Towards Distributed Memory Parallel Program Analysis

Daniel Quinlan[1], Gergo Barany[2] and Thomas Panas[1]

[1] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, CA, USA
{dquinlan,panas}@llnl.gov
[2] Institute of Computer Languages
Vienna University of Technology, Austria
gergo@complang.tuwien.ac.at

Abstract. This paper presents a parallel attribute evaluation for distributed memory parallel computer architectures where previously only shared memory parallel support for this technique has been developed. Attribute evaluation is a part of how attribute grammars are used for program analysis within modern compilers. Within this work, we have extended ROSE, a open compiler infrastructure, with a distributed memory parallel attribute evaluation mechanism to support user defined global program analysis required for some forms of security analysis which can not be addressed by a file by file view of large scale applications. As a result, user defined security analyses may now run in parallel without the user having to specify the way data is communicated between processors. The automation of communication enables an extensible open-source parallel program analysis infrastructure.

Keywords. Program Analysis, Parallel Computing

1 Introduction

Parallel computing has not typically been applied to security analysis. This paper details work to define how significantly more computational resources can be brought to the static analysis of software. Central to our work is the ability to support global analysis of whole applications, either as a single application implemented over many source code files, or as a single binary. We target large scale applications of many millions of lines of code and similarly large scale binaries.

Checking hundreds of syntax and semantic properties of a large scale application requires many resources; especially time. For this reason, parallel evaluation is needed. Our choice of infrastructure to evaluate software in parallel is ROSE [1], an open compiler infrastructure for uniform analysis of both source code and binaries. ROSE is a source to source analysis framework that builds an AST as an immediate representation allowing many forms of static analysis. Previously, the analysis was performed serially. The advantage of choosing a AST based infrastructure is that both syntax as well as semantics (based on control flow information) can be checked. Our work is a framework for what will become significantly more expensive program analysis when more complex forms of program anaysis are handled, e.g. Satisfiability Modulo Theories (SMT) solvers for analysis supporting optimization and/or verification. Moreover, our goal for the future is not only to detect security faults but also to automatically transform programs to correct counterparts, if possible or to even create patches containing such fixes. Therefore, a source to source transformation engine is needed.

ROSE is even more than a source to source engine, it is a meta-tool that permits users to build custom tools for source code and/or binary analysis. Significant attention has gone into providing both a simple and extensible interface to the analysis of software. Because it is an infrastructure to support building many different types of tools, it also provides an infrastructure for *global analysis*. With global analysis we refer to the mechanism were the entire program is loaded into memory and analysis are performed directly on the entire program model. This is in contrast to file by file analysis, where different functions of a program are evaluated separately and summaries are created.

File by file handling can often become more complex to specify and slow when applied to iterative analysis requiring global context, such as inter-procedural data-flow analysis. For binaries the file by file approach is especially problematic since in many cases a large binary will be linked to form a single binary executable file.

Dagstuhl Seminar Proceedings 08161
Scalable Program Analysis
http://drops.dagstuhl.de/opus/volltexte/2008/1568

Our use of global analysis has three fundamental advantages of a file by file approach: 1) any local analysis can be handled by a global analysis, the global analysis infrastructure can be leveraged to automate communication of results across files; 2) iterative analysis can be more efficiently constructed without revisiting files (and redundant processing of common header files); 3) We expect that a global analysis can provide more precise analysis and reduce false positives. The disadvantage of global analysis is that it can be memory intensive. Because we seek to provide a general framework for security analysis we provide an interface to support global analysis and focus on parallelism as a means to address performance and memory resource issues for large scale applications.

In this paper we present our results of parallel program analysis for distributed security analysis using *Compass* [2], a tool built using ROSE which validates an easily specified set of rules defined as patterns on the AST (for source code and/or binaries). We present research work to define a simple to use, yet sophisticated parallel distributed attribute evaluation to express user defined global analysis on both source code and binaries targeting supercomputer architectures. For the size of applications that we target, both source code and binaries, the global representation significantly exceeds the capacity of standard desktop machines. In recent work we have analyzed several million line source code applications and binaries up to 150 MByte in size (containing 4.5 million instructions and over 27 million IR nodes).

2 ROSE/Compass

We are implementing our parallel program evaluation within ROSE [3], a U.S. Department of Energy (DOE) project to develop an open-source compiler infrastructure, which currently can process million line C, C++ and Fortran 2003[1] applications [3].

ROSE permits an attribute grammar based traversal of the AST with trivial attribute evaluation which supports both a simple programming model for users to specify security flaws and more importantly *parallel evaluation*.

For C and C++, ROSE uses the Edison Design Group C++ front-end (EDG) [4] to parse programs. EDG generates an abstract syntax tree (AST) and fully evaluates all types. Translated from the EDG AST, ROSE uses for its internal representation (IR) its own object-oriented AST, SAGEIII, which consists of 240 types of nodes for C and C++. For Fortran 2003 support, ROSE uses the Open Fortran Parser (from Los Alamos National Laboratory), and generates a similar object-oriented AST using many of the same IR nodes as for C and C++, but adding new IR nodes specific to Fortran 90 and Fortran 2003 support.

The 240 types of nodes preserve the high-level C, C++, and Fortran language representation so that no information about the structure of the original application (including comments and templates) is lost, thereby allowing ROSE to perform *sophisticated analysis* as well as *transformations* on the IR.

As an analysis infrastructure, ROSE can perform a wide range of analyses. Analyses may vary from general purpose analyses (that are part of ROSE), such as control flow analysis, data flow analysis, program slicing, etc. to domain-specific analyses (which are part of Compass), such as software quality analysis (e.g. memory issues, pointer issues, expression issues, etc.), data handling analysis (e.g. buffer errors, type errors, numeric errors, etc.), etc., cf. [5]. Domain specific analysis may be based on general analysis, e.g. a null pointer dereference analysis may utilize control flow and data flow information.

Tool builders, who do not necessarily have a compiler background, may utilize the ROSE infrastructure to build their own analysis tools.

Compass is a source code analysis tool built using ROSE to support the detection of violations of externally easily defined rules in source code. Our focus has been on secure coding rules. Where the rules can be defined on the AST, the rules are simple to specify using the language grammar for C, C++, and even for Fortran. Each rule is specified separately, and thus the rules can be evaluated independently. More complex rules may be defined on either the control flow graph (CFG), system dependence graph (SDG), call graph, class hierarchy graph or combinations of these. While most rules are defined on the AST, each may access the control flow graph or other program analysis results as required to express the evaluation of violations of defined rules.

In addition to the availability of general purpose analysis within ROSE and specific domain analysis within Compass, we are extending the ROSE infrastructure, in collaboration with academic groups, to in-

[1] Fortran 2003 is currently under development.

terface with general analysis tools, including PAG [6], OpenAnalysis [7], as well as analysis tools specifically for automated debugging and security, such as Osprey for measurement unit validation [8], MOPS for finite state machine-based temporal specification checking [9], and coverage analysis tools [10].

3 Global Program Analysis

Most static analysis tools use local information or function summaries to identify security flaws, and quite a few security flaws can be identified through this techniques. There are however a number of specific security flaws that are most efficiently solved using global analysis, either because it is efficent for users to specify then this way, or because the use of function summaries looses fundamental information that can make the analysis less precise (e.g. data flow along paths).

This section addresses two examples of global analysis to make it clearer why an extensible framework such as ROSE/Compass needs to provide mechanisms for global analysis. These examples illustrate why we have focused on the parallelization of global analysis instead of the parallelization of local analysis (which could be addressed trivially). Local analysis can be satisfied using a framework for global analysis, doing so risks only bringing greater analysis capability than required.

Since we define an extensible system for security analysis we can not readily anticipate when global analysis is required and it is up to the user to define and use ROSE/Compass as required for their analysis. We present two problems, both as part of the CERT Secure Coding Rules[11].

In summary, many complex analysis and specific security rules are most efficiently posed or required to be posed as global analyzes. Since we cannot aprior know what sort of custom analysis is required we have defined support in ROSE for global analysis. The contribution presented in this paper has been to parallelize the global analysis defined separately by the user's custom analysis.

3.1 One Definition Rule

The One Definition Rule (ODR) is a requirement for all C and C++ programs, and is required to support separate compilation. Basically, where there are two definitions for a declaration (typically though different header files used in an application spread across multiple files), the definitions are required to be token equivalent. Token equivalence is a stronger definition of equivalence than the two structures are the same size, or even that two data structures are structurally equivalent.

```
struct X { static const int a = 0; };
and
struct X { static const int a = 1; };
```

The example above shows two structs appearing in different files (separate compilation units). They are both the same size and structurally equivalent, but are not token equivalent. This is a trivial case of an ODR violation. These problems happen in numerous subtle ways within large codes and it requires global analysis to detect an ODR violation since each file by itself is a legal C/C++ code but together represent an illegal C/C++ program. Because of the requirement for seperate compilation in C/C++, this error is not identified by any compiler. Because all the application files must be seen at one time to detect the violation, this is a trivial example of a requirement for global analysis. The details of ODR violations were presented in [12] together with the global analysis algorithm required to detect ODR violations and its use as a security violation. Note that extending the function summaries to include declarations and saving the token sequence in the summary and processing the application twice would permit the violation to be identified, but this is not often possible.

3.2 Asynch-Safe Signal Handlers

In C/C++ *signal handlers* are functions which carry special restrictions, specifically they are required to be *asynch-safe* [11]. As C/C++ functions they could violate *asynch-safe* rules; which are not checked by the compiler. Even the definition of what is a signal handler can not be done until all the signal function calls have been analyzed. Typically the signal handler is the second parameter to the `signal()` function and

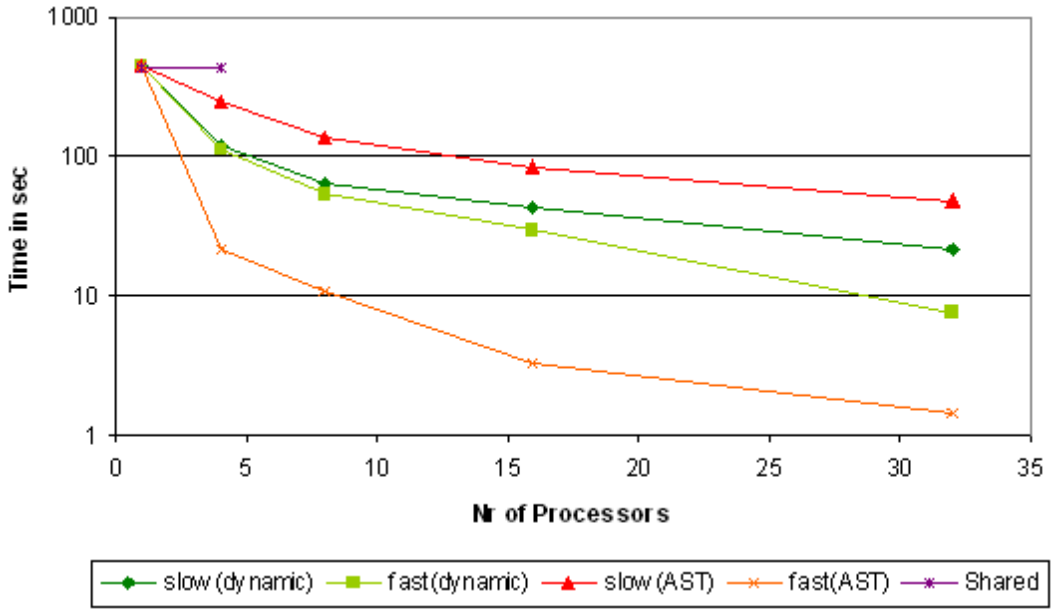

Fig. 1. Illustrated is the performance of compass analyses using a shared and distributed memory model. The distributed model uses two different algorithms, a) the AST size algorithm and b) the dynamic node weight algorithm. For both algorithms the fastest and slowest processor runs are shown.

passed as a function pointer, however it could be significantly more hidden than this by aliasing. To identify the signal functions is a local analysis, but must be performed across all application files. Once identified the analysis of the signal handler is an iterative analysis (since it could cascade to drive the analysis of other functions it calls). This iterative analysis is most efficiently posed as a global analysis, however iterative processing of all the applications files is theoretically possible, just not practical. Alternatively all functions could be analysized for asynch-safe behavior, but that is prohibatively expensive. Global analysis is a simple and efficient approach to checking signal handlers for asynch-safe properties.

4 Performance Optimization

The parallelization of static analysis is the key to achieving good performance in highly complex analysis of very large programs. As desktop systems are increasingly equipped with multiple CPU cores, program analyzers aiming at the best possible use of the available hardware must also become parallel.

Two different approaches at parallelization of Compass have been explored: The first one uses the *pthreads* library in a shared memory setting, running different groups of checkers on a shared AST; the second approach uses distributed memory and the *MPI* library, running the same checkers on different processors assigned to different parts of the load-balanced AST. This approach automates the required message passing to support global analysis, specifically the serialization and evaluation of synthesised attributes required to compute parallel program analysis results. The implementation of our parallelization work is not specific to Compass; it is part of the ROSE infrastructure and usable for general program analyses. Figure 1 shows results of the improvements in execution time by parallelizing Compass using the two methods. The results are discussed further in sections 4.1 and 4.3.

4.1 Shared Memory Parallelization

Figure 1 shows results of the parallelization using two and four threads on an AMD Opteron 2.4 Ghz processor, which is part of a high performance computer. In the shared memory approach a single copy of

the AST is present in memory and is traversed by several threads at a time. Each thread executes different checkers on the AST; as they all run in the same address space, no special effort is necessary to combine the final results of the computation.

Our results show that rather poor results can be expected from specific multi-core optimizations for static analysis common to a lot of checkers. In our experiment we can not observe a substancial performance gain. It seems that splitting up the analysis traversals into different threads has only little impact on performance, i.e. task parallel computation has no significant effects. The speedups achieved using this method are mostly limited by the available memory bandwidth as the work of Compass checkers consists mainly of memory accesses, many of them non-local. More work on this is expected in the future, since most distributed memory machines have shared memory node archetectures.

4.2 Decomposition of the AST for Distributed Memory Parallelization

Figure 2 shows the decomposition of the AST across processors, in this case four functions are distributed over four processors. Importantly, the number of processors used is independent of the number of files in the application. The figure defines five partitions of the AST, a black region at the root of the AST which is executed by all processors, and four colored regions assigned each to a single processor of the distributed memory machine. Because there is a portion of the AST which is operated upon by all processors (the black IR nodes in Figure 2) we can not expect perfect speedup from the parallelization.

The AST in Figure 2 represents only a four functions and a small collection of AST nodes. However, for the analyses in this paper, we are evaluating *CXX_Grammar.C*, which is a ROSE generated file. *CXX_Grammar.C* consists of 15,502 functions and 475,486 nodes and is represented by one binary file in the size of 400 MByte. Similar AST's have been built from binaries where the binary exists only as a single file; the largest of these is from a 150 MByte binary (librose.so) and represented 4.5 million instructions and over 27 million IR nodes (but using only 6.5 GByte of memory to support global analysis). It is these sorts of large scale problems and larger that are the target of our parallel program analysis research.

Load balancing for the assignment of the AST subtrees to processors is done on a function level of granularity (independent of the number of files in the application). The load for each function is computed based on two algorithms:

- *AST size algorithm*. This algorithm calculates statically the amount of AST nodes contained in a function definition. Based on this information, the functions are distributed equally over all available processors.
- *Dynamic node weight algorithm*. This algorithm first determines statically the weight of each function. The weight is based on the most computationally expensive ROSE AST nodes, e.g. nodes that require more time because they are key contributors of expensive Compass checkers. For instance, the *SgPointerDereference* AST node is used for the expensive null pointer dereference analysis. This analysis tracks amongst others the *SgPointerDereference* AST node's associated variable back to the variables definition in order to determine whether the variable is NULL. Once each function is weighted, the functions are sorted ascendend by weight. Finally, at run-time, the functions are distributed to the available processors.

Currently, because a checker on each processor could use data within the AST assigned to any other processor we store the whole application's AST on each processor. Later work may address the optimization of memory usage, to partition the AST into separate pieces.

4.3 Distributed Memory Parallelization

For the shared memory execution, the attribute evaluation needed not to be changed because parallelization was done across the checkers (task parallel) and not across the AST (data parallel). In contrast, the distributed memory parallelization over the AST is done using a specialized attribute evaluation, specifically a distributed memory parallel attribute evaluation. This approach mixes a serialized inherited and synthesized attribute evaluation for the root region of the AST (shown in black) with full parallel processing at deeper levels of the AST (shown in red, green, blue, and yellow), see Figure 2.

Figure 1 shows results of the parallelization using multiple processors (p = 1, 4, 8, 16 and 32). The figure shows the slowest and fastest processor run for both, the *AST size algorithm*, referred to as slow(AST)

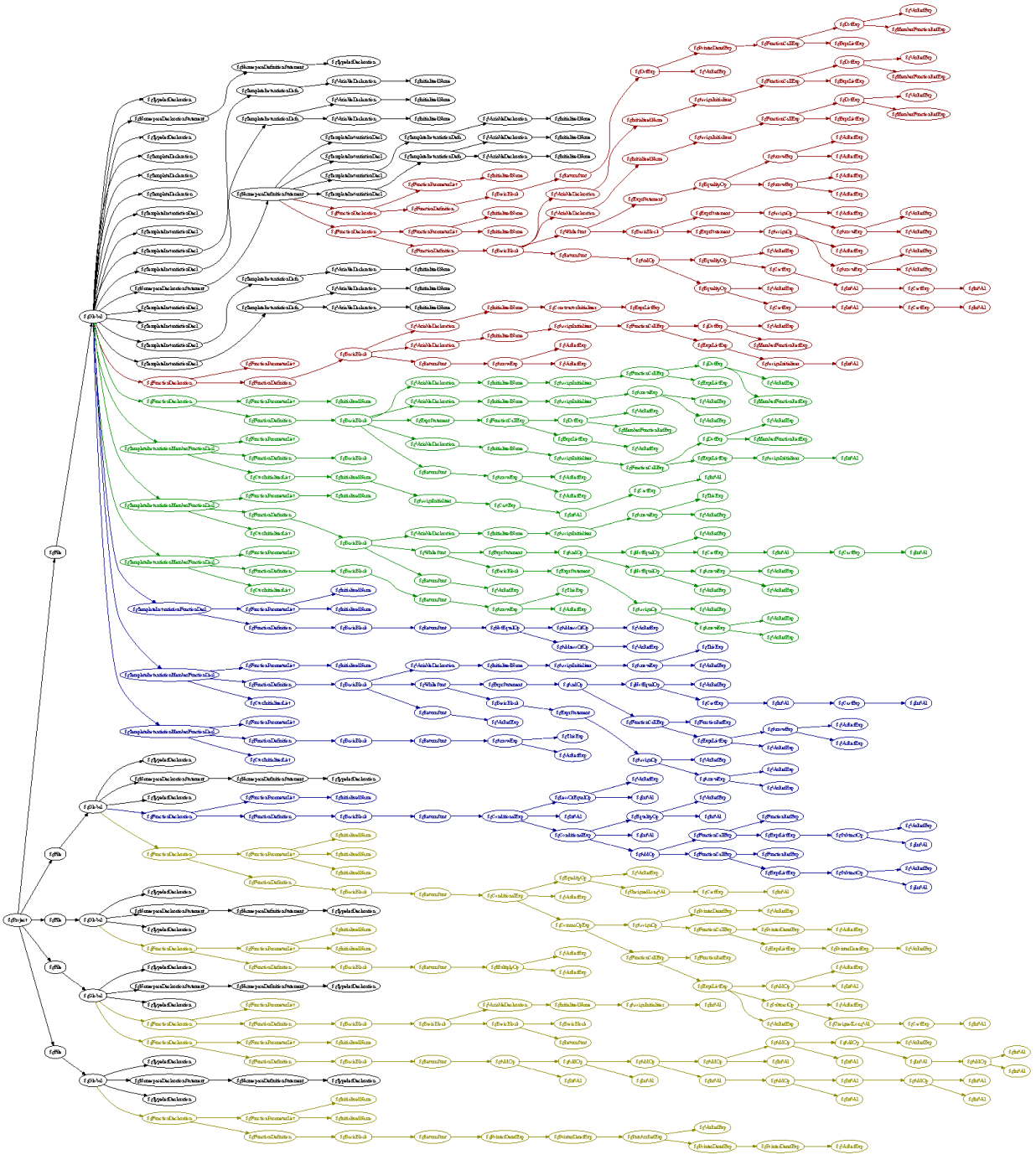

Fig. 2. The distribution of the AST is shown in different colors for 4 processors. This simple AST shows IR nodes of the AST that are processed in serial in black, and the IR nodes processed in parallel in red, blue, green, and yellow. In this example the distribution of data is done at the granularity of functions and the IR nodes in the functions are counted to compute a load balanced AST.

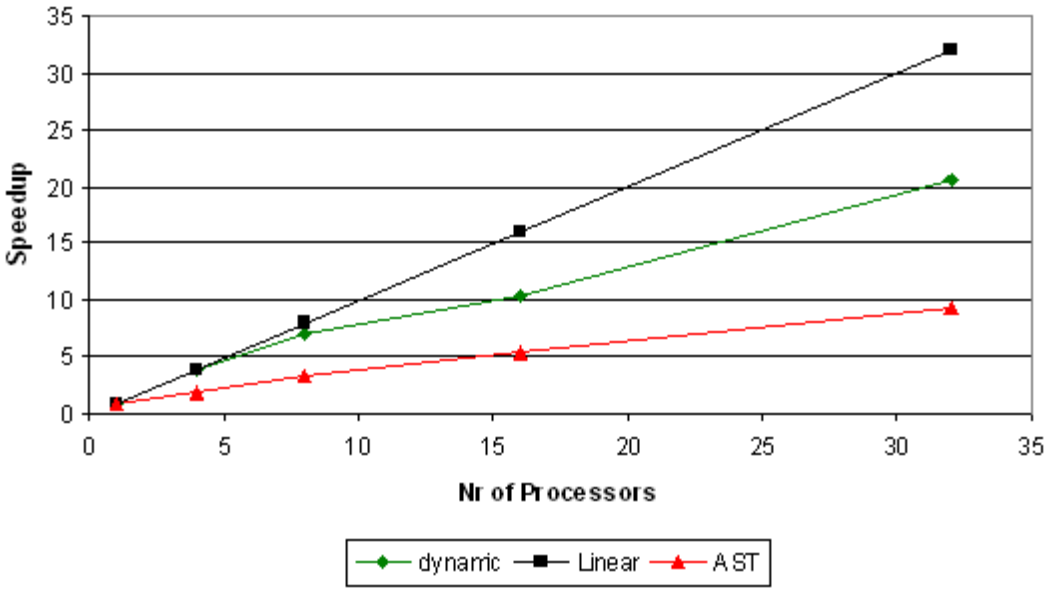

Fig. 3. This graph shows the speedup of our two load-balancing algorithms applied to a variety of processors on a super computer. The image reveals that the distributed memory approach scales well with the number of processors.

and fast(AST), and the *Dynamic node weight algorithm*, referred to as slow(dynamic) and fast(dynamic). Figure 1 reveals several interesting aspects. First and foremost it shows that parallel traversal is improving the performance of analysis substancially. Second, one can note that the dynamic algorithm is superior to the AST size algorithm. Finally, the load balancing indicated by the time gap between the fastest and slowest processor is much better for the dynamic algorithm. These are, however, initial results and future work will gather results from larger numbers of processors.

The speedup of our parallel execution is shown in Figure 3. The AST size algorithm reveals a speedup of about 10 for 32 processors while the dynamic node weight algorithm shows a speedup of about 20 for 32 processors. In the future we will investigate a parallel traversal algorithm that is based on AST nodes rather than functions. This should result in an even higher speedup.

While read-only analyses are usually usable in shared memory parallelization without any modification, in the distributed memory setting the results of the computations from different nodes must be collected and combined in a single process. For this reason distributed memory analyzers must implement a simple interface consisting of a function for serializing their relevant analysis data and a function for combining such serialized data from distributed sources. The data are then communicated using the MPI library, but this detail is completely hidden from the user.

5 Related Work

When parallel computing was attractive as a new research area (late 1980s and early 1990's) there were a number of papers on general program analysis for shared memory architectures, but it does not appear to have been successful and the emphasis on distributed memory appears to have made the subject overly complex. It can be expected that shared memory parallel work will be attractive again with the recent introduction of multi-core processors. This may become attractive and easy to introduce in commercial security analysis tools.

Recent work on Satisfiability Modulo Theory (SMT) solvers addresses a sufficiently expensive type of analysis that parallelization on clusters is an important optimization. Work within the Saturn project, a program analysis system, at Stanford[13] has addressed the distributed memory parallelization of SMT

solvers on clusters. In their paper they present results on a cluster of 80 processors, but appear to only get about 10-25% parallel efficiency. It is not clear that their results are easily comparable to our distributed memory results.

Automatic parallelization and multi-core optimizations can be expected to become popular compiler optimizations. However, within parallel computing for scientific programs they have not often been shown to be effective. It is not clear that any program analysis beyond local analysis run across multiple files can be parallelized trivially; and such techniques rule out the global analysis we have focused on in this paper.

Clearly there are a number of sophisticated security analysis tools focused on moderately large-scale applications. Using function/file summaries many tools can address large scale applications, working on one file at a time. The use of such summaries is however a possible problem for extensible techniques that target user defined rules or are built from accumulated rules built by external groups. Most commercial tools come with a large set of predefined rules and so can tailor their summary information to handle their own rule sets. Except that we know that most commercial tools are extensible, we are unfamiliar with the details or the level of the extensibility provided by commercial tools because we have not had access to the better (and expensive) commercial tools. We expect that all commercial tools address performance using combined traversals of the AST (if available) but we are unaware of any using parallelism for the analysis (shared or distributed).

6 Conclusions and Future work

We have presented a distributed memory parallel attribute evaluation and implemented it within the ROSE program analysis framework. We have demonstrated performance results using the Compass security analysis tool build on ROSE. We have presented both shared and distributed memory performance results. While our initial distributed memory results show speedup of a factor of about 20 for 32 processors, the speedup for the shared memory implementation for two threads is just above one. We expect that the shared memory performance is memory bandwidth limited, but we will investigate this further in future work. We expect that the distributed memory results will be better for larger problems on larger numbers of processors, but that the shared memory results on multi-core processors will not improve significantly (due to bandwidth limitations).

Central to this paper, we have presented initial performance results for the distributed memory parallel attribute evaluation for global analysis. As a result we have defined a simple interface to the development of general program analysis (at least those associated with aspects of security analysis as defined by Compass) for use on distributed memory architectures. We have demonstrated parallel distributed memory efficiencies of over 60% (per processor) for our dynamic load balancing algorithm and over 35% for the static AST based algorithm. The required serial processing associated with the inherited and synthesized attribute evaluation at the top of the AST is the main reason for the less than perfect parallel efficiency.

In the future, we want to investigate data parallel computation in a shared environment. We believe that at least the combination of shared and distributed data parallel computation will result in better analysis performance. Especially when programs become large, a combined approach will be useful. For instance, if a given input program is 4 GByte large and a computation node, consisting of 8 processors and 16 GByte RAM is utilized, then we can only use 4 processors on that node in distributed mode. However, when utilizing also a data parallel shared approach, we could use all 8 processors on that node.

Further, we have noted that certain analysis are so expensive that the computation over one function takes more time than all other functions combined. We have analyzed CXX_Grammar.C with a data-flow analysis (in particular a null pointer dereference analysis) on function granularity and one function can not be executed below 150 seconds. In future work we will thus also focus on the application of our distributed parallel analysis not only on function granularity but also on the finer granularity of AST nodes, whenever possible.

References

1. ROSE: An open source-to-source compiler infrastructure for analysis and optimization of C, C++, Fortran, and Binary applications. (2007) www.roseCompiler.org.

2. Compass: A tool for arbitrary software analysis (2008) www.roseCompiler.org/Projects/projects.html.
3. Schordan, M., Quinlan, D.: A source-to-source architecture for user-defined optimizations. In: Proc. Joint Modular Languages Conference. (2003)
4. Edison Design Group: (EDG front-end) www.edg.com.
5. MITRE Corporation: Common Weakness Enumeration (2007) http://cwe.mitre.org/cwe.mitre.org.
6. AbsInt, Inc.: PAG: The Program Analysis Generator (2006) www.absint.com/pag.
7. Strout, M.M., Mellor-Crummey, J., Hovland, P.D.: Representation-independent program analysis. In: Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. (2005)
8. Jiang, L., Su, Z.: Osprey: A practical type system for validating the correctness of measurement units in C programs. In: Proc. International Conference on Software Engineering, Shanghai, China (2006)
9. Chen, H., Dean, D., Wagner, D.: Model checking one million lines of C code. In: Proc. Network and Distributed System Security Symposium, San Diego, CA, USA (2004)
10. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded Java programs. Concurrency and Computation: Practice and Experience **15** (2003) 485–499
11. CERT: Secure Coding Standards (2007) www.securecoding.cert.org/confluence/.
12. Quinlan, D., Vuduc, R., Panas, T., Härdtlein, J., Sæbjørnsen, A.: Support for whole-program analysis and verification of the One-Definition Rule in C++. In: Proc. Static Analysis Summit, Gaithersburg, MD, USA, National Institute of Standards and Technology Special Publication (2006)
13. Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using boolean satisfiability. ACM Trans. Program. Lang. Syst. **29** (2007) 16