- **Solution: Track sharing % threads**
  - T1 accesses O1, O3, O5, ...
  - T2 accesses O1, O2, O3, ...
  - Sharing % T1 & T2: O1, O3
1. Era of Petaflop Computing
2. PGAS Programming Language
3. Distributed Java Virtual Machine
4. Profile-guided locality management
5. Performance Evaluation
**PG-JESSICA: Profile-Guided Version**

- **Access profiler**: track object access over heap to deduce inter-thread sharing -> *thread-thread relation*
- **Stack profiler**: track the set of frequent objects accessed by each thread -> *thread migration cost*
- **Correlation analyzer**: profile-guided decisions on dynamic thread migration -> global locality improvement
**Thread Correlation Map (TCM)**

- Thitikamol and Keleher; D-CVM (1999)
  - Proposed “Active Correlation Tracking” (Page)
  - **Thread Correlation Map (TCM):** a 2D histogram of shared data volume between each pair of threads.
    - Grayscale(x,y) = sharing amount of thread x and y
    - TCM(1,1) = TCM(2,2) = TCM(3,3) = ... = 0

**Challenge:** Given $M$ objects shared by $N$ threads, TCM building take $O(MN^2)$ time. $M$ can grow into a scalability bottleneck in the system.

Water-Spatial (32 threads placed on 8 nodes)
“Sticky Set”

- **Sticky Set (SS)**: a subset of working set of a thread, includes only those frequently used objects.
- “Sticky”: if the thread is migrated, objects in SS are more likely to be fetched again.
- SS should be detected and moved along with the thread to save most object misses after migration.
Summary of Our Solution

- **What we want to do:**
  1. Model thread sharing (inter-thread correlation)
  2. Model indirect thread migration cost

- **Profiling results:**
  1. Thread Correlation Map (TCM)
  2. Per-thread Sticky Set (SS)

- **Use both to design new migration policy**
  1. Correlation-driven
  2. Cost-aware

- **How we profile them efficiently?**
  1. Adaptive object sampling → TCM
  2. Adaptive stack sampling → SS

Details: King Tin Lam, Yang Luo, Cho-Li Wang, "Adaptive Sampling-Based Profiling Techniques for Optimizing the Distributed JVM Runtime," 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS2010), April 19-23, ATLANTA, USA
Adaptive Object Sampling (AOS)

- Each object has a "sequence number", unique among objects within the same class.
- Sample the object if sequence # is divisible by the current "sampling gap" (selected and changed at runtime to strike a balance of cost and accuracy)
- Sampling rate:
  - \(1X = \text{sample 1 object per page of heap}\)
  - \(1024X\) means "full sampling"
  - For a class of size \(s\), sampling at rate \(nX\), sampling gap = \(S_p / (s \times n)\), where \(S_p\) is the page size (usually 4KB).

<table>
<thead>
<tr>
<th>gap</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
JVM is a “stack machine”

- Stack variables can be hint of constantly accessed objects

- Stack invariants: Those references constantly stay in the stack across snapshots taken. Good hints of SS.

- Usually stack invariants are the entry points of SS and important data structures like Hashmap, TreeMap, Linked List
Adaptive Stack Sampling: Adjustable timer controlling which period of time to do stack sampling. Stack frame added with “visited” flag. If not touched across two sampling rounds, no need to sample it.
Outline

1. Era of Petaflop Computing
2. PGAS Programming Language
3. Distributed Java Virtual Machine
4. Profile-guided locality management
5. Performance Evaluation
Testing Environment: HKU Gideon-II Cluster

- **240 SMP blade servers (19.43 TFlop/s)**
  - Expected to grow to 25+ TFlop/s upon Phase 2’s completion in late 2010.

- **Node configuration:**
  - Dell PowerEdge R610/M610
    - 2 x Intel Nehalem-based Quad-core Xeon 2.53GHz
    - 32 GB 1066MHz DDR3 RAM and SAS disks

- **Networking:**
  - 4X DDR Infiniband (20 Gbit/s): **80 nodes** (not used)
  - Gigabit Ethernet (1 Gbit/s): **160 nodes**
  - OS: RedHat Enterprise Linux, Scientific Linux, Fedora Linux.

- **Production run in September, 2009**

---

**Computer Science (Systems Research Group)**

- **SRG-GbE cluster**
  - 64 blades (32 GB 1066MHz DDR3 RAM)

- **SRG Gatekeepers**
  - (PowerEdge R610 x 4)

- **SRG NFS server**
  - Blade Network G8142 24-port 10Gbe Switch

- **Backup server**
  - Brocade Fastiron SuperX 108-port Gigabit Switch

- **SRG IB cluster**
  - (48 1U IB nodes + Qlogic Silverstorm 9040 48-port DDR switch)

- A system-wide management sub-system
Speedup of JAVA applications on JESSICA2

Speedup of different applications

- Linear
- CPI
- Raytracing
- TSP
- Matrix Multiplication
- Npose

Speedup value vs. # of nodes
Ray Tracing on JESSICA2 (64 PCs)

64 nodes: 108 seconds
1 node: 4402 seconds (1.2 hour)

Speedup = 40.75
## Time and space Overhead Analysis

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Time (seconds)</th>
<th>Space(native code/bytecode)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No migration</td>
<td>Migration</td>
</tr>
<tr>
<td>compress</td>
<td>11.31</td>
<td>11.39(+0.71%)</td>
</tr>
<tr>
<td>jess</td>
<td>30.48</td>
<td>30.96(+1.57%)</td>
</tr>
<tr>
<td>raytrace</td>
<td>24.47</td>
<td>24.68(+0.86%)</td>
</tr>
<tr>
<td>db</td>
<td>35.49</td>
<td>36.69(+3.38%)</td>
</tr>
<tr>
<td>javac</td>
<td>38.66</td>
<td>40.96(+5.95%)</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>28.07</td>
<td>29.28(+4.31%)</td>
</tr>
<tr>
<td>mtrt</td>
<td>24.91</td>
<td>25.05(+0.56%)</td>
</tr>
<tr>
<td>jack</td>
<td>37.78</td>
<td>37.90(+0.32%)</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td>( +2.21%)</td>
</tr>
</tbody>
</table>

(Gideon-I)
Thread migration for irregular applications (1) : TSP

8 nodes, 16 threads, TSP 13 cities, (object sharing: shortest path)

<table>
<thead>
<tr>
<th></th>
<th>Initial placement</th>
<th>Thread migration (5 times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec)</td>
<td><strong>1203.10</strong></td>
<td><strong>793.317</strong> (−33.6%)</td>
</tr>
<tr>
<td>Stdev</td>
<td>438,444.1</td>
<td>152,463.1</td>
</tr>
</tbody>
</table>

(Gideon-I)
# Stack Profiling Overhead

- **Timer-based control of stack sampling phases** saves over half of overheads
- **Lazy extraction** saves up to 1/3 overheads

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Set Size</th>
<th>Baseline Exe Time</th>
<th>Immediate Extraction</th>
<th>Lazy Extraction</th>
<th>Nonstop</th>
<th>Timer-based (100ms)</th>
<th>+ Sticky-set Resolution Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>1K×1K</td>
<td>6201</td>
<td>6216 (0.24%)</td>
<td>6207 (0.10%)</td>
<td>4ms</td>
<td>6714 (8.28%)</td>
<td>6519 (5.13%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6211 (0.17%)</td>
<td>6206 (0.08%)</td>
<td>16ms</td>
<td>6707 (8.17%)</td>
<td>6480 (4.50%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6639 (1.85%)</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>4K</td>
<td>93857</td>
<td>94947 (1.16%)</td>
<td>94657 (0.85%)</td>
<td>4ms</td>
<td>98968 (5.45%)</td>
<td>93649 (-0.22%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>94697 (0.89%)</td>
<td>95209 (1.44%)</td>
<td>16ms</td>
<td>102190 (8.88%)</td>
<td>102334 (9.03%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>97585 (4.20%)</td>
</tr>
<tr>
<td>Waterspatial</td>
<td>512</td>
<td>59105</td>
<td>59232 (0.21%)</td>
<td>59161 (0.09%)</td>
<td>4ms</td>
<td>59834 (1.23%)</td>
<td>59501 (0.67%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>59209 (0.17%)</td>
<td>59124 (0.03%)</td>
<td>16ms</td>
<td>61985 (4.87%)</td>
<td>60313 (2.04%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60002 (0.84%)</td>
</tr>
</tbody>
</table>
Accuracy of AOS (Cont')

(Euclidean distance)

\[ E_{EUC} = \sqrt{\frac{\sum_{i=1}^{N} \sum_{j=1}^{N} (a_{ij} - b_{ij})^2}{\sum_{i=1}^{N} \sum_{j=1}^{N} (b_{ij})^2}} \]

(Absolute distance)

\[ E_{ABS} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} |a_{ij} - b_{ij}|}{\sum_{i=1}^{N} \sum_{j=1}^{N} |b_{ij}|} \]
Profile-Guided Thread Migration

- We assess this using a CRM application “Customer Analytics” with dynamic change in sharing patterns.

![Diagram showing thread migration and data segments]

Epoch 1...
We assess this using a CRM application “Customer Analytics” with dynamic change in sharing patterns.
Effect of Profile-Guided Thread Migration

- Without thread migration, locality is not preserved (out of red boxes denoting node boundaries) as time goes by.
Effect of Profile-Guided Thread Migration

- With correlation-driven thread migration

T2, T3 migrated to node 2

T2, T3 migrated to node 3

epoch1

epoch2

epoch3

epoch4

epoch5

epoch6

epoch7

epoch8
Performance Gain

![Chart showing performance gain with different sampling rates. The chart includes labels for (Full sampling) 1024X, 512X, 256X, 128X, 64X, 32X, 16X, 8X, 4X, and correlation migration enabled. The chart also includes correlation tracking + thread migration, adaptive rate (start at 4X), no correlation and migration, stretch, round-robin, and random. The chart indicates a 21.7% gain.]
Conclusion

- Distributed Java Virtual Machine can provide a high-performance platform for running multithreaded Java applications on clusters.
- Java thread migration helps to improve the performance, flexibility, and scalability of DJVM.
- A couple of advanced profiling strategies for optimizing locality:
  - Adaptive object sampling
  - Online stack sampling
- Towards PGAS Parallel Programming – why not JESSICA (“Easy-to-use”)

54
JESSICA Launched to CNGrid HKU Portal

China National Grid

Tsinghua University 清华大学
Institute of Applied Physics and Computational Mathematics 北京应用物理与计算数学研究所
Xian JiaoTong University 西安交通大学
National University of Defense Technology 国防科学技术大学
ShenZhen Institute of Advanced Technology, Chinese Academy of Sciences 中国科学院深圳先进技术研究院

Computer Network Information Center, Chinese Academy of Sciences 中国科学院计算机网络信息中心
Shandong University 山东大学

HKU Grid Point On-Line Bookstore

The image contains a map of China with nodes representing various universities and institutions, including Tsinghua University, Institute of Applied Physics and Computational Mathematics, Xian JiaoTong University, National University of Defense Technology, ShenZhen Institute of Advanced Technology, Chinese Academy of Sciences, and Shandong University. The image also shows a webpage with links to a bookstore and software products.
Thanks!

For more information:

JESSICA2 Project

C.L. Wang’s webpage:
http://www.cs.hku.hk/~clwang/