High-level Coordination Specification

Operational semantics for Kanor

Joseph A. Cottam Eric Holk
William E. Byrd
CREST [*]
Indiana University
{jcottam, eholk, webyrd}@indiana.edu

Arun Chauhan
School of Informatics and Computing
Indiana University
achauhan@indiana.edu

Andrew Lumsdaine
CREST
Indiana University
lums@indiana.edu

1. Introduction

Coordinating multiple computational processes is one of the largest challenges of contemporary computing. It is found at all computational scales: single-machines, clusters and clouds. Message passing is a well-proven but complex means of coordinating processes. For example, the MPI-3 draft standard describes four types of process groups, seventeen collective communication patterns, and notes on interactions between these and other MPI constructs. This complexity is partially a result of MPI's reliance on predefined communication patterns and imperative style. Kanor takes a different approach: declaratively specifying communication patterns. Kanor employs a small set of basic concepts (reductions, list-comprehensions and filters) to compactly describe communication and a special-purpose compiler to produce appropriate executable code. This arrangement provides for (1) succinctness, (2) interoperability, (3) expressiveness and (4) performance. Expressing BSP-style programs in MPI requires manipulation of low-level details. Actual communication patterns are either forced into pre-set patterns when using collectives [9], or made latent in either control flow (with send and receive) [2] or data structures (with the proposed topological collectives [10]). In contrast, Kanor expresses the communication patterns in self-contained statements. It is flexible enough to handle irregular communication patterns and succinct enough to show regularity (Figure 1).

This abstract provides the evaluation rules for Kanor *Comm* statements, describes optimizations make possible through the semantics and argues for Kanor's expressiveness. This complements earlier work that focused on interoperability and performance [6]. Though the full semantics are not presented here, they have been developed and reducers based on them have been implemented in PLT Redex, the logic programming language miniKanren and through a pattern matching library.

2. Syntax and Semantics

Kanor can be divided into two parts: a sequential language and communicate statements. The sequential language provides the ability to calculate and store values while communicate statements provide the message contents and routing. The sequential language is a subset of a host language, into which Kanor statements are embedded. To support semantic reasoning, the full Kanor semantics provide a sequential language with mathematical operations, let-bindings for atomic values and a heap-like data store for arrays. Preserving this division between the communicate statements and the sequential langauge provides (1) clear restrictions on what can occur inside of a communicate statement (2) simplifies semantic rules and (3) provides a definition for the interface between the host language and Kanor. As communicate statement evaluation is the crux of Kanor evaluation, we focus on their evaluation.

Comm statements produce interprocess communication. Comm statements have three major parts (Figure 2). The *where* and *filter* determine the context for evaluating the *transfer* statement. Generator/filter results and transfer statements combine to form messages. Figure 3 summarizes Comm statement parts and evaluation rules.

[*] Center for Research in Extreme Scale Technologies

All Gather	`(Comm ((lref A (+ i (* j block))) @ k <<= (get B i) @ j) (where (j (range 0 WORLD)) (k (range 0 WORLD)) (i (range 0 block))))`
Gather	`(Comm ((lref A (+ i (* j block))) @ root <<= (get B i) @ j) (where (j (range 0 WORLD)) (i (range 0 block))))`
Bcast	`(Comm ((lref B i) @ j <<= (get A i) @ root) (where (j (range 0 WORLD)) (i (range 0 (len A)))))`
Scan	`(Comm ((lref A i) @ j <<+ (get B i) @ k) (where (j (range 0 WORLD)) (k (range 0 (+ j 1))) (i (range 0 (len B)))))`

Figure 1. Selection of MPI 2.2 collectives expressed in Kanor.

statement	::=	(`Comm` *transfer where filter*)
transfer	::=	((`lref` *var expr*) @ *expr commop expr* @ *expr*)
where	::=	(`where` *clause**)
filter	::=	(`filter` *predicate**)
clause	::=	(*var list_expr*)
expr	::=	Host language expression returning a single value
predicate	::=	Host language expression returning a boolean
list_expr	::=	Host language expression returning multiple values
commop	::=	Commutative and associative op or assignment

Figure 2. Kanor Comm-statement syntax.

[Copyright notice will appear here once 'preprint' option is removed.]

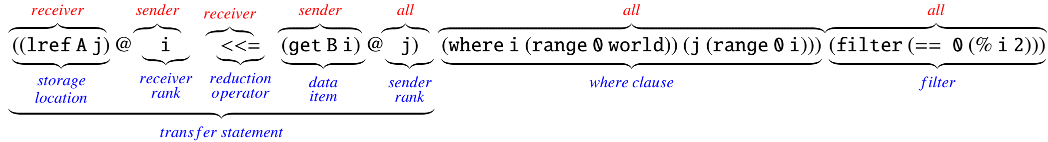

Figure 3. Kanor syntactic elements and evaluation locations.

Evaluation of Comm statements progresses through six phases, only one of which includes cross-process synchronization. The first phases splits each Comm statement into a sending statement (*CommS*) and a receiving statement (*CommR*). In the most general case, the CommS contains all of the parts of the original Comm and the CommR has no arguments. Splitting into Comm enables a number of optimizations and enables more work to be done in parallel across processes. The *where* clauses of the CommS are used to produce partial environments in the second phase. where clauses are expanded using a ConcatMap-based list comprehension technique [7] that results in a list of partial environments where each partial environment has exactly one binding for each variable in the where clause. Partial environments are selected for further processing if the filter clause returns "true" when evaluated with it.

The third phase evaluates the sender rank for each partial environment. Only those partial environments that produce the current process as the sender are used in phase four. The fourth phase produces the actual messages to be sent. Messages are produces by evaluating the *data item* and *receiver rank* expressions in the same way that sender rank was evaluated. (This evaluation is delayed until after sender ranks are evaluated to prevent errors in generating messages that will never be sent.) The *storage location*, *reduction operation* and data item are all combined to form a *remote statement* to be executed on the receiver. The combination method depends on the mathematical properties of the reduction operator used, so the combination styles are limited. Each message is formed from a triple that includes a partial environment with the corresponding receiver and remote statement. The remote statement and partial environment logically form a dynamically scoped closure, which are executed on the receiving process. All free-variables not bound in the partial environment are resolved in the context of the execution (i.e., on the receiver).

Communication occurs in the fifth phase. When **all** processes have finished create messages, they transfer their messages to the queues of the receiving processes. Communication is complete when no process has an undelivered message. Detecting that all messages have been delivered can be achieved through a variety of protocols [3]. *CommR* evaluation is the sixth phase, in which each processes evaluates each received message.

3. Conclusions

The Kanor semantics provide a number of useful properties for Kanor programs. Principally, they provide a clear basis for optimizations. As noted, the MPI-2.2 collectives can be encoded in Kanor. Therefore, when a collective is encoded, the Kanor compiler can identify it and substitute a call to an available MPI implementation. A significant difficulty is that the MPI-2.2 collectives do not have *unique* encodings in Kanor. However, the full Kanor semantics have been used to implement a Kanor reducer in a logic-programming language which can be used to enumerate many encodings of the reductions and thereby improve the possibility that any particular encoding will be known to the compiler.

Identifying and exploiting information known across processes is also possible with the Kanor semantics. When the compiler determines that the where and filter clauses are based on values shared by both sender and receiver, the sender and receiver share *corresponding* knowledge [4]. In such cases, the processes do not need to wait to send until all processes are ready to send. The evaluation of CommR is modified to determine and process the expected messages. This eliminates global synchronization associated with message sending, and related impacts of synchronization overhead and process skew. Global knowledge can be similarly treated.

The full semantic rules also provide a foundation for ongoing work. We are currently to working to delineate when Kanor is deterministic, when it is deadlock free and its unique error conditions. We are exploring automatic communication/computation overlap and message coalescing and other shared-knowledge optimizations. Semantic extensions are being explored to expand reductions beyond associative/commutative operators, safely overlap multiple communicate blocks, and interface BSP with other parallel programming models (such as GPGPU [5, 8] or task-stealing [1]).

Kanor is an effective tool for expressing communication patterns. It enables direct expression of complex communication patterns, including but not limited to those included as standard patterns in MPI. Because of its well-defined semantics, automatic tools for optimization at many different levels are possible.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995. ISSN 0362-1340.

[2] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, Jan. 2004. ISSN 0164-0925.

[3] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. In *Principles and Practice of Parallel Programming*, pages 159–168, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3.

[4] T. Hoefler, J. Willcock, A. Chauhan, and A. Lumsdaine. The Case for Collective Pattern Specification. In *1st ACM Workshop on Advances in Message Passing (AMP'10)*, Jun. 2010.

[5] E. Holk, W. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Declarative parallel programming for GPUs. In *Proceedings of the International Conference on Parallel Computing (ParCo)*, Ghent, Belgium, 09/2011 2011.

[6] E. Holk, W. E. Byrd, J. Willcock, T. Hoefler, A. Chauhan, and A. Lumsdaine. Kanor – A Declarative Language for Explicit Communication. In *Thirteenth International Symposium on Practical Aspects of Declarative Languages (PADL'11)*, Austin, Texas, Jan. 2011.

[7] S. P. Jones and P. Wadler. Comprehensive comprehensions. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 61–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5.

[8] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1*. September 2010.

[9] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 2.2*. September 2009.

[10] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3 (Draft)*. August 2012.