Automatic Construction of Coordinated Performance Skeletons

Jaspal Subhlok* Qiang Xu
University of Houston, Department of Computer Science, Houston, TX 77204

1 Overview

This research is motivated by the problem of performance prediction in dynamic and unpredictable environments where traditional performance modeling has limited success. Examples include shared execution environments and new software or system environments. The 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 goal is to build and validate a framework for automatic construction of performance skeletons for parallel MPI programs. Earlier work in this project constructed and validated memory and performance skeletons and explored their usage in distributed environments [3, 7, 4, 9, 5]. A new procedure for construction of performance skeletons that is being employed now is illustrated in Figure 1 and detailed in [8].

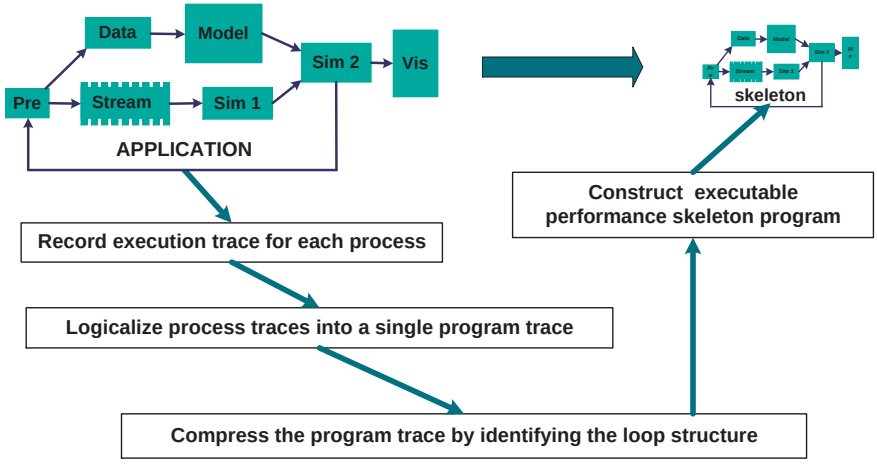

Figure 1: Skeleton construction

This paper outlines the new contributions of this work which are **1) trace logicalization** which is conversion of a suite of execution traces of an SPMD MPI program into a single logical trace, **2) trace compression** which involves identification of the loop structure inherent in the execution trace to capture the core execution behavior, and **(3) skeleton construction** and validation.

*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

2 Trace logicalization

As high performance scientific applications are generally SPMD programs, the traces for different processes are typically similar to each other and the communication is associated with a well defined global pattern. A study of DoD and DoE HPC codes at Los Alamos National Labs [2] and analysis of NAS benchmarks [6] 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 - in the same way as an SPMD program represents a family of processes that typically execute on different nodes. An example physical and logical trace are shown in Table 1.

Table 1: Logical and physical trace for 9-process BT benchmark

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 */*
....	....

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. Message passing patterns are 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.

The key algorithmic challenge is the identification of the application communication topology from the application communication matrix which represents the inter-process communication graph. The reason this is diffiult is 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). Our approach applies the following sequence of steps as a decision tree with simpler tests applied first for efficiency:

1. **Simple Tests:** Finding all possible sizes of grid/tori based on prime factors of the number of processes N, then matching the number of edges and the degree ordered sequence of nodes.
2. **Graph Spectrum Test:** Based on computing eigenvalues - eigenvalue sets of isomorphic graphs are identical.
3. **Isomorphism Test:** Applies graph isomorphism to establish a topology.

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.

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)	**CG stencil** **16×2×2×2 grid**	**CG stencil**	**CG 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.

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* that constructs a dictionary of frequently occuring substrings and replaces each occurence with a representative symbol, and *Sequitur* 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.

Our research took a novel approach to identifying the loop structure in a trace based on Crochemore's algorithm [1] 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 optimal and greedy loop nest discovery procedures were implemented and employed to discover the loop nests in the MPI traces of NAS benchmarks. The 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 towards extracting complete loop nests from execution traces.

Table 3: Results for optimal and greedy compression procedures

Name	Raw Trace Length	Compression Time		Major Loop Structure	Trace Span Covered by Loops	Compressed Trace Length	Compression Ratio
		Greedy (secs)	Optimal (secs)				
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

4 Construction and validation 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 skeleton execution time is controlled by reducing the number of execution iterations relative to the number in the compressed trace. Implementation of the communication in the performance skeleton is based on converting the MPI event symbols to real MPI communication calls. The key issues in ensuring deadlock free communication in a skelton program are the following:

1. **Identifying *local* communication** Most MPI calls in a logical trace are matched: there is a *Recv* in the trace corresponding to every *Send*. However, it is possible that some unmatched MPI Send/Recv calls may exist in a trace as a small volume of communication may not be associated with the global communication pattern. Such *local* calls are identified and removed. While this may cause inaccuracy, it is rare and necesssary to ensure deadlock free execution.
2. **Unbalanced global communication** Even if the logical trace contains only global communication, it may not be balanced, meaning an MPI Send/Receive and its corresponding MPI Receive/Send may not be matched in size or another parameter. Analysis is employed to identify these and ensure a match, e.g., by using the median message size of a Send and Recv.

A framework to build performance skeletons has been implemented and employed to predict performance in a number of scenarios as follows: 1) sharing of CPU with foreign processes, 2)

varying available communication bandwidth, 3) varying number of processors for the same number of processes, and 4) usage of a different communication library. The results are shown in Table 4.

Table 4: Summary of prediction errors under differenct scenarios

Name	CPU sharing with Competing Processes		Communication sharing with Differenct Avaialable Bandwidth				Reduced Number of Processors				Different MPI Lib OpenMPI-1.2.4	
	2 per node	4 per node	50M	20M	10M	5M	Eldorado		Shark		PGH201	Shark
							8	4	8	4		
BT	38.7%	51.7%	1.4%	4.0%	4.4%	5.3%	1.4%	2.5%	2.9%	n/a	9.1%	2.3%
SP	14.8%	17.7%	1.7%	7.0%	7.6%	3.4%	3.6%	3.1%	1.6%	0.3%	10.1%	6.1%
CG	17.1%	17.7%	8.0%	19.9%	17%	23.2%	4.0%	25.7%	3.6%	49.4%	21.7%	58.2%
MG	30.5%	30.7%	0.1%	3.2%	5.4%	9.7%	6.2%	10.5%	24.4%	11.5%	0.5%	14.5%
LU	23.7%	26.7%	1.3%	0.3%	2.9%	7.9%	22.2%	31.9%	15.7%	25.9%	10.7%	5.8%
Average	24.9%	28.9%	2.4%	6.9%	7.5%	9.9%	7.5%	14.8%	9.7%	21.8%	10.4%	17.4%

We observe that the error rates are generally low, except in the following cases. The errors are relatively high for CPU sharing and are generally high for the CG benchmark. The main reasons are that CPU sharing behavior is very sensitive to low level computation details which are not replicated accurately in skeletons. In the case of CG benchmark, we speculate that some of the synchronization behavior is not captured well. More analysis is presented in [8].

5 Conclusions

This paper identifies the major issues and presents new results in construction of performance skeletons to predict application performance. Overall the approach is very effective and some weaknesses are identified. Detailed results are available in referenced related publications.

References

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

[2] 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.

[3] S. Sodhi and J. Subhlok. Skeleton based performance prediction on shared networks. In *IEEE International Symposium on Cluster Computing and the Grid (Grids and Advanced Networks Workshop (GAN'04))*, Chicago, IL, April 2004.

[4] 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.

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

[6] 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.

[7] 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.

[8] Q. Xu. *Automatic Construction of Coordinated Performance Skeletons*. PhD thesis, University of Houston, August 2007.

[9] Q. Xu and J. Subhlok. Automatic clustering of grid nodes. In *Proceedings of the 6th IEEE/ACM Workshop on Grid Computing*, Seattle, WA, Nov 2005.