Static Analysis of Communication Structures in Parallel Programs

So-Yan Ho and Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan 621, R.O.C.
{syh87,naiwei}@cs.ccu.edu.tw

Abstract

Channel-based message passing provides a simple and modular communication abstraction for parallel programs. Parallel programming language CCC provides four types of channels to support various communication structures: pipes for one-to-one communication, spliters for one-to-many communication, mergers for many-to-one communication, and multiplexers for many-to-many communication. The abundance of channel types gives several merits. First, communication structures among parallel tasks are more comprehensive. Second, the specification of communication structures is easier. Third, the implementation of communication structures is more efficient. Fourth, the static analysis of communication structures is more effective. This paper describes a simple and efficient algorithm for static analysis of communication structures in CCC programs. This algorithm can be also applied to other parallel programming systems based on channel-based message passing.

1 Introduction

Message passing is one of the most commonly used communication mechanisms in parallel programs. Using message passing, programmers explicitly specify the transfer of messages among parallel tasks. There are two types of message passing schemes: *task-based* and *channel-based*. In task-based message passing, a task sends (or receives) messages to (or from) another task by directly specifying its task identification. For example,

```
send(task-id, message);
message = receive(task-id);
```

Parallel programming systems that use task-based message passing include CSP [4], Occam [5], Ada [6] and MPI [3]. In channel-based message passing, parallel tasks transfer messages among them by sending (or receiving) messages to (or from) a *channel* or *message queue*. For example,

```
send(channel-id, message);
message = receive(channel-id);
```

A parallel programming system that uses channel-based message passing is Fortran M [2].

Channel-based message passing provides a simple and modular communication abstraction for parallel programs. Channels can be viewed as communication links connecting parallel tasks. Each task can communicate with other tasks (or its environment) via channels. Each task only concerns with communication events rather than specific communication participants. This separation of communication events and communication participants can greatly enhance the modularity of parallel programs.

Task-based message passing only allows one-to-one communication pattern. Many-to-one communication pattern can be achieved by using the statement-level construct: guarded commands [4]. It is possible to support all of the four communication patterns: one-to-one, many-to-one, one-to-many, and many-to-many in channel-based message passing. However, current channel-based programming systems only support one-to-one and many-to-one communication patterns. Although

programmers can implement one-to-many and many-to-many communication patterns using one-to-one and many-to-one communication patterns, the implementations are usually very complicated and often obscure the communication structures of the program.

Parallel programming language CCC provides four types of channels: *pipes* for one-to-one communication, *spliters* for one-to-many communication, *mergers* for many-to-one communication, and *multiplexers* for many-to-many communication. The abundance of channel types gives several merits. First, communication structures among parallel tasks are more comprehensive. Second, the specification of communication structures is easier. Third, the implementation of communication structures is more efficient. Fourth, the static analysis of communication structures is more effective.

To allow dynamic communication structures, channels can be passed via other channels among parallel tasks in CCC programs. We will call a channel *first-order* if it is used to pass data. A channel is *higher-order* if it is used to pass channels. The inclusion of higher-order channels makes the static analysis of communication structures much more complicated. This paper will describe a simple and efficient algorithm for static analysis of communication structures in CCC programs. This algorithm can be also applied to other parallel programming systems using channel-based message passing.

The remainder of this paper is organized as follows. Section 2 summarizes the terminologies used in this paper and gives an overview of the analysis. Section 3 describes the analysis for first-order channels. Section 4 describes the analysis for higher-order channels. Finally, conclusions are given in Section 5.

2 Overview of the Analysis

The *communication structure* of a parallel program presents the *interaction relationships* among parallel tasks in the program. The abundance of channel types in CCC provides a better abstraction to closely related interactions. In particular, each channel is used to abstract a collection of closely related interactions. CCC provides four types of channels: **pipe** for one-to-one communication, **spliter** for one-to-many communication, **merger** for many-to-one communication, and **multiplexer** for many-to-many communication. Each channel is used to transfer data of a specific type and is created using the function **channel**. A simple CCC program for a single-producer and multiple-consumer application is shown in Figure 1.

```
/* A simple CCC program */
task::main()
{
    spliter int ch;

    ch = channel();
    par {
        producer(ch);
        consumer(ch);
        consumer(ch);
    }
}

task::producer(ch)
spliter int ch;
{
    int i;

    for (i = 0; i < num_data; i++)
        send(ch, i);
    send(ch, end_data);
    send(ch, end_data);
}

task::consumer(ch)
spliter int ch;
{
    int data;

    while (1) {
        data = receive(ch);
        if (data == end_data) break;
        process(data);
    }
}
```

Figure 1. This simple CCC program illustrates a single-producer and multiple-consumer application.

A CCC program consists of a set of functions. A function in CCC can be either a sequential or a parallel function. Each sequential function is a traditional C function and each parallel function represents a *task* that can be created and executed in parallel. Each task in CCC is specified by the keyword **task**. This example program has three task definitions. The **main** task is the initial task of the program and it creates a **producer** task and two **consumer** tasks. Tasks are created in the **par** construct via invocations of parallel functions. A channel of type **spliter** is used to transfer data of

type **int** from the **producer** task to the two **consumer** tasks. Data are sent using the function **send** and received using the function **receive**. The **producer** task sends an **end_data** to **consumer** tasks to signal the termination of data transfer. In addition to the interactions via the parameter passing between the **main** task (the caller) and the other tasks (the callees), this program contains interactions between the **producer** task and the two **consumer** tasks via the channel **ch** of type **spliter**. These interactions among tasks consist of the communication structure of this program.

Since channels are dynamically created, even with only first-order channels, the communication structure of a parallel program is a *dynamic* property of the program. Different runs may result in different communication structures. This paper will present an algorithm for static analysis of communication structures for CCC programs. In CCC programs, each channel is typed according to the number of participants involving in the communication. This algorithm will infer the number of senders and receivers for each channel. This information can then be used to check whether each channel is used as it is specified. In CCC programs, the number of participants we are interested in are in the domain {0, 1, 2}, where 2 is used to represent *many*.

The analysis can be divided into two parts: intraprocedural analysis, the analysis for the effects induced inside each function, and interprocedural analysis, the analysis for the effects propagated among functions. The algorithm for the analysis is illustrated in Figure 2.

The intraprocedural analysis is relatively simple. Since data transfers occurring within a function itself must appear in the same task, we only need to know whether a data transfer to (or from) each channel occurs in the function. It is not necessary to know where and how many times data transfers to (or from) each channel occur. Hence, the information about whether a data transfer to (or from) each channel occurs within each function *itself* can be gathered by traversing the program structure once and for all.

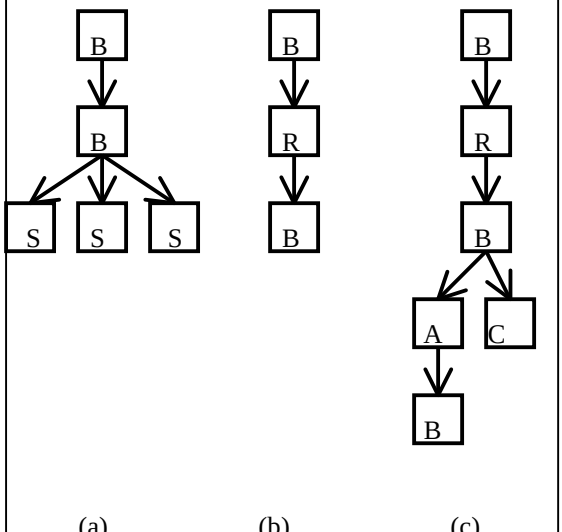
The interprocedural analysis is much more complicated than the intraprocedural analysis. Each parallel function call explicitly creates a task. Although a sequential function call does not explicitly create a task, it may implicitly create tasks by indirectly calling other parallel functions. Hence, both sequential and parallel function calls may have significant effects. However, they still need to be dealt with in different ways. Function calls occurring inside different control structures may result in different effects too. For example, function calls occurring inside the alternative control structure can only be executed mutually exclusively, while function calls occurring inside the repetition control structure may be executed multiple times. Thus, function calls occurring inside different control structures also need to be handled in different ways.

```
analysis() {
  for each function f begin
    intraprocedure_analysis(f);
  endfor
  /* interprocedural analysis */
  for each SCC s in SCCG begin
    while changed begin
      for each CCFG g in s begin
        for each node n in g begin
          node_analysis(n);
        endfor
      endfor
    endwhile
  endfor
}
```

Figure 2. The algorithm for the static analysis of communication structures.

The interprocedural analysis is performed on two program structures: a *compact control flow graph* (CCFG) for each function and a *call graph* (CG) among functions. There are five kinds of nodes in the compact control flow graphs. A *call node* represents a call to a sequential function. A *spawn node* represents a call to a parallel function. A *block node* represents a sequence of nodes. An *alternative node* represents a list of nodes that are mutually exclusive. A *repetition node* represents a node that may be executed multiple times. The three kinds of control nodes: block, alternative, and repetition nodes are used to characterize the control structures of (sequential or parallel) function calls within a function. The compact control flow graphs for the functions in the program in Figure 1 are depicted in Figure 3.

The call graph presents the caller-callee relationships among functions. Each node in the call graph represents a function (either a sequential or a



(a) main (b) producer (c) consumer

Figure 3. The CCFG for the functions in Figure 1.

parallel function). There is an edge from node *n* to node *m* if function *n* calls function *m* in the program. If there is no mutual recursion in the program, the call graph is a directed acyclic graph. In this case, the interprocedural analysis can be performed on each function using the topological order of the call graph. Otherwise, The *strongly connected components* (SCC) of the call graph are found and a *strongly connected component graph* (SCCG), which is a directed acyclic graph, is constructed. Each node of the strongly connected component graph represents a strongly connected component. There is an edge from node *n* to node *m* if there is a function in *n* calls a function in *m*. The nodes in a strongly connected component represent functions that are mutually recursive. These mutually recursive functions need to be analyzed at the same time. Strongly connected components are analyzed using the topological order of the strongly connected component graph. The call graph for the program in Figure 1 is depicted in Figure 4. In that program the strongly connected component graph is the same as the call graph because there is no recursion in the program. We will describe the analysis for each node in a compact control flow graph with respect to the first-order channels and the higher-order channels, respectively, in the next two sections.

3 The analysis for first-order channels

This section presents the analysis for programs that only use first-order channels. For each channel

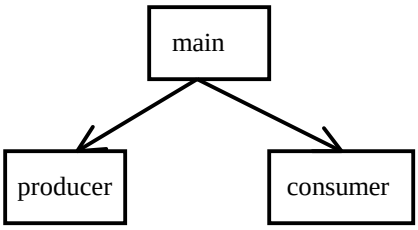


Figure 4. The CG and the SCCG for the program in Figure 1.

ch declared within a function, the analysis will infer the number of tasks that send data to **ch**, **ch.senders**, and the number of tasks that receive data from **ch**, **ch.receivers**. Since each function may create tasks by calling parallel functions, for each channel **ch**, we can divide **ch.senders** and **ch.receivers** as follows.

ch.senders = **ch.self.senders** + **ch.others.senders**
ch.receivers = **ch.self.receivers** + **ch.others.receivers**

where, **ch.self.senders** and **ch.self.receivers** denote the task executing the function and can have values {0, 1}, and **ch.others.senders** and **ch.others.receivers** denote the tasks created by the function and can have vales {0, 1, 2}. We will call the four-tuple {**ch.self.senders**, **ch.self.receivers**, **ch.others.senders**, **ch.others.receivers**} the *mode* of **ch**.

A channel variable can only be assigned a channel value by creating a channel using the function **channel** or by receiving a channel from a higher-order channel. Without higher-order channels, channels can be passed to other functions only via parameter passing. Hence, the aliasing of channels can only occur through parameter passing. The analysis will first perform a simple context-insensitive aliasing analysis to compute the channel aliases induced solely by parameter passing [1].

We consider each strongly connected component in order. For each channel **ch**, the dataflow equations for the mode of **ch** at each node of the compact control flow graph can be derived as follows.

For each block node *n*, let $\angle$ be the sequence of nodes inside *n*. Initially,

$\mathbf{ch_n.self.senders}$ = the values obtained from the intraprocedural analysis,

$\mathbf{ch_n.self.receivers}$ = the values obtained from the intraprocedural analysis,
$\mathbf{ch_n.others.senders}$ = 0,
$\mathbf{ch_n.others.receivers}$ = 0,

if n is the initial node of the compact control flow graph; otherwise,

$\mathbf{ch_n.self.senders}$ = 0,
$\mathbf{ch_n.self.receivers}$ = 0,
$\mathbf{ch_n.others.senders}$ = 0,
$\mathbf{ch_n.others.receivers}$ = 0.

Then,

$\mathbf{ch_n.self.senders} = \mathbf{ch_n.self.senders} \parallel \mathbf{\Omega_{x \in \mathcal{L}}\ ch_x.self.senders}$,
$\mathbf{ch_n.self.receivers} = \mathbf{ch_n.self.receivers} \parallel \mathbf{\Omega_{x \in \mathcal{L}}\ ch_x.self.receivers}$,
$\mathbf{ch_n.others.senders} = \mathit{min}(2, \mathbf{ch_n.others.senders} + \mathbf{\Sigma_{x \in \mathcal{L}}\ ch_x.others.senders})$,
$\mathbf{ch_n.others.receivers} = \mathit{min}(2, \mathbf{ch_n.other.receivers} + \mathbf{\Sigma_{x \in \mathcal{L}}\ ch_x.others.receivers})$,

where $\mathbf{\Omega}$ denotes *logical or* and $\mathbf{\Sigma}$ denotes *summation*.

For each call node n, let $\mathcal{A}$ be the set of channels that are aliased with a channel argument **ch** at n. Then,

$\mathbf{ch_n.self.senders} = \mathbf{\Omega_{x \in \mathcal{A}}\ x.self.senders}$,
$\mathbf{ch_n.self.receivers} = \mathbf{\Omega_{x \in \mathcal{A}}\ x.self.receivers}$,
$\mathbf{ch_n.others.senders} = \mathit{min}(2, \mathbf{\Sigma_{x \in \mathcal{A}}\ x.others.senders})$,
$\mathbf{ch_n.others.receivers} = \mathit{min}(2, \mathbf{\Sigma_{x \in \mathcal{A}}\ x.others.receivers})$.

For each spawn node n, let $\mathcal{A}$ be the set of channels that are aliased with a channel argument **ch** at n. Then,

$\mathbf{ch_n.self.senders}$ = 0,
$\mathbf{ch_n.self.receivers}$ = 0,
$\mathbf{ch_n.others.senders} = \mathit{min}(2, \mathbf{\Omega_{x \in \mathcal{A}}\ x.self.senders} + \mathbf{\Sigma_{x \in \mathcal{A}}\ x.others.senders})$,
$\mathbf{ch_n.others.receivers} = \mathit{min}(2, \mathbf{\Omega_{x \in \mathcal{A}}\ x.self.receivers} + \mathbf{\Sigma_{x \in \mathcal{A}}\ x.others.receivers})$.

For each alternative node n, let $\mathcal{L}$ be the list of nodes inside n. Then,

$\mathbf{ch_n.self.senders} = \mathit{max}_{\mathbf{x \in \mathcal{L}}}\ \mathbf{ch_x.self.senders}$,
$\mathbf{ch_n.self.receivers} = \mathit{max}_{\mathbf{x \in \mathcal{L}}}\ \mathbf{ch_x.self.receivers}$,
$\mathbf{ch_n.others.senders} = \mathit{min}(2, \mathit{max}_{\mathbf{x \in \mathcal{L}}}\ \mathbf{ch_x.others.senders})$,
$\mathbf{ch_n.others.receivers} = \mathit{min}(2, \mathit{max}_{\mathbf{x \in \mathcal{L}}}\ \mathbf{ch_x.others.receivers})$.

For each repetition node n, let m be the node inside n. Then,

$\mathbf{ch_n.self.senders} = \mathbf{ch_m.self.senders}$,
$\mathbf{ch_n.self.receivers} = \mathbf{ch_m.self.receivers}$,
$\mathbf{ch_n.others.senders} = \mathit{if}\ (\mathbf{ch_m.others.senders} == 0)\ \mathit{then}\ 0\ \mathit{else}\ 2$,
$\mathbf{ch_n.others.receivers} = \mathit{if}\ (\mathbf{ch_m.others.receivers} == 0)\ \mathit{then}\ 0\ \mathit{else}\ 2$.

For each function in a strongly connected component, the mode for each channel is computed as specified above. This computation will continue until a fixed point is reached. Strongly connected components are analyzed following the topological order of the strongly connected component graph.

4 The analysis for higher-order channels

This section presents the analysis for programs that also use higher-order channels. In such programs, a channel variable can be assigned a channel value by either creating a channel using the function **channel** or receiving a channel from a higher-order channel. With higher-order channels, the aliasing of channels can thus occur through either parameter passing or message passing. For simplicity, we will consider only second-order channels, which are used to transfer first-order channels. The approach can be extended to higher-order channels.

For each second-order channel **ch**, in addition to the mode of **ch**, the analysis will also infer the set of channels sent to **ch**, **ch.sendSet**, and the set of channels received from **ch**, **ch.receiveSet**, to handle the aliasing occurring through message passing. In intraprocedural analysis, for each function **f**, if there is a channel **x** sent to second-order channel **ch**, then (**f**, **x**) is added into **ch.sendSet**, and if there is a channel **x** received from **ch**, then (**f**, **x**) is added into **ch.receiveSet**.

In interprocedural analysis, for each second-order channel **ch**, the dataflow equations for **ch.sendSet** and **ch.receiveSet** of **ch** at each node of the compact control flow graph can be derived as follows.

For each block node n, let $\mathcal{L}$ be the sequence of nodes inside n. Initially,

$\mathbf{ch_n.sendSet}$ = the values obtained from the

intraprocedural analysis,
ch_n.receiveSet = the values obtained from the intraprocedural analysis,

if n is the initial node of the compact control flow graph; otherwise,

ch_n.sendSet = $\emptyset$,
ch_n.receiveSet = $\emptyset$.

Then,

ch_n.sendSet = **ch_n.sendSet** $\cup$ $\cup_{x \in \mathcal{L}}$ **ch_x.sendSet**,
ch_n.receiveSet = **ch_n.receiveSet** $\cup$ $\cup_{x \in \mathcal{L}}$ **ch_x.receiveSet**.

For each call node or spawn node n, let $\mathcal{A}$ be the set of channels that are aliased with a channel argument **ch** at n. Also, let $\mathcal{C}$ be the set of channel parameters at n, and for each $p \in \mathcal{C}$, let $\mu(p)$ represent the corresponding channel argument of p at n. Then,

ch_n.sendSet = $\cup_{x \in \mathcal{A}}$ **x.sendSet**$\{\forall p \in \mathcal{C}, p/\mu(p)\}$,
ch_n.receiveSet = $\cup_{x \in \mathcal{A}}$ **x.receiveSet**$\{\forall p \in \mathcal{C}, p/\mu(p)\}$.

Where $S\{\forall p \in \mathcal{C}, p/q\}$ denotes a set obtained from the set S by substituting p by q for each $p \in \mathcal{C}$ occurring in the elements of S.

For each alternative node n, let $\mathcal{L}$ be the list of nodes inside n. Then,

ch_n.sendSet = $\cup_{x \in \mathcal{L}}$ **ch_x.sendSet**,
ch_n.receiveSet = $\cup_{x \in \mathcal{L}}$ **ch_x.receiveSet**.

For each repetition node n, let m be the node inside n. Then,

ch_n.sendSet = **ch_m.sendSet**,
ch_n.receiveSet = **ch_m.receiveSet**.

For each function in a strongly connected component, **ch.sendSet** and **ch.receiveSet** for each second-order channel **ch** is computed as specified above. This computation will continue until a fixed point is reached. Strongly connected components are analyzed following the topological order of the strongly connected component graph.

Given **ch.sendSet** and **ch.receiveSet** for each second-order channel **ch**, we can then take into account the aliasing caused by message passing. For each x in **ch.sendSet**, x may be aliased with each y in **ch.receiveSet** because of message passing. Let $\alpha(x)$ be the set of z in **ch.sendSet** that are aliased with x. Then, for each c in $\alpha(x)$ or **ch.receiveSet**,

c.self.senders = $\Omega_{d \in \alpha(x) \cup \text{ch.receiveSet}}$ **d.self.senders**,
c.self.receivers = $\Omega_{d \in \alpha(x) \cup \text{ch.receiveSet}}$ **d.self.receivers**,
c.others.senders = $min(2, \Sigma_{d \in \alpha(x) \cup \text{ch.receiveSet}}$ **d.others.senders)**,
c.others.receivers = $min(2, \Sigma_{d \in \alpha(x) \cup \text{ch.receiveSet}}$ **d.others.receivers)**.

The aliasing becomes more complicated if a channel may be transferred among functions via more than one second-order channel. For example, a channel c may be sent from function f via a second-order channel ch_1 to function g, then c is passed from function g via another second-order channel ch_2 to function h. In this case, the channels in **ch_1.sendSet** may be aliased with the channels in **ch_2. receiveSet** due to the transitivity of the aliasing. Hence, if some channel in **ch_1.receiveSet** is aliased with some channel in **ch_2.sendSet**, each channel **ch_1.sendSet** in is aliased with every channel in **ch_1.receiveSet** $\cup$ **ch_2. receiveSet**.

5 Complexity analysis

This section briefly sketches he complexity analysis of this algorithm. Consider again the algorithm in Figure 2. Since the domain for the analysis is {0, 1, 2}, the number of times the while-loop will execute is bounded by a constant. Let V and E be the number of nodes and edges in the compact control flow graphs in the program. Then the complexity of the algorithm is O(V + E).

6 Conclusions

This paper has described a linear algorithm for the static analysis of communication structures in CCC programs. For each channel **ch** declared within the program, this algorithm infers the number of tasks that send data to **ch**, and the number of tasks that receive data from **ch**. This information can then be used to check whether each channel is used as it is specified. This algorithm can analyze both first-order and higher-order channels. This algorithm uses a simple approach to exploiting both the aliasing caused by parameter passing and message passing.

Acknowledgements

This work was supported in part by the National

Science Council of R.O.C. under grant number NSC-89-2213-E-194-054.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 25 (1), 1995.

[3] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1995.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1984.

[5] D. May. OCCAM. *ACM SIGPLAN Notices*, 25 (4), 1983, pp. 69-79.

[6] Y. Wallach. *Parallel Processing and Ada*. Prentice Hall, 1991.