# Locality Analysis for Parallel C Programs

Yingchun Zhu and Laurie J. Hendren
School of Computer Science
McGill University
Montreal, Quebec, CANADA H3A 2A7

**Abstract**

*Many parallel architectures support a memory model where some memory accesses are local, and thus inexpensive, while other memory accesses are remote, and potentially quite expensive. In order to achieve good parallel performance, it is often necessary to reduce the number of remote memory accesses. This can be done by the programmer, the compiler, or a combination of both. The overall goal is to minimize the work required by the programmer, and have the compiler automate the process as much as possible.*

*This paper reports on compiler techniques for decreasing the number of remote memory accesses using locality analysis for a parallel dialect of C called EARTH-C. The locality analysis uses an algorithm inspired by type inference algorithms for fast points-to analysis. The algorithm estimates when an indirect reference via a pointer can be safely assumed to be a local access. The locality inference algorithm is also used to guide the automatic specialization of functions in order to take advantage of locality specific to particular calling contexts.*

*The locality analysis and automatic specialization has been implemented in the EARTH-C compiler which produces low-level threaded code for the EARTH multithreaded architecture. Experimental results are presented for a set of benchmarks that operate on irregular, dynamically-allocated data structures. The techniques give moderate to significant speedups and they do lessen the burden on the programmer.*

## 1 Introduction

One of the key problems in parallel processing is to provide a programming model that is simple for the programmer. One would like to give the programmer a familiar programming language, and have the programmer focus on high-level aspects such as coarse-grain parallelism and perhaps some sort of static or dynamic data distribution. Compiler techniques are then required to effectively map the high-level programs to actual parallel architectures. In this paper we present some compiler techniques that simplify the programmer's job when expressing locality of pointer data structures.

As reported previously, we have developed a high-level parallel language called EARTH-C [1], and an associated compiler that translates EARTH-C programs to low-level threaded programs that execute on the EARTH multithreaded architecture [2, 3]. Our main emphasis is on the effective compilation of programs that use irregular, dynamically-allocated data structures. Our initial approach provided high-level parallel constructs, and type extensions to express locality. The compiler then used the type declarations and dependence analysis to automatically produce low-level threads.

Although our initial approach did provide a good, high-level, basis for programming the EARTH multithreaded architecture, we found that the programmer was forced to make many function specializations, and to declare the appropriate pointer parameters and locally-scoped pointer variables as local pointers. Thus, in order to experiment with various locality approaches, the programmer needed to edit many places in his/her program, and to make several copies of the same function, each copy specialized for a particular type of locality. In order to ease the burden on the programmer we have developed some new compiler techniques to infer the locality of pointer variables, and then automatically produce the specialized versions of the functions. This allows the programmer to make very minimal changes to his/her high-level program in order to try various approaches to a problem. It also leads to shorter source programs because the programmer does not need to make several similar copies of the same function.

The main idea behind our approach is that we use the information about the context of function calls and memory allocation statements to infer when indirect memory references must refer to local memory. We then automatically create specializations of functions with the appropriate parameters and locally-scoped variables explicitly declared as local pointers. This information is then used by the thread generator to reduce the number of remote operations required in the low-level threads.

In order to test our approach we implemented the techniques in the EARTH-McCAT C compiler, and we experimented with a collection of pointer-based benchmarks. We present experimental measurements on the EARTH-MANNA machine to compare the performance of benchmarks without locality analysis, with locality analysis, and hand-coded versions with "the best" locality.

The rest of the paper is organized as follows. Section 2 presents an overview of the EARTH-C language, the EARTH-McCAT compiler, and the EARTH-MANNA architecture. Section 3 provides some examples to motivate the locality analysis, and Section 4 describes the analysis itself. In Section 5 we give experimental results for our set of benchmarks programs. Finally, in Section 6 we discuss some related work, and in Section 7 we give conclusions and some suggestions on further work.

## 2 The EARTH-C Language

The EARTH-C 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 multithreaded architecture. In this section we provide an overview of the important points about the language, and the target architecture. More complete descriptions of the EARTH project can be found elsewhere [2].

### 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 C program is a valid EARTH-C program, and the compiler will automatically produce a correct low-level threaded program. However, usually the programmer will make some minimal modifications to the program to expose coarse-grain parallelism, and to add information about data locality.

Figure 1 gives two sample list processing functions, written in EARTH-C. 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 interactions 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.

The EARTH-C compiler captures coarse-grain parallelism at the level of function invocations. Ordinary C function calls are translated into lower-level threaded-C `TOKEN` calls. Such `TOKEN` calls are handled by the EARTH runtime load balancer, and the call will be mapped to a processor at runtime. However, in EARTH-C, it is also possible for the programmer to explicitly specify where the invocation should be executed using the syntax `p(..)@expr`. In this case the underlying threaded-C `INVOKE` mechanism is used to explicitly map the invocation to the processor specified by `expr`. For example, in Figure 1(b), the call to `equal_node` is mapped to the processor owning node `x`, whereas the recursive call to `count_rec` is not explicitly mapped to a processor, and it will be assigned by the runtime load balancer.

In both cases, using the `TOKEN` and `INVOKE` mechanisms, the activation frame is allocated on the processor assigned the invocation, and the invocation will remain on the same processor for the lifetime of the invocation. Thus, the EARTH-C compiler can assume that all parameters and locally-scoped variables are local memory accesses. On the contrary, since the invocations are mapped at runtime (either using the runtime load balancer, or according to an expression that is evaluated at runtime), the compiler must assume that all accesses to global variables and all memory accesses via pointer indirections, are to remote memory. Using these assumptions for function `count_rec` in Figure 1(b), we can see that accesses to `head`, `x`, `c1`, and `c2` are local accesses, but the access to `head->next` is a remote memory access. Note that to make the locality easier to see, we underline all remote memory accesses, and we put local pointer declarations in bold type.

As the target architecture for EARTH is a distributed-memory machine, this 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. If the remote memory access turns out to be actually on the same processor as the request, the communication time will be minimal, but it is still significantly more expensive than making a direct local memory access.

Even though multithreaded architectures can hide some communication costs, it is clearly advantageous to maximize the use of local memory, whenever possible. In order to expose more locality to the compiler, EARTH-C has the concept of local pointers. If the programmer knows that a pointer **always** points to local memory, then the keyword `local` may be added to the pointer type declaration. In Figure 1(a), all calls to `equal_node` were made to the owner of the first argument. Thus, in the declaration of `equal_node` we have declared the first parameter to have type `node local *p`, which reading right to left, says that `p` is a pointer to a local node. Thus, in the body of `equal_node`, the EARTH compiler may assume that `p->value` is a local memory reference, but `q->value` is potentially a remote memory reference. Figure 1(b) illustrates the opposite case, where the second parameter of `equal_node` is a local pointer, and in this case the EARTH compiler must assume that `p->value` is potentially remote, whereas `q->value` is local.

EARTH-C also includes another form of function declaration that also expresses locality. Functions may be declared using the keyword `basic`. These `basic` functions must only reference local memory, and they may not call any ordinary (remote) functions. Basic functions are translated into very cheap function invocations in the target threaded-C code, and all memory references within their bodies are ordinary C variable references. Thus, sometimes programmers use basic functions to indicate locality for all variable references

```
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

within a function body.

The purpose of this paper is to help automate the generation of the local pointer declarations and to automatically provide specialized versions of functions for different calling contexts. Thus, the programmer concentrates on expressing where the computation should be mapped, and the compiler infers the locality information for pointers, and inserts the correct local pointer declarations. This reduces the burden on the programmer, leads to shorter source programs, and makes changing the source program less error prone. In the examples in Figure 1, the programmer would only need to declare one version of `equal_node`, and the compiler would automatically generate the appropriate specializations depending on the calling context.

### 2.2 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 important points are that the source program is simplified into an AST-based SIMPLE intermediate representation [4]. At this point programs have been made structured via goto-elimination, and each statement has been simplified into a series of simple, basic statements. For each statement (including assignment statements, conditionals, loops, and function calls), we have the results of side-effect analysis that gives that set of locations read/written by the statement. The availability of this read/write information allows our locality analysis to be simple, and efficient.

The methods presented in this paper are found in Phase II, where parallelization and locality enhancement is done. In these phases we use the results of the analyses from Phase I in order to transform the SIMPLE program representation into a semantically-equivalent program. The transformations presented in this paper introduce locality declarations, and produce new specialized versions of some functions.

Phase III takes the transformed SIMPLE program from Phase II, generates threads, and produces the target threaded-C code. By exposing more locality in Phase II, we allow the thread generator to deal with fewer remote memory accesses, and this should lead to fewer threads, fewer calls to EARTH primitives, and more efficient parallel programs.

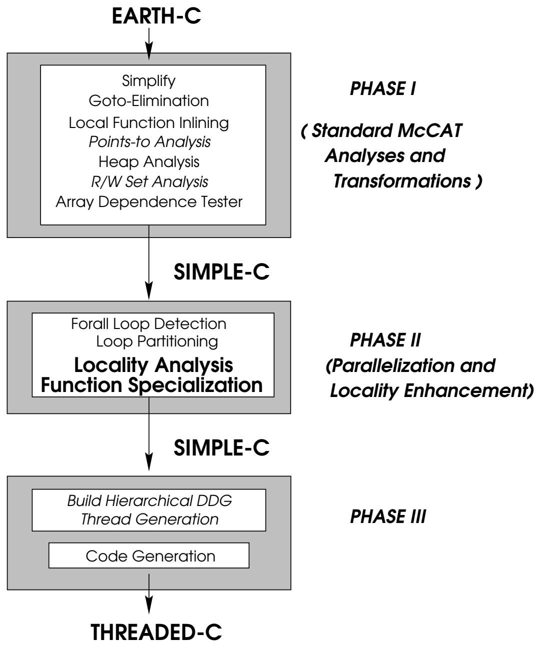


Figure 2: Overall structure of the compiler

### 2.3 The EARTH-MANNA Architecture

In the EARTH model, a multiprocessor consists of multiple EARTH nodes and an interconnection network [2, 3]. As illustrated in Figure 3, 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 shared 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.

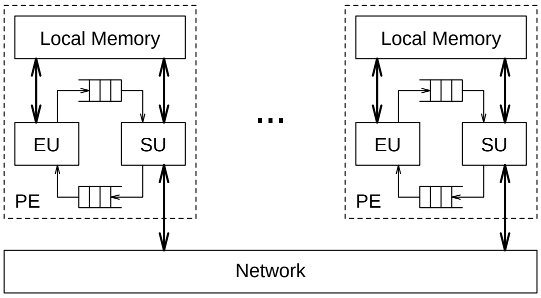


Figure 3: The EARTH architecture

Our experiments have been performed on a multi-threaded emulator built on top of the MANNA parallel machine[5]. 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. Sequentially, loading a remote word takes about $7\mu$s, calling a remote function can be performed in $9\mu$s, and spawning a new remote thread takes about $4\mu$s. When issued in a pipeline these operation take only one third of these times.

## 3 Motivating Examples

In the preceding section (Figure 1) we presented an example where locality analysis could be used to make specialized versions of the `equal_node` function. In this section we present some more typical examples of where locality information is used in EARTH-C programs, and we show how locality analysis and specialization can lead to better programs. These examples should give the intuitive ideas behind the actual locality analysis as presented in Section 4. In each of the example programs, all remote variable references are underlined. Thus, a program with fewer underlined references exhibits more locality, and will be more efficient.

### 3.1 Pointers to Local Variables and Parameters

As outlined in Section 2.1, the underlying EARTH runtime system maps a function's activation frame to the processor executing the invocation, and thus it is safe to assume that parameters and locally-scoped variables are references to local memory. This assumption can be extended to pointer variables, if it can be shown that the pointer must point to locally-scoped variables and/or parameters. Figure 4(a) gives a somewhat contrived example that serves to illustrate the basic point. In function `foo`, pointer `p` points-to `x`, and `x` is a parameter. Since all parameters are allocated in local memory, it is safe to assume that `*p` points to local memory. Pointer `q` points either to parameter `x`, or to locally-scoped variable `y`. Since both `x` and `y` are local, we can assume that `*q` is local as well. Figure 4(b) gives the localized version of function `foo`. Note that the indirections `*p` and `*q` are remote (underlined) references in the original version of `foo`, but are local references in the localized version.

```
int foo(int x)
{ int y, *p,
  int *q;
  y = expr();
  p = &x;
  if (expr())
    q = p;
  else
    q = &y;
  return( *p + *q );
}
```
(a) no locality inference

```
int foo(int x)
{ int y, local *p;
  int  local *q;
  y = expr();
  p = &x;
  if (expr())
    q = p;
  else
    q = &y;
  return(*p + *q);
}
```
(b) after locality inference

Figure 4: Locality for Pointers

### 3.2 Dynamic Memory Allocation

A rich source of locality information comes from the fact that dynamic memory allocation always allocates memory on the processor from which the allocator is called. Thus, if function `f` calls a memory allocation function like `malloc`, then the memory returned by `malloc` is local within the body of `f`. Consider the example in Figure 5(a). Without any locality inference, or type declarations, the compiler must assume that pointer `t` may refer to remote memory. Thus, as indicated by the underlined sections, all indirect references via `t` must be assumed to be possibly remote. However, one can note that `t` only points to the memory returned from `malloc`, and thus `t` can safely be declared as a local pointer, as illustrated in Figure 5(b). In this case, all memory accesses in the body of `alloc_point` can be assumed to be local.

### 3.3 Mapping Computation to the `OWNER_OF` Data

The most common kind of locality information comes from the programmer mapping function invocations to the owner of a piece of data, using the `@OWNER_OF` expression. A typical example is given in Figure 6(a). The function `count_equal` recursively descends through a binary tree `t`, counting the number of nodes with value `v`. The first recursive call, to the left sub-tree, is not explicitly mapped to any particular processor, and so there is no locality information for it. However, the second recursive call is explicitly sent to the owner of the right sub-tree. This means that these invocations can assume that all references via pointer `t` are local. As illustrated in Figure 6(b), to express this properly, a specialized copy of `count_equal` must be created (called

| | |
|---|---|
| <pre>node *alloc_point(double x, double y, int colour)<br>{  node * t;<br>   t = malloc(sizeof (node));<br>   if (t != NULL)<br>     { t->x  = x;   t->y  = y;<br>       t->colour  = colour;<br>     }<br>   return(t);<br>}</pre> (a) no locality inference | <pre>node *alloc_point(double x, double y, int colour)<br>{  node local *t;<br>   t = malloc(sizeof (node));<br>   if (t != NULL)<br>     { t->x = x;  t->y = y;<br>       t->colour = colour;<br>     }<br>   return(t);<br>}</pre> (b) after locality inference |

Figure 5: Locality for Dynamic Memory Allocation

count_equal_spec in the example), and in that copy the parameter t is declared to be a local pointer, and thus all memory accesses in the body are local.

### 3.4 Mapping Computation to HOME

Another common method for mapping computation to specific processors is to use function calls of the form f(...)@HOME. This indicates that f should be invoked on the processor executing the call. From a locality standpoint, this gives us two kinds of information. First, if f returns a pointer value that is local within f, it must also be local within the body of function calling f. Second, if an argument to f is local in the caller, then the corresponding parameter must be local in the body of f.

Figure 7(a) gives an example, and Figure 7(b) gives the result of applying locality analysis. First note that we can infer that t is a local pointer in the function newnode using the ideas presented for dynamic allocation given in section 3.2. Thus, the two calls to newnode in f must return local pointers, and both p and q must be local pointers. Now, consider the call to lessthan in f. Since both arguments, p and q, are local pointers, the corresponding formals, a and b, in the body of lessthan must also be local pointers.

## 4 Locality Analysis

In the last section, we identified the language/program features which are the sources of locality information. In this section we present the complete algorithm and associated analysis rules for locality analysis. The overall algorithm is presented in Figure 8. It works iteratively in two inter-related intra- and inter-procedural steps.

At the beginning of the analysis, all the functions in the program are considered as candidates for specialization and put in the set spclPool. Further, the locality attribute for all formal parameters and global variables is initialized to Remote, unless the programmer has given explicit local pointer declarations. For pointers explicitly declared as local in the program, the locality attribute is initialized to Local. After this initialization, the analysis proceeds in the following two steps:

Step I: This step individually analyzes each function in the pool of functions to be specialized (spclPool). It starts with the current locality attribute of variables, and propagates this information throughout the procedure using a flow-insensitive intraprocedural approach. The details of this step are given in Subsection 4.1.

Step II: This step performs interprocedural propagation of locality, and procedure specialization when applicable. It looks at each call site in the functions belonging to spclPool, which is called with either HOME or OWNER_OF primitive. Based on the locality information at the call-site, it infers locality information for formal parameters of the callee function as illustrated in sections 3.3 and 3.4. If a specialized version of the callee function with this locality already exists, the call-site is modified to invoke this function instead. Otherwise, a newly specialized version of the callee function is created for the given call-site. The locality attributes of the parameters of the newly-created function are appropriately initialized. As call-sites within the specialized function can trigger further specializations, the newly-created function is put in spclPool.

At the end of this step, if spclPool is non-empty we go back to the first step. Clearly this process will terminate because we have only a finite number of functions/parameters that can be specialized, and specializations always add locality information.

In the actual implementation, we just create a new locality context to represent a specialized function, and we do not actually create the complete new function. The decision to actually create specialized functions is taken after the analysis, depending upon the benefit achievable from a particular specialization. The details of the specialization step are given in Subsection 4.2.

### 4.1 Intraprocedural Locality Propagation

We perform intraprocedural propagation of locality information using type-inference techniques [6], which have been previously adapted to perform almost linear points-to analysis[7]. The basic idea of the type inference algorithm is to partition program variables into a set of equivalence classes. To achieve this classification, a merging-based approach is used.

For example, a simple assignment x = y, leads to assignment of the same type class to variables x and y, or in more general terms merging of the current type classes of x and y. If one wants to collect points-to information instead, the assignment would lead to merging of the points-to classes of variables x and y, where the points-to class of a variable contains the set of locations it may point to at runtime. Fast union/find data struc-

```
int count_equal(tree *t, int v)
  { int c1, c2, c3;
    if (t == NULL)
      return(0);
    else
      { {^ c1 = count_equal( t->left,v);
           c2 = count_equal( t->right,v)
                   @OWNER_OF( t->right);
           c3 = t->value == v;
        ^}
        return(c1+c2+c3);
      }
  }
```

(a) no locality inference

```
int count_equal(tree *t, int v)
  { int c1, c2, c3;
    if (t == NULL)
      return(0);
    else
      { {^ c1 = count_equal( t->left,v);
           c2 = count_equal_spec( t->right,v)
                   @OWNER_OF( t->right);
           c3 = t->value == v;
        ^}
        return(c1+c2+c3);
      }
  }
 int count_equal_spec(tree local *t, int v)
  { int c1, c2, c3;
    if (t == NULL)
      return(0);
    else
      { {^ c1 = count_equal(t->left,v);
           c2 = count_equal_spec(t->right,v)
                   @OWNER_OF(t->right);
           c3 = t->value == v;
        ^}
        return(c1+c2+c3);
      }
  }
```

(b) after locality inference and specialization

Figure 6: Locality generated using OWNER_OF

tures can be used to make merging fast. This is the technique used by Steensgaard [7].

To collect locality information, we enhance this technique by attaching an additional locality attribute with each points-to class. The locality attribute can have one of the three possible values: (i) $\perp$: indicating that the locality information is not yet determined, (ii) Local: all locations are *definitely* allocated on *local memory*, and (iii) Remote: some locations *may be* allocated on *remote memory*. When two points-to classes are merged, the new locality attribute is obtained by merging the locality attributes of the two classes using the merge operator $\bowtie$ defined as follows:

| $\bowtie$ | $\perp$ | *Local* | *Remote* |
|---|---|---|---|
| $\perp$ | $\perp$ | *Local* | *Remote* |
| *Local* | *Local* | *Local* | *Remote* |
| *Remote* | *Remote* | *Remote* | *Remote* |

Below we give a small program fragment, and the points-to and locality information obtained by the type-inference based algorithm for it.

```
{ int *a, *b, *c;
  int x, y, z;
S1: if (cond)
       a = &x;
     else
       b = &y;
S2: a = b;
S3: c = &z;
}
```

The points-to and locality information at different program points is as follows:

| | |
|---|---|
| After S1 | PointsToClass(a) = {x, Local}<br>PointsToClass(b) = {y, Local}<br>PointsToClass(c) = {Empty, $\perp$}. |
| After S2 | PointsToClass(a) = {x, y, Local}<br>PointsToClass(b) = {x, y, Local}<br>PointsToClass(c) = {Empty, $\perp$}. |
| After S3 | PointsToClass(a) = {x, y, Local}<br>PointsToClass(b) = {x, y, Local}<br>PointsToClass(c) = {z, Local} |

The locality attribute is Local for all three pointers as they contain addresses of local variables. One can also note that the information provided is flow-insensitive, and there is no kill information (otherwise x should not be in the points-to class of a or b after statement S2). Thus the final information after S3 is conservatively valid for the entire program fragment.

Our locality analysis uses the above type-inference based algorithm in an intraprocedural setting. The focus of the analysis is on accurately computing locality attributes, and not on computing complete points-to information. Thus, our analysis does not account for points-to information that holds due to aliasing between parameters and globals. However, since we make the worst case assumption about the locality of parameters and globals, this loss of information does not affect the correctness of our technique. Further, we have found that this loss of information does not affect the quality of the locality information that we find. Thus, it appears that this very inexpensive intraprocedural propagation is a good choice.

In the following subsections we provide detailed rules for the intraprocedural locality analysis. Our analysis is performed at the SIMPLE intermediate representation of the EARTH-McCAT compiler. The SIMPLE representation provides eight basic statements that can affect points-to/locality information. Below, we provide

| (a) no locality inference | (b) after locality inference |
|---|---|
| <pre>void f()<br>{ node *p, *q;<br>  p = newnode(expr1())@HOME;<br>  q = newnode(expr2())@HOME;<br>  ...<br>  if (lessthan(p,q)@HOME)<br>    { ... }<br> }<br>node *newnode(int val)<br> { node *t;<br>   t = malloc(sizeof (node));<br>   if (t != NULL)<br>     t->value = val;<br>   return(t);<br>  }<br>int lessthan(node *a, node *b)<br> { return( a->val < b->val );<br> }</pre> | <pre>void f()<br>{ node local *p, local *q;<br>  p = newnode(expr1())@HOME;<br>  q = newnode(expr2())@HOME;<br>  ...<br>  if (lessthan(p,q)@HOME)<br>    { ... }<br> }<br>node *newnode(int val)<br> { node local *t;<br>   t = malloc(sizeof (node));<br>   if (t != NULL)<br>     t->value = val;<br>   return(t);<br>  }<br>int lessthan(node local *a, node local *b)<br> { return(a->val < b->val);<br> }</pre> |

Figure 7: Locality generated using `HOME`

locality analysis rules for each statement.

#### 4.1.1 Address Assignment

`x = &y`: Variable `x` points to variable `y`, so we merge the points-to class of `x` with the class to which variable `y` belongs.

```
CreateClass(y)
if (IsGlobal(y))
   Locality(Class(y)) = Remote
else
   Locality(Class(y)) = Local
Merge(PointsToClass(x), Class(y))
```

#### 4.1.2 Dynamic Allocation

`x = malloc()`: For each statement S containing a malloc() call (or a related memory allocation call), we create a new variable called HeapS, and also create a class for it. The locality attribute of this class is initialized as Local. This is done because the EARTH programming model, requires a malloc call to always be allocated memory from the local processor. After creating this new class, we merge it with the points-to class of variable `x`, giving the following rule:

```
HeapS = CreateHeapVar(S)
CreateClass(HeapS)
Locality(Class(HeapS)) = Local
Merge(PointsToClass(x), Class(HeapS))
```

#### 4.1.3 Pointer Assignments

The statements belonging to this category include `x = y`, `x = y op z`, `x = *y`, `*x = y`, and the rules for analyzing them are discussed below.

For the statement `x = y op z`, we merge `x` with both the operands `y` and `z` on right-hand side. This is conservatively safe, and the result does not depend on the operation being performed, or the type of the operands.

For the statement `*x = y`, we need to follow an additional level of indirection on left-hand side. We need to know what `x` points-to to perform the appropriate mergings, i.e. find the points-to-class of the points-to-class of `x`. If such a class does not yet exist, we simply create such a class, which gets filled in as the analysis proceeds. The same argument applies to statement `x = *y` with respect to its right-hand side. The following table summarizes these rules.

| | |
|---|---|
| `x = y` | Merge(PointsToClass(x), PointsToClass(y)) |
| `x = y op z` | Merge(PointsToClass(x), PointsToClass(y))<br>Merge(PointsToClass(x), PointsToClass(z)) |
| `x = *y` | Merge(PointsToClass(x),<br>PointsToClass(PointsToClass(y))) |
| `*x = y` | Merge(PointsToClass(PointsToClass(x)),<br>PointsToClass(y)) |

#### 4.1.4 Function Calls

`x = f(a1,...,an)[@expr]`: A function call can considerably affect locality information. By using pointer arguments and global variables, it can modify the locality attribute, of any set of points-to classes. To avoid always making worst-case assumptions at function calls, locality analysis uses the results of interprocedural read-write (mod/ref) analysis, which is computed by our read-write set analysis in Step I of the compiler (refer to Figure 2 in Section 2.2). Based on the read-write information we have two important cases:

Case I: The function call does not write to any pointer variable visible in the caller (including globals). This guarantees that the call does not affect the points-to and hence also locality information in the caller. In this case the statement statement `x = f(a1,...,an)[@expr]`, can only affect the points-to relationship/locality attribute of variable `x`. The locality attribute of the points-to class of `x` is updated depending upon the locality attribute of the points-to class of `return_f` (`return_f` is a symbolic name which represents the value returned by a function `f`), and the optional `@expr` used for the call.

If the function `f` is a basic function, or called `@HOME`, and `f` returns a pointer (`return_f`) pointing to Local, i.e. Locality(PointsToClass(`return_f`)) is Local, it implies that the location pointed to by `return_f` resides

```
fun locality_analysis(prog) =
  spclPool = allFuncs(prog); /* need to analyze all functions initially */
  initialize_locality(prog);
  while (spclPool != Empty) /* pool of functions to be analyzed is non-empty */
   propagate_locality(spclPool);

fun propagate_locality(prog, spclPool) =
  foreach func in spclPool /* intraprocedural propagation */
    propagate_intraprocedural_locality(func);
    deletFromSplcPool(func, spclPool); /* no need to be analyzed again */
  foreach callSite in prog /* interprocedural propagation */
    callSpecialized = specializeCall(callSite);
    if (callSpecialized) /* if a newly specialized function is created */
      addToSpclPool(callSite.func, spclPool);/*new func needs to be analyzed*/

fun propagate_intraprocedural_locality(func) =
  localitySet = func.localitySet;
  foreach assignmentStmt in func
   localitySet = locality_analyze_stmt(assignmentStmt, localitySet);
  foreach callStmt in func /* handle function calls */
   localitySet = locality_analyze_call(callStmt, localitySet);
  func.localitySet = localitySet;

fun specializeCall(callSite)
  if (callSite.type == @HOME || callSite.type == @OWNER_OF)
   /* find which params will be Local in the callee function */
   funcLocality = mapParams(callSite, callSite.func, callSite.func.localitySet);
   /* if a specialzed version already exists with same locality set */
   if (specialFuncExists(callSite.func, funcLocality))
    callSite.func = specialFunc(callSite.func, funcLocality); /* rename function call */
    return 0; /* no new function is created */
   else
    callSite.func = createSpclFunc(callSite.func); /* new func for callSite */
    callSite.func.localitySet = funcLocality;/*initial locality for new func */
    return 1; /* a new specialized function is created */

fun initialize_locality(prog) =
  foreach func in prog
   /* conservatively assume all parameters and globals point-to remote memory */
   initLocality = setParmsGlobalsRemote(func);
   func.localitySet = initLocality;
```

Figure 8: Overall Algorithm for Locality Analysis

on the same processor. In this case, the new locality attribute of points-to class of **x** is obtained by merging it with Locality(PointsToClass(**return_f**)). Otherwise it is simply assigned Remote, as per the rule below.

```
if (expr == HOME Or IsBasicFunc(f))
  Locality(PointsToClass(x)) ⋈=
    Locality(PointsToClass(return_f))
else
  Locality(PointsToClass(x)) = Remote
```

Case II: The other alternative is that the function call possibly writes to pointer variables of the caller. In this case we make worst-case assumptions and set the locality attribute of points-to classes of all arguments as Remote. We do it recursively for points-to classes of variables reachable via indirection on arguments : i.e. we also set Locality($PointsToClass^*$(arg)) as Remote. We do not need to consider globals here as we already initialize their locality attribute as Remote at the start of the analysis.

### 4.2 Specializing Functions

After completion of Step II of the algorithm, we have computed a set of possible specializations, and the associated locality. We use static estimates of the number of remote accesses saved to decide which specialized functions to actually create. For a given locality context of a function, we compute the following data: (i) weight of the corresponding call-site that reflects its potential execution frequency, and (ii) count of the remote accesses that can be eliminated by creating the specialized function. A specialized function is created for the given locality context if we find its (Weight * Count) estimate greater than our threshold, which we set to 20 by default.

To compute the weight for call-sites, we first initialize the weight of all call sites to one. For each loop or recursion cycle, in which a call-site is embedded, its weight is multiplied by ten. The count of remote accesses saved is similarly estimated. A simple remote access saved is counted as one, while a remote access saved inside a loop is counted as ten. Further if some call sites inside the specialized function can also be specialized, we also add the number of remote accesses saved from this chain-specialization to the count.

## 5 Experimental Results

In order to evaluate our approach, we have experimented with five benchmarks from the Olden suite [8], described in Table 1. All benchmarks use dynamic data structures (trees and lists) except `quicksort`, which uses dynamically-allocated arrays. The benchmark suite is suitable to evaluate our locality analysis focused on pointers.

| 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 |
| quicksort | Parallel version of quicksort | 256K integers |
| tsp | Find sub-optimal tour for traveling salesperson problem | 32K cites |
| health | Simulates the Colombian health-care system using a 4-way tree | 6 levels and 100 iterations |

Table 1: Benchmark Programs

For each benchmark we provide results for three versions: a simple EARTH-C version, a localized EARTH-C version and an advanced EARTH-C version. Henceforth, we will refer to them respectively as *simple*, *localized* and *advanced* versions. The *simple* version implements the benchmark with the *best data distribution* that we have discovered to date for these benchmarks, and exploits locality using the `@OWNER_OF` and `@HOME` primitives. It uses neither local pointers nor basic functions for this purpose. However, it can use basic functions for performing computations. The *localized* version is the benchmark obtained by applying our locality analysis and subsequent function specialization on the simple version. This version tries to find as many local pointers as possible. The *advanced* version is the hand-coded benchmark where the user tries to optimally exploit locality using local pointers, basic functions and other possible tricks. This version is based on the best efforts of our group to produce good speedups. None of these advanced versions were implemented by the authors.

Note that all three versions use the same general dynamic data distribution. However, the generated low-level program exploits locality to different degrees. Statically, we can divide memory references into those that **must** be local, and those that **might** be remote. Each reference that must be local is translated into an ordinary C variable reference, which may be allocated to registers by the target compiler, or may be cached by the architecture. Each reference that might be remote is translated into a call to the EARTH runtime system. These calls to the runtime system may be resolved into memory requests to local memory, or to remote memory, depending on the calling context. If at runtime, it resolves to a local memory reference, then we call this a *pseudo-remote* memory access. Note that *pseudo-remote* references are much more expensive than local references, but not as expensive as real remote references that must read or write data over the communication network.

Our locality analysis and specializations effectively introduce more static declarations for local pointer variables, and thus at runtime we execute fewer *pseudo-remote* references and more local references. As explained in the previous sections, this is done by automatically introducing local pointer declarations, and by introducing specialized versions of functions that capture locality for specific calling contexts.

In Table 2 we summarize the static effect of applying our locality analysis and specializations to the simple versions of our benchmarks. For each benchmark, the first two columns list the number of local pointer declarations introduced, and the number of function specializations made in producing the localized version of the benchmark. The third column gives the relative sizes (lines of code) of the simple and advanced versions. Note that simple versions are all shorter, and sometimes substantially shorter than the advanced, hand-specialized programs.

| Benchmark | #locals | #spcls | $\frac{size(Simple)}{size(Advanced)}$ |
|---|---|---|---|
| power | 41 | 3 | .99 |
| perimeter | 17 | 2 | .89 |
| quicksort | 5 | 1 | .96 |
| tsp | 27 | 2 | .71 |
| health | 10 | 7 | .62 |

Table 2: Static Measurements

In Table 3 we provide data on the actual execution time, in milliseconds, for all three versions of each benchmark. These experiments were performed on the EARTH-MANNA architecture as described in Section 2.3. The last two columns give the percentage speedup obtained over the simple version for the localized and advanced versions respectively. For example, the column labeled "Localized vs. Simple" reports $(T_{simple} - T_{localized})/T_{simple} * 100$, and the column labelled "Advanced vs. Simple" reports $(T_{simple} - T_{advanced})/T_{simple} * 100$. We provide data for benchmark runs on 1, 8 and 16 processors.

In Table 4 we present the actual number of remote data accesses and remote calls performed by different versions of each benchmark. The last two columns of this table give the percentage reduction in the number of remote data accesses/remote calls over the simple version for the localized and advanced versions respectively.

| Benchmark | | Simple EARTH-C (msec) | Localized EARTH-C (msec) | Advanced EARTH-C (msec) | Localized vs. Simple (% impr) | Advanced vs. Simple (%impr) |
|---|---|---|---|---|---|---|
| power | 1 proc | 67158.06 | 64659.42 | 63482.45 | 3.72 | 5.47 |
| | 8 procs | 9132.92 | 8846.54 | 8651.86 | 3.14 | 5.27 |
| | 16 procs | 4829.12 | 4681.82 | 4568.35 | 3.05 | 5.39 |
| perimeter | 1 proc | 7095.55 | 5966.37 | 5255.03 | 15.91 | 25.90 |
| | 8 procs | 1220.71 | 894.86 | 872.59 | 26.70 | 28.50 |
| | 16 procs | 748.31 | 546.23 | 523.32 | 27.00 | 30.06 |
| quicksort | 1 proc | 15913.07 | 3031.04 | 2877.34 | 80.90 | 81.90 |
| | 8 procs | 4118.75 | 810.64 | 795.53 | 80.30 | 80.60 |
| | 16 procs | 3420.16 | 664.22 | 650.31 | 80.60 | 80.90 |
| tsp | 1 proc | 31546.85 | 29605.24 | 25543.91 | 6.15 | 19.00 |
| | 8 procs | 5394.66 | 5020.13 | 4587.74 | 6.94 | 15.00 |
| | 16 procs | 4202.43 | 3899.37 | 3288.15 | 7.20 | 21.80 |
| health | 1 proc | 104534.88 | 92931.51 | 7972.67 | 11.10 | 92.30 |
| | 8 procs | 17193.55 | 15104.78 | 1116.16 | 12.10 | 93.50 |
| | 16 procs | 10678.95 | 9386.62 | 660.00 | 12.10 | 93.80 |

Table 3: Execution Time

Note that the number of remote accesses performed is independent of the number of processors used for a given EARTH-C program, when we do not differentiate between a real remote access and a pseudo-remote access.

The data in Tables 3 and 4 indicates that the localized version always performs better than the simple version, i.e. our locality analysis is always able to identify some additional locality. Further, the percentage improvement can vary a lot, depending upon the benchmark. The localized version comes very close to the advanced version for the first three of the five benchmarks. In the last two cases the localized version does give an improvement, but does not compete with the hand-coded advanced version. We analyze these results in detail individually for each benchmark below.

**Power:** This benchmark implements the power system optimization problem [9]. It uses a four-level tree structure with different branching widths at each level.

Our locality analysis achieves 3–4% improvement over the simple version in execution time, which is quite close to the advanced version (5%). However, it is able to achieve an 80% reduction in the number of remote data accesses (Table 4). This happens because function calls in this benchmark are typically of the format `compute_node(node)@OWNER_OF(node)`, and the function performs numerous scalar data accesses of the form `(*node).item`. Our analysis captures the locality of the pointer `node` and eliminates all remote data accesses with respect to it.

The significant reduction in remote accesses does not reflect in the execution time, as the benchmark spends most of the time in performing floating point operations, which far exceeds the time spent in data accesses. The advanced version achieves 93% reduction in the number of remote calls over both the simple and localized version, by using basic functions. This factor enables it to achieve slightly better speedup than the localized version.

**Perimeter:** This benchmark computes the perimeter of a quad-tree encoded raster image [9]. 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 of each quadrant.

The localized version achieves 15–27% speedup and comes very close to the advanced version for the 8 and 16 processor runs. The localized version has 32% fewer remote accesses. This reduction more significantly affects the execution time than for `power` benchmark, because the benchmark does not involve much computation and spends most of its time in traversing the quad-tree and hence performing data accesses.

It is an irregular benchmark, and each computation requires accesses to tree nodes which may not be physically close to each other. Due to this characteristic, the advanced version cannot exploit additional locality using `basic` functions. The localized version thus competes very well with it.

**Quicksort:** This benchmark is a parallel version of the standard quicksort algorithm. The two recursive calls to quicksort are executed in parallel, with the call for the bigger subarray invoked `@HOME`. Because the size of subarrays in each recursive sorting phase is unknown in advance, dynamically-allocated arrays are used.

In the simple version, function `qsort` copies the incoming array to a `local` array using a `blkmov`, and at the end copies back the `local` array to the incoming array using another `blkmov`. Our locality analysis is able to identify the locality of the incoming array for the `@HOME` call. It generates a specialized version of the recursive `qsort` function, with the incoming array declared as local pointer, and the two `blkmov` instructions substituted by calls to the basic function `memcpy`. This transformation enables the localized version to achieve a significant 80% speedup over the simple version, which is within 1% of the speedup obtained by the advanced version. The advanced version uses some additional basic functions to completely eliminate remote calls, but performs just a little better than the localized version.

| Benchmark | Simple EARTH-C (# remote) | Localized EARTH-C (# remote) | Advanced EARTH-C (# remote) | Localized vs. Simple (% impr) | Advanced vs. Simple (% impr) |
|---|---|---|---|---|---|
| power | 2294179 | 451179 | 204179 | 80.33 | 91.10 |
| perimeter | 2421800 | 1635111 | 1586323 | 32.48 | 34.49 |
| quicksort | 8498635 | 29128 | 216 | 99.65 | 99.99 |
| tsp | 4421050 | 2672068 | 829790 | 39.56 | 81.23 |
| health | 41409726 | 33148575 | 606 | 19.94 | 99.99 |

Table 4: Remote Accesses Saved

**Tsp:** This benchmark solves the traveling salesman problem using a divide-and-conquer approach based on close-point algorithm [9]. This algorithm first searches a sub-optimal 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 spends a significant amount time on data accesses. The localized version achieves 6–8% speedup from a 39% reduction in remote data accesses.

The localized version, however, fails to compete with the advanced version, which achieves upto 20% speedup resulting from an 81% reduction in remote data accesses. This happens because the linked lists representing tours, are distributed in segments and there are only very few links across processors. With the knowledge that an entire sublist is local, the advanced version exploits significant data locality by using basic functions to traverse these local sublists. Our locality analysis is not designed to identify locality of recursive fields. This kind of locality is implicit in the programmer's organization of the data, and is very difficult to find with compiler analyses.

**Health:** This benchmark simulates the Colombian health-care system using a 4-way tree [9]. 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.

With the locality analysis, we are able to achieve up to 12% speedup, resulting from a 19% reduction in remote data accesses. This benchmark is similar to `power` in its call pattern. However, for a call of the format `foo(village_node)@OWNER_OF(village_node)`, it accesses recursive data structures through `village_node` (linked list of patients) as opposed to scalar data items like in `power`. Our analysis is able to eliminate all remote accesses with respect to the `village_node` like `list = (*village_node).list`, but not further accesses like `(*list).patient`.

Thus the localized version gets decent speedup over the simple version, but does not compete with the advanced version. With the knowledge that only top level tree nodes can have remote children, the advanced version eliminates almost all the remote data accesses and calls. In this regard, this benchmark is similar to `tsp`.

### 5.1 Summary

In summary, we find that our locality analysis does give significant improvements in all cases. However, in some cases locality analysis cannot compete with hand-coded locality mapping and declarations provided by the programmer. As the programmer may have some implicit knowledge of data locality, it is important to retain the ability to explicitly declare local pointers.

Since in many cases the declarations can be automated, and the locality analysis and specialization is a source-to-source transformation, one could imagine that the programmer could use the output of the compiler to produce a localized version of the program, and then test the program to see if acceptable speedup is achieved. If there is more locality in the program, then the programmer could add further locality declarations in order to further improve the program. Finally, note that our experiments are performed on EARTH-MANNA. Compared with other distributed memory systems like the IBM SP2, or a network of workstations, EARTH-MANNA has a much smaller memory latency, which sometimes can be further hidden by multi-threading techniques. Therefore, we can expect better speedup from locality analysis for those machines with larger memory latencies and those that do not support multi-threading techniques.

## 6 Related Work

Our intraprocedural locality propagation approach is similar to Steensgaard's [7] linear points-to analysis algorithm. Other related work in the area of parallelizing programs with dynamic data structures is that of Carlisle [9] for distributed memory machines, and of Rinard and Diniz [10] for shared memory machines. In particular, Carlisle's [9] affinity analysis has similar goals as our locality analysis, albeit we target different kinds of locality.

Steensgaard proposed a type-inference based algorithm for points-to analysis with almost linear time complexity. His algorithm is both flow-insensitive and context-insensitive. On encountering a call-site he simply merges the formal parameters with their respective actual arguments. Thus his algorithm cannot distinguish between information arriving at a function from different calling contexts.

For our locality analysis, the calling context information is crucial. The invocation specification (`@HOME`, `@OWNER_OF`) of the call site is a major source of locality information. Thus we do not want to merge locality information arriving from different calling contexts. On the contrary, we want to create specialized versions of a function for each calling context that provides us substantial locality. To this end, we use type-inference to only propagate information intraprocedurally, and employ a different technique for interprocedural propagation as explained in section 4. However, to ensure that our intraprocedural propagation collects conservatively safe information, we need to make conservative assumptions at the start of the procedure and on encountering call-sites (using read-write set information). Thus, although our approach was originally inspired by the points-to analysis, it is really specifically tailored to capture the information most relevant to locality analysis.

Carlisle's affinity analysis is designed to exploit the locality with respect to linked fields. His analysis relies upon the information regarding the probability (affinity) that nodes accessible by traversing a linked field are residing on the same processor. If the affinity is high, then he puts runtime checks to eliminate remote accesses. However, the affinity information is not infered by the compiler, but is provided via programmer annotations.

Our locality analysis is not designed to exploit the locality achievable via linked fields (as discussed in section 5) because we wanted our analysis to be an automatic compiler analysis, and we did not want to burden the user for additional detailed information. Locality of recursive fields can be explicitly declared using local pointer declarations in EARTH-C.

## 7 Conclusions and Further Work

In this paper we have presented a locality analysis for parallel C programs, based on type-inference techniques. Our analysis tries to exploit additional fine-grain locality from the coarse-grain locality information already provided by the user and from other program characteristics like malloc sites. We evaluated its effectiveness for a set of benchmarks and found that it can eliminate significant number of pseudo-remote accesses and provide speedups ranging from a modest 4% up to 80% over the original parallel program. For several benchmarks, this speedup also comes very close to the speedup obtained by an advanced hand-coded version. Furthermore, we found that the locality analysis reduced the burden on the programmer, and allowed us to develop shorter, more general benchmarks.

Based on the encouraging results from this paper, we plan to evaluate our analysis on a wider set of benchmarks. We also plan to use flow-sensitive locality propagation techniques, that can exploit the locality of a pointer, even if it is local within only a specific section of the program. Another goal is to automatically identify basic functions.

Finally, as the locality information for linked fields can sometimes provide significant speedups (for benchmarks *health* and *tsp*), we plan to extend our analysis to capture this type of locality, using profile information, and efficiently-scheduled runtime checks.


## Acknowledgements

We gratefully acknowledge the support from people in the EARTH group, specially Prof. Guang R. Gao and Olivier Maquelin. This research was funded in part by NSERC, Canada.


## References


[1] L. J. Hendren, X. Tang, Y. Zhu, G. R. Gao, X. Xue, H. Cai, and P. Ouellet, "Compiling C for the EARTH multithreaded architecture," in *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT '96)*, (Boston, Mass.), pp. 12–23, IEEE Comp. Soc. Press, Oct. 1996.

[2] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren, "A study of the EARTH-MANNA multithreaded system," *Intl. J. of Parallel Programming*, vol. 24, pp. 319–347, Aug. 1996.

[3] O. Maquelin, G. R. Gao, H. H. J. Hum, K. B. Theobald, and X.-M. Tian, "Polling Watchdog: Combining polling and interrupts for efficient message handling," in *Proc. of the 23rd Ann. Intl. Symp. on Computer Architecture*, (Philadelphia, Penn.), pp. 178–188, May 1996.

[4] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan, "Designing the McCAT compiler based on a family of structured intermediate representations," in *Proc. of the 5th Intl. Work. on Languages and Compilers for Parallel Computing*, no. 757 in Lec. Notes in Comp. Sci., (New Haven, Conn.), pp. 406–420, Springer-Verlag, Aug. 1992. Publ. in 1993.

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

[6] F. Henglein, "Efficient type inference for higher-order binding-time analysis," in *Functional Programming and Computer Architecture*, pp. 448–472, 1991.

[7] B. Steensgaard, "Points-to analysis in almost linear time," in *Conf. Rec. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, (St. Petersburg, Flor.), pp. 32–41, Jan. 1996.

[8] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Trans. on Programming Languages and Systems*, vol. 17, pp. 233–263, Mar. 1995.

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

[10] M. Rinard and P. Diniz, "Commutativity analysis: A new analysis framework for parallelizing compilers," in *Proc. of the ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation*, (Philadelphia, Penn.), pp. 54–67, May 1996.