Qualitative Evaluation Criteria for Parallel Programming Models

Christopher D. Krieger, Andrew Stone, and Michelle Mills Strout

Programming parallel machines has always been difficult, but the new multicore reality has triggered significant research projects such as those part of the DARPA HPCS challenge that put forth programmer productivity as on par with performance. Since programming models are crucial in supporting the programming of parallel machines, this raises the question of how the PLDI community should evaluate programming models along the productivity and performance dimensions.

Meaningful quantitative metrics are the ideal solution. Fortunately, performance metrics are abundant and relatively understood. More problematic is how to evaluate the effect of a programming model on programmer productivity. Existing programmer productivity metrics include things such as SLOC or development time. Other factors have been determined important but are difficult to quantify: readability, maintainability, portability, correspondence with serial code, adaptability of algorithm, etc. These programmer productivity metrics all contribute to the big picture, but have limited meaning and do not necessarily provide a clear direction for future programming model development.

Due to the fact that parallel programming implementation details often obfuscate the original algorithm and make later algorithm modifications and maintenance difficult, we believe that parallel programming models should provide features that enable separation between algorithm and implementation detail specifications. A similar idea was suggested in the Parallel View from Berkeley paper [1], where they suggest programming models provide a separate interface for application programmers and parallel implementors.

To encourage the separation of algorithm and implementation details, we propose programmer control and tangling as two qualitative measures for evaluating programming model constructs. We show example categories within the tradeoff space of programmer control and tangling, and we suggest a methodology for using these qualitative measures for comparing programming models.

We argue that there is a rough correlation between tangling and reduced programmer productivity and between programmer control and performance. We use the term *tangling* to indicate that the implementation details are exposed in the algorithm code in an intermingled manner. In general, it is important that tangling be kept to a minimum while still providing the programmer control over the parallelization and optimization details for performance tuning purposes (i.e., programmer control).

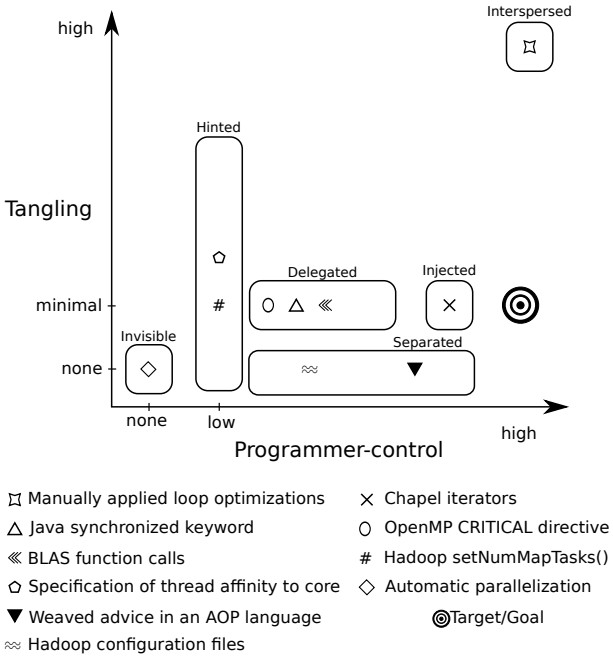

Figure 1: Relative classification of features and their programmer control and tangling impacts.

The Goto BLAS is an example where programmer control has led to strong performance, but at the cost of having loop transformations such as multiple levels of tiling, register tiling, loop fusion, etc. intermingled with algorithm code. On the other hand, no tangling is problematic from the program understanding and debugging point of view. For example, aspect-oriented programs can be difficult to debug when the programmer has no explicit signal in the code as to where aspect advice will be woven in.

We observe that parallel programming models that allow a high degree of control over parallelization details tend to deliver the highest performance. The Goto BLAS example illustrates that the significant control provided by the C programming language enables very good performance. Ideally, programmer control of implementation details should be available, orthogonal to the algorithm specification, and should not be required.

Figure 1 shows where some example programming language features fall within the tangling and programmer control space. The bullseye in the figure shows that the ideal is to have minimal tangling, so that it is clear where implementation details are relevant, while providing maximal programmer control if needed.

We observe current programming model constructs

falling into six categories: interspersed, injected, invisible, hinted, delegated, and separated. The *interspersed* category represents the common but worst case in terms of tangling where implementation details are completely tangled with algorithm code. We place manual application of loop transformations in this category.

The remaining categories include constructs that alleviate some tangling by separating implementation details from algorithm code. For example, Chapel [2] separates the scheduling of an iteration space from algorithm code by enabling programmers to encode loop-iteration logic within a construct called an iterator. The referencing of an iterator in loops (e.g., `for index in iterator`) does tangle algorithm code in a minimal way. This minimal tangling of code by referencing a feature at the point of its applicability is characteristic of the *injected* and *delegated* categories. Calls to injected features in some way invoke programmer written or modifiable code contained elsewhere, whereas calls to delegated features transfer control to code not visible to or modifiable by the programmer. Chapel's iterator construct, as well as calls to modifiable functions, are examples of *injected* features.

The *delegated* category includes calls to unmodifiable functions, for example contained in a system or vendor library, where programmers do not have full access to some of the implementation details, but some degree of programmer control is provided through parameterization. The `synchronized` keyword in Java, which enables the programmer to specify atomic sections and their corresponding locks, is a delegated feature. OpenMP's `critical` directive is similar, but is not passed a lock. Thus, programmers have less control over synchronization with OpenMP's critical directive.

Hinted features provide less programmer control than delegated features as they merely suggest how to address some implementation detail. The programming model is free to ignore these suggestions. Examples include C's `register` keyword, OpenMP's specification of thread-to-core affinity, and Hadoop's `setNumMapTasks()`.

Separated features are similar to injected features except that their invocation points are not explicitly indicated in the program. Separated features contain an additional specification defining where in the algorithm the feature is relevant. As an example, aspect oriented programming (AOP) is a programming paradigm designed for the specification of separated features and where to apply them. Hadoop configuration files are another example of a separated feature. With all separated features, programmer control in is limited by the expressibility of "join" points.

Invisible features enable a complete separation of an implementation detail from the algorithm, but also remove all control over how implementation details are addressed. In a perfect world, the programmer would only have to focus on the algorithm and not have to deal with parallel implementation details at all. This is the vision of approaches like automatic parallelization. Unfortunately some amount of programmer control is often needed to achieve performance in the general setting.

To categorize program model constructs along the programmer control and tangling qualitative measures, we propose a methodology where parallel patterns seen in a large number of parallel apps are expressed within the programming model and then evaluated. In [3], Keutzer and Mattson propose the *Our Pattern Language* (OPL). The implementation strategy patterns that are part of the OPL include various distributed data structures and the parallel loop, SPMD, and master-worker patterns. The realization of these patterns with a given set of programming constructs should guide the qualitative evaluation.

When realizing the implementation strategy patterns in the OPL a number of details must be considered. For example, with the data structure patterns, implementation details include the data distribution; with the loop-parallel pattern, details include how iterations are mapped to threads, how threads are mapped to processors, and the order of iterations on a single thread; with the SPMD pattern, the distribution of data and computation to processes and the number of processes must be determined; with the Master-Worker pattern, details include the distribution of the queue, the number of workers, and the level of precision in the queue semantics. The specification of these implementation details can be used to evaluate how much programmer control a programming model/feature provides for specific implementation details and how much tangling the specification of an implementation detail requires.

In conclusion, we propose using the programmer control and tangling qualitative evaluation criteria within the context of expressing implementation details for parallel patterns as a way of comparing programming model features. These qualitative measures should be presented along with existing quantifiable metrics such as performance, SLOC, and programmer effort. We recommend that program features instead of entire programming models be the focus of any evaluations using these criteria. An important consequence of using the programmer control and tangling qualitative measures is that they will encourage the development of more programming model features and constructs that support the separation of algorithm and implementation specification.

The authors thank Brad Chamberlain, Sudipto Ghosh, and the ParaPLOP 2010 participants especially Ralph Johnson, Kurt Keutzer, Tim Mattson and Beverly Sanders for their helpful comments and suggestions. This work was supported by a DOE Early Career Award.

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[2] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[3] K. Keutzer and T. Mattson. Our Pattern Language (OPL): A design pattern language for engineering (parallel) software. In *ParaPLoP Workshop on Parallel Programming Patterns*, June 2009.