Value Prediction for MPI Message-Passing Systems

Pantelis Panayiotou Yiannakis Sazeides Paraskevas Evripidou

Dept of Computer Science
University of Cyprus
CY-1678 Nicosia
Cyprus

Abstract

The predictability of various types of program information has been the subject of a plethora of work aimed to facilitate performance increases through latency reduction and speculation. Most past studies on prediction have been in the area of microarchitecture.

This paper attempts to characterize the predictability of data in messages exchanged in parallel applications that use MPI message-passing protocol. Value predictors are used, drawn from the area of microarchitecture. The prediction of data in messages can enhance the performance of message passing parallel programs through speculative execution. This hypothesis is based on the dominance of communication latency in the overall execution time of parallel programs.

An infrastructure based on code instrumentation was developed to investigate this predictability and results are presented for five benchmarks. The results show that predictability varies depending on the program and predictor. Some programs exhibit virtually no predictability whereas others 57–94%. Predictability is found at different levels: within a message, across messages of the same process and across messages in different processes. The most influential - both constructive and destructive - component of the information vector used to predict a value in a message buffer is the *offset* of the value in the buffer. The results indicate that data in MPI messages display mainly last-value and/or regularly-repeating behavior. The data suggest that most of the observed predictability can be captured with relatively small predictors.

1 Introduction

In the past High Performance Computing (HPC) was confined to large Processors, Supercomputers or Massively Parallel machines. Last decade's advances in microprocessor technology and high speed networks, have made the undertaking of HPC feasible on Networks of Workstations (NOW). Although NOWs could benefit from high speed networks such as Myrinet [1], their great majority rely on standard networks, mainly Ethernet switches. The network performance in such NOW configurations is a limiting factor [2]. The latest trend in high performance computing is Grid computing [3] and web-based metacomputers [4]. The communication latency for such widespread and diverse networks can incur a severe performance penalty.

It has been shown[5][6] that communication latency can dominate the execution of most parallel applications. This can be attributed to network bandwidth and latency, and to message-passing software overhead. Further delay can be encountered in the exchange of a message, if the receiving processor reaches the receive instruction before the sending processor reaches the corresponding send instruction (see Figure 1). This situation may get worse as the number of processors of the parallel computer increases.

There is a plethora of previous work that examines communication latency issues[7]. Prediction is a basic technique that can be employed to reduce communication latency but has received little attention[8, 9, 10]. Communication overheads can be reduced using speculative execution based on predicted information. This is a well known approach employed heavily at the processor microarchitecture level to eliminate stalls or to increase instruction level of parallelism[11, 12].

Speculative execution driven by **value prediction** for message passing, can work in the same way as at the microarchitecture level[11, 12]. The receiving process instead of waiting, uses the predicted values for speculative execution and thus execution can move faster. Some performance penalty may be incurred in the case of misprediction since corrective action may be needed. The potential of this approach can be significant due to the dominance of communication latency in many message–passing applications.

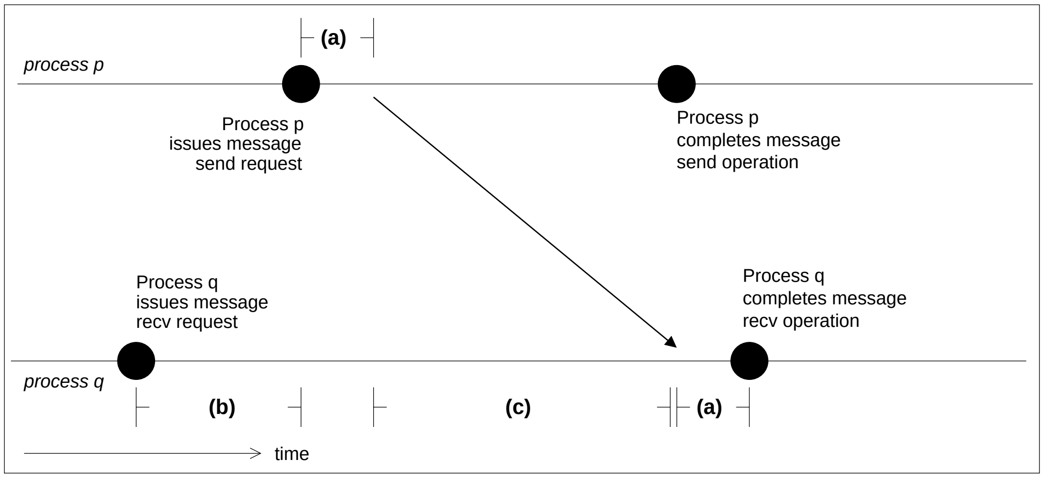
(a)
process p
Process p issues message send request
Process p completes message send operation
Process q issues message recv request
Process q completes message recv operation
process q
(b)
(c)
(a)
time

Figure 1: An example of communication delays because of (a) message-passing software overhead, (b) having the receiving process reach the recv instruction before the sending process reaches the send instruction, and (c) network latency and bandwidth. In this example, processes *p* and *q* exchange a message using point-to-point communication.

1.1 Motivation: How Value Prediction can drive speculative execution in Message-Passing Programs

Figure 2 illustrates how value prediction could be used to reduce the total execution time of an MPI application. Part (a) of Figure 2 shows a scenario where an MPI process *p* receives some data from several other processes, and subsequently makes several calculations on that data to produce a result. This data will arrive in the form of several MPI messages, and *p* will have to idle until all messages have arrived in its receive buffer. The total execution time for *p* can be expressed as:

$Ttotal = Tcomm + Tcomp$

Where $Tcomm$ is the time spent for communication, while *p* remains idle, and $Tcomp$ is the time *p* spends doing actual work (the *computation time*). Note that, as said in Section 1, $Tcomm$ can be longer than $Tcomp$ (*communication time* usually dominates the execution of parallel applications).

Prediction could be used to speed up the execution time of *p*, as shown in part (b) of Figure 2. Process *p* does not wait for the messages to arrive, but rather speculatively receives the data, using a value predictor. It then proceeds with the calculations using the predicted data. Subsequently, when it receives the actual messages and, in case of mis-speculation, it performs the necessary calculations only on the differences. The total execution time for *p* becomes:

$Tnew = Tpred + Tncomp + Tncomm + Tpcomp$

$Tpred$ is the time taken by the predictor to deliver the predicted data. $Tpred$ is expected to be short. $Tncomp$ is the time taken to perform the calculations on the predicted data (note that this should be equal to $Tcomp$). $Tncomm$ is the new *communication time*, where *p* idles waiting for the real data to arrive. This is expected to be shorter than $Tcomm$, since time has already passed during $Tpred$ and $Tncomp$. Finally, $Tpcomp$ is the time spent by *p* to perform new calculations on the real data, in case of mis-speculation.

One can argue that, with high prediction accuracy, the above can be beneficial to the total execution time of process *p*: $Tpcomp$ is expected to be shorter than $Tcomp$ since the mis-speculated data will be fewer than the real data. This will depend on the prediction rate, the programmer, and the problem at hand.

Note that this work only investigates the predictability of data in MPI messages; it does not investigate the potential with speculative execution.

This paper characterizes the predictability of data in messages of MPI Message-Passing Systems. A simulation infrastructure that uses code instrumentation was developed to study this predictability. Value predictors are used, drawn from the area of microarchitecture. The analysis focuses on five benchmarks that utilize the MPI standard.

An information vector that may influence the predictability of the data in MPI messages is derived from the messages' structure and program information. An empirical analysis is performed aimed to (a) determine the extend of predictability with different predictors, and (b) isolate the most influential components of the information vector used for prediction. Pre-

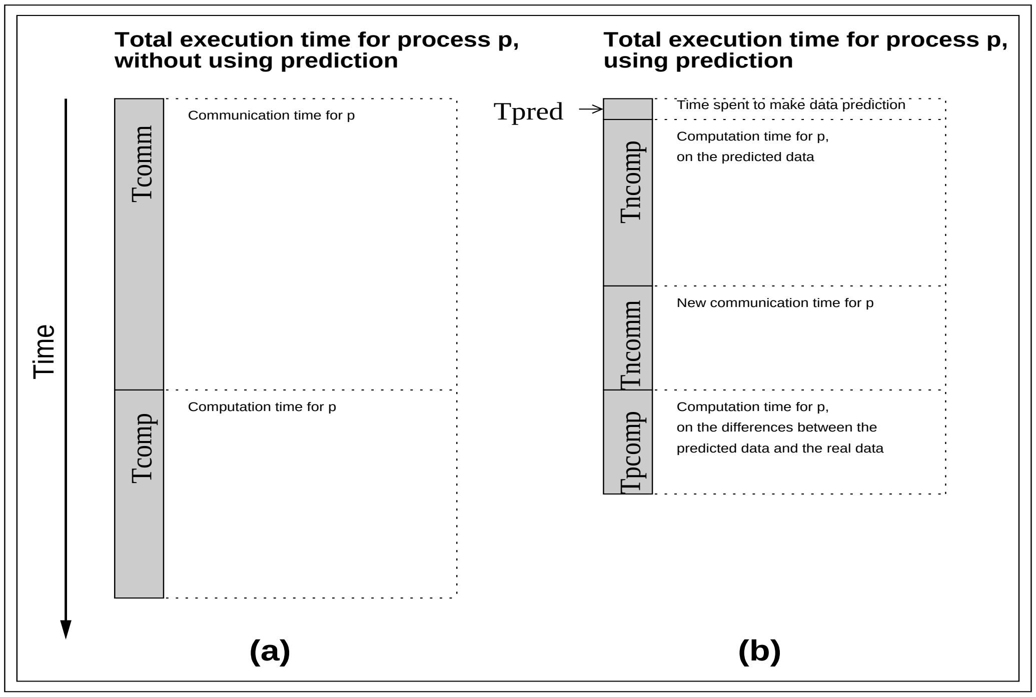
Total execution time for process p, without using prediction
Total execution time for process p, using prediction
Time
Tcomm
Communication time for p
Tcomp
Computation time for p
Tpred
Time spent to make data prediction
Tncomp
Computation time for p, on the predicted data
Tncomm
New communication time for p
Tpcomp
Computation time for p, on the differences between the predicted data and the real data
(a)
(b)

Figure 2: A scenario for speculative execution driven by by data-value prediction in an MPI application

dictability is investigated at different levels: within messages, across messages of the same processes and across messages of different processes.

The results suggest that predictability exists in some but not all of the programs examined. The data also show that only a part of the information vector is useful for higher predictability. The results indicate that small predictors are sufficient to capture most of the predictability.

1.2 Overview

Section 2 reviews related work. Section 3 discusses the design space for the information vector used to predict data in MPI messages. This Section includes an overview of the structure of MPI messages and elaborates on various prediction issues such as granularity and architecture. The experimental framework is presented in Section 4. The empirical results and their discussion is the subject of Section 5. Conclusions and future work are in Section 6.

2 Related Work

2.1 MPI

The two fundamental models for Parallel Processing synchronization are: Shared Memory and Message Passing. In the shared memory model all synchronization takes place through access to the shared memory. In the Message Passing model all the communication and synchronization takes place through the exchange of messages. The message passing model has been traditionally associated with the distributed memory multicomputer systems. Such systems consist of a number of computers connected via a message passing network. The main advantage of the message passing multicomputers is the scalability, and their main disadvantage is the overhead introduced by the movement of large amounts of data in the form of messages. In the early days of distributed memory multicomputers every manufacturer was developing their own version of message-passing libraries. That was followed by the development of portable message passing libraries that were ported to multicomputers of various manufacturers [7]. The Message Passing Interface (MPI) forum [13], a coalition of volunteers from industrial, academic, and governmental organizations, develop the MPI standard which is essentially a third generation system.

In the Message Passing parallel model an application is divided into a number of processes that are executed in parallel. The synchronization and communication among these processes is done through the exchange of messages. MPI is the defacto Message Passing standard [7]. It is also becoming the defacto standard for Grid computing [3]. MPI establishes a practical, portable, efficient, and flexible standard for message passing. It is defined as a set of processes using only local memory, where processes communicate by sending and receiving messages. MPI is a very rich library with more than 100 functions. The communication part of MPI consists of the usual point-to-point communication as well as collective communication/operations. It supports both synchronous and asynchronous communication. The concept of a communicator is used to support communication among the processes. A communicator is a virtual network in which each process has a unique identification number to allow communication with each other. Existing implementations of programming environments for clusters are built on top of a point-to-point communication layer (send and receive) over local area networks. MPI bindings exist for all major programming languages (FORTRAN, C, C++, Java).

2.2 Prediction

Performance is the main motivation for employing predictive techniques in high performance processors. Predictability can drive speculative execution[11] - execution based on predicted information - allowing computations to proceed to execution prior to the deterministic (non-speculative) computation of preceding required information.

Predictability has been considered for architectural and non-architectural information. Examples of architectural information are branch directions[11], branch targets[14], addresses[15], values[16], dependences[17], and coherence protocol information[18]. Examples of non-architectural information are cache bank conflicts[19], cache misses[19], and cache way[20].

Predictability of a sequence is determined by the sequence itself and the predictor used to predict the sequence[21]. Therefore an understanding of the sequence behavior can be essential for choosing an accurate predictor. This work can draw from a wealth of work in computer architecture that examined the behavior and predictability of program information.

Predictability is measured in terms of a particular predictor. Virtually all predictors used in computer architecture can be classified into two categories[22]: *computational* and *context-based*. Other types of predictors exist and are the subject of other fields of studies but are beyond the scope of this paper. Computational predictors predict the next value based on a computation using previous values. Whereas context-based predictors learn the value(s) that follow a particular context, a finite sequence of values, and predict one of the values when the context repeats. These two classes of predictors are mainly aimed to capture the following behaviors: *stride - e.g. 1 2 3 4 5 6 ...* - and *regularly repeating - e.g. 7 -3 1 5 7 -3 1 5 ....*

Some research has focused on the investigation of the actual sources of predictable information in programs[23]. These have shown that the observed predictability is mainly due to program structure not input data.

An orthogonal but very relevant notion to predictability is confidence estimation[24]: assigning likelihood for (in)correctness for a prediction. This may be necessary to employ when prediction accuracy is low and/or the cost of mis-prediction is high.

This work attempts to characterize the predictability of the data in MPI messages, based on the assumption that MPI messages behave similarly to other types of program information. Consequently, the class of predictors considered are drawn from *computational* and *context-based* predictors. In this paper we do not attempt to formalize the causes of the observed predictability but rather isolate the information vector that influences it.

2.3 Prediction and Message-Passing

The use of prediction for message-passing systems has been considered before. For example, it has been shown that parallel programs which use the message-passing model exhibit communication locality[5]: When a source-destination pair is used in a communication event, it has a high probability to be re-used in the near future or in a portion of code ”near” the place that it was used earlier. Based on this observation, prediction techniques can been developed that help avoid re-buffering of incoming messages in MPI systems, by predicting their user-space destination before they arrive[8]. Similar predictors have also been proposed to help avoid re-configuration delays in a reconfigurable optical interconnect network with N nodes and k (possible) direct connections between them[10][9]: the connections between the nodes that will be needed in future communication events are predicted before these events take place, thus reducing the time spent for network re-configuration.

To the best of our knowledge no previous study has considered the use of value prediction for message-passing systems, to reduce *communication time* via speculation.

3 Design Space: What influences the Predictability of Data in MPI Messages

In general, two important decisions influence the predictability of a sequence[21]:

- what is an element of the sequence. This will determine the granularity of prediction, i.e. the amount of information predicted at a time, and
- what is the information vector used to predict each element in the sequence

Since in this paper we are concerned with the predictability of data in MPI messages it is important first to understand the structure of MPI messages and then derive the possible information that may influence the predictability of its data.

3.1 Structure of MPI Messages

MPI provides two distinct communication patterns: Point-to-point communication, where two processes exchange data between them, and collective communication, where data is exchanged between an arbitrary number of processes (usually more than 2).

Point to Point MPI Operations

Typical MPI operations that perform point to point communication between two processes, say $p1$ and $p2$, are:

- MPI_Send and MPI_Recv: data of process $p1$ is transmitted to process $p2$.
- MPI_ISend and MPI_IRecv: data of process $p1$ is transmitted to process $p2$, but in a non-blocking fashion.
- MPI_Sendrecv: data of process $p1$ and data of process $p2$ are swapped.
- MPI_Sendrecv_replace: data of process $p1$ and data of process $p2$ are swapped and replaced.

Collective MPI operations

Typical MPI operations that perform collective communication between a set of processes, say $S = \{p0, p1, p2, ..., pn\}$ are:

- MPI_Bcast: data of a ”root” process (which is a member of S) is sent to all processes in S.
- MPI_Reduce: data of process pi, for each pi in S, are used to calculate a result (using an operation, such as sum, maximum or minimum), which is then sent to a ”root” process, member of S.
- MPI_Allreduce: same as MPI_Reduce, with the difference that the resulting data is received by all processes in S.
- MPI_Gather: data of process pi, for each pi in S, are concatenated and sent to a ”root” process, member of S.
- MPI_Scatter: data of a ”root” process (member of S) is broken up to n parts, $d1, d2, ..., dn$, each of which is sent to $p1, p2, ..., pn$ respectively.
- MPI_Allgather: same as MPI_Gather, with the difference that the gathered data is received by all processes in S.

MPI Message Contents

An MPI message, whether it responds to a collective or a point-to-point operation, typically consists of two parts:

- The actual data that will be transmitted - **the predictability of these data is the focus of this paper.**
- An envelope, which contains meta-information on the message’s data, and the necessary information for the communication to take place.

The message data is typically an array of values, stored in a memory buffer and accompanied by meta-information, which includes its datatype, the buffer’s location in memory, and the buffer’s size. MPI includes support for 13 basic datatypes, eleven of which correspond to the standard C variable types, plus the MPI_BYTE type, usually used for the communication of raw data between nodes with a different architecture, and the MPI_PACKED type.

MPI provides mechanisms to allow non-contiguous data to be transmitted in a single message using *packing* and *derived datatypes*. Packing involves storing arrays of non-contiguous data in a contiguous buffer and then transmitting it using the

MPI_PACKED datatype. The buffer is then un-packed to the original non-contiguous data by the receiving process. The derived datatypes method allows the ad-hoc creation of new, complex datatypes by the programmer. Essentially, a derived datatype is a sequence of pairs $\{(t0, d0), (t1, d1), ..., (tn, dn)\}$ where ti is a basic datatype and di is a displacement into the buffer.

Finally, the envelope of an MPI message contains all the information required for the communication to take place. For brevity, henceforth we will refer to this information as the message's *communication context*. It includes (at a minimum) the rank of the sending process, the rank of the receiving process, and a communicator. A communicator is simply the identifier of a collection of processes that can send messages to each other. A process rank is an integer that uniquely identifies a process within the scope of a communicator. Therefore, a process rank is of consequence only if it is accompanied by a communicator. The envelope of some point-to-point operations also includes a tag, which is an integer that can be used to identify the operation that the message corresponds to. Since MPI messages can be delivered asynchronously, tags are used to distinguish sequences of point-to-point operations with otherwise identical communication contexts.

Based on the above understanding the next three subsections discuss various issues regarding the prediction of MPI messages data: the granularity of prediction, the information vector used to read and train a predictor, and various options for prediction architecture.

3.2 Prediction Granularity

At least two possibilities exist regarding the granularity of prediction of MPI message data: either the data can be treated as an atomic unit or be broken to smaller subparts. Since, in most cases, the data in an MPI message is in the form of an array of values, we opted for the second choice because it allows examination of predictability at a finer granularity within a message. This may be helpful to establish if correlations exist within elements of a message's data buffer.

Consequently to predict one message's data several smaller predictions are made. For example, the data in messages with basic data types (e.g. MPI_CHAR, MPI_INT, MPI_FLOAT, ...) is in the form of an array of values and a prediction will be performed for each array element.

Special treatment may be needed for the handling of messages with derived data types and MPI_PACKED data type. For example, messages with *derived* data types can be broken up into smaller messages, one for each of their basic datatypes and subsequently treated as distinct messages. Messages with the MPI_PACKED data type can be treated as bearers of raw data. The specific choices made in this work for the handling of derived and packed message types are discussed in Section 4.2.

3.3 Information Vector

Based on the the structure of an MPI message an information vector will be derived that can be used for reading and training an MPI message data predictor. The general philosophy behind the use of a particular information type is to enable the predictor to distinguish between different data values in a message buffer. Such a distinction is important to avoid *destructive aliasing* during prediction [25]. Conversely, the omission of a component may be beneficial to prediction when different message elements exhibit identical behavior (constructive aliasing).

The following information, extracted from the the structure of MPI messages, may be influential - constructive or destructive - to the predictability of data in a message:

- The *communication context* of the message. This includes the rank of its sender process, the rank of its receiving process, and the communicator they refer to. Note that the sender process rank does not provide useful information for some MPI operations (such as MPI_Gather, where information is sent by all processes in a communicator). In such cases, the rank of the sender process can be ignored and instead characterize the communication context of the message only by its receiving process and communicator.
- The *datatype* of the message, e.g. MPI_CHAR, MPI_INT, MPI_FLOAT, etc.
- The type of the *MPI operation* that the message responds to, e.g. MPI_Send / MPI_Recv, MPI_Bcast, MPI_Gather, etc.
- The position of the memory buffer where the message's data is stored in the receiving process' address space, hereafter referred to as the *memory position*.
- The offset of the value to be predicted in the message's data buffer. Note that when the offset is part of the information vector used to obtain a prediction, it prevents a predictor from observing predictability within elements of a particular buffer. The use of the offset is aimed to capture predictability across messages for elements with the same offset.

In addition to the above we also considered the position of the MPI operation that receives the message, in its receiving process' address space, hereafter referred to as the *program position*. This position is not part of the MPI message itself, but provides basic information about the control flow that leads to the reception of a particular message. This may be important since messages at different locations in an MPI program may exhibit different behavior. It may also be helpful to distinguish between messages with identical envelopes and data buffers.

3.4 Prediction Architecture: Local vs Global Prediction

There are several prediction architectures that can be employed to measure the predictability of data in MPI messages. Probably the most basic architecture is one that employs a separate predictor for each *program position* of an MPI operation that receives a message. In this scenario, if there are *p* processes and each process contains *m program positions* then *p* x *m* predictors will be used. Each time a message is received, the predictor associated with the *program position* is used to predict all elements in the message. This scheme is referred to as *Per-Message-Prediction*.

Alternatively, a predictor can be used to predict the data in all messages received by one process (i.e. there are only as many predictors as number of processes). We refer to this scheme as *Local-Prediction*. This scheme may be at a disadvantage, as compared to *Per-Message-Prediction*, when predictor capacity is an issue - since one *local-predictor* needs to learn information for all messages in a process. However, Local-Prediction may exploit redundant information contained in different messages received by the same process.

A third option, is a *Global-Prediction* scheme where one predictor is used to predict data for all messages in all processes. This scheme represents possibly not a realistic scheme - since such global view of the messages may be not practical or desirable in real time. But its performance may reveal the amount of predictability across messages sent to different processes.

In the experimental section we explore the performance of Local and Global schemes.

4 Experimental Framework

4.1 Benchmark Suite

We have experimented with five different MPI applications. Three of them are simple parallel programs which solve simple mathematical problems, with pseudo-random numbers (integers or floating-point, depending on the application) as their input data. These are *fox* (a parallel implementation of Fox's algorithm for multiplying two NxN matrices)[26], *bitonic* (an implementation of the parallel bitonic sorting algorithm)[26], and *jacobi* (a parallel implementation of Jacobi's method for solving systems of linear equations)[26]. The other two are real-life MPI applications, available in the public domain. The first, *pov-mpi*[27] is a parallel version of the well-known ray tracing engine Pov-Ray. The second is a parallel version of the *bladeenc*[28] audio MPEG encoder. Both applications were fed with realistic input data (a 3D scene description file in the case of *pov-mpi*, and a 50 MB .wav file in the case of *bladeenc*). Table 1 provides information for each benchmark and its data set.

Name	**Input**	**Output**
fox	two 1024x1024 matrix, populated by random integers, ranging from 0 to 100000	a 1024x1024 matrix that contains the product of the input matrices
bitonic	an array, containing 1000000 integers, ranging from 1 to 100000	The input array, sorted in ascending order
jacobi	a 512x512 and a 1x512 matrix, which represent a system of 512 linear equations	a 512x1 matrix that contains the solution to the input system
bladeenc	a 50 MB uncompressed audio file (16-bit samples, 44 KHz rate), in .wav format	a 4 MB compressed audio file (in 128 Kbps MP3 format)
pov-mpi	a 20 KB text file that contains the description and textures of a 3D scene representing a flying teapot	a 2000x16000 image of the input 3D scene, in uncompressed .tga format

Table 1: Benchmarks Characteristics

4.2 Simulation Infrastructure

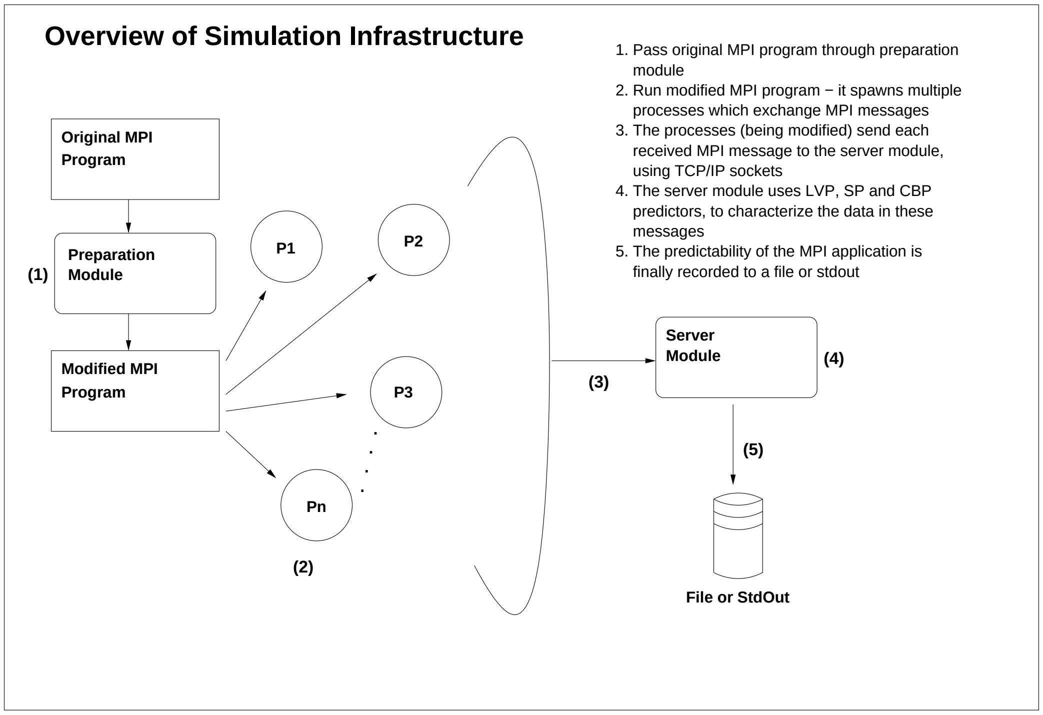
Overview of Simulation Infrastructure
1. Pass original MPI program through preparation module
2. Run modified MPI program − it spawns multiple processes which exchange MPI messages
3. The processes (being modified) send each received MPI message to the server module, using TCP/IP sockets
4. The server module uses LVP, SP and CBP predictors, to characterize the data in these messages
5. The predictability of the MPI application is finally recorded to a file or stdout
Original MPI Program
(1)
Preparation Module
Modified MPI Program
P1
P2
P3
Pn
(2)
(3)
Server Module
(4)
(5)
File or StdOut

Figure 3: Overview of Simulation Infrastructure

To perform the experimentation, a simulation infrastructure was developed that is able: (a) to extract the messages exchanged between the processes of an MPI application during run-time, (b) use a set of predictors to characterize predictability, and, (c) store and analyze the results. Figure 3 provides an overview of the simulation engine developed and used for this study.

The engine is composed of 4 distinct modules: preparation, server, scheduler and analysis. The preparation module is written in C with the help of the GNU flex lexical analyzer. It automatically scans the contents of the benchmarks source files, inserting instrumentation code after each MPI call that results to a message being received (most point-to-point and collective operations are supported). This code uses standard TCP/IP sockets to send the message's envelope and data to the server module of our engine. The preparation module currently supports C and C++ source files, but it can be easily extended to include other programming languages, such as Fortran.

The server module is an application written in C++ and includes the implementation of the predictors used in this work. It behaves as a standard TCP/IP server application, binding itself to a port and continuously accepting the messages that the processes of a (previously altered by the engine's preparation module and re-compiled) MPI application exchange during its life-time. Upon the acceptance of a message, it predicts its elements using the predictors, keeping the results in its local memory. Upon program termination, the results are written to a file.

The server module is parameterized to specify, among other things, the information vector used by the predictors (see Section 4.3). The scheduler module is a shell script that allows to define the set of MPI applications to be tested, along with different predictor configurations. The analysis module is a set of scripts used to summarize the data stored by the server module and generate different graph types.

Handling of Derived and Packed Message Types

Messages with *derived* data types are broken up into smaller messages, one for each of their basic datatypes. For example, a message with the derived data type $\{(MPI\_INT, 0), (MPI\_FLOAT, 16), (MPI\_CHAR, 24)\}$ and a 30-byte buffer

is broken up into 3 messages, whose data is characterized by the following sizes and data types [1].
First message: an array of 4 values of type MPI_INT.
Second message: an array of 2 values of type MPI_FLOAT.
Third message: an array of 6 values of type MPI_CHAR.
Messages with the MPI_PACKED data type are treated as bearers of raw data, arrays of words (whose size depends on the architecture the MPI application is running on). This is specific to the implementation of MPI we were using, which does not provide a mechanism to decode data in an MPI_PACKED buffer. We recognize that the above transformations may impair the success rate of predictors.
Note that in the test platform[2] two thirds of the basic MPI datatypes correspond to C datatypes with one word size.

4.3 Predictors

The predictability of the data in MPI messages was measured using three types of predictors: Last Value[16], Stride[15] and Context-Based[22].
The last value (*LVP*) predictor consists of a fully associative table with replacement guided by LRU. Each entry contains a tag and a prediction. Each tag contains an unhashed information vector used to access the table. The prediction field holds a value associated with a particular information vector. After a misprediction, the prediction in an entry is replaced with the correct value.
The stride predictor (*SP*) is similar to the last-value, with an additional field that contains the difference (stride) between two consecutive values. The prediction for a stride predictor is computed as the sum of the prediction field and the stride field.
The context-based predictor (*CBP*) consists of two tables. Each first level table entry contains a tag, an unhashed information vector used to access the first level table, and a context - n most recent values associated with the tag. The first level table is accessed to obtain a context that is used to index the second table. The second level table that contains a tag and a prediction. The tag in a second level table entry contains the context of a first level table entry.

For all predictors, when there is a tag mismatch - i.e. no entry matches the information vector used to access the prediction table - a prediction is considered incorrect.

We recognize that the predictor organization may be engineered in more efficient manner but the purpose of this paper is mainly to characterize the data predictability of MPI messages not to propose efficient predictors. The use of fully associative tables allows the removal of predictor idiosyncrasies due to aliasing and hashing.

In the experimentation unless indicated otherwise: (a) each predictor table contains 256K entries, (b) for each benchmark the best performing information vector is used (c) for the context-based predictor a context contains four previous values, and (d) the prediction architecture is local (i.e. one predictor per process).

5 Results

5.1 Benchmarks Runtime Behavior

Some of the run time characteristics of the various benchmarks are shown in Table 2. The inputs of the applications were selected in a way that the volume of data exchanged during their run-time (with a 4 process configuration) is similar. More specifically, *fox*, *bitonic* and *jacobi* exchange 3.8-4.7 million data values, and *bladeenc* and *pov-mpi* exchange 16.9-24.3 million. The amount of exchanged messages ranges between 15 and 32816. When run with more processes (configurations with 8 and 16 processes have been tested) the amount of data values increases either linearly (*fox* and *bitonic*) or remains the same (*jacobi*, *bladeenc* and *pov-mpi*).

The applications exchange data mainly of types MPI_INT, MPI_FLOAT, MPI_CHAR and MPI_SHORT and perform both point-to-point (*fox*, *bitonic*, *bladeenc*, *pov-mpi*) and collective communication (*fox*, *jacobi*). The aggregate number of unique *program positions* of MPI operations and *memory positions* of data buffers in the MPI processes' address space is relatively small (except in the case of *fox* which has 3084 unique memory positions).

[1]The sizes of the arrays are calculated by the displacement information in the message's derived data type, and the sizes of MPI_INT (4 bytes), MPI_FLOAT (4 bytes) and MPI_CHAR (1 byte) for the Intel 80386 architecture. In different architectures these sizes may be different.
[2]An SMP Intel 80x86 machine, running RedHat Linux 7.2, LAM/MPI 6.5.4 and GCC 2.96.

	Fox	**Bitonic**	**Jacobi**	**Bladeenc**	**Pov-mpi**
Total Messages (4 procs)	7700	15	32816	207	2409
Total Data Values (4 procs)	3932180	3750000	4720664	24325416	16898238
Total Messages (8 procs)	-	55	65632	256	
Total Data Values (8 procs)	-	6875000	4720688	24408495	
Total Messages (16 procs)	19344	175	131264	344	
Total Data Values (16 procs)	4915344	10937500	4720736	24556151	
Values per Data Type (4 procs)					
MPI_CHAR	-	-	-	3679099	145343
MPI_SHORT	-	-	-	20646144	-
MPI_INT	20	3750000	16	173	24180
MPI_FLOAT	3932160	-	4720648	-	16728715
Values per Operation (4 procs)					
MPI_Recv	3932160	750000	-	24325416	16898238
MPI_Sendrecv	-	3000000	-	-	-
MPI_Sendrecv_replace	8	-	-	-	-
MPI_Bcast	12	-	24	-	-
MPI_Scatter	-	-	525312	-	-
MPI_Allgather	-	-	4195328	-	-
Memory Positions (4 procs)	3084	4	28	72	12
Program Positions (4 procs)	14	5	56	11	4

Table 2: Benchmarks Runtime MPI Characteristics

5.2 Overall Data Value Predictability of MPI Messages

The predictability results for each benchmark for the three predictors are shown in Figure 4. The results show that value predictability in MPI messages varies across benchmarks. The prediction rate for *fox* and *bladeenc* is near zero, while for the rest of the applications the prediction accuracy ranges between 46 % – 94 % for LVP, for SP 41–94 %, and for CBP 36–94 %.

In most cases CBP seems to exhibit the highest success rate, followed by LVP. SP's rate is marginally lower than LVP. Thus, we can speculate that the exchanged messages contain mostly regularly-repeating (e.g. 1, 4, 7 , 1, 4, 7, ...) and last-value sequences (e.g. 2, 2, 2, ...), but not stride sequences (e.g. 1, 2, 3, ...) and therefore, in most occasions SP works as an LVP with a longer learning-time.

The only exception to the above trends is *bitonic*. For this benchmark LVP performs considerably better than CBP. This can be explained by considering that this program's input is an array of 1000000 integer values, which range between 0 and 100000. Based on the bitonic sorting algorithm the program is expected to communicate these integers in arrays which contain either unsorted sequences (hardly predictable), or sorted sequences (very predictable). Because of the data set used, the sorted sequences are expected to contain same-value sub-sequences, with average length converging to 10. Based on the learning times[22] of our predictors (1 observed value for LVP, 2 for SP, and 4 for CBP) we can deduce that, for such sequences, the success rate of each predictor is expected to be: 90 % for LVP, 80 % for SP and 60 % for CBP. This is in agreement with the obtained results for *bitonic* shown in Figure 4.

5.3 Local Versus Global Prediction

The predictability of global and local prediction architectures for all three predictors is depicted in Figure 5. Recall that global prediction architecture uses one predictor for all processes whereas local uses one for each process. The results show that the predictors' success rates for these two cases are very similar, with that of *global-prediction* being marginally higher. This may suggest either or both of the following:

- Most predictability can be found between messages received by the same process, as opposed to messages received by all processes.
- The prediction rate for *global-prediction* could be higher but is limited by the predictors' capacity.

Further investigation revealed that capacity is probably not an issue. Because (a) similar behavior to the above was observed for various predictor sizes (1K, 8K, 64K and 256K entries), and (b) in a *global-prediction* scheme if capacity was an issue its prediction rate would fall more rapidly than that of *local-prediction* as predictor sizes decrease.

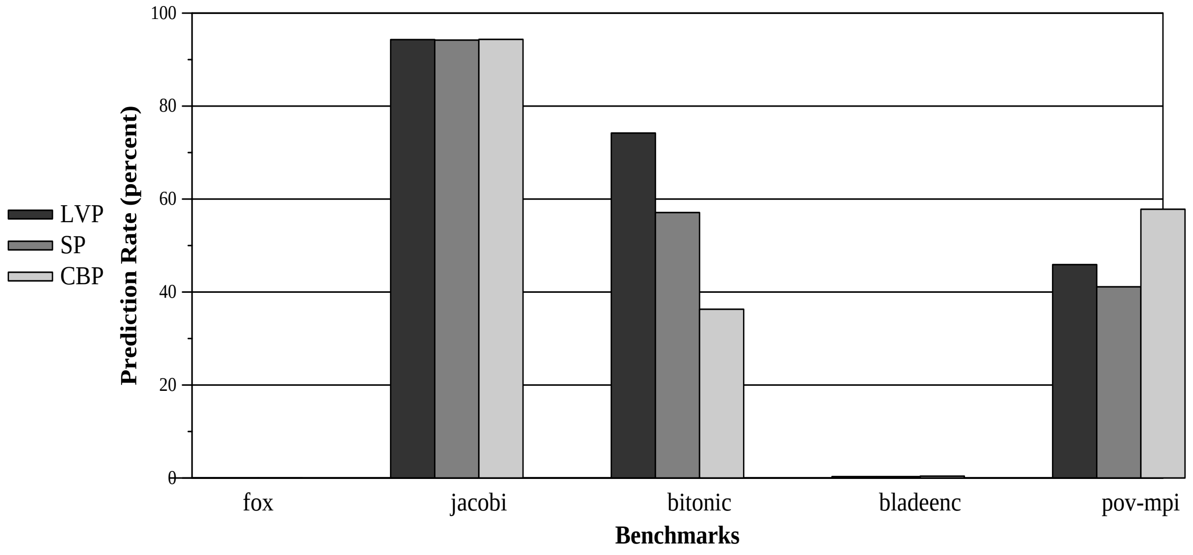

Figure 4: Overall Predictability

For *bitonic* global prediction achieves higher accuracy than *local-prediction* with large predictor sizes. Specifically, prediction rate is 4 - 12 % higher with predictor sizes of 256K entries, while for smaller sizes there is no significant difference. This may suggest that *global-prediction* scheme can achieve better results than *local-prediction* but is limited by capacity problems. This may also be true for other benchmarks and future work should consider the behavior with larger sizes.

In the remaining discussion we focus on the three benchmarks that exhibited data predictability: *bitonic*, *jacobi* and *pov-mpi*.

5.4 Information Vector Combinations

In Section 3 an information vector was defined that may influence the predictability of a data value in an MPI message. It includes: the *offset* in the message's data buffer, the *communication context*, the *program position*, the *memory position*, the *datatype*, and the *MPI operation*.

These can be viewed as six binary variables that can lead to sixty-four combinations of information vectors. Each combination was tested with all benchmarks to assess which vector component(s) is (non)influential to predictability. Figures 6–8 show the prediction rate for each benchmark for all three predictors for these sixty-four combinations. A configuration is represented by a binary six-tuple in the x-axis. A one in a tuple means that the variable it corresponds to is used in the information vector. The order of the variables in the tuples from bottom to top correspond to the above order of variable presentation. For example the *data value offset* corresponds to the bottom variable in the graph, and therefore the first 32 configurations do not include the offset whereas the last 32 do.

Overall, the results suggest that the *data value offset* is the most influential component of the information vector. It affects the success rate of all predictors and applications, either constructively or destructively. This dual behavior can be explained by considering two scenarios that can lead to predictable behavior. In the first, a message can be predictable because the values in its buffer are predictable. In the second, a message is predictable because values at the same *offset* across messages exhibit predictability. For the first scenario the use of the *offset* can be destructive whereas for the second the use of the *offset* can be constructive. The results show that value predictability is also sensitive to the *communication context*, *program position*, and *memory position*. We now proceed to analyze the prediction behavior for each benchmark with the different information vector configurations.

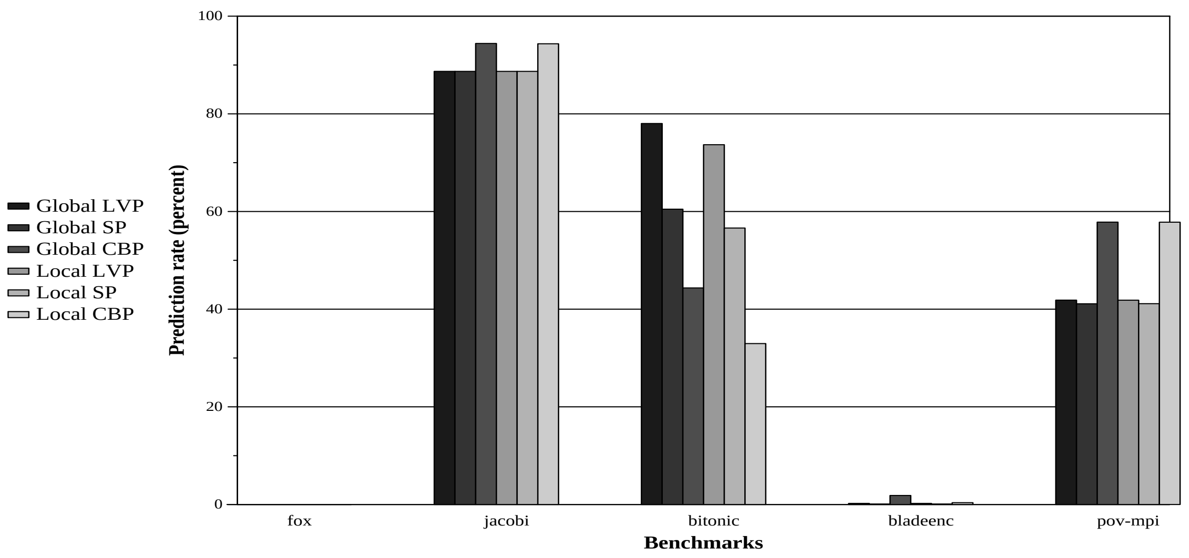

Figure 5: Global vs Local Predictability

Bitonic

In *bitonic*, Figure 6, prediction rates for LVP, SP, and CBP are at 73 %, 57 % and 34 % respectively, when *offset* is not used. This suggests that data within message buffers contains mostly same-value sequences, which are captured with different success rates by different predictors because of their respective learning times. This assumption agrees with our analysis in section 5.2. When *offset* is used, prediction rate for LVP drops to 13 % while for SP it drops to 0 %. This suggests that some last-value sequences, but no stride ones, are found across all messages of this benchmark. These sequences must be short (2 data values long) otherwise SP would have been able to predict some of them. When both *offset* and *communication context* are used, CBP exhibits its lowest prediction rate (26 %). On the other hand, when *offset* is used and *communication context* is not used CBP exhibits its highest (36 %). This may indicate that:

- Regularly-repeating patterns exist across all messages, since the success rate of CBP is in both cases higher than that of LVP in this configuration.
- These patterns are more frequent across messages exchanged by different processes.

Jacobi

In *jacobi*, Figure 7, prediction rate for CBP is at 94 % when *offset* is not used, while prediction rates for LVP and SP are around 88 %. This suggests that many same-value sequences exist within the application's messages. Some regularly-repeating but not same-value sequences also exist, since the success rate for CBP is higher than those of LVP and SP. When *offset* is used, the prediction rate of CBP drops marginally (88 %), which suggests that regularly-repeating sequences are found across messages.

The behavior of LVP and SP when *offset* is set is noteworthy. The prediction rate of LVP drops to 50 % when *program position* and *memory position* are not used, while it drops to 0 % when *memory position* is not used but *program position* is set. The prediction rate of SP drops to 6 % when both *program position* and *memory position* are not set, while it remains at 88 % when *program position* is used and *memory position* is not set. When both *program position* and *memory position* are set, prediction rate for both predictors remains at 88 %, while it rises to 94 % when *program position* is not used and *memory position* is set. This may point to the following:

- *Memory position* increases the prediction rate for both LVP and SP. This leads to the conclusion that many same-value sequences exist across messages that place incoming data to the same memory locations (both predictors can find same-value sequences).
- *Program position* either increases or does not affect the prediction rate for SP, while it lowers it for LVP. This leads to the conclusion that stride sequences exist across messages which correspond to the same MPI operation.

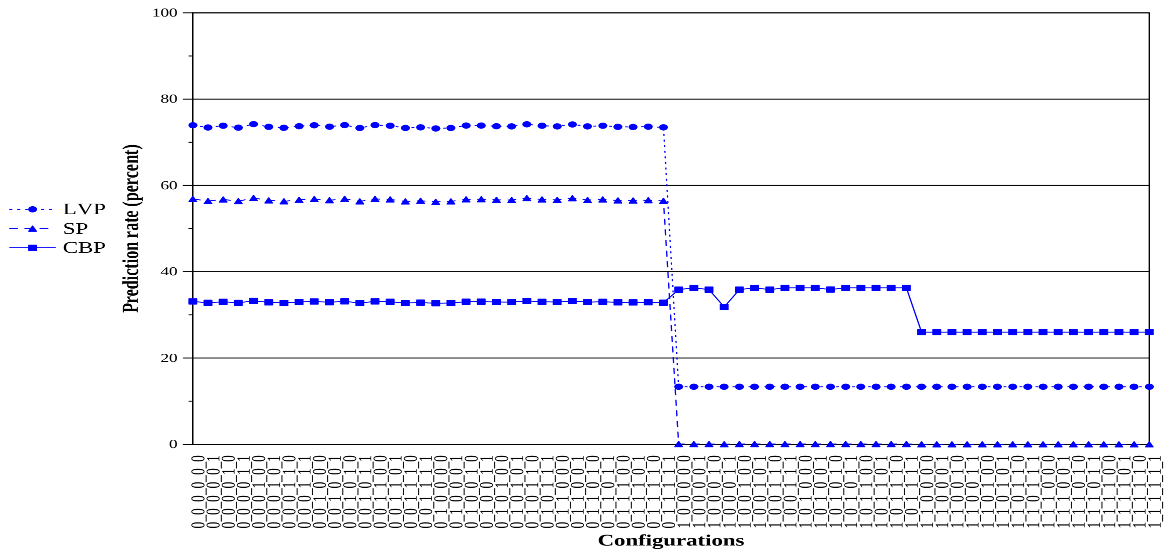

Figure 6: Predictability for various Information Vectors – Bitonic

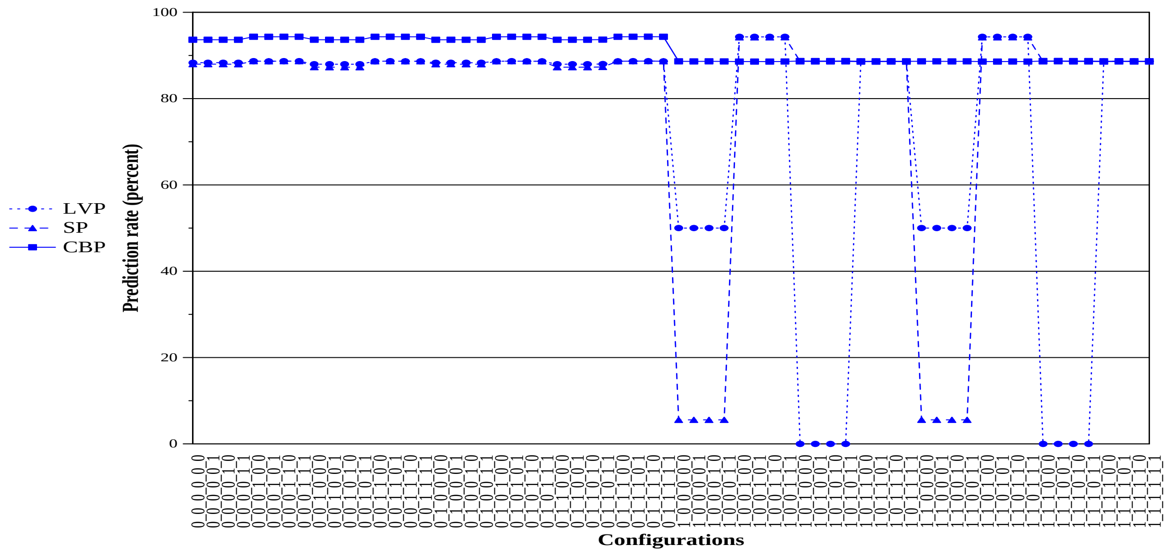

Figure 7: Predictability for various Information Vectors – Jacobi

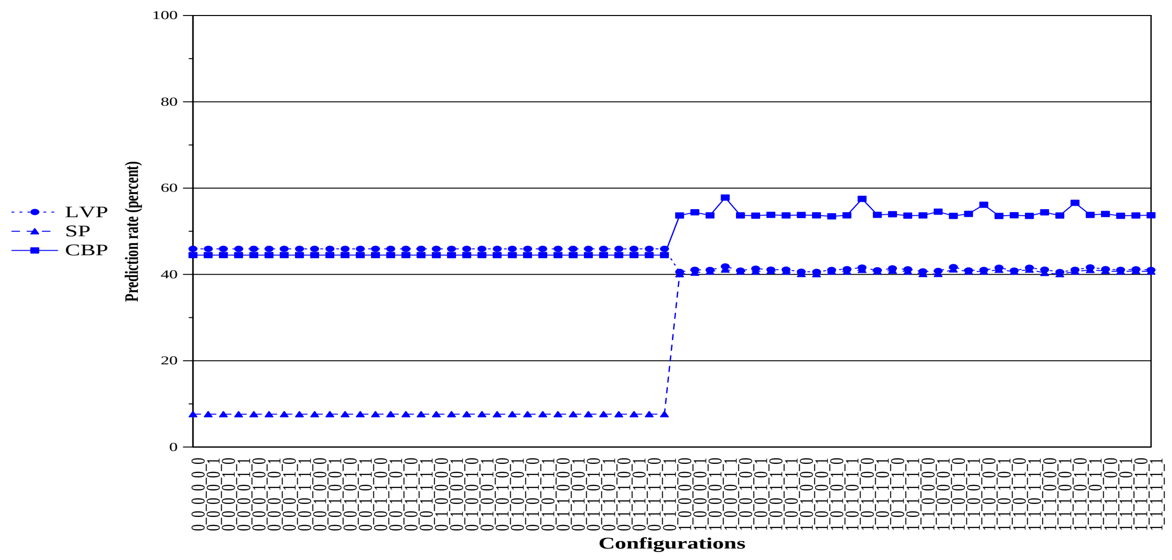

Figure 8: Predictability for various Information Vectors – Pov-MPI

Pov-mpi

In *pov-mpi*, Figure 8, when *offset* is not used the prediction rate for LVP is around 46 %. Thus, same-value sequences must exist within the application's messages. The success rate of SP (learning time = 2) remains at 8 %, therefore most of these sequences can not be more than 2 values long. It is not clear whether the relatively low accuracy of SP can be attributed to stride sequences or to same-value sequences whose size is bigger than 2. CBP has a prediction rate of 44 % when *offset* is not used. This may indicate that regularly-repeating (but not same-value) sequences also exist within the messages of *pov-mpi* because all same-value sequences are short and therefore cannot be learned in time by CBP (learning time = 4). When *offset* is used, the prediction rate of LVP falls to around 41 %. Therefore, same-value sequences must exist across all messages. At the same time, the prediction rate of CBP rises to around 54 %, thus some regularly-repeating sequences also exist. Finally, the prediction rate of SP rises to 41 %, which is similar to the prediction rate of LVP. This is because of the same-value sequences and possible stride sequences that exist.

Overall, the variable behavior observed across benchmarks may suggest that adaptivity[29] may need to be adopted so that the best information vector is used per application based on program behavior. Table 3 shows the "best" performing information for each benchmark. These are the configurations used to obtain the results in the remaining sections.

	Fox	**Bitonic**	**Jacobi**	**Bladeenc**	**Pov-mpi**
Offset	0	0	0	0	1
Communication Context	0	1	1	1	0
Program Position	0	0	1	0	0
Memory Position	0	1	1	1	0
Datatype	1	0	1	0	1
MPI Operation	0	1	0	0	1

Table 3: Best Information Vector Configuration for Each Benchmark

5.5 Distribution of Predictability

Figure 9 shows the *static* distributions of messages according to their data predictability. For the static distribution each message was given the same weight. Figure 10 shows the dynamic distribution, however, each message here is assigned weight equal to its data buffer size. Note that a message belongs to the 100% bucket if all of its data values have been

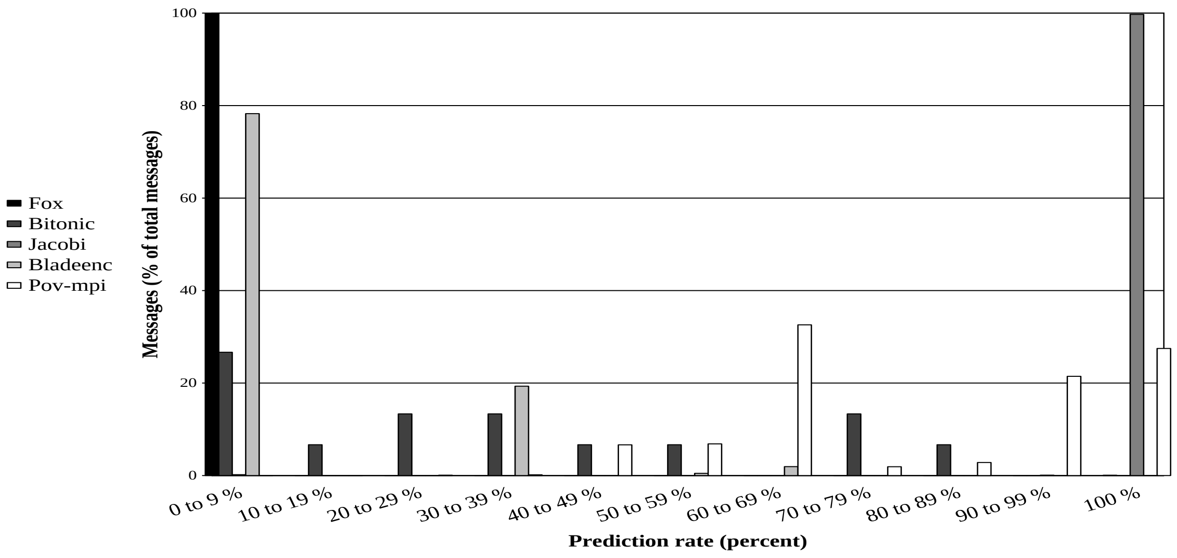

Figure 9: Predictability Distribution (Static) - CBP

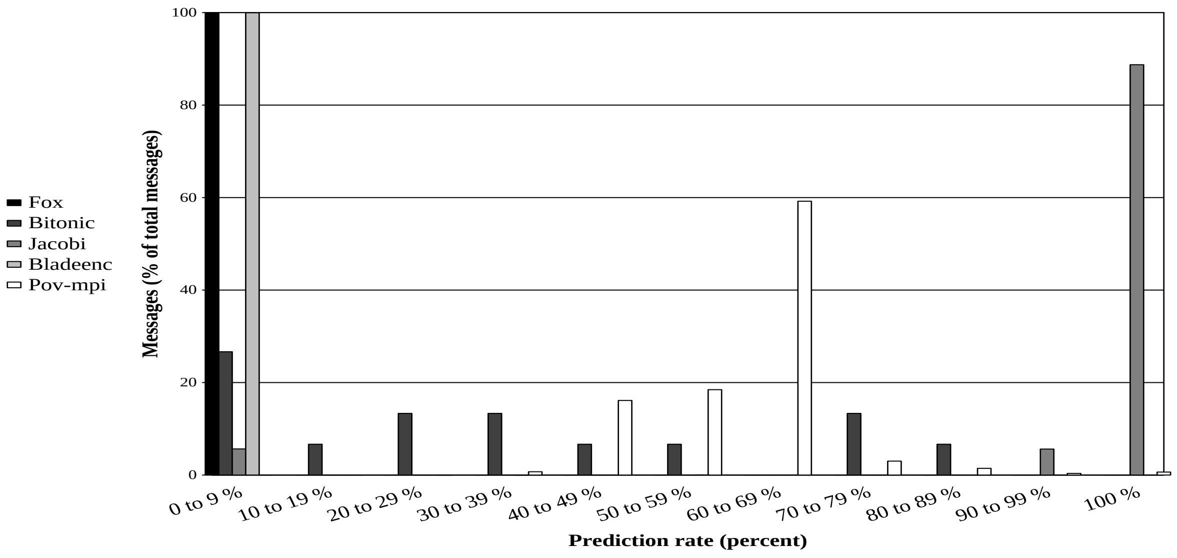

Figure 10: Predictability Distribution (Dynamic) - CBP

predicted correctly. The two graphs show the results for CBP.

For *bitonic* the fraction of messages that is predicted with 70–100 accuracy is 67 % when using LVP, 14 % when using SP, and 20 % when using CBP. Almost 100 % of *jacobi's* messages are predicted with 100% accuracy when using any of the predictors. *Pov-mpi* has 21 % of its messages predicted with 100% accuracy when using LVP and 27 % when using CBP, while 28 % of them are predicted with 90–99% accuracy when using LVP and 21 % when using CBP. Another observation is that all tested applications, including *fox* and *bladeenc*, have some messages that are predicted with 100% accuracy.

The dynamic distributions in Figure 10 follow similar trends to the static, with a tendency toward lower accuracy buckets.

5.6 Predictor Size and Number of Processes

The effect of increasing predictor size, 1K, 8K, 64K, and 256K entries, on the prediction rate for the different predictors and benchmarks is shown in Figure 11.

The results show that good prediction rates can be achieved even with small predictor sizes (e.g. 1024-entry table). CBP exploits better bigger tables than LVP, whose prediction rate is not sensitive to predictor size. For CBP, *jacobi* and *pov-mpi* predictability saturates after 64K entries, suggesting that for these benchmarks the predictor may not require additional space. However, CBP's prediction rate for *bitonic* is getting better even with a 256K configuration.

Figure 12 shows the effect of increasing the number of MPI processes from 4 to 8 and 16, on the predictability. The data show that prediction rate is insensitive to the number of processes. We do recognize that in many realistic situations the number of processes can be much higher, therefore more experimentation may be needed in that direction.

6 Conclusions and Future Work

This paper characterizes the predictability of data exchanged via MPI messages using an infrastructure that relies on code instrumentation. The predictability analysis is based on an information vector derived from the structure of the MPI message and program control flow.

Empirical analysis demonstrated that data predictability does not exist in all message-passing applications. In those applications where predictability is found, is varies between 57–94%. Predictability was found at different levels: within messages, across messages of the same processes and across processes. The most influential component of the information vector is the offset of the data value in a message's buffer. The offset can have constructive or destructive effects on predictability depending on the benchmark. This suggests that adaptivity may be needed to select the best information vector for each program. The data also suggest that the observed predictability can be achieved with relatively small predictors. The best prediction rates are achieved by context-based with close second last-value prediction. The results suggest that typically sequences exhibited last-value and/or regularly-repeating characteristics. Finally the data reveal insensitivity of predictability to increasing number of processes.

This work points to several directions for future research. Further analysis should be performed for other benchmarks and input data sets. This will provide insight about the generality and robustness of data predictability in MPI messages. Another direction for future work is developing predictors that may capture better MPI idiosyncrasies. For example, consider additional information during prediction such as *program position* and *memory position* of the sending process. Ultimately, if MPI message predictability is found to be generic, it can be used to to extend MPI libraries to allow speculative execution based on prediction of data in messages.

References

[1] C. Seitz, "Myrinet- a gibabit per second local-area network," *IEEE Micro*, February 1995.

[2] A. Lachanas and P. Evripidou, "Regional Weather prediction on Small Network of Workstations," in *Proceedings of 24th Euromicro conference*, 1998.

[3] I. Foster and C. Kesselman, "Globus: A toolkit-based grid architecture," in *The Grid: Blueprint for a Future Computing Infrastructure*, pp. 259–278, Morgan Kaufmann, 1998.

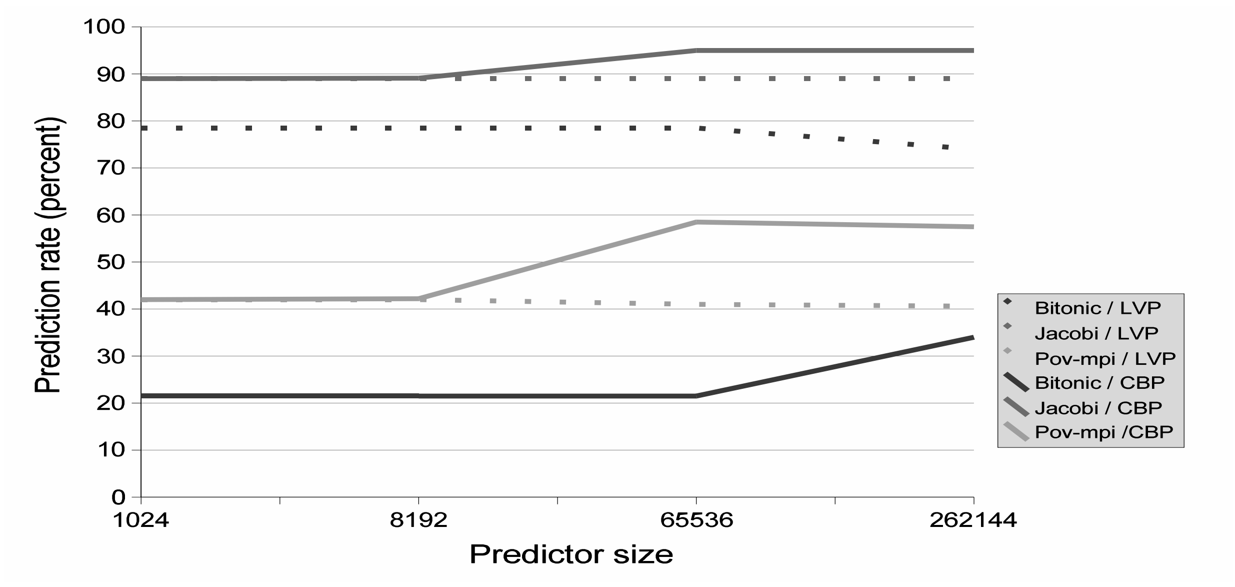

Figure 11: Predictability with Increasing Predictor Size - Local Predictors

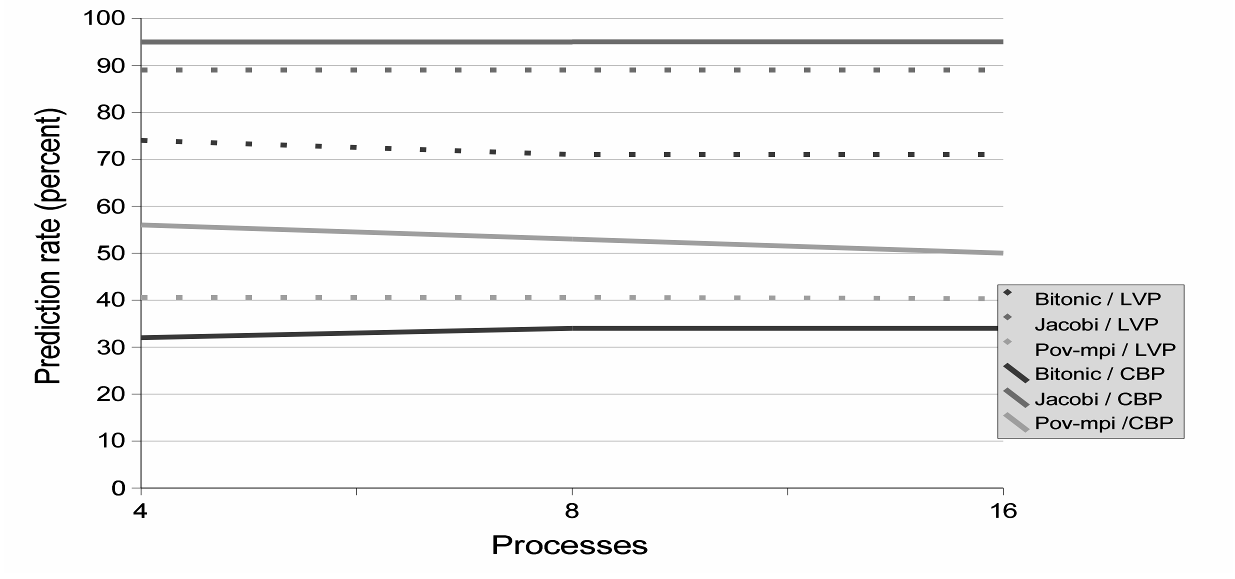

Figure 12: Predictability with Increasing Number of Processes - Local Predictors

[4] P. Evripidou, C. Panayiotou, G. Samaras, and E. Pitoura, "The PaCMAn Metacomputer: Parallel Computing with Java Mobile Agents," *Future generation Computer Systems*, vol. 19, no. 2, pp. 265–280, 2001. FGCS special issue on Java in High Performance Computing.

[5] J. Kim and D. J. Lilja, "Characterization of Communication Patterns in Message-Passing Scientific Application Programs," in *Lecture Notes in Computer Science 1362*, 1998.

[6] S. Karlsson and M. Brorsson, "A Comparative Characterization of Communication Patterns in Applications using MPI and Shared Memory on an IBM SP2," in *CANPC*, 1998.

[7] I. Foster, *Designing and Building Parallel Prgrams*. Addison Weseley, 1995. Includes a succinct and readable introduction to an MPI subset. Also available online at http://www.mcs.anl.gov/dbpp.

[8] A. Afsahi and N. J. Dimopoulos, "Efficient Communication Using Message Prediction for Cluster of Multiprocessors," in *TR ECE-99-5, University of Victoria*, December 1999.

[9] A. Afsahi and N. J. Dimopoulos, "Hiding Communication Latency in Reconfigurable Message-Passing Environments," in *IPPS/SPDP-13*, 1999.

[10] A. Afsahi and N. J. Dimopoulos, "Communications Latency Hiding Techniques for a Reconfigurable Optical Interconnect: Benchmark Studies," in *TR ECE-98-2, University of Victoria*, June 1998.

[11] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[12] M. H. Lipasti and J. P. Shen, "Superscalative Microarchitecture for Beyond AD 2000," *Computer*, September 1997.

[13] M. P. I. Forum, "MPI: A Message-Passing Interface Standard," Tech. Rep. Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.

[14] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, January 1984.

[15] J. L. Baer and T. F. Chen, "An Effective on-chip Preloading Scheme to Reduce Data Access Penalty," in *Proceedings of Supercomputing*, pp. 176–186, November 1991.

[16] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Data Speculation," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, October 1996.

[17] A. Moshovos, S. E. Breach, T. J. Vijaykumar, and G. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 181–193, June 1997.

[18] S. S. Mukherjee and M. D. Hill, "Using Prediction to Accelerate Coherence Protocols," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 179–190, June 1998.

[19] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculative techniques for improving load related instruction scheduling," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[20] K. So and R. N. Rechtschaffen, "Cache Operations by MRU Change," *IEEE Transactions on Computers*, vol. 37, pp. 700–709, June 1988.

[21] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Prentice-Hall Inc., New Jersey, 1990.

[22] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 248–258, December 1997.

[23] Y. Sazeides and J. E. Smith, "Modeling Program Predictability," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 73–84, June 1998.

[24] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 142–152, December 1996.

[25] S. Sechrest, C.-C. Lee, and T. Mudge, "Correlation and aliasing in dynamic branch predictors," in *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 22–32, May 1996.

[26] P. S. Pacheco, *Programming Parallel Processors Using MPI*. Morgan Kaufmann, 1995.

[27] A. Fava, E. Fava, and M. Bertozzi, “Mpipov: A parallel implementation of pov-ray based on mpi,” in *Proceedings of the 6th European PVM/MPI Users’ Group Meeting*, vol. 1697, Lecture Notes in Computer Science-Springer, 1999.

[28] “Parallel mp3 encoding version 0.92.1b5 released: September 3, 2000.”

[29] T. Juan, S. Sanjeevan, and J. J. Navarro, “Dynamic History-Length Fitting: A third level of adaptivity for branch prediction,” in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 155–166, June 1998.