CSC791A Final Project Report 2

Todd Gamblin & Prachi Gauriar

Milestones

We had two main goals for this phase of the project. First, we wanted to find a parallel code for which both OpenMP and MPI versions exist. We wanted to find code that could serve as an example for the transformations that we plan to write in SUIF. Second, we wanted to familiarize ourselves with the SUIF framework and set up a development environment for writing SUIF passes.

1. <u>Selected model program from NAS benchmarks for parallelization</u>

We looked at the NAS Parallel Benchmarks for parallel code representative of the transformation we wish to write. Essentially, we want to modify SUIF's parallelizing framework to find coarser grain parallelism and to insert communication calls in MPI to synchronize data between iterations of loops, rather than simply inserting OpenMP pragmas and assuming shared memory. The NAS Benchmarks are simplified versions real-world computational fluid dynamics code. They are well-suited for our purposes, as for many of the benchmarks sequential, OpenMP, and MPI code is provided. This gives us an idea of the coarseness of parallelism necessary for shared and distributed memory implementations, and of how we will need to modify SUIF's parallelizing code.

We have looked through all of the NAS benchmarks and selected MG (a Multigrid solver) to work from. It contains four key routines: a smoother (*psinv)*, a residual calculator (*resid)*, a residual projection (*rprj3)*, and an interpolation routine (*interp*). These routines all perform some computation, and then synchronize data values between the memories of different processors with communication routines.

In the case of the OpenMP version, the main compute loops of these four routines are parallel, while in the MPI version they are sequential. This is a good example of fine vs. coarse-grained parallelism: the designers felt that in the MPI version it was not worth it to execute the compute loops in parallel, while in the OpenMP version the processors are tightly coupled enough to get some benefit from this. The communication between processors in the MPI version comes in the communication wrapper routines, where data is synchronized between processors for the next stage of the multigrid method. In our next phase we will look into converting these high-level observations into formal analysis and integrating them into SUIF.

2. <u>Built SUIF</u>

We have successfully compiled and tested SUIF 2 on Red Hat Enterprise Linux 3. This was nontrivial, as SUIF development stopped in 2000. The most recent version of SUIF is incompatible with current compilers, and requires that many nonstandard libraries be compiled in addition to the basic SUIF install. To complicated matters, many of these libraries exhibit incompatibilities with current compilers as well. To build successfully on our system, we had to downgrade to gcc version 2.96, and then painstakingly modify portions of SUIF and the Omega dependence analysis library.

Once we built SUIF, we were able to successfully compile a nontrivial C file into SUIF code. Now that we have a stable environment, we should be able to move quickly to the development stages of the project.

3. Learned to write SUIF Passes

Despite the complexity of building SUIF, it is extensively documented and has a modular architecture. Modifications to SUIF are written as *passes*, and Prachi read over the documentation for creating these and created a summary that we can work from in the next phase. We include it in our report here to provide an idea of what we have learned about SUIF so far.

> Since our last progress report, I have been studying the SUIF2 Infrastructure documentation in order to identify the most essential classes and methods for adding MPI to SUIF. In particular, I was interested in what classes were required to operate on functions independently, insert code, and modify control flow.
>
> SUIF provides two options for those that want to extend SUIF. One may either create a pass, which operates on code using the existing intermediate representation defined by SUIF, or one may extend the intermediate representation itself due to the fact that the existing representation is inadequate. As it is highly unlikely that we will need to extend the current IR, I focused on what was required to create a pass.
>
> Creating a pass itself in SUIF is fairly straightforward. Since we won't be doing any interprocedural analysis, we need only subclass the `PipelinablePass` class, which would allow us to apply our transformations on each procedure independently. As our strategy will most likely analyze each procedure definition, we will probably override the `do_procedure_definition`, and perform some analysis on the code in the specified `ProcedureDefinition` object. This object contains all the information relevant to representing a procedure, including its symbol, parameters, body, local definitions, and symbol table for the procedure scope. Of these, the most important is the code in the body, an object of class `ExecutionObject` that contains the code for the body of the procedure.
>
> The `ExecutionObject` has two major subclasses: `Statement` and `Expression`. A `Statement` represents code that does not return a value, whereas an `Expression` does have a return value. Both of these classes have a variety of subclasses for the different types of statements and expressions. Of particular interest for us will be the high-level control flow statements, such as `IfStatement`, `WhileStatement`, and `ForStatement`, as well as `LoadVariableStatement`, `StoreVariableStatement`, `CallStatement`, `EvalStatement`, `UnaryExpression`, and `BinaryExpression`. Each of these classes contain information specific to the code they represent. For example, a `WhileStatement` contains the iteration condition, loop body, and labels for continuing or breaking.

Open Problems & Future Steps

1. Learn more about the SUIF dependence framework
We will need to look at SUIF's dependence analysis framework in-depth to determine how loops are parallelized and OpenMP calls are inserted into SUIF output. Moreover, we will need to determine whether this analysis can be modified in-place to suit our needs, or whether we need to write an entirely new pass. I see two possible outcomes here, and we'll need to decide on one in the next phase:

    a. SUIF's analysis is not sufficient, and we'll need to add our own pass to detect more coarse-grained parallelism, in addition to a pass to insert valid MPI calls.
    b. SUIF's analysis will detect the kind of parallelism we need for MPI communications, and we can devise a way to use this information in our own pass to insert MPI calls.

2. Write simpler versions of the Sequential, MPI, and OpenMP versions of MG in C
The multigrid code we have looked at is long and will be cumbersome to use for testing with SUIF. We will need to extract the important loops and make test programs to feed to SUIF. We can start with the sequential version, compile it, and see what kind of OpenMP code SUIF produces. We can then compare this to the OpenMP versions of the multigrid code, and devise a mapping to MPI.

3. Start developing SUIF code
Once we have devised a plan as per steps 1 and 2 above, we can begin developing with SUIF. We hope to be ready to start by the end of the next project phase.