# Context Sensitive MHP Analysis in Java

Dhruv Maroo, Soham Tripathy

**Abstract**

In this report, we outline the prior work done on MHP analysis, and try to extend the existing MHP algorithm with context sensitivity. We do a theoretical analysis of the new context sensitive algorithm and also use it on some example Java programs, comparing it's output with the context insensitive algorithm. We also discuss the limitations of the context sensitive algorithm.

## Overview

**MHP analysis**, i.e. *may-happen-parallal* analysis, is a very common way to determine which blocks of code (of a parallel program) may run in parallel. The analysis can be very useful in detecting data races, unneeded synchronization, scope of parallelization and many other concurrency issues and improvements.

The analysis is a *"may"* analysis, i.e. if MHP map contains two nodes $N_1$ and $N_2$ which may run in parallel, then they may or may not run in parallel. But every pair of nodes not present in the MHP analysis will *never* run in parallel. This means it is not completely precise and quite liberal about what it includes in the MHP map.

There are a few ways to improve the preciseness of this analysis. One of them is to make use of context sensitive information, leading to a more precise analysis, which is the focus of this report. Usually, context insensitive MHP analysis uses joins the formals and the return value of a method (as explained below), which leads to impreciseness. However, using context sensitive information, we can use the actual parameters and the return value of a method, which leads to a more precise analysis.

In this report, we come up with an algorithm which tries to use the context sensitive information during MHP analysis, and compare their results. As expected, is found that the context sensitive algorithm is more precise than the context insensitive algorithm, but there are other tradeoffs in time and space complexity.

## Previous Work

### MHP analysis

In 1999, Naumovich et al. published a paper[1] on an algorithm for finding the MHP analysis of a Java program containing `synchronized` and thread `start()`, `wait()`, `notify()`, `notifyAll()` and `join()` constructs. In this paper, the authors came up with an algorithm which uses the *program execution graph* (*PEG*) of a Java program to find the MHP analysis by a worklist based algorithm. This algorithm takes care of aliases by using an already provided alias analysis. This alias analysis can significantly affect the results of the algorithm, as we'll see later.

### Improvements

There are mainly two papers which work on making the MHP analysis algorithm incremental and therefore allowing it to run on-the-fly in a quick manner. This can be used to analyze code changes in real-time and for analyzing code changes (in form of contributions, patches, etc.) quickly as well.

The first paper[2] introduces a way to run the MHP analysis in an incremental manner. However, it is focused on X10 like languages, which is why it includes constructs like `async`, `finish` and `atomic`. It also comes up with a more efficient way to figure out if two statements may run in parallel, and

if they do, then under what condition will they run in parallel.

The second paper[3] focuses on improving the MHP analysis by making it feasible to run it on-the-fly for real-time code analysis in IDEs. This again requires and incremental algorithm which would only compute the required changes in the MHP analysis for any change in the code.

### Implementation

There is also another paper[4] which talks about efficient ways to implement the MHP analysis algorithm for Java programs using the *Soot framework*. It comes up with variety of techniques and optimizations for PEG construction, PEG simplification, node identification and classification, callgraph construction, and much more.

## Alias Analysis

### Significance

The MHP analysis algorithm uses the callgraph and the alias analysis to find out the various nodes. Callgraph generation also utilizes alias analysis. Therefore, alias analysis impacts MHP analysis significantly. It can be seen by how adding context sensitivity to the alias analysis would affect the MHP analysis.

### Context sensitivity

When we say context sensitive alias analysis, we mean that the aliases at every node (in CFG) are maintained as a different set, and the inter-procedural analysis for all the method calls are evaluated in a context sensitive manner. This implies that the function's summary won't be unique, but it would depend on the call site, and therefore it would be context sensitive. This would require us to store a separate summary for all the call sites.

## Example 1

### Code

Consider the following code sample:

```
public class A {
    public static void main(String[] args) {
        try {
            Printer x = new Printer();
            Printer y = new Printer();
            Printer z = x.iden(y);
            z = x.iden(y);

            y.start();
            z.start();
            y.join();
            z.join();
        } catch (InterruptedException e) {

        }
    }
}

class Printer extends Thread {
    public Printer iden(Printer t) {
        return t;
    }

    public void run() {
        synchronized (this) {
            System.out.println("Running this thread!");
        }
    }
}
```

### Context insensitive analysis

If we run context insensitive alias analysis algorithm on the above code example, we end up with the following alias information.

$$\texttt{z} \rightarrow \{\texttt{x, y}\}$$

Now applying MHP analysis, we can see that both `run` functions in both the threads `y` and `z` may happen in parallel (this is because the MHP algorithm in Naumovich's paper doesn't add any nodes to $KILL$(`synchronized (a)`) if the variable `a` points to more than one objects). Keeping this analysis' output in mind, let us now use context sensitive alias analysis and compute the resulting MHP analysis.

### Context sensitive analysis

On running context sensitive alias analysis, we get the following alias information.

$$\texttt{z} \rightarrow \{\texttt{y}\}$$

And if we apply the MHP analysis now, using the above alias information, we get that both `run` functions in both the threads `y` and `z` will not happen in parallel (here, ,since $z$ only points to one object, the monitor nodes are added correctly to the $KILL$ set). This is correct and is more precise than the MHP information obtained above using context insensitive MHP analysis.

### Conclusion

Therefore, changing the type of alias analysis does affect the MHP analysis as well. And to get context sensitive MHP analysis, we need to use context sensitive alias analysis.

## Example 2

### Code

Consider the following code sample:

```
1  public class B {
2      public static void main(String[] args) {
3          Data x = new Data();
4          Data y = new Data();
5
6          LateUpdate lu = new LateUpdate();
7          lu.d = x;
8
9          try {
10             lu.start();
11
12             synchronized (x) {
13                 x.val = y;
14                 x.notify();
15             }
16
17             lu.join();
18         } catch (InterruptedException e) {
19
20         }
21     }
22 }
23
24 class LateUpdate extends Thread {
25     Data d;
26
27     public void run() {
28         try {
29             synchronized (d) {
30                 d.wait();
31             }
32         } catch (InterruptedException e) {
33
```

```
34          }
35
36          d.val = new Data();        // d.val → O32
37      }
38  }
39
40  class Data {
41      Data val;
42  }
```

### Alias analysis without MHP information

If we don't have any MHP information, then the alias analysis would be context insensitive, and we would get the following alias information for `x.val`.

$$\texttt{x.val} \rightarrow \{\texttt{y, } \texttt{O}_{32}\}$$

This is because we don't know the flow order, and hence the analysis is flow insensitive in the sense that it considers that all the parallel threads execute in an unordered manner.

### Alias analysis with MHP information

But if we have the MHP information, it establishes an order of execution between statements of different threads. This would help in improving the alias analysis by making it more precise due to the extra "flow sensitivity" introduced by the MHP information. Therefore, the alias information for `x.val` would be as follows:

$$\texttt{x.val} \rightarrow \{\texttt{O}_{32}\}$$

As we can see, this is much more precise.

### Conclusion

Using MHP information during alias analysis will make the alias analysis more precise. And then when we use this more precise alias analysis for the MHP analysis, our MHP analysis will also be more precise. This is a cycle of a sort where each of the two analysis refine the results of the other analysis.

## Fixed Point

#### Definition

We define a fixed point as a set of alias analysis information $A$ and MHP information $M$, such that

$$A = A \cup \text{AA}(M)$$
$$M = M \cup \text{MHP}(A)$$

Here AA and MHP are the functions which convert the MHP information to alias analysis information and vice versa respectively. As we can see, the solutions to the above two equations are the fixed pointes for both the alias analysis and the MHP analysis.

#### Hypothesis

We hypothesize that the fixed point of the alias analysis and the MHP analysis would be the most precise and sound solution for both the analysis. This is because the alias analysis would be using the most precise MHP information, and the MHP analysis would be using the most precise alias analysis information. Any further refinement in the alias analysis or the MHP analysis would not be possible. Therefore, the final fixed point should be the context sensitive MHP information we are looking for, given the alias analysis step is context sensitive.

### Existence

Even though we have an algorithm to reach a fixed point, we haven't shown yet that a fixed point does exist. We can intuitively understand the existence of a fixed point by observing that AA and MHP are monotonic functions. More parallelism (i.e. less precise MHP information) will give a bigger alias analysis output (more aliases). And a bigger alias analysis output will give a bigger MHP analysis output (more parallelism, due to empty $KILL$ sets in cases of variable pointing to more than one objects). Therefore, we can say that at every iteration of the AA-MHP algorithm, we end up with a more precise alias analysis and a more precise MHP analysis. This means that the AA-MHP algorithm will converge to a fixed point, which will be the most precise alias analysis and MHP analysis.

### Termination

We don't prove termination here. This is because for all practical purposes, running the AA-MHP loop for a reasonable number of iterations should be enough to get a fairly precise context sensitive MHP analysis. It is computationally expensive to run the algorithm until a fixed point, because even though the algorithm may terminate, but every pass of the algorithm is computationally expensive.

## Algorithm

### Alias algorithm

Use the classical MHP analysis algorithm but with the PEG (program execution graph) constructed using the MHP analysis. Do not use the CFG (control flow graph) as usually used. THis way, the new alias analysis algorithm will take advantage of the MHP information as well.

### MHP algorithm

The alias analysis algorithm is the same as the one in Naumovich's paper, except that we use context sensitive alias analysis instead of context insensitive alias analysis.

## Limitations

### Computational Cost

The algorithm, as defined above, is fairly computationally expensive. Using the incremental MHP algorithms (or their variants, for varying alias information) defined in papers mentioned above may be very useful in making the algorithm more efficient. But for a naive implementation, it will be magnitudes slower.

The space complexity stays fairly similar and only increases by a constant factor. This is because only the current and past alias information and MHP information needs to be stored. There is no sort of memory hogging which happens for every iteration, and the memory usage stays almost constant after the first iteration.

### Preciseness

Even though it may seem that the fixed point output of this algorithm would be the most precise MHP analysis we can get through context sensitive alias information, but it is incorrect. There are some cases and examples where despite reaching a fixed point, the MHP information is imprecise and can be refined through other techniques.

## Possible Future Work

### Formal Algorithm Definition and Analysis

Define the complete algorithm formally and analyze the various properties of the algorithm (like the time complexity, space complexity, parallelizability, etc.). This would help in understanding the algorithm better and also in proving the correctness of the algorithm.

### Reference Implementation and Testing

This would involve implementing the algorithm in a correct manner and, if possible, adding it to a framework like Soot. This implementation would serve as a reference for any other implementations. This would also involve extensive testing which would help in improving the algorithm with specific optimizations.

### Incremental Analysis

This requires making both the alias analysis and MHP analysis incremental such that the AA-MHP loop can be run faster. Coming up with a good solution for this problem would help significantly in making the algorithm more efficient.

### Special Cases

We should also modify Naumovich's MHP algorithm and the classical alias analysis algorithm to also give precise output for the examples which currently aren't handled precisely by the simple AA-MHP loop algorithm. This may also require adding more steps to the AA-MHP loop, but it would lead to more precise results for such examples.

## References


[1] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. *SIGSOFT Softw. Eng. Notes*, 24(6):338–354, oct 1999.

[2] Aravind Sankar, Soham Chakraborty, and V. Krishna Nandivada. Improved mhp analysis. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 207–217, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Sonali Saha and V. Krishna Nandivada. On the fly mhp analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 173–186, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Lin Li and Clark Verbrugge. A practical mhp information analysis for concurrent java programs. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *Languages and Compilers for High Performance Computing*, pages 194–208, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.