A Systematic Multi-step Methodology for Performance Analysis of Communication Traces of Distributed Applications based on Hierarchical Clustering

Gaby Aguilera [1], Patricia J. Teller [1], Michela Taufer [1], and Felix Wolf [2]

[1] University of Texas-El Paso
El Paso, TX 79968 USA
{maguilera, pteller, mtaufer}@utep.edu

[2] Forschungszentrum Jülich
52425 Jülich, Germany
f.wolf@fz-juelich.de

Abstract

Often parallel scientific applications are instrumented and traces are collected and analyzed to identify processes with performance problems or operations that cause delays in program execution. The execution of instrumented codes may generate large amounts of performance data, and the collection, storage, and analysis of such traces are time and space demanding. To address this problem, this paper presents an efficient, systematic, multi-step methodology, based on hierarchical clustering, for analysis of communication traces of parallel scientific applications. The methodology is used to discover potential communication performance problems of three applications: TRACE, REMO, and SWEEP3D.

1. Introduction

Today's most complex scientific applications require large numbers of calculations to solve problems. Usually, these problems exhibit some type of inherent parallelism by either repeating the same calculation on different data (data parallelism) or performing different calculations on the same data (functional parallelism). This parallelism can be exploited to arrive at solutions faster by developing parallel algorithms and applications to run on large-scale supercomputers. However, designing, programming, debugging, and tuning these applications present challenges, in part due to the level of complexity added by the number of processes that need to explicitly communicate, share data, and synchronize. In addition, it is difficult to track the execution of a program that is being executed simultaneously by multiple processes; this makes it difficult to hand optimize code or to find communication and synchronization bottlenecks.

In order to identify potential application performance problems, there exist several tools, e.g., [1, 2, 3], that provide programmers with the ability to collect pertinent data about the state of the system and program execution during runtime. Usually, a user can specify the computation-, communication-, and synchronization-related events to monitor and the areas of code to instrument. With respect to computation-related events, state-of-the-art processors include a set of on-chip performance counters that record the occurrence of events such as floating-point operations, cache misses, and branch mispredictions. The numbers of events that can be monitored by architectures vary; some support the monitoring of over 100. To aid in the collection of this data, PAPI (Performance Application Programming Interface) gives users a consistent interface to access the performance counter hardware found on major microprocessors [4].

Although instrumentation can facilitate the identification of performance problems and guide subsequent performance tuning efforts, it can generate large amounts of data. For example, even for a program that runs for a few minutes, hundreds of thousands of data points can be collected for each execution instance. Multiplying each instance by the number of events or metrics collected can result in an unmanageable data set, e.g., gigabytes of information. When data sets are so large, the following question arises: How can we discover where performance problems arise or what operations caused delays in the execution of a program? Due to the complexity of parallel programs and the multidimensionality of the collected performance data, it is natural to look at multivariate statistical analysis techniques for data analysis and knowledge discovery. This paper presents a systematic, multi-step methodology, based on a simple but effective statistical technique, i.e., hierarchical clustering of communication performance data, to discover potential communication bottlenecks in distributed applications. The methodology is used to analyze communication-related performance data collected during the execution of three applications. Specifically, this paper demonstrates how performance analysts can use this methodology to identify so called process pairs of interest, i.e., process pairs that are characterized by heavy communication, their physical locations within the distributed system, and potential performance problems.

The remainder of the paper is organized in five parts. Section 2 discusses related work. Sections 3 and 4 present the methodology. Section 5 demonstrates the use of this methodology and, finally, Section 6 concludes the paper and discusses future work.

2. Related Work

This work was inspired by a paper of Vetter [5] that explores the use of multivariate statistical analysis techniques on large sets of performance counter data to extract relevant performance information and give insights into application behavior. The techniques presented in the paper include hierarchical (dendrogram) and non-hierarchical (k-means) clustering, factor analysis, and principal component analysis. The paper shows that when these techniques are applied to hardware counter data, they can result in valuable application performance information and the removal of redundancy in counter data. Identifying redundancy in the collected data, i.e., identifying events that give similar results (or are highly correlated), can reduce the number of events to monitor and, thus, maximize the use of counters and reduce the dimensions of the data.

Vetter and McCracken also apply statistical analysis techniques to communication data to accurately identify communication operations that do not scale well with application problem size [6, 7]. Analyses of nine benchmark profiles indicate poorly scaling communication operations.

This paper extends the work of Vetter and McCracken by providing a systematic, multi-step methodology, based on hierarchical clustering, to analyze communication traces and identify potential communication performance problems.

3. Collection and Storage of Large-Scale Performance Data

Parallel applications developed to run on multiprocessor systems are, by nature, complex and are expected to run for long periods of time. If these applications are instrumented to collect performance data, the longer they run, the larger the size of the recorded performance data set. For example, if performance data is collected for a program executed on thousands of nodes for millions of time steps, and a large number of metrics is collected, the size of the file that stores the data can explode — it can be many gigabytes in size. Accordingly, three important issues arise: efficient data collection, efficient data storage, and efficient access and analysis of stored data. This section briefly describes the various data collection techniques provided by different tools and the data collection and storage techniques used in the work presented in this paper.

3.1. Data Collection

There are a number of tools that collect data to identify potential performance problems present in parallel programs. The data to be collected is identified by manual or automatic instrumentation. The former is accomplished by manually inserting directives in code locations of interest [3]. The latter can be accomplished by modifying a Makefile to enable a tool to automatically preprocess the code, inserting directives automatically before compilation [2, 3]. Alternatively, users can opt for a completely automated and dynamic process, where the executable does not have to be recompiled and instrumentation occurs at execution time [1].

These tools differ not only in the approach adopted for program instrumentation, but also with respect to the information they collect and whether or not the user specifies the types of data to collect. Some tools capture trace files, possibly with time stamps, that record the history of events that occur in the system during program execution. One example of this is a communication trace file that contains communication event information and, when supported and specified, hardware performance counter data [8, 9]. Other tools produce profiles that record the amount of time spent in each function, number of times each function is called, and, if available, other metrics, such as event counts, supplied by hardware counters [1, 3]. In terms of analysis, a tool can provide offline or postmortem inspection and analysis of trace files and/or profiles, or it can provide performance information as the program executes, In the latter case, it might even allow the user to stop execution to respecify the data to be collected or the code sections to be monitored [1].

This paper utilizes communication traces of parallel programs. The tool used for data collection is KOJAK (Kit for Objective Judgment and Knowledge-based Detection of Performance Bottlenecks), developed by Mohr and Wolf [2]. KOJAK is an automatic performance analyzer that can collect and analyze performance data from MPI, OpenMP, and hybrid parallel programs. The latter use OpenMP directives for shared-memory code segments and MPI library calls for message-passing segments. KOJAK uses OPARI (OpenMP Pragma And Region Instrumentor), a source-to-source translation tool, the PMPI library, and TAU (Tuning and Analysis Utilities) to instrument OpenMP directives, MPI functions, and user-defined functions, respectively. Once the source code is instrumented, the compile and link commands are modified to include all necessary libraries. If the user links to the PAPI library, hardware counter performance data also is collected. While the application is executed, an EPILOG trace file is generated, which contains information about MPI-communication, OpenMP, and, if specified, hardware-counter events. An API (Application Programming Interface), EARL (Event Analysis and Recognition Library), can be used to facilitate access to data stored in EPILOG format trace files.

3.2. Data Storage

The use of the EPILOG trace file format and EARL result in efficient storage and access of trace files [11, 12]. The binary trace data format is designed to store performance data associated with the execution of parallel applications [9]. It guarantees that each event is mapped to a physical system location defined as a quadruplet {machine, node, process, thread}. The format also supports collection of information associated with source code, such as line numbers and file names, call sites, and, if available, event counts (from hardware counters). The trace format defines the structure of the trace file, which is composed of three sections: header, definition records, and event records. The header identifies the byte order and trace file format version, while the definition and event records comprise all the performance data. Definition records store information that describes machines, nodes, processes, threads, source-code entities, performance metrics, and MPI communicators. Event records store performance data that vary with the type of event. Two fundamental pieces of data are stored in each record: a location identifier and a time stamp. Location identifiers map occurrences of events to physical locations, while time stamps map events to specific points in time during application execution.

The EARL API facilitates access to all types of event and definition records described in the EPILOG trace file format. EARL gives users the flexibility to access events and manage them as objects so that events have associated attributes. Most importantly, EARL gives users the ability to: randomly access events in a trace, link pairs of related events, such as send/receive communication event pairs, and access the execution state at the time of a specific event.

4. Methodology for Analysis of Communication Data

This section describes a multi-step methodology for analysis of communication data guided by hierarchical clustering. The method is used in Section 5 to analyze the communication traces of three real-world applications.

4.1. Hierarchical Clustering

Hierarchical clustering consists of partitioning a data set into subsets or clusters, so that the data in each subset share a common trait. This is done by using a distance measure to quantify similarity or proximity in data, in this case, communication data.

Typical statistical methods, such as maximum, minimum, and mean, can be used to analyze data, however, when dealing with large amounts of data, these methods are not effective alone. To compliment and increase the effectiveness of these methods, hierarchical clustering can be used to decrease the size of the data set to be analyzed. Hierarchical clustering algorithms form successive clusters using previously formed clusters. In the case of a communication trace, to be analyzed for performance problems, hierarchical clustering can be used to reduce the size of the data to be analyzed by identifying the data associated with processes that exhibit heavy communication. Subsequently, the performance analyst can focus on this data subset, performing a more in-depth analysis of it to identify code that may present performance problems.

To effectively use this method of analysis for this purpose, the question that must be answered is: What metric should be used to evaluate the similarity or distance between processes so that meaningful clusters (of performance data) are formed? A logical metric to use to identify processes that are heavy communicators is the number of bytes exchanged between process pairs. Accordingly, the first attempt at clustering uses this metric, which is quantified by calculating the aggregate number of bytes exchanged between processes. Preliminary results indicate that most process pairs exchange similar numbers of bytes. This is not surprising for two reasons. First, since one of the goals of parallelizing applications is to distribute the workload among a group of processes, the programmer endeavors to evenly distribute the data among the processes. Second, the applications used in this case study have been highly optimized and, as a result, likely exhibit good load balancing.

Given that the aggregate number of bytes exchanged between processes does not differentiate pairs of processes, the second attempt at clustering uses communication time as the differentiating metric for two reasons. First, if two processes communicate frequently, then their aggregate communication times should reflect their frequencies of communication. Second, long communication times between processes that reside in different nodes should be reflected in aggregate communication times larger than those associated with communicating processes in the same node – this information should be highlighted in the analysis since it can help the analyst identify communication patterns that should be modified if possible. Note that in using the aggregate communication time metric, the overall outcome is not affected by one long message but is affected by a series of long messages – this also can help the analyst decide if communication patterns should be changed.

To turn this metric into a distance function that can be used to identify heavily communicating process pairs, the extracted communication data is processed to attain the aggregate communication time for each pair of communicating processes. Because the objective of this first step in the analysis is to group pairs of processes that communicate heavily, the inverse of the aggregate communication time is used as the distance function. As shown in Figure 1, this results in the distance function, *D(a,b)*, for process pair *(a,b)*, where $T(a\ b)_i$ is the time for the i^{th} communication between processes *a* and *b*.

$$\boxed{D(a,b) = \frac{1}{\sum T(a,b)_i}}$$

Figure 1: Distance function.

After the distance functions are calculated for all communicating process pairs and stored in a distance matrix, a hierarchical clustering algorithm is applied to identify heavily communicating processes based on their aggregate communication times. The results can guide the analysis of the communication data. The next section explains in detail the steps followed to perform this analysis.

4.2. Multi-Step Methodology

The methodology is based on a sequence of steps, outlined in Figure 2, implemented by software components written in C++, Perl, and MATLAB. The first step, S1, extracts communication information from an EPILOG format trace file. Trace.C, the code that does this, is a C++ program that uses the EARL API. The second step, S2, summarizes in a text file the extracted communication information (see Table 1 for an example), i.e., the aggregate number of bytes exchanged between process pairs and the related aggregate communication time. Step S3 forms a distance matrix of size *p*-by-*p*, where *p* is the number of processes in the multiprocessor system, using the inverse of the aggregate communication time as the distance function. The distance matrix is the input to the fourth step, S4. This step uses the hierarchical clustering utility in MATLAB to form clusters of the performance data and represent them graphically to identify heavily communicating processes (see Figure 3, 4, and 5 for examples). This information is the input to the final step, S5, comprised of several components, which can: (1) determine if pairs of processors reside on the same node; (2) attain call path information for execution points of interest; and (3) analyze small versus large messages based on message summary data.

S1: Extract communication data from trace file
S2: Summarize extracted communication information
S3: Create distance matrix using data from previous step
S4: Perform hierarchical clustering
S5: Identify process pairs of interest and perform a more in-depth analysis

Figure 2: Multi-step methodology for trace analysis.

5. Use of Methodology

To show the effectiveness of the methodology described in Section 4, this section analyzes the results of using the methodology to analyze traces from three real-world applications, TRACE, REMO, and SWEEP3D.

5.1. Applications

Three real-world applications were used to generate trace files: TRACE, REMO, and SWEEP3D. TRACE [12] was developed at the Research Center in Jülich, Germany. It simulates the subsurface water flow in variably saturated porous media. TRACE uses message passing to communicate. The trace file used in this paper was generated by executing this application on four SMP nodes with four processes, one per processor of a node [13].

REMO [14] is a weather forecast application of the German climate computer center DKRZ (Deutsches

Klima Rechenzentrum). It implements a hydrostatic limited area model, which is based on the German/European weather forecast model of the German Meteorological Services (Deutscher Wetterdienst or DWD). REMO is a hybrid MPI/OpenMP application. The REMO traces used in this paper were taken from an early experimental MPI/OpenMP (i.e., hybrid) version of the production code [13]. As for TRACE, the trace file for REMO was generated by executing the application on four SMP nodes with four processes, one per processor of a node [13].

The benchmark code SWEEP3D [15] represents the core of a real ASCI (Accelerated Strategic Computing Initiative) application. It solves a 1-group time-independent discrete ordinates 3D Cartesian geometry neutron transport problem. SWEEP3D is a hybrid MPI/OpenMP application. Unlike the others, this application was executed by 64 processes on a 64-processor (eight-node) cluster at the High Performance Computing Center of the University of Houston [16].

5.2. Results of Analysis

The multi-step methodology presented in Section 4.2 allows users to determine **physical locations of processes and threads**, i.e., whether they are located on the same node. Table 1 shows the locations of the processes and threads on four nodes for the TRACE and REMO applications, and on seven nodes for SWEEP3D. This information is generated in Step S2 of the methodology. Additionally, the aggregate communication time between any two processes is computed. This information is used in Step S3 to create a distance matrix, **identify clusters of communicating process pairs**, and help identify processes that may be associated with communication bottlenecks. The clustering is generated by using the distance matrix *D(a,b)*, where *a* and *b* are processes, built following the procedure presented in Section 4.2.

Figure 3 shows the resulting clusters of communicating process pairs for the TRACE application. Figures 4 and 5 show the same information for the REMO and SWEEP3D applications, respectively. The dendrograms in the three figures quantify the closeness of process pairs, which is directly related to the amount of time two processes spend communicating: the larger the aggregate communication time, the smaller the distance and the more the processes belong in a communication cluster. The dendrogram in Figure 3 indicates that for the TRACE application the process pairs of interest are (2, 11) and (0, 8) (see pairs in rectangles). For the REMO application, note that process pair (3, 7) is the heaviest communicating pair. For SWEEP3D, the results of the clustering, pictured in Figure 5, show several process pairs of interest and two major clusters: processes 50, 56, 57, 59, 60, 61, 62, and 63 form one cluster, while the other cluster is formed by the remaining processes.

Process pairs of interest, identified through this data clustering, are those that should be further analyzed to identify possible reasons for their higher communication times. This final step of the proposed methodology allows users to further investigate and visualize: (1) execution time versus message size, (2) execution time versus communication time, and (3) message size versus communication time for process pairs of interest.

TRACE. The plot representation of **execution time versus message size** for the process pair of interest (2, 11) in TRACE gives insight into when messages of a particular size are transmitted. As Figure 6 shows, for this process pair there are two message sizes: 128 bytes and 3,456 bytes, which are homogeneously distributed during the entire execution time. This observation suggests an analysis to determine if it is possible to decrease communication time by packaging sets of small messages into larger ones.

	TRACE			**REMO**			**SWEEP3D**		
# events	19712210			11063530			3255168		
#machines	1			1			1		
# nodes	4			4			7		
#processes	16			4			64		
# threads	16			16			64		
Machine, Node, Process, and Thread Identification	**Node**	**Process**	**Thread**	**Node**	**Process**	**Thread**	**Node**	**Process**	**Thread**
	0	0	0	0	0	0	0	0-20	0-20
		1	1			1			
		2	2			2	1	21-32	21-32
		3	3			3			
	1	4	4	1	1	4	2	33-40	33-40
		5	5			5			
		6	6			6	3	41-48	41-48
		7	7			7			
	2	8	8	2	2	8	4	49-56	49-56
		9	9			9			
		10	10			10	5	57-60	57-60
		11	11			11			
	3	12	12	3	3	12	6	61-63	61-63
		13	13			13			
		14	14			14			
		15	15			15			

Table 1: Location of nodes, processes, and threads using the multi-step methodology.

Graphing message size versus communication time gives a summary of the different communication times associated with each message size. One would expect that for a particular process pair the communication time for a fixed message size would not vary much. Although only two message sizes are exchanged between processes 2 and 11, there is a variety of communication times associated with each message size. This could be due to contention for system resources or initialization overhead. For example, as shown in Figure 7, which plots **communication time against message size**, the largest communication time of all the messages transmitted between processes 2 and 11 is 0.05 seconds, and this is associated with the first message exchanged between the two processes.

REMO. Figure 8 depicts the different sizes of messages exchanged between the pair of interest (3, 7) during the execution of the REMO application. There are 11 different message sizes exchanged between processes 3 and 7. By inspecting this figure, one can speculate that the gaps in the graph separate the initialization, computation, and finalization phases of the program. Figure 9 shows the different communication times associated with each message size. From this figure, it can be concluded that communication time does not depend on message size. In fact, small and large messages have similar communication times. For instance, some 7,040-byte messages take about the same time as a 164,032-byte message. This graph shows that the 6,864-byte, 70,400-byte, and 140,800-byte messages have the most consistent communication times, while the 7,040-byte, 60,352-byte, and 164,032-byte messages have communication times that fluctuate the most. This behavior deserves more investigation; program modifications targeted at reducing these communication times could enhance performance.

SWEEP3D. The communication trace data associated with two heavy communicating process pairs, (21, 22) and (52, 60), is analyzed for SWEEP3D in Figure 10. This application uses only one message size to exchange data between processes; therefore, a message size comparison analysis is not meaningful. However, as shown below, an analysis of **execution time vs. communication time** can provide important insights.

For example, for the process pair of interest (21, 22), which communicates within the same node, the execution time versus communication time data allows the performance analyst to see that large communication times for this process pair occur during program execution. In contrast, as shown in Figure 11, for the process pair (52, 60), which communicates between two different nodes, communication times are comparatively short. The hierarchical clustering applied to the SWEEP3D trace data helps to identify such anomalies, and the methodology facilitates investigation of these apparent performance inconsistencies by identifying call paths that initiated the related communication events and the sizes of the corresponding messages. This investigation can help focus tuning efforts on the code regions that potentially exhibit performance problems.

From these examples, it can be seen that the multi-step methodology helps users answer questions such as: Which message sizes should be used when communicating between process pairs, i.e., should multiple smaller messages or one large message be transmitted? Using the methodology, one can figure out when during a program execution certain message sizes are used, i.e. in the beginning, middle, or end of the program.

6. Conclusions

This paper indicates that hierarchical clustering of communication performance data may be a method that can facilitate the processing of large amounts of performance data collected during a parallel program's execution; it deserves further investigation. Hierarchical clustering is a key component of the systematic, multi-step methodology that was presented and used in this paper to investigate the communication performance of three real-world applications, TRACE, REMO, and SWEEP3D. In particular, the methodology allows users to identify processes that might experience communication problems, the physical locations of the processes, the sizes of communications in which they are involved, and the associated communication times.

With peta-scale systems looming in the near future, statistical techniques such as these likely will play an important role in performance analyses that will help zero-in on communication and computation performance problems or bottlenecks. Accordingly, our current and future research includes further investigation of statistical techniques, including factor analysis and principal components analysis, for these purposes.

Acknowledgements

We wish to thank the Department of Defense, in particular, the PET program, and IBM, in the form of an IBM SUR grant, for support of this research. This work was carried out in the PET Computational Environments functional area under Department of Defense Contract No. N62306-01-D-7110/0070.

References

[1] B. Miller, et al. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11) 37–46, Nov. 1995.

[2] B. Mohr and F. Wolf. KOJAK - A tool set for automatic performance analysis of parallel programs. In *Proceedings of the 2003 Euro-Par Conference*, pages 1301–1304.

[3] B. Mohr, D. Brown, and A. Malony. TAU: a portable parallel program analysis environment for pC++: a portable data-parallel programming system for scalable parallel computers. In *Proceedings of CONPAR 94 - VAPP VI*, Sept. 1994.

[4] M. Maxwell, P. Teller, L. Salayandia, and S. Moore. Accuracy of performance monitoring hardware. In *Proceedings of the LACSI Symposium*, Oct. 2002.

[5] D. Ahn and J. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of SC2002*, Nov. 2002.

[6] J. Vetter and M. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2001.

[7] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[8] F. Wolf and B. Mohr. EPILOG binary trace-data format. Technical Report FZJZAM-IB-2004-06, Forschungszentrum Jülich, May 2004.

[9] Structured trace format. http://www.intel.com/software/products/cluster/tcollector/overview.htm.

[10] F. Wolf. EARL - API documentation. ICL Technical Report, ICL-UT-04-03, University of Tennessee-Knoxville, Oct. 2004.

[11] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient pattern search in large traces through successive refinement. In *Proceedings of the European Conference on Parallel Computing (Euro-Par)*, Aug.-Sept. 2004.

[12] Forschungszentrum Jülich. Solute transport in heterogeneous soil-aquifer systems. http://www.kfa-juelich.de/icg/icg4/Groups/Pollutgeosys/trace_e.html.

[13] F. Wolf. Automatic performance analysis on parallel computers with SMP nodes. Ph.D. dissertation, RWTH Aachen, Forschungszentrum Jülich, ISBN 3-00-010003-2, http://www.fz-juelich.de/nic-series/volume17/, Feb. 2003.

[14] E T. Diehl and V. Geulzow. Performance of the parallelized regional climate model REMO. In *Proceedings of the Eighth ECMWF (European Centre for Medium-Range Weather Forecasts) Workshop on the Use of Parallel Processors in Meteorology*, Nov. 1998, pages 181–191.

[15] Accelerated Strategic Computing Initiative [ASCI], The ASCI SWEEP3D benchmark code. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html.

[16] University of Houston. Galaxy cluster at the University of Houston. http://www.suncoe.uh.edu/galaxy/.

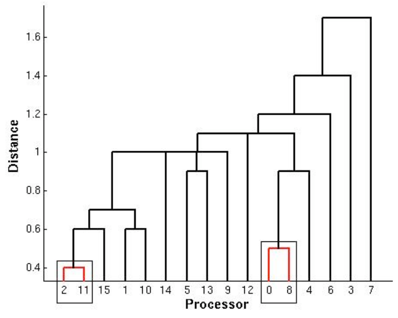

Figure 3: Clusters of process pairs for TRACE.

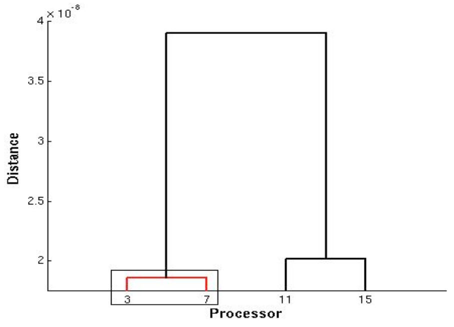

Figure 4: Clusters of process pairs for REMO.

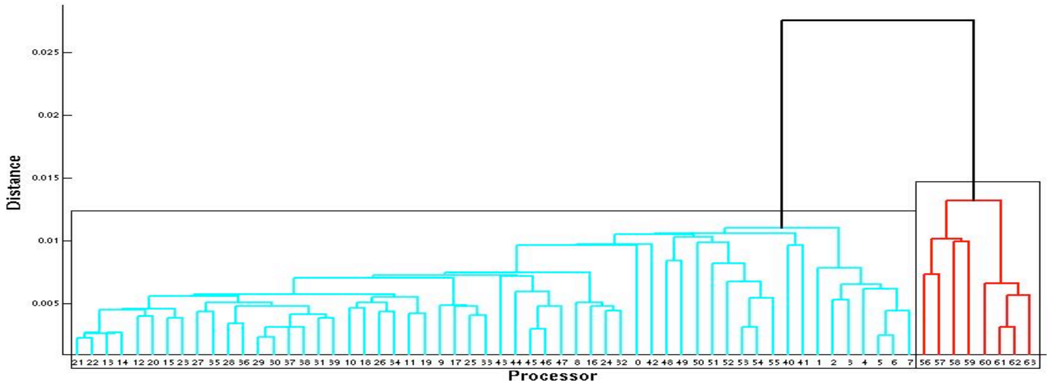

Figure 5: Clusters of process pairs for SWEEP3D.

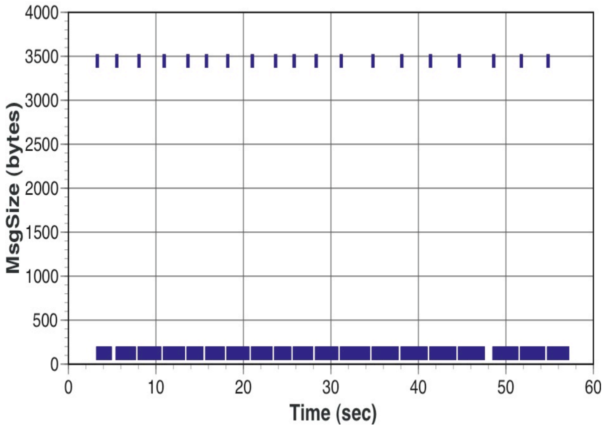

Figure 6: Execution time vs. message size for process pair (2, 11) in TRACE.

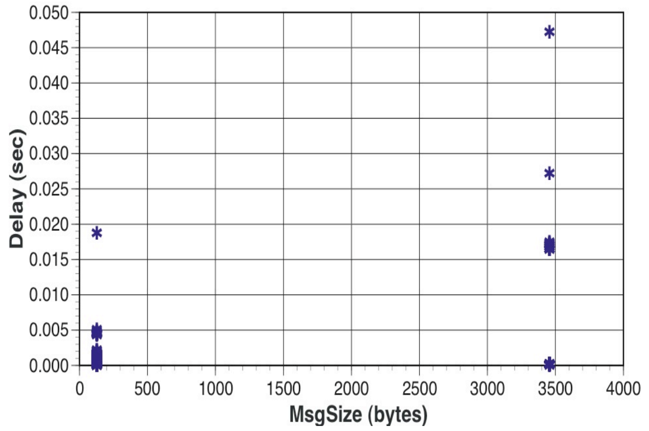

Figure 7: Communication time vs. message size for process pair (2, 11) in TRACE.

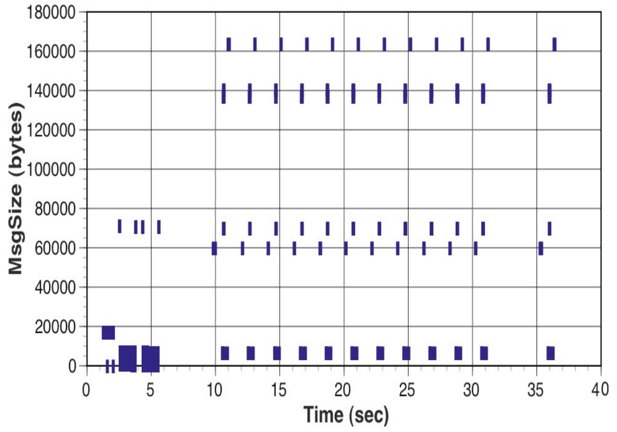

Figure 8: Execution time vs. message size for process pair (3, 7) in REMO.

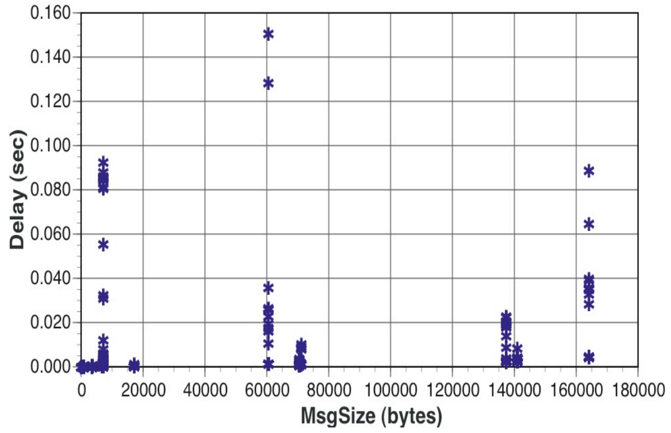

Figure 9: Communication time vs. message size for process pair (3, 7) in REMO.

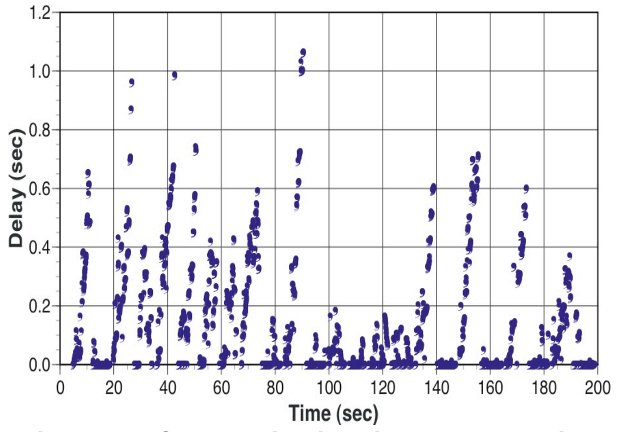

Figure 10: Communication time vs. execution time for process pair (21, 22) in SWEEP3D.

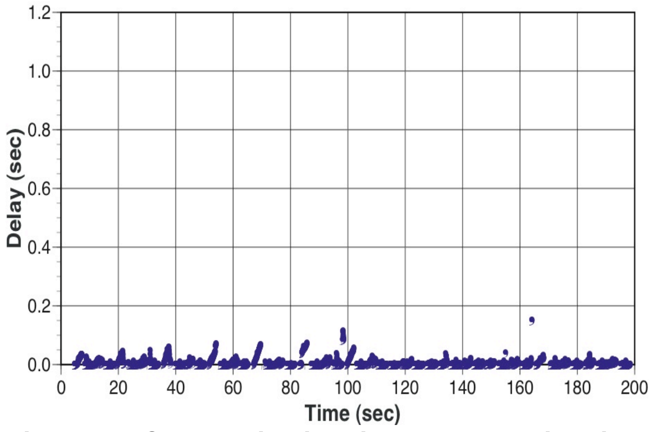

Figure 11: Communication time vs. execution time for process pair (52, 60) in SWEEP3D.