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

Ganesha Upadhyaya
Iowa State University, USA
ganeshau@iastate.edu

Hridesh Rajan
Iowa State University, USA
hridesh@iastate.edu

Abstract

Efficient mapping of message passing concurrency (MPC) abstractions to Java Virtual Machine (JVM) threads is critical for performance, scalability, and CPU utilization; but tedious and time consuming to perform manually. In general, this mapping cannot be found in polynomial time, but we show that by exploiting the local characteristics of MPC abstractions and their communication patterns this mapping can be determined effectively. We describe our MPC abstraction to thread mapping technique, its realization in two frameworks (Panini and Akka), and its rigorous evaluation using several benchmarks from representative MPC frameworks. We also compare our technique against four default mapping techniques: thread-all, round-robin-task-all, random-task-all and work-stealing. Our evaluation shows that our mapping technique can improve the performance by 30%-60% over default mapping techniques. These improvements are due to a number of challenges addressed by our technique namely: i) balancing the computations across JVM threads, ii) reducing the communication overheads, iii) utilizing information about cache locality, and iv) mapping MPC abstractions to threads in a way that reduces the contention between JVM threads.

1. Introduction

Message-passing based concurrency (MPC) is an important approach to concurrency, where there are self-contained concurrently runnable entities that communicate via message passing [16, 20, 22, 25] (henceforth referred to as *MPC abstractions* or *abstractions*). A number of MPC frameworks support building large scale distributed applications on the Java Virtual Machine (JVM) [7, 9, 10, 15, 25].

The MPC model exposes parallelism by design, but abstractions needs to be mapped to cores carefully for utilizing multiple cores. Mapping is a two step process: 1) abstractions to threads mapping and 2) threads to cores mapping (or scheduling). Often, the MPC runtime handles both steps by creating required threads and scheduling them on different cores using an abstraction to core mapping technique.

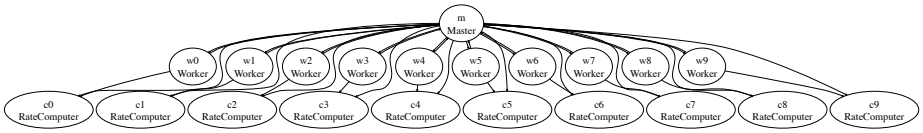
However, in case of MPC frameworks that run on JVM, abstractions to JVM threads mapping is performed by programmers and the OS scheduler handles scheduling of threads on multicore. These frameworks provide several kind of schedulers and dispatchers to programmers using which they can map the abstractions in their applications to JVM threads, e.g. Akka has four choices, Scala has two choices, Panini has four choices, etc. This also suggests that it is important to map MPC abstractions to JVM threads carefully to utilize multicore efficiently.

A large number of discussions on tuning MPC abstractions (or tuning schedulers/dispatchers) [3] indicate that programmers find it hard to manually arrive at the optimal mapping. The reasons pointed out in the discussions are: i) the mapping process is not obvious, ii) the mapping process requires deep knowledge about the tasks each MPC abstraction performs (e.g. does it do any blocking operations, read from I/O) and iii) it is not clear when and why a mapping leads to poor performance (the relationship between mappings and the performance).

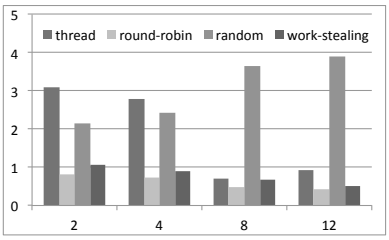
When manual tuning is hard, programmers use the default mappings (default schedulers/dispatchers) and iteratively fine tune the mappings until the desired performance is achieved. This process is easy for simple or embarrassingly parallel applications, however for applications that have sublinear performance[1], improving the performance is tedious and time consuming [29].

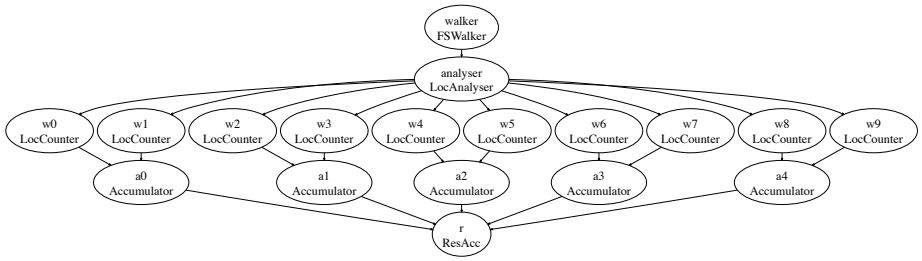
Moreover, a single default mapping strategy may not work well across programs. To illustrate the problem consider two programs shown in Figure 1. In Figure 1 we investigate the performance of four widely used default mappings: i) thread, ii) round-robin, iii) random and iv) work-stealing.

[1] These are concurrent applications that are not designed with parallelism in mind. Their performance degrades on adding more cores.



LogisticMap *from Savina [18]: computes the Logistic Map [21] using a recurrence relation.*





ScratchPad *application from Actor Collections [33]: counts lines for all files in a directory.*

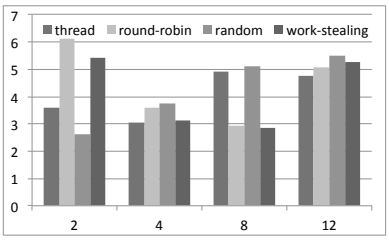


Figure 1. shows communication graphs and program execution times for LogisticMap (logmap) and ScratchPad benchmark programs. x-axis:2, 4, 8, and 12 core settings and y-axis: execution time (in seconds). Lower bars are better.

In thread mapping, every abstraction is assigned a dedicated thread and in other three mappings abstractions are tasks that are assigned to taskpool or threadpool. For LogisticMap program, round-robin task based mapping out-performs other three mappings, whereas for ScratchPad program, there is no clear winner. The relative performance of mappings also varies with number of cores. These results illustrate that single default mapping strategy may not work well across programs.

When manual tuning is hard and default mappings may not produce the desired performance, a brute force technique that tries all possible combinations of abstractions to threads mapping (using different kinds of schedulers/dispatchers) could be used. However, this approach suffers from combinatorial explosion. Even for an MPC program with few kinds of concurrent entities, the number of combinations that must be tried can grow large (# of dispatchers^# of abstractions).

Our key observation is that, *local computation and communication behavior* of a concurrent entity in a MPC program is surprisingly predictive for determining *globally beneficial* mapping to threads. Here, by computation and communication behaviors we mean properties such as: externally blocking behaviors, local state, computational workload, message send/receive pattern, and inherent parallelism. A related observation is that determining these behavior at a coarse/abstract level is sufficient to solve this problem.

To that end, this work makes several contributions[2]:

1. We propose *characteristics vector* (cVector), a representation for computation and communication behavior of an MPC abstraction. Main challenges in coming up with this representation strategy were to select suitable fields and then to formulate cVector in a *language-agnostic* manner.
2. We describe analyses for determining components of a *characteristics vector*. Main challenge here was in coming up with local analyses, which can be performed on single abstraction at a time — especially challenging for communication patterns.
3. We propose a set of novel heuristics that maps cVectors of MPC abstractions to execution policies. We encode these heuristics in the form of a decision diagram whose input is a cVector and output is an execution policy. Our technique uses the combination of cVector, heuristics and execution policies to compute mapping.
4. We implement this technique in the Panini compiler [25] to map capsules, an MPC abstraction, to threads.
5. We evaluate our approach by applying it to two MPC frameworks: Panini and Akka and examine its applicability. In the evaluation, we have selected four default mapping strategies (that are representative schedulers/dispatchers in this domain). We profile program execution time and cpu consumption and use them as metrics to compare our mapping technique against four default mapping strategies. Our results show 30%-60% improvement in program execution times when compared to default mappings. Since our technique is automatic, it could help decrease the efforts required for performance tuning.

The rest of this paper is organized as follows. Next, we describe our technique, §3 describes evaluations, §4 compares and contrasts with related work, and §5 concludes.

[2] We build upon our preliminary AGERE 2014 workshop paper [36]. The cVector representation and analysis techniques in this work is novel, especially the local treatment of incoming and outgoing message patterns. We have considered four standard default mapping strategies to compare against instead of two (thread and round-robin-task). Finally, our mapping technique further improved the program execution time and cpu consumptions compared to [36].

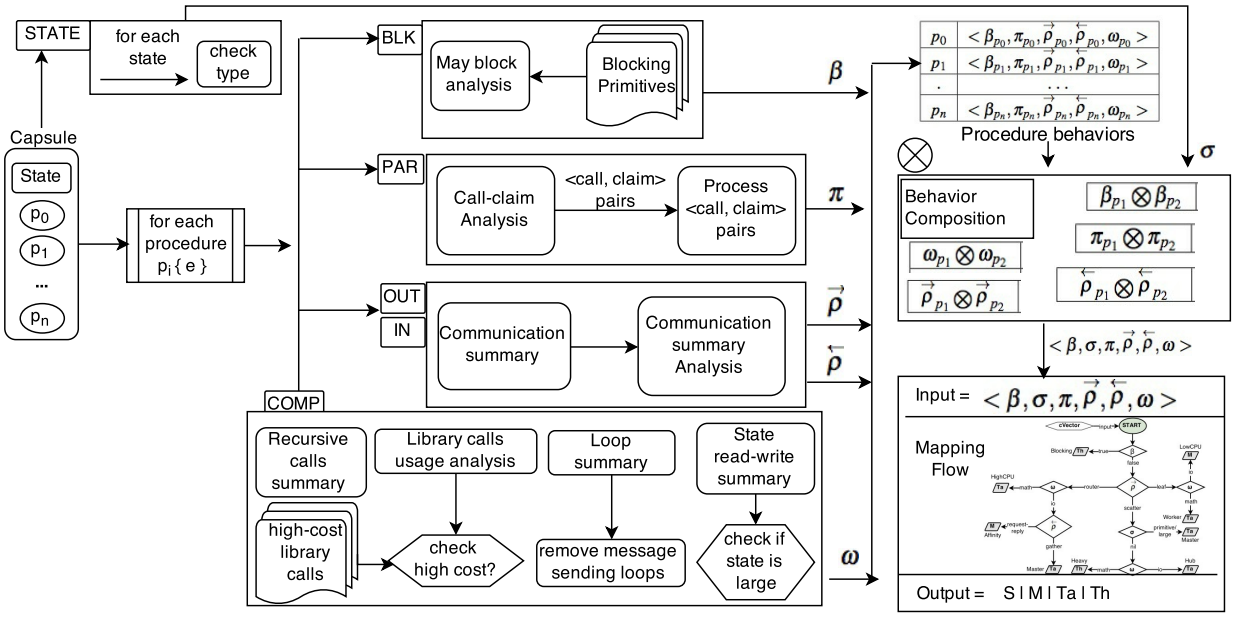


Figure 2. Approach Overview: cVector computation and mapping to execution policy

2. MPC Abstraction to Thread Mapping

In this section we describe our technique for mapping MPC abstractions to threads. An overview is presented in Figure 2. We explain our approach using *capsules* [24, 25], a kind of MPC abstraction and then revisit its applicability to other approaches. A capsule is a template for creating capsule instances. Capsules can declare *procedures (that act as message handlers)*, *states (that act as confined local storage)*, static type of other capsule instances that can be reached, as well as its internal design which may consist of other capsule instances. Each capsule instance strongly confines its state that may consist of primitive types as well as references types, hides its internal design, allows a single thread within its execution that serves incoming requests (represented as procedure calls) in a FIFO manner. Capsules allow a programmer to statically identify interference points and modularly reason about the behavior of interfering tasks at those points [8]. A detailed description appears in §A.

2.1 Computation & Communication Behaviors

We auto-tune MPC applications by balancing the computations (performed by various entities) across available resources and reducing the communication overheads. Selecting computation and communication behaviors to analyze is the first step toward that goal with four challenges: representativeness, analyzability (w.r.t. automated program analysis), completeness, and orthogonality. Based on these criteria, we selected the following archetypal behaviors.

- *The blocking behavior* can arise due to I/O, socket or database blocking primitives. When a concurrent entity, e.g. capsule, externally blocks, it not only adds additional overhead to its computation, it may also lead to starvation of other concurrent entities in the system (the thread processing this entity may not be available to process messages from other entities when they share a thread).
- *The local state* i.e. the set of state variables has important implication on the cache behavior. For example, when a capsule has a large local state, the thread processing this capsule's message can benefit if capsule's local state is available in its cache while processing the subsequent messages (improves cache locality).
- *The computational workload* i.e. the nature of computations performed by the MPC abstraction determines its resource needs. For example, capsules may spend their time performing computations or waiting to receive messages. Capsules that have low computation-to-wait ratio could share threads resulting in overall saving of resources.
- *The communication pattern* (more precisely message send and receive patterns) can help inform colocation of senders and recipients. For instance, consider an application that has two capsules Sender and Receiver. Sender implements a one-to-many send pattern (for every message it receive, it sends N messages to Receiver, where N is a large number). Since Sender communicates with Receiver often, by mapping Sender and Receiver to the same thread, message processing overheads could be greatly reduced.
- *The inherent parallelism* focuses on the scenarios where a response is expected for a request before computation may continue. Between the request and the response, the

ν	::=	$\langle \beta, \sigma, \pi, \overrightarrow{\rho}, \overleftarrow{\rho}, \omega \rangle$	*cVector*
β	::=	*true* | *false*	*blocking behavior*
σ	::=	*nil*	*stateless abstraction*
	|	*fixed*	*fixed size state*
	|	*variable*	*variable size state*
π	::=	*sync*	*synchronous*
	|	*async*	*asynchronous*
	|	*future*	*logically synchronous*
$\overrightarrow{\rho}$	::=	*leaf*	*no outgoing communication*
	|	*router*	*one-to-one communication*
	|	*scatter*	*batch communication*
$\overleftarrow{\rho}$	::=	*gather*	*recv-to-send > 1*
	|	*request-reply*	*recv-to-send <= 1*
ω	::=	*math*	*computation-to-wait > 1*
	|	*io*	*computation-to-wait <= 1*

Figure 3. Characteristics Vector (cVector) Representation.

sender can also perform some of its own computation that does not depend on response thereby exposing implicit parallelism. This metric measures the nature of this computation. For example, if the response is needed immediately after a request is sent then the inherent parallelism is zero.

2.2 Characteristics Vector (cVector) Representation

We denote the computation and communication behaviors described in §2.1 using a new representation that we call *characteristic vector* (cVector) . A cVector can represent computation and communication behavior for both procedures (handlers) and capsules (MPC abstraction). It has six fields as shown in Figure 3. Here, β represents *blocking behavior*, σ represents *local state*, π represents *inherent parallelism*, $\overrightarrow{\rho}$ represents outgoing message pattern, $\overleftarrow{\rho}$ represents the incoming message pattern (or receive pattern), and ω represents *computational workload*.

The field σ can take value *nil* for stateless, *variable* if the capsule state can grow or shrink as a result of message receive event, and *fixed* otherwise. In definition of field $\overrightarrow{\rho}$ *recv-to-send* is a ratio of number of messages received to number of messages sent. In definition of field ω, *computation-to-wait* is a ratio of time spent on performing computation to time spent on waiting to receive messages.

2.3 cVector Analysis

For mapping, we first compute a cVector for each capsule.

2.3.1 Inherent parallelism analysis

The inherent parallelism analysis calculates the field π. It classifies a capsule into one of *sync*, *async*, and *future*. The classification *sync* represents that the communication pattern of current capsule with others is practically synchronous, i.e. wait for result immediately after sending a message and consuming it. The classification *async* represents that a result is not required after sending a message. The classification *future* represents that a result is not immediately required after sending a message.

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

$$\pi_1 \bigoplus \pi_2 = \begin{cases} sync & \text{if } \pi_1 = sync \text{ or } \pi_2 = sync \\ async & \text{if } \pi_1 = async \text{ and } \pi_2 = async \\ future & \text{otherwise.} \end{cases}$$

Algorithm 1: Analyzing call-claim pairs

We use π_p to represent inherent parallelism in the capsule procedure p. A capsule's inherent parallelism is determined by combining the inherent parallelisms of its procedures. To determine π_p of the capsule procedure p, a call-claim analysis (similar to def-use analysis) examines p's body *{e}* and collects (*call*,*claim*) pairs, where *call* and *claim* are subexpressions of p's body *{e}*. Upon collecting the (*call*,*claim*) pairs, they are analyzed to compute the value of π_p for each procedure using Algorithm 1. These values are combined to determine π for the capsule using the behavior composition operator $\bigotimes$ defined in Figure 6. Basic intuition is to clearly identify synchronous behavior, and then distinguish between cases when a response is needed (*future*) or not.

2.3.2 Communication pattern analysis

Generally, the topology is required to determine communication patterns. One of our key challenge was to infer them by analyzing the structure of procedures, especially occurrence of send and receive primitives in the procedure body.

We use $\overrightarrow{\rho}_p$ and $\overleftarrow{\rho}_p$ to refer to message send and receive patterns of a capsule procedure p. We first compute a summary of the procedure p's body *{e}*, which we call *communication summary*, by abstracting away expressions except message sends and state read/write as defined in Figure 4. The communication summaries from private helper procedures (only accessible from within a capsule) are inlined. In the description we use a term *connected capsules* to refer to the set of capsules that a capsule can send message to. Sending multiple messages to a connected capsule and sending a message to all connected capsules are both considered *send all* for the purpose of this intra-procedural analysis.

We then analyze communication summaries to determine communication pattern as shown in Figure 5. The analysis is

ξ	::=		*communication summary*
	|	$\{\}$	*empty*
	|	s	*statement*
s	::=		*statements in communication summary*
	|	*stateRW*	*state read/write*
	|	*exit*	*exit calls*
	|	$!i$	*message send*
	|	$?!i$	*conditional message send*
	|	$\langle !i \rangle$	*send all,* $i \in I$
	|	$?\langle !i \rangle$	*conditional send all,* $i \in I$
	|	$s;s$	*sequence*

where, i is a connected capsule and
I is the set of all connected capsules

Figure 4. Syntax of Communication Summary

a function Ξ that takes communication summary ξ and produces a tuple $(\overrightarrow{\rho}, \overleftarrow{\rho})$. An empty communication summary results in default values (leaf, gather). The symbol $\bullet$ indicates no change to the current behavior.

$$
\begin{array}{lcl}
\Xi(\xi) & ::= & (\overrightarrow{\rho}, \overleftarrow{\rho}) \\
\Xi(\{\}) & = & (\textit{leaf, gather}) \\
\Xi(\textit{stateRW}) & = & (\bullet, \textit{gather}) \\
\Xi(\textit{exit}) & = & (\bullet, \textit{request-reply}) \\
\Xi(!i) & = & (\textit{router, gather}) \\
\Xi(?!i) & = & (\textit{router, gather}) \\
\Xi(\langle !i \rangle) & = & (\textit{scatter, request-reply}) \\
\Xi(?\langle !i \rangle) & = & (\textit{router, gather}) \\
\Xi(s_1;s_2) & = & \Xi(s_1) \bigoplus \Xi(s_2)
\end{array}
$$

$$
(\overrightarrow{\rho}_1, \overleftarrow{\rho}_1) \bigoplus (\overrightarrow{\rho}_2, \overleftarrow{\rho}_2) = (\overrightarrow{\rho}_1 \bigoplus \overrightarrow{\rho}_2, \overleftarrow{\rho}_1 \bigoplus \overleftarrow{\rho}_2)
$$

$$
\overrightarrow{\rho}_1 \bigoplus \overrightarrow{\rho}_2 = \begin{cases} \text{scatter} & \text{if } \overrightarrow{\rho}_1 = \text{scatter or } \overrightarrow{\rho}_2 = \text{scatter} \\ \text{leaf} & \text{if } \overrightarrow{\rho}_1 = \text{leaf and } \overrightarrow{\rho}_2 = \text{leaf} \\ \text{router} & \text{otherwise.} \end{cases}
$$

$$
\overleftarrow{\rho}_1 \bigoplus \overleftarrow{\rho}_2 = \begin{cases} \text{gather} & \text{if } \overleftarrow{\rho}_1 = \text{gather or } \overleftarrow{\rho}_2 = \text{gather} \\ \text{request-reply} & \text{otherwise.} \end{cases}
$$

Figure 5. Communication Summary Analysis

The operator $\bigoplus$ combines the behaviors of sequence of statements in the communication summary. $\overrightarrow{\rho}_1 \bigoplus \overrightarrow{\rho}_2$ defines the dominance relation: *scatter* $\gg$ *router* $\gg$ *leaf* and $\overleftarrow{\rho}_1 \bigoplus \overleftarrow{\rho}_2$ defines the dominance relation: *gather* $\gg$ *request-reply*. This assures every sign of batch communication between capsules is captured. The communication patterns of a capsule are determined by combining communication patterns of capsule procedures. Figure 6 defines how capsule procedure's communication patterns are combined to form capsule's communication patterns.

2.3.3 May block and state analysis

To compute blocking behavior (β) we first leverage a manually created dictionary of blocking library calls as an input to our analysis. The may block analysis is then realized as a flow analysis with message receives as sources and blocking library calls as sinks. A capsule has blocking behavior if any of its procedures have blocking behavior.

For state analysis (to compute σ) we check the type of each state variable. The analysis assigns value *fixed* when state variables are of primitive data types or any data type with fixed size and *variable* when state variables use data structures such as collections, maps etc.

2.3.4 Computational workload analysis

We classify the computational workload to be *math* if a capsule spends time performing computations; when procedure has recursive calls, loops with unknown bounds, makes high cost library calls and read/write to capsule state that is *variable* size, and *io*, if capsule is mostly waiting to receive messages. The analysis gathers computation summaries for each capsule procedure. The computation summary includes recursive function calls, high cost library calls, loops with unknown bounds (input dependent) and read/write to capsule state that is *variable* size. Based on the computation summaries each capsule procedure is assigned *math* or *io*. The computation behavior of the capsule is combined using behavior composition defined in Figure 6.

2.3.5 Behavior Composition

A capsule may have multiple procedures. The behavior of the capsule is determined by combining the behaviors of its procedures using behavior composition function ($\bigotimes$) defined in Figure 6. The behavior composition function takes behaviors of two procedures and produces the combined behavior. The behavior composition function is both commutative and associative. If a capsule has more than two procedures, the behavior composition function can be applied by taking two procedures at a time in any order.

Figure 6 shows the composition function for blocking (β), inherent parallelism (π), message send pattern ($\overrightarrow{\rho}$), message receive pattern ($\overleftarrow{\rho}$) and computational workload (ω) behaviors. The behavior composition for blocking ($\beta_{p_1} \bigotimes \beta_{p_2}$) can be defined as follows. A capsule has blocking behavior if any of its procedures are blocking. Similarly, for message receive pattern ($\overleftarrow{\rho}_{p_1} \bigotimes \overleftarrow{\rho}_{p_2}$), a capsule's message receive pattern is *gather* if any of its procedures have receive pattern *gather*.

For other cVector fields the composition is defined in relation to message receive pattern ($\overleftarrow{\rho}$). For instance, consider the behavior composition function $\omega_{p_1} \bigotimes \omega_{p_2}$ shown in Figure 6. For, $\omega_{p_1} = io$ and $\omega_{p_2} = math$, the result is described by the auxiliary function $cv_3(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2})$ that takes two parameters $\overleftarrow{\rho}_{p_1}$ and $\overleftarrow{\rho}_{p_2}$. A value *io* is assigned only when $\overleftarrow{\rho}_{p_1}$ is *gather* and for all other cases *math* is assigned. In a nutshell, capsule's message send pattern ($\overrightarrow{\rho}$), inherent parallelism (π) and computational workload (ω) behaviors are

$$\beta_{p_1} \bigotimes \beta_{p_2} = \begin{cases} \text{true} & \text{if } \beta_{p_1} = \text{true or } \beta_{p_2} = \text{true} \\ \text{false} & \text{otherwise.} \end{cases}$$

$$\pi_{p_1} \bigotimes \pi_{p_2} = \begin{cases} \text{sync} & \text{if } \pi_{p_1} = \text{sync and } \pi_{p_2} = \text{sync or future} \\ cv_6(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}, \pi_{p1}) & \text{if } \pi_{p_1} = \text{sync and } \pi_{p_2} = \text{async} \\ cv_5(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}, \pi_{p2}) & \text{if } \pi_{p_1} = \text{async and } \pi_{p_2} = \text{sync or future or async} \\ \text{sync} & \text{if } \pi_{p_1} = \text{future and } \pi_{p_2} = \text{sync} \\ cv_6(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}, \pi_{p1}) & \text{if } \pi_{p_1} = \text{future and } \pi_{p_2} = \text{async} \\ \text{future} & \text{if } \pi_{p_1} = \text{future and } \pi_{p_2} = \text{future} \end{cases}$$

$$\overrightarrow{\rho}_{p_1} \bigotimes \overrightarrow{\rho}_{p_2} = \begin{cases} \text{leaf} & \text{if } \overrightarrow{\rho}_{p_1} = \text{leaf and } \overrightarrow{\rho}_{p_2} = \text{leaf} \\ \overrightarrow{\rho}_{p_2} & \text{if } \overrightarrow{\rho}_{p_1} = \text{leaf and } \overrightarrow{\rho}_{p_2} != \text{leaf} \\ \text{router} & \text{if } \overrightarrow{\rho}_{p_1} = \text{router and } \overrightarrow{\rho}_{p_2} = \text{router} \\ cv_1(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}) & \text{if } \overrightarrow{\rho}_{p_1} = \text{router and } \overrightarrow{\rho}_{p_2} = \text{scatter} \\ cv_2(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}) & \text{if } \overrightarrow{\rho}_{p_1} = \text{scatter and } \overrightarrow{\rho}_{p_2} = \text{router} \end{cases}$$

$$\overleftarrow{\rho}_{p_1} \bigotimes \overleftarrow{\rho}_{p_2} = \begin{cases} \text{gather} & \text{if } \overleftarrow{\rho}_{p_1} = \text{gather or } \overleftarrow{\rho}_{p_1} = \text{gather} \\ \text{request-reply} & \text{otherwise.} \end{cases}$$

$$\omega_{p_1} \bigotimes \omega_{p_2} = \begin{cases} \text{io} & \text{if } \omega_{p_1} = \text{io and } \omega_{p_2} = \text{io} \\ \text{math} & \text{if } \omega_{p_1} = \text{math and } \omega_{p_2} = \text{math} \\ cv_3(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}) & \text{if } \omega_{p_1} = \text{io and } \omega_{p_2} = \text{math} \\ cv_4(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}) & \text{if } \omega_{p_1} = \text{math and } \omega_{p_2} = \text{io} \end{cases}$$

$$cv_1(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}) = \begin{cases} \text{router} & \text{if } \overleftarrow{\rho}_{p_1} = \text{gather and} \\ & \overleftarrow{\rho}_{p_2} = \text{request-reply} \\ \text{scatter} & \text{otherwise.} \end{cases}$$

$$cv_2(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}) = \begin{cases} \text{router} & \text{if } \overleftarrow{\rho}_{p_1} = \text{request-reply and} \\ & \overleftarrow{\rho}_{p_2} = \text{gather} \\ \text{scatter} & \text{otherwise.} \end{cases}$$

$$cv_3(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}) = \begin{cases} \text{io} & \text{if } \overleftarrow{\rho}_{p_1} = \text{gather and} \\ & \overleftarrow{\rho}_{p_2} = \text{request-reply} \\ \text{math} & \text{otherwise.} \end{cases}$$

$$cv_4(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}) = \begin{cases} \text{io} & \text{if } \overleftarrow{\rho}_{p_1} = \text{request-reply and} \\ & \overleftarrow{\rho}_{p_2} = \text{gather} \\ \text{math} & \text{otherwise.} \end{cases}$$

$$cv_5(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}, \pi_{p2}) = \begin{cases} \text{async} & \text{if } \overleftarrow{\rho}_{p_1} = \text{gather and} \\ & \overleftarrow{\rho}_{p_2} = \bullet \\ \pi_{p2} & \text{otherwise.} \end{cases}$$

$$cv_6(\overleftarrow{\rho}_{p_1}, \overleftarrow{\rho}_{p_2}, \pi_{p1}) = \begin{cases} \text{async} & \text{if } \overleftarrow{\rho}_{p_1} = \bullet \text{ and} \\ & \overleftarrow{\rho}_{p_2} = \text{gather} \\ \pi_{p1} & \text{otherwise.} \end{cases}$$

Figure 6. Behavior composition operator $\bigotimes$ for cVector fields, where $\bullet$ represents any value from the domain of values.

predominantly defined by the behaviors of the procedure that has *gather* receive pattern.

2.4 Execution Policies for MPC Abstractions

Once cVector is computed, the mapping technique assigns an execution policy to each capsule in the program, which defines how messages are processed. We have considered following representative execution policies (similar to dispatchers available in widely used JVM-based MPC frameworks).

- *THREAD (Th).* A dedicated thread is assigned for processing the messages from the capsule's message queue and executing the corresponding behavior.
- *TASK (Ta).* The capsule messages are processed by the shared thread of the taskpool. The taskpool may contain one or more capsules that abide to *TASK* execution policy. The order in which the messages from different capsules message queue has to be processed could vary. One simple policy is to process one message from each capsule to avoid starvation of other capsules.
- *SEQ (S) / MONITOR (M).* The capsule that sends the message needs to execute the defined behavior at the capsule that received the message. The difference between sequential (S) and monitor (M) policies is that in monitor all message handlers are lock protected using synchronized access.

Execution policies define mapping to threads, e.g. assigning thread execution policy leads to one-to-one capsules to threads mapping. Assigning task execution policy leads to many-to-one capsules to threads mapping.

2.5 Execution Policy Selection

After cVector analysis, our technique utilizes the decision tree shown in Figure 7 to determine a single execution policy for each capsule. This decision function is complete, and assigns a single policy to each capsule. The decision tree encodes several intuitions that we now describe in the rest of this section.

2.5.1 Blocking Heuristics

This heuristic states that a capsule that has externally blocking behavior, should be assigned thread (Th) execution policy. This is because assigning any other policy to blocking capsules may block the executing thread, may lead to starvation of other capsules and system deadlock. Consider a capsule that reads an input file in chunks and sends the read data to other capsule/capsules. This capsule has I/O blocking behavior. If this capsule is assigned a policy other than thread (Th), for instance, task (Ta), other task capsules that share a processing thread could starve until the capsule reading the input file finishes.

2.5.2 Heavy Heuristics

This heuristic states that a capsule that is non-blocking, communicates often with other capsules and performs useful computations (determined by ω = *math*) should be assigned thread (Th) execution policy. The rationale is that the dedicated thread can perform its computations in parallel with other threads without voluntarily interruptions (capsules sharing a thread leads to interruptions to avoid starvation and enable fairness). If these capsules are assigned task (Ta) policy, the thread processing such capsules is tied up by the uninterrupted long computations. This may lead to starvation of other capsules (that are assigned task policy) which are served by the same thread. The cVector of such a capsule is: <*false*, *nil*, •, *scatter*, •, *math*>. Note that, • as value for any cVector field indicates that any value from the domain of values can be assigned and the value does not influence the mapping decision.

2.5.3 HighCPU Heuristics

A capsule that is non-blocking, communicates less often with other capsules and performs useful computations should be assigned task (Ta) execution policy. The cVector of such capsules are: <*false*, •, •, *router*, •, *math*>. By assigning task execution policy, the capsule can share a thread with other capsules (which are also assigned task policy) and it can benefit from the work-stealing optimizations that are available for the task/thread pools. Note that, if thread (Th) policy is assigned, these capsules cannot avail those optimization benefits. Unlike the capsules described under Heavy Heuristics, these capsules communicates with other capsules less often, which makes them the suitable candidates for availing the work-stealing optimization benefits (work-stealing performs better when tasks are independent computations without much interactions [23]).

2.5.4 LowCPU Heuristics

A capsule that has cVector like <*false*, •, •, *leaf*, •, *io*> should be assigned monitor (M) execution policy. Such a capsule does not need a thread of its own for processing messages and the threads of the capsules that sends the message themselves will process the messages of the capsule. An example of such a capsule is a capsule that prints "hello world!". If this capsule is assigned thread/task policy, the overheads involved in message processing exceeds the computations performed in the capsule.

2.5.5 Hub Heuristics

This heuristic states that hub capsules (capsules that have the tendency of communicate often with a set of affinity capsules) should be assigned task (Ta) execution policy. Hub capsules are represented using cVector <*false*, *nil*, •, *scatter*, •, *io*>. In the literature, it is known that hub and its affinity capsules must be placed closer to each other to reduce communication overheads [11]. We achieve this by assigning task policy to hub capsules such that the shared thread that is processing the hub capsules also processes the affinity capsules. This can also be achieved by assigning thread (Th) policy to hub capsules however, hub capsules may not need the thread of their own as they are not computationally intensive (indicated by cVector field ω = *io*).

2.5.6 Affinity Heuristics

Affinity capsules should be assigned monitor (M) execution policy. Affinity capsules have following cVector <*false*, •, •, *router*, *request-reply*, *io*>. By assigning monitor execution policy, affinity capsules are colocated with hub capsules. The thread that is processing the hub capsule will also process its

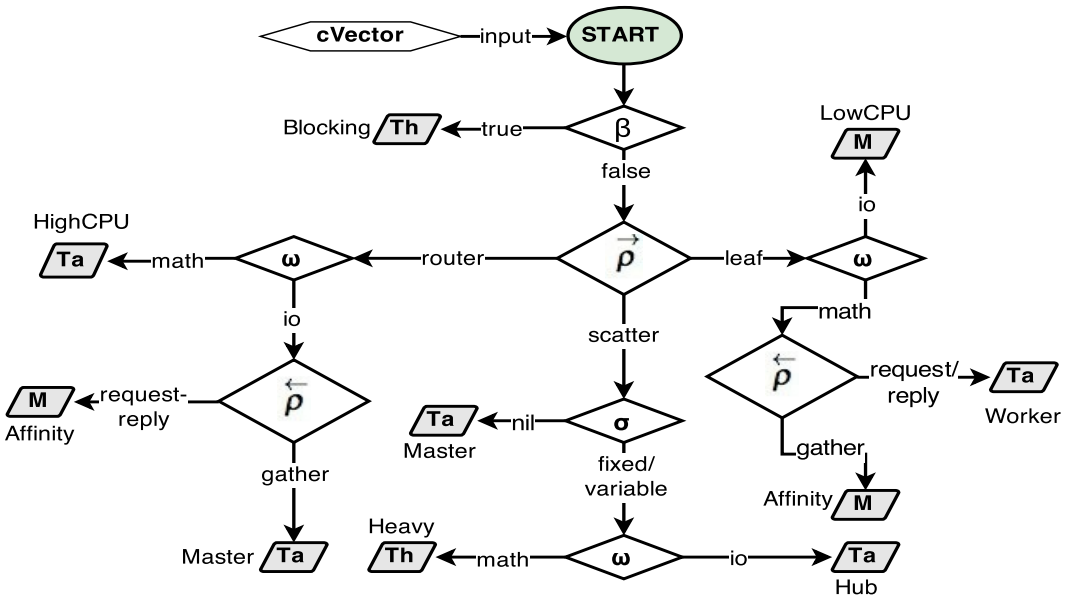


Figure 7. Decision diagram for execution policy selection.

affinity capsules. This makes our policy assignment process modular as we bind the hub capsules with their affinities without using the topology.

2.5.7 Master Heuristics

There are two types of master capsules. The first type has cVector <*false*, *fixed/variable*, •, *scatter*, •, *io*>. This type of master capsules sends messages to worker capsules and may not receive reply from the workers. The second type has cVector <*false*, •, •, *router*, *gather*, *io*>. The rationale for selecting task execution policy for these capsules is the following. These capsules either dispatch work to other capsules and go idle or they stay idle until the expected number of messages are received from other capsules. In either case, these capsules can share a thread with any other capsules. Assigning thread policy may lead to a case where the dedicated thread is spending majority of its time idling. Hence, we assign task (Ta) policy for this class of capsules.

2.5.8 Worker Heuristics

This heuristic states that worker capsules should be assigned task (Ta) execution policy. The cVector of worker capsules is <*false*, •, •, *leaf*, •, *math*>. Similar to HighCPU capsules, these capsules can utilize the load-balancing strategies applied to the task/thread pool.

In summary, the heuristics are designed to assign execution policies to capsules based on the characteristics described by their cVectors. Changing the policy may lead to increase in mailbox contention, message passing overhead, message processing overhead and cache-misses. Our performance analysis section further demonstrates these consequences.

2.6 Illustrative Example

We now present an example where we construct cVectors for capsules and apply our mapping function to determine the execution policies for capsules. Consider the capsule communication graph (topology) of LogisticMap program from Savina [18] shown in Figure 1. This Panini program has three types of capsules: Master, SeriesWorker (10 instances) and RateComputer (10 instances). Master communicates with each SeriesWorker and RateComputer. Each SeriesWorker capsule replies back to Master and communicates with its RateComputer capsule instance. One can see that the communication graph of LogisticMap Panini program is not simple and determining the execution policies to capsules (or capsules to threads mapping) for such a program is non-trivial. For capsules in LogisticMap Panini program, we construct cVectors and determine the execution policies using our technique.

2.6.1 Master

This capsule does not use blocking calls, hence $\beta = false$. It has local state defined using primitive data types, hence $\sigma = fixed$. Figure 8 shows two capsule procedures begin and process. Using Figure 4, communication summary of begin and process are $\langle !i \rangle;\langle !i \rangle$ and $\langle ?!i \rangle;\langle !i \rangle$. The communication pattern *(scatter, gather)* is assigned to both procedures using the analysis described in Figure 5. Using behavior composition ($\otimes$) the communication pattern *(scatter, gather)* is assigned to Master capsule. The inherent parallelism, $\pi = async$, as all message sends are asynchronous in capsule procedures and $\omega = io$, as there is no computation intensive operations in either of the procedures. Hence, cVector for Master capsule is, <*false*, *fixed*, *async*, *scatter*, *gather*, *io*>

```
void begin(StartMessage sm) {
  int i = 0;
  while (i < LogisticMapConfig.numTerms) {
    for (SeriesWorker worker : workers) {
      worker.nextTerm();
    }
    i += 1;
  }
  for (SeriesWorker worker : workers) {
    worker.getTerm();
    numWorkRequested += 1;
  }
}
```

```
void process(ResultMessage rm) {
  termsSum += rm.term;
  numWorkReceived += 1;
  if (numWorkRequested == numWorkReceived) {
    System.out.println("Terms sum: " + termsSum);
    for (SeriesWorker worker : workers) {
      worker.done();
    }
    for (RateComputer computer : computers) {
      computer.done();
    }
  }
}
```

```
double curTerm = 0;
void nextTerm() {
  int senderId = id;
  curTerm = computer.compute(new
    ComputeMessage(senderId, curTerm));
}
void getTerm() {
  master.process(new Result(curTerm));
}
void done() {
  exit();
}
```

```
double rate = 0.0;
Result compute(ComputeMessage cm) {
  double result = computeNextTerm(cm.term, rate);
  int senderId = cm.senderId;
  return new Result(result);
}
void done() {
  exit();
}
```

Figure 8. First two code snippets shows two procedures begin and process of Master capsule. Third and fourth code snippets shows procedures of SeriesWorker and RateComputer capsules respectively. For brevity, we show only the required code and the complete source code is available in [4]

and by following the mapping function (shown in Figure 7) for this cVector gives us task (Ta) execution policy.

2.6.2 SeriesWorker

This capsule is non-blocking and has local states defined using primitive data types, hence $\beta = false$ and $\sigma = fixed$. Figure 8 shows three capsule procedures, nextTerm, getTerm and done. The communication summaries and the assigned pattern are listed below.

nextTerm	*stateRW;!i*	*(router, gather)*
getTerm	*!i*	*(router, gather)*
done	*exit*	*(leaf, request-reply)*

By using the behavior composition, we can determine that the communication pattern of SeriesWorker capsule is *(router, gather)*. None of the procedures have intensive computations, hence $\omega = io$. The procedure, nextTerm uses the returned result immediately, hence $\pi = sync$. The procedure, getTerm uses asynchronous send, hence $\pi = async$. For SeriesWorker, $\pi = sync$ using behavior composition ($\otimes$). The cVector for SeriesWorker capsule is, <$false$, $fixed$, $sync$, $router$, $gather$, io> and the execution policy for this cVector is task (Ta) execution policy.

2.6.3 RateComputer

This capsule is non-blocking and has local state defined using primitive data types, hence $\pi = false$ and $\sigma = fixed$. Figure 8 shows procedures, compute and done. The communication summaries and the assigned pattern are listed below.

compute	*stateRW*	*(leaf, gather)*
done	*exit*	*(leaf, request-reply)*

By using the behavior composition, we can determine that the communication pattern of RateComputer capsule is *(leaf, gather)*. For this capsule, $\pi = async$ and $\omega = io$. Hence, cVector is, <$false$, $fixed$, $async$, $leaf$, $gather$, io> and execution policy for this cVector is monitor (M) execution policy.

2.7 Applicability to Other MPC Frameworks

In general, the proposed technique is applicable to other JVM-based MPC frameworks. In the proposed technique, we have selected abstraction behaviors that are commonly seen in MPC frameworks. We have represented the abstraction behaviors as cVectors, the local program analysis for determining coarse/abstract values to cVectors only uses the abstraction (does not rely on the topology), the execution policies described are available in most MPC frameworks and the heuristics that maps abstraction cVectors to execution policies are based on the intuitions of general MPC abstractions. In §3.4 we evaluated cVector mappings for Akka [9] as a proof of concept.

Suite	Program	LOC	#CT	#CI	#Messages	Type & Pattern	Purpose	Source
BenchErl	bang	64	2	441	387200	Concurrency	Measure overheads in message delivery, mailbox contention	Erlang
	mbrot	105	2	9	10911			Erlang
	serialmsg	86	3	241	491640			Erlang
CLBG	Fannkuchred	248	2	9	17	Data Parallelism	Load-balancing	Java
	fasta	269	4	6	133511	Task Parallelism	Synchronous communication	Java
	Knucleotide	228	3	48	104	Data Parallelism	Load-balancing	Java
AC	FileSearch	242	4	13	16551	Concurrency, Parallelism	Messaging throughput	Akka/Scala
	ScratchPad	188	5	18	80005			Akka/Scala
	Polynomial	191	3	502	20504			Akka/Scala
Streamit	BeamFormer	301	7	50	610000	Task and Pipeline Parallelism	Messaging throughput, Latency	Streamit
	DCT	336	8	42	720012			Streamit
Savina	logmap	176	3	21	525054	Concurrency	Synchronous request-response	Scala
	concdict	138	3	22	400063		Reader-Writer concurrency	Scala
	concsll	257	3	22	320063			Scala
JG	RayTracer	905	2	11	15	Data Parallelism	Load-balancing	Java

Figure 9. Benchmark characteristics. #CT: Number of capsule types, #CI: Number of capsule instances.

3. Evaluation

We have evaluated our technique by applying it to two MPC frameworks: Panini [1] (see §3.3) and Akka MPC framework [9] (see §3.4). We have also evaluated the precision of cVector analysis (see §3.5).

3.1 Benchmarks

For evaluating our technique, we have selected representative programs from Erlang BenchErl Suite [6], Actor Collections [33], Computer Language Benchmarks Game [12], JavaGrande [31], StreamIt Benchmarks [34], and Savina Actor Benchmarks [18]. The selected benchmark programs are concurrent or parallel applications, which exhibit different concurrency patterns and parallelisms (data, task, pipeline). These applications show super-linear, linear and sub-linear speedups and they may not scale well when additional cores are allocated to them. A detailed characteristics of the benchmark programs can be found in Figure 9. Our idea is to evaluate a wide range of programs rather than be repetitive. While selecting the representative programs from different benchmark suites, we have included programs that consists of abstractions with different behaviors and their interactions are not straightforward. We have translated a total of fifteen programs to Panini [1, 25] and eight programs to Akka [9] for evaluation. The translations of these programs have one-to-one correspondence with the source language program (meaning, MPC abstractions in the translated program maps to the MPC abstractions in the source program).

3.2 Methodology

We compare our mapping technique against four widely used execution policies in JVM-based MPC frameworks: 1) *thread*, 2) *round-robin*, 3) *random* and 4) *work-stealing* (hereon, we refer to these policies as default policies). In *thread* policy, each abstraction (capsule, actor, etc) is assigned a dedicated JVM thread. In *round-robin* policy, a collection of abstractions are served by a pool of threads in round-robin manner. In *random* policy, a collection of abstractions are served by a pool of threads at random, i.e. a thread picks an abstraction to serve at random. In *work-stealing* policy, abstractions are assigned to threads but these threads can steal work from other abstractions, if idle. We have implemented these four default policies in Panini Capsules and our comparison uses the same Panini program.

We measure program *runtime* and *CPU consumption* for *thread*, *round-robin*, *random*, *work-stealing* and our technique when the steady-state performance is reached by following the methodology of Georges *et al.*[13]. We compare program *runtime* and *CPU consumption* for these five policies on 2, 4, 8, and 12 cores settings (Linux taskset utility is used for altering core settings on 12-core system). The experiments are conducted on 12-core system (2 Six-Core AMD Opteron ® 2431 Processors) with 24GB of memory running the Linux version 3.5.5 and Java version 1.7.0_06. A Java VM max heap size of 2GB is sufficient to run all of our experiments.

3.3 Performance Evaluation

For comparing the performance of our technique against default policies, we define I_{th}, I_{rr}, I_r and I_{ws} as percentage reduction (or improvement) in program *runtime* over *thread*, *round-robin*, *random* and *work-stealing* policies respectively. Note that, we reuse these metrics for comparing the percentage reduction in program *CPU consumption* over *thread*, *round-robin*, *random* and *work-stealing* policies.

We compute I_{th}, I_{rr}, I_r and I_{ws} for each benchmark program for *runtime* and *CPU consumption* on 2, 4, 8, and 12 core settings. We also compute average I_{th}, I_{rr}, I_r and I_{ws} to determine overall performance improvement of our technique over default policies for program *runtime* and *CPU consumption*.

3.3.1 Results

Figure 10 shows I_{th}, I_{rr}, I_r and I_{ws} for both program *runtime* and *CPU consumption* for our benchmark programs.

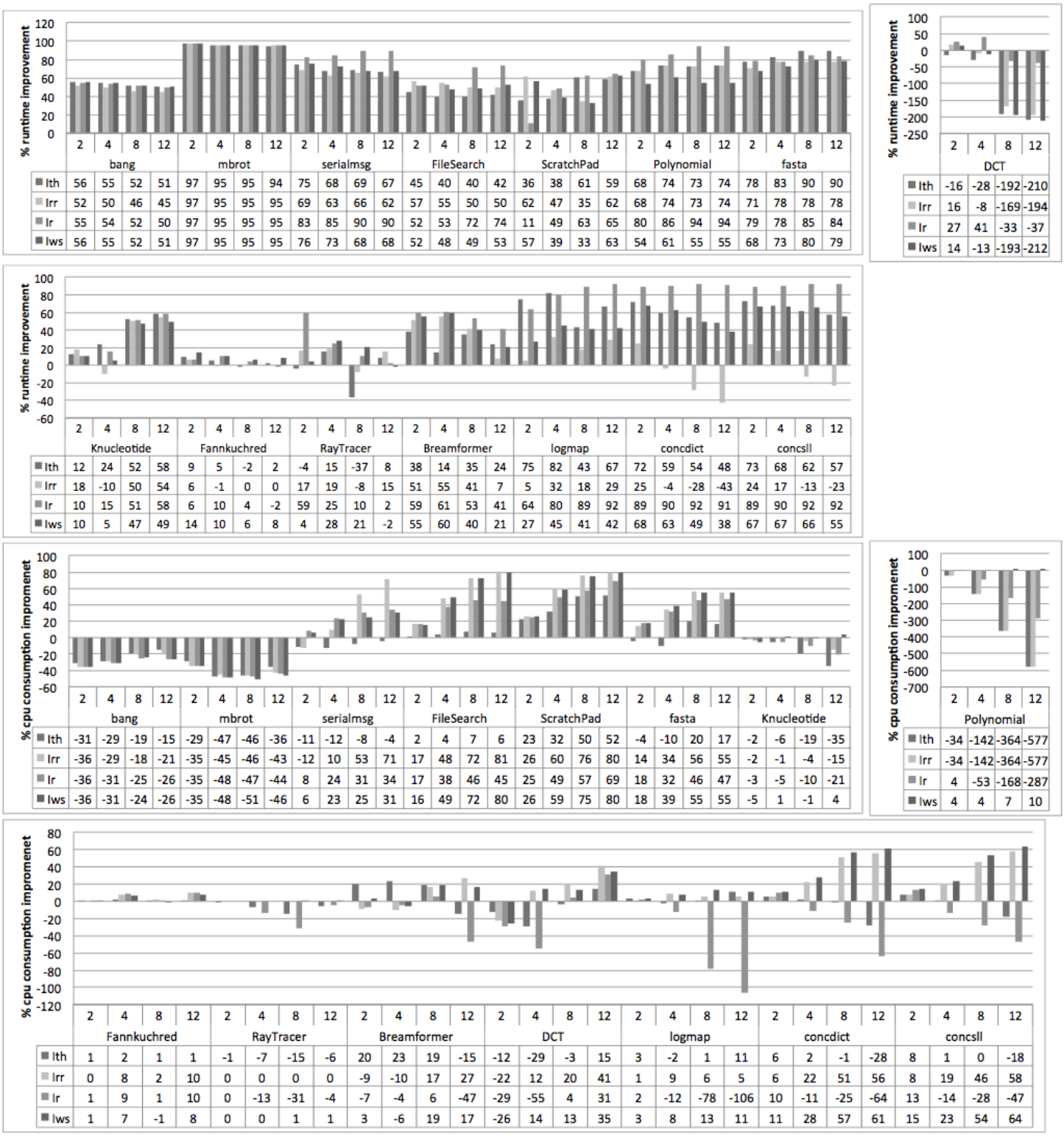


Figure 10. First two rows (3 charts) show % runtime improvement of our mapping technique over default mappings, next two rows (3 charts) show % cpu consumption improvement over default mappings for fifteen benchmarks. For each benchmark there are four core settings (2, 4, 8, 12-cores) and for each core 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.

Overall, our technique showed 40.56%, 30.71%, 59.50%, and 40.03% improvements over *thread*, *round-robin*, *random* and *work-stealing* policies respectively (Range: 30% to 60%). We also saw -21.48%, 4.78%, -12.30%, 14.58% changes in cpu consumption (Range: -21% to 15%).

3.3.2 Analysis

For understanding the factors that contributed to the improvements in program *runtime* and *CPU consumption* we profiled the program execution (using *perf*) and collected following data: number of context switches (both voluntary and involuntary), overall cache-misses (cache load/store requests that could not be served by any level cache), L1-dcache-load-misses and LLC-load-misses. Measuring context-switches helps us quantify the mailbox contentions as a result of capsules to threads mapping, and measuring *#cache-miss*, *#L1-miss* and *#LLC-miss* helps us quantify the cache locality of threads processing capsules. We now describe major categories in our results.

1) Reduced mailbox contentions. For Panini benchmarks bang, mbrot, BeamFormer and Polynomial, improvement in performance is due to the reduction in mailbox contentions between threads processing various capsules in the program.

A capsule while sending a message contends to acquire access to the recipient's mailbox. The access not be available right away due to other capsules holding the access (performing message send) or the thread of the recipient is fetching the message from the mailbox for processing. This scenario leads to mailbox contentions. In general, this scenario can be observed when a group of capsules is trying to send a large number of messages to a smaller group of capsules. This pattern is known as hub and affinity groups.

By assigning task policy to hub capsules and monitor policy to affinity capsules, the thread processing the hub capsule will also process the affinity capsule without resulting in mailbox contentions. Our results show that it is an effective strategy.

The general pattern described above can be explained using the bang program, where a large number of mailbox contentions occur between 440 instances of Sender capsule and a single instance of Receiver capsule. Here, Sender capsules form hub and Receiver capsule form affinity. By applying Hub and Affinity heuristics, our technique assigned monitor (*M*) policy to Receiver capsule and task (*M*) policy to Sender, which eliminated mailbox contentions. The reduction in the mailbox contentions is captured in the profile data (reduction in number of context-switches) shown in Figure 11.

2) Reduced message passing and processing overheads. For Panini benchmarks FileSearch, concdict and concsll, the improvement in performance is due to reduced message passing overheads.

The message passing overhead for a capsule constitutes, message preparation time, time to acquire access to the recipient mailbox and time to process the result received (if intended). The message processing overhead for a capsule constitutes, time to acquire access to its mailbox, time to fetch a message from the mailbox and identify the correct message handler to be executed.

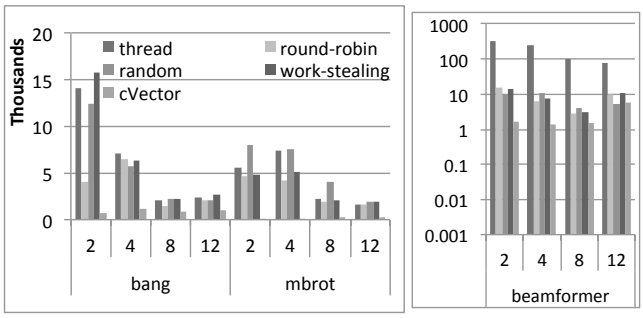


Figure 11. Shows reduction in #context-switches in our technique when compared to default policies for bang, mbrot and BeamFormer benchmarks (last bar represents our technique). Lower bars are better.

The general pattern that leads to overheads can be described as follows: i) capsules that have low computational workload and mainly spends time sending messages to other capsules face message passing overhead problem. ii) message processing overhead becomes substantial for capsules that have procedures with low computations.

By assigning task policy (Hub heuristics, Master heuristics), a thread is shared between such capsules and it helps to increase the computational workload and reduce the message passing overhead. To reduce the message processing overhead, monitor (LowCPU heuristics) policy can be assigned. By assigning monitor policy, the capsule that sends message itself will process the message at the recipient right away. Our results show that these two decisions can help reduce message passing and processing overheads.

For instance, consider concdict Panini program that simulates concurrent dictionary access. Capsules in this program are Master, Worker (20 instances) and Dictionary. The concurrently running Worker capsules performs 180152 dictionary reads and 19848 dictionary writes. Our technique assigns thread (*Th*) policy to Master (Master, waits for the results from workers), task (*Ta*) policy to Worker (Worker heuristics) and monitor (*M*) policy to Dictionary (LowCPU heuristics). By assigning monitor policy to Dictionary and task policy to Worker capsules, Worker capsules process 200000 messages themselves. This reduces the message passing overheads at the Worker capsules and message processing overhead at Dictionary capsule substantially (when compared to assigning *Th* or *Ta* policy to Dictionary).

3) Reduced mailbox contentions and cache-misses. For Panini benchmarks serialmsg, ScratchPad, fasta, the improvement in performance is due to reduction in mailbox contentions and improved cache locality.

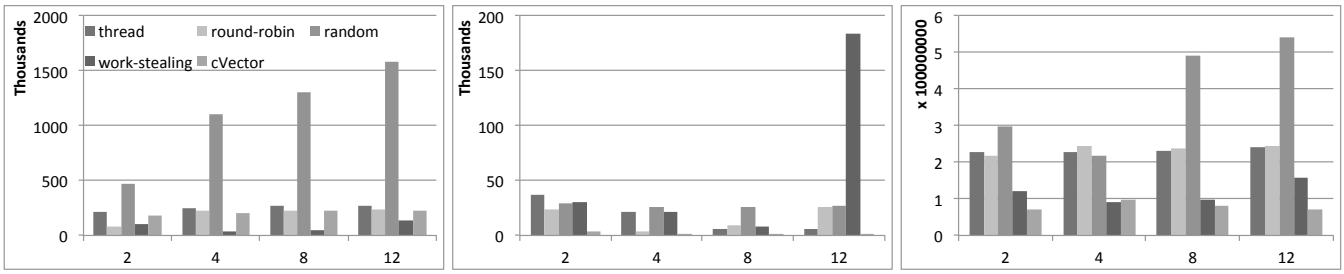


Figure 12. Shows reduction in *voluntary context-switches*, *involuntary context-switches* and *#cache-miss* for serialmsg benchmark in our cVector based mapping when compared to default mappings. Lower bars are better.

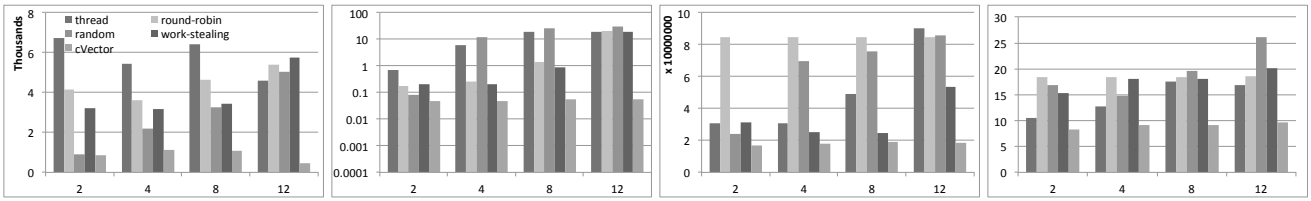


Figure 13. Shows reduction in *voluntary context-switches*, *involuntary context-switches*, *#cache-miss* and *#LLC-miss* for ScratchPad benchmark in our cVector based mapping when compared to default mappings. Lower bars are better.

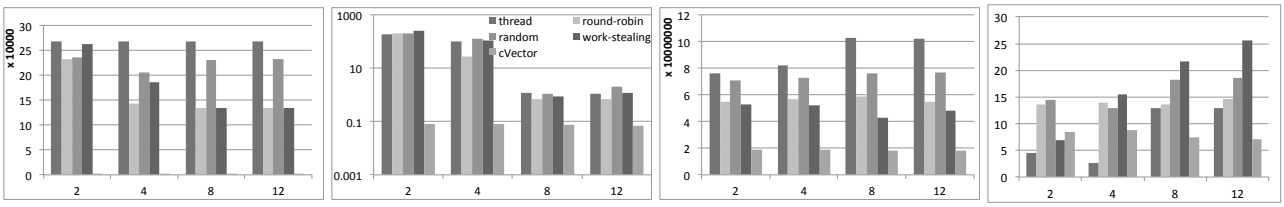


Figure 14. Shows reduction in *voluntary context-switches*, *involuntary context-switches*, *#cache-miss* and *#LLC-miss* for fasta benchmark in our cVector based mapping when compared to default mappings. Lower bars are better.

Cache misses happens when a capsule receives large number messages, processing of messages requires access/update of capsule's local state (that is variable) and different threads process different messages of a capsule. In such a case, the data locality per thread is decreased.

A dedicated thread can be assigned to process the messages of such capsules. Our results show that this does improve cache locality. Depending on the outgoing communication pattern either thread (Heavy heuristics) or task (High-CPU heuristics, Worker heuristics) policy can be assigned. We now provide several examples that demonstrate this.

serialmsg. BenchErl serialmsg is about message proxying through a dispatcher. The benchmark spawns 120 instances of Receiver capsule, one Dispatcher capsule, and 120 instances of Generator capsule. The dispatcher forwards the messages that it receives from generators to the appropriate receiver. Each generator sends a large number of messages to a specific receiver and hence causing a large number of mailbox contentions. Our technique assigned task (*Ta*) policy to Generator and monitor (*M*) policy to Dispatcher and Receiver capsules. This allows binding of every Generator instance to its respective Receiver instance and hence reducing the mailbox contentions largely. Also binding Generator capsules to their respective Receiver improved cache-locality. Figure 12 shows reduction in #context-switches and *#cache-miss* which supports our hypothesis.

ScratchPad. This Panini program computes line of count for files in the input directory. The objective of this program is to achieve high messaging throughput. Capsules in this program are FSWalker, LocAnalyser, LocCounter (20 instances), Accumulator (10 instances) and ResultAcc. Capsule FSWalker browses through files in the input directory and sends file names to LocAnalyser capsule which then picks a LocCounter at random and asks to perform line counting. Accumulator and ResultAcc collects results from LocCounter capsules. For achieving good messaging throughput, FSWalker and LocAnalyser must have good latency (messages are processed as soon as they are received) and Accumulator and ResultAcc must not become performance bottlenecks. This is achieved in our technique, which assigns thread (*Th*) policy to FSWalker and task (*Ta*) policy to LocAnalyser. This makes capsules FSWalker and LocAnalyser process messages in uninterrupted manner (as FSWalker 's dedicated thread can send messages to the taskpool

thread of the LocAnalyser). Assigning monitor (M) policy to Accumulator and ResultAcc reduces their communication overheads with LocCounter capsules. Figure 13 shows reduction in #context-switches, *#cache-miss* and *#LLC-miss* which supports our hypothesis.

fasta. This Panini program generates and writes random DNA sequences and exhibits task parallelism. Capsules in this program are RandomFasta (2 instances), RepeatFasta, FloatProbFreq (3 instances) and Writer. Both RandomFasta and RepeatFasta capsules independently generate sequence with the help of their respective FloatProbFreq capsules and send messages to Writer capsule for printing. Here, the communication of RandomFasta and RepeatFasta capsules with their respective FloatProbFreq capsules is too large. Our technique assigns sequential (S) policy to FloatProbFreq capsules to reduce this overhead. Figure 14 shows reduction in #context-switches and *#cache-miss* which supports our hypothesis.

Our technique achieved small improvements for three programs (RayTracer, Fannkuchredux, and DCT). These programs are mainly data-parallel applications with embarrassingly parallel behavior. The results support our earlier intuition that for embarrassingly parallel applications, it is easy to determine abstractions to threads mappings, as abstractions independently perform their tasks. For instance, in RayTracer program Runner acts as master that distributes the work to a set of RayTracer worker capsules. RayTracer worker capsules perform independent computations. For this program, mapping is intuitive. Runner could be assigned thread execution policy and each RayTracer worker could be assigned task execution policy. Hence, it is easy to map capsules to threads and there is very little opportunity for further improving the mapping.

3.4 Applying cVector to Akka MPC Framework

In this section we evaluate applicability of our technique for the Akka framework [9]. Akka is a widely used industrial strength actor framework for building large scale distributed applications. This evaluation helps to demonstrate that our technique is applicable to other JVM-based MPC frameworks.

3.4.1 Methodology

We have ported eight of fifteen benchmark programs to Akka framework for which our technique performed better than default policies in Panini (this excludes data-parallel programs for which our technique performs on par with default policies). The default policies in Akka are: i) thread-all (or pinned-dispatcher), ii) task-all (or thread-pool-executor) and iii) fork-join (fork-join-executor). In thread-all policy, every actor is mapped to a JVM thread using pinned-dispatcher. In task-all policy, all actors are mapped to a thread pool that is run by #cores threads, and in fork-join policy, all actors are mapped to a fork-join pool containing #cores threads. We map the cVector execution policies using the methods that are already implemented in Akka as follows. For cVector thread policy we use pinned-dispatcher in Akka, for task policy, we use Akka's thread-pool-executor and for sequential and monitor policies, we use Akka's *CallingThreadDispatcher*. A detailed description of the policies in Akka can be found in [2]. We measure program execution time as described in our previous evaluations and compute the percentage improvements of our technique over thread-all (I_{th}), task-all (I_{ta}) and fork-join (I_{fj}) default policies of Akka.

3.4.2 Results and Analysis

Figure 15 shows the performance improvements of our technique over default policies in terms of program execution time. Our technique achieves on average 30.87%, 27.41% and 31.39% improvements over thread-all, task-all and fork-join policies. The performance improvements are consistent with the performance improvements seen in Panini Capsules. A further analysis using the framework presented earlier (profiling and collecting context-switches, cache-miss etc.) suggests that the performance improvements happen due to the ability of our technique to reduce the mailbox contention, the message passing overhead and improve cache locality. Overall, our technique performed better than default policies that are used in Akka for the benchmarks that are evaluated.

3.5 cVector Accuracy

As cVector is computed statically, sometimes it may misclassify the actual runtime behavior. In this section we measure the misclassification of cVector fields for our corpus, discuss the reason for misclassification and analyse the impact of misclassification on performance.

We profile the programs using Java Interactive Profiler (JIP) [5] and collect the data representing the actual runtime behaviors of capsules such as, number of externally blocking calls, type of capsule state, number of messages sent and received by the capsule, type of message sends (synchronous, asynchronous), time spent on actual computation and time spent waiting for messages. We use this data along with manual inspection of the source code to compute actual runtime behavior of the capsule in terms of cVector fields. We then compare two cVectors to determine misclassification. The details about computing runtime behavior cVector using profile data can be found in §C.

For the corpus of 15 benchmarks that contains 54 capsules, we found no misclassification of cVector fields β, σ, $\overrightarrow{\rho}$, two misclassification instances of π, one misclassification instance $\overleftarrow{\rho}$ and five misclassification instances of ω. These results were expected as the analysis of determining β, σ and $\overrightarrow{\rho}$ does a sound approximation of the behavior β, σ and $\overrightarrow{\rho}$. That is, the existence of blocking calls, type (fixed size or variable size) of state and capsule's outgoing communications (message sends) can be accurately found from the code of the capsule. Whereas, determining capsule's incoming communication pattern without considering global infor-

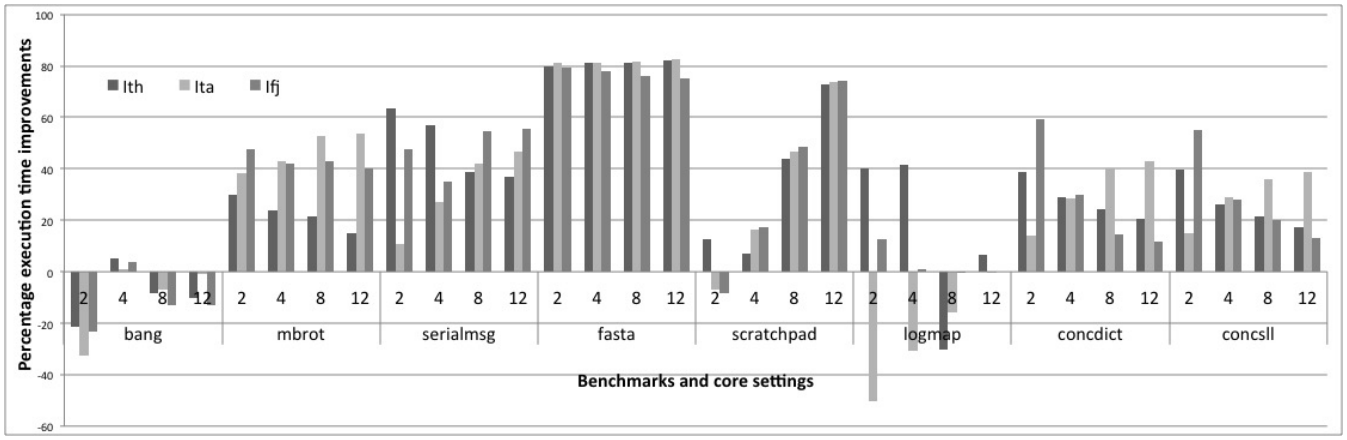


Figure 15. Comparing performance improvements of our technique over thread-all, task-all and fork-join default policies of Akka. For each benchmark there are three bars that shows improvements over three default policies on 2, 4, 8 and 12 core setting. X-axis: Benchmark programs and core setting, Y-axis: Percentage improvements in terms of program execution time.

mation (the topology) could be inaccurate. Also the computations performed by the capsule may be input dependent (dynamic in nature). These intuitions are supported by the misclassification instances of $\overleftarrow{\rho}$ and ω. The details about misclassification instances can be found in §C.

None of the misclassifications had effect on capsules to threads policies (or the performance). For instance, the misclassification instance of π (where the value should have been *sync* and it is assigned *future*) has no effect on capsules to threads mapping because both *sync* and *future* led to same capsules to threads mapping. This happens because cVector fields have different predictive powers. The cVector fields β and $\overrightarrow{\rho}$ are most influential (as seen in our execution policy selection Figure 7, every path goes through them) and any misclassifications of these can be unforgiving. The misclassification of less influential fields may or may not change the mapping.

3.6 Threats to Validity

A threat to validity of our evaluation is that we may not be able to extrapolate our findings to programs that have completely different characteristics compared to our benchmarks. To mitigate this threat, we have selected a wide variety of benchmarks from varied sources. Figure 9 describes the characteristics of our benchmarks in detail.

Second threat to validity of our evaluation is that it is Panini centric (because, i] default policies are implemented in Panini, and ii] source programs are translated to Panini). To mitigate this threat, we have applied our technique to Akka MPC framework and demonstrated the applicability.

Third threat to validity of our evaluation is that it compares our technique that use flexible policies (abstractions can be assigned different execution policies) against default policies that use single mapping for everything (abstractions are assigned single execution policy). There exists no automatic technique that assigns flexible policies like ours. In the current state of the art, programmers use default policies (or default scheduler/dispatcher). They customize the policies when the performance is poor (which may involve mixing default policies). We agree that, a comparison against the best manually tuned program would help to strengthen the contribution of our cVector based mapping. However, we could not find such a manually tuned program in the benchmark suites that we have used.

Fourth threat to validity of our evaluation is that we have used a multicore (6+6 core) for evaluation, which may not have the same platform characteristics as a distributed cluster. However, we believe that our results would still be applicable, e.g. for a number of cases we reduce the message passing overhead, which would be specially significant for distributed cluster.

Finally, we have compared only with one candidate from round-robin, random, and work-stealing algorithms, but our selection is a representative from each class of these scheduling algorithms. Future work can explore other variations.

4. Related Work

Frameworks such as Akka [9], Kilim [32], Scala Actors [15], Jetlang [26], ActorFoundry [7], SALSA [10] and Actors Guild [19] allow programmers to map their actors to JVM threads and fine tune their application using schedulers and dispatchers. The default mappings evaluated in this paper represents these schedulers and dispatchers. Akka provide four kinds of dispatchers: default, pinned, balancing, and calling thread. The default dispatcher is used if programmer does not specify (this is similar to our random mapping strategy). In Kilim, actors are runnable tasks which are assigned to a thread-pool and the scheduling policy is round-robin. Scala Actors allow creation of thread-based and event-based (uses task-pool and round-robin scheduling policy) actors. SALSA allows creation of heavy-weight and light-weight actors using Stage and by default maps actors to a set of

stages (can be considered as thread) using round-robin policy. Likewise, other actor frameworks use default actors to threads mapping or programmer specified mappings (using schedulers/dispatchers). When compared to these works, our technique automatically assigns capsules to threads.

Several works on performance improvement of non-JVM MPC frameworks exists. Francesquini *et al.* [11] proposes a technique implemented in Erlang [37] runtime that places Erlang actors on multi-core efficiently. Their technique showed that by placing frequently communicating actors (hub-and-affinity) together, over two times improvement in the application performance can be achieved. However, programmers need to identify hub and its affinity actors and annotate the program for runtime to perform the desired mapping. Our technique uses many more characteristics along with hub-and-affinity and performs essential program analyses to automatically determine the mappings.

Mapping application on to multi-core is a well studied problem. The application is represented as task graph and the mapping problem is defined as how to map different tasks to CPU cores to minimize application runtime. A recent survey [30] lists different static, dynamic and hybrid techniques that map task graph to multi-core with performance, energy consumption and temperature as different goals of determining optimal mapping. Researchers have explored the problem of mapping application tasks that communicate via both message passing and shared memory on homogeneous and heterogeneous cores [14, 17, 27, 28]. These techniques are not directly applicable to JVM-based MPC frameworks, because threads to cores mapping is left to OS scheduler and only MPC abstraction to threads mapping can be optimized. However, abstraction to threads mapping technique can utilize solutions proposed for general task graph mapping problem. In our cVector based mapping technique, we utilize characteristics and interaction behaviors similar to task characteristics and task graph in general task graph mapping problem.

Note that the mapping problem in MPC programs is different from the mapping problem in general multi-threaded programs. In multi-threaded programs, the mapping problem is defined as scheduling and load-balancing of threads on multi-cores. This also involves binding of threads to physical cores. However in MPC programs, the mapping problem is two-fold: mapping MPC abstractions to threads and scheduling of threads on multi-cores. Tousimojarad and Vanderbauwhede [35] propose efficient strategies for mapping threads to cores for OpenMP multi-threaded programs. When compared to this work, our technique maps capsules to threads and not threads to cores. Threads to cores mapping is handled by OS scheduler in JVM-based MPC frameworks.

5. Conclusion

Performance optimization is one of the leading reasons for breaking abstraction boundaries. In this work we targeted this problem for message-passing abstractions on JVM, where performance concerns may lead to deformed designs. We proposed a technique to automatically map such abstractions to JVM threads using capsules as a specific use case. We assign execution policies to capsules to achieve capsules to threads mapping. Our mapping technique utilizes a set of properties about capsules and their communications to select execution policies. We have evaluated our mapping technique against four commonly used default mapping techniques in JVM-based MPC frameworks. We have evaluated on a wide-variety of MPC benchmarks that are both concurrent and parallel applications exhibiting different concurrency patterns and parallelisms (data, task, pipeline). Our results show 30%-60% improvement in program execution times when compared to default mappings. We have also applied on technique to Akka, where for the studied benchmarks we observed similar performance improvements. Our mapping technique does not require any changes to the design of the application; it mainly defines how MPC abstractions will be compiled, preserving design while improving performance.

6. Acknowledgments

This work was supported in part by the NSF under grants CCF-08-46059, CCF-11-17937, and CCF-14-23370. This work significantly benefited from constructive comments and suggestions by the OOPSLA 2015 reviewers, the AGERE! 2014 reviewers, audience of the AGERE! 2014 workshop, Walter Binder, and Steven M. Kautz.

References

[1] Panini Programming Language. http://paninij.org/.

[2] Dispatchers in Akka. http://doc.akka.io/docs/akka/snapshot/scala/dispatchers.html.

[3] Tuning Dispatchers: Discussion threads. http://tinyurl.com/o6utwkp,http://tinyurl.com/ovypwnu,http://tinyurl.com/ponekm6,http://tinyurl.com/o7xzh8s,http://tinyurl.com/ohwsesn.

[4] Evaluation Artifacts: cVector-based mapping. https://github.com/oopsla2015/strategy.

[5] The Java Interactive Profiler. http://jiprof.sourceforge.net/.

[6] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A scalability benchmark suite for Erlang/OTP. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang*, pages 33–42, 2012.

[7] M. Astley. The Actor Foundry: A Java-based Actor Programming Environment. In *Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-99*.

[8] M. Bagherzadeh and H. Rajan. Panini: A Concurrent Programming Model for Solving Pervasive and Oblivious Interference. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 93–108. ACM. URL http://doi.acm.org/10.1145/2724525.2724568.

[9] J. Boner. Akka, TypeSafe Inc. http://akka.io.

[10] T. Desell and C. A. Varela. SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency. In *Concurrent Objects and Beyond, Lecture Notes in Computer Science, 2014*.

[11] E. Francesquini, A. Goldman, and J.-F. Méhaut. Actor scheduling for multicore hierarchical memory platforms. In *Proceedings of the twelth ACM SIGPLAN workshop on Erlang*, pages 51–62, 2013.

[12] B. Fulgham and I. Gouy. Computer language benchmarks game. http://benchmarksgame.alioth.debian.org/.

[13] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76. ACM, 2007. URL http://doi.acm.org/10.1145/1297027.1297033.

[14] F. Guirado, A. Ripoll, C. Roig, and E. Luque. Performance prediction using an application-oriented mapping tool. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 184–191. IEEE, 2004.

[15] P. Haller and M. Odersky. Actors That Unify Threads and Events. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, pages 171–190, 2007. URL http://dl.acm.org/citation.cfm?id=1764606.1764620.

[16] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[17] A. Heydarnoori and W. Binder. A graph-based approach for deploying component-based applications into channel-based distributed environments. *Journal of Software*, 6(8):1381–1394, 2011.

[18] S. M. Imam and V. Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 67–80. ACM, 2014. URL http://doi.acm.org/10.1145/2687357.2687368.

[19] T. Jansen. Actors guild. https://code.google.com/p/actorsguildframework/.

[20] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3): 381–404, 1983.

[21] R. M. May. Simple mathematical models with very complicated dynamics. In *The Theory of Chaotic Attractors*, pages 85–93. Springer, 2004.

[22] O. M. Nierstrasz. Active Objects in Hybrid. In *Proceedings of the International Conference on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 243–253. ACM. URL http://doi.acm.org/10.1145/38765.38829.

[23] J.-N. Quintin and F. Wagner. Hierarchical Work-stealing. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, pages 217–229. Springer-Verlag, 2010. URL http://dl.acm.org/citation.cfm?id=1887695.1887719.

[24] H. Rajan. Capsule-oriented programming. In *Proceedings of the 37th International Conference on Software Engineering: NIER Track*, May 2015.

[25] H. Rajan, S. M. Kautz, E. Lin, S. L. Mooney, Y. Long, and G. Upadhyaya. Capsule-oriented Programming in the Panini Language. Technical Report 14-08, 2014.

[26] M. Rettig. Jetlang. http://code.google.com/p/jetlang/.

[27] C. Roig, A. Ripoll, M. A. Senar, F. Guirado, and E. Luque. Modelling Message-passing Programs for Static Mapping. In *Proceedings of the 8th Euromicro Conference on Parallel and Distributed Processing*, EURO-PDP'00, pages 229–236, 1999. URL http://dl.acm.org/citation.cfm?id=1897179.1897214.

[28] C. Roig, A. Ripoll, and F. Guirado. A New Task Graph Model for Mapping Message Passing Applications. *IEEE Trans. Parallel Distrib. Syst.*, 18(12):1740–1753, 2007. URL http://dx.doi.org/10.1109/TPDS.2007.1117.

[29] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-based Manycore Partitioning. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 107–116. ACM, 2012. URL http://doi.acm.org/10.1145/2370816.2370833.

[30] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 1:1–1:10. ACM, 2013. URL http://doi.acm.org/10.1145/2463209.2488734.

[31] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 8–8. ACM, 2001. URL http://doi.acm.org/10.1145/582034.582042.

[32] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 104–128. Springer-Verlag, 2008. URL http://dx.doi.org/10.1007/978-3-540-70592-5_6.

[33] S. Tasharofi and R. Johnson. Actor Collection. http://actor-applications.cs.illinois.edu/.

[34] W. Thies and S. Amarasinghe. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 365–376. ACM, 2010. URL http://doi.acm.org/10.1145/1854273.1854319.

[35] A. Tousimojarad and W. Vanderbauwhede. An Efficient Thread Mapping Strategy for Multiprogramming on Manycore Processors. *Journal of Parallel Computing, 2014*.

[36] G. Upadhyaya and H. Rajan. An Automatic Actors to Threads Mapping Technique for JVM-based Actor Frameworks. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 29–41. ACM, 2014. URL http://doi.acm.org/10.1145/2687357.2687367.

[37] J. Zhang. Characterizing the scalability of erlang vm on many-core processors. Master's thesis, KTH, 2011.

A. Capsules: an MPC Abstraction

A capsule is an MPC abstraction implemented in the programming language Panini [1, 8, 24, 25]. Figure 16 presents an example HelloWorld program in this language. In this program there are three capsules HelloWorld, Greeter and Console and they are connected as *HelloWorld* → *Greeter* → *Console*.

```
1  signature Stream { //A signature declaration
2    void write(String s);
3  }

5  capsule Console () implements Stream { //Capsule declaration
6    void write(String s) { // Capsule procedure
7      System.out.println(s);
8    }
9  }

11 capsule Greeter (Stream s) { //Requires an instance of Stream
12   String message = "Hello World!"; // State declaration
13   void greet(){                    // Capsule procedure
14     s.write("Panini: " + message); // Inter−capsule procedure call
15     long time = System.currentTimeMillis();
16     s.write("Time is now: " + time);
17   }
18 }

20 capsule HelloWorld() {
21   design {          // Design declaration
22     Console c; // Capsule instance declaration
23     Greeter g; // Another capsule instance declaration
24     g(c);      // Wiring, connecting capsule instance g to c
25   }
26   void run() { // An autonomous procedure
27     g.greet(); // Inter−capsule procedure call
28   }
29 }
```

Figure 16. HelloWorld Program in Panini

In Panini's programming model, capsules are independently acting entities. Capsules provide interfaces to communicate to other capsules via capsule procedures. When a capsule wants to communicate with other capsules it does so using inter-capsule procedure calls. In the *HelloWorld* program described above, *g.greet()* in line 27 is an inter-capsule procedure call between *HelloWorld* capsule and *Greeter* capsule. If a capsule requires return result of inter-capsular call then the caller receives a future as a proxy for the actual return value (void return values are allowed). If the value is not used immediately, the caller can continue execution.

Capsules internally use message-passing based concurrency mechanism to process inter-capsule procedure calls. A capsule contains a message queue for receiving messages, a thread for processing messages, a set of state variables that represents its local state and a message processing logic containing set of message handlers (mapped to capsule procedures). Capsules cannot share data, multiple threads cannot process capsule's messages simultaneously, and capsules can have finite number of nested capsules. In case of capsules that are assigned dedicated threads for processing their messages, the message queue blocks when empty. In all other cases (when capsules are assigned task, sequential or monitor policy), the message queue does not block the executing thread when empty.

B. cVector+

A higher level communication pattern that cVector+ optimizes is many-to-one communication and it can be described as follows. In a many-to-one communication, there are a number of broadcasting capsules that performs small computations in stages and after each stage sends a partial result to an aggregator capsule (aggregator capsule collects partial results from many other capsules). In such scenarios, the overall communication (between broadcasting capsules and aggregator capsule) overtakes the computations performed by the broadcasting capsules. So, one can define this scenarios as many-to-one communication where communication overtakes computation. In such scenarios, spawning fewer threads to process broadcasting capsules hugely reduces the communication overheads (due to both message processing logic and mailbox contention).

Analysis. cVector+ analysis is performed using capsule communication graph (CCG) and execution policies of capsules. A CCG is a directed graph G(V,E) where, $V = A_0, A_1, ..., A_n$ is a set of nodes, each representing a capsule, and E is a set of edges (A_i, A_j) for all i,j such that there is communication from A_i to A_j. An example CCG is shown in Figure 1. This specific case can be described as follows: there exists a capsule with monitor (M) execution policy and it communicates with a set of parent capsules (capsules that send message to this capsule) that have task (Ta) execution policies, these parent capsules have $\omega = io$, $\overrightarrow{\rho} = router$ and $\overleftarrow{\rho} = request\text{-}reply$. In such a case, the parent capsules will be part of a taskpool that is served by a set of threads (size = #cores). The solution we propose is to cut-down the size of the taskpool by half (size = #cores/2). This will improve the program runtime due to reduced number of lock contentions between threads that are processing parent capsules when trying to communicate with a capsule that is assigned monitor execution policy. It also helps to reduce the CPU consumption of the program by increasing the workload on threads (more tasks per thread). In our benchmark suite, we have improved both program runtime and cpu consumptions for six Panini programs that exhibits this special case using cVector+ mapping. One such Panini program is bang. This program contains a number of *Sender* capsules that are assigned task execution policy and they all communicates with a single *Receiver* capsule which has monitor policy assigned to it.

B.1 Evaluation

We see a number of benchmark programs that could benefit from our enhanced cVector (cVector+) mapping strategies

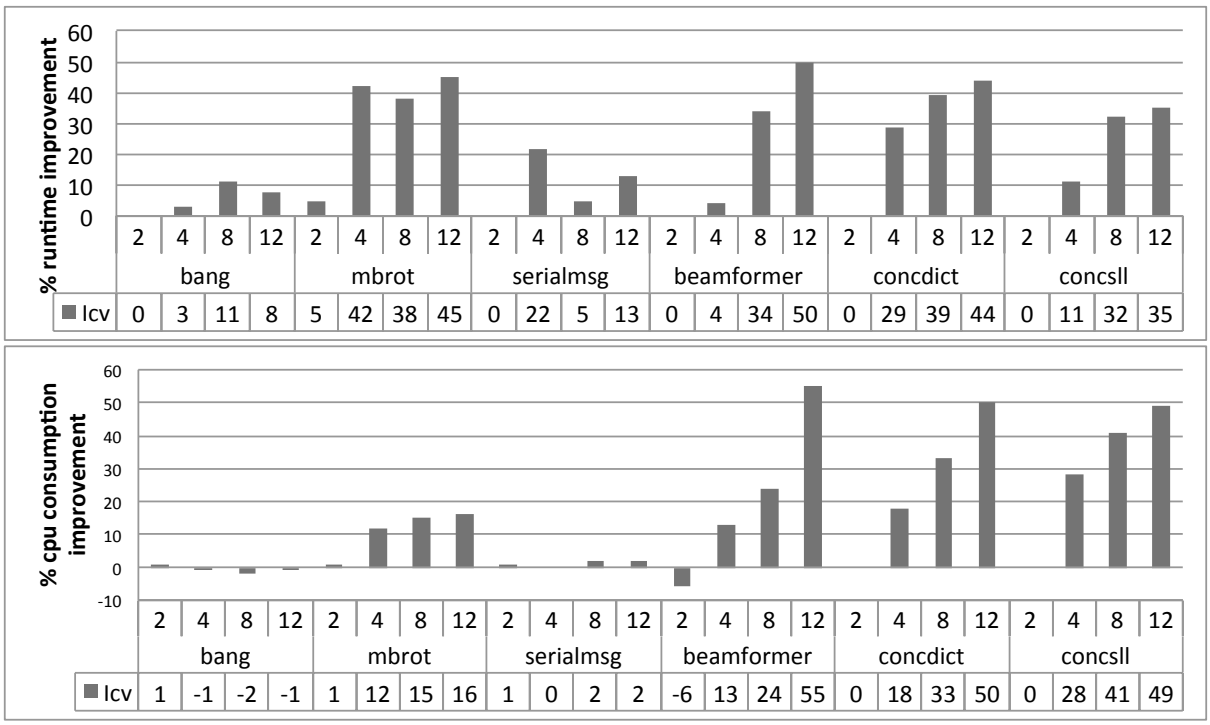


Figure 17. Improvements in program runtime and cpu consumptions over cVector based mapping for six programs measured on 2, 4, 8 and 12 core settings (higher is better). Overall improvements of cVector+ over cVector is shown in Figure 18

Metric	cVector				cVector+			
	I_{th}	I_{rr}	I_r	I_{ws}	I_{th}	I_{rr}	I_r	I_{ws}
runtime	40.56	30.71	59.50	40.03	43.51	36.26	60.93	43.00
cpu consumption	-21.48	-4.78	-12.30	14.58	-14.93	-0.65	-4.42	18.61

Figure 18. Compares average improvements in program runtime and cpu consumptions for cVector and enhanced cVector (cVector+) mappings.

(described in §B). The programs are *bang*, *mbrot*, *serialmsg*, *beamformer*, *concdict* and *concsll*.

Methodology. For these programs, we re-run the experiments with our cVector based mapping but reduced the size of the taskpool to half (initial size of the taskpool was #cores, we reduced it to #cores/2).

Results. Figure 18 compares the improvements of cVector+ against cVector mapping strategies and Figure 17 provides details about the improvements for each of the six programs. Figure 18 shows that, with proposed enhancement to our cVector based mapping, we are able to further reduce program *runtime* for six programs and a substantial improvements can be seen with respect to program *CPU consumptions*. This happens due to reduced context-switches, reduced cache-misses and reduction in cpu consumption as less number of threads are operating. The results supports the fact that reducing the number of threads (whenever necessary) reduces program runtime and cpu consumptions, however determining which programs can benefit is the key. We determine this using abstractions cVectors, execution policies and topology.

C. cVector Accuracy

Methodology. We profile the programs using Java Interactive Profiler (JIP) [5] and collect the data representing the actual runtime behaviors of capsules such as, number of externally blocking calls (#blk), type of capsule state, number of messages sent and received by the capsule (#sent and #recv), type of message sends (synchronous, asynchronous) (#sync, #async), time spent on actual computation (%work) and time spent waiting for messages (%wait). We use this data along with manual inspection of the source code to compute actual runtime behavior of the capsule in terms of cVector fields. The capsule runtime behavior is also represented using six fields: <β',σ',π',$\overrightarrow{\rho}$',$\overleftarrow{\rho}$',ω'>.

- β' is assigned **true** if #blk > 0, **false** otherwise

- σ' is assigned nil if capsule has no state, fixed if capsule message handlers does not increase the size of the state and variable otherwise
- π' is assigned sync if #sync > #async and async otherwise
- $\overrightarrow{\rho}$' is assigned leaf if capsule does not communicate with other capsules, scatter if #sent > #recv and router otherwise
- $\overleftarrow{\rho}$' is assigned gather if #recv > #sent and request−reply otherwise
- ω' is assigned math if work > wait and io otherwise

We then compare cVector <β,σ,π,$\overrightarrow{\rho}$,$\overleftarrow{\rho}$,ω> with actual runtime behavior vector <β',σ',π',$\overrightarrow{\rho}$',$\overleftarrow{\rho}$',ω'> to determine misclassification.

Results and Analysis. For the corpus of 15 benchmarks that contain 54 capsules, we found no misclassification for cVector field β. Eight capsules had β *true* and the actual runtime behavior β' from profile result showed the presence of externally blocking calls in these capsules. We also found no misclassification of cVector field σ. This result is expected as our state analysis assigns *variable* for states that are of collection types and only the size of collection types can be increased. There were ten capsules with state *variable* and the profile results showed instances of message handlers increasing size of the capsule state. There were two instances of π misclassification (2 of 54 = 3.7% misclassification) and both were due to the limitation of our inherent parallelism analysis. The misclassification case can be described using the code snippet shown below:

```
List<Double> resObjs = new ArrayList<Double>();
for (int i=0; i<subintervals; i++) {
  double res = computeActors[i].areaUnderTheCurve(...)
  // store the result Future into a collection
  resObjs.add(res);
}
for (Double result : resObjs) {
  // retrieve Future objects and claim their value
  areaSum += result.doubleValue();
}
```

Our analysis assigns π a value *future*, whereas actual runtime behavior is synchronous. The cVector field $\overrightarrow{\rho}$ had no misclassification, where there was one instance of $\overleftarrow{\rho}$ misclassification. This misclassification can be described as follows:

```
capsule Master(...) {
  void begin(StartMessage sm) {
    while (i < numTerms) {
      for (SeriesWorker worker : workers) {
        worker.nextTerm();
      }
    }
  }
  void process(ResultMessage rm) {
    termsSum += rm.term;
    numWorkReceived += 1;
    if (numWorkRequested == numWorkReceived) {
      System.out.println("Terms sum: " + termsSum);
    }
  }
}
```

Our analysis assigns $\overleftarrow{\rho}$ *gather* because $\overleftarrow{\rho}_{begin}$ is *request-reply* and $\overleftarrow{\rho}_{process}$ is *gather* so $\overleftarrow{\rho}$ for capsule *Master* will be *gather*. The profile results indicates that *Master* capsule receives less number of messages. Finally, for cVector field ω we found five instances of misclassifications. All of them are of same type where the capsule is assigned ω *math* and the profile result indicate that the capsule mostly spends it time waiting for messages. We feel these are genuine misclassifications where our computational workload analysis found behaviors that leads to cpu intensive computations such as, recursion, loop with input dependent bound etc, while the profile results (actual runtime behavior) showed that the computations performed by these capsules were negligible.

To summarize, we found zero misclassification of cVector fields β, σ, $\overrightarrow{\rho}$, two misclassification instances of π, one misclassification instance $\overleftarrow{\rho}$ and five misclassification instances of ω.

Impact of Misclassification. The misclassification instance of π (where the value should have been *sync* and it is assigned *future*) has no effect on capsules to threads mapping because both *sync* and *future* led to same capsules to threads mapping. The misclassification instance of $\overleftarrow{\rho}$ changed the value from *gather* to *request-reply* which did not change the mapping. Similarly, all five instances of ω misclassifications changed ω value from *math* to *io* did not change the mapping. As discussed previously, cVector fields have different predictive powers. The cVector fields β and $\overrightarrow{\rho}$ are most influential (as seen in our mapping flow shown in Figure 7, every path goes through them) and any misclassifications of these can be unforgiving. The misclassification of less influential fields may or may not change the mapping.