Practical approach to access the impact of global variables on program parallelism

Thu-Trang Nguyen*, Hue Nguyen*, Quang-Cuong Bui*,
Pham Ngoc Hung*, Dinh-Hieu Vo*, and Shigeki Takeuchi†
* University of Engineering and Technology, Vietnam National University, Hanoi
Email:{trang.nguyen, 17021156, bqcuong, hungpn, hieuvd}@vnu.edu.vn
† GAIO TECHNOLOGY CO., LTD., Tokyo, Japan
Email: takeuchi.s@gaio.co.jp

***Abstract*—Global variables may have a significant impact on preventing programs from automatic parallelism. This paper introduces a practical approach to measure the effect of global variables on program parallelism. First, we conduct static data dependence analysis among program variables and represent such dependencies by a Variable Dependence Graph. Then, we analyze this graph for measuring and identifying which global variables have a significant impact on program parallelism. To evaluate this approach, we conduct experiments on 20 benchmark programs and an industrial application. The experimental results show that half of the studied programs contain large impact variables which may be the cause of preventing programs from parallel execution.**

***Index Terms*—Global variables, parallelism, program analysis, data dependence, variable dependence graph**

I. Introduction

Nowadays, with the development of the infrastructure and the requirement of performance improvement [1], the software industry has been driven by the multi-core processors. Specifically, in the automotive industry, the number of Electronic Control Units (ECUs) in one car is exponentially increasing. Initially, there were only 4 - 5 ECUs but currently, in modern vehicles, they have been integrated hundreds of ECUs [2].

To adapt this growth and utilize these multi-core computing resources, companies are on-demand to parallelize their programs [3] [4]. However, rewriting an application to enable parallelization is time-consuming and painstaking. Additionally, refactoring source code to create a parallel program is also wearisome because it requires changes in various lines of code. It is further error-prone and non-trivial because developers must guarantee non-interference of parallel operations [5]. Particularly, refactoring large systems such as embedded systems, which have a high demand for parallelization, is extremely challenging [6]. Those systems often perform a large number of computations and contain hundreds of thousands of code statements.

One reason which prevents programs from executing in parallel and makes it hard to refactor is data dependencies between code blocks, which are often caused by global variables. Global variables are used throughout the program, frequently make the coupling between program functions and increase risks of data concurrency problems [7], [8]. Using global variables has much been debated upon. Nevertheless, global variables are quite prevalent, especially in embedded systems. It is because embedded applications often suffer from limited memory available. Therefore, they often use global variables to handle shared data that is referenced throughout the application [9].

In the code statement perspective, if data dependencies between code blocks are weak, it may be easier to refactor and parallelize a program [10]. A solution to reduce data dependencies and to enable parallelization of a program is reducing dependencies caused by global variables. In order to do that, it is necessary to measure the impact of global variables on program dependence to figure out which variables have a large impact on preventing the program from executing in parallel. These large impact variables should be starting points to aid developers in mining and refactoring source code. However, it still lacks such studies.

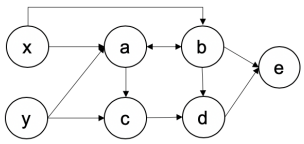
Regarding this topic, Binkley et al. introduced a method to measure the impact of global variables on program dependence [11], as well as figure out which vertices in the System Dependence Graph (SDG) lead to the mutually inter-dependencies of a large number of program statements [12]. They called such vertices are *linchpin*. This approach takes them 8 days to analyze and point out linchpin vertices of a 30.000 line-of-code program. However, that program is only a mid-sized program, and 8 days to analyze a program is still quite long. So, it is hard to apply this approach to analyze industrial programs.

In fact, in a program, two statements can not be executed at the same time if one statement is directly or transitively data-dependent on the other. In addition, statements are implemented by operations on variables. Therefore, data dependencies of program statements can be estimated by analyzing data dependencies between variables in the program. Furthermore, in a program, the number of variables is often much fewer than the number of statements. Thus, analyzing variable dependencies can help us save much time to find the causes that prevent the program from executing in parallel.

To solve this problem, in this paper, we introduce a practical approach to access the impact of global variables on program parallelism. In our approach, we conduct static data dependence analysis on the source code of the program. Then, data dependencies between variables are represented by a Variable Dependence Graph (VDG). In this graph, nodes are program

```
1  int x, y, a, b, c, d;
2
3  void foo(){
4      a = x + y;
5      b = sin(a + x);
6      a = 2 * b;
7      c = cos(a * y);
8      d = b - c;
9  }
10
11 int main(){
12     x = sin(70);
13     y = cos(230);
14     foo();
15     int e = b + d;
16     printf("%d", e);
17     return 0;
18 }
```

(a) Example source code



(b) Variable Dependence Graph

Fig. 1: Example of data dependencies between variables

variables and edges are data dependencies between variables. The probability to run a program in parallel is estimated based on dependencies between variables. If ignoring a variable's dependencies in a program, the higher parallelable probability of the program increase, the larger impact that variable has on program parallelism.

To demonstrate our approach, we conduct experiments on 20 benchmark programs and industrial automotive software. Our experimental results show that half of the studied programs contain variables which have a considerable impact on program parallelism. Also, in some programs, several variables have extremely large influences.

In summary, in this paper, our main contributions are:

- An approach to approximate program data dependence by VDG.
- A practical approach to measure the impact of global variables on program parallelism.
- An empirical study showing the effectiveness and scalability of this approach for industrial embedded systems.

The remainder of this paper is outlined as follows: Section II introduces background of this research. Our proposed approach to assess the impact of global variables on program parallelism is shown in Section III. Next, Section IV discusses our experimental results, and then related studies are reviewed in Section V. Section VI shows the main threats of this work. Finally, Section VII concludes our presentation.

II. Background

The major concept of our work is program variables. Instead of analyzing dependencies between statements, we estimate the probability to execute a program in parallel by analyzing data dependencies between variables.

Two variables are considered to have a data dependence relationship if the value of one variable is potentially influenced by the value of the other. We concern two kinds of relationship *influence* and *dependence*. These are two common terms in program analysis [13] [14], in this work, we define *influence* and *dependence* as follows.

Given a program P, V is the set of all variables in P, and a variable $v \in V$.

Definition 1: **(Influence).** We define influence function IS: $V \rightarrow 2^V$, IS(v) = $\langle$ a set of variables which influence v $\rangle$. Formally, $\forall v_i, v_j \in V, v_i \in IS(v)$ if:

- v_i is used to defined v or
- v_i is used to defined v_j and $v_j \in IS(v)$

Definition 2: **(Dependence).** We define dependence function DS: $V \rightarrow 2^V$, DS(v) = $\langle$ a set of variables which are dependent on v $\rangle$. Formally, $\forall v_i, v_j \in V, v_i \in DS(v)$ if:

- v is used to defined v_i or
- v_j is used to defined v_i and $v_j \in DS(v)$

Definition 3: **(Variable Dependence Graph [15]).** Variable Dependence Graph (VDG) of program P is a directed graph in which $\forall v \in V$ are graph nodes and it exists an edge from node v_i to v_j $(i \neq j)$ if v_i is used to defined v_j, $\forall v_i, v_j \in V$.

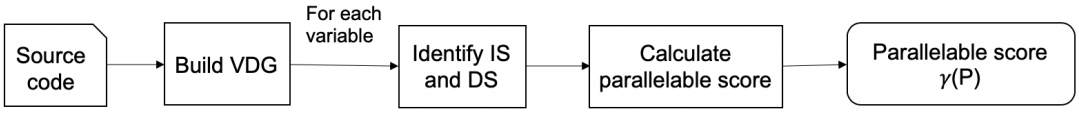
Fig. 1 (a) shows a simple code snippet and its corresponding VDG is represented in Fig. 1 (b). In this example, seven variables of the program $V = \{x, y, a, b, c, d, e\}$ are represented by seven nodes in the graph.

In the VDG, *IS(v)* is a set of nodes which has at least a path from those nodes to node *v*. Besides, *DS(v)* is a set of nodes which has at least a path from node *v* to those nodes. For instance, the influence set of variable *a* is IS(a) = {x, y, b}, since there are paths from *x*, *y* and *b* to *a* in the VDG. Its dependence set is DS(a) = {b, c, d, e}, since there are paths from *a* to these nodes.

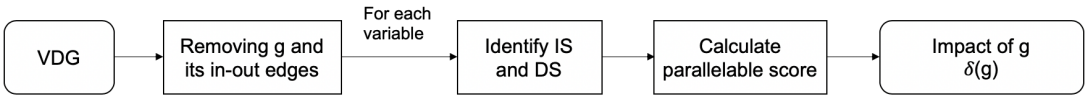
In Fig. 1 (a), value of variable *a* is influenced by the values of *x* and *y*, so we cannot define variables *a*, *x* and *y* at the same time. In addition, the values of variables *b, c, d*, and *e* are dependent on the value of variable *a*. This means that, variables *a*, *b*, *c*, *d*, and *e* also cannot be defined at the same time. Therefore, statement 12 (or 13), 4, 5, 6, 7, 8, and 15 cannot be executed in parallel.

Thus, $DS(a) \cup IS(a) = \{x, y, b, c, d, e\}$ is a set which consists of variables that cannot be defined at the same time with variable *a*. In other words, $DS(a) \cup IS(a)$ is a set of unparallelable variables of *a*. As a result, statements that define these variables cannot be parallelly executed with statements that define variable *a*.

In this example, $T = DS(x) \cup IS(x) = \{a, b, c, d, e\}$ is the set of the unparallelable variables of *x*. $V \setminus T = \{y\}$ is the set of variables that can be defined at the same time with *x*. So y is considered as a parallelable variable of x. In fact, there is no data-dependence between variables *x* and *y*. Consequently, these two variables can be defined at the same time, or two statements 12 and 13 can be executed in parallel.



(a) Calculating parallelable score of a program



(b) Measuring the impact of a global variable g on program parallelism

Fig. 2: Accessing the impact of global variables

Definition 4: **(Parallelable Variables).** Given a program P, and the set of variables V, set of parallelable variables of $v \in V$ is $K(v) = V \setminus (DS(v) \cup IS(v) \cup \{v\})$.

The main idea of our approach is that for two variables that do not have a dependence relationship (in terms of both influence and dependence), then they are parallelable variables of the other. Statements that define them can be executed in parallel. We take the total number of parallelable variables of all variables in the program to estimate the parallelable probability (parallelable score) of the program. If the total number of parallelable variables of variables in the program is large enough, the program will have more chances to execute in parallel.

Definition 5: **(Parallelable Score).** Given a program P and the set of variables V

- Parallelable score of variable v: $\varphi(v) = \|V\| - \|K(v)\|$.
- Parallelable score of program P: $\gamma(P) = \sum_{v \in V} \varphi(v)$

To measure the impact of a global variable g on program parallelism, we compare the parallelable score of the program with and without the presence of g.

Definition 6: **(Impact of a Global Variable on Program Parallelism).** Given a program P and G is the set of its global variables, impact of a global variable $g \in G$ on program parallelism is $\delta(g) = \gamma(P \setminus \{g\}) - \gamma(P)$, where $P \setminus \{g\}$ is the program P without the presence of g.

In practice, $\delta(g)$ may be a negative number, in this case, g is not the cause which prevents the program from being executed in parallel. Since $\gamma(P \setminus \{g\})$ may be smaller than $\gamma(P)$ when variable g is a parallelable variable of most of the remaining variables in the program, then removing g from the program may reduce parallelable score of those variables. In fact, reducing such variables does not affect the probability to execute the program in parallel.

III. Accessing the impact of global variables on program parallelism

This section introduces our proposed approach. The approach overview is shown in Fig. 2. Fig. 2 (a) demonstrates how to calculate parallelable score of a program. Fig. 2 (b) shows our approach to measure impact of a global variable on program parallelism. Detail of each step will be briefly described below.

Calculating parallelable score of a program

In order to calculate the parallelable score of a program, the process consists of several steps, they are: building VDG from source code, identifying IS and DS for each variable, and then calculating the parallelable score.

For a program, to build a VDG, first, we conduct data dependence analysis and identify data-dependent variables of each variable. In this step, we leverage srcSlice which is a lightweight slicing tool [16] [17]. VDG is generated by creating a directed graph. Each variable in the program is a node in the graph, V is the set of nodes. For each $v \in V$, we construct a directed edge from $v \in V$ to $v' \in V$ if v' is in the list of data-dependent variables of v, which is obtained by srcSlice.

In order to identify the influence set (IS) and dependence set (DS) of each node in the VDG, the VDG is analyzed to detect strongly-connected components (SCC) first. Then, all nodes in an SCC is replaced by a single representative node. In fact, any influence set or dependence set that consists of a node in an SCC will include all the others from that SCC. Therefore, replacing all nodes in an SCC by a representative node will help to reduce redundant work.

IS and DS of nodes are identified using graph traversal techniques. The IS(v) of node $v \in V$ is the set of nodes that are reachable from v. And, the DS(v) of node $v \in V$ is the set of nodes that have at least a path to v.

According to Definition 5, parallelable score of a variable $v \in V$ is $\varphi(v) = \|V\| - \|K(v)\|$. Then, parallelable score of the program is the total number of variables that can be parallely defined with each variable in the program, $\gamma(P) = \sum_{v \in V} \varphi(v)$.

Measuring the impact of a global variable g on program parallelism

Given a program P and its set of global variables G, in order to measure the impact of a global variable $g \in G$ on program parallelism, we compare the parallelable score of the program with and without the presence of variable g. To estimate parallelable score of the program without the presence

of g ($P \setminus \{g\}$), variable g and all of its in-out edges are removed from the VDG. The obtained graph is called $VDG \setminus \{g\}$.

Following, IS and DS of each remaining variables are calculated in this $VDG \setminus \{g\}$ graph. Next, parallelable score of each variable and the parallelable score of the program $P \setminus \{g\}$ can be obtained. The impact of this variable on program parallelism is $\delta(g) = \gamma(P \setminus \{g\}) - \gamma(P)$.

If g is ignored from the VDG and the program's parallelable score increases, it means that removing g from the source code increases chances to execute the program in parallel. The more parallelable score increases, the more impact g has on the program parallelism. In other words, the higher $\delta(g)$ is, the more likely g is the cause that prevents program components from executing in parallel.

In order to find out variables that tighten program functions and make them hard to run in parallel, we identify the impact of each global variable on program parallelism and figure out which variables have a large impact. In fact, this approach can be done for all variables in the program, however, local variables seem to have less influence on connecting program functions than global variables due to their scope. So, to reduce redundant work, we only concern global variables.

IV. Experiments and discussion

To evaluate this approach, we conduct experiments on a set of benchmark source code and an industrial program to answer the following research questions:

- **RQ1**: *Overall quantitative impact of global variables.* Over all programs, how many global variables have a considerable impact on program parallelism?
- **RQ2**: *Quantitative programs containing large impact global variables.* Whether large impact global variables occur in most of the programs or they are only used in some special cases? (This question makes more sense with the results for RQ1.)
- **RQ3**: *Causes of the large impact of the global variable on program parallelism.* What patterns are found on program dependencies with and without the presence of the large impact global variable?
- **RQ4**: *About time complexity* How long does this approach take to measure the impact of a global variable?

A. Subject system

Experiments were conducted on 20 programs, all written in C. They are used as benchmark source code in related studies [11], [18]. We obtained these programs from online repositories such as Free Software Directory[1] and Source Code[2]. Their size range from 6KLOC to 81KLOC. Table I briefly introduces these 20 programs. In this table, LoC is lines of code which is counted by the UNIX utility wc; SLOC is none-comment non-blank lines of code which is counted by the utility sloc [19]; and Variables column is the number of variables in the program.

[1]https://directory.fsf.org/wiki/Main$_{P}age$

[2]https://source-code.com

TABLE I: The 20 subject programs studied

Program	LoC	SLoC	Variables	Brief Description
a2ps	63,600	40,222	3658	ASCII to Postscript
acct-6.3	10,182	6,764	435	Process monitoring utilities
barcode	5,926	3,975	427	Barcode generator
bc	16,763	11,173	906	Calculator
ctags	18,663	14,298	1450	C tagging
diffutils	19,811	12,705	1483	File differencing
ed	13,579	9,046	460	Line text editor
empire	58,539	48,800	8313	Conquest Game
EPWIC-1	9,579	5,719	1137	Wavelet image encoder
findutils	18,558	11,843	1155	File finding utilities
flex2-5-4	21,543	15,283	991	Lexical Analyzer Builder
gnuchess	17,775	14,584	1045	Chess player
gnugo	81,652	68,301	9361	Go player
indent	6,724	4,834	433	Text formatter
snns	79,170	52,798	7634	neural network analyzer
termutils	7,006	4,908	292	Unix terminal emulation
time-1.7	6,965	4,185	135	CPU resource measure
userv	8,009	6,132	912	Access control utility
wdiff.0.5	6,256	4,112	174	Diff front end
which	5,407	3,618	180	Unix utility
sum	475,707	343,000	40,581	
average	23,785	17,165	2029	

In addition, we also show the effectiveness and scalability of our approach by conducting an experiment on the industrial embedded program provided by our partner[3]. This program size is approximately 300KLOC, it contains 6197 functions and 39770 variables, in which 3611 are global variables.

B. Experiment on benchmark source code

According to Definition 6, the unit of the impact of a global variable on program parallelism is the number of variables, i.e., the number of parallelable variables. So, whether the impact of a variable is considered to be large or not, it much depends on the program size. For ease of comparison, this paper uses the percentage of increased parallelable variables over the program's parallelable score. For instance, a program P, impact of a global variables $g \in V$ is $\delta(g) = \dfrac{\gamma(P \setminus \{g\}) - \gamma(P)}{\gamma(P)}$.

RQ1: Overall quantitative impact of global variables

In 20 programs, we evaluated the impact of totally 3682 global variables on their program parallelism. Most single globals do not have a significant impact, even removing them from the program, the probability to execute the program in parallel do not change. However, more importantly, there do exist some global variables that have a considerable impact on connecting other variables in the program. Specifically, reducing their dependencies will dramatically increase the chance to execute program components in parallel.

Fig. 3 shows 16 global variables of different programs which have highly significant impact. The y-axis shows how

[3]https://www.gaio.com/

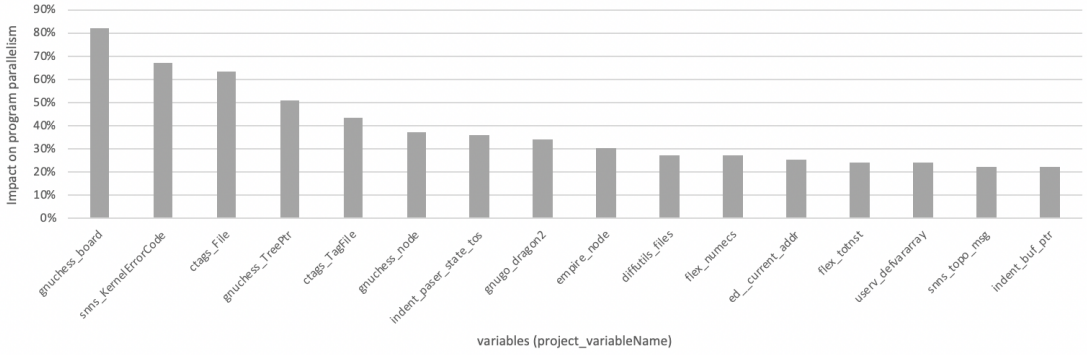


Fig. 3: Impact of ignoring dependencies for each single global variables on program parallelism

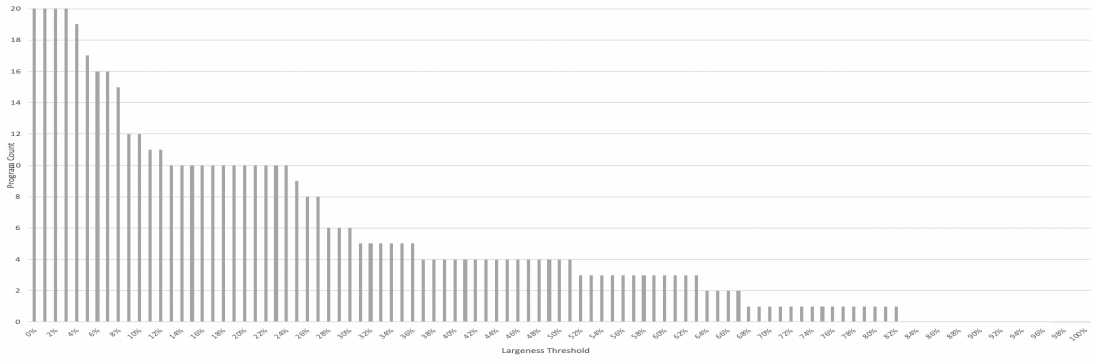


Fig. 4: The number of programs having large-impact variables for various largeness thresholds

the parallelable score of each program is increased if the dependencies of each variable shown on the x-axis are ignored. In the x-axis, variables are indicated by the format *ProgramName_VariableName*. For example, with *gnuchess_board* means that after removing variable *board* in the program *gnuchess*, the parallelable score of this program increased more than 80%.

The answer to RQ1 is that not all global variables have the same level of impact on program parallelism. There are only several variables that have a considerable impact on preventing program components from executing in parallel. During refactoring source code to enable parallelization, developers can consider reducing the dependencies of such variables.

RQ2: Quantitative programs containing large impact global variables

According to the answer of RQ1, there are only several global variables that have a large impact on program parallelism. Another question RQ2 is that whether such variables occur in general or they are only used in some cases.

Fig. 4 shows a count of programs with large-impact variables for various largeness thresholds. The y-axis shows the number of programs containing variables with the impact shown on the x-axis. At an extreme a threshold of zero, all 20 programs contain variables that have an impact 0% on program parallelism. Impact 0% means that removing each of these variables from the program, the probability to execute the program in parallel with and without their presence is unchanged.

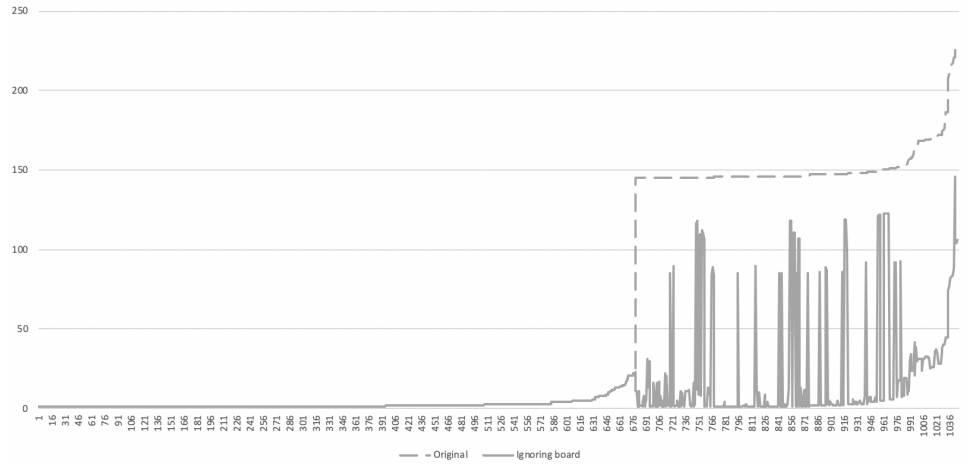
If the largeness threshold is set to be 10%, it means that a global variable g is considered to have a large impact if ignoring dependencies of g, then it will increase the program parallelable score 10%. In this case, there are 12 out of 20 programs contain large impact global variables. More strictly, if the largeness threshold is 20%, there are half of the programs containing large impact global variables.

Specifically, there are four programs that contain variables, which have an extremely large impact on program parallelism. Removing each of them will help to increase more than 50% parallelable scores of the program. And, in 20 studied programs, there do not exist any variables that have an impact on parallelism larger than 82%.

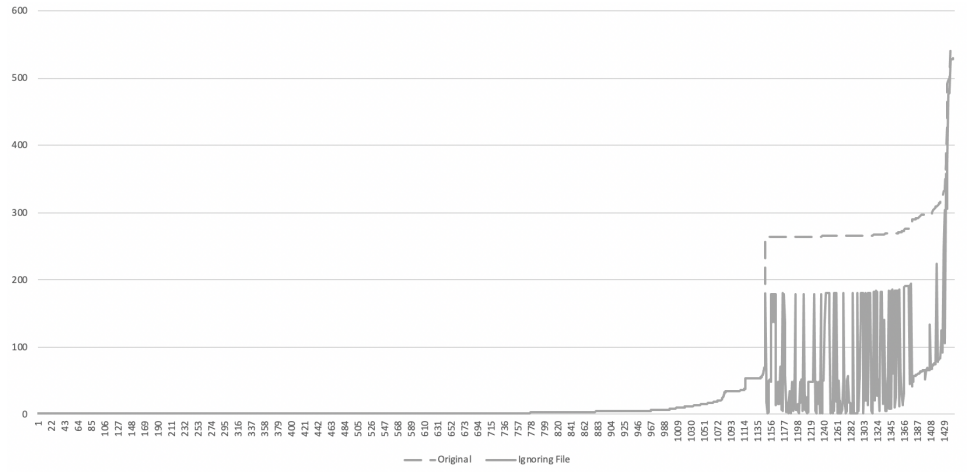
RQ3: Causes of the large impact of the global variable on program parallelism

The major purpose of this paper is to present the method to access the impact of global variables. We also show evidence to suggest that there are several single global variables that have a detrimental effect on program parallelism.

This research question gives a glimpse consideration on these large impact variables. The answer of this question can



(a)



(b)

Fig. 5: Impact of ignoring dependencies of variable *board* and *File* on IS and DS of other variables in program *gnuchess* and *ctags* respectively

give some suggestions for future studies about techniques to reduce the dependencies of these variables.

Fig. 5 shows the impact of ignoring dependencies of the variable *board* in the program *gnuchess* and the variable *File* in the program *ctags*, respectively. In these figures, the y-axis shows the number of variables in DS and IS of each variable whose id shown on the x-axis. The original program is shown in dash line, and the program after removing variable *board* (or variable *File*) is shown in the solid line.

In Fig. 5 (a), after removing variable *board*, the number of variables in the DS and IS of the other variables (id from 676 to 1036) in the program *gnuchess* dramatically drop. Also, in Fig 5 (b), after removing variable *file*, the number of variables in the DS and IS of the other variables (id from 1156 to 1429) in the program *ctags* fail significantly.

It demonstrates that there are many transitive dependencies and transitive influences are created via variables *board* and *File* in these two programs. In each Fig 5 (a) and 5 (b), at points, the larger differences between these two lines (the dash line and the solid line), the more transitive dependencies and influences the corresponding variables have via the ignoring variable.

RQ4: About time complexity

Table II shows the execution time that our experiment takes to measure the impact of global variables in each program. In this table, the time unit is second. Our approach analyses various programs in less than 10 seconds, three programs (i.e, *a2ps*, *gnugo*, and *snns*) take us several hundred seconds to analyze their source code. Especially, program *empire* takes us 1177 seconds to measure its global variables' impact. On average, it took us approximately 0.6 seconds to calculate the impact of a global variable on its program parallelism.

Furthermore, the number of variables is often much fewer than the number of statements so the number of nodes in VDG will be much fewer than the number of nodes in SDG. As a result, analyzing dependencies on the VDG helps us to save a large amount of computation space and execution time

TABLE II: Execution time

Program	Execution time (s)	Program	Execution time (s)
a2ps	109	acct-6.3	3
barcode	2	bc	5
ctags	14	diffutils	6
ed	2	empire	1177
EPWIC-1	8	findutils	5
flex2-5-4	4	gnuchess	10
gnugo	488	indent	4
snns	334	termutils	2
time-1.7	1	userv	5
wdiff.0.5	1	which	1

compared to previous studies.

Particularly, table I shows the number of statements (which is approximately equal to SLoC) and the number of variables. Each code statement is a node in the SDG and each variable is a node in our VDG. As shown in this table, the number of nodes in SDG is about 8 times higher than the number of nodes in our VDG. Specifically, with *wdiff.0.5* program, the number of statements is 23 times higher than the number of variables.

C. Experiment on industrial source code

This section discusses our experimental result on an industrial program that is provided by our partner. This result also shows the compliance with the experiments on benchmark source code, most of the single global variables do not have a significant impact on program parallelism. However, there also does exist large impact global variables.

For instance, there is one global variable that ignoring it causes the program's parallelable score to increase 8%. Specifically, there is one variable that has an extremely large impact. After ignoring the dependencies of this variable, the parallelable score of the program increased by 61%. Due to the security policy of industrial programs, we cannot provide more details about these variables.

About time complexity, it took us about 25414 seconds to analyze the program and measure the impact of 1377 global variables. Thus, it is practical and scalable to apply this approach to industrial source code.

In practice, the current compiler cannot automatically execute this program in parallel. Our partner is required to refactor this source code to parallelize it. However, with large source code, it is hard for developers to figure out which are the reasons preventing this program from automatic parallelization and which code block should be refactored.

Our solution suggests a method to access the impact of global variables and we can rank variables according to their impact on program parallelism. Large impact variables can be starting points to examine. Based on this suggestion, developers can consider reducing dependencies of large impact variables in the source code refactoring process.

V. Related work

There are several studies were conducted to measure dependencies among code statements. Binkley et al. [12], [11], [20] already introduced a solution to assess the impact of a global variable in program dependence and showed a method to figure out causes of large dependence clusters. A dependence cluster is a set of statements that all of them are mutually dependent on the others. However, this method works based on analyzing SDG so it costs a large amount of time and unscalable. In addition, removing a global variable to break dependence clusters into smaller ones can help to reduce dependencies between program functions but it is not enough to guarantee the parallelization of the program.

In order to deal with global variables in the parallelization problem of C/C++ programs, Sankaranarayanan et al. [7] proposed a method to localize global variables and then pass them as corresponding parameters of functions. Their tool transforms every global variable into locals. However, not every global variable prevents program parallelism. So refactoring all global variables causes the redundant changes of many statements, but indeed they do not need to change to be executed in parallel.

Similar to this idea, Smith et al. [21] also introduced an approach to localize global variables. In this research, they focus on localizing globals to make C programs thread-safe. However, users need to manually categorize global variables and then their tool will automatically localize globals variables in some categories to make them thread-safe.

VI. Threats to validity

This section considers the threats to the internal and external validity of the experiment results in Section IV.

One of the primary external threat arises from the possibility that the selected programs are not representative of programs in general. The studied source code contains various kinds of programs such as applications, utilities, and games. In addition, these programs' source code are used as benchmark for evaluation in other related research [11], [18]. So, it could be pretty confident in the obtained result. However, it would be premature to infer this result to other programming languages, because studied programs are all written in C.

An internal threat comes from identifying data-dependent variables of given variables. In other words, the threat may come from potential faults of the dependence analysis tool. In this experiment, srcSlice is employed. It is a lightweight tool, so it is fast, scalable, however less accurate. Alomari et. al. [17] has demonstrated their tool works well regarding the accuracy of slices when compare with CodeSurfer[4], a mature commercial slicing tool. To estimate the impact of global variables in program parallelism and the trade-off between effectiveness, scalability, and accuracy, it is reasonable to apply srcSlice in our dependence analysis step.

VII. Conclusion and future work

This paper introduces a practical approach to access the impact of single variables on program parallelism. Our approach analyzes data dependencies on the VDG to figure out

[4]https://www.grammatech.com/codesurfer-binaries

which variables may be the cause of preventing the program from automatic parallelism. This approach approximates data dependencies between statements by data dependencies between variables. So, this approach can reduce a large amount of analyzing time compared to previous studies, but it may be less accurate than analyzing the SDG. However, in terms of analyzing to find out the reasons to support other tasks such as program comprehension or source code refactoring, it is scalable and acceptable to apply this approach in the industry.

Our experimental result shows that most of the single variables do not have a significant impact, while few variables have a considerable impact on program parallelism. In addition, although there are few large impact variables, many programs (half of the studied programs) have at least one large impact global variable. Reducing dependencies of such variables may dramatically increase the chance to execute the programs in parallel.

Further work will consider patterns of variables that have a large impact. Also, we will research techniques to suggest/-support to reduce data dependencies of large impact variables to enable parallelization.

ACKNOWLEDGMENT

This research has been supported by GAIO TECHNOLOGY CO., LTD (https://www.en.gaio.co.jp).

This work has been partly supported by VNU University of Engineering and Technology under project number CN20.26.

REFERENCES

[1] R. Leupers, M. A. Aguilar, J. Castrillon, and W. Sheng, "Software compilation techniques for heterogeneous embedded multi-core systems," in *Handbook of Signal Processing Systems*. Springer, 2019, pp. 1021–1062.

[2] M. Dendaluce, "System-on-chip-based highly integrated powertrain control unit for next-generation electric vehicles: harnessing the potential of hybrid embedded platforms for advanced model-based control algorithms," *World Electric Vehicle Journal*, vol. 7, no. 2, pp. 311–323, 2015.

[3] R. Stewart, B. Berthomieu, P. Garcia, I. Ibrahim, G. Michaelson, and A. Wallace, "Graphical program transformations for embedded systems," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 647–649.

[4] Z. Han, G. Qu, B. Liu, and F. Zhang, "Task decomposition and parallelization planning for automotive power-train applications," in *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 2019, pp. 398–404.

[5] D. Dig, "A refactoring approach to parallelism," *IEEE software*, vol. 28, no. 1, pp. 17–22, 2010.

[6] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: a systematic literature review," *Software Quality Journal*, pp. 1–44, 2019.

[7] H. Sankaranarayanan and P. A. Kulkarni, "Source-to-source refactoring and elimination of global variables in c programs," 2013.

[8] G. Zheng, S. Negara, C. L. Mendes, L. V. Kalé, and E. R. Rodrigues, "Automatic handling of global variables for multi-threaded mpi programs," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 2011, pp. 220–227.

[9] N. Nguyen, A. Dominguez, and R. Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 3, pp. 1–32, 2009.

[10] J. Rafael, I. Correia, A. Fonseca, and B. Cabral, "Dependency-based automatic parallelization of java applications," in *European Conference on Parallel Processing*. Springer, 2014, pp. 182–193.

[11] D. Binkley, M. Harman, Y. Hassoun, S. Islam, and Z. Li, "Assessing the impact of global variables on program dependence and dependence clusters," *Journal of Systems and Software*, vol. 83, no. 1, pp. 96–107, 2010.

[12] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and Z. Li, "Efficient identification of linchpin vertices in dependence clusters," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 2, Jul. 2013. [Online]. Available: https://doi.org/10.1145/2491522.2491524

[13] J. M. P. Cardoso, J. G. de Figueiredo Coutinho, and P. C. Diniz, *Embedded Computing for High Performance: Source code analysis and instrumentation*. Morgan Kaufmann, 2017.

[14] M. S. Sadi, L. Halder, and S. Saha, "Variable dependency analysis of a computer program," in *2013 International Conference on Electrical Information and Communication Technology (EICT)*. IEEE, 2014, pp. 1–5.

[15] M. Matsubara, K. Sakurai, F. Narisawa, M. Enshoiwa, Y. Yamane, and H. Yamanaka, "Model checking with program slicing based on variable dependence graphs," *Electronic Proceedings in Theoretical Computer Science*, vol. 105, 12 2012.

[16] H. W. Alomari, M. L. Collard, J. I. Maletic, N. Alhindawi, and O. Meqdadi, "srcslice: very efficient and scalable forward static slicing," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 931–961, 2014.

[17] C. D. Newman, T. Sage, M. L. Collard, H. W. Alomari, and J. I. Maletic, "srcslice: A tool for efficient static forward slicing," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 621–624.

[18] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence clusters in source code," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 1, pp. 1–33, 2009.

[19] D. A. Wheeler, "Sloc count user's guide, 2005," 2015.

[20] D. Binkley and M. Harman, "Identifying 'linchpin vertices' that cause large dependence clusters," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 89–98.

[21] A. R. Smith and P. A. Kulkarni, "Localizing globals and statics to make c programs thread-safe," in *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, 2011, pp. 205–214.