Construction and Evaluation of Coordinated Performance Skeletons

Qiang Xu, Jaspal Subhlok

Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-08-09
May 26, 2008

Keywords: Performance skeletons, Performance prediction, Trace compression

Abstract

Performance prediction is particularly challenging for dynamic foreign environments that cannot be modeled well, such as those involving resource sharing or foreign system components. Our approach is based on the concept of a performance skeleton which is a short running program whose execution time in any scenario reflects the estimated execution time of the application it represents. The fundamental technical challenge is automatic construction of performance skeletons for parallel MPI programs. The steps are 1) generation of process execution traces and conversion to a single coordinated logical program trace, 2) compression of the logical program trace, and 3) conversion to an executable parallel skeleton program. Results are presented to validate the construction methodology and prediction power of performance skeletons. The execution scenarios analyzed involve network sharing, different architectures and different MPI libraries. The emphasis is on identifying the strength and limitations of this approach to performance prediction.

Construction and Evaluation of Coordinated Performance Skeletons

Qiang Xu Jaspal Subhlok*

University of Houston
Department of Computer Science
Houston, TX 77204

Abstract

Performance prediction is particularly challenging for dynamic foreign environments that cannot be modeled well, such as those involving resource sharing or foreign system components. Our approach is based on the concept of a performance skeleton which is a short running program whose execution time in any scenario reflects the estimated execution time of the application it represents. The fundamental technical challenge is automatic construction of performance skeletons for parallel MPI programs. The steps are 1) generation of process execution traces and conversion to a single coordinated logical program trace, 2) compression of the logical program trace, and 3) conversion to an executable parallel skeleton program. Results are presented to validate the construction methodology and prediction power of performance skeletons. The execution scenarios analyzed involve network sharing, different architectures and different MPI libraries. The emphasis is on identifying the strength and limitations of this approach to performance prediction.

Index Terms
Performance skeletons, Performance prediction, Trace compression

1 Introduction

Traditional performance prediction and scheduling for distributed computing environments is based on modeling of application characteristics and execution environments, with some example systems discussed in [1, 7, 8, 11]. However, this approach is of limited value in some dynamic and unpredictable execution scenarios as modeling is impractical or impossible for a variety of reasons. Some example scenarios are execution with sharing of network or compute resources, execution with varying number of available processors, or execution with new system architectures or software libraries.

*This material is based upon work supported by the National Science Foundation under Grant No. ACI- 0234328 and Grant No. CNS-0410797. Contact email: jaspal@uh.edu

A new approach to performance prediction in such foreign environments is based on the concept of a *performance skeleton* which is defined to be a short running program whose execution time in any scenario reflects the estimated execution time of the application it represents. When the performance skeleton of an application is available, an estimate of the application execution time in a new environment is obtained by simply executing the performance skeleton and appropriately scaling the measured skeleton execution time. The main challenge in this approach is automatic construction of performance skeletons from applications. Earlier work in this project developed basic procedures for construction of communication and memory skeletons and explored their usage in distributed environments [13, 9, 10].

This paper introduces *scalable* construction of *coordinated* performance skeletons and evaluates their ability to predict performance in a variety of execution scenarios. The skeletons developed are "coordinated" implying that a single SPMD skeleton program is constructed instead of a family of process level skeletons. Improved compression procedures were developed that allow fast and nearly linear time skeleton construction. Finally, experimentation is conducted in a wide variety of scenarios including shared network bandwidth, shared processors, variable number of processors, different cluster architectures, and different MPI communication libraries. The results highlight the power and limitations of this approach.

We outline the construction of performance skeletons for parallel MPI programs. Clearly a performance skeleton must capture the core execution and communication characteristics of an application. The skeleton construction procedure begins with the generation of process traces of an MPI application, primarily consisting of the message passing calls interspersed with computation segments. The first processing step is **trace logicalization** which is the conversion of the suite of MPI process level execution traces into a single logical trace. This is followed by **trace compression** which involves identification of the loop structure inherent in the execution trace to capture the core execution behavior. Final **skeleton construction** consists of generation of a deadlock free skeleton SPMD program from the compressed logical trace. The key steps are illustrated in Figure 1.

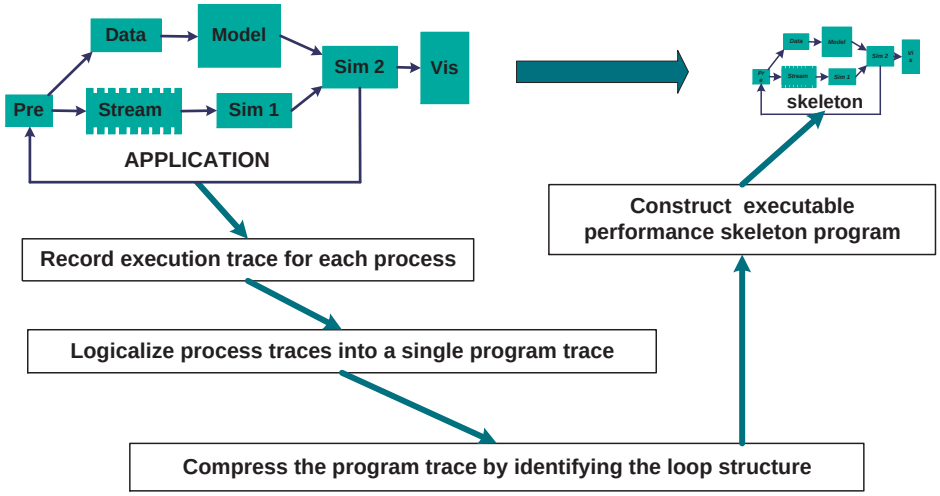


Figure 1. Skeleton construction

The paper is organized as follows. Section 2 presents the procedure for logicalization of MPI

traces and section 3 presents the procedures developed for the compression of the logical trace. Section 4 introduces deadlock free skeleton program generation from the compressed trace. Section 5 presents and discusses results from the application of performance skeletons for performance prediction. Section 6 contains conclusions.

2 Trace logicalization

As high performance scientific applications are generally SPMD programs, in most cases, the traces for different processes are similar to each other and the communication between processes is associated with a well defined global communication pattern. A study of DoD and DoE HPC codes at Los Alamos National Labs [3] and analysis of NAS benchmarks [12] shows that an overwhelming majority of these codes have a single low degree stencil as the dominant communication pattern. These characteristics expose the possibility of combining all processor traces into a single *logical program trace* that represents the aggregate execution of the program - in the same way as an SPMD program represents a family of processes that typically execute on different nodes. For illustration, consider the following sections of traces from a message exchange between 4 processes in a 1-dimensional ring topology.

Process 0	**Process 1**	**Process 2**	**Process 3**
...	*...*	*...*	*...*
snd(P1,...)	*snd(P2,...)*	*snd(P3,...)*	*snd(P0,...)*
rcv(P3,...)	*rcv(P0,...)*	*rcv(P1,...)*	*rcv(P2,...)*
...	*...*	*...*	*...*

The above physical trace can be summarized as the following logical trace:

Program

...

snd(P_R,...)

rcv(P_L,...)

...

where P_L and P_R refer to the logical left and logical right neighbors, respectively, for each process in a 1-dimensional ring topology.

Beside reducing the trace size by a factor equal to the number of processes, the logical program trace captures the parallel structure of the application. Note that this logicalization is orthogonal to *trace compression* which is discussed in the following section.

The logicalization framework has been developed for MPI programs and proceeds as follows. The application is linked with the PMPI library so that all message exchanges are recorded in a trace file during execution. Summary information consisting of the number of messages and bytes exchanged between process pairs is recorded and converted to a binary *application communication matrix* that identifies process pairs with significant message traffic during execution. This matrix is then analyzed to determine the application level communication topology. Once this global topology is determined, a representative process trace is analyzed in detail and transformed into a logical program trace where all message sends and receives are to/from a logical neighbor in terms of a logical communication topology (e.g a torus or a grid) instead of a physical process rank. An example physical trace and the corresponding logical trace are shown in Table 2.

PHYSICAL TRACE	***LOGICAL TRACE***
....	....
MPI_Isend(... **1**, MPI_DOUBLE, 480, ...)	MPI_Isend(...**EAST**, MPI_DOUBLE, 480, ...)
MPI_Irecv(... **3**, MPI_DOUBLE, 480, ...)	MPI_Irecv(...**WEST**, MPI_DOUBLE, 480, ...)
MPI_Wait() */* wait for Isend */*	MPI_Wait() */* wait for Isend */*
MPI_Wait() */* wait for Irecv */*	MPI_Wait() */* wait for Irecv */*
....	....
MPI_Isend(... **4**, MPI_DOUBLE, 480, ...)	MPI_Isend(...**SOUTH**, MPI_DOUBLE, 480, ...)
MPI_Irecv(...**12**, MPI_DOUBLE, 480, ...)	MPI_Irecv(...**NORTH**, MPI_DOUBLE, 480, ...)
MPI_Wait() */* wait for Isend */*	MPI_Wait() */* wait for Isend */*
MPI_Wait() */* wait for Irecv */*	MPI_Wait() */* wait for Irecv */*
....	....
MPI_Isend(... **7**, MPI_DOUBLE, 480, ...)	MPI_Isend(...**SOUTHWEST**, MPI_DOUBLE, 480, ...)
MPI_Irecv(...**13**, MPI_DOUBLE, 480, ...)	MPI_Irecv(...**NORTHEAST**, MPI_DOUBLE, 480, ...)
MPI_Wait() */* wait for Isend */*	MPI_Wait() */* wait for Isend */*
MPI_Wait() */* wait for Irecv */*	MPI_Wait() */* wait for Irecv */*
....	....

Table 1. Logical and physical trace for the 16-process BT benchmark

The key algorithmic challenge in this work is the identification of the application communication topology from the application communication matrix which represents the inter-process communication graph. The communication topology is easy to identify if the processes are assigned numbers (or ranks) in a well defined order, but is a much harder problem in general. This is illustrated with a very simple example in Figure 2. The figure shows 9 executing processes with a 2D grid communication topology. In Figure 2(a) the processes are assigned numbers in row major order in terms of the underlying 2D grid. However, if the processes were numbered diagonally with respect to the underlying 2D grid pattern as indicated in Figure 2(b), the communication graph with process nodes laid out in row major order would appear as Figure 2(c). Clearly, the underlying 2D grid topology is easy to identify in the scenario represented in Figure 2(a) by a pattern matching approach but much harder when process numbering follows an unknown or arbitrary order, a relatively simple instance of which is the scenario represented in Figure 2(c). The state of the art in identifying communication topologies assumes that a simple known numbering scheme is followed [3].

The reasons topology identification is difficult are 1) establishing if a given communication graph matches a given topology is equivalent to solving the well known *graph isomorphism* problem for which no polynomial algorithms exist and 2) there are many different types of topologies (different stencils on graph/torus, trees, etc.) and many instantiations within each topology type (e.g., different number and sizes of dimensions even for a fixed number of nodes). In order to

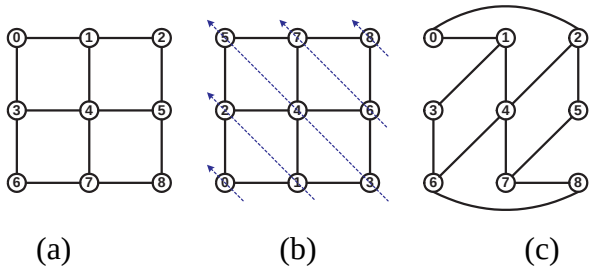


Figure 2. 2D grid topology with row major and other numberings

identify if a given communication matrix matches any known topology, the following sequence of steps are applied as a decision tree with simpler tests applied first for efficiency:

1. **Simple Tests:** First all possible sizes of grid/tori/tree based on the number of processes N are identified with prime factoring. Then the number of edges and the degree ordered sequence of nodes for the given communication matrix are matched with those for the suite of known topologies. This step typically eliminates all but 1 or a few topologies as possible matches.

2. **Graph Spectrum Test:** Based on computing eigenvalues - eigenvalue sets of isomorphic graphs are identical. Hence if the eigenvalues do not match, the topologies are not a match.

3. **Isomorphism Test:** Applies graph isomorphism to establish that a given communication matrix exactly represents a specific topology.

The details of this process are described in [14]. The tracing required for logicalization procedure is very low overhead in computation time and volume as only high level message passing calls are recorded. The analysis required for each process trace is minimal - only the collection of gross communication data, such as the number of messages and bytes exchanged. Detailed processing is limited to a single representative process trace that is transformed to a logical program trace.

Table 2 presents observations from the application of this procedure to selected NAS benchmarks. The topologies that remain as candidates after each of the tests and the final established topology are listed along with processing times. Clearly the procedure is effective and efficient.

3 Trace compression

An important step in the process of construction of performance skeletons is the identification of repeating patterns in MPI message communication. Since the MPI communication trace is typically a result of loop execution, discovering the executing loop nest from the trace is central to the task of skeleton construction. The discovery of "loops" here technically refers to the discovery of tandem repeating patterns in a trace (presumably) due to loop execution.

Common compression procedures include *gzip* [16] that constructs a dictionary of frequently occurring substrings and replaces each occurrence with a representative symbol, and *Sequitur* [4, 5] that infers the hierarchical structure in a string by automatically constructing and applying grammar rules for reduction of substrings. Such methods cannot always identify long range loop patterns

Benchmark (Processes)	Simple Tests	Graph Spectrum Test	Isomorphism Test	Trace Length Records(size)	Time (secs)
BT (121)	**11×11 6-p stencil**	**11×11 6-p stencil**	**11×11 6-p stencil**	50874 (2106KB)	30.76
SP (121)	**11×11 6-p stencil**	**11×11 6-p stencil**	**11×11 6-p stencil**	77414 (3365KB)	49.16
LU (128)	**16×8 grid**	**16×8 grid**	**16×8 grid**	203048 (9433KB)	134.30
CG (128)	**3-p stencil** **16×2×2×2 grid**	**3-p stencil**	**3-p stencil**	77978 (3224KB)	47.89
MG (128)	**8×2×2×2×2 torus** 8×4×2×2 torus 8×4×4 torus	**8×2×2×2×2 torus** 8×4×2×2 torus 8×4×4 torus	**8×2×2×2×2 torus** 8×4×2×2 torus 8×4×4 torus	9035 (386KB)	7.33

Table 2. Identification of communication topologies of NAS benchmarks. Unique topologies are listed in boldface with other isomorphic topologies below them.

because of early reductions. An alternate approach is to attempt to identify the longest matching substring first. However, simple algorithms to achieve this are at least quadratic in trace length and hence impractical for long traces. A practical tradeoff is to limit the window size for substring matching, which again risks missing long span loops [6].

Our research took a novel approach to identifying the loop structure in a trace based on Crochemore's algorithm [2] that is widely used in pattern analysis in bioinformatics. This algorithm can identify all repeats in a string, including tandem, split, and overlapping repeats, in $O(nlogn)$ time. A framework was developed in this research to discover the loop nest structure by recursively identifying the longest span tandem repeats in a trace. The procedure identifies the optimal (or most compact) loop nest in terms of the span of the trace covered by loop nests and the size of the compressed loop nest representation. However, the execution time was unacceptable for long traces; processing of a trace consisting of approximately 320K MPI calls took over 31 hours.

The results motivated us to develop a greedy procedure which intuitively works bottom up - it selectively identifies and reduces the shorter span inner loops and replaces them with a single symbol, before discovering the longer span outer loops. While the loop nest discovered by the greedy algorithm may not be optimal, it has well defined theoretical properties. A key analytical result is that the reduction of a shorter span inner loop as prescribed in the greedy algorithm can impact the discovery of a longer span outer loop only in the following way: if the optimal outer loop is L_o then a corresponding loop L_g will be identified despite the reduction of an inner loop. L_o and L_g have identical but possibly reordered trace symbols, but L_g may have up to 2 less loop iterations than L_o. Hence, the loop structure discovered by the greedy algorithm is *near optimal*. The theoretical basis for this procedure is treated in depth in [15].

The optimal and greedy loop nest discovery procedures were implemented and employed to discover the loop nests in the MPI traces of NAS benchmarks. The key results are listed in Table 3. As expected, the optimal algorithm discovered perfect loop nests as validated by direct observation.

The loop nests discovered by the greedy algorithm were, in fact, identical to the optimal loop nests except for a minor difference in the case of CG benchmark - the compressed trace had 21 symbols instead of 10 and the loop structure was slightly different. However, the time for greedy loop discovery was dramatically lower, down from 31 hours to 61 seconds for one trace. To the best of our knowledge, this is the first effort toward extracting complete loop nests from execution traces.

Name	Raw Trace Length	Compression Time Greedy (secs)	Compression Time Optimal (secs)	Major Loop Structure	Trace Span Covered by Loops	Compressed Trace Length	Compression Ratio
BT B/C	17106	8.91	311.18	$(85)^{200} = (13 + (4)^3 + ... + (4)^3)^{200}$	99.38%	44	388.77
SP B/C	26888	7.61	747.73	67^{400}	99.67%	89	302.11
*CG B/C	41954	8.48	2021.78	$(552)^{75} = ((21)^{26} + 6)^{75}$	98.68%	10	4195.4
MG B	8909	8.64	113.48	$(416)^{20}$	93.39%	590	15.1
MG C	10047	10.88	144.54	$(470)^{20}$	93.56%	648	15.5
LU B	203048	33.16	44204.82	$(812)^{249} = ((4)^{100} + (4)^{100} + 12)^{249}$	99.58%	63	3222.98
LU C	323048	61.9	113890.21	$(1292)^{249} = ((4)^{160} + (4)^{160} + 12)^{249}$	99.58%	63	5127.75

Table 3. Results for optimal and greedy compression procedures

4 Construction of performance skeletons

The final step in building a performance skeleton is converting a logicalized and compressed trace into an executable program that recreates the behavior represented in the trace. The trace at this stage consists of a loop nest with loop elements consisting of a series of symbols, each symbol representing an MPI Call or computation of a certain duration of time. The trace is converted to executable C code with the following basic steps:

- The loop nest in the trace is converted to a program loop nest with the number of iterations reduced to match the desired skeleton execution time.
- The collective and point-to-point communication calls in the trace are converted to MPI communication calls that operate on synthetic data. The point to point calls generate a global stencil communication pattern matching the application topology.
- The computation sections are replaced by synthetic computation code of equal duration.

Note that the procedure is simplistic with respect to reproducing computation. The instruction mix may be different and memory behavior is not reproduced. This is a limitation of the current work although memory skeletons have been investigated separately in [13].

A direct conversion of MPI trace symbols to MPI calls can result in executable code that may deadlock. The key issues in ensuring deadlock free communication in a skeleton program are as follows:

1. **Identifying *local* communication** Most MPI calls in a logical trace are matched: there is a *Recv* in the trace corresponding to every *Send*. We refer to these calls as *global* and their inclusion in the performance skeleton will lead to a stencil communication pattern across executing nodes. However, it is possible that some unmatched MPI Send/Recv calls may exist in a trace even when there is a dominant global communication pattern, i.e. there may

be *Send to WEST* in the trace but no corresponding *Send to EAST*. Such calls are labeled *local* and either removed or matched with synthetically generated calls. While local calls imply inaccuracy, they are rare in structured codes and necessary to ensure deadlock free execution. The procedure for marking communication calls as local or global is outlined in Figure 3. It is based on the basic deadlock free patterns of point to point communication which are 1) a non blocking Send/Recv with a matching Recv/Send before a corresponding Wait and 2) One or more blocking Send/Recv calls followed by matching Recv/Send calls. Note that in the latter case, the code generated for end nodes in the stencil is different from others, e.g. Send followed by Recv, when it is Recv followed by Send for all other nodes.

2. **Unbalanced global communication** Even when a pair of communication calls is matched, it may not be balanced, meaning an MPI Send/Receive and its corresponding MPI Receive/Send may not be equal in size. Analysis is employed to identify these and force a match, e.g., by using the median message size of a Send and Recv.

while *next-call*= First unmarked Send or Recv call in the code exists **do**
  if *next-call* is a non-blocking iSend (iRecv) **then**
    Let *match-wait* be the corresponding matching Wait call.
    Let *match-call* be the next matching Recv/iRecv (Send/iSend) in the code.
    if *match-call* is after *match-wait* or *match-wait* or *match-call* does not exist **then**
      Mark *next-call* as local communication.
    else
      Mark *next-call* and *match-call* as global communication.
    end if
  else
    [*next-call* is a blocking Send (Recv).]
    Let *match-call* be the next matching Recv/Irecv (Send/Isend) in code.
    if no *match-call* exists or there is a blocking Send or Recv between *next-call* and *match-call* **then**
      Mark *next-call* as local communication.
    else
      Mark *next-call* and *match-call* as global communication.
    end if
  end if
end while

Note: Matching calls have the same datatypes and match in terms of the directions in a communication pattern, e.g, logical East and West in a 2D torus.

Figure 3. Identification of Global and Local Send and Recv communication calls

5 Experiments and results

A prototype framework for automatic construction of performance skeletons has been implemented. Automatically generated skeletons were employed to estimate the execution time of cor-

responding applications in a variety of scenarios. Prediction accuracy was measured by comparing the predicted performance with actual application performance.

5.1 Skeleton construction and properties

Skeletons were constructed on "PGH201", which is a compute cluster composed of 10 Intel Xeon dual CPU 1.7 GHz machines with 100 Mbps network interfaces connected by a full crossbar Gigabit Switch. The execution was under MPICH 2.0 library. Experiments were conducted on 16-process class C NAS benchmarks. The methodology employed allows skeletons to be constructed to approximate a target skeleton execution time (or equivalently, a target ratio between application and skeleton execution times). However, there is a minimum execution time for a "good skeleton" which corresponds to the execution of a single iteration of the main execution loop. This also determines the maximum possible ratio between the application and skeleton execution times. For the experiments conducted, the objective was to build the longest running skeleton with execution time under one minute or a skeleton that executes for approximately 10% of the application execution time, whichever was lower. The execution times of NAS benchmarks and their skeletons are shown in Table 5.1. The table also shows the expected execution time ratio for the shortest running good skeleton, i.e., the maximum possible application to skeleton runtime ratio.

Benchmark	Execution Time(s)		Execution Time Ratio	
Name	Skeleton	Benchmark	Actual skeleton	Max possible
BT	45.6	1129.6	24.8	200
CG	40.3	607.6	15.1	75
MG	8.3	79.1	9.5	20
LU	39.1	637.4	16.3	249
SP	43.1	1069.2	24.8	400

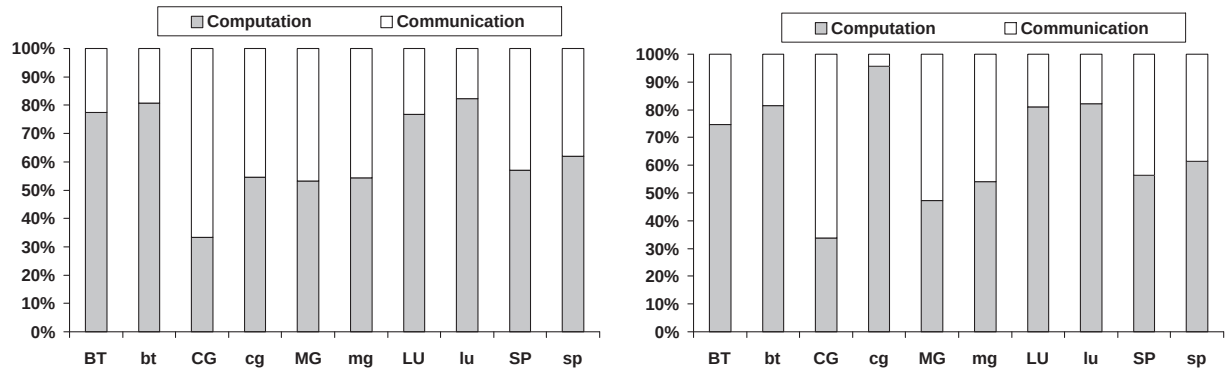
Table 4. Benchmark and skeleton execution times for 16 process class C NAS benchmarks

An application and the corresponding performance skeleton should have approximately the same percentage of time spent in computation and communication. These were measured for execution under MPICH 2.0 as well as execution under Open MPI library. The results are presented in Figure 4.

We note that the computation/communication time percentage is generally very close for benchmarks and corresponding skeletons. One exception is the CG benchmark, where the difference is especially striking for execution under Open MPI. We will present the performance results for other benchmarks first and then specifically analyze the CG benchmark.

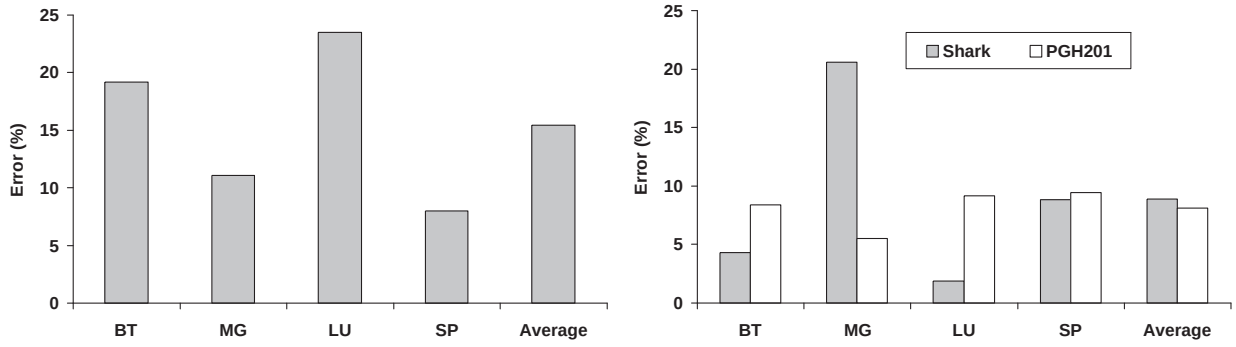
5.2 Prediction across MPI libraries and cluster architectures

Skeletons constructed with MPICH 2.0 on PGH201 cluster were employed to predict performance under Open MPI library and on a different cluster called "Shark" which is composed of 24 SUN X2100 nodes with 2.2 GHz dual core AMD Opteron processor and 2 GB main memory. All nodes are connected through 4x InfiniBand Network Interconnect and Gigabit Ethernet Network Interconnect. The results are plotted in Figure 5.



(a) MPICH 2.0 (b) OpenMPI

Figure 4. Computation/communication time percentage for benchmarks (uppercase) and skeletons (lowercase)



(a) Across architectures (PGH201 to Shark) (b) Across libraries (MPICH 2.0 to OpenMPI)

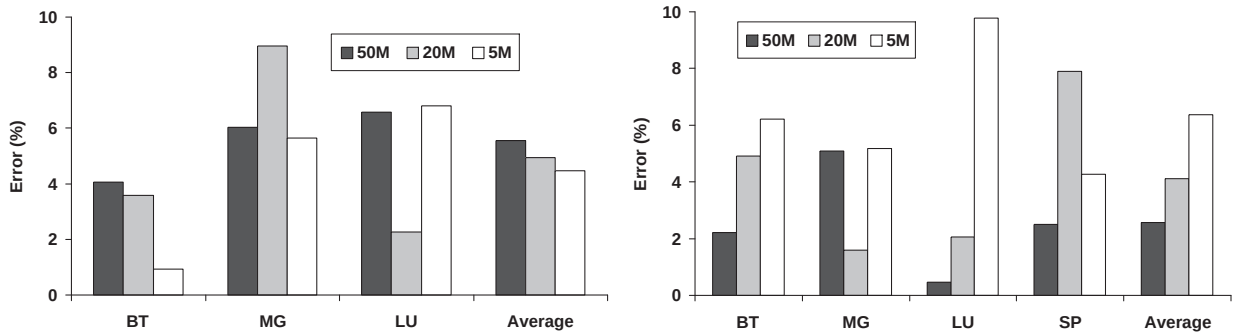
Figure 5. Prediction results across different MPI libraries/architectures

The prediction errors across the architectures average around 15%. The skeleton construction procedure employed makes no effort to reproduce the precise execution or memory behavior and only reproduces the execution times in skeletons with synthetic computation code. Hence, inaccuracy is expected across clusters with different processor and memory architectures. In the remainder of this paper, for validation purposes, the skeletons employed on Shark were "retuned" implying that the length of the computation blocks was adjusted to maintain the original ratio between reference skeleton and application execution.

Figure 5(b) shows the accuracy of performance predicted for OpenMPI with skeletons constructed with MPICH 2.0 on the two clusters. The errors are modest averaging below 10% for both clusters.

5.3 Prediction for bandwidth sharing

Figure 6 shows results from performance prediction with network sharing simulated by artificially reducing the available bandwidth to 50, 20, and 5Mbytes/sec with Linux *iproute2*. The results are presented for the older MPICH 1.2.6 MPI library, in addition to the MPICH 2.0 library. We consider the predictions to be excellent; the maximum prediction error is below 10% and the avarage prediction error varies between 2% and 6% for different scenarios. The results validate that the methodolgy employed models communication accurately,



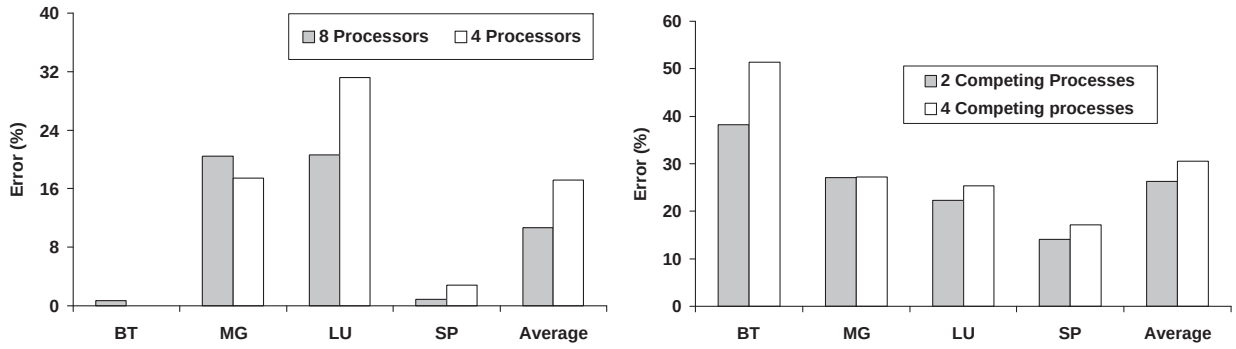
(a) MPICH 1.2.6 (b) MPICH 2.0

Figure 6. Prediction results with reduced bandwidth availability

5.4 Prediction for processor sharing

A set of experiments were conducted to estimate the accuracy of performance prediction with processor sharing. Each node has an independent CPU scheduler and no gang scheduling is employed. First, 16 process jobs were run on 8 and 4 processors. (The results are shown for Shark in this case as all cases cannot run on PGH 201 because of limited memory). The results in Figure 7(a) show that the average prediction error is in around 10% for 8 processors and 5% for 4 processors, but the maximum errors are over 20% for 8 processors and over 30% for 4 processors. Figure 7(b) plots the accuracy of performance prediction on 16 processors with 2 or 4 synthetic competing compute bound processes on each node. The prediction errors are rather high averaging around 30%.

These results point out the limitation of the methodology employed as it does not model computation, synchronization, or memory behavior accurately. Performance with independent CPU schedulers and sharing is sensitive to these factors. We speculate that the main reason for the relatively low accuracy in the above scenarios is that the skeleton construction procedure does not model the idle periods caused due to synchronization accurately and some of them are replaced by computations in skeletons. In the case of processor sharing, the idle periods will be effectively used by other competing processes making the performance as predicted by skeletons to be inaccurate. In this set of experiments, errors were the result of the application executing times being less than those predicted by skeleton execution.

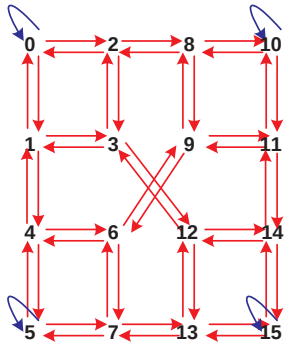


(a) Execution of 16 process job on 8/4 processors (Shark) (b) Execution with synthetic competing processes

Figure 7. Prediction results for processor sharing

5.5 CG benchmark

The prediction errors for the CG benchmark were significantly higher than the rest of the benchmark suite for most scenarios, and the results were not included in earlier charts in order to streamline the discussion. As examples, the prediction error for CG was around 4 times the average for other benchmarks for prediction across libraries and prediction with reduced bandwidth. CG benchmark is very communication intensive and it was observed that the performance of the CG benchmark was very sensitive to the placement of processes on nodes. The communication topology of CG benchmark is shown on the left in Figure 8. The table on the right shows the execution time for various mappings of processes to nodes. The execution time varies by a factor of two depending on the location of the processes. The skeleton construction procedure makes no effort to manage placement of processes on nodes, and the placement for the skeleton can be different from the placement of the application. Since the performance is placement sensitive, the framework cannot deliver meaningful results. No other benchmark examined exhibited such sensitivity to process placement.



Config	Node 1	Node 2	Node 3	Node 4	Time(s)
A	0,1,8,9	2,3,10,11	4,5,12,13	6,7,14,15	496
B	0,1,4,5	2,3,6,7	8,9,12,13	10,11,14,15	568
C	0,1,2,3	4,5,6,7	8,9,10,11	12,13,14,15	272

Figure 8. CG Topology and prediction results. The picture shows the process communication topology. The table shows the execution time of the benchmark for various placements of processes on nodes.

6 Conclusions and future work

This paper has presented and evaluated a framework for the construction of performance skeletons for message passing MPI programs from execution traces. The objective is prediction of application performance in scenarios where modeling of performance is challenging. A key innovation is that the performance skeletons developed are *coordinated*, i.e., a single SPMD skeleton program is generated for a family of process level traces. The paper describes customized procedures for logicalization and compression of execution traces that were developed for efficient and scalable generation of performance skeletons.

Results are presented to validate the prediction ability of performance skeletons in different scenarios. It is observed that the skeletons are very effective in predicting performance when dynamics of communication change, e.g., when the bandwidth is limited or a new communication library is deployed. However, the prediction power is limited in other scenarios where the computation dynamics change, e.g., when multiple processes must share a processor. This is not unexpected as the methodology captures the communication primitives precisely but attempts to recreate the periods of execution coarsely. In particular, the instruction level execution and memory behavior are not captured.

The fundamental limitation of this approach to performance prediction is that it is only applicable to structured applications with a repeating communication pattern for which a representative input data set is sufficient to capture the execution behavior. However, this covers a large class of scientific applications. The framework developed can be improved in several ways. The general computation and memory behavior and the distribution of computation sections across the computing processes can be captured and incorporated in skeletons. We believe that these enhancements will overcome the limitations that were pointed out in discussion of results.

Acknowledgement: This material is based upon work supported by the National Science Foundation under Grant No. ACI- 0234328 and Grant No. CNS-0410797

References

[1] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-level middleware for the grid. In *Supercomputing 2000*, pages 75–76, 2000.

[2] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[3] D. Kerbyson and K. Barker. Automatic identification of application communication patterns via templates. In *18th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, September 2005.

[4] C. Nevill-Manning, I. Witten, and D. Maulsby. Compression by induction of hierarchical grammars. In *Data Compression Conference*, pages 244–253, Snowbird, UT, 1994.

[5] C. G. Nevill-Manning and I. H. Witten. Sequitur. `http://SEQUITUR.info`.

[6] M. Noeth, F. Mueller, M. Schulz, and B. de Supinskii. Scalable compression and replay of communication traces in massively parallel environments. In *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, April 2007.

[7] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *7th IEEE International Symposium on High Performance Distributed Computing*, july 1998.

[8] A. Snavely, L. Carrington, and N. Wolter. A framework for performance modeling and prediction. In *Proceedings of Supercomputing 2002*, 2002.

[9] S. Sodhi and J. Subhlok. Automatic construction and evaluation of performance skeletons. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, April 2005.

[10] S. Sodhi, Q. Xu, and J. Subhlok. Performance prediction with skeletons. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2007. Accepted.

[11] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 163–172, Atlanta, GA, May 1999.

[12] T. Tabe and Q. Stout. The use of the MPI communication library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan, Nov 1999.

[13] A. Toomula and J. Subhlok. Replicating memory behavior for performance prediction. In *Proceedings of LCR 2004: The 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Houston, TX, October 2004. Published in the ACM Digital Library.

[14] Q. Xu, R. Prithivathi, J. Subhlok, and R. Zheng. Logicalization of MPI communication traces. Technical Report UH-CS-08-07, University of Houston, May 2008.

[15] Q. Xu and J. Subhlok. Efficient discovery of loop nests in communication traces of parallel programs. Technical Report UH-CS-08-08, University of Houston, May 2008.

[16] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.