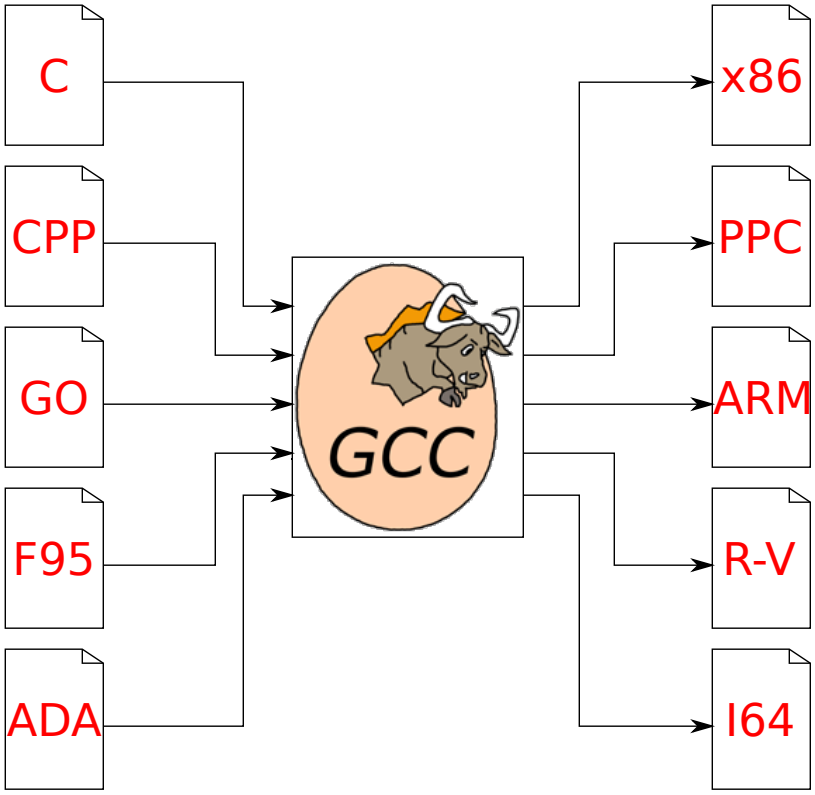
Parallelizing GCC with Threads



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: March 29, 2019

1 About Me

Currently pursuing a Master's Degree in Computer Science at the University of São Paulo with a Computer Science Bachelor's degree in 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, and I am currently conducting research in compiler parallelization. Therefore, I am applying to the GNU Compiler Collections (GCC) both as a research project in parallelization of compilers, and mainly to become a part of the free software community.

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

1.1 Contributions to GCC

I've submitted some patches, mainly adding inline optimizations to trigonometric functions. These kind of patches requires deep testing to guarantee that the optimization did not yield severely incorrect results, which concluded in a blog post about the patch *Optimize* $\sin(\arctan(x))$. I did this blog post both to register how to add these new kinds of optimizations, and also to encourage newcomers to contribute to GCC.

Name	Status
Optimize $\sin(\arctan(x))$	Accepted
Optimize $\sinh(\text{arctanh}(x))$	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 Paralelization Project

While looking for topics in compiler's field that touched subjects that I am interested in for the subject of my master's thesis, I found this parallelization project in GSoC 2018. With this in mind, I started a discussion in the mailing list to understand what this project is about, which was parallelizing the GCC internals to be able to compile big files faster, which got my interest[1]. This was way before the list of GSoC accepted organizations.

As stated in PR84402[2], 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[3] reported that he is facing a similar issue with his project, stating that parallelizing the compiler may solve his problem. These topics supports the interest in the community for this project.

[1] https://gcc.gnu.org/ml/gcc/2018-11/msg00073.html
[2] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84402
[3] 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 parallelism bottleneck in GCC compilation due to huge files such as `gimple-match.c` in a 128-cores machine. He also posted a 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 in my university.

Unfortunately, I found this approach not easy to reproduce, as it requires compiling and installing a custom version of Make, generated gigabytes of data which requires parsing by a script that often crashed due to a very large SVG. With this in mind, I created a set of tools[4] using a complete distinct approach, generating less data, is more stable, and plotted 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. Analyzing the most time-consuming file in GCC's project (`gimple-match.c`), I found that the method `finalize_compilation_unit` of class `symbol_table` takes around $50s$ to compile with a `-disable-checking` GCC under `-O2`, with the `expand_all_functions` routine taking most of this time.

Currently, my strategy to parallelize `expand_all_functions` is to make `GIMPLE` passes per function parallel, as suggested by Richard Biener [5]. However, there is also a pipelined alternative, where the functions are fed into a `GIMPLE` passes pipeline, which then processes the functions.

Furthermore, I am also studying the theoretical background behind `GIMPLE`, and `cgraphs`. I have read the `GIMPLE` documentation[6], and I am looking for how `cgraph` works internally, both in theory and in 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:
  Here I expect to deliver the refactored version of `cgraph_node::expand()` and `expand_all_functions`, splitting `IPA`, `GIMPLE` and `RTL` passes, plus some documentation about the global states of the `GIMPLE` passes.

- Week 8 - June 24 to 28: **First Evaluation**
  Here I expect to deliver the refactored version of `cgraph_node::expand()` and `expand_all_functions`, plus some documentation about the global states of the `GIMPLE` passes.

- Week [9, 11] - July 1 to 19:
  Continue the documentation and refactor of the `GIMPLE` passes.

[4] `https://github.com/giulianobelinassi/gcc-timer-analysis`
[5] https://gcc.gnu.org/ml/gcc/2019-03/msg00249.html
[6] `https://gcc.gnu.org/onlinedocs/gccint/`

- Week 12 - July 22 to 26:
  Attendance to Debconf 19 (Curitiba, Brazil). Please let me know if someone else from GCC will attend.

- Week 12 - July 22 to 26: **Second Evaluation**
  Here I expect to deliver a working non-optimized parallel version of `expand_all_functions`. Here the point is not to perform better than the sequential version, but to still be correct in a multi-threaded environment.

- Week [13, 15] - July 29 to August 19:
  Iteractively improve the current implementation.

- Week 16 - August 19: **Final evaluation**
  Here I expect to have a better-optimized version of the function, with speedup over the sequential version when compiling `gimple-match.c` and reducing the total compilation time in `GCC` compilation without bootstrap.

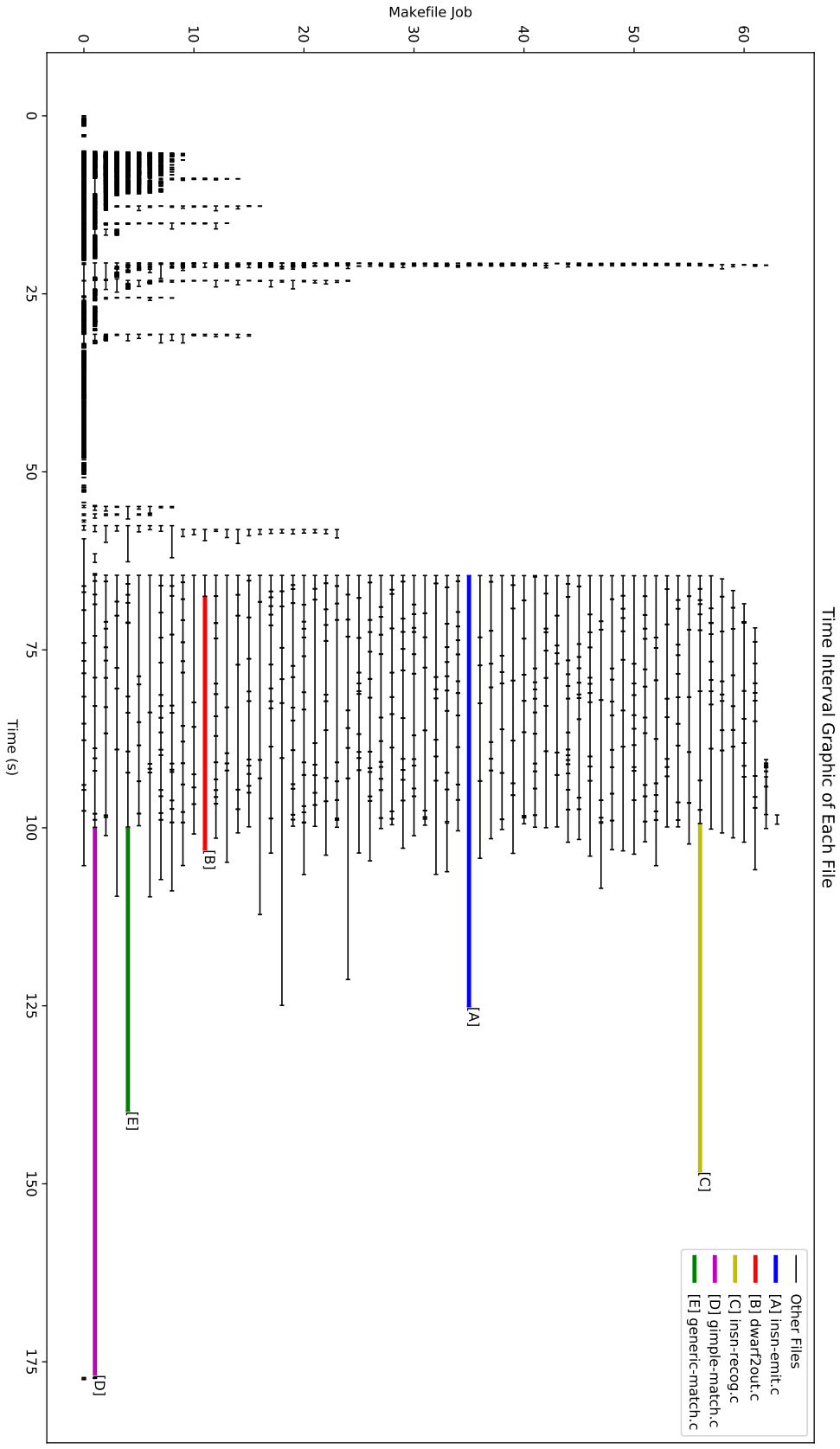


Figure 1: Elapsed time analysis in GCC compilation for a 64 cores machine, No bootstrap