This page is the first in a series of pages on my ideas in how to coordinate making GCC multithreading as to related to improving build speed. In this page we will explore various ideas on how to make GCC scale well and improve build speed on systems with many cores due to make or various other build systems not scaling well internally with its job server implementation of splitting up and running multiple gcc or compiler instances in parallel.
Please note that this page is to discuss theoretical issues of making gcc multithreaded and not the actual implementation details as these are still under investigation due to more heuristics and other data as related to shared state being required. Later pages will discuss the data and actual implementation details at a code level.
Contents

Rewriting core data structures to be multi-threading safe

While the other student has come to the conclusion that core data structures can be made per thread in implementing multi-threading support in gcc. This seems unlikely due to two major problems with this approach that can arise, the first being that for each data structure like struct function or tree node types being working on we would have to alloc per thread making the memory usage on systems with tens of cores use much more ram than before when enabling this feature depending on the number of threads enabled. While this might not be an issue if the machines aren't shared, traditionally machines with large amounts of cores are in shared build farms making this a poor way to implement sharing data on those machines and probably one of the primary interested parties in the project not able to use it.

A second but perhaps not as obvious issue is that per thread variables are limited to a rather small stack size depending on the architecture and the processor of the machine. For example my system is limiting to 8KB or 16KB depending on how certain parameters of the system are configured. Therefore due to the size of some structures or even the number at a time storing this in per thread stack space as implemented by __thread is not viable.

Due to this we will need to make these threads properly multithreaded and thread safe for sharing between threads. While there are many ways to do this it seems that the three mostly ways depending on the type of structure and how it is used:

1. A reader lock or RCU based implementations for reader only or non update heavy structures i.e. structures using for analysis only by passes or gcc
2. reference counting or spinlocks by structures that are about 50/50 in terms of being written to and read from

3. Swapping atomically for small enough structures that can be swapped on the processor in atomic instruction sets

SSA Dominator Tree and Related Issues

GCC like other modern compilers including LLVM transforms a user's code into SSA or Single state assignment form for functions and LCSSA or local conditional single state assignment for loops. While both SSA or LCSSA formats are well known to be good optimizations, the way that we deal with them internally in dominator trees needs to be updated.

There currently are two ways to deal with the issue of dominator trees as the internal graph representation of both SSA and LCSSA in gcc. The first and perhaps the best way is to make the trees lockless or atomic in nature of insert of new dominating nodes of the tree. The other and perhaps last resort would be to find another graph type or data structure if it becomes impossible to make the current choice of dominator trees workable for a multi-threaded gcc. More work will need to be done in order to figure out which of the two options is better in terms of data collecting of how the current dominator trees operate within gcc for preserving both SSA and LCSSA formats.

Making the pass manager and related infrastructure multi-threaded aware

While fixing Intra Procedural Analysis of various passes or partitioning them between threads fixes multithreading at a per pass level there is another issue. What happens if a pass can be run in parallel or on another thread due to later passes not touching the same GIMPLE or original function or other various code regions?

Therefore fixing the pass manager to be multi-threaded aware based on detecting what future passes will require or not be required to wait on from earlier passes may improve scaling GCC out much better. This is due to the reality of the pass manager still only really running on one thread and even if we partion various passses correctly we must still operate and scale at linear or one thread for the pass manager.

Due to this a rewrite of the pass manager will be required in order to detect and launch threads based on sharing data between themselves and future depended passes. For example, if a function is not touched by a GIMPLE pass for inlining or can't be it be launched on another thread and joined up to the next pass that is required to work on it. Currently I've come to the conclusion that there are two ways of implementing the pass manager to support this. The first requires rewriting the current passes themselves in order to do this and leaving the pass manager as a static member in this. While this is probably easier to implement due to passes having internal data, this runs into the issue of what occurs if a new pass is added. Therefore due to this I'm leading toward a second in which a new set of core functions and class(s) are required for the major function passes from IPA to RTL. The one one issue with this approach is that of being able to translate between different parts of the compiler or pass sets. Due to this the easiest way is to support a hierarchy of classes that share the same state if possible with differences being done at the per pass class type level i.e. GIMPLE e.t.c. Or the other way of having a translation function between the pass types but I would lead against this due to a) overhead of thread and other data structure killing or b) Being possibly slow and therefore actually added a bottleneck. If none of these work we can just work up to the end of the set of the set of passes like GIMPLE and then just wait for the standard translation.

Garbage Collector and Memory Allocator Issues

The biggest issue with the current memory allocator is not that it has a global lock but that it needs to be rewritten in a multithreaded environment to not stall what will be the main or set of threads for passes. Due to this we must rewritten it to either have a free list that caches the data and gives it to the threads doing internal pass or other non memory allocation related work. In addition the other way is to have a set of threads do the allocation and give the memory to the main pass threads therefore removing the stalling issue of memory allocation.

Interaction with Exposing or Getting Data from Make and other Build Systems

In order to make this project successful we must interact with other projects most notably other build systems including make, cmake e.t.c. This is due to the traditional option of making a program built in parallel is with something like make with the option j for the number of jobs to run. Currently there are two ways to support this feature with both having tradeoffs that will need to be discussed. The first way to interact with make would be to implement or interact and allow make to rather than run multiple compilers with make switch its behaviour to launch a number of threads equal to the number used with -j. The biggest advantage of this is that it's the simplest to write and implement, however the cost of allowing the user to know how many internal compiler threads are ideal for their build requires trial and error plus leading to user inconvenience. Therefore, the other way of interacting with make is to allow hook gcc's multi-threading support into build systems job servers and launch separate threads as related when the job servers require it. The biggest advantage of this is that the user will not have to do trial and error in order to get make or other build systems to scale properly when using -j. However the disadvantage in this approach is complexity in the interaction between make and the build system now being more tightly coupled than before in parallel builds.

Heuristics data for partitioning and when to partition gcc passes between threads

In gcc there are several passes most notably pre RTL for SSA form that are not known to be benefit successfully always from multithreading or partitioning the to be run in separate threads. Therefore for these and other passes we will be required to add heuristics data to detect when partitioning into several threads is ideal. Currently I'm exploring and thinking about what data to detect on but have not come to conclusions based either on what gcc currently has for this, which is very little or what to add in terms of support for it.

Conclusions and future pages related to the project

While this article was rather brief considering the ideas, the goal was to get a theoretical overview of how to best extend gcc to improve build times by making it multi-threaded in various areas. The complexity of the project will require several phases related to discussed its implementation and gathering better data as related to shared state in gcc. Currently I've divided the project into four major phases of which this wiki article is the first. In addition as other parts are completed we may need to write secondary extensions to the gcc wiki or manual to support this project.

Phases of the Project

1. Theoretical Overview and Ideas of the Project - Completed
2. Figuring out how to and collect heuristics data from gcc in order to scale correctly
3. Test Implementation of Various Features Based on the data found in phase 2
4. Final implementation of the project based off the test implementation but scaled for the whole project