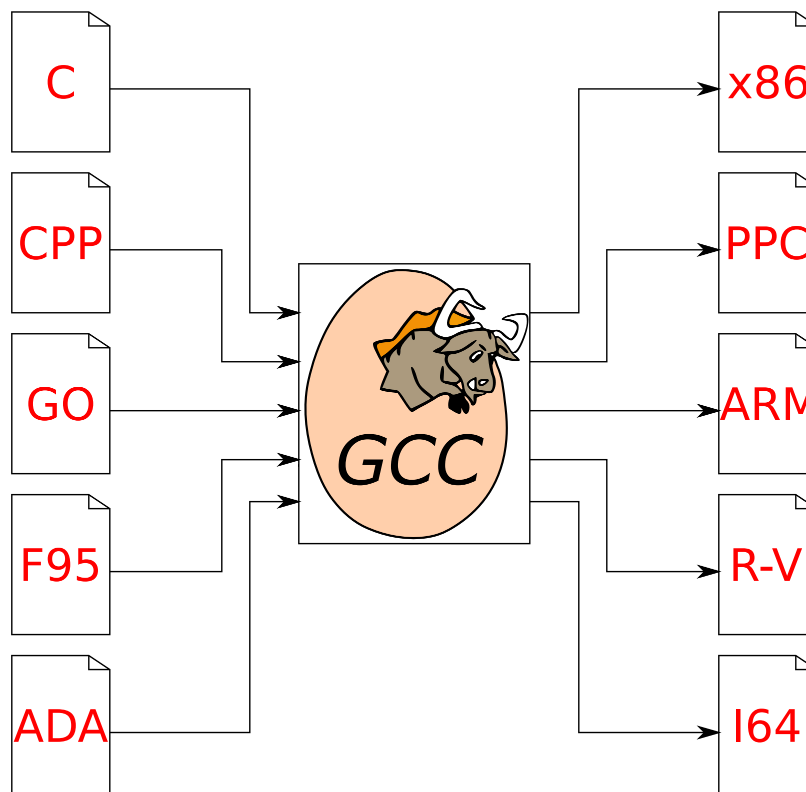
Parallelizing GCC with Threads



Giuliano Belinassi
Timezone: GMT−3:00
University of São Paulo – Brazil
IRC: giulianob in #gcc
Email: `giuliano.belinassi@usp.br`
Github: `https://github.com/giulianobelinassi/`
Date: April 1, 2019

1 About Me

Computer Science Bachelor (University of São Paulo), currently pursuing a Masters Degree in Computer Science at the same institution. I've always been fascinated by topics such as High-Performance Computing and Code Optimization, having worked with a parallel implementation of a Boundary Elements Method software in GPU. I am currently conducting research on compiler parallelization and, therefore, a GSoC internship working with the GNU Compiler Collection (GCC) on a related topic is an excellent opportunity for me to become more involved with the Free Software community.

Skills: Strong knowledge in C, Concurrency, Shared Memory Parallelism, command line utilities (grep, sed...) and other typical programming tools.

1.1 Contributions to GCC

I've submitted some patches, mainly adding inline optimizations to trigonometric functions. This kind of patch requires exaustive testing to guarantee that the optimization does not yield severely incorrect results. This work resulted in a blog post[1] about the patch *Optimize sin(arctan(x))*. I did this blog post both to register how to add this kind of optimization and also to encourage newcomers to contribute to GCC.

Name	Status
Optimize sin(arctan(x))	`Accepted`
Optimize sinh(arctanh(x))	`Accepted`
Fix typo 'exapnded'	`Accepted`
Split 'opt and gen' variable	`Working on`
Update sin(arctan(x)) test	`Waiting Stage1`
Fix PR89437	Wilco Dijkstra version accepted

2 Parallelization Project

While looking for topics in the compiler field that touched subjects that I am interested for my masters thesis, I found a parallelization project proposed in GSoC 2018. With this in mind, I started a discussion in the mailing list to understand what that project is about, which was parallelizing GCC internals to be able to compile big files faster[2]. As can be seen in the discussion, I started to work on this subject way before the list of GSoC accepted organizations was made public.

As stated in PR84402[3], there is a parallelism bottleneck in GCC concerning huge files (with hundreds of thousands of lines of code). In the course of the discussion, Bin Cheng[4] reported that he is affected by this issue, stating that parallelizing the compiler may solve his problem. These discussions demonstrate the community interest in this project.

[1] `https://flusp.ime.usp.br/gcc/2019/03/26/making-gcc-optimize-some-trigonometric-functions/`
[2] `https://gcc.gnu.org/ml/gcc/2018-11/msg00073.html`
[3] `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84402`
[4] `https://gcc.gnu.org/ml/gcc/2018-12/msg00079.html`

2.1 Current Status

In PR84402, Martin Liška posted a graphic showing the existence of a parallelism bottleneck in GCC compilation due to huge files such as `gimple-match.c` in a 128-cores machine that `make -j128` alone could not alleviate. He also posted an amazing patch to GNU Make to collect the data and a script to plot the graphic, which I used to reproduce the same behaviour in a 64-cores machine that is available at my university.

Unfortunately, I found this approach not easy to replicate, as it requires compiling and installing a custom version of Make and generates gigabytes of data which require parsing by a script that often crashes, as it struggles to generate a very large SVG. With this in mind, I created a set of tools[5] using a completely distinct approach, which generates less data, is more stable, and plots better graphics, such as the one in Figure 1.

I also explored the GCC codebase in order to find the performance bottleneck for such huge files. I compiled GCC with the `-disable-checking` and `-O2` flags, which give a more performant (but less reliable) compiler and used this version to compile `gimple-match.c` (the largest file in GCC). In this context, I found that the method `finalize_compilation_unit()` of class `symbol_table` takes around $50s$ of the compilation time, with the `expand_all_functions` routine taking most of it. Therefore, this routine is a strong candidate for parallelization.

Currently, my strategy to parallelize `expand_all_functions()` is to perform the `GIMPLE` processing step in parallel, as suggested by Richard Biener[6]: each function may be independently processed by the many passes of `GIMPLE`. There is also a pipelined alternative: a distinct thread is responsible for a given `GIMPLE` pass and each function is fed sequentially to each of them. This allows many functions to be processed simultaneously, each by a different pass.

Furthermore, I am also studying the theoretical background behind `GIMPLE` and `cgraphs`. I have read the `GIMPLE` documentation[7] and I am looking at how `cgraph` works internally, both in theory and within GCC.

2.2 Planned Tasks

I plan to use the GSoC time to develop the following topics:

- Week [1, 7] – May 6 to June 21:
  Refactor `cgraph_node::expand()` and `expand_all_functions()`, splitting `IPA`, `GIMPLE` and `RTL` passes, and some documentation about the global states of the `GIMPLE` passes.

- Week 8 – June 24 to 28: **First Evaluation**
  Deliver a patch with the refactored version of both functions mentioned above, plus the partial documentation with regard to `GIMPLE`.

[5] https://github.com/giulianobelinassi/gcc-timer-analysis

[6] https://gcc.gnu.org/ml/gcc/2019-03/msg00249.html

[7] https://gcc.gnu.org/onlinedocs/gccint/

- Week [9, 11] – July 1 to 19:
  Continue documentating and refactoring the `GIMPLE` passes, and prototype a parallelization of `expand_all_functions()`.

- Week 12 — July 22 to 26: **Second Evaluation**
  Attend to Debconf 19 (Curitiba, Brazil). Please let me know if someone else from GCC will attend.
  Deliver a working non-optimized parallel version of `expand_all_functions()` with the point of being correct in a multi-threaded environment.

- Week [13, 15] — July 29 to August 16:
  Iteractively improve the current implementation.

- Week 16 - August 19: **Final evaluation**
  Optimize the current implementation, so that there is a speedup over the sequential version when compiling `gimple-match.c` and reducing the total compilation time in `GCC` compilation without bootstrap.

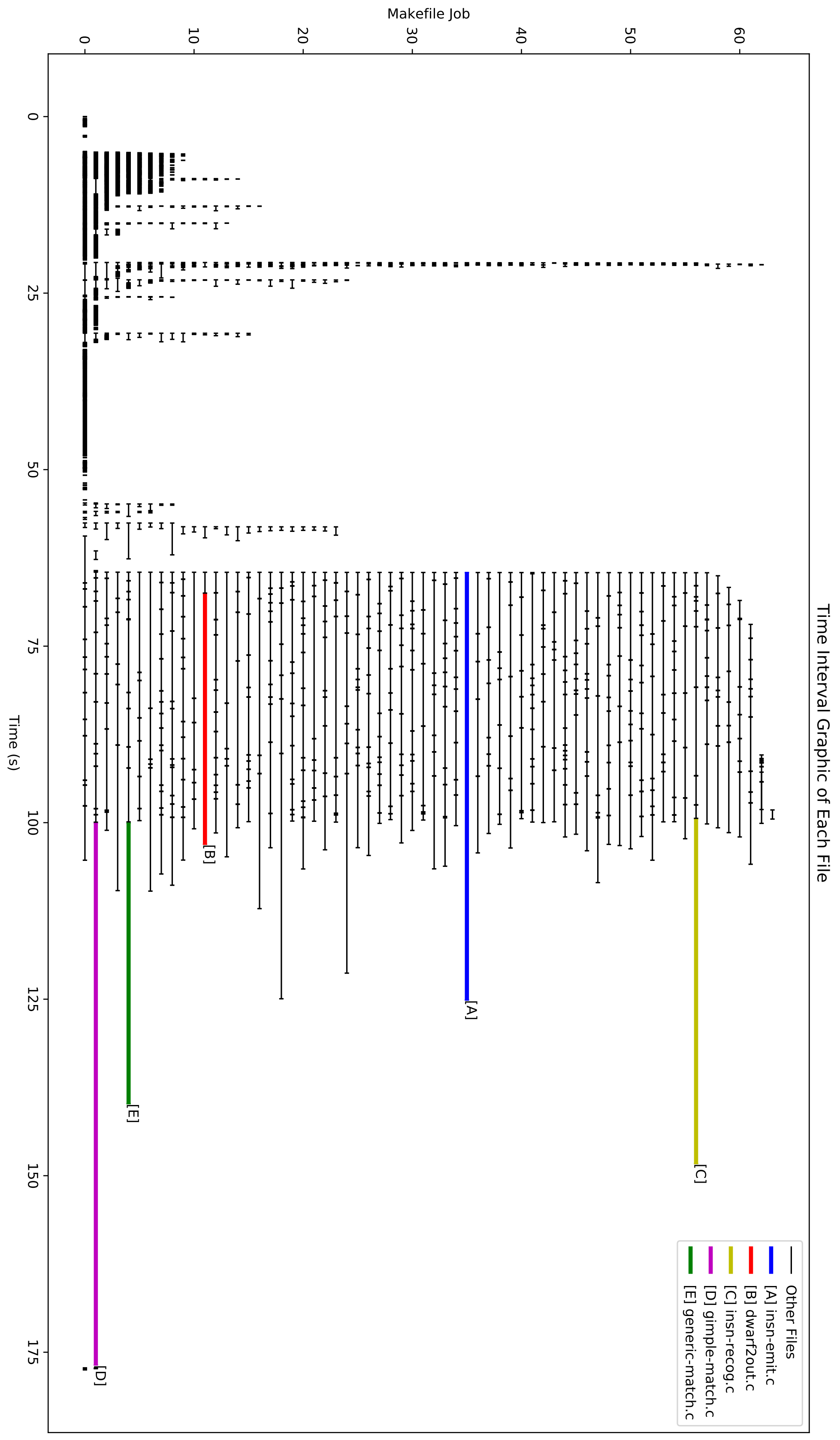


Figure 1: Elapsed time analysis in GCC compilation for a 64 cores machine, No bootstrap