N90-16454

A GENERIC FINE-GRAINED PARALLEL C

L. Hamet J. Dorband

NASA/Goddard Space Flight Center/635
Greenbelt, MD 20771

ABSTRACT

With the present availability of parallel processors of vastly different architectures, there is a need for a common language interface to multiple types of machines. Our parallel C compiler, currently under development, is intended to be such a language. This language is based on the belief that an algorithm designed around fine-grained parallelism can be mapped relatively easily to different parallel architectures, since a large percentage of the parallelism has been identified. The compiler generates a FORTH-like machine-independent intermediate code. A machine-dependent translator will reside on each machine to generate the appropriate executable code, taking advantage of the particular architectures. The goal for this project is to allow a user to run the same program on such machines as the Massively Parallel Processor, the CRAY, the Connection Machine, and the CYBER 205, as well as serial machines such as VAXes, Macintoshes and Sun workstations.

Keywords: Fine-Grained Parallelism, Portability, Operator Overloading, Massively Parallel, SIMD, MPP, C, Data Parallel.

INTRODUCTION

As the variety and availability of parallel machines increases, the need for a portable parallel compiler becomes critical. To be effective, however, this compiler must be able to take full advantage of each machine's unique architecture. Two concepts are necessary to achieve these goals: modularity or layering and fine-grained parallelism.

A modular compiler design allows machine-dependent characteristics to be separated from the machine-independent (generic) characteristics. The object code generator (or P-code translator) is the lone machine-dependent piece of the compiler; consequently, parsing may be done once, with the output submitted to various versions of the machine-dependent layer, one version for each unique machine.

Fine-grained parallelism assigns the task of extracting parallelism within code to the programmer. The user-defined parallelism may be mapped to any architecture, since it may be easily assembled into a serial implementation or a parallel implementation of any desired degree. A parallel description of a program is much easier and straightforward to assemble into a serial description than a serial description into a parallel one.

It may seem as though it is an unreasonable task to expect a skilled programmer, let alone an unskilled one, to extract parallelism from an algorithm when it is recognized that this is such a difficult task for a compiler. The members of the MPP Working Group have shown that both skilled and unskilled programmers alike may easily extract fine-grained parallelism. In reality, extracting fine-grained parallelism is no more difficult than recognizing what code must be repeated within a loop in serial code.

PROGRAMMING MODEL

The C language implementation supported here is based on a model of computation where there is one serial processor (the control unit) and many independent SIMD parallel processors (ALUs controlled by the control unit). Serial data is stored in the control unit memory (S) and parallel data elements are stored in the memory (P) of the parallel SIMD processors. The same operation is performed simultaneously on parallel data elements. The only exception to this occurs if a processor is masked out of the operation. Parallel control structures using this mask capability provide a means of restricting operations performed within a parallel processor to only those operations that apply to the data in that processor.

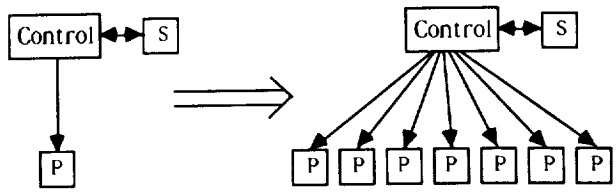

Figure 1. Logical View of SIMD Processing Model

Programming such a model (Figure 1) can be viewed as programming a single controller that has two memories, P and S. If data

from S is used with data from S, the result remains in S. If data from P is used with data in P, the result remains in P. However, if data from S is used with data in P, the result must remain in P. Totally independent operations need not be concerned with the fact that there are many P memories. This model is an oversimplification, since it does not account for inter-processor interactions.

Most operations performed by a SIMD processor do not involve data from different processors, but the need for inter-processor interaction does arise. Inter-processor communication and data reduction operations facilitate the ability for parallel data elements to interact . Data reduction operations produce a single result from data in many parallel processor memories (P) and store the result in S.

Consider more closely Figure 1, containing two views of SIMD processing: a simplified model and a complete model. The simplified model consists of a control unit and two memories, S and P. The data in S is considered serial data and data in P is considered parallel data. The complete model differs in that it has multiple P memories. If the complete model is run with all but one processor masked out, it will give the same result as the simplified model (with the exception of inter-processor communications). The results from the simplified model should be the same as a serial processor, where S and P make up the memory of the serial processor.

In the simplified model, the control structure should act the same whether the condition is based on results in P or results in S. The complete model has multiple P memories; the data in each should be manipulated only by those instructions that are pertinent to it. This means that some processors must be turned off based on conditions computed in them. These conditions result in a determination that the corresponding conditionally executable code is not pertinent. Actually, only code that effects the user detectable state is masked. This includes assignment statements and conditional expressions (?:).

The preceding has several subtle implications. 1) Code within control structures, where the conditional result is in P, must be executed as long as the condition is true for at least one processor. 2) More subtly, however, if the condition is not true for any processor, then the code must not be executed. The subtlety is that although no processors' memory (P) will be modified by the code within the parallel control structure, data in S might be modified by executing this code. However, this violates the above constraints of the simplified model, and consequently must be prevented from occurring.

LANGUAGE DESCRIPTION

The popularity and flexibility of C made it the natural choice as the language to be implemented in such a manner. Parallelism is achieved through operator and control structure overloading (to be further explained). This preserves the Kernighan and Ritchie[1] look of C, yet allows a wide range of levels of parallelism to be implemented, depending on the targeted machine. The parallelization of C is based on experience learned in the development of MPP Parallel Forth[2]. The only syntactical addition to the language is the storage class PARALLEL.

The language has been altered slightly to accommodate the parallelism. Due to the different architectures of the machines using this compiler, the storage class REGISTER has been eliminated. On the other hand, for the sake of bit serial processors, the ability to specify number of bits in a declaration has been expanded to all variables, not only to fields within a structure or union. Depending on the machine architecture, however, the programmer may get more precision than requested, but never less. All cases of precision increases will be consistent and documented for each version of the compiler.

COMPILER DESIGN

The C compiler is divided into four components: the scanner/parser, intermediate code generator, intermediate code translator, and virtual machine. The scanner/parser and intermediate code generator are machine-independent; the translator and virtual machine must be rewritten for each machine type.

The scanner/parser is an SLR(1) parser, written without the use of the UNIX™ utilities yacc™ and lex™, due to Macintosh memory partitioning limitations. (The Macintosh II is the first machine for which a version of the compiler is being written.)

The intermediate code generator generates postfix P-code. This style was chosen because of its speed and minimal size. Furthermore, because it is English-based, it is not difficult to read. Since the intermediate code generator is really only a postfix converter, this module remains machine-independent.

The translator converts the P-code to a FORTH-like "assembly". FORTH, a stack-oriented language, was chosen because of its speed and register simplification. Furthermore, based on prior experience with the Massively Parallel Processor, FORTH has been demonstrated to be a logical and efficient language to run as a virtual machine for SIMD architectures; each processing element memory is treated as a stack.

The virtual machine is a simple FORTH engine, actually coded in C, which executes the "object module" output from the translator.

Although we have implemented both the translator and virtual machine as machine-dependent modules, the translator could be generalized so that it would be machine-independent, requiring only recompilation with a modified include file. However, we opted against this, avoiding as much unnecessary overhead (speed loss) as possible.

OVERLOADED OPERATORS

There are no operators added to the parallel C; all existing operators are overloaded. The version of each operator routine called by the translator is determined by the types of the oper-

ands. Unary operators are trivial; there is a version for each parallel and serial type. For binary operations, if both operands are serial or both parallel, the result corresponds. Standard C conversion rules still apply to both serial and parallel (always to the greater precision, signed to unsigned, and integer to floating point.) Added to these rules, however, is serial to parallel conversion when these two types of operands are operated on together. A serial to parallel conversion is equivalent to a broadcast of the serial value.

The bit shift operators (<< and >>) take on interesting results when done in parallel. For these operators, where the left operand is the value and the right operand is the number of bits by which to shift, the unusual case occurs when the number of bits by which to shift is parallel. If the operand is a variable, different processors may contain different values. The operation is implemented with a parallel mask, where, after each bit shift, the processors which have completed the required number of shifts are masked out, until all have completed.

Parallel logical operators (&& and ||) are implemented with parallel versions of the if-else structure (See Overloaded Control Structures). Parallel addressing operators (* and &) are undefined, as parallel pointers are not implemented in the current implementation of the compiler.

Parallel Pointers

Although parallel pointers are not implemented in this version of the compiler, serial pointers to parallel variables are legal. They must be declared in two parts. The parallel data object must be declared as a type, then the pointer variable is declared as a pointer to that type, in a separate declaration. To illustrate,

```
typedef parallel int A;
A *ptr;
```

is legal, whereas the declaration

```
parallel int *ptr;
```

would be recognized as a parallel pointer declaration, and flagged as an error.

Parallel Assignment Operators

Assignment operators (=, +=, \=, &=, etc.) do not observe the standard conversion rules, because the resultant type must be the type of the left operand — the one receiving the final value. Serial—serial and parallel—parallel left-right operand pairs are trivial; no serial—parallel conversion is necessary. Parallel—serial requires a standard serial to parallel conversion. Serial—parallel, however, yields interesting results.

Up to this point, no operations involve the data in different parallel processors. When parallel data is assigned to a serial variable, a data reduction operation must be performed. This involves data in all the parallel processors. A simple assignment (=) of a parallel to a serial is implemented as a bitwise cumulative OR over all values of the parallel operand, with the serial operand being set to the resulting value.

Each complex assignment operator is treated uniquely. The addition-assignment (+=) is implemented with a cumulative sum added to the serial operand; the subtraction-assignment (-=) subtracts the cumulative sum from the serial operand. Multiplication-assignment (*=) and division-assignment (/=) are treated comparably, with a cumulative product.

Bitwise AND-,OR-,and XOR-assignments (&=, |=, and ^=) are implemented as expected: a cumulative AND/OR/XOR is done over all values of the parallel operand, with the serial operand being set to the resulting value.

Modulus-assignment (%=) and shift-assignment (>>=, <<=) are undefined for the serial—parallel case.

OVERLOADED CONTROL STRUCTURES

All control structures in the compiler apply to both serial and parallel conditions. Each structure is executed in parallel if the test expression evaluates to parallel.

For SIMD machines, all structures must use a parallel mask, to mask out processors which have failed the test condition. A bit in the mask is set or cleared based on the value of the test expression in the corresponding processor.

Parallel control structures consist of the same structures as serial control structures: if-else,while, for, and switch. For code to be executed in a parallel control structure, at least one parallel processor must require it.

In a parallel 'for' loop, either the initialization or incrementation expressions (or both) may be serial, as long as the test expression is parallel.

An example of a parallel 'while' loop is the C code:

```
parallel int a,b;
long c = 0;
        .
        .
        .
while (a >5) {
  b *= a;
  a—;
  c++;
}
```

resulting in the statements inside the loop being executed for each processor where that element of the array a is greater than five. Since c is a serial variable, it will be incremented each time the loop is executed, thus counting the maximum times the loop is executed for any processor.

INTER-PROCESSOR COMMUNICATION

Inter-processor communication allows data from different processors to interact. This is performed by adding an integer value (n) to a pointer to a parallel value (i.e. *(para+n)). The data (*para) in processor m+n modulo the number of processors will be moved to processor m. However,this does not in any way imply the time it takes to perform the move. It is totally architecture dependent.

COMPILER CONSTRUCTION

Whereas the scanner/parser is standard SLR(1) and the code generator is a straightforward postfix converter, the interesting design issues pertain to the translator. The most notable points are the symbol table setup and serial and parallel memory allocation.

The symbol table is "a linked list of linked lists". All variables are chained in a list in the order encountered in the code. Each variable is, in turn, the beginning of a definition chain. Since C allows loosely formatted type definitions, the only consistent method to create a definition, for type checking purposes, is to chain the "pieces" of the type definition. A piece can be a base type (e.g., int, char, float) or a modifier (e.g., pointer, array dimension, parallel). Type checking is done by walking the chain.

Memory allocation is handled with four constants, defined by the virtual machine. These are LP, GP, PLP, and PGP — local pointer, global pointer, parallel local pointer, and parallel global pointer, containing the starting address of serial local variables, serial global variables, parallel local variables, and parallel global variables, respectively. The translator keeps track of the last space allocated in terms of offsets for each of those constants. Variables are thereafter referred to by address in the object code generated by the translator. Because the translator is FORTH-like, the virtual machine is composed of FORTH "words", functions executed when named. The virtual machine defines LP+, GP+, PLP+, and PGP+, to add these constants. Consequently, addresses appear as an offset, followed by one of those words. (Remember that FORTH uses postfix format.) The lone requirement of the virtual machine is that a block of memory be explicitly allocated before manipulated. Therefore, allocation statements may appear throughout the generated object code. Two more FORTH words are defined by the virtual machine — ALLOC and PALLOC, for serial and parallel memory allocation.

Parallel variables are actually allocated both parallel and serial memory. A serial longword (four bytes) is allocated to contain two word-length values: parallel starting address and size. Hence, parallel variables are referenced just as serial variables. The information in the serial longword is used at execution time to locate the parallel variable.

POSSIBLE USES

This compiler will have versions on both serial and parallel machines. (On serial machines, parallel structures and operations are implemented serially.) Furthermore, a program need only be retranslated, as opposed to completely recompiled, to be run on a new machine. Consequently, a natural use of this type of compiler is to use a serial machine as a simulator for a parallel machine. This would offload much of the traffic on more costly parallel machines during parallel code debugging.

Another possible use for this type of compiler is in conjunction with the newest computational strategy: network computing. Each node of the network would have its own version of the machine-dependent portion of the compiler. An entire program would be run through the machine-independent section of the compiler, then each piece of the program would be translated by the machine-dependent portion corresponding to the machine on which that section is to be run.

IMPLEMENTATIONS AND FUTURE PLANS

The first complete version of this compiler is currently being implemented in Macintosh Programmer's Workshop (MPW) C on an Apple Macintosh II workstation, to execute serially. Since the code itself is written in C, using only the simplest library routines (to ensure portability), it will be trivial to port the same code to other serial workstations. The next target is the Sun.

For parallel machines, a new version of the translator and virtual machine must be written. The first type of parallel architecture for which a version will be written is an array processor, such as the Massively Parallel Processor or other commercially available SIMD processor. After this implementation there are plans for a vector processor version, such as for a CRAY.

SUMMARY

By isolating the machine-dependent and machine-independent pieces of a compiler, we have created a compiler which reflects a high degree of portability: the same code may be run on very different machines (architecturally) with only partial recompilation. Because the programmer extracts the parallelism, the degree to which this parallelism is utilized becomes a completely machine-dependent issue. Consequently, each architecture is utilized to its fullest, without any code modification.

REFERENCES

1. Kernighan, B. W., and Ritchie, M. D., *The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, NJ.*

2. Dorband, J. E., *MPP Parallel Forth*, Frontiers of Massively Parallel Scientific Computation, NASA CP-2478, September 1986, pp. 275-283.