Compiler Engineering
CS6383 - Sep 2019
Mini Assignment 5:

May Happen in Parallel (MHP)

INSTRUCTOR: Dr. Ramakrishna Upadrasta

Report By:

Vijay Tadikamalla	- CS17BTECH11040
Sai Harsha Kottapalli	- CS17BTECH11036
Sagar Jain	- CS17BTECH11034
Shraiysh Vaishay	- CS17BTECH11050
Jatin Sharma	- CS17BTECH11020

भारतीय प्रौद्योगिकी संस्थान हैदराबाद

Indian Institute of Technology Hyderabad

What is May-Happen-in-Parallel(MHP) analysis?

May-Happen-in-Parallel, like the name suggests, has to do with the question, can two statements be run in parallel. This analysis determines if instances of given statements can be executed parallelly. Using MHP we can:

a) Given a statement, determine all the other statements which may be run in parallel with it.
b) Given two statements, what are the conditions for the execution instances of the two statements to run parallelly without causing any issues (say, race condition).

The information from this analysis is used ubiquitously in other analyses as well as optimizations.

How can MHP be useful in program analysis in finding bugs?

The following are some important ways in which MHP results can be used to find bugs:

a) **Narrowing down data race candidates**: If we know that there is data race condition being reached during the execution of some code, using MHP we can be assured that the pairs which MHP output as can-happen-in-parallel are not leading to the data race and we can just check the rest of the instructions.

b) **Static analysis of parallel code**: If certain sections of given source code are known to be executed in parallel (for example, using pragmas), we can use MHP to check if all these instructions are safe to run in parallel, if not we can issue a warning or error about it.

c) **Improving the Efficiency of Traditional Debuggers**: Given a block of code, MHP partition the block into portions of code which can be run in parallel without any issues. Such portions can be assumed to be independent of each other since they do not influence the execution of other portions in any way. Now, if there happens to be a bug or error(static or run-time) arising from this block of code, the debugger can view these sets as unique sets of statements and find a bug separately in each set. This way the debugger to check for a lesser number of relations (usually O(n^2)) which improves both its time and space complexity. Another perk would be that the debugger can itself also be parallelized.

Does MHP on classic C/C++ (C99 and C++03) makes sense?

In C99 or C++03, there was no feature to support parallelism.
Posix threads (pthreads) were introduced later.
Even now, unstructured languages such as C/C++ are not able to harvest the full power of MHP analysis. This is because unstructured and low-level fork-join constructs allow programmers to express rich and complicated patterns of parallelism.
For example, a thread may outlive its spawning thread or can be joined partially along one program path (a partial join) or indirectly in one of its child threads (a nested join). Thus, a sophisticated interprocedural analysis is needed to capture the MHP relations in C programs.

What language extensions enable parallel programming in C/C++?

- **Parallel Algorithm Scheduling Library (PASL)**
  This library uses C++ to let us run a parallel program on a machine's multicores with support for understanding the behavior of those programs(speed plots, processor utilization, etc).
  Example: native **fork()** is expressed as **fork2()** where, this function takes in two arguments, each being a compound statement.
  The end of the **fork2()** function signifies the join point.
- **OpenMP**
  This library is designed for parallel programming in symmetric multiprocessors or shared memory processors. It primarily requires **pragmas** for specifying the code which is to be parallelized. The pragmas serve in thread creations through fork() calls. It also allows the user to control the scheduling of threads(static, dynamic and guided).
  Example:
  ```
  #pragma omp parallel
  {
    ...
  }
  ```
- **Boost**
  It allows the use of multiple threads with shared data in C++ with tools that help in synchronizing data.
- **POSIX Threads**
  It allows the use of threads with mutexes, conditional variables, and synchronization.
- **C++ STL Thread**
- **Dlib**
- **Parallel Patterns Library**
- **POCO C++ Library**
- **TBB**
- **IPP**

Note: More programming languages based on C/C++ have been made with supports parallel programming such as Cilk, Cilk++, Cilk Plus, Milt, C=.

How we can run MHP on parallel C/C++ programs to find bugs?

Languages such as C/C++ provide unstructured and very low=level POSIX threads, also called Pthreads. These allow programmers to express much richer parallelism patterns thereby also complicating the program from an analytical point of view. For example, a thread may be created by one thread but be joined by another. It may even be partially joined along some program paths or indirectly in one of its child threads.
There is also a possibility of threads sending synchronization signals to arbitrary threads. This increases the flexibility manifold but exponentially complicates MHP analysis.

One of the ideas to tackle this is the concept of ***Vector clocks***. We have vector clocks for all instructions in the program. They capture the *happens-before* relationship between events in concurrently running threads. They store the current timestamp for events for each running thread. But these are runtime. For static analysis, we can extend these to something called ***static vector clocks*** which act as an approximation of runtime vector clocks. Data-flow algorithms can be built to compute these static vector clocks which will eventually encode the MHP relationship between any two events.

Another idea is to use the llvm frontend clang and reduce the program to llvm IR. dependence analysis algorithms can be used on this to extract the MHP relationship between events/instructions.

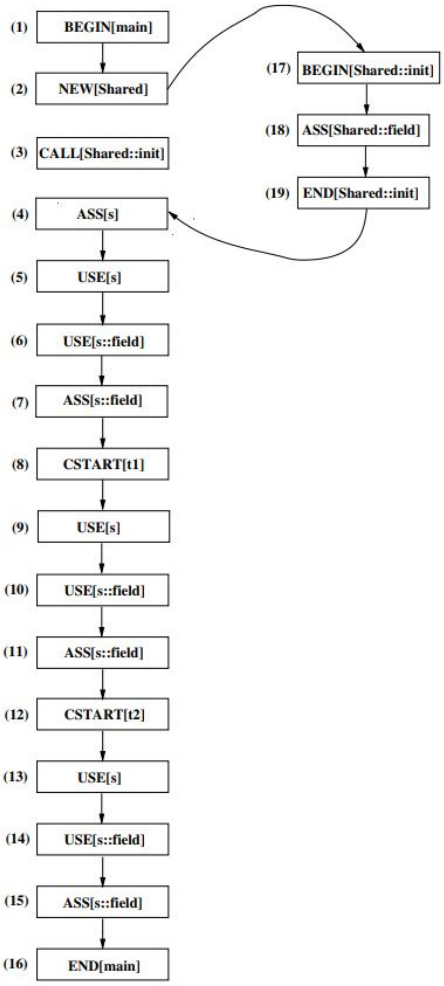
What data structures/program representations are required to support such an analysis in high-level language?

We need certain data structures to be able to efficiently run MHP analysis on high-level languages. There are mainly the following data structures that are required for this:

1. Intra-thread control flow graph

   The control-flow structure of an abstract thread is represented in an intra-thread control flow graph also called ICFG, in short. It contains several elements for each abstract thread t_i :
   $ICFG(t_i) = < V(t_i) , E(t_i) >$
   $V(t_i)$ consists of :

   - USE - set of shared read access
   - ASS - set of shared write access
   - NEW - set of allocation nodes
   - BEGIN - set of method entry nodes
   - END - set of method exit nodes
   - ENTRY - the unique thread entry node
   - EXIT - the unique thread exit node
   - CSTART - set of abstract thread start nodes
   - CJOIN - set of abstract thread join nodes
   - CALL - set of method call nodes
   - ACQUIRE - set of monitor enter nodes
   - RELEASE - set of monitor exit nodes

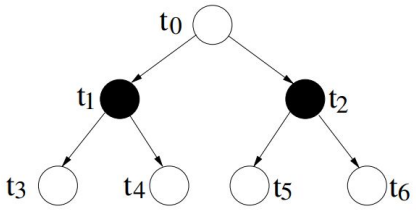


   $E(t_i)$ contains intra-procedural and inter procedural control mow edges in t_i.

2. Must-join

To keep track of the situation where some threads create subsidiary threads and join them later, which is actually a pretty common pattern in parallel programs, we have another data structure called Must-join. Basically, a thread t_j is termed as a must-join abstract thread if $t_i = t_j$ and CJOIN(t_i,t_j) postdom CSTART(t_i,t_j).

3. Thread Creation Tree (TCT)



Threads can be structured according to their start-relationships. The thread creation tree or the TCT encodes this information. Abstract threads are represented as nodes, edges encode the start relation. The must-join information for each node in the TCT is also encoded.

The main thread constitutes the root and threads started by the main thread are found at the first hierarchy level.

MHP in LLVM

- May-happen-in-parallel can be implemented in LLVM using ***dependence analysis***. Once we have a dependence analysis for the program, we can segregate the program into small chunks that can be run in parallel. They will not affect the execution of other pieces.
- For sequential instructions (not jump statements), simple dependence analysis can help us determine which instructions can run in parallel. For loops, each execution instance can be analyzed using ***polyhedral dependence analysis*** for these instructions.
- Polyhedral dependence analysis gives us more insight into individual iterations of a loop. Hence it prevents false negatives for may-happen-in-parallel in case of loops.
- For LLVM, it makes sense to analyze MHP on the instruction level as that gives us the most accurate analysis. The time complexity will be $O(N^2)$. (***Instruction level granularity***)

References

- http://www.cse.iitm.ac.in/~krishna/courses/2015/even-cs6013/BarikLCPC05.pdf
- http://www.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html
- https://www.cs.cmu.edu/~15210/pasl.html
- https://en.wikipedia.org/wiki/List_of_C%2B%2B_multi-threading_libraries
- https://dl.acm.org/citation.cfm?id=3168813
- https://ieeexplore.ieee.org/document/7349644
- http://lists.llvm.org/pipermail/llvm-dev/2018-October/126971.html