Kanor

A Declarative Language for Explicit Communication

Eric Holk[1], William E. Byrd[1], Jeremiah Willcock[1], Torsten Hoefler[2], Arun Chauhan[1], and Andrew Lumsdaine[1]

[1] School of Informatics and Computing
Indiana University
Bloomington, IN 47405, U.S.A.
{eholk,webyrd,jewillco,achauhan,lums}@cs.indiana.edu
[2] Blue Waters Directorate
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.
htor@illinois.edu

Abstract. Programmers of high-performance applications face two major implementation options: to use a high-level language which manages communication implicitly or to use a low-level language while specifying communication explicitly. The high-level approach offers safety and convenience, but forces programmers to give up control, making it difficult to hand-tune communications or to estimate communication cost. The low-level approach retains this control, but forces programmers to express communication at a verbose, tedious, and error-prone level of detail.

We advocate a complementary third approach in which the programmer declaratively, but explicitly, specifies the essence of the communication pattern. The programmer lets the implementation handle the details when appropriate, but retains enough control to hand-encode communications when necessary. In this paper we present Kanor, a language for declaratively expressing explicit communication patterns, and demonstrate how Kanor safely, succinctly, and efficiently expresses both point-to-point and collective communications.

1 Introduction

Large parallel computers, and the software that runs on them, are important to many areas of science and engineering. The largest of these computers consist of many separate nodes, connected by a high-performance network. These computers implement a message passing model for parallelism: processes have separate address spaces and communicate through messages. Programming languages and libraries can abstract this model, exposing instead a model with a global address space and implicit communication of data. Thus, programmers face a choice between these two approaches.

The implicit approach to communication is exemplified by languages such as X10 [4], UPC [14], and Co-array Fortran [12]; the *de facto* standard for explicit communication is the Message Passing Interface (MPI) [11]. There is a tradeoff between the implicit and explicit approaches to message passing, however. Implicit approaches are easier to

program, but have more opaque performance characteristics, and thus their performance is harder to predict or tune. Explicit approaches are more difficult to program, requiring communication to be specified at a very fine-grained level of detail and thus leading to more errors, but allow more knowledge and control over a program's behavior and thus its performance.

In this paper we advocate a third, complementary approach in which the programmer uses a high-level declarative language to explicitly specify communication within an otherwise imperative program. This approach allows both programmer control and ease of programming, while allowing programs to be incrementally converted from fully explicit approaches to our declarative language.

Many, if not most, MPI applications are written in a bulk synchronous parallel (BSP) style [15]: each process runs the same program—*Single-Program, Multiple Data* (SPMD)—and alternates between steps of purely local computation and communication. One compute/communicate phase is called a *superstep*; there is a global synchronization at the end of each superstep. Examples of programs that are conveniently expressed in BSP style include iterative solvers, sparse matrix-vector multiplication, and many n-body algorithms.

We have designed a high-level, declarative language, *Kanor*, for specifying collective communication in BSP-style programs. Kanor provides a balance between declarativeness and performance predictability and tunability. We have implemented a prototype compiler for Kanor which infers the types and sizes of the data being sent automatically and generates efficient code. As a result, the programmer can express communication safely, simply, and concisely, while paying little to no abstraction penalty, as shown by a performance evaluation. The declarative, high-level nature of Kanor, combined with its simple parallel assignment semantics, exposes opportunities for future optimizations.

Our paper makes the following contributions:

- A declarative language, Kanor, for explicitly expressing collective communication within BSP-style programs concisely and declaratively (Section 4). Kanor extends C++'s type enforcement to communication patterns, and avoids deadlocks, unintentional race conditions, non-deterministic behavior based on message size, and other semantic pitfalls of MPI.
- A categorization of common communication patterns based on the knowledge available to the communicating processes (Section 3). We then show how Kanor takes advantage of this classification to communicate efficiently (Sections 5 and 6).
- Evaluation rules for Kanor (Section 4.1). The details of these rules are important to the language's properties: even a small change to the evaluation rules can radically change the language's expressiveness.
- A set of core algorithms that can be used to implement Kanor's evaluation scheme efficiently (Section 5).
- A prototype implementation of Kanor, which compiles Kanor expressions into C++ and MPI code (Section 6).
- A performance evaluation of Kanor against MPI, demonstrating that the convenience of Kanor's abstractions imposes minimal abstraction penalty when compared to point-to-point MPI communication (Section 7).

2 Motivation

Despite MPI's utility and popularity, MPI has its shortcomings. Consider this BSP-style MPI communication, in which every processor sends a different value to every processor whose process identifier (or *rank*) is even.

```
r = 0;
if(rank % 2 == 0)
  for(j = 0; j < P; j++)
    MPI_Irecv(&A[j], 1, MPI_INT, j, tag, MPI_COMM_WORLD, &reqs[r++]);
for(i = 0; i < P; i++)
  if(i % 2 == 0)
    MPI_Isend(&B[i], 1, MPI_INT, i, tag, MPI_COMM_WORLD, &reqs[r++]);
MPI_Waitall(r, reqs, MPI_STATUSES_IGNORE);
```

The *MPI_Isend* and *MPI_Irecv* functions perform nonblocking sends and receives, respectively, while *MPI_Waitall* returns once all of the sends and receives have completed.

As this example shows, MPI requires programmers to write a considerable amount of code at an error-prone level of detail to express even very simple communication pattens. MPI does provide functions to concisely perform a fixed set of *collective* communications, such as broadcasts. Collective statements are desirable because they are concise, and can be optimized in system-specific ways to vastly outperform their point-of-point equivalents. Even the simple communication pattern above, however, is awkward to express using these collectives.

The programmer needs the ability to express collective communication succinctly and declaratively, allowing the compiler to infer details such as the type of data being sent. Ideally, the programmer would write the communication above as similarly to:

```
@communicate {
  A[j]@i <<= B[i]@j where i in world, j in world, i % 2 == 0
}
```

The semantics of this idealized language would be based on parallel assignment, relieving the programmer from worries about deadlock and race conditions.

In addition to its verbosity, another problem with MPI is that it defeats C++'s type enforcement. Consider this MPI snippet:

```
double b = 0.0;
float a = 1.0;

if(recv_rank == rank)
  MPI_Recv(&b, 1, MPI_DOUBLE, send_rank, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
if(send_rank == rank)
  MPI_Send(&a, 1, MPI_FLOAT, recv_rank, 0, MPI_COMM_WORLD);
```

This example compiles and runs, but the result is clearly unintended. In some MPI implementations, rank 0 may end with the value *5.26354e−315* for *b*, rather than the desired value of *1.0*. Not only is this result incorrect, this program's behavior is undefined according to the MPI specification, meaning the program may misbehave in subtle and

mysterious ways on different implementations. It is easy to see how this error might occur in a large program—a programmer might initially write a program using ***float***s, then change it to use ***double***s. If the programmer misses a ***float***, the program will likely produce incorrect results.

Instead, the programmer should be able to write something like:

```
double b = 0;
float a = 1.0;

@communicate { b@recv_rank <<= a@send_rank }
```

and let the compiler and runtime ensure the ***float*** value is implicitly converted to ***double***, preserving the intended behavior for compatible data types. When the data types are incompatible, the code should fail to compile, rather than behaving incorrectly at runtime.

Kanor has all the desirable properties described above—indeed, the *@communicate* blocks above are correct, running Kanor code.

3 Exploiting Communication Knowledge

For a declarative language for communication to be efficient, it is essential to exploit all available information about each communication pattern. Kanor's design is informed by a multi-level classification of communications, based on each process's knowledge of the global communication pattern. These patterns are a refinement of those our group identified previously [7].

Global Knowledge Each process can determine the entire communication pattern. This global knowledge may enable tree-based communication with logarithmic rather than linear overhead. An advantage of Kanor is that it allows the compiler to generate tree-based communication as an optimization, without forcing the programmer to write special-case code for efficiency.

Corresponding Knowledge Each process knows only a subset of the complete communication topology, but has complete knowledge of the communication in which it will be participating.

Sender Knowledge Senders know only the destinations they will send to; receivers do not know which senders they will receive from. The sender knowledge case requires the receiver to perform a termination protocol. The Kanor runtime uses the Nonblocking Barrier eXchange (NBX) protocol [8], which allows receiver processes to receive an unknown amount of data with minimal overhead.

Receiver Knowledge Receivers know only the senders they will receive from; senders do not know which receivers they will send to. This case requires the receiver to notify the sender processes from which it wishes to receive data. After this notification, communication then becomes equivalent to the corresponding knowledge case.

These categories do not cover all possible applications or communication patterns. For example, some communications might fit a third-party knowledge pattern. However, these categories cover the majority of today's parallel applications [15], and thus

simplify and direct our language design. It is important to note that other types of communication patterns can be transformed into to one of the three categories that Kanor supports by performing additional communication steps.

Communication in Kanor is *sender-driven*. Each @*communicate* block corresponds to either the global knowledge, corresponding knowledge, or sender knowledge case; in all three cases, the sender knows the destination. The receiver knowledge case, which is not as common as the first three cases, and requires an additional communication step, can be expressed in Kanor as two independent @*communicate* blocks. In the current Kanor implementation, the programmer may annotate each @*communicate* block with an optional pragma indicating the global, corresponding, or sender knowledge case; if no pragma is supplied, the compiler assumes the sender knowledge case. A future version of the compiler should be able to infer this pragma in most cases. Incorrectly specifying the hint is erroneous, and can lead to unspecified program behavior.

4 The Kanor Language

Figure 1 contains the grammar for Kanor. The nonterminals *integer*, *variable*, and *expr* represent standard C++ integer literals, identifiers and expressions. The grammar extends C++ by allowing statements to also include the *collective_stmt*.

```
collective_stmt ::= @communicate hint comprehension
hint            ::= ϵ | global | corresponding | sender
remote_ref      ::= expr @ expr
reduction       ::= remote_ref <<= remote_ref
                  | remote_ref << variable << remote_ref
                  | reduction , reduction
comprehension   ::= { reduction where (clause , )* clause }
                  | { reduction }
set_expr        ::= expr | { expr ... expr }
clause          ::= variable in set_expr | expr
```

Fig. 1. Grammar for Kanor

Kanor allows *set comprehensions*, similar to the comprehensions found in Python and Haskell. The comprehension contains *generator* clauses, which bind variables to values in a set, and *filter* clauses, which restrict messages to be sent only when the filters' Boolean expressions evaluate to true. As might be expected, data is sent from the sender's process to the storage location on the receiver process; the complete evaluation rules are given in Section 4.1.

Each top-level @*communicate* block encapsulates a logical communication, which comprises one or more logically independent reductions (described below). The @*communicate* form supports an optional compiler hint, which must be either *global*, *corresponding*, or *sender*. These hints correspond to the first three classes of communication knowledge described in Section 3; the compiler's use of these hints is essential for

good performance (Section 6). It should be possible for the compiler to infer this hint in many cases; when in doubt, the compiler can use the default *sender* hint.

A remote reference, of the form $e_0@e_1$, can appear only within a reduction. The right-hand-side of a remote reference must evaluate to a processor rank; the left-hand-side must evaluate to a data item (on the sender) or a location for data to be stored (on the receiver). The rules for evaluating e_0 and e_1 are critical to the design of Kanor, and are described in detail in Section 4.1.

Our fundamental unit of communication is the generalized reduction construct, of the form:

$$e_0@e_1 << op << e_2@e_3 \texttt{ where } e_4$$

From left-to-right, the reduction comprises four major parts: receiver remote reference, reduction operator, sender remote reference, and qualifier. The variable *op* must evaluate to a reduction function (described below); e_4 is the *qualifier* of the set comprehension. Because the majority of communication statements simply move data, we allow *transfer* statements, which are merely syntactic sugar for reductions using a special operator that performs assignment. A transfer of the form:

$$e_0@e_1 <<= e_2@e_3 \texttt{ where } e_4$$

is equivalent to:

$$e_0@e_1 << \texttt{assign} << e_2@e_3 \texttt{ where } e_4$$

The qualifier portion of a reduction uses comprehension syntax, which expresses the "control structure" of a communication pattern more succinctly than conditionals or loops. Comprehensions allow declarative specification using concepts and notation many programmers are already familiar with. The body of the comprehension contains generator expressions and filter expressions. Generator expressions are of the form

$$x_0, \ldots, x_n \texttt{ in } e$$

where x_0 through x_n are variables to be bound, and e is an expression that must evaluate to a set S. When the comprehension is evaluated, the variables x_0 through x_n are independently assigned values from the set S, in effect forming a Cartesian product. A filter expression is an arbitrary Boolean expression, which may reference variables bound in any generators that appear before it. Each filter expression is evaluated once for each generator assignment of variables that are in scope. If every filter expression evaluates to true for a given set of variable assignments, the sender and receiver remote references are evaluated with those variable bindings and a message is sent; otherwise, the remote references are not evaluated for those bindings. Details of these evaluation rules are given in Section 4.1.

Once a message is received, the receiver updates the values within the storage location by means of a reduction operator *op*. The operator has the signature:

$$op(e_0 : \textbf{ref } \tau_0,\ e_2 : \tau_1)$$

where e_0 represents the storage address to be updated, and e_2 represents the message's data. The reduction operator is called once per message received. As is explained in

Section 4.1, the operator expression is evaluated, and the resulting operator applied, on the receiving process. Here is a simple reduction operator written in C++, which updates the storage location with the sum of the values received:

```
template<typename T>
void sum(T &left, T right) {
  left += right;
}
```

We assume user-defined reduction operators are both commutative and associative. The behavior of non-commutative or non-associative reduction operators is undefined.

The order of evaluation is unspecified between e_1, e_2, and e_3 on the sender and between e_0 and *op* on the receiver. The operator *op* is call-by-value, and is applied only after all of its arguments have been evaluated.

The dependency chain within a Kanor communication can only be of length one: no read can depend upon another read. Kanor also uses parallel assignment semantics: all reads occur before all writes. This allows us to perform analyses similar to those in static single assignment languages. Parallel assignment is an important part of Kanor's semantics, as it guarantees there are no dependencies within a communication block. Parallel assignment makes it much easier to write programs whose communication patterns contain cycles, such as circular shift. Values destined for the same location are accumulated using the reduction operator. However, it is erroneous to make multiple writes to the same location using Kanor's assignment operator (<<=); the semantics of overlapping assignments is undefined.

4.1 Evaluation Rules

Kanor is a language for *explicit* communication: the programmer specifies the sending and receiving processes, the data to be sent, and where the data should be stored upon receipt. To make specifying this information easy, Kanor's comprehension syntax allows programmers to build sets of variable bindings (that is, environments) using generators and filters. Consider this communication, similar to the first one presented in Section 2, annotated to show information explicitly provided by the programmer:

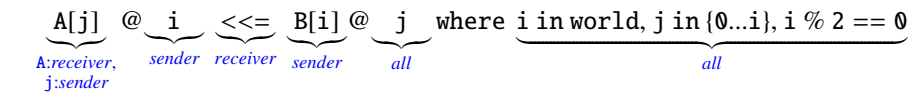
```
A[j] @ i  <<=  B[i] @ j  where i in world, j in {0...i}, i % 2 == 0
```

storage location — *receiver rank* — *reduction operator* — *data* — *sender rank* — *generator* — *generator* — *filter*

It may seem that this information is sufficient to fully specify the communication, given that Kanor's parallel assignment semantics allows the order in which messages are sent to be left unspecified. However, this is not the case; it is also necessary to specify *where* and *in which environment* each of these sub-expressions should be evaluated.

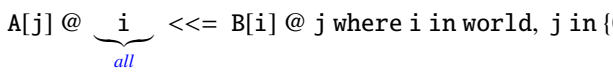
An important note about terminology: when we say that an expression *e* is evaluated on process *p* in some environment *env*, we are referring to the semantics of Kanor, rather than its implementation. As an optimization, the Kanor implementation may use a completely different evaluation strategy, so long as the program behaves *as if* expression *e* were evaluated in *env* on process *p*. (On a related note, side-effecting expressions within a communication block should be avoided, as their behavior is unspecified.)

The programmer could be required to specify explicitly where expressions should be evaluated—for example, indicating whether the storage location expression should be evaluated on the sender or the receiver. Although this is the most general approach, the level of detail required would make even the simplest communication cumbersome to write. Instead, the rules for evaluating Kanor expressions are fixed and implicit:



The sender's rank, along with *where*-bound generators and filters, are evaluated by every process; this is necessary to determine which processes are senders. Furthermore, the *where*-bound clauses are evaluated from left to right—this ordering is necessary since clauses may reference variables introduced in previous clauses. The data expression and the receiver's rank are evaluated on the sender's process. Evaluation of the storage expression is more complicated: the expression is evaluated on the receiver's process, except for *where*-bound variables, which are evaluated on the sender's process and sent if necessary.

The evaluation rules presented above are subtle: the slightest change can radically change the expressiveness of the language. For example, it may seem that the receiver rank should be evaluated on every processor, rather than just on the sender:



The symmetry of this approach is intuitively appealing: the expressions for both sender and receiver ranks use the same evaluation rules. However, using this scheme the programmer cannot directly express the sender knowledge case, in which the ranks of receiver processes are known only to the senders.

Consider another example of changing the evaluation rules. If all subexpressions, other than *op*, are evaluated only on the sender, the evaluation model is equivalent to a remote memory *put* or *accumulate* operation. If all subexpressions are evaluated on the receiver, the model is equivalent to a remote memory *get*. Some collectives, such as *MPI_Alltoallv*, cannot be expressed using only one stage of communication using *put* or *get*, but can be expressed as a single stage in Kanor. (Kanor supports, but is not restricted to, *put*.) For this reason, some operations from other languages and interfaces that superficially resemble Kanor's transfer statement, such as MPI 2's one-sided operations, actually have very different properties because of the different locations at which the subexpressions are evaluated.

5 Core Algorithms

The evaluation semantics described in Section 4.1 can be implemented in a straightforward manner, using the algorithms presented below. We present algorithms for both the *corresponding* and *sender* communication cases. Communication blocks that are marked as *global* or *corresponding* use the Corresponding Communication Algorithm. Sender-knowledge communication blocks use the Sender Communication Algorithm.

The Corresponding Communication Algorithm is given in Figure 2. In the corresponding case, both parties are able to determine which messages will be sent and in what order. This allows the receiver to post nonblocking receives ahead of time and thereby avoid more complicated communication protocols. For each environment generated by the *where* clauses, each process p checks to see if it is sending, receiving, or both. The algorithm then evaluates and applies the reduction operator to the received messages.

```
1  Algorithm: Corresponding Communication Algorithm
2  Input: Receiver rank expression: E_e
           Sender rank expression: E_s
           Data expression: E_d
           Storage location expression: E_l
           List of environments for where clauses: EnvSet
           Local environment: L
           My rank: m
3  receives ← empty list
4  foreach e in EnvSet do
5      e' ← extend_env (L, e)
6      sender ← eval (E_s, e')
7      receiver ← eval (E_r, e')
8      if sender = m then
9          data ← eval (E_d, e')
10         start sending data to receiver
11     if receiver = m then
12         begin receiving data from sender
13         loc ← eval (E_l, e')
14         operator ← eval (E_o, e')
15         receives ← append(receives, ⟨loc, data, operator⟩)
16 wait for all sends and receives to complete
17 foreach ⟨loc, data, operator⟩ in receives do
18     apply operator to ⟨loc, data⟩
```

Fig. 2. Algorithm for the corresponding knowledge case.

Sender-knowledge communication blocks use the algorithm given in Figure 3. This is a slight modification of the NBX algorithm [8]. The algorithm first posts nonblocking sends as before, but the sends in this case require an acknowledgment from the receiver before completing the send. It then enters the NBX termination loop. This loop tests to see if an incoming message is pending, and if so receives the message and stores the result in the output list. It then checks if all pending sends have completed. If so, the algorithm begins a nonblocking barrier which will signal that all processes have finished communicating. Each process continues to receive messages until the barrier is completed.

```
1  Algorithm: Sender Communication Algorithm
2  Input: Receiver rank expression: E_e
           Sender rank expression: E_s
           Data expression: E_d
           Storage location expression: E_l
           List of environments for where clauses: EnvSet
           Local environment: L
           My rank: m
3  foreach e in EnvSet do
4      e' ← extend_env (L, e)
5      sender ← eval (E_s, e')
6      if sender = m then
7          data ← eval (E_d, e')
8          fv ← vars(e) ∩ free_vars(E_l)
9          vals ← lookup(fv, e')
10         send ⟨data, vals⟩ to eval (E_r, e')
11 done ← false
12 barrier_active ← false
13 while not done do
14     probe for message
15     if message waiting then
16         receive ⟨data, e', sender⟩
17         send acknowledgment to sender
18         e' ← extend_env (L, e')
19         loc ← eval (E_l, e')
20         operator ← eval (E_o, e')
21         apply operator to ⟨loc, data⟩
22     if barrier_active then
23         if barrier is complete then
24             done ← true
25     else
26         if all sends have been acknowledged then
27             start nonblocking barrier
28             barrier_active ← true
```

Fig. 3. Algorithm for the sender knowledge case.

In this case, receivers do not know how many messages they will receive, in which order the messages will arrive, or the environments used on the senders to generate the messages. For this reason, the sender must include the values of *where*-bound variables that are used by the receiver. As a message is received, the reduction operator is applied within the environment included in the message rather than the one available locally.

6 Implementation

Our prototype implementation consists of two parts: a compiler written in Scheme and a runtime library written in C++. The Scheme portion of the compiler converts Kanor expressions into C++. The resulting code relies heavily on the runtime library, which performs most of the work in the communication and reduction operations.

6.1 Compiler

There is a small wrapper script for the compiler that extracts Kanor *@communicate* blocks, converts them into S-expressions, and passes them to the main Kanor compiler, which compiles them into C++ using MPI. The resulting C++ code then replaces the *@communicate* block. The design of the Scheme portion of the compiler is modeled after the nanopass framework [13]. Structuring the compiler into many passes, each of which performs very little work, enables rapid experimentation with a variety of implementation approaches, which is crucial during this early prototype implementation phase.

At a high level, the compiler converts *where* clauses into C++ ***for*** loops or ***if*** statements as appropriate. The innermost body adds transfer expressions to a context implemented by the runtime. For example,

```
@communicate corresponding { a@i <<= b@0 where i in world, (i % 2) == 0 }
```

would compile into something like:

```
{
  corresponding_communicate ctx;
  for(int i = 0; i < world.size(); i++)
    if((i % 2) == 0) { ctx.add_transfer(a, i, b, 0); }
}
```

The other main function performed by the compiler is synthesis of message structures and associated reduction operators. This is necessary for sender-only knowledge *@communicate* blocks in order to handle environments correctly. For example, consider the statement:

```
@communicate sender {
  A[k]@i <<= B[k]@j
  where i in world, j in world, k in {0 ... 10}, k % stride == 0
}
```

Here, receivers cannot predict where to store values they receive, because the location depends on the value of *stride* on the sender processor. Thus, when a sender sends the value of *B[k]*, it must also send the associated value of *k* so the receiver can store it in the correct location. In order to do this, the compiler generates a structure such as:

```
struct send_data {
  int k;
  double B_k;
  send_data(int k, double B_k) : k(k), B_k(B_k) {}
};
```

The compiler also wraps the user-specified reduction operator (assignment in this case) with a new operator that unpacks the message structure, such as:

void *set_array(**double** *A, send_data msg) { A[msg.k] = msg.B_k; }*

Finally, the compiler also generates serialization code for messages, such as those using array slices, that might have variable lengths.

6.2 Runtime

The primary purpose of the runtime library is to implement the various communication protocols. This is facilitated by a context class, as mentioned previously. The context class provides an *add_transfer* method, which indicates that a certain data transfer will take place. The context class then executes the set of transfers. We provide contexts for both corresponding and sender communication protocols, and the compiler selects the correct context based on the user-supplied hint. The context is also responsible for managing any temporary buffer space needed to realize Kanor's semantics.

The corresponding context implements the algorithm in Figure 2. For corresponding communication, the receiver can always tell how many messages it will receive, and therefore can start a receive for each transfer in which it is the receiver. Likewise, the context initiates a send for each transfer where a given processor is sending.

The sender context is somewhat more complicated because the communication protocol is more complex. Since receivers cannot determine the amount of data to expect, we must use the algorithm given in Figure 3. This algorithm handles, with minimal overhead, the case where each processor may receive an unknown amount of data. For each transfer in which a given processor is sending, the sender context initiates a synchronous send—i.e., a send that will not complete unless the message is received and acknowledged. After all sends have been started, the context enters a receive and termination loop. If messages are waiting to be received, the context receives the message and applies the reduction operator. Once all of a process' sends have completed, it starts a nonblocking barrier [5]. The barrier completes only after all processes have received all the data that they will receive.

7 Performance

Programmers using a declarative language for communication can enjoy the benefits discussed in Section 2 while paying little or no *abstraction penalty*. That is, the resulting communication can be as efficient as the MPI equivalent. Furthermore, the declarative approach enables optimizations that can make some communications more efficient than their lower-level equivalents.

To show that our approach is feasible, we have conducted preliminary benchmarks for our unoptimized Kanor implementation. Evidence that Kanor is competitive with point-to-point MPI code can be seen in Figure 4, which shows the time in microseconds to execute the first example from Section 1 as the number of processors increases. The graph shows three communication variants: a Kanor version using the corresponding-knowledge algorithm, a Kanor version using the sender-knowledge algorithm (NBX),

and a point-to-point MPI version. The reported time is the arithmetic mean of four runs, each of 15,000 collective operations. Measurements were performed on Odin, a 128-node InfiniBand cluster (Single Data Rate). Each node is equipped with two 2 GHz Dual Core Opteron 270 CPUs and 4 GiB RAM. In order to emphasize the communication cost, we limited the program to only one task per node. Figure 4 shows that there is minimal overhead as a result of using Kanor.

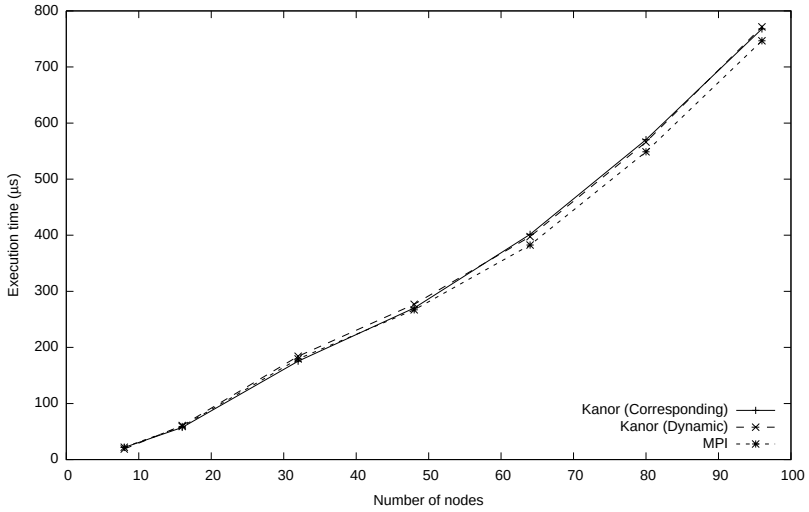


Fig. 4. Time required to execute the first example communication statement from Section 1 as the number of processes increases. This graph compares point-to-point MPI communication with Kanor versions using both the corresponding sender algorithms.

8 Related Work

Partitioned Global Address Space (PGAS) languages, such as UPC [14], Co-array Fortran [12], X10 [4], and Chapel [3], provide programmers with explicit control over data placement, but still use shared-memory-like semantics with implicit communication. They distinguish between references to local vs. remote memory, however; the earlier High-Performance Fortran (HPF) language was similar to PGAS, but without this explicit distinction [9]. These languages differ from Kanor in several ways: they provide a global address space, they do not allow (or expect) programmers to specify communication explicitly, they do not segregate communication from computation, and they do not provide collective semantics for general communications. Kanor, on the other hand, exposes a distributed address space without direct access to remote memory; communication operations must be specified explicitly, rather than implicitly through a memory consistency model. Kanor also has separate communication blocks, separated from an application's computation. These communication operations are collective, matching the BSP model often used in message-passing programs.

Kanor *@communicate* blocks are similar to *sparse collective operations* as proposed for the upcoming MPI-3 standard [6]. The main difference is that Kanor allows a simpler high-level specification of the patterns while retaining all optimization possibilities.

Erlang is a declarative programming language designed for developing highly scalable, reliable concurrent applications [1]. Erlang supports dynamic creation and destruction of processes. Although Erlang is a declarative, functional programming language, its message passing abstractions resemble those in imperative languages since programmers write individual sends and receives. Unlike Kanor, Erlang does not provide or encourage collective communication.

Eden [10] is a high-level declarative language for parallel programming. Their approach is to start with a more declarative language (Haskell) and add support for parallelism. In contrast, we are adding declarative features to C++. While performing on par with MPI is an explicit non-goal for Eden, there is nothing fundamental about the design of Kanor that prevents it from achieving performance similar to pure MPI programs. Eden has a much richer processor abstraction than is provided by Kanor.

Data Parallel Haskell (DPH) adds *parallel arrays* to Haskell along with operations on parallel arrays, e.g., `fold`. DPH lacks mechanisms to send messages explicitly, and lacks X10- or Chapel-like constructs to express locality. However, it supports nested parallelism, similar to NESL, which is a nested data-parallel language [2].

XcalableMP's [16] *gmove* construct specifies collective communications as pragmas using concepts similar to those in Kanor. However, a single *gmove* statement cannot perform data reductions and covers only the global knowledge case while other cases require a mix of multiple pragmas and serial code.

9 Conclusion and Future Work

We demonstrated it is both feasible and desirable to use a declarative domain-specific language to express communication patterns explicitly. Programs that use Kanor are shorter, simpler, and safer than their MPI equivalents.

Perhaps the greatest limitation of our approach is that some problems are not naturally expressed in BSP style. Another limitation is that receiver-only knowledge patterns can't be expressed within a single *@communicate* block. Receiver-only patterns are inherently inefficient, however, so this limitation is minor.

Our model is not restricted to message passing over distributed memory. We also hope to explore a shared memory version of Kanor. There are two obvious approaches to integrating shared memory into the Kanor model. The first is to allow multiple threads per Kanor process. The second is to allow multiple Kanor processes within a single address space, but allow them to communicate only through Kanor.

We also plan to add communication optimizations to the Kanor execution engine. Those optimizations are similar to optimizations for sparse collective operations [6].

We intend to continue exploring different evaluation rules, to better understand their effects on the expressiveness of Kanor. We plan to explore one-sided and non-blocking communication to exploit the communication/computation overlap inherent in BSP applications. An interesting problem is to infer the *global*/*corresponding*/*sender* annotations.

We may also allow additional programmer annotations (for example, whether a reduction operator is commutative) to enable additional optimizations.

Acknowledgments This research was supported in part by NSF grant CSR-0834722 and by a grant from the Lilly Foundation. The Odin cluster used for our benchmarks was purchased using NSF grant EIA-0202048.

We thank Pushkar Ratnalikar and Nilesh Mahajan for porting several MPI programs to Kanor, investigating several papers describing related work, and for their comments on the paper. Amr Sabry and Dan Friedman provided helpful comments on earlier drafts. We also thank Nick Edmonds, Josh Hursey, Joseph Cottam, and members of Indiana's Open Systems Lab and PL Wonks groups for many helpful suggestions. We appreciate the insightful comments provided by the anonymous referees.

References

1. Armstrong, J.: The development of Erlang. In: International Conference on Functional Programming. pp. 196–203. ACM, New York, NY, USA (1997)
2. Blelloch, G.: NESL: A nested data-parallel language (version 3.1). Tech. Rep. CMU-CS-95-170, CMU (Jan 1995)
3. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int'l. Journal of High Performance Computing Applications 21(3), 291–312 (Jan 2007)
4. Charles, P., Grothoff, C., Saraswat, V.A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Object-Oriented Programming, Systems, Languages, and Applications. pp. 519–538 (2005)
5. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and performance analysis of non-blocking collective operations for MPI. In: Supercomputing. IEEE/ACM (Nov 2007)
6. Hoefler, T., Traeff, J.L.: Sparse collective operations for MPI. In: International Parallel & Distributed Processing Symposium, HIPS'09 Workshop (May 2009)
7. Hoefler, T., Willcock, J., Chauhan, A., Lumsdaine, A.: The Case for Collective Pattern Specification. In: 1st ACM Workshop on Advances in Message Passing (AMP'10) (Jun 2010)
8. Hoefler, T., Siebert, C., Lumsdaine, A.: Scalable communication protocols for dynamic sparse data exchange. In: Principles and Practice of Parallel Programming. pp. 159–168. ACM, New York, NY, USA (2010)
9. Kennedy, K., Koelbel, C., Zima, H.: The rise and fall of High Performance Fortran: An historical object lesson. In: History of Programming Languages III. pp. 7-1–7-22. ACM, New York, NY, USA (2007)
10. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel functional programming in Eden. Journal of Functional Programming (15), 431–475 (2005)
11. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4th 2009)
12. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. SIGPLAN Fortran Forum 17(2), 1–31 (1998)
13. Sarkar, D., Waddell, O., Dybvig, R.K.: A nanopass infrastructure for compiler education. SIGPLAN Not. 39(9), 201–212 (2004)
14. UPC Consortium: UPC Language Specification, v1.2 (May 2005), `http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf`
15. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM 33(8), 103–111 (1990)
16. XcalableMP Specification Working Group: Application Program Interface Version 1, Draft 0.7. Tech. rep. (November 2009)