Capturing and Visualizing Event Flow Graphs of MPI Applications

Karl Fürlinger[1] and David Skinner[2]

[1] Computer Science Division, EECS Department
University of California at Berkeley
Soda Hall 593, Berkeley CA 94720, U.S.A.
fuerling@eecs.berkeley.edu
[2] Lawrence Berkeley National Lab
Berkeley, California
deskinner@lbl.gov

Abstract. A high-level understanding of how an application executes and which performance characteristics it exhibits is essential in many areas of high performance computing, such as application optimization, hardware development, and system procurement.
Tools are needed to help users in uncovering the application characteristics, but current approaches are unsuitable to help develop a structured understanding of program execution akin to flow charts. Profiling tools are efficient in terms of overheads but their way of recording performance data discards temporal information. Tracing preserves all the temporal information but distilling the essential high level structures, such as initialization and iteration phases can be challenging and cumbersome.
We present a technique that extends an existing profiling tool to capture event flow graphs of MPI applications. Event flow graphs try to strike a balance between the abundance of data contained in full traces and the concise information profiling tools can deliver with low overheads.
We describe our technique for efficiently gathering an event flow graph for each process of an MPI application and for combining these graphs into a single application-level flow graph. We explore ways to reduce the complexity of the graphs by collapsing nodes in a step-by-step fashion and present techniques to explore flow graphs interactively.

1 Introduction

Understanding performance characteristics of applications at a high level is essential in many diverse areas of high performance computing. Application developers, hardware engineers, or computing center support and procurement experts use tools to establish that the application uses the available resources efficiently or if there is potential for improvement.

Many techniques made available by current tools are insufficient for getting a high-level understanding of the "execution flow" of an application. Most performance tools can be categorized into either profiling or tracing. Profiling tools are

efficient in terms of overheads but their way of recording performance data discards temporal information. Tracing preserves all the temporal information but uncovering the essential high level structures, such as initialization and iteration phases can be challenging and cumbersome.

We present a technique that extends an existing profiling tool to capture event flow graphs of MPI applications with very low overhead. Event flow graphs try to strike a balance between the abundance of data contained in full traces and the concise information profiling tools can deliver with low overheads. The graphs are similar in concept to flow charts used to describe algorithms and design software systems.

We describe our technique for efficiently gathering an event flow graph for each process of an MPI application and for combining multiple graphs into a single application-level flow graph. We explore ways to reduce the complexity of the graphs by collapsing nodes in a step-by-step fashion and present techniques to explore flow graphs interactively.

The rest of this paper is organized as follows: In Sect. 2 we give a short overview of the integrated performance monitoring (IPM) tool that we extended to capture event flow graphs. In Sect. 3 we describe our approach to recording the flow graphs in MPI applications, and in Sect. 4 we describe techniques for the interactive visualization and exploration of the graphs and apply the tool to some example applications. In Sect. 5 we survey related work and in Sect. 6 we conclude and discuss areas for future work.

2 Application Profiling and Workload Characterization with IPM

IPM is a profiling and workload characterization tool for MPI applications. IPM achieves its goal of minimizing monitoring overhead by recording performance data in a fixed-size hash table resident in memory and carefully optimizing time-critical operations. At the same time, IPM offers very detailed and user-centric performance metrics. IPM's performance data is delivered as an XML file that can subsequently be used to generate HTML pages, avoiding the need for special graphical user interfaces. Pairwise communication volume between processes, communication time breakdown across ranks, MPI operation timings, and MPI message sizes (buffer lengths) are some of IPM's most widely used features. IPM is available from `http://ipm-hpc.sourceforge.net` for download and is distributed under the LGPL license.

3 Recording Event Flow Graphs of MPI applications

We assume the following general model of performance monitoring for MPI applications: An MPI application is composed of n *processes* each identified by an integer in $[0, \ldots, n-1]$, its *rank*. A set of events $E_i \subseteq E$ happen in each process i. We do not further formally specify what the events are, but we assume they

occur at a certain time and have duration. Each event e has an associated *signature* $\sigma(e) \in S$ which captures the characteristics we are interested in. $\sigma : E \mapsto S$ is the signature function. Concretely we think of a signature $\sigma(e)$ as a k-tuple $\sigma(e) = (\sigma^1(e), \sigma^2(e), \ldots, \sigma^k(e))$, where each $\sigma^j()$ is a signature *component*. Useful components of signature functions are listed in Fig. 1.

Signature component	**Signature function**	**Data type**	**Typical Size (#bits)**
Wallclock time	$time(e)$	floating point	32/64
Sequence number	$seq(e)$	integer	32
Type of MPI call	$call(e)$	integer	8
Data size	$size(e)$	integer	32
Data address	$address(e)$	integer	64
Own rank	$rank(e)$	integer	32
Partner rank	$partner(e)$	integer	32
Callsite ID	$csite(e)$	integer	16
Program region	$region(e)$	integer	8

Fig. 1. Components of an event signature function.

Our goal for performance observation is to get an event inventory of an application (i.e., understand the events that happened and their characteristics) by associating performance data (number of occurrences, statistics on the duration) with event signatures. If the signature includes $time()$ this essentially models tracing; if it does not we have a model for profiling.

IPM is a profiling tool and for efficiency reasons we would like to keep the signature space much smaller than the event space ($|E| >> |S|$). In this case the signature function is not injective and performance data can be envisioned as a table indexed by the signature, with a number of columns for the statistics we are interested in. In IPM we implement this indexing using a hash table resident in memory, the hash keys are 64 to 128 bits long and the hash values are on the order of 20 bytes big.

Evidently, if the signature does not include $time()$ or $seq()$ we lose the temporal dimension of the performance data, and with it the ability to understand which events happened before or after each other from the measured data. In this paper we show that some important temporal information can be recovered by keeping track of the sequence of event signatures. We call the resulting graphs which are akin to control flow graphs event signature flow graphs or simply event flow graphs.

To construct a flow graph consider an application executing with n processes and let $E_i = \{e_0, e_1, \ldots\}$ be the sequence of events at rank i, $\sigma : E_i \mapsto S_i$ be the signature function at rank i, and $s_i^0 \in S_i$ some initial signature value. Then σ' with

$$\sigma'(e_0) = (s_i^0, \sigma(e_0))$$
$$\sigma'(e_i) = (\sigma(e_{i-1}), \sigma(e_i)) \qquad (i > 0)$$

is the history signature for σ. The directed weighted graph $G = (N_i, L_i, w_i, s_i^0)$ with

$$N_i = \{\sigma(e_i)\} \qquad e_i \in E_i$$
$$L_i = \{\sigma'(e_i)\} \qquad e_i \in E_i$$
$$w_i : L_i \mapsto \mathbb{N} \qquad w_i(l) = |\{e_i : \sigma'(e_i) = l\}| \qquad l \in L_i$$

is the event signature flow graph for rank i and s_i^0 is the start node of the graph. An example flow graph is shown in Fig. 3. Nodes correspond to MPI calls, edges between the nodes correspond to transitions between them.

3.1 Merging Graphs from Multiple Processes

Event flow graphs form different processes can be merged in a straightforward way to form a multigraph (a graph with multiple edges between a pair of nodes). We build the merged graph by identifying similar nodes among the graphs (i.e., having identical $\sigma(e_i)$ with respect to some equality criterion) and inserting the edges and weights from each depending on the signature component we are concerned with.

The best way for identifying two event signatures $\sigma(e_i)$ as being identical depends on the signature components:

Signature component	**Equality test**
Type of MPI call	Exact equality
Data size	Exact equality or approximate (same magnitude) equality
Data address	Exact equality
Own rank	Discarded, since we merge across ranks
Partner rank	Equality of relative ranks
Callsite ID	Equality in unified calltrees
Program region	Exact equality

Fig. 2. Unification of signature components across MPI processes.

Identifying identical callsite IDs requires us to unify the calltree of each rank. During the execution a calltree is recorded (nodes are the callsites of MPI calls) and numerical IDs are assigned consecutively. Depending on the sequence in which functions are executed, the same callsites can be assigned different IDs on different processes. A unification step which IPM performs after the program terminates guarantees a consistent assignment of callsite IDs.

For comparing ranks we use differences (relative ranks) since in many applications parallelism is exploited in the form 2-D or 3-D domain decomposition and the MPI communication pattern is often based on the topological position of a processor (i.e, nearest neighbor communication in a grid). For this reason it is most often convenient to convert the absolute partner rank of a communication event into a relative rank (e.g., a `MPI_Send` to processor with relative rank -4).

One further simplification step can be performed on the edges between nodes. To simplify presentation and understanding of the graphs we cluster edges with the same multiplicity or weight together.

An example for a resulting application level event flow graph for a very simple application is shown in Fig. 3. The program is executed with four MPI processes, where ranks $0, 2$ perform a receive operation and ranks $1, 3$ perform a send.

```
void main(int argc, char* argv[]) {
  MPI_Init(...);
  MPI_Comm_size(...);
  MPI_Comm_rank(..., &myrank);

  for(i=0; i<10; i++) {
    if(myrank is odd)
      MPI_Send(10 doubles to rank -1);
    else
      MPI_Recv(10 doubles from rank +1);
  }
  MPI_Finalize();
}
```

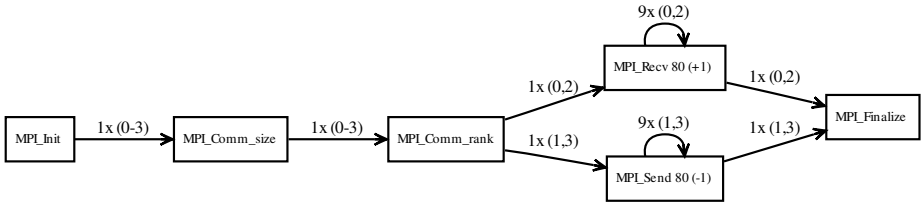

Fig. 3. A simple MPI program, executed with four MPI processes and its accompanying merged event flow graph using relative rank addressing.

3.2 Implementation in IPM

We have implemented the event flow recording scheme as described in Sect. 3 in our profiling tool IPM. IPM keeps event statistics (number of occurrences, total duration, and so on) in a hash table and the hash key derived from the MPI communication events correspond to the event signatures. To record the event flow information, the hash key is extended to contain both the current signature $\sigma(e_i)$ as well as the signature of the previous event $\sigma(e_{i-1})$. The previous event's signature is recorded in a variable and updated on each insert into the hash table.

Using this scheme event statistics are now correlated with pairs of event signatures that form the edges of the event flow graph. Upon program termination, the hash table is inspected and the flow graph is reconstructed from the hash table by looking for matching pairs of event flow edges.

4 Visualizing and Exploring Event Flow Graphs

The flow-graphs are recorded by IPM on a per-rank basis and written to a file. The merging and unification step is performed by a perl script in a post-processing step which generates a number of event flow graph files suitable for input into Graph::Easy [1] and further layout by dot [3].

Consider the table in Fig. 4. It shows the number of events in a full trace of several applications of the NAS parallel benchmark suite as well as the number of nodes in the event flow graph using the signature components indicated in the first column. These application contain no developer-provided phase markers and the MPI call type can be derived from the callsite ID, so *size*(), *partner*(), *csite*() provide the largest signature space and a subset of these signature functions will generally lead to fewer nodes in the flow graphs.

Evidently, the callsite ID is essential to achieve a large signature space. In fact, adding *partner*() and *size*() components does not add more nodes to the flow graphs for all but one application (MG). For MG both the communication partner and the transmit data size need to be added to differentiate between all events.

Method	**BT**	**CG**	**EP**	**FT**	**IS**	**LU**	**MG**	**SP**
Full Trace	29856	20184	36	85	165	255213	8988	49828
Event Flow Graph:								
size(), *partner*(), *csite*()	404	240	36	45	57	277	2796	352
size(), *partner*()	184	72	28	36	45	93	236	124
size(), *csite*()	404	240	36	45	57	277	2644	352
partner(), *csite*()	404	240	36	45	57	277	1140	352
size()	76	60	28	36	45	75	220	76
partner()	76	48	24	32	41	56	60	60
csite()	404	240	36	45	57	277	852	352

Fig. 4. Number of events in the full traces and number of nodes in the event flow graphs for the NAS parallel benchmark suite (size A, 4 processors).

Fig. 5 shows the flow graph of the IS application. For this small application the entire flow graph is easily visualized. For larger applications the direct approach becomes infeasible. Considering the results from Fig. 4, we decided to focus the on methods to interactively explore the event flow graphs along the callsite dimension by developing a combined calltree-eventgraph display.

The user is presented with a calltree display alongside with a *portion* of the flowgraph which depends on the node selected in the calltree. An example for

Fig. 5. Full event flow graph of the IS application from the NAS parallel benchmark suite.

this is shown in Fig. 6. The leafes of the calltree on the left correspond to the MPI events that comprise the flowgraph on the right.

Assume a user selects an internal node `foo()` of the calltree. Then there is effectively a partitioning of the flowgraph nodes into three sets: (1) nodes that are immediate children of the selected node, (2) those that are children but not immediate children, and (3) all other nodes.

Set (1) corresponds to MPI functions called directly from `foo()` (i.e, leaves one level below `foo()`; these nodes and their transitions are shown directly in the flowgraph display to the right. Nodes in set (2) correspond to functions called from functions called from `foo()` (leaves two or more levels below `foo()`). Those nodes are replaced by a representative, which is the child function called from `foo()` that leads to their execution. Nodes in set (3) are not displayed at all unless there is a transition to a visible node (from sets (1) or (2)). In this case the node is displayed with a dotted border and a dotted line, indicating a control flow coming from the "outside".

An example of this display technique is shown in Fig. 6. This method is very effective at narrowing down the set of nodes in the flow graph to a manageable set for interactive exploration and understanding of application code.

5 Related Work

Control flow graphs are an important topic in the area of code analysis, generation, and optimization. In that context, CFGs are usually constructed based on a compiler's intermediate representation (IR) and are defined as directed multi-graphs with nodes being basic blocks (single entry, single exit) and nodes representing branches that a program execution *may* take. The difference to the CFGs in our work is primarily twofold. First, the nodes in our graphs are not basic blocks but communication events. Second, the edges in our graphs record transitions that have actually happened during the execution and also contain a count that shows how often the transition occurred.

Dragon [2] is a performance tool from the OpenUH compiler suite. It can display static as well as dynamic performance data such as the calltree and control flow graph. The static information is collected from OpenUH's analysis of the source code, while the dynamic information is based on the feedback guided optimization phase of the compiler. In contrast to our approach, the displays are based on the compiler's intermediate representation of source code. The elements of our visualization are the constructs of the user's model of execution

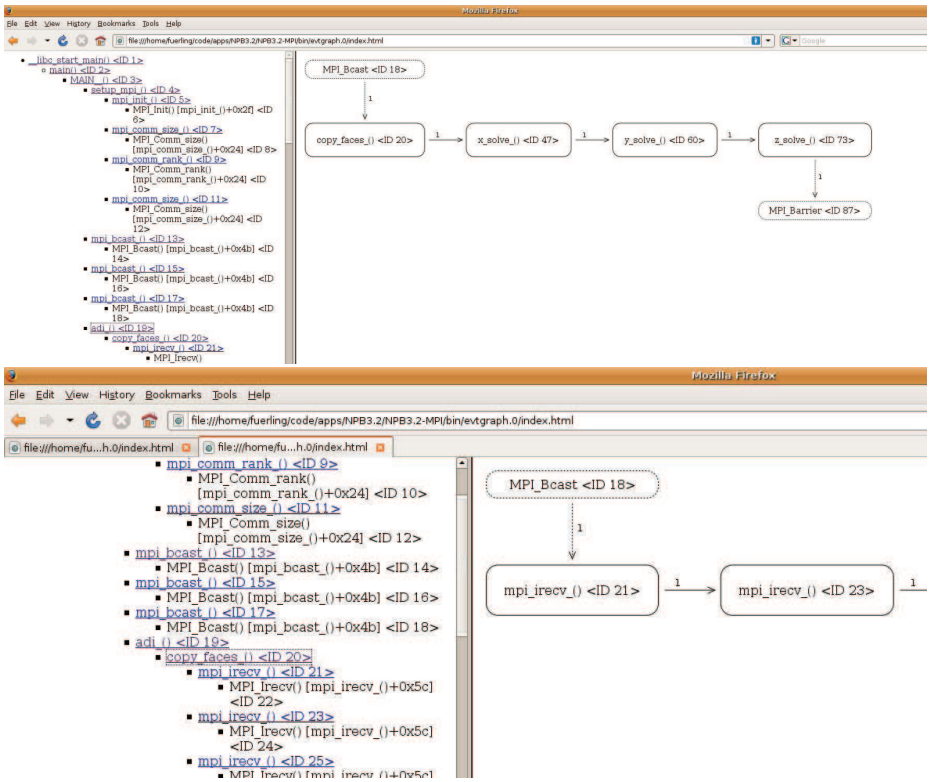

Fig. 6. Combined calltree-controlflow visualization. The user selects a node on the calltree to the left and depending on the selection only a subset of the event flow graph is presented on the right.

to contribute to a high-level understanding of the program execution characteristics.

The work of Preissl et al. [5] tries to detect recurring patterns of communication events for optimization purposes. Events are recorded as an array of 32-bit integer values (i.e., a trace) and repeating sequences of events are searched for by either a convolution or suffix-tree based method. The identified and matched repeating sequences, together with source code analysis using Rose [6], are the basis for source code transformations such as replacing a series of point to point operations with the corresponding collective. Compared to their method, our technique avoids the overhead of generating, storing, and analyzing traces. Instead our technique of recording the execution control flow directly exposes repetitive structures as loops in the flow graphs.

Finally, the work of Noeth [4] shares some similarities with our approach. In this trace compression scheme, region descriptors are applied to perform both intra-node and inter-node compression. Although this approach is able to reduce traces from applications employing regular communication patterns to near constant size independent of the number of nodes, runtime overhead is incurred for establishing and maintaining the region descriptors. In contrast, our approach has negligible cost at runtime, while we don't guarantee that the trace can be recovered completely. In fact, the potential to recover the original trace employing node ordering heuristics is part of our ongoing work.

6 Conclusion

We have discussed a technique to efficiently gather an event flow graph from MPI applications. Nodes in the graph are representations of MPI communication events and edges represent the number of transitions between them. Event flow graphs try to strike a balance between the abundance of data contained in full traces and the concise information profiling tools can deliver with low overheads. The graphs are conceptually similar to flow charts used in algorithm and application development. We presented ideas to reduce the complexity of the graphs by collapsing nodes in a step-by-step fashion and presented techniques to explore flow graphs interactively.

Future work is planned with respect to several directions. First, while timing statistics are already recorded for each edge of the flow graph, they are currently not used in the visual display. It should be straighforward to develop a coloring scheme to color nodes according to MPI communication time and the data volume sent or received. This would draw the user's attention to the most interesting parts of the graph for optimization purposes.

As a bigger step we plan to explore the usability of the flow graphs to perform MPI process clustering at petascale. With very large numbers of MPI processes used at that scale, performance data visualization that involves the rank ID as a dimension becomes impractical or even impossible. An automated clustering of ranks into a small number of processes that qualitatively exhibit the same behavior would be a solution to this problem. Another area for future exploration

is the application of techniques from graph theory to our flow graphs. Examples include cycle detection and extraction to automatically delineate computational and iterative phases.

References

1. The graph::easy web page: `http://search.cpan.org/~tels/Graph-Easy/`.
2. Oscar Hernandez, Chunhua Liao, and Barbara Chapman. Dragon: A static and dynamic tool for OpenMP. In *Proceedings of the Workshop on OpenMP Applications and Tools (WOMPAT 2004)*, pages 53–66, 2004.
3. Eleftherios Koutsofios and Stephen C. North. *Drawing graphs with dot.* Murray Hill, NJ, October 1993.
4. Michael Noeth, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. Scalable compression and replay of communication traces in massively parallel e nvironments. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS '07)*, pages 1–11. IEEE, 2007.
5. Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. Supinski, and Daniel J. Quinlan. Using MPI communication patterns to guide source code transformations. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part III*, pages 253–260, Berlin, Heidelberg, 2008. Springer-Verlag.
6. Daniel J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, September 2000.