Joint NCC & IIS Bull., Comp. Science, 15 (2001), 125–135
© 2001 NCC Publisher

Overview of approaches to global static error analysis of parallel programs[*]

D. Yu. Perchine

There are lots of different approaches developed for global static program error analysis. Most of them are concentrated on sequential programs. Analysis of parallel programs is considered complicated. The overview contains brief information on obsolete and modern approaches to global static error analysis of parallel programs.

R. N. Taylor has proposed Call History Graph to analyse anomalies in parallel programs. R. Rugina and M. Rinard have developed context-sensitive flow-sensitive pointer aliasing analysis for multithread programs to detect regular errors. In LASER laboratory, the approach of computing MHP (May Happen in Parallel) information for a Java program has been developed. M. Dwyer has proposed the extension of the classical semi-lattice model of the data flow analysis to the complete lattice model.

1. Introduction

There are lots of different approaches developed for global static program error analysis. Most of them are concentrated on sequential programs. Analysis of parallel programs is considered complicated. That is why most of the advances in static analysis have been achieved for sequential programs only. Exponential behavior of algorithms for parallel programs was always the main problem, although we should note that context-sensitive flow-sensitive inter-procedural data flow analysis of sequential programs is also of the same complexity.

At present, most of the platforms heavily support different models of parallelism. As an example we will consider threads used in Java and POSIX threads for C. At the moment these models are of the most interest among others due to a wide use of Java and C with POSIX threads. We will mostly talk about Java, but everything can be applied to C with POSIX threads, as well.

We will mostly concentrate on context-sensitive inter-procedural global static error analysis of parallel programs. Its goal is to find as many errors in a program as possible. We will not review analyses focused on specific errors. We will not also concentrate on algorithms that detect only parallel specific errors.

[*]Supported by the Russian Foundation for Basic Research under Grant 0001-00820.

This article presents an overview of different approaches to global static error analysis of parallel programs. We will discuss its distinctions from analysis of sequential programs, the main directions and major problems.

2. Background

2.1. Basic parallel models

The most classical model is a Unix process model. It implies separated memory for different processes. The main problem here is that it is necessary to copy data memory every time a new process is created. Modern operating systems copy pages on write only, but this anyway place significant overhead on the system, however most Unices support the concept of System V shared memory. Such an approach is quite expensive compared to the thread shared memory, but it provides more protection against errors since only one process can map pages on write to the process own address space at the same time. Some Unix-like OS's support System V semaphores and BSD message queues.

At the period between late 70's and early 80's, the idea of light weight processes (processes with all memory shared) came back as an alternative to "heavy" Unix processes. The idea of processes with memory shared between them is rather old. Dijkstra [3] is usually mentioned as its inventor. Later such processes were called threads to distinguish them from Unix processes.

Programs which use the thread model of parallelism are called multithreaded. Usually these are Java or C/C++ with POSIX threads programs. In a multithreaded environment, the synchronization facilities are mutexes, semaphores, and conditions.

2.2. Parallel program models

There are different models of parallelism for parallel programs. Several aspects of a parallel model should be considered.

Memory models:

- shared memory; all memory is shared between all processes; typical examples are Java, POSIX threads;
- shared memory with global private variables; a part of memory is shared between some of the processes; we can consider Unix processes using this type of memory models in presence of System V shared memory;
- message passing; no memory is shared; information is transferred using messages between processes; its example is MPI (message passing

interface), which is used as a basic block of computing clusters.

Process creation models:

- the number of parallel parts is known; the language environment allows us to create a finite number of processes that should be specified before its run and requires them to be finished before one can create new processes; used in Ada and Parallel Fortran;
- one additional process can be created at any program position; binary parallelism; in this model one can create only one process at once, but there is no need to wait until it is finished; this allows one to create infinite number of processes; used in Java, POSIX threads and Unix processes;
- many additional processes can be created at any program position; the only difference from binary parallelism is that many processes can be created at once.

Synchronization models:

- all processes are started at the same time; a program is finished when all of them are finished (used in early Ada implementations, Parallel Fortran);
- all processes reach some position and wait for the others (rendezvous model, barrier synchronization; used in Ada);
- in one program only one (or n) process can execute a particular part of the program (critical sections). They are used in Java, POSIX threads and Unix processes. Critical sections are implemented using mutexes or semaphores in the n-th case;
- process(es) can wait for a signal from other processes. A process can send a signal either to one or to all processes (the same as above);
- processes can wait for other processes to finish (the same as above).

Parallel program models (their descriptions are in the corresponding sections):

- CHG (Concurrency History Graph) is used by Charles McDowell;
- PFG (Parallel Flow Graph) is equivalent to PEG and used by Rinard;
- PEG (Parallel Execution Graph) is used by LASER laboratory for Java.

2.3. Abstract interpretation

Data flow analysis of a program is mostly formalized by the so-called abstract interpretation invented by Patrick Cousot [2]. His approach is the following.

In static analysis, we need to find out some properties of a program. One of the ways is to instrument the program, run it on specific input data and derive these properties. This solution will give the most accurate result, but for specific input data only. Also, we should consider the case of programs which are never finished, like servers and operating systems.

Cousot has proposed a solution which allows us to obtain the properties without actual execution of a program.

Let us represent a program as a rooted directed graph, $G = (N, E, n_0)$, where N is a set of nodes, E is an edge relation over nodes, and $n_0 \in N$ is a distinguished start node, such as $\forall m \in N : (m, n_0) \notin E$. This graph is also known as Control Flow Graph (CFG). With each node we associate the properties, $X[n]$, we want to compute. Let us also define the set of predecessor nodes $Preds(n) = m : (m, n) \in E$.

We have a meet semi-lattice $L = (V, \sqsubseteq, \sqcap)$ formed from a set of properties V. A partial order $\sqsubseteq$ is defined on V. ϕ-function is used as a meet operation $\sqcap$. ϕ-function is a function used in the case when a node has more than one predecessor and we need to merge properties flowing from them. For example, we have a conditional node with 2 branches. At the point where these branches are merged, we have the property X_1 flowing from the first branch, and the property X_2 flowing from the second one. Application of the ϕ-function will produce the property $X = \phi(X_1, X_2) = X_1 \cup X_2$.

Let us define a function space over a meet semi-lattice as a set of functions, $f : V \to V$, defined over the lattice values. These functions are called transfer functions.

A meet semi-lattice is bound to the function space with a function map, $M : N \to F$. Let us denote $M(n)$ by f_n for simplicity.

We can formulate data flow analysis directly as a set of equations that give us the information for a node in terms of properties of its predecessors and the corresponding transfer functions:

$$\begin{aligned} X[n_0] &= \top, \\ \forall n \in N - \{n_0\} : X[n] &= \sqcap_{i \in Preds(n)} f_i(X[i]). \end{aligned}$$

Cousot has proved that if a function space is monotonic, then this set of equations always has a fixed point solution and it is unique.

2.4. General properties and problems of parallel program analysis

The only way we can analyse a parallel program by Classical Cousot approach would be treating the process creation sites as function calls, or, in other words, analysing a parallel program as a sequential one. Therefore, this approach [2] does not produce results suitable for parallel programs.

From wide practical experience with Wasp statical analyzer (`www.waspsoft.com`), we can conclude that such an approach will produce incorrect results. For some cases, incorrect messages are generated. Sometimes errors are not detected.

For example, we have a parallel program in which one thread initializes a variable x and waits for another thread to send a signal. Another thread uses the value of the variable x. It is possible that the analyzer that implements the above approach will produce 'Uninitialized variable usage' error message, which is obviously incorrect here. We will call such errors regular.

Parallel programs rise an additional set of problems in the static analysis. We can divide such problems into several categories:

1. Memory access violations (lack of synchronization). In parallel programs many processes can read and write memory at the same time. This is not a problem when we talk about values which are of a machine word size. They are read and written in one operation (although we should insert the memory barriers to make changes visible immediately for processes running on other processors). The problem appears when we face more complex data structures. If we have many processes that only read memory, this is not an issue here. But when we start writing to it, we should block other processes from both reading and writing. Otherwise they will read partially updated information which would be inconsistent.
2. Deadlocks. Deadlocks are the result of incorrect usage of locks. A classical example of a deadlock is when 2 processes are waiting for each other. This can happen if there are two resources and the first process locks the first resource and then the second one. The second process does other way around. It initially acquires the lock on the second resource, and then gets a lock on the first resource. Such a program can work perfectly for a long time, until both processes obtain their first locks in approximately the same time. After that they will wait for each other. This situation is just a simple example. In real life one can face much more complicated situations where the chain size can be of tens of locks.

The main problems of both situations is that a program will certainly work for some time without many problems. But after some time (usually when the system hits the load), these problems will appear either as crashes in completely unrelated places or as a program hang.

The deadlocks are easier to debug, because one can attach a debugger to the hanged program and try to figure out in which locations it is waiting for locks and what are call stacks and variable values.

In the cases of memory violation, the situation is much worse. One will always get suspicious crashes in completely unrelated locations. The only way to find the problem is to carefully examine the code. This solution can be unacceptable in some cases. It is nearly impossible to audit manually million lines of code in a reasonable time. That is why static analyzers are more needed for parallel programs than for sequential ones.

These errors will be denoted as parallel specific ones.

3. Algorithms used for parallel program analysis

3.1. State graph approach

In 1980, Taylor proposed to use Call History Graph (CHG) for analysis of anomalies in parallel programs [9], [8]. Analysis is able to detect both regular and parallel specific errors.

CHG is a sort of Petri nets called a reachability graph [6]. It denotes a structured history of all program states. It is a rooted directed graph, where nodes are program states. A state contains a control state and data state. The control state is the latest executed node in the synchronization graph (the graph represents synchronization inside the program). The data state is a set of properties we want to extract in this particular state of the program. There can be infinite number of different CHGs for a programs. Taylor has proposed an algorithm that builds a minimal CHG for any program.

The algorithm complexity and memory is exponential of the program size.

Later in 1987, Charles McDowell proposed [5] the way to build a reduced version of CHG. He uses several optimizations, like including only two identical processes in the CHG where multiple ones can be created, etc. Although the worst complexity still remains exponential, it was polynomial in average.

3.2. Rinard works

Radu Rugina and Martin Rinard [7] have developed the context-sensitive flow-sensitive pointer aliasing algorithm for multithreaded programs. Pointer Aliasing is one the basic problems in the static analysis. In languages with explicit (C, Modula-2, etc.) or implicit (Java) pointers, it is necessary for analysis to know which memory location each pointer variable may point to. This will help later to locate regular errors. The problem itself is proven to be unsolvable. It is only possible to find an approximation of this information.

They use the points-to graph to represent pointer alias information. The points-to graph has the location sets as its nodes. There is a direct edge from one location set to another if memory location represented by the first node may point to the memory location represented by the second node. The location set is a triple $\langle name, offset, stride\rangle$, where $name$ is a variable name, $offset$ is used to represent an array index, and $stride$ is used for recurrent structures. For example, the location set $\langle n, o, s\rangle$ represents all sets of locations $\{o + is | i \in N\}$ inside the block n.

Rinard analyses Cilk programs (Cilk is a multithreaded extension to C). Synchronization is not considered in their algorithm. With each program node, a triple $\langle C, I, E\rangle$ is associated, where C is the current points-to graph, I is the set of aliases created by other concurrent threads, E is the set of aliases created by the current thread.

The algorithm first initializes all points-to information with an empty set except for thread begin node, which is initialized with all pointers pointing to unknown location.

Sequential parts of the program are analysed using usual dataflow equations for pointer alias analysis.

In the point where the thread t_i starts, we should calculate the sets I_i and C_i as follows:

$$I_i = I \cup \bigcup_{1\le j\le n, j\ne i} E_j,$$
$$C_i = C \cup \bigcup_{1\le j\le n, j\ne i} E_j.$$

The logic behind is that we should include all edges created for other parallel threads in interference information I_i and points-to graph C_i flowing into a thread.

On the thread exit, we should merge information from the threads to be exited (Cilk can create multiple processes at a time) into sets E and C. We will use the union operator $\cup$ for merging the created edges into the set E' and intersection operator $\cap$ for merging points-to information C':

$$C' = \bigcap_{1\le i\le n} C'_i, \text{ where i enumerates all threads to be exited,}$$
$$E' = E \cup \bigcup_{1\le i\le n} E_i.$$

This combination of parallel and sequential dataflow equations is solved using the fixed-point approach.

This algorithm is running in polynomial time of the program size ($O(n^4)$).

3.3. Approach of LASER laboratory of University of Massachusetts

A Laboratory for Advanced Software Engineering Research (LASER) of the University of Massachusetts has developed an interesting approach to computing MHP information (May Happen in Parallel information for Java program; this information is extracted from a parallel program and could be used to find regular errors; for each statement in the program, a set of statements that may happen in parallel is associated) for parallel programs [1].

The data flow approach is used here. The Parallel Execution Graph (PEG) is used to represent a program. PEG is CFGs of all threads combined with multithread communication edges with functions inlined.

For convenience, calls to communication methods are labeled as (`object`, `name`, `caller`), where `name` is a method's name, `object` is an object owning the method `name`, and `caller` is the calling thread. Any or all of the triple elements can be replaced with `*`. For example, (`*`, `start`, `t`) will denote any start node in the thread `t`, or in other words, any place where threads started in the thread `t`.

The following functions are defined to map each node to the set of specific nodes of the predecessor type.

LocalPred(n) returns the set of all immediate local predecessors of n.

NotifyPred(n) returns the set of all notify predecessors of a notified-entry node n.

StartPred(n) returns the set of all start predecessors of a begin node n.

WaitingPred(n) returns a single waiting predecessor of a notified-entry node n.

Functions for successors LocalSucc(n), StartSucc(n), and WaitingSucc(n) are defined similarly.

$notifyNodes(obj) = (\texttt{obj}, \texttt{notify}, *) \bigcup (\texttt{obj}, \texttt{notifyAll}, *);$
$waitingNodes(obj) = (\texttt{obj}, \texttt{waiting}, *);$

$$NotifySucc(n) = \begin{cases} \{m | m \in (\texttt{o}, \texttt{notified-entry}, *) \wedge \\ WaitingPred(m) \in M(n)\}, & \text{if } n \in notifyNodes(o); \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$N(t)$ denotes all nodes for the thread $t \in T$. The function $thread : N \to T$, which maps each node to the thread it belongs, is also defined. Let $Monitor_{obj}$ denote the set of nodes for the monitor for the lock of the object *obj*, and $Monitor_{obj}(t)$ denote the part of the monitor executed by the thread t.

It is proposed to associate the sets M and OUT with each program node. The set M represents all statements which may happen in parallel with this node; The set OUT represents all MHP statements which should be propagated to the successor nodes.

Also, for each node we will compute intermediate sets $GEN_{notifyAll}$, GEN, and $KILL$. The set $GEN_{notifyAll}$ contains all nodes from which the node n could be notified. GEN is a set of nodes which can be executed in parallel with n's successors. $KILL$ is a set of nodes which cannot be executed in parallel with n's successors.

$$GEN_{notifyAll}(n) = \begin{cases} \emptyset, & \text{if } n \notin (\mathtt{obj}, \mathtt{notified\text{-}entry}, *); \\ \{m | \exists \mathtt{o} : m \in (\mathtt{o}, \mathtt{notified\text{-}entry}, *) \wedge & \\ WaitingPred(n) \in M(WaitingPred(m)) \wedge & \\ (\exists r \in N : r \in (\mathtt{o}, \mathtt{notifyAll}, *) \wedge & \\ r \in (M(WaitingPred(m)) \cap & \\ M(WaitingPred(n))))\}, & \text{otherwise.} \end{cases}$$

$$M(n) = m(n) \bigcup \begin{cases} (\bigcup_{p \in StartPred(n)} OUT(p) & \\ \backslash N(thread(n))), & \text{if } n \in (*, \mathtt{begin}, *); \\ ((\bigcup_{p \in NotifyPred(n)} OUT(p)) & \\ \bigcap OUT(WaitingPred(n))) & \\ \bigcup GEN_{notifyAll}(n), & \text{if } n \in (*, \mathtt{notified\text{-}entry}, *); \\ (\bigcup_{p \in LocalPred(n)} OUT(p), & \text{otherwise.} \end{cases}$$

$$GEN(n) = \begin{cases} (*, \mathtt{begin}, \mathtt{t}), & \text{if } n \in (\mathtt{t}, \mathtt{start}, *); \\ NotifySucc(n), & \text{if } n \in notifyNodes(o); \\ \emptyset, & \text{otherwise.} \end{cases}$$

$$KILL(n) = \begin{cases} N(t), & \text{if } n \in (\mathtt{t}, \mathtt{join}, *); \\ Monitor_o, & \text{if } n \in (\mathtt{o}, \mathtt{entry}, *) \cup (\mathtt{o}, \mathtt{notified\text{-}entry}, *); \\ waitingNodes(o), & \text{if } (n \in (\mathtt{o}, \mathtt{notifyAll}, *)) \vee \\ & (n \in (\mathtt{o}, \mathtt{notify}, *) \wedge |waitingNodes(o)| = 1); \\ \emptyset, & \text{otherwise.} \end{cases}$$

$OUT(n) = (M(n) \cup GEN(n))\ KILL(n).$

For each node after M(n) and OUT(n) are computed, there should be a symmetry step taken

$$\forall n_1 \in M(n_2) \Leftrightarrow M(n_1) = M(n_1) \cap n_2.$$

Except for this step, it is a usual forward data flow algorithm.

Time complexity of the algorithm is cubic.

3.4. Complete lattice approach

The most radical idea has been proposed by Matthew Dwyer [4].

Classical data flow analysis of programs is done on a semi-lattice. He has proposed to extend the algebraical model to a complete lattice. Mostly this approach is interesting in analysing regular errors.

There are two types of nodes in his approach: normal nodes, which exist in Cousot abstract interpretation and represent program statements, and synchronization nodes which represent synchronization in the program.

For merging data flows at normal nodes, ϕ-function should be used. For synchronization nodes, he introduced a new operator, ψ-function, used for merging data flows from parallel processes at synchronization nodes. For example, two branches are merged at the synchronization node. The property X_1 flows from the first branch and the property X_2 from the second one. The result of ψ-function will be $X = \psi(X_1, X_2) = X_1 \cap X_2$.

As it was mentioned above, the idea is quite radical, but it can give good results because it naturally describes the parallel program logic in an algebraic language and allows us to use lots of already proven algebraic results.

4. Conclusion

It should be noted that there are quite a lot of papers on static analysis of parallel programs, but the main problem still remains lack of the basic theory which will significantly ease and empower further development in this area. The only work that can be viewed as an attempt to build some sort of such a theory is Dwyer's one.

One of the further ways to extend approaches mentioned above would be a combination of Dwyer and Rinard ideas. The main drawback of the Dwyer approach is the following. He assumes that updates of the shared variables are merged only at synchronisation nodes. This is rarely the case. In the real-life programming languages, like Java, the parallel model is underdefined, and we should consider the worst case: shared variables are updated immediately.

To accomplish this, we can try to apply the ψ operator to previous synchronisation nodes of all threads involved in the current syncronisation operation, instead of only current ones. Although this will force us to do additional iterations to guarantee information propagation, it will allow us to meet requirements to real-life models mentioned above.

Acknowledgements. I would like to thank Charles McDowell here, as he quickly responded to my request of his paper and personally sent a copy of it.

References

[1] Naumovich G., Avrunin G. S., Clarke L. A. An efficient algorithm for computing MHP information for concurrent Java programs // Lect. Notes Comput. Sci. — 1999. — Vol. 1687. — P. 338–354.

[2] Cousot P., Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints // Conf. Record of the 4th Annual ACM SIGPLAN-SIGACT Sympos. on Principles of Programming Languages. — ACM Press, NY, 1977. — P. 238–252.

[3] Dijkstra E. W. Cooperating equential processes // Programming Languages. — Academic Press, 1968. — P. 43–112.

[4] Dwyer M. B. Data Flow Analysis Frameworks for Concurrent Programs. — Manhattan, 1995. — (Tech. Rep. / Department of Computing and Information Sciences, Kansas State University; № 8).

[5] McDowell Ch. E. A practical algorithm for static analysis of parallel programs // J. Parallel and Distributed Computing. — 1989. — № 6. — P. 515–536.

[6] Peterson J. Petri Net Theory and the Modeling of Systems. — Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[7] Rugina R., Rinard M. Pointer analysis for multithreaded programs // Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, May 1999. — № 5.

[8] Taylor R. N. A general-purpose algorithm for analyzing concurrent programs // Communs. ACM. — 1983. — Vol. 26, № 5. — P. 362–376.

[9] Taylor R. N., Osterweil L. J. Anomaly detection in concurrent software by static data flow analysis[1] // IEEE Trans. Software Eng. — 1980. — Vol. SE-6, № 3. — P. 265–278.

[1]Here there was the first proposal of CHG