Kanor: an EDSL for Declarative Communication

Nilesh Mahajan
Indiana University
nnmahaja@cs.indiana.edu *

1. INTRODUCTION

Writing efficient parallel programs continues to be a challenge for the programming community. The Message Passing Interface (MPI) enjoys popularity in the high performance system community for writing large-scale parallel programs. MPI programs are coded using the partitioned address space model in which processes communicate by sending explicit messages. Different functions are provided to accommodate for various communication patterns. Furthermore, asynchronous versions of these calls are provided to enable optimizations. This leads to a large and complicated interface requiring considerable programming expertise.

Kanor takes a different approach. It is a domain-specific language (DSL) that allows programmers to specify communication patterns *declaratively*, at a high level, in Bulk Synchronous Parallel (BSP) style [3]. The semantics of the language are carefully defined to guarantee correctness properties, such as deadlock freedom and determinism, while allowing efficient execution. The language is highly *expressive*, able to succinctly describe all the existing MPI collective operations, and allowing users to create their own custom collectives that could be detected and optimized. The BSP style of Kanor also makes it amenable to source-level optimizations that are well understood [1], including those that exploit shared memory for efficient intra-node communication [2].

2. Kanor SYNTAX

Fig. 1 shows the BNF grammar for Kanor. A Kanor communication statement (CommS) updates memory location (Dst) on receiver process, with value (Src) from sender process. The operation bracketed by << can be any binary functor, which is applied to the source and the destination to update the destination. The operation may be omitted, in which case the operation defaults to identity operation. If specified the operation is assumed to be associative and commutative. The senders and receivers are determined by evaluating conditions (Cond) on all processes. The communication statement uses the topology object (TopologyObject) to send, receive messages. Finally, optional hints can be provided specifying the knowledge and invariance of the communication statement.

3. Kanor SEMANTICS

*advisors: Arun Chauhan, Indiana University and Google Inc. and Andrew Lumsdaine, Indiana University

```
CommS    ::= Dst [≪ Op] ≪ Src | Cond [With]
Dst      ::= Lval _at_ ProcIdExpr
Src      ::= Rval _at_ ProcIdExpr
Cond     ::= Clause [& Clause]*
Clause   ::= ForEach | If | Let | TopoSpec | Hints
ForEach  ::= _for_each(PVar, ProcSet) |
         ::= _for_each(PVar, Beg, End)
If       ::= _if ( Cond )
Let      ::= _let ( PVar, Val )
With     ::= _with Hints [& TopologyObject] |
         ::= _with TopologyObject
Hints    ::= Hint [& Hint]*
Hint     ::= GLOBAL | CORRESPONDING |
         ::= SENDER | INVARIANT
```

Figure 1: Formal syntax of Kanor. A communication statement expressed in Kanor is represented by the non-terminal, *CommS*.

Kanor follows the single program multiple data (SPMD) paradigm similar to MPI. A Kanor communication statement can be thought of as parallel assignment of receiver locations by sender values. We say a Kanor program is well-formed iff 1) All processes participating in communication statement C, execute C, 2) All processes participating in communication statements C_1 and C_2, execute C_1 and C_2 in the same order, 3) There are no local errors such as segfaults.

In well-formed Kanor programs, the communication command, treated as parallel assignment, is always successful. All other commands act locally and do not block, hence there is no **deadlock**. Local computations in Kanor are deterministic. A potential source of non-determinism is the communication command. The reduction operator might be non-commutative, e.g. assignment. If such an operator operates with different values on the same memory location, then the result might be non-deterministic. In this case, we make the operator application deterministic by choosing a particular evaluation order. Determinism of local computations along with determinism of communication statements make Kanor **deterministic**. Kanor can not and does not check for well-formedness of programs. This is a task left to the programmer.

While deadlock-freedom and determinism are important properties, they do not tell us enough to identify the communication pattern or to reuse it. Kanor needs to determine how much *knowledge* each process has about the communication pattern and if the pattern can be reused. Com-

munication knowledge describes the extent to which Kanor processes agree on the values of the expressions involved in a communication statement. We say that the knowledge is 1) *Global* when each process knows senders and receivers of all other processes, e.g. well-known MPI collectives. 2) *Corresponding* when a receiver process knows its sender and that sender knows the receiver. The sender and receiver processes may not know about other processes' senders and receivers. Wavefront communication is an example of this case. 3) *Sender* when only the senders know who they are sending to. The receivers do not know who they are receiving from. In this case, a global synchronizing step is needed to complete the communication.

We also identify three core *characteristics* of a communication statement: *length* of the messages, the *contiguity* of the data in memory, and the *process sets* involved in sending and receiving data. We say that the communication is *invariant* if none of the communication characteristics change. Invariance of communication allows us to cache the communication pattern and reuse it in later instances of the same communication statement.

4. DETECTION OF COLLECTIVES

The presence of global or corresponding knowledge allows Kanor to identify well-known collectives.

```
Input: Communication Statement S
Output: Set of Collective Calls C
// G is a directed graph,
// in which vertices are process IDs,
// an edge connects sender to a receiver
G = build from S;
n = number of vertices in vertex set V(G);
if each vertex v in V(G) has degree n then
    if send and receiver buffers contiguous then
        C = {Alltoall};
    else
        C = {AllGather};
    return;
// build rooted collectives
// to be executed independently
foreach v in V(G) with no incoming edges do
    if send and receiver buffers contiguous then
        C = C ∪ {broadcast};
    else
        C = C ∪ {scatter};
```

Algorithm 1: Algorithm to detect MPI collectives.

The pattern detection algorithm 1 works on communication graph G, generated after evaluating the conditions in a communication statement. The algorithm tries to find an all-to-all type of pattern in graph, failing which it returns a set of rooted collectives. Pattern detection as well as caching can be done by runtime in the presence of `GLOBAL` and `INVARIANT` hints and without involving the Clang LibTooling based compiler.

Performance evaluation of the detection and caching system on Cholesky kernel, NAS IS and FT benchmarks is shown in the poster. We found that detection overhead is considerable for small messages sizes and all-to-all patterns. Detection starts to match MPI for larger sizes. Kanor collectives with caching enabled, start to match MPI even for smaller message sizes.

5. OPTIMIZATIONS

Large number of MPI applications use asynchronous point-to-point MPI calls to achieve good performance. Unfortunately, this means that the compiler has to statically match send and receive calls to do further optimizations. This is a very hard problem in the presence of features such as pointers, references, multidimensional arrays, wild-cards (MPI_SOURCE_ANY) etc. BSP nature of Kanor avoids this problem making it easier to do further optimizations.

Following optimizations are possible with the Clang LibTooling based transformations:

- **Async Window Expansion** - In this optimization, asynchronous send and receives are posted as early as possible. Corresponding wait calls are posted just before the data is needed. Asynchronous calls allow us to start communication early and overlap it with computation.
- **Strip Mining** - This optimizations lets us adjust the granularity of communication with computation that produces or uses the communicated data. Loop and communication data strip mining is always legal, hence no complicated data dependence analysis is needed.
- **Buffer Expansion** - Computation depends on communication to finish to make use of the needed data. True dependences cannot be completely eliminated but we can make copies of the communication buffer. The computation uses one copy of the buffer while the communication works with other copy. The copy updated by communication is then used by subsequent computation that depends on it, thus achieving overlap of communication with computation.

The analyses described do not require complex loop transformations but can further benefit from them.

6. CONCLUSION

Declarative nature of Kanor makes it possible to write complex communication patterns including but not limited to MPI collectives. BSP nature of Kanor avoids the problem of statically matching send, receives. This means the Kanor compiler can perform optimizations without complex compiler analyses. Thus Kanor provides and incremental but higher level of abstraction to specify communication in parallel programs.

7. REFERENCES

[1] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. Mpi-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 316–325, New York, NY, USA, 2009. ACM.

[2] F. Jiao, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Partial globalization of partitioned address spaces for zero-copy communication with shared memory. In *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, 2011. DOI: 10.1109/HiPC.2011.6152733.

[3] L. G. Valiant. Bulk-synchronous parallel computers. In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons, 1989.