Effectively Mapping Linguistic Abstractions for Message-passing Concurrency to Threads on the Java Virtual Machine

Ganesha Upadhyaya
ganeshau@iastate.edu
Iowa State University

Hridesh Rajan
hridesh@iastate.edu
Iowa State University

Supported in part by the NSF grants CCF- 08-46059, CCF-11-17937, and CCF-14-23370.
Effectively Mapping Linguistic Abstractions for Message-passing Concurrency to Threads on the Java Virtual Machine

Ganesha Upadhyaya  
ganeshau@iastate.edu  
Iowa State University

Hridesh Rajan  
hridesh@iastate.edu  
Iowa State University

1) Local computation and communication behavior of a concurrent entity can help predict the performance.
2) A modular and static technique can solve the problem.

Supported in part by the NSF grants CCF-08-46059, CCF-11-17937, and CCF-14-23370.
MPC: Message-passing Concurrency, a1, a2, a3, a4 are MPC abstractions
MPC: Message-passing Concurrency, a1, a2, a3, a4 are MPC abstractions
Problem

MPC System

a1 → a3 → a2 → a4 → a5

Mapping

OS Scheduler

JVM threads

Architecture

core0 | core1

core2 | core3

MPC: Message-passing Concurrency, a1, a2, a3, a4 are MPC abstractions

In this work: Mapping Abstractions to JVM threads
PROBLEM: Mapping Abstractions to JVM threads

GOAL: Concurrency
PROBLEM: Mapping Abstractions to JVM threads

GOAL: Concurrency + Performance (Fast and Efficient)
Motivation

- State of the art: Abstractions to Threads mapping is performed by programmers.
Motivation

• Abstractions to Threads mapping is performed by programmers

• MPC frameworks (Akka, Scala Actors, SALSA) provide *Schedulers and Dispatchers* to programmers for mapping Abstractions to Threads
Availability of wide-variety of Schedulers and Dispatchers suggests that.

- Programmers can chose the ones that works best for their applications
- And, perform the mapping carefully
Motivation

- Abstractions to Threads mapping is performed by programmers.
- MPC frameworks (Akka, Scala Actors, SALSA) provide *Schedulers and Dispatchers* to programmers for mapping Abstractions to Threads.

**Programmers find it hard to manually perform the mapping.**
- Start with an initial mapping and incrementally improve the mapping.
- This process can be tedious and time consuming.
Motivation

• Abstractions to Threads mapping is performed by programmers
• MPC frameworks (Akka, Scala Actors, SALSA) provide Schedulers and Dispatchers to programmers for mapping Abstractions to Threads
• SO discussions about configuring and fine tuning the mapping suggests that,
  – Randomly tweaking the mapping without finding the root cause of performance problem doesn’t help and
  – Without knowing the nature of the task performed by the abstractions, the mapping task becomes hard.
When manual tuning is hard,
• Programmers use default mappings (default schedulers/dispatchers)
When manual tuning is hard,

- Programmers use default mappings (default schedulers/dispatchers)
- Problem: **a single default mapping may not work across programs**
When manual tuning is hard,

- Programmers use default mappings (default schedulers/dispatchers)
- Problem: a single default mapping may not work across programs

**Motivation**

**LogisticMap**
(computes logistic map using a recurrence relation)

**ScratchPad**
(counts lines for all files in a directory)
When manual tuning is hard,

- Programmers use default mappings (default schedulers/dispatchers)
- Problem: **a single default mapping may not work across programs**

**LogisticMap**
(computes logistic map using a recurrence relation)
When manual tuning is hard,

- Programmers use default mappings (default schedulers/dispatchers)
- Problem: **a single default mapping may not work across programs**
When manual tuning is hard,

- Programmers use default mappings (default schedulers/dispatchers)
- Problem: **a single default mapping may not work across programs**

![ScratchPad](counts lines for all files in a directory)

- Core setting:
  - thread
  - round-robin
  - random
  - work-stealing

![Execution time (s)](2 4 8 12 Core setting)
When manual tuning is hard,
• Programmers use default mappings (default schedulers/dispatchers)
• Problem: **a single default mapping may not work across programs**
When manual tuning is hard,
- Programmers use default mappings (default schedulers/dispatchers)
- Problem: a single default mapping may not work across programs

**LogisticMap**  
(computes logistic map using a recurrence relation)

**ScratchPad**  
(counts lines for all files in a directory)
When manual tuning is hard and default mappings may not produce the desired performance,

- Brute force technique that tries all possible combinations of Abstractions to Threads mapping could be used
When manual tuning is hard and default mappings may not produce the desired performance,

- brute force technique that tries all possible combinations of Abstractions to Threads mapping could be used
- Problem: Combinatorial explosion
When manual tuning is hard and default mappings may not produce the desired performance,

- brute force technique that tries all possible combinations of Abstractions to Threads mapping could be used
- Problem: Combinatorial explosion
- For example: an MPC program with 8 kinds of abstractions, trying all possible combinations of 4 kinds of schedulers/dispatchers requires exploring 65536 ($4^8$) different combinations (some may even violate the concurrency correctness property)
When manual tuning is hard and default mappings may not produce the desired performance,

- brute force technique that tries all possible combinations of Abstractions to Threads mapping could be used
- Problem: Combinatorial explosion
- For example: an MPC program with 8 kinds of abstractions, trying all possible combinations of 4 kinds of schedulers/dispatchers requires exploring $65536 (4^8)$ different combinations (some may even violate the concurrency correctness property)
- Also, a small change to the program may require re-doing the mapping.
When manual tuning is hard and default mappings may not produce the desired performance,

- brute force technique that tries all possible combinations of Abstractions to Threads mapping could be used
- Problem: Combinatorial explosion
  - For example: an MPC program with 8 kinds of abstractions, trying all possible combinations of 4 kinds of schedulers/dispatchers requires exploring $65536$ ($4^8$) different combinations
- Also, a small change to the program may require redoing the mapping.

A mapping solution that yields significant performance improvement over default mappings is desirable
1) Local computation and communication behavior of concurrent entity is predictive for determining globally beneficial mapping
Key Ideas

1) Local computation and communication behavior of concurrent entity is predictive for determining globally beneficial mapping

- Computation and Communication behaviors,
  - externally blocking behavior
  - local state
  - computational workload
  - message send/receive pattern
  - inherent parallelism
1) Local computation and communication behavior of concurrent entity is predictive for determining globally beneficial mapping

- Computation and Communication behaviors,
  - externally blocking behavior
  - local state
  - computational workload
  - message send/receive pattern
  - inherent parallelism

2) Determining these behavior at a coarse/abstract level is sufficient to solve the mapping problem
• Represent Computation and Communication behavior of MPC abstractions (language-agnostic manner),
Solution Outline

• Represent Computation and Communication behavior of MPC abstractions (language-agnostic manner),

• Perform local program analyses statically to determine behaviors (proposed solution is both modular and static)
Solution Outline

- Represent Computation and Communication behavior of MPC abstractions (language-agnostic manner)

- Perform local program analyses statically to determine behaviors (proposed solution is both modular and static)

- A Mapping function that takes represented behaviors as input and produces an execution policy for each abstraction

Execution policy: describes how messages of MPC abstraction are processed (in detail later)
Panini Capsules

General MPC framework → Panini Capsules
MPC Abstraction → Capsule
Message Handlers → Procedures
Panini Capsules

General MPC framework → Panini Capsules

MPC Abstraction → Capsule

Message Handlers → Procedures

Capsule

```c
capsule Ship {
    short state = 0;
    int x = 5;
    void die() { state = 2; }
    void fire() { state = 1; }
    void moveLeft() { if (x>0) x--; }
    void moveRight() { if (x<10) x++; }
}
```
Solution Outline

Static Analyses

- State Analysis
- May-Block Analysis
- Call-claim Analysis
- Communication Summary Analysis
- Computational Workload Analysis

Capsule
- State
- p0
- p1
- ...
- pn

for each procedure pi

Procedure Behavior Composition

cVector

Input Program → cVector Analysis → Mapping Function → Execution Policy
Solution Outline

Input Program → cVector Analysis → Mapping Function → Execution Policy

Static Analyses:
- State Analysis
- May-Block Analysis
- Call-claim Analysis
- Communication Summary Analysis
- Computational Workload Analysis

Capsule:
- For each procedure pi:
  - State
  - p0
  - p1
  - pn

Procedure Behavior Composition → cVector
Solution Outline

- Input Program
- cVector Analysis
- Mapping Function
- Execution Policy

- Capsule
  - State
  - p0
  - p1
  - ... pn

- May-Block Analysis
- Call-claim Analysis
- Communication Summary Analysis
- Computational Workload Analysis

- Procedure Behavior Composition

- EP
Representing Behaviors

Characteristics Vector (cVector) $<\beta, \sigma, \pi, \rho, \rho, \omega>$

- **Blocking behavior** ($\beta$)
  - represents externally blocking behavior due to I/O, socket or db primitives
  - $\text{dom}(\beta): \{true, false\}$

- **Local state** ($\sigma$)
  - local state variables
  - $\text{dom}(\sigma): \{nil, primitive, large\}$

- **Inherent parallelism** ($\pi$)
  - inherent parallelism exposed by capsule when it communicates with other capsules
  - $\text{dom}(\pi): \{sync, async, future\}$

- **Computational workload** ($\omega$)
  - represents computations performed by capsule
  - $\text{dom}(\omega): \{math, io\}$

- **Communication Pattern** ($\rho, \rho$)
Representing Behaviors

Characteristics Vector (cVector) $\langle \beta, \sigma, \pi, \rho, \rho, \omega \rangle$

- **Blocking behavior ($\beta$)**
  - represents externally blocking behavior due to I/O, socket or db primitives
  - $\text{dom}(\beta): \{\text{true, false}\}$

- **Local state ($\sigma$)**
  - local state variables
  - $\text{dom}(\sigma): \{\text{nil, primitive, large}\}$

- **Inherent parallelism ($\pi$)**
  - inherent parallelism exposed by capsule when it communicates with other capsules
  - $\text{dom}(\pi): \{\text{sync, async, future}\}$

- **Computational workload ($\omega$)**
  - represents computations performed by capsule
  - $\text{dom}(\omega): \{\text{math, io}\}$

- **Communication Pattern ($\rho, \rho$)**

All behaviors $\langle \beta, \pi, \rho, \rho, \omega \rangle$ except $\sigma$ is defined for capsule procedures and combined to form behaviors for capsule using behavior composition (described later)
Representing Behaviors

Characteristics Vector (cVector) \( <\beta, \sigma, \pi, \rho, \rho, \omega> \)

- **Blocking behavior** (\( \beta \))
  - represents externally blocking behavior due to I/O, socket or db primitives
  - \( \text{dom}(\beta) : \{true, false\} \)

- **Local state** (\( \sigma \))
  - local state variables
  - \( \text{dom}(\sigma) : \{nil, primitive, large\} \)

- **Inherent parallelism** (\( \pi \))
  - inherent parallelism exposed by capsule when it communicates with other capsules
  - \( \text{dom}(\pi) : \{sync, async, future\} \)

- **Communication Pattern** (\( \rho \))

- **Computational workload** (\( \omega \))
  - represents computations performed by capsule
  - \( \text{dom}(\omega) : \{math, io\} \)

BufferedReader \( br = \) new BufferedReader(new InputStreamReader(System.in));
try {
    userName = br.readLine(); // blocking
} catch (IOException ioe) {
}

**May Block Analysis**
- **Input**: Manually created dictionary of blocking library calls
- **Analysis**: Flow analysis with message receives as sources and blocking library calls as sinks
Representing Behaviors

**Characteristics Vector (cVector)** \(<\beta, \sigma, \pi, \rho, \rho, \omega>\)

- **Blocking behavior (beta)**
  - represents externally blocking behavior due to I/O, socket or db primitives
  - \(\text{dom}(\beta): \{\text{true, false}\}\)

- **Local state (\(\sigma\))**
  - local state variables
  - \(\text{dom}(\sigma): \{\text{nil, fixed, variable}\}\)

- **Inherent parallelism (\(\pi\))**
  - inherent parallelism exposed by capsule when it communicates with other capsules
  - \(\text{dom}(\pi): \{\text{sync, async, future}\}\)

- **Communication Pattern (\(\rho\))**

- **Computational workload (\(\omega\))**
  - represents computations performed by capsule
  - \(\text{dom}(\omega): \{\text{math, io}\}\)

**State Analysis**
- **Input**: State variables
- **Analysis**: Checks the type of state variables that composed capsule state for primitive or collection types.
Characteristics Vector (cVector) \(<\beta, \sigma, \pi, \rho, \rho, \omega>\)

- **Blocking behavior (beta)**
  - represents externally blocking behavior due to I/O, socket or db primitives
  - \(\text{dom}(\beta): \{true, false\}\)

- **Local state (sigma)**
  - local state variables
  - \(\text{dom}(\sigma): \{nil, primitive, large\}\)

- **Inherent parallelism (\(\pi\))**
  - kind of parallelism exposed by capsule while communicating
  - \(\text{dom}(\pi): \{sync, async, future\}\)

```java
capsule Receiver (Sender sender) {
    void receive() {
        // sync
        int i = sender.get();
        // print i;
    }
}
capsule Receiver (Sender sender) {
    void receive() {
        void receive() {
            sender.done(); // async
        }
    }
}
capsule Receiver (Sender sender) {
    void receive() {
        // future
        int i = sender.get();
        // some computation
        // print i;
    }
}
```
Characteristics Vector (cVector) \(<\beta, \sigma, \pi, \rho, \rho, \omega>\)

- **Blocking behavior (beta)**
  - represents externally blocking behavior due to I/O, socket or db primitives
  - \(\text{dom}(\beta)\): \{true, false\}

- **Local state (sigma)**
  - local state variables
  - \(\text{dom}(\sigma)\): \{nil, primitive, large\}

- **Inherent parallelism (\(\pi\))**
  - \(\text{dom}(\pi)\): \{sync, async, future\}

- **Computational workload**
  - represents computations performed by capsule
  - \(\text{dom}(\omega)\): \{math, io\}

- **Communication Pattern**

---

Inherent Parallelism Analysis

```plaintext
Input: List of (call,claim) pairs
Output: \(\pi_p\)
initialize \(\pi_p :=\) async;
foreach element (call,claim) in pairs do 
  if claim == null then
    \(\pi_p \oplus\) async;
  else
    if call.next == claim then
      \(\pi_p \oplus\) sync;
    else
      \(\pi_p \oplus\) future;
  end
end
```

Algorithm 1: Analyzing call-claim pairs
Representing Behaviors

Characteristics Vector (cVector) \( <\beta, \sigma, \pi, \rho, \rho, \omega> \)

- **Blocking behavior (beta)**
  - represents externally blocking behavior due to I/O, socket or db primitives
  - \( \text{dom}(\beta) : \{true, false\} \)

- **Local state (sigma)**
  - local state variables
  - \( \text{dom}(\sigma) : \{nil, primitive, large\} \)

- **Inherent parallelism (pi)**
  - \( \text{dom}(\pi) : \{\text{sync}, \text{async}, \text{future}\} \)

- **Computational workload (\( \omega \))**
  - represents computations performed by capsule
  - \( \text{dom} \omega : \{\text{math, io}\} \)
  - \( \text{math} : \text{computation-to-wait} > 1 \)
  - \( \text{io} : \text{computation-to-wait} \leq 1 \)

Computational Workload Analysis

- Computation summary: recursive calls, high cost library calls, unbounded loops, complete read/write to state that is \textit{variable} size.
• Communication pattern
  – Message send pattern
    \( \text{dom} (\, \rho \,): \{\text{leaf, router, scatter}\} \)
  – Message receive pattern
    • \( \text{dom} (\, \rho \,): \{\text{gather, request-reply}\} \)

\[
\begin{align*}
\text{leaf} &: \text{no outgoing communication} \\
\text{router} &: \text{one-to-one communication} \\
\text{scatter} &: \text{batch communication} \\
\text{gather} &: \text{recv-to-send} > 1 \\
\text{request-reply} &: \text{recv-to-send} \leq 1
\end{align*}
\]
• Communication pattern
  – Message send pattern
    \( \text{dom} \ (\rho) : \{\text{leaf, router, scatter}\} \)
  – Message receive pattern
    • \( \text{dom} \ (\rho) : \{\text{gather, request-reply}\} \)

- Builds Communication Summary which abstracts away expressions except message send/receive, state read/writes.
- Analysis is a function that takes communication summary of a procedure and produces message send/receive pattern tuple as output.
Procedure Behavior Composition

- Input Program
- cVector Analysis
- Mapping Function
- Execution Policy

- Capsule
  - State
    - p0
    - p1
    - ... pn

- for each procedure pi

- State Analysis
- May-Block Analysis
- Call-claim Analysis
- Communication Summary Analysis
- Computational Workload Analysis

- Procedure Behavior Composition
- Mapping Function
- EP

- cVector
• Capsule may have multiple procedures

• Behavior of the capsule is determined by combining the behaviors of its procedures

• For instance, a capsule has blocking behavior if any of its procedures are blocking

• Key idea: Capsule behavior is predominantly defined by the procedure that executes often
**Execution Policies**

- **THREAD**, capsule is assigned a dedicated thread,

- **TASK**, capsule is assigned to a task-pool and the shared thread of the task-pool will process the messages,

- **SEQ/MONITOR**, calling capsule’s thread itself will execute the behavior at callee capsule.

**Execution Policies**

- **Th**: Thread  
  - **A**: Th, **B**: Th

- **Ta**: Task  
  - **A**: Ta, **B**: Ta

- **S**: Sequential  
  - **A**: Th, **B**: S

- **M**: Monitor  
  - **A**: Th, **B**: M
• Encodes several intuitions about MPC abstractions

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Execution Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking</td>
<td>Th</td>
</tr>
<tr>
<td>Heavy</td>
<td>Th</td>
</tr>
<tr>
<td>HighCPU</td>
<td>Ta</td>
</tr>
<tr>
<td>LowCPU</td>
<td>M</td>
</tr>
<tr>
<td>Hub</td>
<td>Ta</td>
</tr>
<tr>
<td>Affinity</td>
<td>M/S</td>
</tr>
<tr>
<td>Master</td>
<td>Ta</td>
</tr>
<tr>
<td>Worker</td>
<td>Ta</td>
</tr>
</tbody>
</table>

Th - Thread, Ta - Task, M - Monitor and S - Sequential
Mapping Heuristics

• Encodes several intuitions about MPC abstractions

• Examples
  – Blocking Heuristics
    • capsules with externally blocking behaviors
    • should be assigned Th execution policy
    • rationale: other policies may lead to blocking of the executing thread, starvation and system deadlocks.

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Execution Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking</td>
<td>Th</td>
</tr>
<tr>
<td>Heavy</td>
<td>Th</td>
</tr>
<tr>
<td>HighCPU</td>
<td>Ta</td>
</tr>
<tr>
<td>LowCPU</td>
<td>M</td>
</tr>
<tr>
<td>Hub</td>
<td>Ta</td>
</tr>
<tr>
<td>Affinity</td>
<td>M/S</td>
</tr>
<tr>
<td>Master</td>
<td>Ta</td>
</tr>
<tr>
<td>Worker</td>
<td>Ta</td>
</tr>
</tbody>
</table>

Th - Thread, Ta - Task, M - Monitor and S - Sequential
Mapping Heuristics

• Encodes several intuitions about MPC abstractions
• Examples
  – Blocking Heuristics
    • capsules with externally blocking behaviors
    • should be assigned Th execution policy
    • rationale: other policies may lead to blocking of the executing thread, starvation and system deadlocks.

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Execution Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking</td>
<td>Th</td>
</tr>
<tr>
<td>Heavy</td>
<td>Th</td>
</tr>
<tr>
<td>HighCPU</td>
<td>Ta</td>
</tr>
<tr>
<td>LowCPU</td>
<td>M</td>
</tr>
<tr>
<td>Hub</td>
<td>Ta</td>
</tr>
<tr>
<td>Affinity</td>
<td>M/S</td>
</tr>
<tr>
<td>Master</td>
<td>Ta</td>
</tr>
<tr>
<td>Worker</td>
<td>Ta</td>
</tr>
</tbody>
</table>

Th - Thread, Ta - Task, M - Monitor and S - Sequential

The goal of heuristics,
✓ Reduce mailbox contentions, message passing and processing overheads and cache-misses
Mapping function takes cVector as input and assigns an execution policy
- Encodes several intuitions (heuristics) as shown in the figure
- It is complete and assigns a single policy
Evaluation

• Benchmark programs (15 total)
  – that exhibits data, task, and pipeline parallelism at coarse and fine granularities.
• Comparing cVector mapping against thread-all and round-robin-task-all, random-task-all, and work-stealing-task-all
• Measured reduction in program execution time and CPU consumption over default mappings on different core settings.
% runtime improvement over default mappings, for fifteen benchmarks. For each benchmark there are four core settings (2, 4, 8, 12-cores) and for each core setting there are four bars (lth, Irr, Ir, Iws) showing improvement over four default mappings (thread, round-robin, random, work-stealing). Higher bars are better.
Results: Improvement In Program Runtime

% runtime improvement over default mappings, for fifteen benchmarks. For each benchmark there are four core settings (2, 4, 8, 12-cores) and for each core setting there are four bars (Ith, Irr, Ir, Iws) showing improvement over four default mappings (thread, round-robin, random, work-stealing). Higher bars are better.
% runtime improvement over default mappings, for fifteen benchmarks. For each benchmark there are four core settings (2, 4, 8, 12-cores) and for each core setting there are four bars (lth, Ir, lr, lws) showing improvement over four default mappings (thread, round-robin, random, work-stealing). Higher bars are better.
Results: Improvement In Program Runtime

- 12 of 15 programs showed improvements
- On average 40.56%, 30.71%, 59.50%, and 40.03% improvements (execution time) over thread, round-robin, random and work-stealing mappings respectively
- 3 programs showed no improvements (data parallel programs)

% runtime improvement over default mappings, for fifteen benchmarks. For each benchmark there are four core settings (2, 4, 8, 12-cores) and for each core setting there are four bars (lth, Irr, Ir, Iws) showing improvement over four default mappings (thread, round-robin, random, work-stealing). Higher bars are better.
% runtime improvement over default mappings, for fifteen benchmarks. For each benchmark there are four core settings (2, 4, 8, 12-cores) and for each core setting there are four bars (Ith, Irr, Ir, Iws) showing improvement over four default mappings (thread, round-robin, random, work-stealing). Higher bars are better.
We presented cVector mapping technique for capsules, however the technique should be applicable to other MPC frameworks.

- Proof of concept: evaluation on Akka
  - Similar results for most benchmarks
  - Data parallel applications show no improvements (as in Panini)
Related Works

- Placing Erlang actors on multicore efficiently, Francesquini et al. [Erlang’13 Workshop]
  - considers only hub-affinity behavior that is annotated by programmer
  - out technique takes care of many other behaviors

- Mapping task graphs to cores, Survey [DAC’13]
  - not directly applicable to JVM-based MPC frameworks, because threads to cores mapping is left to OS scheduler

- Efficient strategies for mapping threads to cores for OpenMP multi-threaded programs, Tousimojarad and Vanderbauwhede [Journal of Parallel Computing’14]
  - our technique maps capsules to threads and not threads to cores
Placing Erlang actors on multicore efficiently, Francesquini et al. [Erlang’13 Workshop]

considers only hub-affinity behavior that is annotated by programmer

- **Not directly applicable to JVM-based MPC frameworks**
- **Non-automatic**

Efficient strategies for mapping threads to cores for OpenMP multi-threaded programs, Tousimojarad and Vanderbauwhede [Journal of Parallel Computing’14]

our technique maps capsules to threads and not threads to cores
Conclusion

MPC System

Mapping

Architecture

Mapping Abstractions to JVM threads
Evaluated on Panini and Akka

Questions?

Ganesha Upadhyaya and Hridesh Rajan
{ganeshau,hridesh}@iastate.edu

Supported in part by the NSF grants CCF- 08-46059, CCF-11-17937, and CCF-14-23370.