The Habanero Multicore Software Research Project

Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Jisheng Zhao, Vivek Sarkar

Computer Science Department, Rice University, 6100 Main Street, Houston, TX 77005, USA

{rajbarik, zoran, vc8, cs20, yguo, dmp, raghav, shirako, sagnak, yanyh, jz10, vsarkar}@cs.rice.edu

Abstract

Multiple programming models are emerging to address an increased need for dynamic task parallelism in multicore shared-memory multiprocessors. This poster describes the main components of Rice University's Habanero Multicore Software Research Project, which proposes a new approach to multicore software enablement based on a two-level programming model consisting of a higher-level coordination language for domain experts and a lower-level parallel language for programming experts.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Languages, Performance

1. Introduction

The Habanero project [14] at Rice University was initiated in Fall 2007 to address the multicore software challenge by developing new programming technologies — languages, compilers, managed runtimes, concurrency libraries, and tools — that support portable parallel abstractions for future multicore hardware with high productivity and high performance. Our goal is to create a software platform that allows application developers to reuse their investment across multiple generations of homogeneous and heterogeneous multicore hardware. A highly desirable solution to the multicore software productivity problem is to introduce high-level declarative programming models that are accessible to developers who are experts in different domains but lack deep experience with imperative parallel programming, while still enabling programming experts to make tuning and deployment decisions for the application. To that end, we propose a *two-level programming model* consisting of a *higher-level coordination language for domain experts* and a *lower-level parallel language for programming experts*. We use the *Concurrent Collections* (CnC) programming model [9, 11, 4, 3] as the foundation for the higher-level coordination language in the Habanero project, and the *Habanero Java* (HJ) language as the foundation for the lower-level parallel language[1].

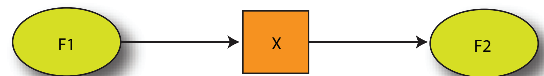
This approach enables an important separation of concerns between *domain experts* who specify the inherent semantics of the computation at the CnC level and *programming experts* who make decisions on how the parallelism should be exploited on a given multicore platform. The programming expert is not necessarily a human — it could also be a static compiler, dynamic runtime, or an auto-tuning component, for example. To tie the levels together, we use an integrated runtime system that supports co-scheduling of CnC steps and HJ tasks (activities). An initial release of our two-level execution model is available at [17].

[1] A Habanero-C language is also in development for C programmers.

Copyright is held by the author/owner(s).

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-768-4/09/10.

Data Dependence



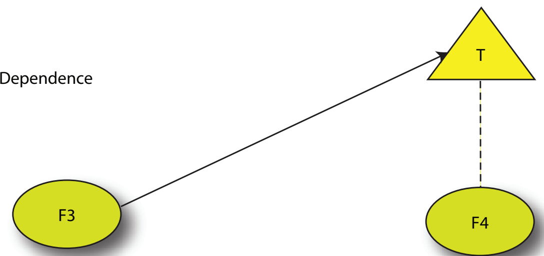


Figure 1. Data and Control dependences in a CnC program

2. Concurrent Collections Programming Model

The three constructs in the Concurrent Collections (CnC) model[2] are *computation steps*, *data items*, and *control tags*. Statically, each of these constructs is a *collection* representing a set of dynamic *instances*. Step, item and tag[3] instances are the units for scheduling parallel computations, communicating and synchronizing data accesses, and creating new step instances, respectively.

The program is represented as a graph. The computation step, data item and control tag collections are represented as circles, boxes and triangles respectively (see Figure 1). We represent the graph in textual form using `( )` for computation steps, `[ ]` for data items and `< >` for control tags.

The edges in the graph specify the partial ordering constraints required by the semantics. One type of ordering constraint arises from a *data dependence* [1]. For example, in the top of Figure 1, an instance of step `(F1)` produces an instance of item `[X]` which is consumed by an instance of step `(F2)`. Clearly the producing step instance must execute before the consuming step instance.

[2] The CnC model builds on past work on TStreams [10].

[3] Any hash-able value can be used as a tag.

Another type of ordering constraint arises from a *control dependence* [7], where one computation step determines if another computation step will execute. In the bottom of Figure 1, an instance of step `(F3)` produces an instance of control tag `<T>` which in turn leads to the creation of an instance of step `(F4)`. The producing step instance must execute before the newly created step instance.

A step instance may consume and produce data items in multiple item collections, and produce control tags in multiple tag collections thereby leading to more general graph structures than the simple examples in Figure 1. Tags play a key role in distinguishing instances of data items and computation steps. For item collections, the tag is akin to a primary key used to access an item instance; the *single assignment rule* ensures that at most one item is produced in an item collection with a given tag. For tag collections, the tag is akin to an iteration tuple or calling context that uniquely identifies a step instance; again, the single assignment rule ensures that at most one step is produced with a given tag.

There are a number of properties of the CnC model that make it attractive for use by domain experts in our two-level programming model. First, the domain expert does not have to explicitly think about sequential or parallel execution of steps; the ordering constraints among steps are specified exactly by the control and data dependencies in the CnC graph. Second, all CnC programs are data-race-free *i.e.,* the single-assignment rule ensures that each read of a data item will return the same value regardless of the order in which write operations are performed (since there can be at most one write with a given tag value in a given item collection). Third, all CnC programs are deterministic *i.e.,* all executions that read the same input from the environment will result in the same output, regardless of the order in which step instances are executed.

3. Habanero Programming Language Constructs

The Habanero Java (HJ) language under development at Rice University [14] builds on past work with the X10 project at IBM [6], and proposes an execution model for multicore processors that builds on four orthogonal constructs:

1. Lightweight dynamic task creation and termination using "async" and "finish" constructs [8]. The *async [(Place)] [phased(c...)] Stm* statement creates a new child activity that executes statement *Stm*, registered on all the phasers in the phased(...) list. It returns immediately and may only reference final variables in enclosing blocks. The *finish Stm* statement executes *Stm* and waits until all (transitively) spawned asyncs have terminated. The finish statement has a rooted exception model that traps all exceptions thrown by spawn activities and throws an (aggregate) exception if any spawned async terminates abruptly [6].
2. Locality control with task and data distributions using the "place" construct [5]. Places enable co-location of async tasks and data objects.
3. Mutual exclusion and isolation among tasks using the "isolated" construct [2]. The *isolated [(Place List)] Stm* statement executes *Stm* in isolation with respect to the list of places. As advocated in [12], we use the *isolated* keyword instead of *atomic* (as it is named in X10) to make explicit the fact that the construct supports weak isolation rather than strong atomicity. Commutative operations, such as updates to histogram tables and insertions in a shared data structure, are a natural fit for isolated blocks executed by multiple activities.
4. Collective and point-to-point synchronization using the "phasers" construct [15, 16]. The *phaser p = new phaser(mode)* statement allocates a new phaser in a registration mode, which can be signal, wait, signal-wait or single. Activities can use phasers to achieve collective barrier or point-to-point synchronization. Phasers are dynamic (number of activities using a phaser can change at runtime), deadlock-free in absence of explicit wait operations, and lightweight. The *next* statement suspends the activity until all phasers that it is registered with can advance.

Since HJ is based on Java, the use of certain primitives from the Java Concurrency Utilities [13] is also permitted in HJ programs, most notably operations on Java Concurrent Collections such as `ConcurrentHashMap` and on Java Atomic Variables.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.

[2] R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *The Eighteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2009. (To appear).

[3] Z. Budimlić et el. Declarative aspects of memory management in the concurrent collections parallel programming model. In *DAMP 2009: Workshop on Declarative Aspects of Multicore Programming*, January 2009.

[4] Z. Budimlić et al. Multi-core implementations of the concurrent collections programming model. In *CPC '09: 14th International Workshop on Compilers for Parallel Computers*. Springer, January 2009.

[5] S. Chandra et al. Type inference for locality analysis of distributed data structures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 11–22, New York, NY, USA, 2008. ACM.

[6] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[7] J. Ferrante et al. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] Y. Guo et al. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *IPDPS '09: International Parallel and Distributed Processing Symposium (To Appear)*, 2009.

[9] Intel (r) concurrent collections for c/c++. `http://softwarecommunity.intel.com/articles/eng/3862.htm`.

[10] K. Knobe and C. D. Offner. Tstreams: A model of parallel computation (preliminary report). Technical Report HPL-2004-78, HP Labs, 2004.

[11] K. Knobe and V. Sarkar. The concurrent collections parallel programming model - foundations and implementation challenges. PLDI 2009 tutorial. `http://www.cs.virginia.edu/kim/publicity/pldi09tutorials/CnC-tutorial.pdf`.

[12] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[13] T. Peierls et al. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[14] Rice University. *Habanero Multicore Software Research project*.

[15] J. Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

[16] J. Shirako et al. Phaser Accumulators: a New Reduction Construct for Dynamic Parallelism. In *23rd IEEE IPDPS*, 2009.

[17] Habanero Team. Download site for initial release of Concurrent Collections (CnC) and Habanero Java (HJ) integrated runtime system. `http://www.cs.rice.edu/~vsarkar/downloads/cnc_distrib_2009_07_21.zip`.