Communication Optimizations for Parallel C Programs *

Yingchun Zhu and Laurie J. Hendren
School of Computer Science
McGill University
Montreal, Quebec, CANADA H3A 2A7
{ying,hendren}@cs.mcgill.ca

Abstract

This paper presents algorithms for reducing the communication overhead for parallel C programs that use dynamically-allocated data structures. The framework consists of an analysis phase called *possible-placement analysis*, and a transformation phase called *communication selection*.

The fundamental idea of *possible-placement analysis* is to find all possible points for insertion of remote memory operations. Remote reads are propagated upwards, whereas remote writes are propagated downwards. Based on the results of the possible-placement analysis, the *communication selection* transformation selects the "best" place for inserting the communication, and determines if pipelining or blocking of communication should be performed.

The framework has been implemented in the EARTH-McCAT optimizing/parallelizing C compiler, and experimental results are presented for five pointer-intensive benchmarks running on the EARTH-MANNA distributed-memory parallel architecture. These experiments show that the communication optimization can provide performance improvements of up to 16% over the unoptimized benchmarks.

1 Introduction

In programming for distributed-memory parallel processors, one important aspect is to minimize communication overhead. As distributed-memory processors incur a significant penalty for accessing memory that is remote (not local to the processor), programmers and/or compilers often try to maximize locality. However, there remain many situations where remote accesses cannot be made local. In some cases the programmer implements a naive algorithm that does not fully exploit locality, and in other cases applications have inherent irregularities that require a significant number of remote memory accesses.

The focus of this paper is on reducing the overhead for remote memory accesses. In particular, we focus on reducing the communication overhead for irregular programs that use dynamically-allocated pointer data structures. Our framework includes: (1) code movement to issue remote reads earlier and writes later, (2) code transformations to replace repeated/redundant remote accesses with one access, and (3) transformations to block or pipeline a group of remote requests together. Our framework consists of an analysis phase that determines where it is safe to move communication, and a transformation phase that selects the best location and type of communication primitives.

To test our approach we implemented the techniques in the EARTH-McCAT C compiler [12], and we experimented with a collection of pointer-based benchmarks that were written in EARTH-C, a high-level parallel dialect of C. The main goal of the EARTH-C project is to provide a high-level parallel language that exposes coarse-grain parallelism and data locality to the programmer, but hides all the details of communication and thread generation. Thus, it is the EARTH-McCAT compiler's job to insert the appropriate communication primitives and to generate appropriate threads and synchronization. This paper focuses on how to optimize the placement and type of communication.

In order to present relatively simple and efficient algorithms we take advantage of the structured nature of the McCAT SIMPLE representation [19], and the presence of accurate read/write sets for both stack and heap-directed pointers [8, 11]. However, the general ideas should be applicable to other compilers that support pointer analysis and read/write sets for indirect references. We present experimental measurements for the EARTH-MANNA distributed-memory parallel architecture [13], and compare the performance of benchmarks with and without communication optimization.

The remainder of the paper is organized as follows. Section 2 gives the essential background to the EARTH-C language and the EARTH-McCAT compiler. Section 3 provides some small examples to motivate the communication optimizations, and Section 4 presents the analysis itself. In Section 5 we provide experimental results, and illustrative examples, for our set of benchmark programs. Finally, in Section 6 we discuss some related work, and in Section 7 we give conclusions and some suggestions on further work.

2 Background

The EARTH-McCAT compiler has been designed to accept a high-level parallel C language called EARTH-C, and to produce a low-level threaded-C program that can be executed on the EARTH-MANNA parallel architecture. In this section we provide an overview of the important points about the language and compiler [12].

*This work supported, in part, by NSERC and FCAR.

2.1 The EARTH-C Language

The EARTH-C language has been designed with simple extensions to C. These extensions can be used: to express parallelism via parallel statement sequences and a general type of forall loop; to express concurrent access via shared variables; and to express data locality via data declarations of local pointers. Any sequential C program is a valid EARTH-C program, and the compiler automatically produces a correct low-level threaded program. Usually the programmer uses the EARTH-C constructs to expose coarse-grain parallelism, and to add some information about data locality. The compiler performs analysis to infer additional locality[22], to expose fine-grain parallelism via data dependence analysis, and to reduce communication (the topic of this paper).

Two sample list processing functions, written in EARTH-C, are given in Figure 1. In both cases the functions take a pointer to a list `head`, and a pointer to a node `x`, and return the number of times `x` occurs in the list. Figure 1(a) uses a `forall` loop to indicate that all iterations of the loop body may be performed in parallel. Since a `forall` loop must not have any loop-carried dependences on ordinary variables, we have used the shared variable `count` to accumulate the counts. Shared variables must always be accessed via atomic functions and in this case we have used the built-in functions `writeto`, `addto` and `valueof`. Figure 1(b) presents an alternative solution using recursion. In this example we use a parallel sequence (denoted using `{^ ... ^}`), to indicate that the call to `equal_node` and the recursive call to `count_rec` can be performed in parallel.

```
int count(node *head, node *x)
{ shared int count;
  node *p;

  writeto(&count,0);
  forall(p=head; p != NULL; p=p->next )
    if (equal_node(p,x)@OWNER_OF(p))
      addto(&count,1);

  return(valueof(&count));
}
int equal_node(node local *p, node *q)
{  return(p->value == q->value);
}
```

(a) iterative solution

```
int count_rec(node *head, node *x);
{ node *next;
  int c1, c2;
  if (head != NULL)
    { {^ c1 = equal_node(head,x)@OWNER_OF(x);
         c2 = count_rec( head->next ,x);
      ^}
      return(c1+c2);
    }
  else
    return(0);
}

int equal_node(node *p, node local *q)
{  return( p->value == q->value);
}
```

(b) recursive solution

Figure 1: Example functions written in EARTH-C

As the target architecture for EARTH is a distributed-memory machine, the distinction between local memory accesses and remote memory accesses is very important. Local memory accesses are expressed in the generated lower-level threaded-C program as ordinary C variables that are handled efficiently, and they may be assigned to registers or stored in the local data cache. However, remote memory references must be resolved by calls to the underlying EARTH runtime system. Thus, for remote memory accesses, there is the additional cost of the call to the appropriate EARTH primitive operation, plus the cost of accessing the communication network. In compiling the EARTH-C language the compiler can assume that all direct (non-pointer) references to parameters and locally-scoped variables are local references. In contrast, unless further information is known, the compiler must assume that all indirect (via a pointer) references and all references to global variables are remote. In our program examples we underline all remote references.

Explicit `local` pointer declarations, and locality analysis can be used to provide extra information to the compiler. In Figure 1(b) the call to `equal_node(head,x)` is specified to occur at the `OWNER_OF(x)`. This means that within the body of `equal_node`, the second parameter can be assumed to be a `local` pointer. This locality can be exposed either through explicit local declarations, or automatically via locality analysis [22].

2.2 Memory Model

An important point about the EARTH-C language is that parallel computations expressed via parallel statement sequences or `forall` loops, may **not** interfere except on explicit shared variables. It is the programmer's responsibility to ensure this non-interference for explicit parallel constructs. The compiler ensures this non-interference for all parallelizing transformations.

Shared variables are explicitly declared using the keyword `shared`, and they may be any type (scalars, pointers, arrays or structs). Shared variables must be accessed via atomic operations. The compiler is allowed to reorder shared variable references within a thread, as long as data dependences are maintained. Thus, shared variables are handled with kind of weak consistency model where each thread sees its own writes in an order that obeys data dependences. However, since independent writes can be reordered, other threads may see those writes in a different order from which they appeared in the original program. In EARTH-C programs, shared variables are seldom used for large data structures, and are most often used for shared counters, or for shared structure headers.

Any variable that is not shared is called an *ordinary* variable and most memory accesses in EARTH-C programs are made via ordinary variables. The program given in Figure 1(b) shows one case of parallel computations on large dynamic data structures. Since the subcomputations are only reading the data structure they do not interfere, thus we do not need shared variables. Another typical example of non-interfering parallelism occurs when two or more recursive calls can be done in parallel on independent pieces of a large data structure.

Execution of an EARTH-C program can lead to many concurrent threads. The memory model interacts with concurrency as follows:

- A *local memory reference* in some thread T **must** access memory that is local to the processor executing T, and no other concurrent thread may interfere with this memory location (i.e. if one thread writes to the location, no other concurrent thread can read or write the location). Local memory references are inexpensive and they can be allocated in registers or in a local cache. They arise from references to locally-scoped variables, parameters, and accesses via local pointers.

- A *shared remote memory reference* in some thread T accesses memory that **may** be on another processor. Other threads may concurrently access the same memory location, but all accesses to this location must be done via explicit shared memory operations, which guarantee atomicity. Shared memory references are the most expensive sort of reference, and they are typically used sparingly.

- An *ordinary remote memory reference* in some thread T accesses memory that **may** be on another processor. No other concurrent thread T' may interfere with this memory location (i.e. if one thread writes to the location, no other concurrent thread can read or write the location). Ordinary remote memory references are compiled to low-level EARTH primitives, and are considerably more expensive than local memory references. They arise from accesses via global variables and non-local pointers.

This paper focuses on the communication optimization of ordinary remote memory references that are made via pointers. For example, the remote memory references underlined in Figure 1 would be potential targets for our communication optimization.

2.3 The EARTH-McCAT C Compiler

This paper builds upon the existing EARTH-McCAT C compiler. The overall structure of the compiler is given in Figure 2. The compiler is split into three phases. Phase I contains our standard transformations and analyses. The methods presented in this paper are found in Phase II. We use the results of the analyses from Phase I in order to transform the SIMPLE program representation into a semantically-equivalent program. Phase III takes the transformed SIMPLE program from Phase II, generates threads, and produces the target threaded-C code that can be run on the EARTH-MANNA architecture.

3 Motivating Examples

In this section we present two small program examples that motivate the types of communication optimizations that we consider. These examples also illustrate the various tradeoffs that need to be considered when applying the communication optimizations.

Let us first consider the simple function `distance` given in Figure 3(a). This function takes a pointer to a structure `Point`, and returns the distance of the point from the origin. As no special locality information is given for the parameter `p`, the compiler must assume that each indirect reference via `p` is potentially a remote operation. Each program example shows remote operations underlined, and in Figure 3(a), it is clear

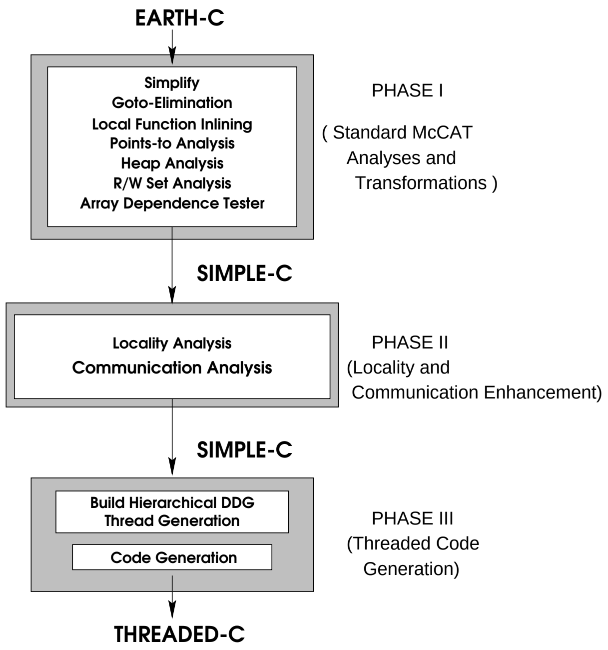

Figure 2: Overall structure of the compiler

that there are four remote reads via `p`. In order to apply the communication optimizations, each program is first simplified so that each statement has at most one remote read or write. Figure 3(b) shows the simplified version of `distance`.

One goal of communication optimization is to move the reads earlier in the program. For example, the four remote reads in Figure 3(b) can be moved to the beginning of the function. Moving remote reads earlier has several advantages. First, because the remote operations are split-phase, by issuing the remote read as early as possible, one allows communication to overlap with the computation following the read. Secondly, moving remote reads can also expose some opportunities for discovering redundant communication. Figure 3(c) shows that two remote reads are all that are necessary. Finally, by moving remote reads together, communication may be pipelined or blocked. Figure 3(d) shows one such transformation for blocking. In this case the entire structure is moved to a local struct (`bcomm1`) by one remote operation (`blkmov`), and then local reads are made with respect to the local struct. The choice of whether to use blocked operations depends on two factors: (1) the relative cost of pipelined scalar reads vs. block reads, and (2) the extra overhead of reading spurious fields of a struct (i.e. the fields required may not be contiguous in memory).

Figure 4 illustrates an example where both remote reads and remote writes are optimized. Note that the two remote reads are moved earlier in the program, whereas the two remote writes are moved later in the program. In the case of remote reads, moving the operations earlier in the program has two advantages. First, it allows overlapping communication with computation, and further it exposes opportunities for pipelined and blocked communication. Thus, it is always a good idea to move remote reads earlier. However, in the case of remote writes there are two conflicting goals. Moving remote writes earlier may improve the overlap of communication and computation, but moving remote writes

```
double distance(Point *p)
{ double dist_p;
  dist_p = sqrt(( p->x  *  p->x ) +
                ( p->y  *  p->y ));
  return(dist_p);
}
```

(a) original C code

```
double distance(Point *p)
{ double dist_p;
  double temp1, temp2, ..., temp7;
  temp1 =  p->x ; temp2 =  p->x ;
  temp3 = temp1 * temp2;
  temp4 =  p->y ; temp5 =  p->y ;
  temp6 = temp4 * temp5;
  temp7 = temp3 + temp6;
  dist_p = sqrt(temp7);
  return(dist_p);
}
```

(b) SIMPLE C code

```
double distance(Point *p)
{ double dist_p;
  double temp3, temp6, temp7;
  double comm1, comm2;
  comm1 =  p->x ;
  comm2 =  p->y ;
  temp3 = comm1 * comm1;
  temp6 = comm2 * comm2;
  temp7 = temp3 + temp6;
  dist_p = sqrt(temp7);
  return(dist_p);
}
```

(c) Collecting and Moving Reads

```
double distance(Point *p)
{ double dist_p;
  double temp3, temp6, temp7;
  Point bcomm1;
  blkmov(p,&bcomm1,sizeof(Point)) ;
  temp3 = bcomm1.x * bcomm1.x;
  temp6 = bcomm1.y * bcomm1.y;
  temp7 = temp3 + temp6;
  dist_p1 = sqrt(temp7);
  return(temp7);
}
```

(d) Blocking Reads

Figure 3: Optimizing remote reads

```
int scale_point(Point *p, double k)
{  p->x  = scale( p->x ,k);
   p->y  = scale( p->y ,k);
}
```

(a) original C code

```
int scale_point(Point *p, double k)
{ double temp1, temp2, temp3, temp4;
  temp1 =  p->x ;
  temp2 = scale(temp1,k);
   p->x  = temp2;
  temp3 =  p->y ;
  temp4 = scale(temp3,k);
   p->y  = temp4;
}
```

(b) SIMPLE C code

```
int scale_point(Point *p, double k)
{ double temp2, temp4;
  double comm1, comm2;
  comm1 =  p->x ;
  comm2 =  p->y ;
  temp2 = scale(comm1,k);
  temp4 = scale(comm2,k);
   p->x  = temp2;
   p->y  = temp4;
}
```

(c) Collecting and Moving Reads/Writes

```
int scale_point(Point *p, double k)
{ double temp2, temp4;
  Point bcomm1;
  blkmov(p,&bcomm1,sizeof(Point)) ;
  temp2 = scale(bcomm1.x,k);
  temp4 = scale(bcomm1.y,k);
  blkmov(&bcomm1,p,sizeof(Point)) ;
}
```

(d) Blocking Reads/Writes

Figure 4: Optimizing remote reads/writes

later may expose opportunities for blocking. Figure 4(c) shows the remote writes moved later, and this is a good idea if the blocking transformation in Figure 4(d) is performed. However, it is a bad idea if blocking should not be performed. Thus, we can see that there is a fine balance between the placement of remote write operations and the blocking transformations.

The two examples presented in this section show two simple applications of the communication optimizations. In general, one needs more sophisticated analyses that determine when it is safe to move communication, where to place the communication, and where to apply blocking.

4 Communication Optimizations

In this section, we describe the algorithm for communication optimization. It consists of an analysis phase followed by a transformation phase. The analysis phase, called *possible-placement analysis*, collects the set of remote reads and writes that can possibly be placed at each program point. The transformation phase, called *communication selection*, picks "the best" location for each remote read/write and performs the appropriate transformation, applying blocking when applicable.

The algorithm operates on SIMPLE, a compositional intermediate representation for C programs [19]. SIMPLE programs are composed of basic statements (assignments and function calls), statement sequences, conditionals (**if** and **switch**), and loops (**while**, **do**, and **for**). Each basic statement has a unique label, and can have at most one remote operation (remote read or remote write). Basic statements that involve remote operations are called *remote basic statements*, those without remote operations are called *local basic statements*. There is no irregular flow of control since SIMPLE programs have been automatically structured using goto-elimination [9]. Thus, we give our analysis rules in a structured form.

The algorithm makes use of advanced side-effect analysis. Each basic and compound statement is decorated with the set of locations read/written. In addition, statements involving pointers to the heap are decorated with heap read/write sets that have been computed using connection analysis [10, 11].

4.1 Possible-Placement Analysis

Possible-placement analysis collects sets of remote communication expressions. Each *remote communication expression* is a 4-tuple $(p, f, n, Dlist)$ where p is a pointer variable, f is the field of a struct, n is an estimated frequency, and $Dlist$ is a set of basic statement labels. For convenience we sometimes write these tuples combining the first two components, so that $(p, f, n, Dlist)$ is sometimes written as $(p{\rightarrow}f, n, Dlist)$. Possible-placement analysis computes two sets RemoteReads and RemoteWrites, defined as follows:

RemoteReads(S): the set of remote reads that may be safely placed just before statement S;

RemoteWrites(S): the set of remote writes that may be safely placed just after statement S.

These sets are collected using structured analyses. RemoteReads are propagated via a backwards analysis, whereas RemoteWrites are propagated via a forwards analysis. Each analysis is completed with a single traversal of the structured SIMPLE representation, no iteration is required. The structured analysis rules are given in Figures 5 and 6.

The first rule in Figure 5 shows the driving rule, collectCommSet, that selects the appropriate rule depending on the kind of the statement. The simplest rule, for basic statements, is collectCommSetBasic. A basic remote statement of the form $S_i : lhs = p{\rightarrow}f$ generates the RemoteRead tuple $(p{\rightarrow}f, 1, Si)$. A statement of the form $S_i : p{\rightarrow}f = rhs$ generates a RemoteWrite tuple of the form $(p{\rightarrow}f, 1, Si)$.

The last two rules in Figure 5, collectCommReadsSeq and collectCommWritesSeq, demonstrate how the tuples are propagated through statement sequences. Let us first focus on collectComReadsSeq, the rule for propagating RemoteReads earlier in the program. In this case RemoteRead tuples are propagated from S_n to S_1. At each step i, the RemoteRead set has already been calculated for the program point just before S_i (currCommReadSet). Statement S_{i-1} is analyzed to find the set of RemoteReads tuples that it generates (predCommReadSet), and then all tuples from currCommReadSet that are not killed by S_{i-1} are added to predCommReadSet, giving the set of RemoteReads valid at the program point just before S_{i-1}. Determining the kill set is actually the tricky part of the algorithm and requires relatively detailed side-effect analysis. It is obvious that if S_i writes to pointer p (either directly or via an alias), then p is no longer pointing to the same structure, and all tuples with p as the first item must be killed. However, one must also kill all tuples of the form $(p{\rightarrow}f, *, *)$ if S_i writes to $p{\rightarrow}f$ **via an alias**. Note that a direct write via $p{\rightarrow}f$ should not be killed because in the best case we want to replace all reads and writes via pointer p with accesses to a local structure. We use our connection analysis with anchor handles to distinguish between direct reads and aliased reads [11].

The rule for propagating RemoteWrites forwards, collectCommWritesSeq, is similar to the previous function. In this case the statement sequence is processed from S_1 to S_n. The other crucial difference is that tuples of the form $(p{\rightarrow}f, *, *)$ must be killed if S_i reads or writes $p{\rightarrow}f$ via an alias.

To demonstrate the backward propagation of RemoteReads, consider the example program in Figure 7. This program traverses a list pointed to by `p`, looking for a point that is within `epsilon` distance to the point pointed to by `t`. The last such point is pointed to by `close`. After the loop the differences between the `x` and `y` fields of `t` and `close` are computed.

The backward propagation proceeds as follows. First, consider the outer statement sequence $S1; ...; S8$. The analysis proceeds starting with $S8$ which has no remote accesses, and therefore generates the empty set. Statement $S7$ generates $(t{\rightarrow}y, 1, S7)$, statement $S6$ generates $(close{\rightarrow}y, 1, S6)$, and so on until statement $S3$ where the set $\{(t{\rightarrow}y, 1, S7), (close{\rightarrow}y, 1, S6), (t{\rightarrow}x, 1, S4), (close{\rightarrow}, 1, S3)\}$ has been calculated.

Figure 6 gives the rules for handling loops and conditionals. In the case of `if` and `switch` statements each alternative is analyzed to give the sets of RemoteReads or RemoteWrites generated by each alternative. For the case of RemoteReads analysis, all tuples from all alternatives are propagated out of the conditionals. This is because we are optimistically propagating reads as early as possible, since it is safe to read spurious fields, and never use their value.[1] However, we must be more conservative since it is certainly not safe to include spurious writes. Thus, for RemoteWrites we only include tuples that occur in all alternatives. To reflect the conditional nature of the control flow, we adjust the frequencies in the tuples. Our simple scheme simply divides the current frequency by the number of alternatives, although clearly either static or dynamic branch prediction information would be very valuable here. When moving tuples out of conditionals, tuples referring to the same location are merged by summing their adjusted frequencies and taking the union of their definition sets.

Loop statements are handled by first analyzing the body of loop giving the RemoteReads/RemoteWrites valid for the top/bottom of the loop body. We then propagate all tuples that can not be killed by the loop. Tuples propagated outside of the loop have their frequencies increased, corresponding to the expected number of times the loop will execute.

To demonstrate RemoteReads analysis for loops, consider the example in Figure 7, statement $S2$. The analysis of the body (statements $S9 \ldots S15$) results in the RemoteReads set $\{(p{\rightarrow}next, 1, S15), (t{\rightarrow}y, 1, S12), (t{\rightarrow}x, 1, S11), (p{\rightarrow}y, 1, S10), (p{\rightarrow}x, 1, S9)\}$. Since p is written in the body, all tuples with p must be killed, and only $\{(t{\rightarrow}y, 1, S12), (t{\rightarrow}x, 1, S11)\}$ are generated by the loop. The loop also writes the pointer *close*, and so it kills the tuples $(close{\rightarrow}y, 1, S6)$ and $(close{\rightarrow}x, 1, S3)$ that were valid at $S3$. Combining the remaining tuples at $S3$, $\{(t{\rightarrow}x, 1, S4), (t{\rightarrow}y, 1, S7)\}$, with the frequency-adjusted tuples generated by the loop $\{(t{\rightarrow}x, 10, S11), (t{\rightarrow}y, 10, S12)\}$, gives the set $\{(t{\rightarrow}x, 11, S11 : S4), (t{\rightarrow}y, 11, S12 : S7)\}$ at $S2$. This set propagates to $S1$ as well.

4.2 Communication Selection

After possible-placement analysis, each statement is associated with a set of remote communication expressions (RCEs), which can be safely placed before/after the statement. Based on this information we have developed some heuristics for placing the communications, and for selecting the correct kind of communications.

In compiling with split-phase remote memory operations it is usually beneficial to place these operations as early as possible. This allows some overlap of communication with computation. Thus, we follow an earliest placement policy for placing remote reads. This is

[1] By moving tuples out of conditionals and loops we may move a pointer dereference to a spot in the program that will cause a dereference that would not have occurred in the original program. In the transformation stage we ensure that it is safe to issue such a dereference.

```
/* computation of communication read expressions for a statement */
fun collectCommSet(stmt, accessType) =
  case (type(stmt)) of
    <Basic> => return (collectCommSetBasic(stmt, accessType));
    <Seq> =>  if (accessType == READ)
                return(collectCommReadsSeq(stmt));
              else /* must be a WRITE */
                return(collectCommWritesSeq(stmt));
    <Loop> => return(collectCommSetLoop(stmt, accessType));
    <If> ==>  return(collectCommSetIf(stmt, accessType));
     ...             /* Other stmts, switch...*/

/* collect remote communication expressions for a basic statement */
fun collectCommSetBasic(stmt, accessType) =
  if ((accessType == READ) && isRemoteAccess(rhs(stmt)))
     return <basevar(rhs(stmt)),field(rhs(stmt)),1,label(stmt)>;
  else if ((accessType == WRITE) && isRemoteAcess(lhs(stmt)))
     return <basevar(lhs(stmt)),field(lhs(stmt)),1,label(stmt)>;
  return EMPTY;

/* computation of communication read expressions for a statement
 * sequence, using a backward propagation scheme */
fun collectCommReadsSeq(stmtSeq) =
  [s_n:s_1] = getStmts(stmtSeq);
  currCommReadSet = collectCommSet(s_n, READ);
  foreach stmt s_i in [s_n:s_2] /* backward propagation from s_n to s_1 */
    j = i - 1; /* subscript for the predecessor statement in the sequence */
    predCommReadSet = collectCommSet(s_j, READ);
    foreach commExpr (p,f,n,d) in currCommReadSet
      if (varWritten(p, s_j)) /* base variable p itself is written */
        continue; /* commExpr cannot be propagated above s_j */
      if (accessedViaAlias(p, f, d, s_j, WRITE))
        continue; /* p->f possibly written via an alias, say t->f: can't be propagated */
      addToSet((p,f,n,d), predCommReadSet); /* propagate commExpr to s_j */
    currCommReadSet = predCommReadSet;
  return(commReadSet(s_1));

/* computation of communication write expressions for a statement
 * sequence, using a forward propagation scheme */
fun collectCommWritesSeq(stmtSeq) =
  [s_1:s_n] = getStmts(stmtSeq);
  currCommWriteSet = collectCommSet(s_1, WRITE);
  foreach stmt s_i in [s_1:s_n-1] /* forward propagation from s_1 to s_n */
    j = i + 1; /* subscript for the successor statement in the sequence */
    succCommWriteSet = collectCommSet(s_j, WRITE);
    foreach commExpr (p,f,n,d) in currCommWriteSet
      if (varWritten(p, s_j)) /* base variable p itself is written */
        continue; /* commExpr cannot be propagated above s_j */
      if ( accessedViaAlias(p, f, d, s_j, READ  ) ||
           accessedViaAlias(p, f, d, s_j, WRITE )    )
        continue; /* p->f possibly written via an alias, say t->f: can't be propagated */
      addToSet((p,f,n,d), succCommWriteSet); /* propagate commExpr to s_j */
    currCommWriteSet = succCommWriteSet;
  return(commWriteSet(s_n));
```

Figure 5: Possible Placement Analysis - Basic Rules

achieved via a top-down traversal of the SIMPLE representation. At the beginning of the traversal a hash table is generated that is used to contain all remote memory operations that have already been selected. Each remote memory operation is a triple (p, f, d), also written $(p{\rightarrow}f, d)$, where p is a pointer, f is a field, and d is the label of the statement containing the memory reference. Initially the hash table is empty. At each statement, S_i, the RemoteReads set is examined, and any tuples already in the hash table are removed from the RemoteReads set. For each remaining access of the form $(p{\rightarrow}f, n, Dlist)$, a decision is made whether or not to to insert the remote read at this point. If the frequency n is 1 or more and it is safe to place a dereference to p at this point in the program, then this is a candidate for inclusion.[2] If the tuple is a candidate, then an entry

[2]There are several ways of ensuring that the dereference is valid. One method is to check that there exists some dereference to p on all program paths starting at S_i. Another method is to use a *nilness* analysis to determine which pointers are definitely not nil. In our run-time system we also have the option of issuing a remote operation to a potentially-invalid address. In this case one could speculatively issue the remote operation, even for an invalid address.

```
/* computation of communication read/write sets for an if stmt : the return
 * commSet consists of commExprs that can be moved out of the if stmt */
fun collectCommSetIf(ifStmt, accessType) =
  commIfSet = EMPTY;
  commThenSet = collectCommSet(ifStmt.thenpart, accessType);
  commElseSet = collectCommSet(ifStmt.elsepart, accessType);
  if (accessType == READ)
   foreach commExpr (p,f,n,d) in commThenSet /* merge then set */
      n = adjustFrequency(n, ifStmt);
      addToSet((p,f,n,d), commIfSet);
   foreach commExpr (p,f,n,d) in commElseSet  /* merge else set */
      n = adjustFrequency(n, ifStmt);
      addToSet((p,f,n,d), commIfSet);
   return (commIfSet);
  else /* if (accessType == WRITE) */
   foreach commExpr (p,f,n,d) in commThenSet
     if ((p,f,n1,d1) exists in commElseSet ) /* Same field expression is written in else part */
       n = adjustFrequency(n, ifStmt);
       addToSet((p,f,n,d), commIfSet);
       n1 = adjustFrequency(n1, ifStmt);
       addToSet((p,f,n1,d1), commIfSet);
   return (commIfSet); /* commExprs that can be moved below the if stmt */

/* computation of communication read/write sets for a loop: the return
 * commSet consists of commExprs that can be moved out of the loop */
fun collectCommSetLoop(loopStmt, accessType) =
  commLoopSet = EMPTY;
  commBodySet = collectCommSet(loopStmt.body, accessType);
  if (accessType == READ)
   foreach commExpr (p,f,n,d) in commBodySet
     if (varWritten(p, loopStmt) || accessedViaAlias(p, f, d, loopStmt, WRITE))
       continue; /* cannot move it above the loop */
     n = adjustFrequency(n, loopStmt);
     addToSet((p,f,n,d), commLoopSet);
  else /* if (accessType == WRITE) */
    if (executesOnce(loopStmt))
     foreach commExpr (p,f,n,d) in commBodySet
       if (varWritten(p, loopStmt) ||
           accessedViaAlias(p, f, d, loopStmt, READ) ||
           accessedViaAlias(p, f, d, loopStmt, WRITE))
         continue; /* cannot move it below the loop */
       n = adjustFrequency(n, loopStmt);
       addToSet((p,f,n,d), commLoopSet);
  return(commLoopSet); /* commExprs that can be moved above/below the loop */

fun adjustFrequency(freq, stmt) =
  if (isLoopStmt(stmt))
    return (freq * 10); /* moving commExpr out of a loop */
  if (isIfStmt(stmt))
    return (freq / 2); /* moving commExpr out of an if statement */
  if (isSwitchStmt(stmt))
    n = numberOfCaseStmts(stmt); /* number of case stmts involved */
    return (freq / n); /* moving commExpr out of a switch statement */
  return (freq); /* return original value */
```

Figure 6: Possible Placement Analysis - Compound Rules

for each statement label in *Dlist* is made into the hash table.

After all the candidates at a program point are selected we then determine if some of them should be blocked. This choice depends upon the target architecture. For example, for our target architecture, pipelining is better for two remote accesses, but blocked communication is better for three or more accesses. If the structure being read is very large compared to the number of fields actually required, then the tradeoff shifts slightly towards pipelined communication. We use a cost model based on the number of words needed, and the size of the block to determine the best choice. Note that if a pointer p is only used for reads, then a block move is safe, even if we read spurious fields.

Consider the example in Figure 8(a) (the same program as used in Figure 7 to illustrate possible-placement analysis). In placing the communication, the program is processed starting at statement $S1$. At this point there are two entries for pointer t, each with a frequency count of 11, and it is safe to dereference t at this point. Thus, both of these tuples are selected as candidates. Since

```
S1: p = head; {(t→x, 11, S11 : S4), (t→y, 11, S12 : S7)}
S2: while (p!=NULL) {(t→x, 11, S11 : S4), (t→y, 11, S12 : S7)}
    S9:  { ax = p->x ; {(p→next, 1, S15), (t→y, 1, S12), (t→x, 1, S11), (p→y, 1, S10), (p→x, 1, S9)}
    S10:   ay = p->y ; {(p→next, 1, S15), (t→y, 1, S12), (t→x, 1, S11), (p→y, 1, S10)}
    S11:   bx = t->x ; {(p→next, 1, S15), (t→y, 1, S12), (t→x, 1, S11)}
    S12:   by = t->y ; {(p→next, 1, S15), (t→y, 1, S12)}
    S13:   dist = f(ax,ay,bx,by); {(p→next, 1, S15)}
    S14:   if (dist < epsilon) close = p; {(p→next, 1, S15)}
    S15:   p = p->next ; {(p→next, 1, S15)}
         }
S3: cx = close->x ;   {(t→y, 1, S7), (close→y, 1, S6), (t→x, 1, S4), (close→x, 1, S3)}
S4: tx = t->x ;       {(t→y, 1, S7), (close→y, 1, S6), (t→x, 1, S4)}
S5: diffx = cx - tx;  {(t→y, 1, S7), (close→y, 1, S6)}
S6: cy = close->y ;   {(t→y, 1, S7), (close→y, 1, S6)}
S7: ty = t->y ;       {(t→y, 1, S7)}
S8: diffy = cy - ty;  {}
```

Figure 7: Example propagation of RemoteRead Sets

there are only two reads, they are pipelined (as shown in Figure 8(b)). Entries for $(t \rightarrow x, S11)$, $(t \rightarrow x, S4)$, $(t \rightarrow y, S12)$, and $(t \rightarrow y, S7)$ are made in the hash table. At statement $S2$ there will be no remaining remote tuples (after removing all entries in the hash table), and so nothing is done. At statement $S9$ there are three remaining tuples, all referencing pointer p, and thus a blocked communication is selected. For statements $S10$ to $S15$ there are no remaining tuples, and the next non-empty list is at statement $S3$. At statement $S3$ there are two pipelined reads via pointer *close*.

Handling the insertion of RemoteWrites is somewhat more difficult. In this case we must balance the cost of delaying some writes in order to expose opportunities for blocked writes. Further, it is not safe to write spurious fields in a block move. We handle these situations by propagating special RemoteFill tuples using the same algorithm as for RemoteReads. These tuples ensure that all fields in a struct will be read before a blocked write is inserted.

5 Experimental Results

In order to evaluate our approach, we have used EARTH-MANNA distributed-memory parallel system as our target architecture. In this section, we provide a brief description of the architecture, and the experimental results we obtained.

5.1 The EARTH-MANNA Architecture

In the EARTH model, a multiprocessor consists of multiple EARTH nodes and an interconnection network [13, 17]. As illustrated in Figure 9, each EARTH node consists of an *Execution Unit* (EU) and a *Synchronization Unit* (SU), linked together by buffers. The SU and EU share a local memory, which is part of a *distributed memory* architecture in which the aggregate of the local memories of all the nodes represents a global memory address space.

The EU processes instructions in an active thread, where an active thread is initiated for execution when the EU fetches its *thread id* from the ready queue. The EU executes a thread to **completion** before moving to another thread. It interacts with the SU and the network by placing messages in the event queue. The SU fetches these messages, plus messages coming from remote processors via the network. The SU responds to remote synchronization commands and requests for data, and also determines which threads are to be run and adds their thread ids to the ready queue.

Memory operations are supported as split-phase transactions - computation synchronizes on the completion of a remote access, not on the issue of the operation. EARTH-MANNA supports split-phase primitives for reading/writing scalar data fields (char, integer, float, double), and also primitives for reading/writing blocks of data.

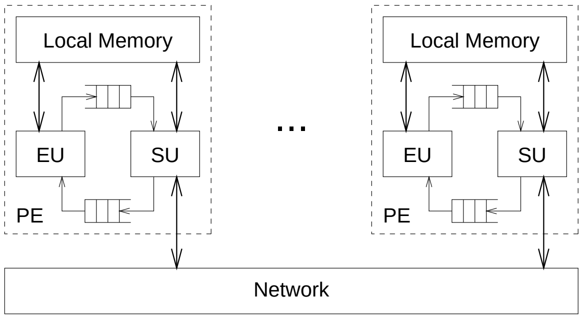

Figure 9: The EARTH architecture

Our experiments have been done using the EARTH-MANNA parallel machine[3]. Each MANNA node consists of two Intel i860 XP CPUs, clocked at 50MHz, 32MB of dynamic RAM and a bidirectional network interface capable of transferring 50MB/S in each direction. The two processors on each node are mapped to the EARTH EU and SU.

The EARTH runtime system supports efficient remote operations. Typically, the cost depends on the location of the data that is referenced. Table I shows the cost of communication in two extreme cases on EARTH-MANNA, *Sequential* and *Pipelined*. The sequential value indicates how long it takes to perform the complete operation, including context switching. In the pipelined case, operations are issued as fast as possible, without the need to synchronize before issuing the next operation. Obviously, the pipelined numbers are lower, as the EU, SU and network can all work in parallel. Therefore it is always better to pipeline these operations. The trade-off between pipelining and block-moving the fields that are allocated together depends on the size of data. Even though a block-move instruction

```
    {(t→x, 11, S11 : S4), (t→y, 11, S12 : S7)}
S1: p = head;
S2: while (p!=NULL)
        {(p→next, 1, S15), (p→y, 1, S10), (p→x, 1, S9)}
    S9:   { ax = p->x ;
    S10:    ay = p->y ;
    S11:    bx = t->x ;
    S12:    by = t->y ;
    S13:    dist = f(ax,ay,bx,by);
    S14:    if (dist < epsilon) close = p;
    S15:    p = p->next ;
          }
    {(close→y, 1, S6), (close→x, 1, S3)}
S3: cx = close->x ;
S4: tx = t->x ;
S5: diffx = cx - tx;
S6: cy = close->y ;
S7: ty = t->y ;
S8: diffy = cy - ty;
```

(a) selected RemoteRead sets

```
S1': comm1 = t->x ;
S1": comm2 = t->y ;
S1: p = head;
S2: while (p!=NULL)
    S9':  { blkmov(p,&bcomm1,sizeof(POINT) ;
    S9:     ax = bcomm1.x;
    S10:    ay = bcomm1.y;
    S11:    bx = comm1;
    S12:    by = comm2;
    S13:    dist = f(ax,ay,bx,by);
    S14:    if (dist < epsilon) close = p;
    S15:    p = bcomm1.next;
          }
S3': comm3 = close->x ;
S3": comm4 = close->y ;
S3: cx = comm3;
S4: tx = comm1;
S5: diffx = cx - tx;
S6: cy = comm4;
S7: ty = comm2;
S8: diffy = cy - ty;
```

(b) after transformation

Figure 8: Applying Communication Selection

EARTH Operation	*Sequential Remote*	*Pipelined Remote*
Read word	7109 ns	1908 ns
Write word	6458 ns	1749 ns
Blkmov word	9700 ns	2602 ns

Table I: Cost of communication on EARTH-MANNA

is more expensive for one word, we have found that a block-move is better when three or more words can be moved together. Thus, in our experiments we used a threshold of three to determine when to issue pipelined operations and when to block them.

5.2 Experimental results

We have experimented with five benchmarks from the Olden suite [18], described in Table II. All five benchmarks use dynamic data structures (trees and lists). Thus the benchmark suite is suitable to evaluate our communication analysis focused on pointers.

The benchmarks were all written in EARTH-C, they use the best data distribution strategy we have discovered to date for each benchmark, and they exploit the parallelism available in an efficient way. We performed our experiments using the EARTH-McCAT compiler, comparing the performance with and without communication optimizations. We refer to the unoptimized programs as the *simple* versions, and the optimized programs as *optimized* versions.

5.2.1 Dynamic Counts of Communication Operations

In Figure 10, we compare the number of remote communication operations for the simple and optimized versions. The number under each benchmark name gives the total number of communication operations in millions. Each benchmark has two bars, the left bar represents the number of communication operations performed by the simple version, normalized to 100. The right bar shows the number of remote operations performed by the optimized version. In each bar we indicate the proportion that are read-data, write-data and blkmovs.[3] It is clear that in all cases the total number of communication operations reduces. The number of read-data and write-data operations reduce because of redundant communication elimination and blocking. The number of blkmov operations increases because some individual read-data/write-data operations were combined into new blkmov operations.

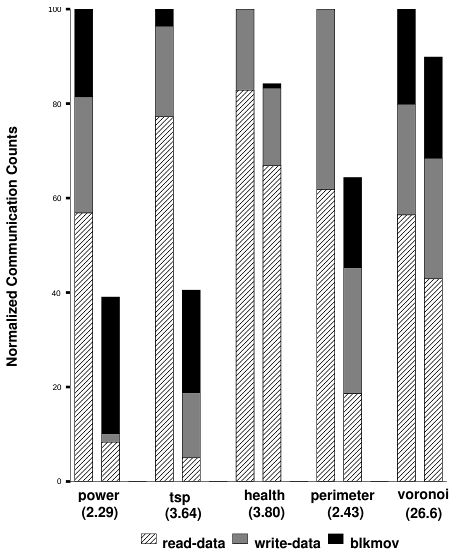

Figure 10: Improvement on dynamic communication counts

5.2.2 Performance Improvement

In Table III we provide the performance improvement achieved for each benchmark, via communication optimization.

[3]Note that the unoptimized versions contain some blkmovs because the compiler inserts blkmovs for assignments to entire `structs`.

Benchmark	Description	Problem Size
power	Optimization Problem based on a variable k-nary tree	10,000 leaves
perimeter	Computes the perimeter of a quad-tree encoded raster image	Maximum tree-depth 11
tsp	Find sub-optimal tour for traveling salesperson problem	32K cites
health	Simulates the Colombian health-care system using a 4-way tree	4 levels and 600 iterations
voronoi	Computes the Voronoi Diagram of a set of points	32K points

Table II: Benchmark Programs

The first data column gives the sequential execution time for each benchmark. This time is measured for a purely sequential version of the benchmark running on 1 node of the MANNA machine. In these versions all data is local, there is no parallelism, and there are no calls to any EARTH runtime operations. Thus, this is a truly sequential program, with no extra overhead.

The next two columns give the times for the simple and optimized parallel versions of the benchmark, for 1, 2, 4, 8 and 16 processors. This is followed by two columns that give the speedup of the simple and optimized versions over the sequential version. Finally, the last column gives the performance improvement due to communication analysis. Note that communication analysis gives some performance improvement for all benchmarks. In general the performance improvement increases as the number of processor increases. We discuss each benchmark in more detail below.

Power: This benchmark implements the power system optimization problem. It uses a four-level tree structure with different branching widths at each level.

Communication optimization achieves up to 6.9% speed-up for this benchmark. The main benefit comes from blocking. It is computation-intensive benchmark, with functions operating on a particular node of the tree. These functions typically read the fields from a node into scalars at different points, perform computation, and write the values back into the field.

Since all field accesses are with respect to a given node, communication optimization is able to place all the read accesses early, and all the write accesses late in the function. Subsequently, these accesses are blocked, as they correspond to fields within a specific node. An example code fragment from this benchmark is shown in Figure 11(a), which illustrates this observation.

Perimeter: This benchmark computes the perimeter of a quad-tree encoded raster image. The unit square image is recursively divided into four quadrants until each one has only one point. The tree is then traversed bottom-up to compute the perimeter of each quadrant.

It is an irregular benchmark, and each computation requires accesses to tree nodes which may not be physically close to each other. Thus it is a communication-intensive benchmark, and the benefits of communication optimization are more visible for it. We are able to achieve upto 15% speed-up over the simple version. The main optimization applied for this benchmark is also blocking. An example code fragment from *perimeter* is shown in Figure 11(b). Here the `blkmov` replaces three remote read operations. The optimization is applied within a recursive function, where the program spends most of the time. Hence substantial speed-up is obtained.

Health: This benchmark simulates the Colombian health-care system using a 4-way tree. Each village has four child villages, and a village hospital, treating patients from the villages in the same subtree. At each time step, the tree is traversed, and patients, once assessed, are either treated or passed up to the parent tree node. The 4-way tree is evenly distributed among the processors and only top-level tree nodes have their children spread among different processors.

This benchmark has relatively few remote data accesses. Thus the speed-up obtained via communication optimization is not as significant. The communication optimizations applicable to this benchmark include pipelining, and redundancy elimination, as illustrated via the code fragment taken from it, shown in Figure 11(c).

Tsp: This benchmark solves the traveling salesman problem using a divide-and-conquer approach based on close-point algorithm. This algorithm first searches a suboptimal tour for each subtree(region) and then merges subtours into bigger ones. The tour found is built as a circular linked list sitting on top of the root nodes of subtrees.

Similar to perimeter, this benchmark is irregular in nature and performs a significant time in data accesses. We are able to obtain upto 11.98% speed-up for it. The benchmark mainly benefits from redundant communication elimination and communication pipelining.

Voronoi: The voronoi benchmark computes a Voronoi Diagram for a random set of points. The points are generated and stored in a binary tree, and the algorithm computes the Voronoi Diagrams of the two subtrees recursively, and merges them to form the final diagram.

The merge phase walks along the convex hull of the two sub-diagrams, alternating between in an irregular fashion, so the benchmark spends a significant time in data accesses. Therefore, we obtain upto 15.48% speed-up by communication optimization. This mainly comes from redundant communication elimination and blocking.

6 Related Work

The closest related work is previous research at McGill on techniques to reduce communication overhead in parallel EARTH-C programs that use pointer-based dynamic data structures [22, 20].

The *locality analysis* presented in [22] focuses on eliminating pseudo-remote memory operations, i.e., operations that are assumed to be remote by the compiler, but actually access the local memory. Using the high-level data distribution information provided by the programmer, this analysis identifies pointers in the EARTH-C program, that can be declared as `local`. A

Benchmark		Sequential C (sec)	Simple EARTH-C (sec)	Optimized EARTH-C (sec)	Simple Speedup	Optimized Speedup	Optimized vs. Simple (%impr)
power	1 proc	62.4	67.76	66.76	0.92	0.93	1.48
	2 procs		35.96	34.41	1.74	1.81	4.31
	4 procs		18.58	17.58	3.36	3.55	5.38
	8 procs		9.77	9.12	6.39	6.84	6.65
	16 procs		5.23	4.86	11.93	12.84	7.07
tsp	1 proc	23.8	26.95	26.26	0.88	0.91	2.56
	2 procs		14.02	13.56	1.70	1.76	3.28
	4 procs		7.71	7.33	3.09	3.25	4.93
	8 procs		4.67	4.29	5.10	5.55	8.14
	16 procs		3.77	3.32	6.31	7.17	11.93
health	1 proc	143.72	144.09	144.05	0.99	0.99	0.03
	2 procs		84.25	80.72	1.71	1.78	4.19
	4 procs		48.32	44.78	2.97	3.21	7.33
	8 procs		42.74	37.69	3.36	3.81	11.82
	16 procs		33.88	28.84	4.24	4.98	14.88
perimeter	1 proc	5.45	7.32	6.75	0.74	0.81	7.79
	2 procs		3.67	3.35	1.49	1.63	8.72
	4 procs		2.16	1.94	2.52	2.81	10.19
	8 procs		1.12	0.98	4.87	5.56	12.50
	16 procs		0.75	0.63	7.27	8.65	16.00
voronoi	1 proc	3.98	7.27	6.78	0.55	0.59	6.74
	2 procs		3.23	2.85	1.23	1.40	11.76
	4 procs		2.52	2.13	1.58	1.87	15.48
	8 procs		1.31	1.17	3.04	3.40	10.69
	16 procs		1.04	0.88	3.83	4.52	15.38

Table III: Performance Improvement Results

`local` pointer is assumed to be always pointing to local memory by the EARTH-C compiler, and thus they are not mapped to remote memory operations. This problem is orthogonal to the communication analysis presented here. In this paper we are reducing the communication cost for accesses that might be remote.

Tang et. al [20] use standard compiler optimizations like common subexpression elimination, loop- and location-invariant removal to eliminate redundant pointer dereferences, and examine the effect of these optimizations on the quality of threaded code produced by the EARTH-C compiler. The communication analysis presented in this paper can also reduce redundant pointer dereferences (for example, when remote memory accesses are moved out of loops). However, in this paper we are also concerned about the placement of the pointer dereferences, pipelining remote operations, and blocking of associated remote operations.

Another approach for compilation of dynamic data structure based applications on distributed memory machines was proposed by Carlisle and Rogers [5, 4]. They propose the Olden runtime system, that uses a trade off between software caching of remote data and computation migration, depending on the data distribution and the amount of communication required. This decision is guided, in part, by programmer-specified *path-affinity* hints for recursive data structures. If accessing remote data is chosen, a software caching scheme is used to minimize the overhead of the communication. In our approach, the programmer makes the choice of whether or not to migrate computation using explicit constructs such as `@OWNER_OF` and `@HOME`, which enable the programmer to invoke functions at a given processor. Therefore, our communication optimization focuses on further reducing the overhead of communication that is considered necessary by the EARTH-C programmer, i.e., the communication that cannot be effectively substituted by computation migration. Instead of using a software caching scheme which requires runtime checks, our analysis decides at compile-time which pointer dereferences are reused, and which references can be blocked together. An interesting point would be to consider how the software caching scheme could benefit from our analysis.

Another form of communication optimization is prefetching. The most relevant work is compiler-based prefetching for recursive data structures by Luk and Mowry [16]. This work was directed towards reducing the memory latency time for superscalar processors. The basic idea is to automatically insert prefetch instructions for pointer references using three different schemes, including a scheme whereby nodes pointed to by some pointer p are greedily prefetched (for example, prefetching the nodes pointed to by $p \rightarrow left$ and $p \rightarrow right$ as soon as $p \rightarrow value$ is accessed. Our analysis is not trying to prefetch along a traversal chain, rather we are concentrating on moving pointer dereferences early, and on blocking related derefences together (i.e. we would read the complete node pointed to by p at the same time, we would not speculatively dereference any of those fields.) Further, in our case we are concerned with distrubuted memory parallel processors, and so we have a different cost model. In our case it is always a good idea to move accesses early (our results are written to memory and not a cache), and, in general, it is too expensive for us to speculatively prefetch pointer references.

For array-based scientific computations, a significant amount of work has been done on communication optimization for distributed memory compilation [1, 2, 6, 21]. The approach proposed by Chakrabarti et. al [6] is the most recent. They propose a global

```
Branch Compute_Branch(br, theta_R, ....)
{ ...
  blkmov(br,&bcomm7, sizeof(branch));
  if ((next != 0))
    { ...
      bcomm7.D.P = (temp_258 + temp_259);
      ...
      bcomm7.D.Q = (temp_260 + temp_261);
    }
  else
    { bcomm7.D.P = tmp.P;
      bcomm7.D.Q = tmp.Q;
    }
  temp_263 = bcomm7.R;
  temp_262 = (temp_263 * temp_263);
  temp_266 = bcomm7.X;
  temp_265 = (temp_266 * temp_266);
  ...
  bcomm7.alpha = (a / temp_313);
  ...
 bcomm7.beta = (b / temp_315);
 temp_317 = bcomm7.D;
 blkmov(&bcomm7, br, sizeof(branch));
  ...
}
```

(a) power

```
int R_sum_adjacent(p, q1, q2, size)
 { ...
  blkmov(p,&bcomm,sizeof(quad_struct)) ;
  temp_110 = bcomm.color;
  if ((temp_110 == 2))
  { switch (q1)
    { case 0: p1 = bcomm.nw;
              break;
      case 1: p1 = bcomm.ne;
              break;
      ...
    }
    switch (q2)
    { case 0: p2 = bcomm.nw;
              break;
      case 1: p2 = bcomm.ne;
              break;
      ...
    }
    x = R_sum_adjacent(p1, q1, q2, size_1);
    y = R_sum_adjacent(p2, q1, q2, size_1);
  }
  else
  { temp_112 = bcomm.color;
    if ((temp_112 == 1))
    ...
  }
}
```

(b) perimeter

```
void check_patients_inside(village, list)
{ ...
  comm6 = (*village).hosp.free_personnel;
  while ( (list != 0) )
  { p = (*list).patient;
    comm1 = (*list).forward;
    comm5 = (*p).time_left;
    comm5 = comm5 - 1;
    (*p).time_left = comm5;
    temp_56 = comm5;
    if ((temp_56 == 0))
    { t = comm6;
      comm6 = (t + 1);
      l = (&(*village).hosp.inside);
      R_removeList(l, p);
      l = (&(*village).returned);
      ...
    }
    list = comm1;
  }
  (*village).hosp.free_personnel = comm6  ;
}
```

(c) health

Figure 11: Sample extracts from optimized programs

scheduling approach for communication optimization of F90/HPF programs. Our strategy is similar to their approach in the scheduling aspect. One difference is that they consider only *read* remote communications, while we optimize both *read* and *write* remote accesses. Further, we follow the earliest placement policy. Chakrabarti et. al show that in some cases late placement can expose more opportunities for other optimizations like message combining for arrays. We are studying this interaction in the context of dynamic data structure based applications. In the set of benchmarks presented in this paper, we have not found any case where earliest placement policy inhibits other communication optimizations.

Agrawal et. al [1] indicated that interprocedural partial redundancy elimination can be beneficial for eliminating redundant communication for array-based scientific applications. This observation also applies to programs that use dynamic data structures. In our *tsp* benchmark, one of the pointer parameters passed to the function `distance` remains invariant across several calls to the function, and all the field accesses with respect to this pointer can be placed before the first call by interprocedural partial redundancy elimination. Currently, we achieve this effect via function inlining.

Krishnamurthy and Yelick [15] present communication optimizations in the context of compiling explicitly parallel programs to Split-C programs. Their optimizations include message pipelining by converting remote read/writes into their split-phase analogues, eliminating acknowledgement traffic, and reusing values from remote memory accesses. Their source language includes shared scalar variables and distributed arrays, but does **not** include global pointers. The emphasis of their work is an optimization framework that handles explicitly parallel programs that may read/write the same memory locations, and thus they must correctly handle interfering parallel computations. In contrast, we have a simpler parallel programming model, where our parallel threads do not interfere on ordinary remote memory accesses. However, we **do** allow global pointers, and in fact the focus of our work is optimizing programs using global pointers. Presumably the strengths of these two approaches could be combined to handle both the more general parallel programming model and global pointers.

For shared-memory models, transformations similar to communication optimization, can be used to reduce the synchronization overhead of the program [7]. Diniz and Rinard [7] present an approach that focuses on coalescing multiple critical sections that acquire and release the same lock multiple times, into a single critical region that acquires and releases the lock only once. To this end, they perform *lock movement* and *lock cancellation* transformations, which try to reduce the frquency with which the program acquires and releases locks. These transformations are similar in spirit to the redundant communication elimination optimization.

The elimination of redundant code has been examined quite rigorously for scalar computations. As an example, Knoop et. al. optimize the computation by optimal code motion [14]. They issue the computation as late as possible to avoid unnecessary register pressure while maintaining computational optimality. They focus on scalar variables, while we are interested in pointer dereferences. Further their algorithms treat each computation independently, while we consider the interaction between tuples in determining the best placement, and we consider the costs of different communication strategies to decide between pipelining and blocking.

7 Conclusions and Further Work

In this paper we have presented a communication analysis framework that is composed of *possible-placement analysis* and *communication selection*. The fundamental idea is that we wish to move remote read operations as early as possible in order to allow for overlap between communication and computation. Further, we wish to pipeline or block remote reads and writes when beneficial.

We presented *possible-placement* analysis as a structured analysis on the SIMPLE representation of the EARTH-McCAT compiler. In this analysis RemoteRead tuples are propagated optimistically upwards, and RemoteWrite tuples are propagated conservatively downwards. In order to determine when it is safe to propagate tuples, both stack and heap read/write sets are required.

The *communication selection* transformation uses the results of *possible-placement analysis*. The goal of the analysis is to locate the earliest point for remote reads, and to generate either pipelined or blocked communication. For remote writes, the communication is delayed if this enables blocked communication.

We implemented these techniques in the EARTH-McCAT compiler, and experimented with a set of five benchmarks These results show that the optimizations improve performance from about 2% to 16% over the unoptimized programs.

This work fits into the overall framework of the EARTH-McCAT compiler project where we are developing techniques to automatically generate low-level threaded programs from high-level parallel C programs. In this paper we have focused on one component - the communication optimization. By adding this component to the compiler we get one step closer to closing the gap between compiler-generated threaded programs and hand-coded threaded programs.

Our next step is to study the interaction of communication analysis with locality analysis and the thread generation algorithm. We would also like to add techniques for finding the best organization for fields within each `struct`. By placing those fields that are accessed remotely located close to one another, we can further improve the efficiency of the blocked communication.

References

[1] Gagan Agrawal, Joel Saltz, and Raja Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 258–269.

[2] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 126–138, Albuquerque, N. Mex., Jun. 1993.

[3] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 704–709, Cancún, Mexico, Apr. 1994. IEEE Comp. Soc.

[4] Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines.* PhD thesis, Princeton University Department of Computer Science, June 1996.

[5] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proc. of the Fifth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pages 29–38, Santa Barbara, Calif., Jul. 1995.

[6] Soumen Chakrabarti, Manish Gupta, and Jong-Deok Choi. Global communication analysis and optimization. In *Proc. of the ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 68–78, Philadelphia, Penn., May 1996.

[7] Pedro Diniz and Martin Rinard. Synchronization transformations for parallel computing. In *Conf. Rec. of the 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 187–200, Paris, France, Jan. 1997.

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation* , pages 242–256.

[9] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proc. of the 1994 Intl. Conf. on Computer Languages*, pages 229–240, Toulouse, France, May 1994.

[10] Rakesh Ghiya. *Putting pointer analysis to work.* PhD thesis, McGill U., Montréal, Qué., Nov. 1997.

[11] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Conf. Rec. of the 25th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 121–133, Jan. 1998.

[12] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Shereen Ghobrial, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. *Intl. J. of Parallel Programming*, 25(4):305–337, Aug. 1997.

[13] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *Intl. J. of Parallel Programming*, 24(4):319–347, Aug. 1996.

[14] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Trans. on Programming Languages and Systems*, 16(4):1117–1155, Jul. 1994.

[15] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation* , pages 196–204.

[16] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. of the Seventh Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, Mass., Oct. 1996.

[17] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proc. of the 23rd Ann. Intl. Symp. on Computer Architecture*, pages 178–188, Philadelphia, Penn., May 1996.

[18] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[19] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill U., Montréal, Qué., Sep. 1992.

[20] Xinan Tang, Rakesh Ghiya, Laurie J. Hendren, and Guang R. Gao. Heap analysis and optimizations for threaded programs. In *Proc. of the PACT'97*, pages 14–25, San Francisco, Nov. 1997. North-Holland Pub. Co.

[21] Reinhard von Hanxleden and Ken Kennedy. Give-N-Take — a balanced code placement framework. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation* , pages 107–120.

[22] Yingchun Zhu and Laurie Hendren. Locality analysis for parallel C programs. In *Proc. of the PACT'97*, pages 2–13, San Francisco, Nov. 1997. North-Holland Pub. Co.