Reduction of Complexity and Automation of Parallel Execution Through Loop Level Parallelism

Robert A. Tefft
Department of Computer Science
Central Michigan University
Mount Pleasant, Michigan 48859
Email: bobtefft@gmail.com

Roger Y. Lee
Software Engineering and Information
Technology Institute (SEITI)
Central Michigan University
Email: acis@acisinternational.org

Abstract

SIMD (Single Instruction Multiple Data) is a processor architecture classification from Flynn's taxonomy. The concept is that a single instruction set operates on multiple units of data simultaneously. Computers use this processor architecture are known as array processors or vector processors. Most computers in use today are SISD (single instruction single data) though allowing a single instruction to operate on multiple data can also be applied to a virtual machine that is capable of parallel execution through the use of multi-threading/multi-core processors, or distributed parallel execution on a multi-computer grid. This paper proposes a language structure that applies the SIMD concept to the Java virtual machine. The motive is to reduce the complexity of the code and ease implementation of parallelization by running a single set of instructions concurrently on an entire collection of objects.

Index Terms

Concurrency control, Parallel languages, Parallel processing, Parallel programming, Parallelizing compilers, Vector processing

1. Introduction

Parallel executing language statements do exist for supercomputing environments but to my knowledge not for general purpose personal computers. The statement discussed in this paper is an unconditional branch statement which will be referred to as an "all statement." The ability to map this instruction to current parallel code is proved through an algorithm that translates Java code with the statement block to standard multi-threaded Java code.

At the time of this writing Multi-threading and Multi-core processors are beginning to flood the personal computer and server markets. Processors such as the Intel core duo series and the AMD Athlon X2 series have brought multiple cores to the personal computer market, and these companies currently have introduced quad core processors.

As size and heat factors are limiting processor clock speeds the trend to add more and more processors in a single package will continue. For example in the server market the Sun Microsystems T2000 server has eight cores, and even the Playstation three game console has eight processing elements with its IBM cell processor[1]. In addition to multiple cores most of these processors are able to handle more then one thread of execution on each core. This adds up to quite a few simultaneously executing threads. For instance eight simultaneous threads on a quad core processor with hyper threading or even thirty two simultaneous threads on the Sun Utra-Spark processor with its fine grained multi threading technology[2].

As can be seen by the previous examples many new processor architectures are emerging that emphasize parallelism and multi-threading. On previous single core processors with a single thread of execution conventional wisdom held that multi threading was only beneficial for programs that mixed computation with input and output. In this scenario multiple threads would all run on the same processor having the same effect as a single thread. When I/O was involved threads would be blocked while waiting for events, allowing a single processor to benefit through executing additional threads. With new more parallel processors parallelization now benefits all programs. Computationally bound programs can now be split up with computations carried out simultaneously.

In addition to the fact that most processors have historically been single core, concurrent programming turns out to be more difficult introducing: more

Seventh International Conference on Quality Software (QSIC 2007)
0-7695-3035-4/07 $25.00 © 2007 IEEE

complex code, race conditions, and interprocess communication issues. For these reasons most software has yet to take advantage of the increased ability of processors.

Furthermore programming languages designed for the single processor model need to have non standard extensions or complicated library routines in order to run programs in multiple threads. In many cases development of multi threading is avoided as it would add another layer of complexity to a programming project. An easier method of parallelization is needed.

The idea is to test a statement that introduces loop level parallelism to the Java programming language. Loop level parallelism works much like the SIMD (Single Instruction Multiple Data) processor architecture classification from Flynn's taxonomy[9]. The concept is to take a collection and apply a single block of code to every object in the collection in parallel. An example is the FORALL statement in HPF[8]. What I propose is adding such a statement, called an "all statement", to a language (Java in the example). This will certainly not solve the problem in itself, and would indeed (at this point) be just another non-standard language extension, but it is expected that this study sheds perhaps a little light on the problem of parallel execution.

In order to better explain the statement first we will introduce the non parallel "for next" style loop as found in recent scripting languages such as Python, and recently added to Java with the release of Java 5. An example of this style of for loop:

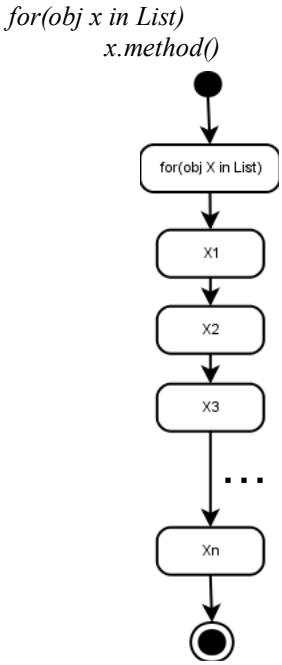

Fig. 1. for next loop

For each iteration inside the body of the loop (here just one statement), the object *x* refers to the next object in the collection named *List*. In this manner the collection is iterated over with no need for an incremental index number as found in many standard for loop implementations. Any collection with ordering is traversed and each object in the collection is given to the block of code in sequence. If the constraint of guaranteed in order traversal is taken away this allows for the code block to execute in any order through the list.

Interestingly enough this allows the code to be applied to all elements at the same time since finishing each thread at different times would not be a problem. This is the idea behind "loop level parallelism" and the "all" statement. The body of the "loop" is executed on each object in the list in parallel. As can be seen on the diagram below the statement is actually an unconditional branch, not a loop. The syntax though is nearly identical:

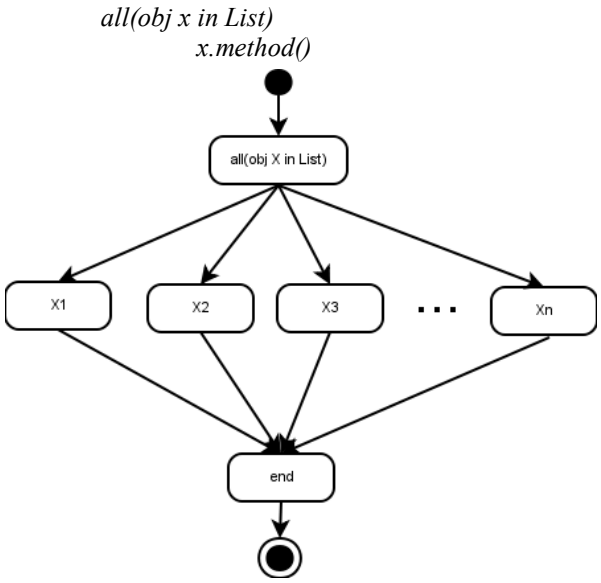

Fig. 2. all statement

2. Related Work

Concurrency support has long been a topic of research in the high performance computing area, and is an active topic of research due to the increased availability of multi-processor computers. In addition there are examples of applying the single instruction on multiple data even with no aim toward parallelism. These language features are simply implemented for reduction of code. Some examples of this sort of language feature are: the Python map built in function[3].

0-7695-3035-4/07 $25.00 © 2007 IEEE

More equivalent statements to the one presented in this paper add parallel execution as a goal of the statement. These can be found in language extensions to standard programming languages aimed at increasing parallelism in high performance computation. Examples that add loop level parallelism are in high performance Fortran and high performance C (HPF and HPC respectively). The statement in these languages is the FORALL statement[8]. More examples of parallel language extensions and sets of compiler directives include MPI (message passing interface)[7], OpenMP (Open Message Passing)[4]. and ADAPTOR (Automatic DAta Parallelism TranslatOR)[10].

Another approach that is more appropriate if reduction of complexity is the goal is to write a language that inherently supports concurrency. Many such efforts exist such as AKL[5], Fortress[6], Mozart/Oz, Erlang, E, and many more. Other then Java most of these languages are still in development or have yet to have a major following. It is worth mentioning that concurrency is possible in other languages such as C, but it is not considered a concurrent language because the parallelism is not built in.

3. Objective

Allow for parallel execution with as clean and simple code as possible. Standard multi threaded Java code requires modification to several parts of the code. It includes writing an object that implements the runnable interface, overriding the abstract run method, and later invocation of the method. The goal of this study is to reduce this complexity while maintaining the advantages of Java cross platform code.

Furthermore I intend to demonstrate that this syntax is complete through mapping the code to standard Java and running the code. Furthermore I will verify if multi-threaded Java code runs on multiple processing cores. In addition the parallel code will be tested on fine grained and course grained computation. I suspect that there will be a large speed up for course grained computation, but the fine grained computation would be worse off due to the overhead of generating threads. In addition I hope to further clarify what conditions are necessary for loop level parallelization. If such conditions are well known perhaps the compiler could detect them in standard for loops and generate parallel code where applicable. Though automatically detecting the extent of a computation remains a problem.

4. Method

The idea will be tested by an algorithm that maps the "all statement" embedded in Java code to standard multi threaded Java code. The ability to map one type of code to the other shows that the code has equivalent meaning.

The mapping results in two versions of the multi threaded code. The nonstandard version with the all statement, and the standard multi-threaded Java code. The reduction of complexity benefit will be measured by comparing the number of lines of code in the two versions. In addition the number of different places the code introduces modifications will be considered.

```
// Given a statement:
// all(Object X : Collection)
//     Body

file inFile
file outFile

copy inFile to outfile

if("all" ∈ inFile):
   add "import java.lang.Thread" to outFile
end

for(allStatement a in inFile):
   //Modify outFile as follows
   add an inner class extending Thread
   add a member variable representing X
   add a constructor that initializes X
   copy "Body" from inFile
   add "Body" to the run() method of outFile
   replace the "all" with a "for" in outFile
   in outFile initiate threads in for loop
   in outFile wait for threads to finish
end
```

Fig. 3. Mapping Algorithm Pseudocode

5. Results

A program that increments all the values in a vector was written as a simple example of the statement. This simple case with only one line of code in the body of the statement allows for analysis of the complexity of the two versions. The version with the all statement to

add loop level parallelism and one with standard parallel Java code.

The version with the new statement is completely compatible with the Java 5 “for each” style statement if one only replaces the “all” with “for.” That said the added complexity is minimal and requires a single point of modification.

The lines of code is two or more; counting at least one for the body and one for the statement.

```
//Java code with loop level parallelism
package forall;
import java.util.*;

public class IncrementTest {
   static Vector v = new Vector(1000);
   public IncrementTest() {
      for(int n=0;n<1000;n++){
         v.add(n);
      }
      all(Object x : v){
         x = ((Integer)x).intValue() + 1;
      }
   }
}
```

Fig. 4. Java Code With "all" Statement

The “standard threaded” version of the code was more complex and is included in appendix B. It required modification to eight or more lines of code (not counting curly braces). The code took five points of modification as follows:

1. Include of the library
2. A new class for the threaded code
3. Implementation of the run() method
4. Invocation of the threads
5. Implementation of code in the constructor to reference an object in the collection

The thread objects were created in an inner class to the original object. This gives the threads access to the host objects instance variables.

For safety sake the threads should take care on writes to such variables. The fact that each thread is passed a reference to an individual element of the vector allows each thread to have an individual object that is safe to modify.

```
//Standard multi-threaded Java code
package forall;
import java.util.*;
import java.lang.Thread;

public class Threaded {
   static Vector v = new Vector(1000);
   public Threaded() {
      for(int n=0;n<1000;n++){
         v.add(n);
      }
      for(Object x : v){
         new TheThread(x).start();
      }
   }
   class TheThread extends Thread{
      Object x;
      TheThread(Object x){
         this.x = x;
      }
      public void run(){
         x = ((Integer)x).intValue() + 1;
      }
   }
}
```

Fig. 5. Multi-Threaded Java Code

In a less naive implementation the modifications would be constrained to the collection object, however, access to the outer object will be maintained to handle any valid block of code that could be in the original block. Special clones of the host (outer) object could be made immutable and passed with the thread to a cluster of remote computers for processing.

6. Conclusion

The auto-threaded statement has marginally less code. Although the reduction in code in this instance was five lines when dealing with large blocks of code the difference will be minimal.

The forall statement however, has been shown to be far less complex in points of modification one vs. five. When dealing with the complexity of the code separate places that need modification are going to quickly burden the designers and programmers of the system. The human mind can only contain so many separate ideas at one point in time. This makes a single point to consider far more desirable.

7. Further Study

Further study need to be done on the performance characteristics of such a statement. Including the possibility of distributed parallel execution.

In addition there might be an even easier way to perform operations in parallel on sets. Some possible examples are the use of closures as in Smalltalk, Ruby, or Groovy where a block of code can be passed to a method. The following two lines is an entire single threaded simple test program using closures in the groovy programming language. Adding an "all" method instead of an "each" method would be to only change needed to multi thread if the all method existed on the Array class.

```
myList = (0..999).toList()
myList.each{ it++ }
```

Fig. 6. Single-Threaded Groovy Code

The map and reduce functions that operate on every element in a collection as found in Python and other scripting languages can also be made to operate in parallel.

In addition an implementation might benefit if after dispatching all the worker threads the parent thread could wait for all the child threads to finish before continuing past the parallel statement.

The question remains if such a specialized structure is needed. Loop level parallelism would definitely benefit a language that can operate on all the new multiprocessing desktop systems, but it may not require something as drastic as a new language statement. Implementation as a standard library would have the same reduction of complexity with only the added requirement of including a library.

References

[1] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick. The Potential of the Cell Processor for Scientific Computing. ACM CF'06, May 2006, Ischia, Italy.

[2] J. De Gelas, SUN's UltraSparc T1 - the Next Generation Server CPUs. http://www.anandtech.com

[3] Python Language Reference http://docs.python.org/lib/built-in-funcs.html

[4] L. Dagum, R. Menon. OpenMP: An Industry Standard API for Shared Memory Programming. IEEE Computational Science & Engineering, January 1998

[5] S Janson, S Haridi. An Introduction to AKL A Multi-Paradigm Programming Language. Swedish Institute of Computer Science , December 14, 1993

[6] E Allen, D Chase, J Hallett , V Luchangco ,JW Maessen, S Ryu, G. Steele, S Tobin- Hochstadt et. al. The Fortress Language Specification. Sun Microsystems, Inc. September 19, 2006

[7] R. Graham, G. Shipman, B. Barrett, R. Castain, G. Bosilca, A. Lumsdaine. Open MPI: A High-Performance, Heterogeneous MPI. Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks. Barcelona, Spain September, 2006

[8] High Performance Fortran http://hpff.rice.edu/

[9] R. Duncan. A Survey of Parallel Computer Architectures. IEEE Computer. February 1990, pp. 5-16.

[10] S. Benkner, T. Brandes. Efficient Parallel Programming on Scalable Shared Memory Systems with High Performance Fortran. Concurrency and Computation: Practice and Experience, John Wiley & Sons Ltd. Special Issue of HPF Users Group Meeting 2000, Tokyo