Programming Models

Lecture #9: Tuesday, 29 April 2003
Lecturer: Paul W. Lee, Rebecca Schultz, Varun S. Malhotra
Scribe: Sorav Bansal, Brad Schumitsch

1 Introduction

The need for a simple, correct and general programming model cannot be over-emphasized. A good programming model is instrumental in mapping the parallelism in the application to the support for parallelism in the hardware. There are two perspectives to keep in mind when designing a programming model - the perspective of the programmer and the perspective of the compiler-writer.

From the point of view of the programmer

1. The interface should be easy and intuitive
2. The model should make it easy for the programmer to identify parallelism and indicate it. The complexity should be hidden by the compiler.
3. The programming model should be architecture-independent
4. The programmer should not need to worry about details like *locality-of-usage*.

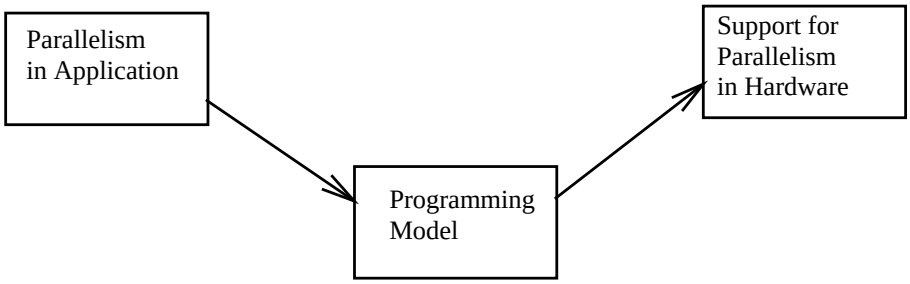
From the point of view of the Compiler-Writer
The program structure should convey some parallelism that the compiler can exploit. This parallelism could be either ILP, TLP or ILP. Present-day programming languages like C hide everything but ILP.

The papers we discussed ([1], [2], [3]) present three different approaches to programming model design.

2 Paper Summaries

2.1 Container-Centric Approach For Parallelizing Applications ([1])

Parallelizing compilers have been successful in array based numerical applications, but the success has been limited in other areas. To effectively parallelize applications for



these other areas, the compiler should be aware of the attributes of a program in terms of its aggregate data structures.

This paper presents the concept of containers, any general-purpose aggregate data type such as matrices, lists, tables, graphs and I/O streams. The use of containers is very similar in most programs independent of implementation, and can be abstracted for use in data analysis and parallelization. The paper uses two of the most common containers to demonstrate its ideas. Linear containers are containers where its elements are accessed in a linear manner, such as lists, stacks and queues. Content-addressable container elements are accessed through keys. Examples of this are tables, sets, and maps.

Containers are central to the flow of data, and are a primary source of data parallelism, but conventional compilers are not aware of the structure of containers and thus cannot take advantage of the information at this level. In order to describe concrete containers to the compiler, the paper introduces abstract containers and methods. The compiler takes as input, in addition to the source code, the container description in terms of abstract containers, to perform container specific analysis and transformation to parallelize the program.

The paper discusses container-based transformation techniques such as loop parallelization, container privatization, and exploiting associativity. All of these transformations are analogous to similar transformations on arrays. Dependency test methods are also explored. Range checks are used for linear containers, and commutativity analysis is used for content-addressable containers. Disjoint keys result in commutativity, but overlapping keys are very difficult to analyze. The pattern search-otherwise insert is suggested as a possible commutative case.

A number of Java, C++, and C applications were studied to identify possible transformation techniques. The applications were manually parallelized using the above-mentioned techniques, and large parts of the applications could be parallelized. The paper unfortunately does not give any results to indicate how effective these transformations were in terms of performance improvement.

The paper proposes using the information at the data structure level to parallelize programs. The widespread use of containers such as C++ STL makes this approach very attractive as a way to attain higher levels of parallelization without burdening the programmer. The notion of abstract containers is useful as a specification language for

containers, and provides a starting point in using this approach in automatic parallelization. The information from containers could be useful in other ways as well. It can be used for more efficient data placement or prefetching, for example.

2.2 Coarse Grain Parallel Programming in Jade ([2])

Upcoming architectures have different support for exploiting different forms of parallelism (ILP, DLP, TLP) in the applications. There is need for good programming models that can map the parallelism inherent in the programs efficiently onto these architectures to leverage maximum benefits. Efforts in this area have been directed at (1) coming up with restrictive programming models which cater to a subset of applications and mapping them well onto the underlying architectures, and (2) new programming models wherein programmer gives information to the compiler and the underlying runtime system implicitely or explicitely helping them to exploit parallelism. This paper introduces one such scheme wherein programmer provides information to the compiler explicitely by specifying the data usage patterns in the program.

Traditionally programmers have been writting programs for parallel machines using explicitely parallel languages. However, programmers using one of the explicit parallel languages have to directly implement program's global concurrency using low constructs to create parallel tasks and explicit connections between these parallel tasks. Although, explicit parallel languages give programmer maximum control over the parallel execution which can be exploited to generate extremely efficient parallel programs, but it puts the burden on the programmers to manage many of the low level aspects associated with mapping of computation onto the parallel machine. Another problem with these explicitely parallel languages is that it is difficult to port a program written for one machine to another machine with a substantially different programming interface.

This paper presents Jade, a language which has provision for expressing coarse grain parallelism while sticking to the sequential programming paradigm. Jade is data-oriented language for parallelizing programs written in a serial, imperative programming language. Programmer augments those sections of code to be parallelized (called tasks in Jade terminology) with the data usage information. Jade compiler and the run time system uses this local data usage information to relax program's sequential order. Thus, it is the Jade implementation, not the programmer, which extracts and enforces the global concurrency pattern implicit in the program's data usage.

Jade tasks interact through accesses to shared objects and programmer specifies task's effects by explicitely specifying the visible shared objects that to be read and/or written within the task body in the "specification" section. It is interesting to see that the specification section for a task may contain dynamically resolved variables references and control flow constructs such as conditionals, loops, and function calls which are resolved at runtime by the underlying Jade implementation.

This paper discusses some of the constructs and datatypes in Jade. Few of these constructs give information that helps with synchronisation and concurrency of the tasks.

Although these contructs ensure correctness of the Jade programs but there can be some pathological situations wherein these constructs don't exploit the inherent parallelism in the program. This motivates the need for decoupling "positive" information (which helps in synchronisation) and "negative" information (helps in concurrency) and other constructs that provide only positive or only negative information are discussed. Such constructs are used to reduce the enclosing task's specified set of effects in the hope that this reduction may eliminate conflicts between the enclosing task and tasks occuring later in the sequential execution order. The paper also discusses the Jade datatype, tokens, which are helpful in matching the coarse grain tasks with the coarse pieces of data they access. Each token functions as a synchronisation data abstraction by carrying the dependence for a conceptual unit of data. The Jade implementation presented in this paper is only applicable to machines with single address space but method has been suggested for extending it to machines with separate address spaces. In order to fully utilize Jade's capabilities, the grain size should be huge. If grain size is fine, then the overhead of the Jade's runtime systme may override the benefits we gain from parallelizing the tasks.

The main strengths of this paper is that it sticks to the sequential programming model paradigm without putting the burden of managing explicit concurrency constructs on the programmer. The authors claim that Jade achieves speedup comparable to most efficient parallel programs howver, the experiments discussed in the paper are not exhaustive to support this argument. It is not very apparent from the paper where to draw the task boundaries given a sequential program. Another problem with Jade is that it cannot express algorithms requiring bidirectional task communication since synchronization and data flow unidirectionally from tasks occurring earlier in the sequential execution order to tasks occurring later, in Jade. This is a classic example of a trade-off between "generality" of a system and its "efficiency".

2.3 A Stream Compiler for Communication-Exposed Architectures ([3])

StreamIt is a high level language intended for streaming applications. The language requires has a specific rigid structure, allowing the compiler to simply extract communication and parallelism, and the programmer to ignore architectural details such as the granularity of the topology. The language is intended for the RAW chip-multiprocessor system but the designers state that it could be used for any parallel multiprocessor system. However, StreamIt can only be used to express a program where the filters have fixed input and output rates; it would be unable to do data compression or decompression, for example.

The programming model requires the programmer to express the intended function as a series of filters to be applied to a stream of data. These filters are connected together using a set of commands expressing their structure. Each filter statement is a single input-single output block. Communication between filters is abstracted as an infinite

FIFO. Within this block, the stream data can be accessed via three methods, push, pop and peek, allowing removal, addition and inspection without removal of the stream items. The statements for connected the filter nodes together are pipeline statements, which establish a sequential series of filters, splitjoin statements, which generate parallel streams that diverge from a single stream, and feedbackloop statements, which create a cycle in the graph.

The StreamIt compiler analyzes the stream graph and maps it onto the hardware. It is a three phase process. In the first phase is called the portioning phase. The compiler merges and splits the filter statements until there is a single filter for each processing element in the hardware topology and these filters are balanced to have approximately the same execution time. The second phase of the compiler is the layout phase. In this phase the compiler takes the filters from the portioning phase and chooses which processing element should execute each one. A cost function is used to represent the cost of communication latencies and a simple simulated annealing algorithm is used. The third phase is the communication scheduling phase. In this phase the compiler converts the communication between filters implicit in the stream graph into communication instructions appropriate for the hardware. The compiler must be careful to avoid starvation or deadlock.

The StreamIt language and compiler are appropriate for the RAW architecture for which it was designed, but despite the authors claim that it is useful for other parallel architectures, a number of features reduce the reusability of the design. For one thing the model extracts parallelism from the individual processors during the portioning phase. Once filters have been merged and split into a single filter for each processing node, the filter is executed sequentially on that processing element. There are no opportunities to extract additional parallelism at the elements. Additionally, while the implicit communication model in the stream language maps well to the communication network in RAW, it may not work well on other architectures.

The StreamIt programming language also has the additional weaknesses that it requires static rate streams. Also, sends and receives can not be interleaved, all data must be received by a filter before it can be processed. Despite some weaknesses, the programming model does provide a way of efficiently executing a stream program that fits its requirements on a hardware architecture with which it is compatible.

3 Discussion Notes

After studying three different programming models, the discussion time was spent in identifying the strengths and weaknesses of the three suggested approaches.

3.1 Perspectives

Why have these approaches not been widely adopted?
It has been noticed that, while people have suggested various programming models, most

of them have found it very hard to be widely adopted by other people. Some of the reasons for this are:

1. The processors have been improving at 55% each year. So people do not feel the need to shift to a totally new programming paradigm, when they can get the desired performance by waiting for a faster processor.
2. There are not too many parallel machines in use. Now, parallel machines are becoming a lot more common.
3. It is hard to get a programmer to shift to a new programming model.

What would it take for a programmer to shift to a new Programming approach?

1. It should be as close to the present day languages (say C) as possible.
2. There should be a very good tool/library support, so that it is easy to use for the programmer.
3. If possible, the new programming model should provide backwards-compatibility with existing programming languages.
4. The programming model should be general. It should be applicable to a wide class of applications.
5. The programming model should help in a huge-huge performance gain. A performance gain of 100% is not enough to justify the shift. The gain should be of the order of 10X to get the programmers to shift.

This is from the perspective of a programmer. The new programming model should cater to the needs of the compiler-writer as well. This brings us to the question:

As a compiler writer, what do you want to know?

1. **Control Dependences** - Which will let you identify independent parallel tasks.
2. **Shared/Non-Shared Variables** - Data dependence between the tasks.
3. **Memory Access Pattern** - If possible, the programming model should be able to disambiguate memory accesses.

Which of these issues are addressed by each of the these programming models?

- **Containers**: Containers disambiguates memory accesses for the compiler. There are no random pointers. It tells the compiler about the data access pattern. However, containers do not help at all in identifying task level parallelism.

- **Jade**: The Jade programming model tells the compiler about the data and control dependencies. It dilineates different tasks and explicitly identifies the shared data. However, it tells the compiler nothing about the memory access patterns.

- **StreamIt**: The Streamit model helps identify task-level parallelism (eg. different filters), data dependencies, as well as memory access patterns (eg. FIFOs). Thus, it makes the job of the compiler writer relatively easier. The catch, however, is that the programming model caters to a very limited set of applications. It requires fixed rate streams exhibiting fixed communication patterns. Variable output rates (eg. Data compression) cannot be handled very well by the StreamIt programming model.

- **Brook**: Brook is designed to program the next version of Imagine. It has support for multiple dimension streams to allow for scientific computing. In Brook, function calls to variable rate streams are allowed, thus ameliorating some of the drawbacks of StreamIt. The programming model is not tied to the number of processors in the underlying machine. However, no loop carried dependencies are allowed in a kernel. The kernels need to be data-parallel. In StreamC, the programmer had to do cluster communication herself, while in Brook communication is inferred.

3.2 Shared Memory vs Message Passing Models

The programming models for parallel programming can be broadly categorized into the Shared-Memory Model (SMP) and the Message Passing Model (MPP). The shared memory model works by using fork and join to create and synchronize processes. The main difficulty in the Shared Memory model is fine-grained synchronization of shared memory. The canonical ways to synchronize shared memory is through locks and barriers. Programmers often complain about the difficulty in guarding against race conditions and deadlocks in the SMP model. The Jade programming model also suffers from this lacuna since the programmer needs to explicitly provide information about shared variables.

The StreamIt model falls into the MPP category. The programmer does not need to worry about synchronization problems. However, writing the program in StreamIt may itself be quite tough.

Using containers, the programmer doesn't have to worry about synchronization. Different sections of the container may be executed in parallel, but this is managed by libraries and the programmer need not be concerned.

In general, message passing code is harder to write the first time, but it is easier to debug. Conversely, writing a program with locks and joins is easy to get running at the first pass. However, it is hard to debug and optimize these programs since the locks and joins hide memory transactions.

On a different note, it is interesting to see why the container approach seems promising. It has been noticed, that a small number of data structures are used over and over again in most programs. It turns out there are approximately 10-15 data structures that cover most cases: linked-lists, hash tables, etc. Hence, it makes sense to have these 10 data structures implemented in a library (such as containers) and we can expect performance gains in most cases.

3.3 How well do these models scale?

Jade scales fine only to the thread-level. Since there is not that much thread-level parallelism, it may not scale too well to too many nodes for many applications.

StreamIt scales very well, to as many filters as your application can support. It scales as long as the application data rates scale.

Containers scale well only for data-level parallelism. They cannot scale to task-level parallelism. The scalability of the container approach depends on the size of the data structures. For example, you would need a linked list with thousands of elements for this approach to scale.

3.4 Polymorphism Opportunities

It would be good if the programming model can somehow indicate, which processor should the base architecture mimic to obtain maximum performance.

If your program is in StreamIt, it is easy for the compiler to decide to run your program on an Imagine-like architecture, since most StreamIt programs will have a lot of data-parallelism. In some sense, this is the complier choosing the configuration of the machine. However, in some sense, the decision was taken by the programmer when she chose to write the program in StreamIt. It is trivial to map threads from StreamIt code to processors. The compiler can statically build the flow graph and then assign the nodes to filters in a one-to-one manner. The communication between filters can be aided by configuring parts of the processor as hardware FIFOs.

In the container based approach, some hardware optimizations can be done to make the data structures faster. For example, a hardware FIFO could be used for a queue data structure.

In Jade, the run-time system must keep track of dependencies and it needs to schedule non-blocked tasks to existing processors. Threads that are blocked and switched out of a processor should be returned to the same processor due to cache warming (*aka* affinity

scheduling). The hardware needs to be optimized to perform these runtime operations.

3.5 What else can help the compiler?

What other features can we have in the programming model, which can be useful for the compiler to extract parallelism. Some of the ideas are the following:

- Support for templates for parallel algorithms would be useful. Just like programming languages today support templates for common data structures, templates for common parallel algorithms could be incorporated in the programming languages. For instance, a template to search a tree in parallel could be useful.

- It would be useful if the language specified two types of memory: *shared* and *private*. Clearly, private data could be stored in each processor's private cache, without any need to invoke any complicated cache-coherence mechanism.

- Moreover, if the programming language had notions of transactions built in, it would be easier for the compiler. One could have a section of one's memory which follows transactional semantics. In that case, compiler has support to undo actions without leading to an inconsistent state. This can help the compiler in speculating for task level parallelism (which is very difficult in the current programming models).

4 Conclusion

To conclude, it would be nice to answer the question: What features should a new programming model have? Some of the key requirements are:

- The performance benefit should be substantial

- The model should be independent of the architecture

- Easy-to-use Programming Tools must be provided

- Should be easier to adopt (if possible, the model should be close to the present-day programming languages)

It is hard to answer the question about which programming model will be ultimately adopted. Perhaps, there will be multiple domain-specific solutions.

	Container	Jade	StreamIt
Programmer's viewpoint	Need to identify supported data structures	Need to explicitly manage synchronization	Need to explicitly identify streams and filters
Compiler Writer's viewpoint	Exposes Memory Access Pattern and DLP. Does not expose TLP.	Exposes task-level parallelism and shared structures. No info on memory access patterns	Exposes both task level parallelism and memory access patterns
Generality	Helps for applications with DLP. Almost all applications use supported data structures.	Helps for applications with TLP, any program with TLP can use Jade	Suited to a very limited class of applications
Scalability	Parallelism Limited by the size of data structure.	Scales only to the number of independent threads identified by the programmer	Scales very well as long as the data rates scale. Limited by by number of filters
Polymorphism Opportunity	Allows compiler to optimize hardware for data structures (e.g. Queue = hardware FIFO)	Can optimize hardware to implement runtime support to schedule tasks in parallel.	Preferred configuration is a stream architecture like IMAGINE

References

[1] P. Wu, D Padua. Beyond Arrays: A Container-centric Approach on Parallelization of Real-world Symbolic Applications. Proceedings of the 11th International. Workshop of Language and Compilers for Parallel Computing, NC, August 1998.

[2] Monica S. Lam and Martin C. Rinard. Coarse-Grain Parallel Programming in Jade. In Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, April 1991.

[3] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A.S. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS 2002*, San Jose, CA USA, October 2002.