Automatic Parallelization of Sequential C Code

Pete Gasper
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
peter.gasper@gold.sdsmt.edu

Caleb Herbst
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
calebherbst@hotmail.com

Jeff McGough
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
jeff.mcgough@sdsmt.edu

Chris Rickett
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Christopher.Rickett@gold.sdsmt.edu

Gregg Stubbendieck
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
gregg.stubbendieck@sdsmt.edu

Abstract

As the cost of hardware declines and the demand for computing power increases, it is becoming increasingly popular to turn to cluster computing. However, in order to gain the benefits of cluster computing, an existing software base must be converted to a parallel equivalent, or a new software base must be written. Both options require a developer skilled in both parallel programming, as well as the problem domain at hand. The ability to automate a conversion from sequential C code to a cluster-based equivalent offers a developer the power of parallel computing with a minimal learning curve.

The following paper describes an ongoing project with the goal of automating the conversion from sequential C code to cluster-based parallel code. Currently the project relies on user input to guide the automation process, focusing on loop level parallelization. Long term goals center on using dependency analysis to automate the parallelization process.

1 Introduction

As the cost of hardware declines and the demand for computing power increases, it is becoming increasingly popular to turn to cluster computing. However, in order to gain the benefits of cluster computing an existing legacy software base must be converted to a parallel equivalent, or a new software base must be written. Both options require a developer skilled in both parallel programming, as well as the problem domain at hand. The ability to automate a conversion from sequential code to a cluster-based equivalent offers a developer the power of parallel computing with a minimal learning curve. This paper will discuss the South Dakota School of Mines and Technology Cluster Group's approach to automatic parallelization of legacy code.

Parallelization of sequential algorithms can be a difficult task. The tool produced by the project aids the user in understanding a given program and the existing data dependencies. Analyzing the sequential code and providing a graphical depiction of the program execution are examples of such aids. The graphical representation then provides a means for the user to define areas of code to be parallelized, as well as describe data blocks involved. The user-provided information is necessary for the automated parallelization process, in which Message Passing Interface (MPI) based code is produced to replace the sequential code. One future project goal is to automate the dependency analysis aspect of the user interaction. However, a certain level of user interaction is often needed, or may improve, the parallelization process. In general the automated approach may take a more conservative approach in determining data dependencies, therefore the user may be able to describe a further degree of parallelization.

2 Development Language

Java was chosen as the development language due to its cross-platform and object oriented appeal. The project currently focuses on converting legacy C code to an C/MPI parallel equivalent. The object-oriented nature of Java will allow the project to expand its capabilities to further target languages such as Fortran 77 and Fortran 90, by adding the necessary language objects.

3 Design Stages

Below is a brief description of the design stages chosen to implement the project. Each step will receive greater detail throughout the remaining document.

1. Parse legacy C source code.
2. Create an abstract syntax tree representation of original C source.
3. Create a flow graph representation of the original C source code.
4. Create an intuitive, user-friendly, interface for the application.
5. Perform dependency analysis to determine parallelizable code segments.
6. Develop a workpool model for work distribution among nodes.
7. Emit the parallelized version of the source code.

The design diagram is shown in Figure 1.

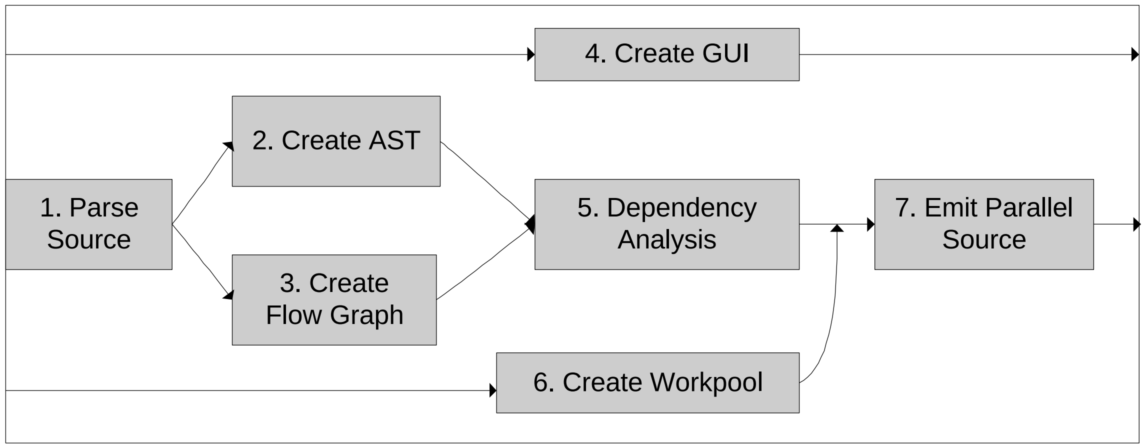

Figure 1: Application Development Model

3.1 Parsing C Source Code

Parsing the source code paves the way for building the abstract syntax tree and flow graph representation of the source code. Several language translation utilities exist. ANTLR (<u>A</u>nother <u>T</u>ool for <u>L</u>anguage <u>R</u>ecognition) translator generator was chosen due to its ability to generate Java source, as well as its inherent ability to generate parsers capable of creating an abstract syntax tree on the fly [5].

ANTLR generates LL(k)-based recognition tools, meaning they perform a left-to-right scan, using a leftmost derivation technique, looking ahead *k* tokens. A leftmost derivation means the leftmost non-terminal is always expanded [5].

ANTLR was used to generate our C lexical analyzer and parser as Java source files. These two files are named CLexer.java and CParser.java accordingly. The lexical analyzer is created by defining tokens to match in the input stream. These tokens are then passed to the parser, which matches a pattern of tokens into predefined grammar rules. For ANTLR to generate the parser, a C grammar file must be created. The grammar file defines the valid C language syntax rules. Below is an example parser rule defined within the C grammar file.

```
assignmentOperator
      : ASSIGN
      | MUL_ASSIGN
      | DIV_ASSIGN
      | MOD_ASSIGN
      | ADD_ASSIGN
      | SUB_ASSIGN
      | LEFT_ASSIGN
      | RIGHT_ASSIGN
      | AND_ASSIGN
      | XOR_ASSIGN
      | OR_ASSIGN;
```

The above rule states that an assignment operator consists of either an =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, or |= symbol.

The next segment of the document will discuss the building of the abstract syntax tree using our ANTLR generated parser.

3.2 Abstract Syntax Tree

In order to analyze the original source, the code must be stored for repeated access. The data structure chosen to represent the source code internally is an abstract syntax tree (AST) [4]. The AST represents the syntactic structure of the original source code. It stores enough information to both analyze and reproduce the original source code [2].

The AST is designed such that blocks of code are grouped into a single subtree, with a root node used to recognize the structure beneath. For example, the code contained within a 'for' loop would be contained under a root node labeled 'forLoopStart.' These labeled root nodes offer an easy traversal, allowing possible parallelizable code blocks to easily be found.

The AST consists of nodes; each node holds the following information:

1. Token text - Text used to identify the token that was parsed and is stored in the node.
2. Token type - The type associated with the node. For example, XAssignmentOperator is the type associated with an '=' sign.

Other information may also be included, depending on the node type. For example, nodes that contain compound statements also have the scope of the node included for finding information about variables within the symbol table. ANTLR provides methods that will allow the information held by a node to be manipulated.

The AST is built during the parsing of the input source code. ANTLR allows for an AST to be generated automatically based upon the parser grammar rules. ANTLR also allows the AST to be hand generated using user added directives within a given parser rule [5]. By providing the preferred handling of the rules, some rules may be ignored and others operated on to better structure the AST to match project needs.

The recursive nature of the grammar, specifying parsing rules, and the building of an AST during the parsing, simplified the process of grouping code blocks into subtrees. An AST node is defined for almost every token, except for tokens such as the ';' at the end of statements. Furthermore, descriptive root nodes were added to label the subtrees, such as the previously mentioned forLoopStart node. In general, the default ANTLR structure of the AST was used, with minor changes in the ordering of child nodes and the removal of others. These changes were made to simplify the AST structure, and do not necessarily preserve precedence of the included tokens. Further, some ANTLR generated nodes were not necessary for the analysis and were removed to reduce the size of the AST. For example, the left and right parentheses of a function call were removed. The parentheses may be assumed during the emitting of the code, later once the validity of the syntax has been verified.

ANTLR has another feature, which is the ability to generate a tree walking routine based on a grammar. The grammar is used to specify the structure of any subtree that may be underneath a given node. By using a tree grammar, it is easier to define all possible subtrees for a given node. These tree grammars offer a means to traverse a given AST, while providing the ability to perform analysis operations while walking the nodes [5].

Rules defined by the tree grammar are not the same as those defined in the parser, due to the removal of unnecessary syntax during the AST generation. Once the AST is built, the rules for describing the subtrees of a node became simpler. However, one problem encountered is rule ambiguity. By removing syntax, many rules in the tree walker could be reduced to the same token leading to ambiguous rules. These ambiguities must be removed to allow the tree walker to traverse the entire tree.

The AST offers a good generalized data structure to organize syntax, however to maintain the overall flow of the program, another structure must be employed. The flow graph representation is used to maintain the needed structure.

3.3 Flow Graph Representation

The flow graph provides the organization needed to analyze the flow of the program, and to discover whether sections of the source are parallelizable. The flow graph is formed by classifying the source code in terms of basic blocks.

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [1]. Basic blocks are connected to each other using directed edges. A flow graph has two special purpose blocks that define the entrance and exit points of the block. The starting block is the single point at which execution starts in the flow graph, and the exit block is the single point where execution leaves the flow graph.

A basic block contains a block number, a vector of AST root nodes, and the type of the basic block. The block number is used to preserve ordering of the basic blocks. The AST nodes are the roots to subtrees containing the source within the basic block. These AST root nodes allow the source code contained within a basic block to be directly accessed from that block. The basic block type represents the type of source contained within the basic block. For example, the block may be of type DoWhileStart, Conditional, ForLoopStart, etc.

A sample flow graph is shown graphically in Figure 2.

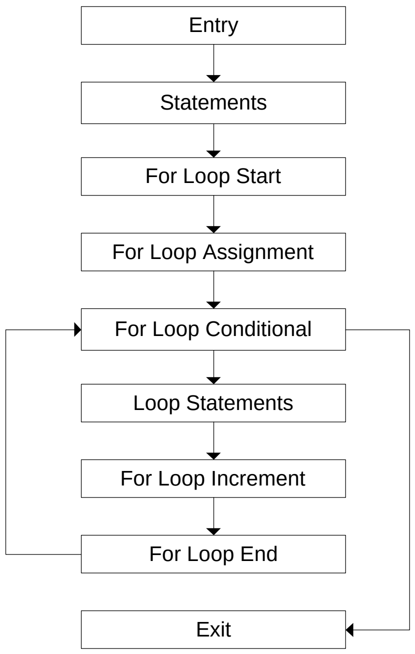

Figure 2: Flow Graph for 'For' Loop

The flow graph and abstract syntax tree store a generic representation of any serial source file. They will be our main data structures for determining parallelizable code segments.

3.4 User Interaction

The long-term goals of this project focus on generating dependency analysis algorithms capable of automatically recognizing parallelizable structures within the AST. Currently, the application relies on user interaction through the graphical user interface. This interaction allows the user to select parallelizable code segments from a graphical depiction of the serial source code flow graph. The user may then define supporting parameters to guide the generation of the parallel source code.

The following terminology will be used while discussing the parallelization process. A *parallel scope* is a section of the flow graph selected by the user that contains parallelizable components. The parallel scope is then broken into specific tasks. Finally the tasks are broken into blocks. There are two layers of possible parallelism inherent in the method. First of all, it may be possible to execute multiple tasks concurrently. Furthermore, each task may be composed of blocks, which allow for data parallelism.

3.4.1 Defining Parallel Scopes and Tasks

After studying the flow graph resulting from the serial source code, the user must decide what to parallelize. The user may rubber band a section of the flow graph to be parallelized. Within that rubber-banded section, the user may then rubber band individual tasks to be performed. There may be multiple tasks that will yield parallelism. Once a user has defined a task, the graphical user interface will provide a means to enter parameters needed to describe blocks for each specific task.

3.4.2 Block Definition

As mentioned, a task is comprised of blocks to promote parallelism. A block is not restricted, in general, to representing a contiguous data item. Each block contains supporting data to describe properties of the block. The following example is taken from Dongarra, Duff, Sorensen, and van der Vorst [2], page 45. Figure 3 depicts a triangular solver block diagram. Each block is defined using a user entered size, shape, and movement. Each block represents an m by n group of data elements.

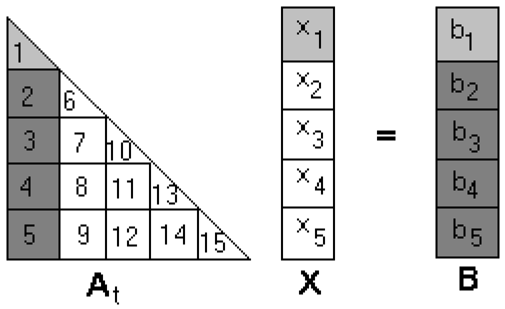

Figure 3: Triangular Solver Block Diagram

The first step in solving the system is computing the system $A_1x_1 = b_1$. After x_1 has been computed, the result may be used to update B in the following manner.

$b_2 = b_2 - A_2x_1$
$b_3 = b_3 - A_3x_1$

$b_4 = b_4 - A_4x_1$
$b_5 = b_5 - A_5x_1$

The equations may be executed in parallel due to the lack of data dependency between them. The light gray blocks in Figure 3 represent the initial $A_1x_1 = b_1$ blocks that must used to solve x1. However, the dark gray blocks represent the update steps that may be parallelized.

We will focus on the first iteration of the triangular solver update step to give example block definitions. For matrix A_t in Figure 3, block 2 may be with a width of m, and a height of n, entered by the user. For this example, block 2 does not need any movement. Block 2 could simply be sent with blocks x_1 and b_2 to perform the update on a given compute node.

The information provided by the user describing the blocks and movements involved in a task is used to create a directed acyclic graph (DAG) describing the ordering of tasks for execution. A DAG further describes the degree of parallelism possible. Figure 4 shows the DAG for the first step of the triangular solver. In the figure, tasks 2-5 depend upon task 1. Once task 1 is completed, tasks 2-5 may be executed in parallel. Upon their completion, task 6 can be executed.

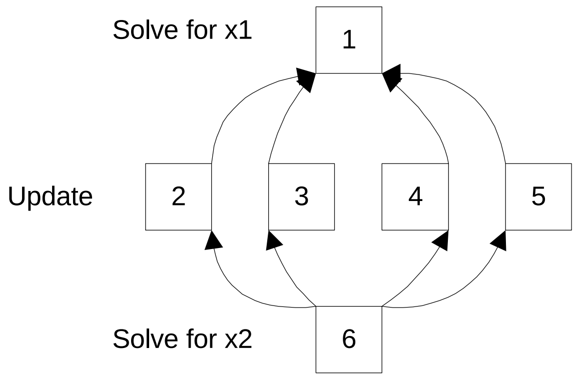

Figure 4: Update Step DAG

4 MPI Workpool Model

The modified Message Passing Interface (MPI) Workpool is a model for accomplishing tasks through a master-slave scheme. The workpool uses MPI for inter-node communication purposes. The master node manages the workpool by keeping a list of work to be done, keeping track of order of execution, and dispatching work. Worker nodes request work, and act on that work using minimal instructions obtained from the master node. After a worker completes a piece of work, it again queries the manager for further work.

5 Source Emitter

The final step in the automation process is the MPI code generation. In this last step a code emitter interacts with the flow graph, AST, and various data structures to recreate the sequential code in parallel form. The emitter is designed such that the general structure of the emitter remains the same, with only the problem specific portions changing between source conversions.

The MPI-based parallel code is emitted into two files: master.c and slave.c. This differs from the generally used single program multiple data model. These two files contain the code that will run on the master node and slave nodes respectively. The emitted master source contains the non-parallelizable source with newly inserted function calls to the MPI workpool where parallelized sections have been moved to the slave source. The master source also contains newly emitted helper functions to initialize the workpool model for each removed parallel section. These functions are responsible for initializing any arguments to be handed to the worker nodes, as well as any blocks to be defined for the workers, and finally the general MPI initialization such as obtaining the number of worker nodes available within the cluster.

While the master node has the job of managing the workpool and the program’s flow, the worker nodes are restricted to parallelizable work presented via the workpool. Each worker node is capable of executing several different work items depending upon the master's request. Upon startup, the worker nodes perform a basic initialization, including determining how many nodes are present in the cluster, as well as the initialization of the Message Passing Interface engine. When a task is delegated to a worker node, that worker selects the corresponding source to execute and performs the request. These tasks are emitted into the slave source in the form of a case statement, with a given TaskID as the lookup.

The goal of the source code emitter is to create a generalized means to generating parallel source code. The overall structure for problem solving, using the workpool model, remains consistent. However, problem specific portions of the master and slave source must remain flexible enough to support the changing individual application demands. Much of the support for generalization comes from the expandable objects inherent in the Java language. Objects such as vectors and lists, offer a means to create dynamically sized data structures. An example for such a demand might be the varying number of blocks needed to describe different parallel segments within an application, or the varying number of function prototypes to be emitted from application to application. The more frequently such general data structures may be used, the more flexible the emitter becomes.

6 Conclusion

The preliminary stages of this project centered on building the framework necessary to promote automatic parallelization of sequential C code. The framework provides a means to parse the sequential code into a generalized data structure and create a graphical representation of the overall program flow of execution. Currently the user has the ability to guide the automation process. Future work will focus on performing background dependency analysis, reducing the amount of user input necessary to complete the equivalent MPI source generation. In order to gain optimal parallelization, a balanced combination between automation and user input may need to be reached.

This project is a step towards creating a smooth transition from a sequential code base, to a cluster-based equivalent. By reducing the time and effort needed to produce a parallel code base, more time may be dedicated to solving the problem domain.

References

1. Aho, A. V., Sethi, R., Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools.* Murray Hill, New Jersey: Bell Telephone Laboratories, Inc.

2. Dongarra, J. J., Duff, I. S., Sorenson, D. C., & van der Vorst, H. A. (1991). *Solving Linear Systems on Vector Shared Memory Computers*. Philadelphia, Pennsylvania: Society for Industrial and Applied Mathematics.

3. Morgan, R. (1998). *Building an Optimizing Compiler*. Woburn, Massachusetts: Butterworth-Heinemann.

4. Muchnick, S. S. (1997). *Advanced Compiler Design & Implementation*. San Francisco, California: Morgan Kaufmann.

5. Parr, Terence (02/16/2003). *ANTLR – Complete Language Translation Solutions*. Retrieved June, 2003, from http://www.antlr.org.