A parallel framework for the GCC

The Gomp Development Team

March 12, 2003

Abstract

We discuss the possible infrastructure for our OpenMP realisation

1 Introduction

This document discuss how to made concurrency-aware the current realisation of the Gnu Compiler Collection (GCC). The goal of the model discussed here is to set up a framework for introducing OpenMP directives in the compiler(s). We keep the discussion as general as possible, with in mind the possibility of reusing much of this work for other shared memory (SM) parallel models, although the specific nature of OpenMP will sometimes drive our choice to a precise direction. We will underline OpenMP specific points where they will appear.

2 GCC and concurrency

The realisation of OpenMP we have in mind is not a source-to-source translation, like e.g. OdinMP. We aim at tight integration with the compiler. This also mean detailed tree analysis and optimisation. Control flow analysis in parallel programs is different from that in serial program. A correct analysis needs concurrence-awarness in the middle end. GCC in its present state is not concurrency aware. Our first goal then must be to introduce concurrency-awarness in the compiler. The following sections will expose a still incomplete, very modest and probably wrong proposal.

3 From Directives to IR

To be concrete, we consider the C/C++ specifications for OpenMP (**Scott: can I talk of C/C++ specification or should I make a distinction here?**).

The OpenMP specifications have three fundamental parts: directives (introduced in form of pragmas), environment variables and a library. For the problem we are discussing here we only need OpenMP pragmas. Those specify parallel regions, parallel loops, sections, barriers, critical regions and atomic writes. We can group them into two subcategory: directives that have to do with the fork/join of the threads (the first three) and directives that govern synchronisation/locks (the last three)[1]

3.1 Thread Creation

On entering a parallel regions, the program will fork the appropriate number of threads, as specified in the order by a call to openmp_set_num_threads(), the corresponding environment variable and the default. What characterise a thread are the list of the variable shared among the threads, the list of the variables private to each thread and the code to be executed by a thread. At the frontend level, we should create a scheme for the threads suitable for the tree representation. Essentially, a program section that looks like

```
int foo;
flaot bar;
...
#pragma omp parallel shared(foo) private(bar)
{
bar = F1(thread_id) + F2(foo)
}
```

with F1 and F2 given functions, must go to something like (**Notation borrowed from Diego, to me it seems appropriate here**)

```
int foo;
real bar;
...
cobegin
T0:
int thread_id = 0;
float bar = 0.0;
bar = F1(thread_id) + F2(foo)
T1:
int thread_id = 1;
float bar = 0.0;
bar = F1(thread_id) + F2(foo)
...
TJ:
int thread_id = J;
```

[1]Actually things are more complicated, since the parallel loops and sections directives can appear either inside a parallel region or (with a different pragma) independently, indicating in this case a parallel region that will serve a single purpose (a parallel loop or a section case). More on this in section 3.1.

```
float bar = 0.0;
bar = F1(thread_id) + F2(foo)
coend
```

Note that the private variable must be initialised to zero within each thread, while the shared variable is inherited from the serial part.

This should be enough, the other fork/join cases can be reduced to this one. Let's see it case by case

1. *for directive within a parallel directive*: there are two new private variables, the starting point and the end point of the loop, which are functions of the thread identificator *thread_id*
2. *sections directive within a parallel directive*: it is really a series of *if* case based on *thread_id*
3. *parallel for directive*: it is a simplified case of 1
4. *parallel sections directive*: it is a simplified case of 2

Note also that although we have started from an OpenMP-specific program, the structure to which we arrive is general and can be used by any threading model. (**not sure though about the generality of the minimal subset.**)

That structure is then propagated to the Middle End and will be the basis for the intermediate representation. What we do from here on will depend: (a) on the Optimisation framework (**still no clue from my behalf for the time being**); (b) on the chosen threading model (**I still regard Posix Threads as a good candidate**). Note that (b) the choice of (b) does not create any constraint at this stage: the chosen model can be dropped in favour of another one in a following moment, without too much of a disruption to the framework.

3.2 Synchronisation, Mutual Exclusion and Memory Issues

(**I still have to get to thread synchronisation directives, like barrier and flush, but I suspect this will largely depend on the used threading model. We should deal with this by creating a wrapper.**)

The *critical* and *atomic* directives can be implemented with a mechanism of locks.

Memory synchronisation should not be a big concern (**AFAIK**) if we want to tackle just OpenMP: it is responsibility of the user not to assume implicitely how some variable he is using in a thread have been processed by another thread. If the programmer wants a shared variable to be immediately visible to all threads, it must explicitely flush it to memory (with all the implications this will have for the update of the cached copy...).