New Algorithms for Performance Trace Analysis Based on Compressed Complete Call Graphs

Andreas Knüpfer and Wolfgang E. Nagel

Center for High Performance Computing,
Dresden University of Technology, Germany
{knuepfer, nagel}@zhr.tu-dresden.de

Abstract This paper addresses performance and scalability issues of state-of-the-art trace analysis. The Complete Call Graph (CCG) data structure is proposed as an alternative to the common linear storage schemes. By transparent in-memory compression CCGs are capable of exploiting redundancy as frequently found in traces and thus reduce the memory requirements notably. Evaluation algorithms can be designed to take advantage of CCGs, too, such that the computational effort is reduced in the same order of magnitude as the memory requirements.

1 Introduction

Todays High Performance Computing (HPC) is widely dominated by massive parallel computation, using very fast processors [1]. HPC performance analysis and particularly tracing approaches are affected by that trend. The evolution of computing performance combined with more advanced monitoring and tracing techniques lead to very huge amounts of trace data. This is becoming a major challenge for trace analysis - for interactive investigation as well as for automatic analysis. With interactive work flows the requirement for fast response times is most important for analysis tools. For automatic or semi-automatic tools that use more or less computational expensive algorithms and heuristics this is a severe problem, too. Both have in common that the effort depends on the amount of trace data at least in a linear way.

The state of the art in-memory data structures for trace data suggest linear storage only, i.e. arrays or linear linked lists [2, 11, 4, 14]. Even though they are fast and effective for raw access, they lack opportunities for improvement.

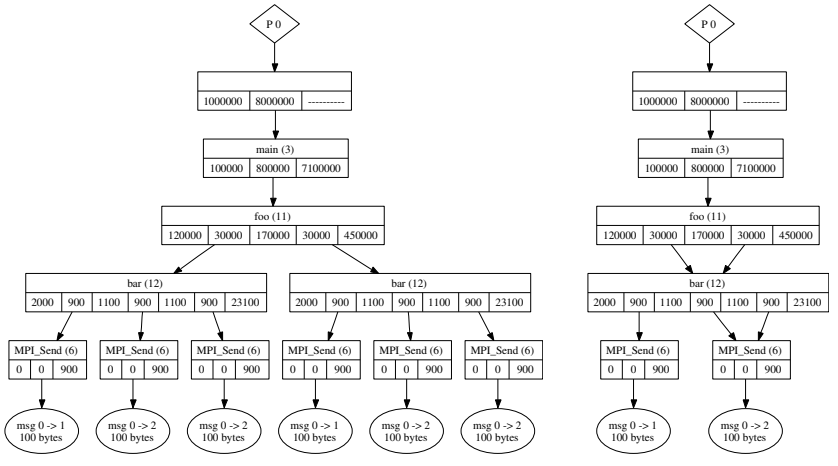
The Compressed Complete Call Graph (cCCG) is a promising alternative approach for storing trace data. Contrary to linear data structures it offers fast hierarchical access. Furthermore, it supports transparent compression of trace data which saves memory consumption as well as computational effort for some kinds of queries.

The following Section 2 gives a concise overview and definition of cCCGs as well as its properties. Based on that, the concept of Cached Recursive Queries onto cCCGs is developed in Section 3. Also, it provides some typical examples of use for this algorithm with performance results. Finally, Section 4 concludes the paper and gives an outlook on future work.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 116–123, 2005.

2 Compressed Complete Call Graphs (cCCGs)

An alternative way to the traditional linear scheme is to re-create the complete function call tree. This preserves the temporal order as well as the call hierarchy. A Complete Call Tree (CCT) contains the comprehensive function call history of a process. This makes it different from ordinary Call Trees which store a summarized caller to callee relation only [6]. Furthermore, not only function call events but also all other kinds of events can be contained such as message send/receive events, I/O events or hardware performance counter samples. However, the function call hierarchy determines the structure of the CCT. Figure 1(a) shows an example of a Complete Call Tree including some `MPI_Send()` calls.



(a) uncompressed (b) compressed

Fig. 1. An example Complete Call Graph (a) and its compressed counterpart (b)

This figure shows also another difference to the traditional way of storing trace data. Instead of time stamps the CCT holds time durations which is most important for the compression part. Of course, the back and forth transformation between time stamps and time durations is simple. In terms of expressiveness a CCT is equivalent to a traditional linear data structure with time stamps.

2.1 Compression

The more structured CCT representation makes it easy to detect identical nodes and sub-trees of pairwise identical nodes. All groups identical sub-trees are then replaced by references to a single representative item. All other instances can be eliminated completely. This removal of redundancy is a typical strategy for data compression and transforms CCTs to Compressed Complete Call Graphs (cCCGs). Figure 1(b) shows the compressed counterpart of the previously mentioned CCT in Figure 1(a). Of course, this transformation destroys an essential

property of tree graphs, namely the single parent node property. However, all construction and query algorithms can be implemented in a way not to rely on that. Thus, this kind of compression can be said to be completely transparent with respect to read access.

So far, cCCGs offer a completely lossless compression scheme for trace data. HPC programs show a high degree of repetition and regularity. This is reflected in traces as well. Therefore, this compression strategy works reasonably well.

At this point, it is possible to allow not only equal but also *similar* subtrees to be mapped onto one another. This takes this approach another step further introducing lossy compression. However, this is applicable for selected members of the node data structure only. For example, identifiers for functions or processes must not be subject to lossy compression because this would render the information invalid. Other data members as time durations, operation count, message volumes etc. are robust against small deviations in the data.

So, all those thoughts need to be considered when defining what *similar* is supposed to mean for sub-graphs. Furthermore, all deviations introduced must be limited by selectable bounds. This will induce error bounds for all results computed from data carrying deviations.

Following this construction scheme plainly there arises one major disadvantage in terms of general graph data structures. As the structure of a CCG is determined by the call hierarchy alone, the tree's branching factor is unbounded and probably very large. This causes two negative effects. First, large branching factors are most undesirable for tree traversal algorithms. Second, the compression ability is enhanced by rather small branching factors By introducing special nodes the branching factor can be bounded to an arbitrary constant ≥ 2 [7].

2.2 Compression Metrics

In order to make the compression comparable a measure for compression is needed. For cCCGs there are two metrics suitable for different purposes:

$$R_m := \frac{\text{Memory}_0}{\text{Memory}_{compressed}} \ , \ R_n := \frac{\text{Nodes}_0}{\text{Nodes}_{compressed}} = \frac{N}{n} \tag{1}$$

First, the ratio R_m of the raw memory consumption of graph nodes including references (pointers) to child nodes is suitable for estimating memory consumption. This is a the key issue as soon as data compression is concerned. Second, the node count ratio R_n is very useful when estimating the computational effort for tasks that perform constant amount of work per graph node. Since single nodes have variable memory footprints R_n is not proportional to R_m.

Practical experiments with real world traces from 20 MB up to 2 GB have shown very promising results [7]. For zero time deviation bounds R_m ranges from 2 to 8 and R_n lies in between 5 and 14. For large traces with midrange deviation bounds for time information of 1000 ticks (timer units) or 50 % the memory compression ratio R_m rises up to 44 while the node compression ratio R_n climbs up to 93. With more relaxed bounds R_m and R_n rise over 1000!

Compression ratios of $R_X < 1$ are impossible, and the memory requirements for uncompressed CCGs and traditional linear data structures are about the

same. In general, it can be stated that more relaxed error bounds lead to better compression. Furthermore, larger traces usually yield better compression than shorter ones. Moreover, the higher the final compression ratio will grow, the faster the compression itself will be. Within the CCG creation algorithm the construction and compression steps are closely integrated such that at no point the whole uncompressed graph needs to be stored. The overall complexity for cCCG construction is $O(N \cdot m)$ with the node count in the uncompressed CCG N and a rather small factor m. For construction, split and compression algorithms, complexity analysis and experimental performance results see [10].

3 Cached Recursive Queries

After creation from trace or re-creation from a previously saved version the Compressed Complete Call Graph is available in main memory for querying. This might involve automatic evaluation procedures or interactive user queries. This article focuses on interactive queries particularly with regard to visualization tasks while, of course, the cCCG data structure is suitable for performing automatic analysis, too.

One of the two most important kinds of queries is the so called *Summary Query*. It computes a certain property for a given time interval and a given set of processes. Typical examples for summary queries are exclusive or inclusive run time per function, message volumes per pairs of processes, average hardware performance counter values and many more.

The traditional algorithm to derive summary statistics performs a single linear read-through of all process traces and uses some additional temporary memory to re-play the call stack. While this read-through in temporal order can be emulated by cCCGs another algorithm is proposed here. Following the tree-like graph structure a recursive traversal of the graph seems most obvious. This is well suited to calculate the query's result in a divide and conquer style. Considering an uncompressed CCG the graph itself is its own evaluation graph as shown in Figure 2 for a single process. From the computational complexity point of view this algorithm is equal to the traditional way with $O(N)$ effort.

For successive queries with overlapping time intervals this evaluation scheme can be improved by caching of intermediate results at graph nodes. Especially for interactive navigation within a trace such sequences of successive queries with non-disjoint time intervals are very common. Most of the time, it involves an initial global query followed by multistage zooming into some interesting regions for closer examination.

Caching avoids re-computation of intermediate results that appear multiple times. That means, whenever the evaluation encounters existing cached results the computation graph is pruned, i.e. the underlying sub-tree must not be traversed. See Figure 2 for an illustrated example.

Typical cache strategies like Most Frequently Used (MFU) or Most Recently Used (MRU) are not feasible here, assuming that the cache is small in comparison to node count. When inserting newly computed results this would lead to

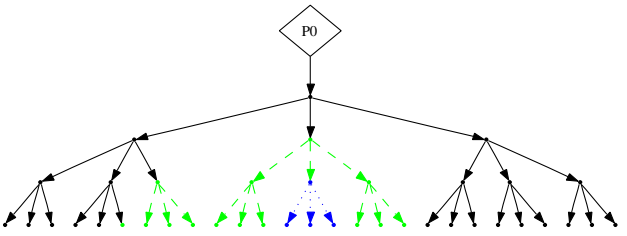


Fig. 2. Evaluation graph for querying a uncompressed CCG. This is identical to the CCGs own structure. Marked green (dashed) and blue (dotted) there are two successive nested queries to the initial global query. For both the recursion can be pruned as soon as a cached intermediate result is found for one of the graph nodes

continuous cache thrashing. Instead, heuristics are utilized to limit the number of entries. For example, one could insert every n'th result into the cache. Another convenient strategy is to insert only nodes of certain depth levels d_{node} with

$$d_{node} \text{ modulo } x = y, \;\; y < x. \tag{2}$$

The latter strategy allows to estimate the computational effort, because there are at maximum x depth levels to traverse before there are cached results for all nodes. With maximum branching factor b this yields worst case effort of $O(b^x)$.

In addition to this effort, there is a preceding effort of finding the smallest graph node containing the given time interval. This involves following a single path from the root node downwards if maximum length d which is the maximum tree depth. Furthermore, any result for nodes intersecting the bounds of the current time interval cannot be cached but must be computed in a special way which is aware of the interval bound. Therefore, all nodes intersecting the time interval bounds must be handled separately. For both interval bounds this involves at maximum $2 \cdot d$ nodes. Thus, the overall complexity is $O(d + b^x) \leq O(N)$.

3.1 Application to Compressed CCGs

This algorithm can be applied to compressed CCGs as well. Actually, compression contributes to a tremendous improvement of this evaluation scheme. Since graph nodes are referenced repeatedly, caching pays off even for initial queries. In addition, the compressed node count $n \ll N$ causes a reduced cache size. This saving is proportional to the node compression ratio $R_n = \frac{N}{n}$. At the same time, the cache re-use frequency is increased by that factor. Alternatively, the parameters for the caching heuristics can be relaxed - compare Equation (2). Figure 3 gives an impression how the compressed counterpart to the example in Figure 2 might look like and how initial and successice queries are performed.

The second of the two most important kinds of interactive queries is the Timeline Visualization Query. It shows the behavior of some properties over the course of time representing values with a color scale. Subject of timeline displays might again be statistical values like run time share, communication volumes or

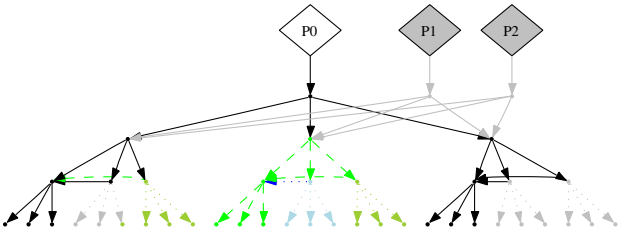


Fig. 3. Evaluation graph of successive queries on a compressed CCG. Just like within the CCG itself some sub-trees are replaced by references to other instances. Thus, some sub-trees are referenced more than once. Intermediate results for such sub-trees can be re-used within the initial query and in all successice queries. Sub-trees might even be shared by multiple processes, e.g. processes P1 and P2 might references sub-trees originally from P0

hardware performance counter values. It might also be the characterization of the function call behavior, maybe even including the call hierarchy. This is in fact the most commonly used variety of timeline displays [3, 2].

Always, timeline visualizations are rendered for a given horizontal display resolution of $w \ll N$ pixels. With traditional data structures this requires linear effort $O(N)$ for a single read-through at least. Based on cCCGs, this quite different task can be converted to a series of Cached Recursive Queries, each one restricted to the time interval associated with a single pixel. This allows to transfer the reduced effort algorithm (with $O(d + b^x)$) to this problem, too.

3.2 Performance Results

After the theoretical considerations some performance results shall emphasize the practical advantages of the cCCG based Cached Recursive Queries. As test candidate a trace file from the ASCI benchmark application IRS [13] was used which is 4 GB in size (in VTF3 format [12]) and contains 177 million events. This measurements were performed on an AMD Athlon 64 workstation with 2200 MHz speed and 2 GB of main memory.

Figure 4 shows the evaluation time for a Cached Recursive Query computing exclusive and inclusive time as well as occurrences count per function all at once. It is presented depending on the node compression ratio R_n of the cCCG and the cache strategy parameter x as in Equation (2). The left hand side shows initial queries which take $1s$ to $23s$ depending on compression rate. There is only a minor influence of the caching parameter. On the right hand side, run times for successive queries are shown, again with global scope. Here, the run time ranges from $50ms$ to $400ms$ which is without doubt suitable for truly interactive responses. For non-global queries restricted to a smaller time interval both, initial and successive queries will be even faster.

In comparison to traditional evaluation on linear data structures this is an important improvement. For example, the classical and well known trace analysis tool Vampir [2] is incapable of providing any information about the example

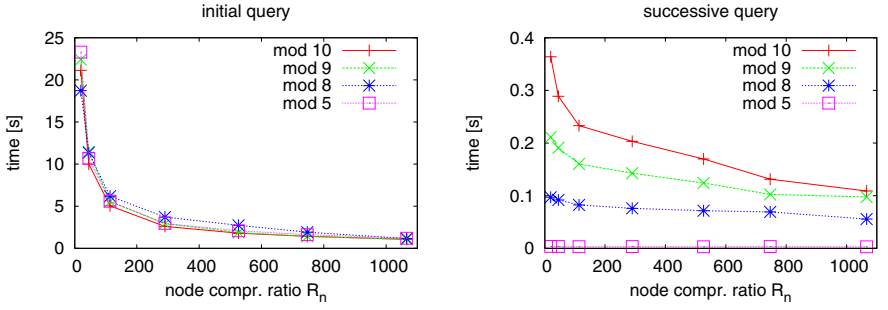


Fig. 4. Run time for initial (left) and successive (right) global Cached Recursive Queries on cCCGs with different compression levels and caching parameters

trace just because of its size on the given workstation as 2 GB main memory are insufficient to contain the trace. Furthermore, on any 32 bit platform the address range is not large enough. Thus, the new approach is not only a an improvement in speed but also in receiving any valid information or not.

4 Conclusion and Outlook

The paper presented a novel evaluation algorithm for Compressed Complete Call Graphs. This Cached Recursive Query is capable of delivering results in a truly interactive fashion even for larger traces. This is especially necessary for convenient manual analysis and navigation in traces. For large traces, this is superior to the traditional scheme. Furthermore, this algorithm unites the tasks of computing statistical summary queries and of generating timeline diagrams.

Parallelizing and distributing the Compressed Complete Call Graph approach and the Cached Recursive Query algorithm is another option to extend the range of manageable trace file sizes. This has already been implemented in a successful experiment [8] introducing the cCCG data structure into Vampir NG [5].

Future research will focus on automatic and semi-automatic performance analysis techniques based on cCCG data structures. First, this aims at applying known procedures to cCCGs taking advantage of compression and reduced memory footprint. Second, this extends to developing new methods. Some result have already been published in [9].

References

1. George Almasi, Charles Archer, John Gunnels, Phillip Heidelberger, Xavier Martorell, and Jose E. Moreira. Architecture and Performance of the BlueGene/L Message Layer. In *Recent Advances in PVM and MPI. Proceedings of 11th European PVM/MPI Users Group Meeting*, volume 3241 of *Springer LNCS*, pages 259–267, Budapest, Hungary, September 2004.

2. H. Brunst, H.-Ch. Hoppe, W.E. Nagel, and M. Winkler. Performance Otimization for Large Scale Computing: The Scalable VAMPIR Approach. In *Proceedings of ICCS2001, San Francisco, USA*, volume 2074 of *Springer LNCS*, page 751. Springer-Verlag Berlin Heidelberg New York, May 2001.
3. H. Brunst, W. E. Nagel, and S. Seidl. Performance Tuning on Parallel Systems: All Problems Solved? In *Proceedings of PARA2000 - Workshop on Applied Parallel Computing*, volume 1947 of *LNCS*, pages 279–287. Springer-Verlag Berlin Heidelberg New York, June 2000.
4. Holger Brunst, Allen D. Malony, Sameer S. Shende, and Robert Bell. Online Remote Trace Analysis of Parallel Applications on High-Performance Clusters. In *Proceedings of ISHPC'03 Conference*, volume 2858 of *Springer LNCS*, pages 440–449, 2003.
5. Holger Brunst, Wolfgang E. Nagel, and Allen D. Malony. A Distributed Performance Analysis Architecture for Clusters. In *IEEE International Conference on Cluster Computing, Cluster 2003*, pages 73–81, Hong Kong, China, December 2003. IEEE Computer Society.
6. David Grove and Craig Chambers. An assessment of call graph construction algorithms. `http://citeseer.nj.nec.com/grove00assessment.html`, 2000.
7. Andreas Knüpfer. A New Data Compression Technique for Event Based Program Traces. In *Proccedings of ICCS 2003 in Melbourne/Australia, Springer LNCS 2659*, pages 956 – 965. Springer, Heidelberg, June 2003.
8. Andreas Knüpfer, Holger Brunst, and Wolfgang E. Nagel. High Performance Event Trace Visualization. In *13th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Lugano, Switzerland, Feb 2005.
9. Andreas Knüpfer, Dieter Kranzlmüller, and Wolfgang E. Nagel. Detection of Collective MPI Operation Patterns. In *Recent Advances in PVM and MPI. Proceedings of 11th European PVM/MPI Users Group Meeting*, volume LNCS 3241, pages 259–267, Budapest, Hungary, September 2004. Springer.
10. Andreas Knüpfer and Wolfgang E. Nagel. Compressible Memory Data Structures for Event Based Trace Analysis. *Future Generation Computer Systems by Elsevier*, 2004. [accepted for publication].
11. Dieter Kranzlmüller, Michael Scarpa, and Jens Volkert. DeWiz - A Modular Tool Architecture for Parallel Program Analysis. In *Euro-Par 2003 Parallel Processing*, volume 2790 of *Springer LNCS*, pages 74–80, Klagenfurt, Austria, August 2003.
12. S. Seidl. VTF3 - A Fast Vampir Trace File Low-Level Library. personal communications, May 2002.
13. The ASCI Project. The IRS Benchmark Code: Implicit Radiation Solver. `http://www.llnl.gov/asci/purple/benchmarks/limited/irs/`, 2003.
14. F. Wolf and B. Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. Technical report, Research Center Jülich, April 1998. FZJ-ZAM-IB-9803.