**Slide 1**

My PhD research focus on low overhead data race detection for large structured parallel applications, such as big parallel program used in the High Performance Computing world.

In this proposal I will present what this research is about, the results we obtained so far and what we intend to do in the future to reach our goals.

**Slide 2**

In my thesis statement I claim that combining that:

* we need to combine the best of static and dynamic analysis techniques for data race detection
* Exploit the concurrency structure of structured parallel language, such as OpenMP

Only in this way we can guarantee precise data race detection for large HPC application while maintaining it practical which means building tools with a low runtime and memory overhead.

What are the motivation of this and why do we need better data race checking techniques for HPC application?

**Slide 3**

Scientists need to model world phenomena in order to understand and control them.

These kind of phenomena, such as climate modeling, hydrodynamics modeling, laser simulation and many others are really complex and scientists need high computational power to model them and make their experiment to understand them.

**Slide 4**

The computational power come from the supercomputers that HPC centers all over the world are building to obtain capacity of the order of PFLOPS.

Here we have two examples of two of the most powerful computer in the world:

* Tianhe-2 with a computational power of 33 PFLOPS
* Sequoia with a computational power of 17 PFLOPS

These supercomputer consists of thousand of processors usually connected trough a super fast network to setup a cluster that can reach these peaks of computational power.

However they also demand an high amount of energy, on the order of MWatts.

So it’s very important to gain higher performance at a given energy budget, and this is done using multicore CPUs and increasing the on-node parallelism in large software applications.

**Slide 5**

Multithreaded programming is achieved through many programing models, the most popular is probably Pthreads.

But in HPC the predominant paradigm of choice is OpenMP, which guarantees portability and ease of use.

A clear example is our collaboration with the Lawrence Livermore National Laboratory where one of the main tasks if the porting of critical multiphysics applications to OpenMP.

This is not a simple task, indeed explosion of on-node parallelism also introduce new sources of non-determinism.

One of the hardest to find are definitely data races!

So, this porting needs to be backed up by capable and scalable debugging tools that help scientists in this process.

**Slide 6**

A recent example is given by the porting of a very large application called HYDRA which has over a million of lines of code.

HYDRA uses MPI and OpenMP thread-level parallelism.

During the porting of this application to the Supercomputer Sequoia, a non-deterministic crash was happening on one of the library used by HYDRA, called Hypre which indeed uses OpenMP thread-level parallelism.

This crash was happening above certain scale and compiler optimization levels.

The scientists suspected a data race but even after months of debugging (and so many working hours) they did not figure out the problem and disabled the threading in the Hypre library, losing also in performance in addition to time.

**Slide 7**

Data races are very challenging to identify, especially in large OpenMP applications with millions of lines of code!

Data races is probably one of the most widely studied problems in concurrent programs design and debugging.

It has been show that it is a NP-hard problem, indeed still today there are not tools that can identify this problems with high precision and accuracy and a low amount of computational resources.

This table summarizes the most popular data races detection tools and their feature.

Only one of them applies static techniques Intel SSA and as we can see it is not very accurate and precise on OpenMP programs and does not support Pthreads at all, static techniques are known to have a low overhead so that’s the only positive point about this tool, since it is also not portable beside Intel architectures.

The other tools, Intel Inspector, Helgrind and ThreadSanitizer all apply runtime analysis.

They work well on pthreads program, but only Inspector barely supports race detection on OpenMP programs, and all of them have a very high runtime and memory overhead, only Tsan does better than the others it also is the only one portable on other architectures (I personally did the porting on PPC64).

So we definitely need more precise, accurate, and scalable data race detection tools.

**Slide 8**

As we saw from the tools table, we can categorize data race detection techniques in static and dynamic.

**Slide 9**

Static analysis techniques allow to reason about all the inputs and all the threads interleavings/scheduling.

They are in general very imprecise, in fact they easily miss races and report false alarms, which means races that are not actually races.

However this techniques are very scalable, and of course don’t have any runtime overhead.

An example of tool that uses static analysis for data race detection is Intel Static Security Analysis which support OpenMP parallelism on C/C++ and Fortran programs.

**Slide 10**

On the other hand dynamic analysis techniques are very precise and do not report false positives, however differently from static analysis, can not reason about all inputs and interleaving, but can report races only about the execution and branch of the program that they are analyzing.

Dynamic techniques have very high runtime and memory overhead because during the analysis they need to keep several information about the threads and the memory accesses that may take a lot of additional computational time and memory.

There exists different dynamic techniques, some of them are lockset algorithms, happens-before relation and hybrid techniques that combine both lockset and happens before.

These techniques are in general the most used in data race detection tools, and they were basically developed for fork-join thread-parallelism.

However, they don’t fit very well for the OpenMP fork-join paradigm, in the sense that they need more information about synchronization mechanism used by OpenMP for the threads, furthermore in a structured parallelism paradigm as OpenMP, techniques such as happens-before relation based on vector clocks do not perform very well in terms of runtime and memory overhead.

**Slide 11**

Indeed not many OpenMP data race detectors are out there!

Commercial tools such as Inspector and Sun Studio Data-Race Detection tool both support OpenMP, but experiments show very high slowdown of the program execution during the race detection process.

Other tools such as Helgrind and TSan work well on PThreads programs, do not support OpenMP and also their overhead is very high.

Tsan is the best tool and it introduces a slowdown of 5x-20x and increase the memory consumption between 2x-10x.

But in general all these tools when applied to large HPC applications can introduce an 100x slowdown and increase the memory consumption of 50x!

**Slide 12**

The motivations seen so far and the state of the actual situation of data race detector for OpenMP applications give us a strong reason to try to design and implement a better data race detection technique for OpenMP programs.

We believe that combining static and dynamic analysis we can reduce the amount of code to be analyzed at runtime reducing the overall runtime and memory overhead.

In particular, with static analysis can help find race free regions of code that can be excluded from the dynamic check.

Since our target are OpenMP programs, we can make some importation observation about the OpenMP programming paradigm.

For example, its fork-join approach is simpler and more structure, one first observation that we can do is that in OpenMP a parallel region (so a team of threads) join in the same point to a unique threads.

With these simple observation we think that we develop new runtime technique that do not use heavily happens-before relation and we can still maintain higher precision and accuracy reducing the overhead for instance avoiding the use of large vector clocks.

**Slide 13**

Our main contributions can be summarized in the following 5 points:

* Sequential Blacklisting: We design a static analysis to identify the code executed sequentially in a OpenMP program and exclude it from the runtime analysis
* Data Dependency Analysis: We will design a static analysis technique that identify race free code in a OpenMP program to exclude it as well from the runtime analysis.
* Archer v1: We will combine the static analysis with the existing Clang/LLVM ThreadSanitizer data race detection tool and we will make Tsan capable of detect data races in OpenMP programs, as part of a first version of a data race detector called Archer.
* Clock-less runtime algorithm: We will design and implement a new runtime analysis technique that exploits the structured parallelism of the OpenMP paradigm in order to reduce the runtime and memory overheads.
* Archer v2: We will embed the aforementioned new runtime technique in a second version of Archer as a replacement of the TSan runtime and combine it with the static analyses.

We’ll see now in detail the single contributions and how we intend to integrate them together.

**Slide 14**

Our sequential blacklisting idea comes from the observation that in OpenMP, differently from Pthreads, we can clearly distinguish the code that will be executed sequentially from the code that will be executed in parallel.

In a Pthread program, as we can see in the figure, the main thread keeps doing its own work while it spawns new threads.

Often, the threads join just before the end of the program, so from a static analysis point of view would be really hard to identify the code executed only by one thread as sequential code.

**Slide 15**

In OpenMP instead, the structured fork-join parallelism show clearly what is executed by the only main thread.

Indeed, every instruction outside a OpenMP parallel construct is executed only by the master thread.

And in general in OpenMP programs parallel regions are alternated by sequential code that can be excluded by the runtime analysis reducing the runtime and memory overhead.

So in the figure in the point 1, 2 and 3 the main threads may execute code sequentially such initialization, etc.

This is our first contribution to reduce the amount of code to analyze at runtime and reduce the overhead.

**Slide 16**

The second contributions, it’s that we exploit an existing static analysis technique to identify race free regions of code.

OpenMP uses heavily for-loops and large arrays that are parallelized dividing them in chunks and assigning the chunks of the array to different threads.

In general the accesses to the array does not need any synchronization mechanism since the array is divided in a way that the threads will access different location of the vector.

This is not always the case, especially when we introduce data dependencies or indirection in the array accesses.

The example in green, shows a correct parallelization of the for-loop, since in this case each threads will access different chunks of the array a.

So we say that the loop does not have any loop-carried data dependency and so its parallelization is race free, so we can blacklist this kind of code and exclude it from the runtime analysis.

The second example instead shows a for-loop with a data dependency in the array access.

What will happen is that the array will be divided in chunks in the same way as the previous example, but because of the data dependency multiple threads may access the same location simultaneously and since no synchronization mechanism is used it may result in a data race.

In this case each thread access the last element of the previous chunk.

In general when there is a data dependency we don’t know if the race will manifest, so it is better to check this situations at runtime.

A code like this won’t be blacklisted but instrumented for further checks at runtime.

**Slide 17**

We combine these two static analyses with the ThreadSanitizer runtime to create an OpenMP data race detector called Archer.

The figure shows the entire workflow of Archer.

It is based on the compiler infrastructure LLVM/Clang.

We implement the static analyses as LLVM Passes, we have a custom pass for the sequential blacklisting and an existing pass called Polly for data dependency analysis that we modify to analyze OpenMP code.

We compile a OpenMP program with Clang and the flags to run also our custom passes and the Tsan passes.

Our custom passes analyze the code and generate information about the line of code that we want to blacklist from the runtime analysis because they are race free.

After that the Tsan instrumentation pass has to analyze the code to insert the runtime calls for the runtime analysis.

We modify the Tsan instrumentation pass to exploit the information generated by our passes and instrument only the potentially racy code.

Once the code is instrumented and the compiler generates the executable we can just run the program normally and the Tsan runtime will perform the race check.

**Slide 18**

As I said at the beginning, Tsan is a data race checker for Pthreads programs, however since an OpenMP program is translated by the compiler using Pthreads directives it can perform some of the analysis.

We made some experiments and Tsan on OpenMP programs generates many false alarms, misses races and most of the time it crashes.

This is because of some of the synchronization mechanism used by the OpenMP runtime are unknown by Tsan and it does not have enough information to build a correct happens-before relation between the threads.

Our solution consists in annotate the OpenMP runtime in order to communicate the Tsan runtime the correct happens-before information in presence of unknown synchronization primitives, such as openmp barriers, openmp critical sections, etc.

**Slide 19**

Here we have a full example of an OpenMP program and the different situations for the static analyses.

The green rectangle show the code that will be blacklisted, so it won’t be instrumented by the Archer instrumentation process.

Line 3 is a function executed sequentially so excluded by the runtime analysis.

The OpenMP for-loop is race free and then blacklisted.

Then again another sequential function at the end.

The red rectangles are instead code executed in parallel, in particular the OpenMP for has a data dependency so we check it at runtime.

**Slide 20**

Here another example and the Archer report after the runtime analysis.

Again also in this case the first for loop is blacklisted and excluded by the runtime analysis.

While the second parallel region is instrumented and on the right we have the Archer report that precisely shows that the thread T2 performed a read on a specific location and previously there was a write in the same location by another thread with no synchronization and the race is reported showing also the line of code in the program where the read and write are performed.

**Slide 21**

This first part of the contribution is already implemented and we actually published already a workshop paper at the LLVM-HPC workshop at SC’14 in New Orleans and a conference paper at IPDPS’16 that where we will present Archer at the end of May.

At the best of our knowledge Archer is the very first open source and usable OpenMP data race checker.

Our first experiments on some OpenMP benchmark and HPC benchmark show about a 25% of runtime overhead reduction on average respect other tools such as inspector, etc.

And most importantly it’s being used by the HPC community and our goal is to make it an important tool for HPC debugging.

Here I have presented the work that we already accomplished during my research.

This results are nice and a good start, but we need to reduce much more the runtime and memory overhead of at least an order of magnitude, because with the increase of on-node parallelism the existing techniques will generate too much overhead making the race detection unfeasible.

So now I’ll talk about the next steps we would like to take to reach these goals.

**Slide 22**

Our next contribution is to design a race checking algorithm ad-hoc for OpenMP applications.

I say ad-hoc because OpenMP fork-join model is much more simple and structured than other fork-join paradigms such as Pthreads.

We believe that the OpenMP concurrency structure may help to find race in a lighter and simpler way than rely on the classical vector clock solution.

That is way also we call our algorithm clock-less, because we want to design an algorithm that does not rely completely on vector clocks.

If we look at the image in this slide, the graph show the concurrency structure of a simple OpenMP program.

In this case we have the thread 0 that start the first parallel region and forks two new threads.

Each of the new threads create another parallel region each and so on.

So we have nested parallelism and at the end all the threads they join again to one single thread.

From this graph we can make many observation about the OpenMP concurrency model from the race checking point of view.

For example, we can see that threads that belong to the same parallel region could race (thread 3 and 4).

Also threads that belongs to different but concurrent parallel regions they may race as well, and this is the case for examples of threads 3 and 5, or 8 and 5.

However we can notice that in case of consequent parallel regions, such as the two regions forked by thread 1 and 7 there can not be any races, since one happens before the others, so the threads that belong to these regions can not race to each other.

We are exploring different techniques that help to identify if two threads in an OpenMP execution are concurrent or not, so that we can limit the race check only to concurrent threads.

Existing techniques propose labeling methods for the threads that allow to identify quickly this relation between threads, so we plan to go in this direction.

**Slide 23 - 24**

We define the OpenMP formal concurrency model through a state machine and a set of transition rules.

As I show in the proposal, the state machine is well defined with a global state and local state for each thread, to keep track of the evolution of the system in terms of parallel region begin and end, barriers encountered by the threads, critical sections, read and writes accesses, etc.

The state machine allow us to reason correctly about the concurrency structure of an OpenMP program and to consider all possible corner case from a race checking point of view.

So we define different rules that model the fork and join operation, the rule to model the threads that hit a barrier, the rule to model the load and store and store the information about these operation.

Then we also define a “RaceCheck” rule that can fire anytime during the machine execution and check for the existence of data races.

The state machine assume infinite memory and consider every memory accesses so it guarantees to find any data race that manifest in a OpenMP program execution.

Of course this can not be directly implemented in reality, but it allows us to consider all the corner cases in OpenMP data race checking.

We are looking in possible implementation that can approximate correctly the state machine to obtain a low overhead data race detection while guaranteeing precision and accuracy.

We believe that we can keep the runtime concurrency structure of the OpenMP programs in a fast and lock-free data structure, maintain a representative set of memory accesses, such as sampling, and perform race checking only at specific point of the program execution, for example at each barrier checkpoints.

Furthermore, the OpenMP runtime provide Tools API that allow to get information about the OpenMP constructs at runtime, this will be a very important resource for the data race checking algorithm implementation and we plan to base our implementation on this API since it would also guarantee portability across different OpenMP runtime.

This coming summer I will be a student intern at IBM and I will actually work on the development of this API, which will be a very valuable experience for the implementation of our race detector.

So we will continue exploring possible implementation that go in this direction.

**Slide 25**

We first published a feasibility study at the LLVM-HPC WS at Supercomputing 14 in New Orleans.

Then as I said we have an accepted paper at IPDPS’16 in Chicago where I will present the first version of Archer.

We plan to implement our new runtime algorithm and submit it in a conferences like POPL or PPoPP.

Then we will embed the static analysis and the new algorithm for a new version of Archer and we plan to do exhaustive testing especially on HPC benchmark to measure correctly the runtime and memory overhead.

We hope to be able to publish our results at PLDI, IPDPS or PACT.