

Construction and Evaluation of Coordinated Performance Skeletons

Qiang Xu and Jaspal Subhlok*

University of Houston
Department of Computer Science
Houston, TX 77204
jaspal@uh.edu

Abstract. Performance prediction is particularly challenging for dynamic environments that cannot be modeled well due to reasons such as resource sharing and foreign system components. The approach to performance prediction taken in this work 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 addressed in this paper is the automatic construction of performance skeletons for parallel MPI programs. The steps in the skeleton construction procedure 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.

1 Introduction

Traditional performance prediction and scheduling for distributed computing environments is based on modeling of application characteristics and execution environments. 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.

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

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

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 [1,2,3].

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. Validation experiments were 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 procedure for 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.

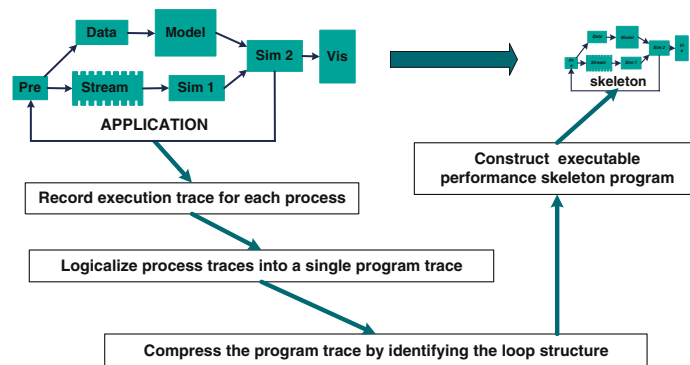


Fig. 1. Skeleton construction

The state of the art in performance prediction and scheduling for distributed computing environments is based on modeling of application characteristics and execution environments, with some example systems discussed in [4,5,6]. The research presented in this paper is fundamentally different in being based on synthetically generated executable code as the primary vehicle for performance prediction. Trace analysis has also been addressed in the context of trace replay tools such as the work in [7].

DIMEMAS [8] presents a promising approach to performance prediction in distributed environments based on replay of an execution trace in a simulated environment. The underlying problems addressed in skeleton construction have many aspects in common with on-the-fly trace compression methods, in particular the work presented in [9,10]. However, the approaches are algorithmically different; specifically the approach presented performs logicalization first and compresses only the logical trace.

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 [11] 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
...
<i>snd</i> (P_1, \dots)	<i>snd</i> (P_2, \dots)	<i>snd</i> (P_3, \dots)	<i>snd</i> (P_0, \dots)
<i>rcv</i> (P_3, \dots)	<i>rcv</i> (P_0, \dots)	<i>rcv</i> (P_1, \dots)	<i>rcv</i> (P_2, \dots)
...

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

Program

```

...
snd( $P_R, \dots$ )
rcv( $P_L, \dots$ )
...

```

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* 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 to record all message exchanges 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 1.

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

<i>PHYSICAL TRACE</i>	<i>LOGICAL TRACE</i>
.....
MPL_Isend(... 1, MPL_DOUBLE, 480, ...)	MPL_Isend(...EAST, MPL_DOUBLE, 480, ...)
MPL_Irecv(... 3, MPL_DOUBLE, 480, ...)	MPL_Irecv(...WEST, MPL_DOUBLE, 480, ...)
MPL_Wait() /* wait for Isend */	MPL_Wait() /* wait for Isend */
MPL_Wait() /* wait for Irecv */	MPL_Wait() /* wait for Irecv */
.....
MPL_Isend(... 4, MPL_DOUBLE, 480, ...)	MPL_Isend(...SOUTH, MPL_DOUBLE, 480, ...)
MPL_Irecv(...12, MPL_DOUBLE, 480, ...)	MPL_Irecv(...NORTH, MPL_DOUBLE, 480, ...)
MPL_Wait() /* wait for Isend */	MPL_Wait() /* wait for Isend */
MPL_Wait() /* wait for Irecv */	MPL_Wait() /* wait for Irecv */
.....
MPL_Isend(... 7, MPL_DOUBLE, 480, ...)	MPL_Isend(...SOUTHWEST, MPL_DOUBLE, 480, ...)
MPL_Irecv(...13, MPL_DOUBLE, 480, ...)	MPL_Irecv(...NORTHEAST, MPL_DOUBLE, 480, ...)
MPL_Wait() /* wait for Isend */	MPL_Wait() /* wait for Isend */
MPL_Wait() /* wait for Irecv */	MPL_Wait() /* wait for Irecv */
.....

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 [11].

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*

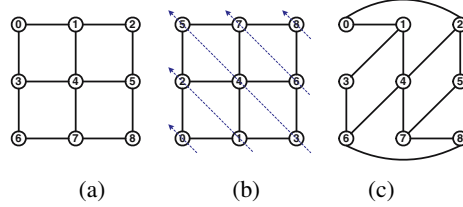


Fig. 2. 2D grid topology with row major and other numberings

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). A framework consisting of the following tests was employed for topology identification

1. **Simple tests:** All possible sizes of grid/tori/tree based on the number of processes N are identified with prime factoring. The number of edges and the degree ordered sequence of nodes for the given communication matrix are compared to instances of known topologies and those that do not match are eliminated.
2. **Graph Spectrum test:** Eigenvalue sets of isomorphic graphs are identical. The topologies whose eigenvalues do not match those of the communication matrix are eliminated.
3. **Graph Isomorphism test:** Graph isomorphism checking procedure is applied to establish that a given communication matrix exactly represents a specific topology. The *VF2* graph matching algorithm [13] was used.

The above steps are listed in increasing order of computation complexity and applied in that order as a decision tree. The simple tests and the graph spectrum test are employed to eliminate topologies that are provably not a match for the given communication matrix, but they cannot prove a match. Only the graph isomorphism test can establish an exact match.

Table 2 presents observations from the application of the logicalization procedure to selected NAS benchmarks running with 121/128 processes. The topologies that remain

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

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 torus 8×4×2×2 torus 8×4×4 torus	8×2×2×2 torus 8×4×2×2 torus 8×4×4 torus	8×2×2×2 torus 8×4×2×2 torus 8×4×4 torus	9035 (386KB)	7.33

as candidates after each test and the final established topology are listed in columns corresponding to the tests. The trace length and size as well as the trace processing times are also listed. The tracing overhead is low as only gross communication data, such as the number of messages and bytes exchanged are recorded and analyzed. It is clear that the simple tests are very effective in reducing the number of topologies that are possible match candidates which is the key to the overall efficiency of the framework. The logicalization process and performance characteristics are described in detail in [14].

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* [15] that constructs a dictionary of frequently occurring substrings and replaces each occurrence with a representative symbol, and *Sequitur* [16,17] 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 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 [9].

Our research took a novel approach to identifying the loop structure in a trace based on Crochemore’s algorithm [18] 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 $O(n \log n)$ 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 [19].

Table 3. Results for optimal and greedy compression procedures

Name	Raw Trace Length	Compression Time Greedy (secs)	Optimal (secs)	Major Loop Structure	Trace Span Covered by Loops	Compressed Trace Length	Compression Ratio
BT B/C	17106	8.91	311.18	$(85)^{200} = (13 + (4)^3 + \dots + (4)^3)^{200}$	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

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. The loop nest structure is represented in terms of the number of loop elements and the number of loop iterations. As illustration, the CG benchmark loop structure is denoted by $(552)^{75} = ((21)^{26} + 6)^{75}$ implying that there is an outer loop with 75 iterations enclosing 552 elements in the form of an inner loop with 21 elements iterated 26 times, and another 6 elements. 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.

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 without regard to the actual computation characteristics.

The procedure is simplistic in 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 [1].

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 leads to a stencil communication pattern across executing nodes. However, typically some unmatched MPI Send/Recv calls 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 ignored for code generation. An alternate approach is to match the local calls 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

```

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.

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

corresponding MPI Receive/Send may not be equal in size. Analysis is employed to identify these and force a match by using the median message size of a Send and Recv.

5 Experiments and Results

A framework for automatic construction of performance skeletons has been implemented. Automatically generated skeletons were employed to estimate the execution time of corresponding 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”, a compute cluster composed of 10 Intel Xeon dual CPU 1.7 GHz machines with 100 Mbps network interfaces. The execution was under MPICH 2.0 library. Results are reported for 16-process class C NAS benchmarks. Execution is on 16 dedicated processors, except when noted otherwise. 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. 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 reference execution times of NAS benchmarks and their skeletons are shown in Table 4.

Table 4. Benchmark and skeleton execution times for NAS benchmarks on 16 processors

Benchmark Name	Execution Time(s)	
	Skeleton	Benchmark
BT	45.6	1129.6
CG	40.3	607.6
MG	8.3	79.1
LU	39.1	637.4
SP	43.1	1069.2

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.

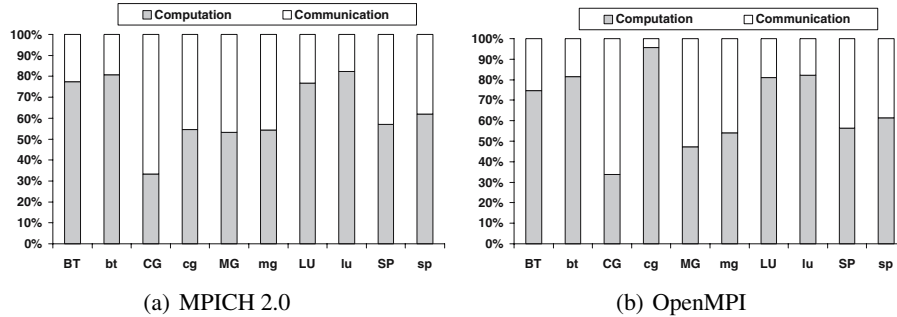


Fig. 4. Computation/communication time percentage for benchmarks (uppercase) and skeletons (lowercase)

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.

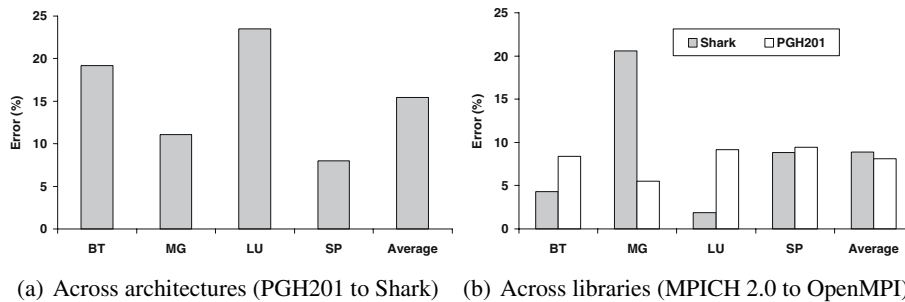


Fig. 5. Prediction results on 16 processors across MPI libraries/architectures

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 average prediction error varies between 2% and 6% for different scenarios. The results validate that the methodology employed models communication accurately.

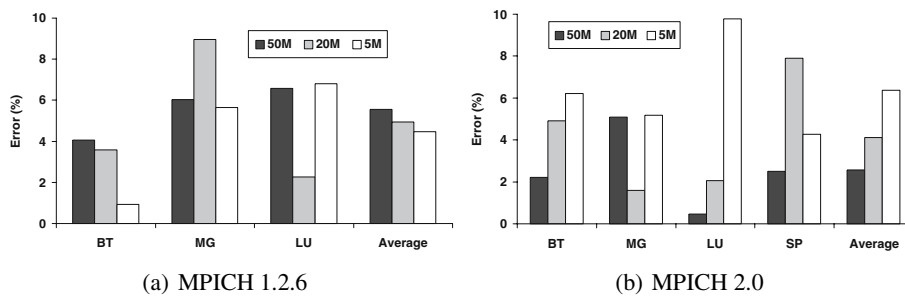


Fig. 6. Prediction results on 16 processors with reduced bandwidth availability

5.4 Prediction for Processor Sharing

A set of experiments was 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 and corresponding skeletons were run on 8 and 4 processors. (The results are shown for the Shark cluster in this case as all cases cannot run on the PGH 201 cluster because of limited memory). The results in Figure 7(a) show that the average prediction error is 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 execution times being less than those predicted by skeleton execution.

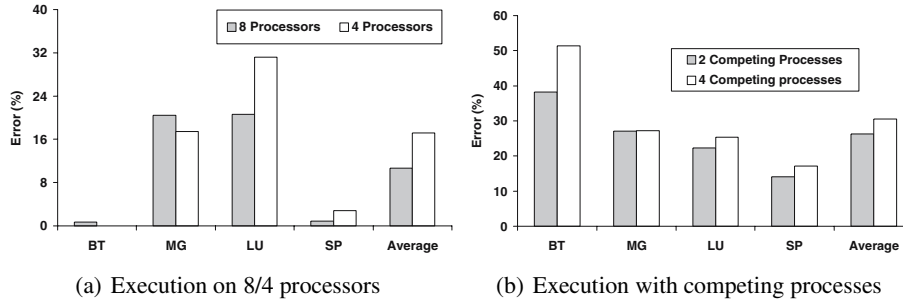


Fig. 7. Prediction results for execution of 16 process jobs with processor sharing

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 strong sensitivity to process placement.

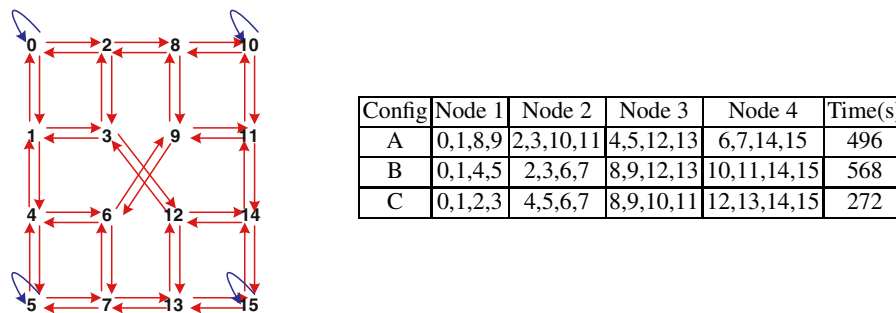


Fig. 8. CG Topology and prediction results. The picture shows the communication topology. The table shows the execution time of the benchmark for various placements of processes on nodes.

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 outlines the logicalization and compression procedures and lists related publications that contain the details.

Results presented 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 where the computation dynamics change, e.g., when multiple processes must share a processor. This is not entirely unexpected as the methodology captures the communication primitives precisely but attempts to recreate the periods of execution coarsely. If the computation regions in the skeleton were created to represent the instruction level execution and memory behavior, the approach would be significantly enhanced.

A basic limitation of this approach to performance prediction in its current form 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. Extending this approach to unstructured applications and building skeletons that can simply take the data size as a parameter and predict performance appropriately are significant challenges to be addressed in future research.

References

1. Toomula, A., Subhlok, J.: 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. The ACM Digital Library (2004)
2. Sodhi, S., Subhlok, J.: 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)
3. Sodhi, S., Xu, Q., Subhlok, J.: Performance prediction with skeletons. *Cluster Computing: The Journal of Networks, Software Tools and Applications* 11(2) (June 2008)
4. Casanova, H., Obertelli, G., Berman, F., Wolski, R.: The AppLeS Parameter Sweep Template: User-level middleware for the grid. In: *Supercomputing 2000*, pp. 75–76 (2000)
5. Raman, R., Livny, M., Solomon, M.: Matchmaking: Distributed resource management for high throughput computing. In: *7th IEEE International Symposium on High Performance Distributed Computing* (July 1998)
6. Snively, A., Carrington, L., Wolter, N.: A framework for performance modeling and prediction. In: *Proceedings of Supercomputing 2002* (2002)
7. Huband, S., McDonald, C.: A preliminary topological debugger for MPI programs. In: *1st International Symposium on Cluster Computing and the Grid (CCGRID 2001)* (2001)
8. Badia, R., Labarta, J., Gimenez, J., Escalé, F.: DIMEMAS: Predicting MPI applications behavior in Grid environments. In: *Workshop on Grid Applications and Programming Tools (GGF8)* (2003)

9. Noeth, M., Mueller, F., Schulz, M., de Supinskii, B.: 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)
10. Ratn, P., Mueller, F., de Supinski, B., Schulz, M.: Preserving time in large-scale communication traces. In: 22nd ACM International Conference on Supercomputing, June 2008, pp. 46–55 (2008)
11. Kerbyson, D., Barker, K.: Automatic identification of application communication patterns via templates. In: 18th International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV (September 2005)
12. Tabe, T., Stout, Q.: 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 (November 1999)
13. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: Performance evaluation of the VF graph matching algorithm. In: Proc. of the 10th ICIAP, vol. 2, pp. 1038–1041. IEEE Computer Society Press, Los Alamitos (1999)
14. Xu, Q., Prithivathi, R., Subhlok, J., Zheng, R.: Logicalization of MPI communication traces. Technical Report UH-CS-08-07, University of Houston (May 2008)
15. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23(3), 337–343 (1977)
16. Nevill-Manning, C., Witten, I., Malsby, D.: Compression by induction of hierarchical grammars. In: Data Compression Conference, Snowbird, UT, pp. 244–253 (1994)
17. Nevill-Manning, C.G., Witten, I.H.: Sequitur, <http://SEQUITUR.info>
18. Crochemore, M.: An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.* 12(5), 244–250 (1981)
19. Xu, Q., Subhlok, J.: Efficient discovery of loop nests in communication traces of parallel programs. Technical Report UH-CS-08-08, University of Houston (May 2008)