PARLOT: Low-Overhead Tracing of Large-Scale Parallel Programs

Sindhu Devale
Department of Computer Science
Texas State University
San Marcos, Texas, U.S.A.
Email: sindhu.devale@gmail.com

Saeed Taheri
School of Computing
University of Utah
Salt Lake City, Utah, U.S.A.
Email: staheri@cs.utah.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, Texas, U.S.A.
Email: burtscher@cs.txstate.edu

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, Utah, U.S.A.
Email: ganesh@cs.utah.edu

***Abstract*— added commit**

Effective HPC debugging of many classes of bugs on heterogeneous platforms requires efficient tracing methods that can track control flow in terms of function calls and returns, and also optionally captures a record of executions through user and system codes. Binary instrumentation is the only approach that can accomplish this goal; unfortunately, available tools either do not produce the kinds of traces that can help with debugging or incur huge overheads when the traces are brought out of the cores. In this paper, we present PARLOT, a tool that extends Intel's PIN tool to not only perform binary-level tracing, but adds several key features: (1) it deploys a set of highly efficient and programmable trace compression methods that reduces the trace volume exported from a chip dramatically, thus making tracing affordable. (2) it extends PIN's tracing methods to not only record function calls and returns, but using heuristic approaches corrects the stack pointer to accommodate non-standard returns. (3) it produces a wealth of statistics including caller/callee relations, call frequencies, and a linear trace of entire executions at the granularity the user opts for. This paper establishes that comparable capabilities are unavailable by evaluating the best alternative tracing options on runs up to 1,024 cores on the NAS parallel benchmarks on two HPC platforms. Our experiments show that parlot can produce full function call trace with only 0.85 kB required bandwidth while slowing down applications only by as low as 38%(fig ??).

***Index Terms*—tracing, HPC, compression;**

I. INTRODUCTION

The act of rendering any software design into code invariably introduces bugs. The code is tested till it stops readily revealing bugs, and then it is put into production. The remaining lurking bugs often remain dormant, but rear their head when the inputs are changed or the code is ported to new platforms. While traditional software engineering often achieves high quality control, these methods are largely inapplicable to HPC where concurrency combined with problem-scale and the presence of very sophisticated domain-specific math makes programming really hard. Good quality HPC software almost always requires extremely skilled developers. Unfortunately, good developers are rare, and the need to deploy HPC software only keeps growing, thus putting a lot of burden on the shoulders of those who still are maturing. The only solution is that debugging tools must begin doing far more heavy lifting than traditionally expected.

This work was supported in part by NSF grant 1438963.

In HPC, bugs are a function of both flawed program logic as well as unspecified and illegal interactions between various concurrency models (*e.g.*, PThreads, MPI, OpenMP, etc.) that coexist in any large application. The best hope for debugging in many cases lies in being able to efficiently capture detailed execution traces and compare them against traces emerging from previous (and more stable) software versions. This paper offers ParLot, an efficient binary-level tracing tool that (1) traces compiled applications with very little overhead, (2) employs advanced data compression methods to bring out traces across the memory hierarchy with dramatically reduced bandwidth needs, and (3) provides debuggers (humans and debugging tools) with traces that capture many valuable pieces of information such as function call/return chains, function call frequencies, and the total number of various instruction types executed. The shocking thing is that despite being at the threshold of exascale computing, the community does not have a tracing tool that has anywhere near the level of efficacy that ParLot has.

This paper itself does not propose debugging methods. Instead, the purpose of this paper is to really take stock of what else is available for tracing and debugging large-scale applications, and perform a fairly thorough evaluation of their capabilities vis-a-vis ParLot. Thus, we aim to scientifically establish that we indeed have solved the tracing problem better than anyone else has. We then appeal to past work (some of it being our own past successes) showing that we now are in a position to breathe new life into such tracing-based debugging tools, now that ParLot will be able to efficiently provide trace information to them.

Here are the main contribution of this paper:

- Tracing
- Compression
- Stack correction

The remainder of this paper is organized as follows. In section II, we introduce the basic ideas and infrastructures behind ParLot. Section **??** describes existing tracing tools and techniques and their relevance to our approach. We present the implementation of Parlot in Section IV. Section **??** and **??** shows our evaluation of different aspects of Parlot and its comparsion with *Callgrind*.

II. Background

A. PIN

Recording a log of events during the execution of an application is essential for better understanding of the behavior of the program and, in case of a failure, to locate the problem. Recording this type of information requires instrumentation of the code either at the source-code or binary-code level. Instrumenting the source code is easier for developers but not for users since developers can add specific pieces of code to certain points of the source code to collect the needed information. But it requires modification of the source code and recompilation, which make it more difficult and less straight-forward for users. In addition, binary instrumentation makes the process of tracing language independent and portable. It also provides machine-level insight of the behavior of the application. Binary codes can be instrumented *statically*, where the additional code is inserted into the binary before execution, which results in a persistent modified executable, or *dynamically*, where the modification of the executable is not permanent. In dynamic binary instrumentation, code can be discovered at runtime, making it possible, for example, to handle dynamically-generated and self-modifying code. Furthermore, it may be possible to attach the instrumentation to running processes, which is particularly useful for long-running applications.

We designed ParLOT on top of PIN [??], a dynamic binary instrumentation framework for the IA-32, x86-64, and MIC instruction-set architectures that enables the creation of dynamic program analysis tools. During execution, ParLOT tracks the function call stack and captures every entry (call) and exit (return) of every function. ParLOT not only captures the functions of the main image but also in library code.

B. Compression

When dealing with large-scale parallel programs, any attempt to generate traces will likely result in a huge amount of data. Moreover, such tracing will also incur significant overhead due to the need to transfer and store the vast amount of data. For example, collecting just one byte of information per executed instruction generates on the order of a gigabyte of data per second on a single high-end core. Storing the resulting multi-gigabyte traces from many cores can be a challenge, even on today's large hard disks.

Hence, we need a way to decrease the space and runtime overhead. A compression mechanism to encode the generated data into a smaller number of bits makes transmitting and storing more efficient. Although every encoded data needs to be decoded for analysis, compressing and encoding the trace data while it is being collected enable us to gather much more information.

In ParLOT, the traced information is incrementally compressed while the application is running, typically resulting in just a few kilobytes of data needing to be written per thread and per second. The traces are decompressed later (at no additional cost to the execution of the application). From the decompressed full function-call trace, the complete call graph, the function call frequency, and the caller-callee relations can be extracted. This can be done at the granularity of a thread, group of threads, or the whole application.

III. Related Tools

Instrumenting, profiling and tracing large-scale applications have become more popular to researchers and companies [ddt] due to high demand of HPC users. Dyninst [1] is a dynamic instrumentation API which gives developers the ability to measure the performance [2] [3] and develop correctness debuggers [4]. It instruments the binary without any need of recompilation and also gives the developer the ability to attach instrumentation to a running process. VampirTrace [5] also uses Dyninst API to provide a library for collecting logs from program execution.

The idea of analyzing execution traces for debugging purposes have been used in STAT [4] where it groups the processes with similar function-call stack and trying to find abnormal behavior like divergence in the function call-graph by delta debugging. The idea of tracing for debugging purposes have been used in other tools but first of all the overhead they add to the target application is high and are not straight-forward for HPC users who might not be a developer. They either need static instrumentation by inserting code-snippets and macros, or/and recompilation of the source code. Valgrind [6] is shadow value DBI framework that maps and records every register and memory value. It gives developers the capability of instrumenting system calls and instructions. Many error detectors such as Memcheck [**?**] have beeen built on top of Valgrind. Callgrind [7] is a profiling tool on Valgrind platform that records the call history among functions in a program's run as a call-graph by measuring the number of instructions executed and their relationship to source lines. Our experiments (section V) shows that Callgrind adds up to ??? overhead to the target application to collect just the call-graph while generating huge trace files without any compression applied.

and [8] [3] [**?**] [**?**]

The idea of compressing large-scale traces have been used in [9] for compressing performance traces and in ScalaTrace [10] where it uses reptetive nature of timestep simulation in parallel scientific applications to compress traces[[11]]. Only small fraction of compression is happening on-fly and the focus is on reducing inter-node communication.

IV. Implementation

Fig **??** shows the general overview of ParLOT's workflow. (a flowchart to be added) and more explanations can be added here.

A. Tracing Operation

ParLOT is built on top of PIN. In particular, it instructs PIN to instrument every thread launch and termination in the application as well as every function entry and exit. The thread-launch instrumentation code initializes the per-thread

tracing variables and opens a file into which the trace data from that thread will be written. The thread-termination code finalizes any ongoing compression, flushes out the remaining buffer entries, and closes the trace file. PARLOT assigns every static function in each image (main program and all libraries) a unique unsigned 16-bit ID, which it records in a separate file together with the image and function name. This file later serves to map IDs back to image/function-name pairs.

For every function *entry*, PARLOT executes extra code that has access to the thread ID, function ID, and current stack-pointer (SP) value. Based on the SP value, it performs call-stack correction if necessary (see below), adds the new function to a data structure it maintains that holds the call stack (which is different from the applications runtime stack), and emits the function ID into the trace file via an incremental compression algorithm (see below). All of this is done independently for each thread. Similarly, for every function *exit*, PARLOT also executes extra code that has access to the thread ID, function ID, and current SP value. Based on the SP value, it performs call-stack correction if necessary, removes the function from its call-stack data structure, and emits the reserved function ID value of zero into the trace file to indicate an exit. As before, this is done via an incremental compression algorithm. We use zero for all exits rather than emitting the function ID and a bit to specify whether it is an entry or exit because using zeros results in more compressible output. After all, this way, half of the values in the trace will be zero.

B. Call-Stack Correction

To be able to decode the trace, i.e., to correctly associate each exit with the function entry it belongs to, our trace reader maintains an identical call-stack data structure. Unfortunately, and as pointed out in the PIN documentation, it is not always possible to identify all function exits. For example, in highly optimized code, a functions instructions may be inlined and interleaved with the callers instructions, making it infeasible for PIN to identify the exit. As a consequence, PARLOT has to work correctly even when PIN occasionally misses an exit. This is where the SP values come into play.

During tracing, PARLOT not only records the function IDs in its call-stack data structure but also the associated SP values. This enables it to detect missing exits and to correct the call stack accordingly. Whenever a function is entered, it checks if there is at least one entry in the call stack and, if so, whether its SP value is higher than that of the current SP. If it is lower, we must have missed at least one exit since the runtime stack grows downwards and, therefore, the SP value decreases with every function entry and increases with every exit. If a missing exit is detected in this manner, PARLOT pops the top element from its call stack and emits a zero to indicate a function exit. It repeats this procedure until the stack is empty or its top entry has a sufficiently high SP value. The same call-stack correction technique is applied for every function exit whose SP value is inconsistent. Note that the SP values are only used for this purpose and are not included in the emitted trace data.

The result is an internally consistent trace of function entry and exit events, meaning that parsing the trace will yield a correct call stack. This is essential so that the trace can be decoded correctly. Moreover, it means that the trace includes exits that truly happened in the application but that were missed by PIN. Note, however, that our call-stack correction is a best-effort approach and may, in rare cases, temporarily not reflect what the application actually did. This can happen for functions that do not create a frame on the runtime stack.

C. Incremental Compression

PARLOT immediately compresses the traced information even before it is written to memory. In other words, it compresses each function ID before the next function ID is known. The conventional approach would be to first record uncompressed function IDs in a buffer and later compress the whole buffer once it fills up. However, this makes the processing time very non-uniform. Whereas almost all function IDs can be recorded very quickly since they just have to be written to a buffer, processing a function ID that happens to fill the buffer takes a long time as it triggers the compression of the entire buffer. This results in sporadic blocking of threads during which time they make no progress towards executing the application code. Initial experiments revealed that such behavior can be detrimental when one thread is polling data from another thread that is currently blocked due to compression. For example, we observed a several order of magnitude increase in entry/exit events of an internal MPI library function when using block-based compression.

To remedy this situation, the compressor must operate incrementally, i.e., each piece of trace data must be compressed when it is generated, without buffering it first, to ensure that there is never a long-latency compression delay. Few existing compression algorithms have been implemented in such an incremental way because it is more difficult to code up and possibly a little slower. Nevertheless, we managed to do it for our algorithm (discussed next) so that each trace event can be compressed with similar latency.

D. Compression Algorithm

to be rewritten: I used a tool called CRUSHER to determine a good compression algorithm based on several uncompressed training traces I had recorded. CRUSHER reported that an LZ component followed by a ZE component would work well. Since my tool supports up to 65535 unique function IDs, the trace entries are two-byte words, which are fed into the LZ component. Its output is interpreted as a sequence of bytes, which is fed into the ZE component for further compression. The output of the ZE component is stored to disk.

The LZ component implements a variant of the LZ77 algorithm. It uses a hash table to identify the most recent prior occurrence of the current value in the trace. Then it checks whether the three values immediately before that location match the three trace entries just before the current location. If they do not, the current trace entry is emitted and the component advances to the next entry. If the three values

match, the component counts how many values following the current value match the values following that location. The length of the matching substring is emitted and the component advances by that many values.

The ZE component emits a bitmap in which each bit corresponds to one input byte. The bits indicates whether the corresponding bytes in the input are zero or not. Following each eight-bit bitmap, ZE emits the non-zero bytes.

V. Results

VI. Summary, Conclusion and Future Work

A. Summary and Conclusion

B. Future Work

VII. Appendix

Acknowledgment

...

References

[1] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *IEEE Computer*, vol. 28, no. 11, pp. 37–46, 1995. [Online]. Available: https://doi.org/10.1109/2.471178

[2] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open | speedshop: An open source infrastructure for parallel performance analysis," *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008. [Online]. Available: https://doi.org/10.3233/SPR-2008-0256

[3] S. S. Shende and A. D. Malony, "The Tau parallel performance system," *International Journal on High Performance Computer Applications*, vol. 20, pp. 287–311, May 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1125980.1125982

[4] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2007, pp. 1–10.

[5] M. Jurenz, R. Brendel, A. Kn¨upfer, M. M¨uller, and W. E. Nagel, "Memory allocation tracing with vampirtrace," in *Proceedings of the 7th International Conference on Computational Science, Part II*, ser. ICCS '07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 839–846.

[6] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 44–66, 2003. [Online]. Available: https://doi.org/10.1016/S1571-0661(04)81042-9

[7] J. Weidendorfer, "Sequential performance analysis with callgrind and kcachegrind," in *Tools for High Performance Computing - Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, 2008, pp. 93–113.

[8] K. Fürlinger, N. J. Wright, and D. Skinner, "Performance analysis and workload characterization with IPM," in *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, 2009, pp. 31–38.

[9] X. Aguilar, K. Frlinger, and E. Laure, "Online mpi trace compression using event flow graphs and wavelets," *Procedia Computer Science*, vol. 80, no. Supplement C, pp. 1497 – 1506, 2016, international Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[10] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalatrace: Scalable compression and replay of communication traces for high-performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 696 – 710, 2009, best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007).

[11] F. Freitag, J. Caubet, and J. Labarta, *On the Scalability of Tracing Mechanisms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 97–104.

TABLE I

THIS TABLE CONTAINS SLOWDOWNS OF PARLOT AND CALLGRIND (SLOWDOWNS ARE RELATIVE TO PURE RUN). THE INPUT SIZE IS **B**. NAS BENCHMARK INPUT SIZES ARE AS FOLLOWS : $size(A) < size(B) < size(C) < size(D)$. IN LATER TABLES AND CHARTS I SHOW THAT PARLOT HAS BETTER PERFORMANCE ON LARGER INPUTS (LIKE C AND D). I WAS NOT ABLE TO RUN CALLGRIND WITH INPUT SIZE OF C AND D SINCE IT WAS TIME CONSUMING, CRASHING AND WASTING SUS. ALSO I ONLY INCLUDED THE RESULTS FOR UP TO 16 NODES (256 CORES) IN THIS TABLE. ALMOST ALL OF THE EXPERIMENTS WITH CALLGRIND ON 64 NODES (1024 CORES) CRASHED [I DOCUMENTED ALL SORT OF CRASHING REASONS OF CALLGRIND ON 1024 CORES]. PARLOT RESULTS OF 64 NODES WILL APPEAR IN LATER TABLES. I GROUPED THE RESULTS OF EXPERIMENTS WITH SIMILAR INPUT SIZES AND NODES (GROUP OF 3 ROWS). EACH ROW IS IN THIS FORMAT **TOOL.INPUT.NODES**. LAST COLUMN OF THE TABLE (GM) IS GEOMEAN OF ALL VALUES IN THAT ROW. BY COMPARING THE VALUES OF GM ROW, WE CAN SEE THAT PARLOT(BOTH MAIN AND ALL) HAS BETTER PERFORMANCE COMPARING TO CALLGRIND. HOWEVER, IT SEEMS THAT CALLGRIND SCALES BETTER (MORE ABOUT THIS IN NEXT TABLE). (FIG 1)

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.B.1	1.60	1.81	3.93	1.53	3.05	1.21	3.61	1.53	2.08
pinAll.B.1	2.01	3.03	4.63	2.43	6.86	1.66	9.10	1.56	3.20
callgrind.B.1	12.47	19.56	17.07	9.67	7.82	11.32	15.66	10.50	12.47
pinMain.B.4	1.67	3.78	2.43	2.91	4.04	2.83	3.54	2.93	2.92
pinAll.B.4	3.95	8.96	3.52	7.67	10.25	5.83	6.18	3.70	5.81
callgrind.B.4	7.83	12.64	6.07	6.18	2.89	20.14	4.70	12.04	7.69
pinMain.B.16	3.68	7.80	9.90	5.45	4.41	3.81	3.51	2.41	4.65
pinAll.B.16	6.55	12.53	6.47	10.40	5.45	4.01	6.35	4.73	6.61
callgrind.B.16	10.95	28.44	9.23	7.90	4.04	8.89	8.55	6.16	9.00

TABLE II

STAT: SD TOOLS: PINMAIN , PINALL , CALLGRIND , INPUTS: B , NODES: 1 , 4 , 16 , DESC: PRIMARY

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.B.1	1.67	1.48	3.40	1.93	1.93	1.10	2.08	1.04	1.71
pinAll.B.1	1.97	2.55	4.15	2.73	3.71	1.55	3.96	1.31	2.53
callgrind.B.1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
AVG	1.21	1.34	2.52	1.55	1.88	0.88	2.01	0.78	1.41
pinMain.B.4	1.63	1.57	1.69	2.41	2.33	1.48	1.53	3.87	1.95
pinAll.B.4	2.33	2.68	2.84	4.03	4.59	2.41	2.89	3.47	3.07
callgrind.B.4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
AVG	1.32	1.42	1.51	2.15	2.31	1.30	1.47	2.45	1.67
pinMain.B.16	2.23	1.65	2.08	1.82	2.66	1.84	1.57	1.60	1.90
pinAll.B.16	4.20	2.68	4.85	2.71	8.53	3.34	2.81	2.98	3.70
callgrind.B.16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
AVG	2.14	1.44	2.31	1.51	3.73	1.73	1.46	1.53	1.87

TABLE III

THE VALUES IN THIS TABLE IS IDENTICAL TO TABLE I BUT GROUPED DIFFERENTLY TO SHOW THE SCALABILITY OF EACH TOOL. THE SLOWDOWN OF CALLGRIND DROPS DRASTICALLY WITH INCREASING CORES FROM 16 TO 64. FOR PARLOT (FOR BOTH MAIN AND ALL), SLOWDOWNS ARE HIGHER FOR LARGER NUMBER OF CORES. HOWEVER, THE AVERAGE GEOMEAN OF ALL SLOWDOWNS (NUMBERS IN BOLD) SHOW THAT PARLOT HAS BETTER OVERALL PERFORMANCE. THE **AVG** ROWS CONTAIN THE AVERAGE OF ITS ABOVE 3 VALUES.(FIG 1)

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.B.1	1.60	1.81	3.93	1.53	3.05	1.21	3.61	1.53	2.08
pinMain.B.4	1.67	3.78	2.43	2.91	4.04	2.83	3.54	2.93	2.92
pinMain.B.16	3.68	7.80	9.90	5.45	4.41	3.81	3.51	2.41	4.65
AVG	2.32	4.46	5.42	3.30	3.83	2.62	3.55	2.29	**3.22**
pinAll.B.1	2.01	3.03	4.63	2.43	6.86	1.66	9.10	1.56	3.20
pinAll.B.4	3.95	8.96	3.52	7.67	10.25	5.83	6.18	3.70	5.81
pinAll.B.16	6.55	12.53	6.47	10.40	5.45	4.01	6.35	4.73	6.61
AVG	4.17	8.17	4.87	6.83	7.52	3.83	7.21	3.33	**5.21**
callgrind.B.1	12.47	19.56	17.07	9.67	7.82	11.32	15.66	10.50	12.47
callgrind.B.4	7.83	12.64	6.07	6.18	2.89	20.14	4.70	12.04	7.69
callgrind.B.16	10.95	28.44	9.23	7.90	4.04	8.89	8.55	6.16	9.00
AVG	10.42	20.21	10.79	7.92	4.92	13.45	9.64	9.57	**9.72**

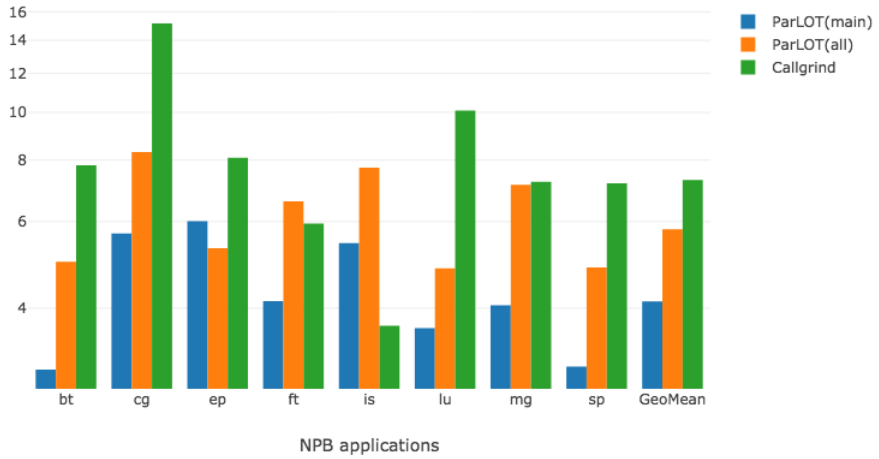

Fig. 1. Slowdown of ParLOT(main,all) and Callgrind. Each bar is the average slowdown of each tool on each application for 1, 4 and 16 nodes (16, 64 and 256 cores). Last group of bars is GeoMean (from bold numbers in table III).

TABLE IV
STAT: SD TOOLS: PINMAIN , PINALL , CALLGRIND , INPUTS: B , NODES: 1 , 4 , 16 , DESC: PRIMARY

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.B.1	1.67	1.48	3.40	1.93	1.93	1.10	2.08	1.04	1.71
pinMain.B.4	1.63	1.57	1.69	2.41	2.33	1.48	1.53	3.87	1.95
pinMain.B.16	2.23	1.65	2.08	1.82	2.66	1.84	1.57	1.60	1.90
AVG	1.84	1.57	2.39	2.05	2.31	1.47	1.73	2.17	1.85
pinAll.B.1	1.97	2.55	4.15	2.73	3.71	1.55	3.96	1.31	2.53
pinAll.B.4	2.33	2.68	2.84	4.03	4.59	2.41	2.89	3.47	3.07
pinAll.B.16	4.20	2.68	4.85	2.71	8.53	3.34	2.81	2.98	3.70
AVG	2.83	2.64	3.95	3.16	5.61	2.43	3.22	2.59	3.10
callgrind.B.1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
callgrind.B.4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
callgrind.B.16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
AVG	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

TABLE V

THE PURPOSE OF THIS TABLE IS TO SHOW THAT PARLOT IS DOING BETTER JOB ON LARGER INPUT SIZES. NAS BENCHMARK INPUT SIZES ARE AS FOLLOWS : $size(A) < size(B) < size(C) < size(D)$. THE OVERHEAD IT ADDS TO THE APPLICATION IS SMALLER FOR INPUT SIZE C AND I BELIEVE THE REASON IS THE REDUNDANT CAPTURED DATA (FUNCTION CALLS) FOR EACH RUN HELPS THE PERFORMANCE OF COMPRESSING PROCESS, THUS HELPS THE OVERALL PERFORMANCE. I CAN RUN THESE EXPERIMENTS WITH SMALLER INPUT SIZE (A) OR LARGER (D) AND INCLUDE THEM IN THIS TABLE. RUNNING NAS APPLICATIONS WITH *A* MAKES RUNNING TIMES SO SMALL (LESS THAN A SECOND FOR SOME OF APPLICATIONS) AND RUNNING WITH *D* IS GOING TO CONSUME A LOT OF SUS, IF WE DECIDE TO DO THAT. ALSO THESE ARE THE RESULTS WHEN I USE THE LATEST VERSION OF *Pin* WHICH IS **3.5**. TABLE VI (NEXT ONE) IS IDENTICAL TO THIS TABLE BUT USING VERSION **3.0** OF *Pin*

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.B.1	1.60	1.81	3.93	1.53	3.05	1.21	3.61	1.53	2.08
pinMain.C.1	1.39	1.26	3.12	1.22	1.95	1.07	1.49	1.21	1.50
pinMain.B.4	1.67	3.78	2.43	2.91	4.04	2.83	3.54	2.93	2.92
pinMain.C.4	1.79	2.79	3.22	1.78	3.35	1.38	2.93	1.63	2.24
pinMain.B.16	3.68	7.80	9.90	5.45	4.41	3.81	3.51	2.41	4.65
pinMain.C.16	2.88	4.38	4.38	3.53	5.14	3.24	3.50	2.17	3.54
pinMain.B.64	5.07	9.31	7.78	6.67	10.19	6.73	5.56	5.30	6.87
pinMain.C.64	5.08	8.56	6.61	7.35	7.44	5.21	3.94	3.81	5.77
pinAll.B.1	2.01	3.03	4.63	2.43	6.86	1.66	9.10	1.56	3.20
pinAll.C.1	1.49	1.56	3.72	1.41	3.25	1.22	2.65	1.16	1.87
pinAll.B.4	3.95	8.96	3.52	7.67	10.25	5.83	6.18	3.70	5.81
pinAll.C.4	2.27	3.42	4.08	2.52	5.00	2.15	4.78	1.83	3.05
pinAll.B.16	6.55	12.53	6.47	10.40	5.45	4.01	6.35	4.73	6.61
pinAll.C.16	4.89	3.75	5.48	4.38	6.21	3.82	5.04	2.56	4.38
pinAll.B.64	7.38	8.72	6.58	5.88	8.31	7.78	6.88	9.37	7.53
pinAll.C.64	7.37	7.30	5.77	6.38	6.40	4.80	4.53	5.19	5.88

TABLE VI

THIS TABLE IS SIMILAR TO PREVIOUS ONE (TABLE V) BUT WITH VERSION **3.0** OF PIN. THERE ARE SOME ZEROS IN THE TABLE WHICH SHOWS THOSE EXPERIMENTS HAVE CRASHED (PROBABLY BECAUSE OF INCOMPATIBLITY OF PIN WITH CURRENT CONFIGURATION ON PSC NODES). ALSO SOME OF THE SLOWDOWNS ARE SMALLER THAN 1. THUS THE VALUES OF THIS TABLE ARE NOT ACCURATE AND PROBABLY INVALID (AT LEAST FOR LARGER NUMBER OF CORES). BUT I JUST WANTED TO INCLUDE THEM HERE, IN THE FIRST DRAFT OF PAPER, TO SHOW SOME DIFFERENCES BETWEEN PIN VERSIONS AND IN CASE YOU ARE WONDERING WHY THE NUMBERS THAT I HAD IN OCTOBER WAS MUCH BETTER THAN THESE NUMBERS (TABLES THAT CAME PREVIOUSLY), THE REASON IS PIN VERSIONS.

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.B.1	1.54	1.81	3.92	1.71	4.74	1.37	3.88	1.47	2.26
pinMain.C.1	1.40	1.25	3.60	1.30	2.98	1.24	1.64	1.27	1.68
pinMain.B.4	2.59	3.58	3.76	2.71	4.62	2.20	7.22	2.57	3.40
pinMain.C.4	1.66	2.70	3.59	1.82	3.01	1.57	2.82	1.61	2.24
pinMain.B.16	3.57	4.71	3.79	4.48	3.77	3.18	1.91	2.25	3.32
pinMain.C.16	0.95	3.30	6.55	1.88	1.96	3.09	3.75	0.00	0.00
pinMain.B.64	5.39	3.04	2.42	3.65	0.00	3.01	2.83	1.48	0.00
pinMain.C.64	1.91	6.75	1.82	5.03	4.21	1.56	6.91	0.89	2.88
pinAll.B.1	1.69	2.31	5.00	1.99	5.96	1.54	5.57	1.57	2.73
pinAll.C.1	1.43	0.52	4.42	1.39	3.34	1.30	1.97	1.30	1.63
pinAll.B.4	1.69	3.89	4.36	5.15	4.40	2.97	5.25	3.54	3.71
pinAll.C.4	2.09	2.48	4.32	1.99	3.46	1.89	3.35	0.45	2.14
pinAll.B.16	4.48	5.09	7.41	2.96	4.70	5.13	2.67	3.45	4.27
pinAll.C.16	2.71	6.39	6.02	2.66	8.30	3.89	3.22	3.53	4.23
pinAll.B.64	9.06	2.65	2.81	0.68	3.08	2.88	3.22	4.20	2.93
pinAll.C.64	6.57	0.00	4.65	1.74	2.38	2.56	0.00	3.66	0.00

TABLE VII
STAT: SD TOOLS: PINMAIN , PINALL , INPUTS: B , C , NODES: 1 , 4 , 16 , 64 , DESC: PRIMARY

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.B.1	1.67	1.48	3.40	1.93	1.93	1.10	2.08	1.04	1.71
pinMain.C.1	2.52	1.05	4.63	1.98	1.57	0.62	1.96	1.06	1.63
AVG	2.09	1.27	4.01	1.96	1.75	0.86	2.02	1.05	1.67
pinMain.B.4	1.63	1.57	1.69	2.41	2.33	1.48	1.53	3.87	1.95
pinMain.C.4	2.78	1.49	4.19	1.99	2.68	1.32	3.43	0.91	2.10
AVG	2.21	1.53	2.94	2.20	2.50	1.40	2.48	2.39	2.02
pinMain.B.16	2.23	1.65	2.08	1.82	2.66	1.84	1.57	1.60	1.90
pinMain.C.16	2.00	2.84	5.19	1.32	4.07	2.45	3.05	1.80	2.61
AVG	2.12	2.25	3.64	1.57	3.37	2.15	2.31	1.70	2.25
pinMain.B.64	2.59	2.64	2.04	2.59	2.38	2.69	2.41	2.82	2.51
pinMain.C.64	2.39	2.80	2.41	2.18	2.48	2.48	2.75	2.39	2.48
AVG	2.49	2.72	2.23	2.38	2.43	2.58	2.58	2.60	2.50
pinAll.B.1	1.97	2.55	4.15	2.73	3.71	1.55	3.96	1.31	2.53
pinAll.C.1	1.42	1.28	3.32	2.21	2.87	0.64	2.59	1.66	1.79
AVG	1.69	1.92	3.74	2.47	3.29	1.09	3.27	1.48	2.16
pinAll.B.4	2.33	2.68	2.84	4.03	4.59	2.41	2.89	3.47	3.07
pinAll.C.4	3.40	3.75	8.72	5.22	7.27	1.68	3.29	1.18	3.59
AVG	2.87	3.21	5.78	4.62	5.93	2.04	3.09	2.33	3.33
pinAll.B.16	4.20	2.68	4.85	2.71	8.53	3.34	2.81	2.98	3.70
pinAll.C.16	2.90	2.58	5.04	2.18	4.49	4.73	5.45	2.13	3.45
AVG	3.55	2.63	4.95	2.45	6.51	4.04	4.13	2.55	3.58
pinAll.B.64	5.28	4.91	4.01	5.02	4.37	6.09	4.82	6.31	5.05
pinAll.C.64	4.68	4.83	4.68	4.73	4.82	5.65	4.97	4.73	4.88
AVG	4.98	4.87	4.34	4.88	4.60	5.87	4.89	5.52	4.96

TABLE VIII
THIS TABLE IS SHOWING THE REQUIRED BANDWIDTH FOR EACH APPLICATION (KILOBYTES PER CORE PER SECOND). $ReqBW_x = TraceSize_x(KB)/(\#ofcores)_x/Runtime_x(S)$ BECAUSE OF THE CRASHING PROBLEMS OF CALLGRIND ON *C* INPUT AND 64 NODES, I ONLY INCLUDE *B* INPUT AND 1, 4, AND 16 NODES RESULTS. CLEARLY PARLOT(MAIN) IS BEATING CALLGRIND WHILE THEY BOTH GENERATE THE SAME INFORMATION (I STILL BELIEVE PARLOT(MAIN) GENERATED TRACES ARE MORE INFORMATIVE AND RICH). PARLOT(ALL) BANDWIDTH IS THE HIGHEST BUT WITH CAPTURING ALL OF THE FUNCTION CALLS WITHIN A SINGLE EXECUTION, THERE IS NO SURPRISE. ANOTHER INTERESTING FACT FROM THIS TABLE IS, FOR PARLOT(MAIN), BANDWIDTH DROPS FROM **0.62** FOR 16 CORES TO **0.27** FOR 256 CORES (GOOD SCALABILITY). IT IS THE OPPOSITE FOR CALLGRIND WHERE THE REQUIRED BANDWIDTH JUMPS FROM **3.28** (KB/S) FOR 16 CORES TO **33.06** (KB/S) FOR 256 CORES. I ALSO HAVE THE RESULTS OF REQUIRED BANDWIDTH OF PARLOT FOR 64 NODES(1024 CORES) AND INPUT C BUT I DID NOT INCLUDE THEM HERE BECAUSE I DID NOT HAVE THEM FOR CALLGRIND (EXPLAINED ABOVE). FIG 2 VISUALIZE THESE NUMBERS (AVERAGE VALUES)

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.B.1	2.54	22.73	0.99	0.33	0.46	0.10	0.43	0.06	0.62
pinMain.B.4	3.20	18.31	0.52	0.20	0.12	0.11	0.23	0.10	0.46
pinMain.B.16	2.20	9.66	0.09	0.14	0.06	0.08	0.19	0.13	0.27
AVG	2.65	16.90	0.53	0.22	0.21	0.10	0.28	0.10	**0.45**
pinAll.B.1	20.80	39.89	14.75	21.05	31.00	19.55	34.37	12.37	22.53
pinAll.B.4	21.85	33.10	22.15	17.05	16.19	35.84	34.53	35.99	25.81
pinAll.B.16	25.31	29.33	23.31	20.07	23.78	56.58	29.09	42.96	29.57
AVG	22.65	34.11	20.07	19.39	23.66	37.32	32.66	30.44	**25.97**
callgrind.B.1	0.90	2.40	2.53	4.02	24.33	1.39	14.64	1.22	3.28
callgrind.B.4	4.27	9.11	12.02	16.78	59.03	2.97	35.78	5.20	11.25
callgrind.B.16	17.48	12.87	46.08	51.94	87.05	19.13	48.00	33.18	33.06
AVG	7.55	8.13	20.21	24.25	56.80	7.83	32.81	13.20	**15.86**

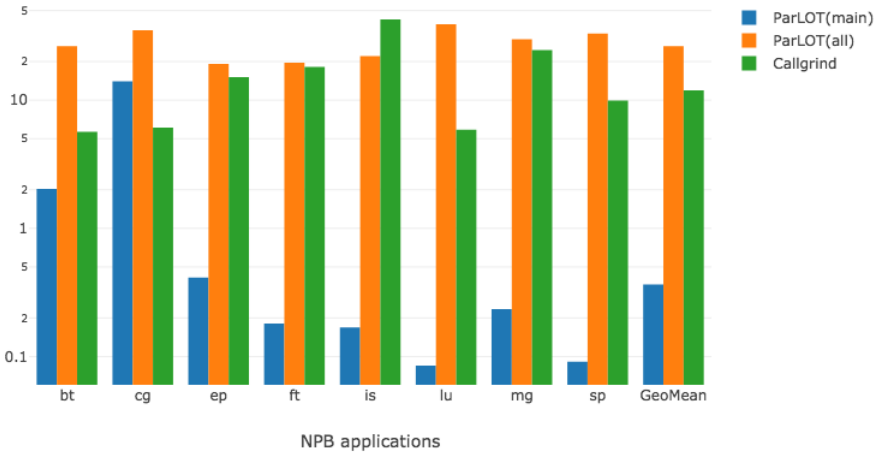

Fig. 2. Required Bandwidth (KB) per core per second for ParLOT and Callgrind. (Input: B)

TABLE IX
SIMILAR TO PREVIOUS TABLE (TABLE VIII) FOR ALL NUMBER OF NODES FOR **C** INPUT

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.C.1	1.08	13.81	1.26	0.13	0.66	0.03	0.20	0.02	0.34
pinMain.C.4	1.49	26.18	0.92	0.15	0.18	0.07	0.19	0.05	0.40
pinMain.C.16	2.81	18.28	0.41	0.13	0.08	0.08	0.14	0.07	0.34
pinMain.C.64	1.69	10.24	0.09	0.06	0.03	0.04	0.09	0.06	0.17
AVG	1.77	17.13	0.67	0.12	0.24	0.06	0.15	0.05	**0.31**
pinAll.C.1	7.42	22.04	6.47	9.30	22.51	8.11	20.92	3.73	10.43
pinAll.C.4	20.76	60.33	13.91	23.08	28.33	36.31	30.00	20.85	26.71
pinAll.C.16	26.08	65.31	20.39	29.43	26.92	71.97	28.06	41.71	35.13
pinAll.C.64	31.06	47.34	17.55	20.57	21.77	61.31	23.31	42.41	30.25
AVG	21.33	48.76	14.58	20.59	24.88	44.42	25.57	27.18	**25.63**

TABLE X
COMPRESSION RATIOS FOR B AND C INPUTS AND 1, 4, 16 AND 64 NODES.

	bt	cg	ep	ft	is	lu	mg	sp	GM
pinMain.C.1	14051.90	85.18	12104.15	31406.62	36422.85	20423.90	1169.56	22561.69	7393.79
pinMain.C.4	8249.54	85.36	12067.24	24222.90	33655.89	11191.63	650.18	10441.99	5189.93
pinMain.C.16	1985.31	85.55	11921.80	11426.88	25812.61	7353.92	353.38	4810.38	3048.85
pinMain.C.64	709.30	85.74	11374.25	6738.89	13360.80	6916.77	250.95	4624.28	2174.51
AVG	6249.01	85.46	11866.86	18448.82	27313.04	11471.56	606.02	10609.58	4451.77
pinAll.C.1	2579.37	89.08	21213.12	6818.31	7698.78	135.16	89.48	272.48	978.90
pinAll.C.4	1441.93	301.17	13242.06	1412.63	1122.81	709.17	857.56	773.11	1199.60
pinAll.C.16	1954.84	413.29	6721.88	1531.17	1793.81	430.15	1317.55	820.22	1273.86
pinAll.C.64	1195.58	891.83	5537.37	3191.36	2461.46	676.34	2412.46	967.72	1710.36
AVG	1792.93	423.84	11678.61	3238.37	3269.22	487.71	1169.26	708.38	1290.68
pinMain.B.1	5403.04	74.95	12067.06	12704.90	33655.87	8750.65	541.98	10426.05	4234.22
pinMain.B.4	2129.01	75.33	11922.18	10745.40	25812.57	5128.06	364.22	4042.15	2820.40
pinMain.B.16	729.64	75.70	11374.86	3508.41	13360.75	4371.76	205.67	3265.36	1746.25
pinMain.B.64	3551.97	76.08	9608.18	2790.59	4563.29	4031.59	164.19	4030.09	1750.60
AVG	2953.41	75.52	11243.07	7437.32	19348.12	5570.52	319.02	5440.91	2637.87
pinAll.B.1	750.27	68.40	16927.14	2584.15	2304.36	101.34	69.80	119.94	507.33
pinAll.B.4	1693.37	649.30	6797.11	2165.42	2402.91	727.52	836.96	672.73	1413.43
pinAll.B.16	1265.26	1138.20	4140.31	1794.78	1993.44	620.41	1477.61	883.86	1427.94
pinAll.B.64	1294.03	1218.33	5187.85	3086.80	3266.69	1044.65	2520.99	1167.23	1997.57
AVG	1250.73	768.56	8263.10	2407.79	2491.85	623.48	1226.34	710.94	1336.57

TABLE XI

TABLE XI, XIII, XV AND XVII ARE SHOWING THE DETAIL SLOWDOWNS ADDED TO THE CODE BY EACH PHASE OF PARLOT(MAIN AND ALL) FOR INPUT SIZES OF *B* AND *C*. I PUT MY OBSERVATIONS OF ALL FOUR TABLES OVER HERE. **NPIN** IS JUST THE SLOWDOWN CAUSED BY INITIALIZING PIN'S ROUTINES ON TOP OF THE TARGET APPLICATION WITHOUT DOING ANYTHING ELSE (NO INSTRUMENTATION, TRACING, COMPRESSION AND I/O. **DPIN** IS ALMOST IDENTICAL TO PARLOT EXCEPT IT STORES THE GENERATED COMPRESSED TRACES TO "/DEV/NULL". THE PURPOSE OF *dpin* IS TO SEE HOW MUCH OF THE OVERALL OVERHEAD IS BECAUSE OF I/O AND DATA-RELATED SLOWDOWNS. IN **WPIN**, AND ALL COLLECTED DATA WOULD BE STORED AS IS TO THE DISK. THE RESULTS OF THIS TOOLS SHOWS HOW MUCH EFFICIENCY OUR COMPRESSION APPROACH ADDS TO PARLOT. LAST ROW OF TABLES SHOWS GEOMETRIC MEAN OF EACH OF ITS ABOVE VALUES SHOWING HOW MUCH EACH PHASE OF PARLOT SLOWS DOWN THE NATIVE EXECUTION. IN GENERAL, WE ALL EXPECT THAT THE SLOWDOWNS OF $npin < dpin < ParLOT < wpin$. BUT MAJORITY OF NUMBERS ARE NOT LIKE THAT. FOR LARGER INPUT SIZES (TABLES XVII AND XIII) AND ALSO FOR FOR PARLOT(ALL) (TABLES XV AND XVII) THE GEOMEAN ROW NUMBERS MAKE MORE SENSE. I DOUBLE CHECKED THE RESULTS OF CHPC AND STAMPEDE AND THE PATTERNS ARE KIND OF IDENTICAL. IN MOST OF THE TABLE ENTRIES (IN PARTICULAR FOR SMALLER NUMBER OF CORES), THE DIFFERENCES BETWEEN THE AVERAGE SLOWDOWN OF *dpin* AND *ParLOT* IS VERY INSIGNIFICANT WHICH SHOWS THAT PARLOT IS NOT AN I/O-BOUNDED TOOL. ONE OF THE PROBLEMS THAT I HAD AND STILL HAVE IS RUNNING *wpin* ON INPUT *C*. I DID NOT HAVE THE RESULTS OF *wpin* ON *C* INPUT. WHILE I WAS PREPARING THESE TABLES AND I FELT SECURE ABOUT NOT WASTING SUS, I LET *wpin* RUN ON *C* INPUT. UNFORTUNATELY, MOST OF THE EXPERIMENTS CRASHED DUE TO A *PMPI* ERROR SO THE RESULTS ARE UNRELIABLE. THAT IS WHY I PUT 0 IN TABLE XIII AND XVII UNDER *wpin*. FIGURES 3, 4 AND 5 VISUALIZE NUMBERS FROM THESE TABLES.

Input: B	Nodes :	1				4				16				64			
	Detail Tools:	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin
Main	bt	1.49	1.52	1.60	11.63	1.17	1.58	1.67	8.53	1.86	2.99	3.68	8.81	4.22	4.51	5.07	13.64
	cg	1.78	1.76	1.81	3.19	2.64	5.02	3.78	6.76	4.86	6.27	7.80	12.38	3.92	8.43	9.31	12.96
	ep	3.82	3.37	3.93	22.47	1.84	2.26	2.43	7.94	4.97	5.02	9.90	9.09	3.27	7.22	7.78	7.67
	ft	1.60	1.51	1.53	3.23	2.43	4.36	2.91	4.31	3.52	5.90	5.45	5.87	3.24	6.87	6.67	6.69
	is	3.61	3.55	3.05	14.60	3.20	3.68	4.04	5.88	3.85	4.82	4.41	4.67	4.80	9.99	10.19	10.46
	lu	1.39	1.20	1.21	1.58	1.85	2.07	2.83	3.52	1.64	3.00	3.81	5.53	3.17	6.77	6.73	13.70
	mg	3.91	3.51	3.61	3.84	3.19	3.34	3.54	3.96	2.95	3.36	3.51	3.82	3.70	5.57	5.56	5.70
	sp	1.25	1.43	1.53	1.65	2.13	4.38	2.93	3.63	2.98	3.48	2.41	3.86	4.28	5.78	5.30	8.56
	GM	2.11	2.03	2.08	5.00	2.20	3.11	2.92	5.26	3.11	4.18	4.65	6.21	3.79	6.71	6.87	9.45

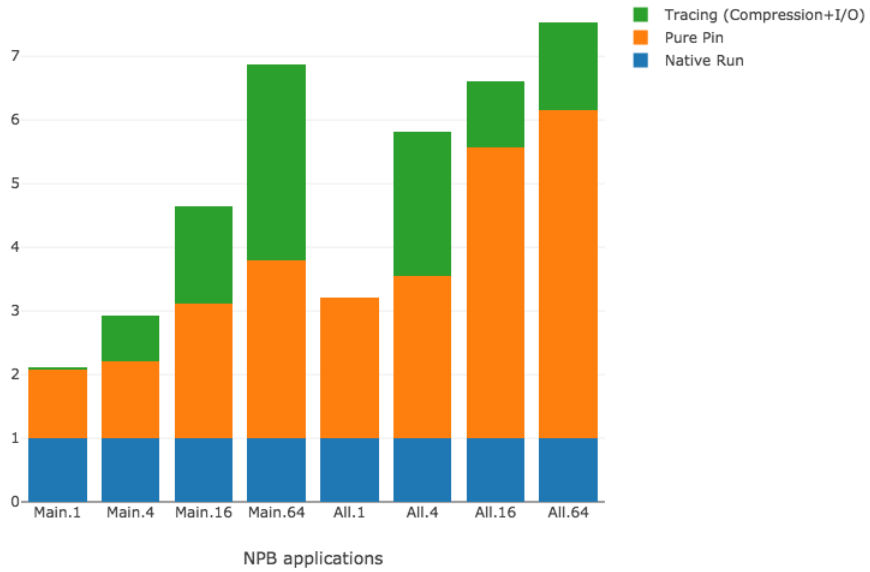

Fig. 3. Input size: **B**. Each bar is stacked value of slowdowns : $NativeRun = 1$, $PurePin = npin - 1$, $Tracing = ParLOT - npin$. Label of each bar is, (main/all).(1/4/16/64). This graph shows why ParLOT does not scale that well. The overhead that Pin itself is adding to the native run is growing with higher number of cores. The green part of each bar (tracing) is the overhead that our approach is adding. Fig 4 shows the effectiveness of our compression approach

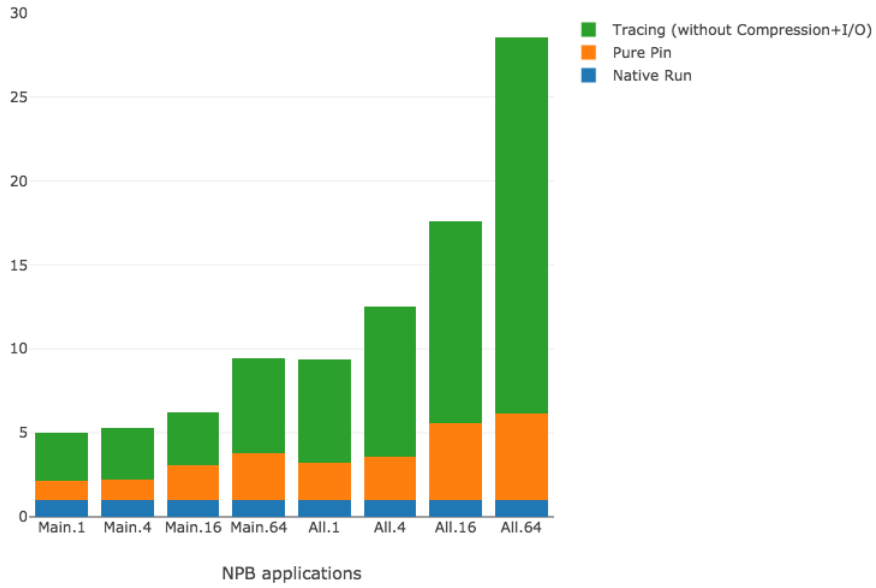

Fig. 4. Input size: **B**. Each bar is stacked value of slowdowns : $NativeRun = 1$, $PurePin = npin - 1$, $Tracing(w/o\_compression) = wpin - npin$. This graph clearly shows how much impact our compression method has on the performance of ParLOT.

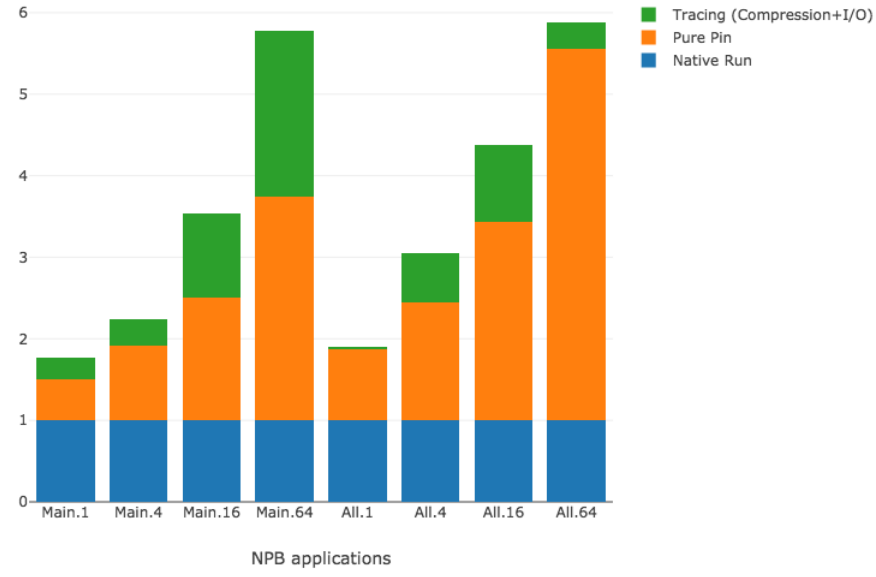

Fig. 5. Input size: **C**. This chart is similar to fig 3 but for larger input size C. As I mentioned in the table description, ParLOT seem to have better performance on larger input sizes. General shape of this chart matches fig 3 which shows the deterministic behavior of our tool and application.

TABLE XII
STAT: SD TOOLS: INPUTS: B , NODES: 1 , 4 , 16 , 64 , DESC: DETAIL REPORT

Input: B	Nodes :	1				4				16				64			
	Detail Tools:	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin
Main	bt	2.11	1.67	1.67	6.86	1.24	1.57	1.63	9.99	1.25	2.18	2.23	6.96	2.38	2.54	2.59	3.26
	cg	1.50	2.85	1.48	3.33	1.46	2.71	1.57	3.28	2.25	2.80	1.65	4.32	2.54	2.73	2.64	3.55
	ep	3.09	3.92	3.40	16.33	1.49	1.63	1.69	7.96	1.60	2.24	2.08	3.75	2.11	2.28	2.04	2.69
	ft	1.58	1.91	1.93	8.71	2.19	2.90	2.41	6.75	2.24	2.28	1.82	3.94	2.73	2.46	2.59	2.49
	is	2.01	3.00	1.93	2.03	2.35	2.32	2.33	4.60	2.41	3.88	2.66	5.24	2.33	2.26	2.38	2.31
	lu	1.16	1.23	1.10	1.21	1.71	2.72	1.48	2.58	2.50	2.83	1.84	3.05	2.61	2.54	2.69	3.11
	mg	2.25	2.05	2.08	4.22	1.38	2.40	1.53	2.71	2.09	2.45	1.57	3.04	3.04	2.87	2.41	3.09
	sp	0.98	1.80	1.04	3.26	1.89	2.11	3.87	8.24	2.49	2.49	1.60	2.81	2.82	2.58	2.82	2.82
	GM	1.73	2.17	1.71	4.27	1.67	2.24	1.95	5.11	2.05	2.60	1.90	3.96	2.55	2.53	2.51	2.89

TABLE XIII
INPUT: C , MAIN

Input: C	Nodes :	1				4				16				64			
	Detail Tools:	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin
Main	bt	1.35	1.38	1.39	0.00	2.21	1.70	1.79	0.00	2.18	2.76	2.88	0.00	3.61	4.64	5.08	0.00
	cg	1.24	1.79	1.26	0.00	2.06	2.69	2.79	0.00	2.03	4.54	4.38	0.00	5.66	8.73	8.56	0.00
	ep	3.58	3.36	3.12	0.00	2.32	3.16	3.22	0.00	3.43	4.26	4.38	0.00	4.53	6.86	6.61	0.00
	ft	1.29	1.23	1.22	0.00	1.56	1.77	1.78	0.00	2.57	3.41	3.53	0.00	4.33	7.55	7.35	0.00
	is	2.96	1.98	1.95	0.00	2.66	3.20	3.35	0.00	3.48	6.15	5.14	0.00	4.21	7.49	7.44	0.00
	lu	2.56	1.07	1.07	0.00	1.29	1.35	1.38	0.00	2.97	3.21	3.24	0.00	2.81	5.20	5.21	0.00
	mg	1.55	2.73	1.49	0.00	2.44	2.87	2.93	0.00	2.61	2.95	3.50	0.00	2.82	3.63	3.94	0.00
	sp	1.05	1.21	1.21	0.00	1.30	1.63	1.63	0.00	1.46	2.57	2.17	0.00	2.84	3.39	3.81	0.00
	GM	1.77	1.71	1.50	0.00	1.91	2.18	2.24	0.00	2.50	3.58	3.54	0.00	3.74	5.63	5.77	0.00

TABLE XIV
STAT: SD TOOLS: INPUTS: C , NODES: 1 , 4 , 16 , 64 , DESC: DETAIL REPORT

Input: C	Nodes :	1				4				16				64			
	Detail Tools:	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin
Main	bt	1.01	1.22	2.52	8.85	1.09	2.75	2.78	11.55	2.76	3.39	2.00	29.71	2.03	2.45	2.39	0.00
	cg	0.93	0.98	1.05	1.36	2.09	2.59	1.49	3.10	2.27	3.07	2.84	5.65	2.28	2.66	2.80	0.00
	ep	6.58	3.88	4.63	43.66	3.41	4.18	4.19	27.31	4.70	4.86	5.19	18.92	2.27	2.24	2.41	0.00
	ft	1.38	1.99	1.98	13.92	2.87	1.99	1.99	11.81	2.16	2.20	1.32	6.57	2.33	2.55	2.18	0.00
	is	2.56	1.54	1.57	1.84	2.06	2.00	2.68	3.78	3.94	3.96	4.07	5.42	2.49	2.62	2.48	0.00
	lu	0.51	1.16	0.62	1.22	1.22	2.54	1.32	1.95	2.29	2.90	2.45	3.84	2.38	2.27	2.48	0.00
	mg	1.93	3.37	1.96	2.30	1.92	3.32	3.43	3.91	3.08	2.97	3.05	3.52	2.31	2.52	2.75	0.00
	sp	1.04	1.06	1.06	2.12	1.29	0.92	0.91	3.80	1.88	1.95	1.80	2.33	2.52	2.27	2.39	0.00
	GM	1.47	1.66	1.63	4.10	1.85	2.35	2.10	5.79	2.76	3.05	2.61	6.59	2.32	2.44	2.48	0.00

TABLE XV
INPUT: B , ALL

Input: B	Nodes :	1				4				16				64			
	Detail Tools:	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin
All	bt	1.79	1.74	2.01	9.85	1.65	1.80	3.95	11.66	2.82	4.91	6.55	19.22	5.53	7.54	7.38	35.77
	cg	3.41	2.76	3.03	5.50	4.47	4.54	8.96	15.72	6.74	10.29	12.53	26.52	5.35	7.62	8.72	28.50
	ep	4.85	4.46	4.63	71.39	3.06	3.27	3.52	23.90	10.03	10.18	6.47	19.03	4.96	5.46	6.58	11.81
	ft	2.40	2.31	2.43	7.40	4.09	4.08	7.67	10.68	5.63	6.23	10.40	15.70	5.97	5.42	5.88	17.76
	is	7.07	6.35	6.86	21.77	5.50	5.40	10.25	10.21	7.04	4.25	5.45	8.92	8.73	6.01	8.31	19.58
	lu	1.73	1.55	1.66	2.75	2.76	5.27	5.83	14.45	4.34	2.55	4.01	26.05	6.39	4.49	7.78	83.68
	mg	8.60	8.32	9.10	11.11	5.80	10.41	6.18	10.42	5.22	5.05	6.35	14.75	7.95	5.97	6.88	21.89
	sp	1.50	1.50	1.56	3.09	3.07	5.66	3.70	8.43	5.38	3.06	4.73	17.85	5.29	6.31	9.37	57.42
	GM	3.21	2.97	3.20	9.36	3.55	4.55	5.81	12.53	5.57	5.20	6.61	17.63	6.15	6.02	7.53	28.54

TABLE XVI
STAT: SD TOOLS: INPUTS: B , NODES: 1 , 4 , 16 , 64 , DESC: DETAIL REPORT

Input: B	Nodes :	1				4				16				64			
	Detail Tools:	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin
All	bt	1.53	1.97	1.97	9.39	2.17	2.30	2.33	33.46	3.45	4.12	4.20	72.18	4.20	4.59	5.28	0.00
	cg	2.14	2.29	2.55	4.05	2.82	4.97	2.68	43.98	4.43	4.81	2.68	149.23	4.28	4.93	4.91	0.00
	ep	4.02	4.78	4.15	16.34	2.92	3.02	2.84	9.24	4.19	4.74	4.85	6.41	4.24	4.74	4.01	25.39
	ft	2.64	2.72	2.73	5.39	4.03	3.98	4.03	12.50	4.54	4.90	2.71	9.93	4.40	4.79	5.02	68.34
	is	3.58	4.17	3.71	8.37	4.31	8.59	4.59	12.97	7.73	4.51	8.53	20.20	4.40	4.41	4.37	55.11
	lu	1.50	1.56	1.55	2.63	2.41	2.36	2.41	31.25	4.51	5.18	3.34	104.27	4.34	5.08	6.09	0.00
	mg	3.76	3.92	3.96	7.23	5.16	3.08	2.89	29.48	4.72	2.72	2.81	37.70	4.99	5.52	4.82	0.00
	sp	1.24	1.31	1.31	8.92	5.60	6.24	3.47	24.00	4.55	4.98	2.98	107.64	4.24	4.56	6.31	0.00
	GM	2.33	2.58	2.53	6.83	3.48	3.90	3.07	21.68	4.65	4.42	3.70	39.44	4.38	4.82	5.05	0.00

TABLE XVII
INPUT: C , ALL

Input: C	Nodes :	1				4				16				64			
	Detail Tools:	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin
All	bt	1.40	1.43	1.49	0.00	1.47	1.88	2.27	0.00	2.61	4.94	4.89	0.00	6.22	6.75	7.37	0.00
	cg	1.47	2.27	1.56	0.00	3.03	2.96	3.42	0.00	3.20	3.31	3.75	0.00	8.32	6.56	7.30	0.00
	ep	3.84	3.48	3.72	0.00	3.14	3.92	4.08	0.00	4.85	5.16	5.48	0.00	5.06	5.33	5.77	0.00
	ft	1.48	1.41	1.41	0.00	2.15	2.16	2.52	0.00	3.79	4.09	4.38	0.00	7.48	7.74	6.38	0.00
	is	4.12	3.15	3.25	0.00	4.33	4.34	5.00	0.00	5.82	5.48	6.21	0.00	7.33	6.31	6.40	0.00
	lu	1.33	1.15	1.22	0.00	1.58	1.57	2.15	0.00	2.75	2.80	3.82	0.00	3.12	4.38	4.80	0.00
	mg	2.41	2.34	2.65	0.00	4.11	3.97	4.78	0.00	3.97	6.87	5.04	0.00	5.70	3.96	4.53	0.00
	sp	1.11	1.10	1.16	0.00	1.55	1.53	1.83	0.00	1.96	1.87	2.56	0.00	3.58	5.27	5.19	0.00
	GM	1.90	1.87	1.87	0.00	2.45	2.58	3.05	0.00	3.43	4.02	4.38	0.00	5.56	5.66	5.88	0.00

TABLE XVIII
STAT: SD TOOLS: INPUTS: C , NODES: 1 , 4 , 16 , 64 , DESC: DETAIL REPORT

Input: C	Nodes :	1				4				16				64			
	Detail Tools:	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin	npin	dpin	ParLOT	wpin
All	bt	1.09	1.43	1.42	8.50	1.36	3.36	3.40	0.00	4.35	5.81	2.90	0.00	3.83	4.35	4.68	0.00
	cg	1.14	1.18	1.28	7.20	2.15	2.30	3.75	0.00	4.22	4.62	2.58	0.00	3.93	4.23	4.83	0.00
	ep	7.93	3.31	3.32	24.69	8.97	3.90	8.72	24.88	8.30	8.56	5.04	28.00	4.23	4.32	4.68	0.00
	ft	1.84	2.19	2.21	10.65	4.02	2.64	5.22	12.74	3.53	4.18	2.18	18.84	4.11	4.48	4.73	0.00
	is	5.06	2.57	2.87	3.00	7.33	4.08	7.27	26.91	7.84	9.05	4.49	49.16	4.56	4.70	4.82	0.00
	lu	0.63	0.57	0.64	1.37	1.54	3.01	1.68	14.17	4.30	4.21	4.73	0.00	3.79	4.16	5.65	0.00
	mg	2.48	5.28	2.59	21.97	5.87	3.27	3.29	40.39	5.07	5.37	5.45	90.99	4.81	5.09	4.97	0.00
	sp	1.12	1.62	1.66	3.12	1.10	1.20	1.18	0.01	3.03	3.41	2.13	0.00	3.62	4.41	4.73	0.00
	GM	1.89	1.88	1.79	6.79	3.06	2.81	3.59	0.00	4.79	5.35	3.45	0.00	4.09	4.46	4.88	0.00