Native handling of Message-Passing communication in Data-Flow analysis

Valérie Pascual and Laurent Hascoët

Abstract Automatic Differentiation by program transformation uses static data-flow analysis to produce efficient code. This data-flow analysis must be adapted for parallel programs with Message-Passing communication. Starting from a context-sensitive and flow-sensitive data-flow analysis scheme initially devised for sequential codes, we extend this scheme for parallel codes. This extension is independent of the particular analysis and does not require a modification of the code's internal representation, i.e. the flow graph. This extension relies on an accurate matching of communication points, which can't be found automatically in general, and thus new user directives prove useful.

Key words: Data-Flow Analysis, Activity Analysis, Automatic Differentiation, Message-Passing, MPI

1 Introduction

Static data-flow analysis of programs is necessary for efficient automatic transformation of codes. In the context of Automatic Differentiation (AD), most of the classical data-flow analyses prove useful as well as specific analyses such as activity and TBR analyses [5]. Parallel programs with message-passing pose additional problems to data-flow analysis because they introduce a flow of data that is not induced by the control-flow graph ("flow graph" for short). We propose an extension to data-flow analysis that captures this communication-induced flow of data. This extension applies in the very general framework of flow-sensitive analysis that sweep over the

Valérie Pascual
INRIA, Sophia-Antipolis, France, Valerie.Pascual@inria.fr

Laurent Hascoët
INRIA, Sophia-Antipolis, France, Laurent.Hascoet@inria.fr

flow graph, possibly using a worklist for efficiency. This extension makes no particular hypothesis on the specific analysis and only introduces new artificial variables that represent communication channels, together with a generic modification of the flow-sensitive propagation strategy.

2 Context-sensitive and Flow-sensitive Data-Flow analysis

To reach the accuracy that is necessary to generate an efficient transformed program, data-flow analysis should be context-sensitive and flow-sensitive. Context sensitivity operates at the call graph level. In a context sensitive analysis, each procedure uses a context that is built from the information available at its call sites. Even when making the choice of generalization, which means using only one context that summarizes all call sites, this context allows the analysis to find more accurate results inside the called procedure. Flow sensitivity operates at the flow graph level. In a flow-sensitive analysis the propagation of data-flow information follows an order compatible with the flow graph, thus respecting possible execution order.

Data-flow analysis works by propagating information through call graphs and flow graphs. Call graphs may be cyclic in general, due to recursivity. Flow graphs may be cyclic, due to loops and other cyclic control. Completion of the analysis requires reaching a fixed point both on the call graph and on each flow graph. The most effective way to control this fixed point propagation uses worklists [7].

In a naïve implementation a data-flow analysis of a calling procedure would require a recursive data-flow analysis of each called procedure, before the analysis of the calling procedure is completed. This would quickly cause a combinatorial explosion in run-time and in memory. To avoid that, it is wise to introduce a “relative” version of the current analysis that summarizes the effect of each called procedure on the information computed for any calling procedure. For instance in the case of Activity analysis, a variable is active if it depends on an independent input in a differentiable way (it is “varied”) and the same time it influences the dependent output in a differentiable way (it is “useful”). This results in two data-flow analyses, both top-down on the call graph: The “varied” analysis goes forward on the flow graph, and the “useful” analysis goes backward on the flow graph. When any of the two reach a procedure call, we don’t want to call the analysis recursively on the called procedure. Instead, we use a “differentiable dependency” summarized information that relates each output of the called procedure to each of its inputs on which it depends in a differentiable way. This relative information occupies more space than plain activity, typically the square of the number of visible variables, but it is easily and quickly used in place of the actual analysis of the called procedure. It takes a preliminary data-flow analysis to compute this “dependency”, which is this time bottom-up on the call graph. This strategy may have a cost: the summarized relative information may be less accurate than an explicit data-flow analysis of the callee. On the other hand combinatorial behavior is improved, with an initial bottom-up sweep on the call graph to compute the relative information, followed by a top-down sweep

to compute the desired information. Each sweep analyses each procedure only once, except for recursive codes.

3 Impact of Message-Passing on Data-Flow analysis

The above framework for data-flow analysis is originally designed for sequential programs. It does not handle message-passing communication, which introduces a new flow of data unrelated to the flow graph, and that may even apparently go backwards the static flow graph, e.g. in a SPMD context, from a `send` to a `receive` located several lines before. See also in Fig. 1 the data-flow from `MPI_SEND` to `MPI_RECV` that is unrelated to the static flow graph. The propagation algorithm must be extended to capture this additional flow of data. Little research has been done in the domain of static analysis of message-passing programs [3]. Bronewtsky [1] defines parallel control-flow graphs, an extension of flow graphs that is the finite cross-product of the flow graphs of all the processes. This a theoretical framework useful for reasoning about analyses but that does not easily lend itself to implementation in our tools.

In the context of AD, several methods have been tried to solve this problem. `Odyssée` [2] introduced fictitious global communication variables but let flow graph propagation unchanged. This alone cannot capture the effect of communication that goes against the static flow graph order, and may give incorrect results.

A more radical method is to assign the analysis' conservative default value to all variables transmitted through message-passing. This leads to degraded accuracy of data-flow results and a less efficient differentiated code that may contain unnecessary derivative computation, useless differentiated communications, or useless trajectory storage in adjoint mode. This can be compensated partly with user directives understood by the AD tool.

Strout, Kreaseck and Hovland [10] use an "interprocedural control-flow graph" and augment it with communication edges between possible `send`/`receive` pairs [9]. Heuristics keep the number of communication edges low, based on constant propagation and the `MPI` semantics. This extended data-flow analysis improves the accuracy, e.g. for activity analysis. However these extra edges in the flow graph correspond to no control and have a special behavior: only the variables involved in the communication travel through these edges.

4 Data-Flow analysis with Flow Graph local restart

We believe that introducing new global variables to represent communication channels as in [2] is an element of the solution. A channel is an artificial variable that contains all values currently in transit. However to cope with communication that goes against the flow graph we prefer to modify the data-flow propagation algorithm

rather than modifying the flow graph itself. The arrows of the flow graph really represent an execution order, and adding arrows for communication may blur this useful interpretation. Note that adding flow arrows requires an interprocedural control-flow graph. In either case, modifying the propagation algorithm or modifying the graph it runs on, this can be done in a way mostly independent from the particular analysis.

The run-time context in which a given procedure is executed contains in particular the state of the various channels. During static data-flow analysis the context in which a given procedure is analyzed is an abstraction of this run-time context, only it represents several actual run-time contexts together. Therefore this static analysis context also includes the information on channels. When analysis of a given procedure reaches a communication call that changes the status of a channel, this change must be seen by all processes running in parallel and therefore possibly by all procedures of the code. In particular the static analysis context for the given procedure must change to incorporate the new channel status, and the analysis itself must restart from the beginning of the procedure. However this restart remains local to the given flow graph, as shown by figure Fig. 1. The effect on the other procedures' analysis will be taken care of by the "relative" version of the analysis. Thus this restart, illustrated by Fig. 1, remains local to the current flow graph: after

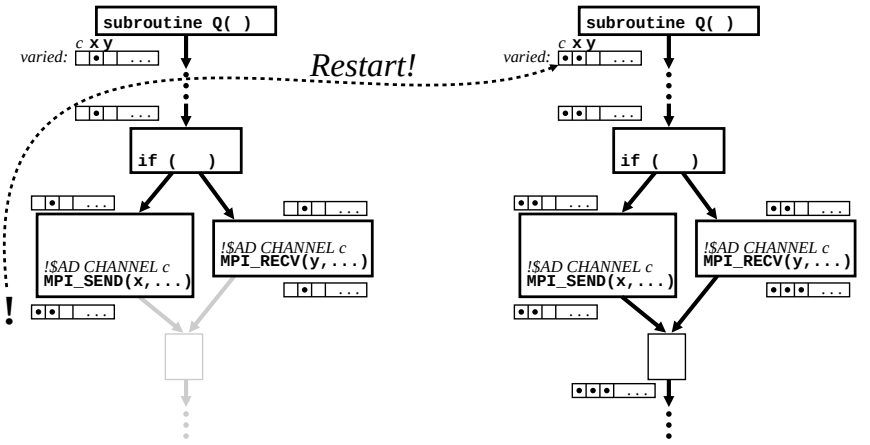

Fig. 1 Flow Graph local restart after communication, in the case of the "varied" analysis

the `MPI_SEND` is executed with a varied `x`, the artificial variable `c` that represents this particular communication channel becomes varied. The changing "varied" status of `c` restarts the current propagation from the entry of the flow graph. This new sweep, when reaching the `MPI_RECV` that reads the same channel, makes `y` varied in turn. In the frequent case when propagation order of the data-flow analysis is done with a worklist, the restart is achieved by just adding the entry block on top of the worklist, or the exit block in case of a backward propagation. This results in the framework Alg. 1, common to any forward data-flow analysis. Navigation in the

Algorithm 1 Extension of forward analysis to message-passing communication

```
Given entryInfo:
01 ∀ Block b, in(b) := ∅; out(b) := ∅
02 out(EntryBlock) := entryInfo
03 worklist := succ(EntryBlock)
04 while [worklist ≠ {ExitBlock}]
05     b := firstof(worklist)            // i.e. the element with lowest dfst index
06     worklist := worklist\{b}
07     i := ∪_{p∈pred(b)} out(p)
08     o := propagate i through b
09     if [o/channels > out(b)/channels
10             && out(EntryBlock) ≯ o/channels]
11         out(EntryBlock) := out(EntryBlock) ∪ (o/channels)
12         worklist := worklist ∪ succ(EntryBlock)
13     if [o > out(b)]
14         out(b) := o
15         worklist := worklist ∪ succ(b)
16 exitInfo := ∪_{p∈pred(ExitBlock)} out(p)
```

flow graph only needs the EntryBlock, the ExitBlock, plus the successor (succ) and predecessor (pred) sets for every block of the flow graph. Blocks are labelled with their *dfst* index, which is such that the index of a block is most often lower than the index of its successors. Actual propagation of the data-flow information through a given block is represented by the analysis-specific "propagate" operation. Operation "o/channels" builds a copy of data-flow information o that concerns only communication channels. Alg. 1 lines 01-08 and 13-16 is the usual sequential data-flow analysis. Our proposed extension is exactly lines 09-12.

Consider now the call graph level. During the bottom up computation of the "relative" analysis, every individual procedure Q is analyzed with an extended algorithm following Alg. 1, therefore taking care of channels. The relative information that is built thus captures the effect of the procedure on the channels. For instance, the relative "differentiable dependency" information for the procedure Q of Fig. 1 will contain in particular that the output values of both y and channel *c* depend on the input values of x and of channel *c*. During analysis of a procedure P that calls Q, analysis of the call to Q may modify the information attached to the channels accordingly. In other words, analysis of the call to Q has an effect similar to analysis of a call to an elementary message-passing procedure. This triggers the local restart mechanism of Alg. 1 at the level of the flow graph of P, and eventually leads to completion of the analysis inside procedure P.

5 Performance discussion

We will discuss the consequences of introducing the Flow Graph local restart on termination and execution time of the analyses. These questions partly depend on the specific data-flow analysis, and each analysis deserves a specific study. However, we saw that our proposed extension to message-passing is essentially done on the general analysis framework Alg. 1 so that some general remarks apply to any analysis.

About termination, the argument most frequently used is that the data-flow information, kept in the variables in(b) and out(b) for each block b, belong to a set of possible values that is finite, with a lattice structure wrt the partial order $>$ compatible with the union $\cup$. If one can show that propagation for the particular analysis represented by line 08:

o := propagate i through b

is such that propagation of a larger i returns a larger o, then termination is granted. This argument is still valid when we introduce the local restart. Every local restart makes out(EntryBlock) grow, so that restarts are in finite number and the process terminates.

The local restart clearly affects the execution time of the analysis. For each propagation through one flow graph, the execution time on a non parallel code depends on its nested loop structure. Classically, one introduces a notion of “depth” of the flow graph which measures the worst-case complexity of the analysis on this graph. On well-structured flow graphs, one can show that this “depth” is actually the depth of the deepest nested loop. On a code with message-passing, extra propagation is needed when the status of a channel variable changes. When the approach chosen to represent communication is to add extra flow edges, the “depth” of the flow graph changes [6]. When these new edges make the graph irreducible, evaluation of the depth even becomes NP-hard in general. Nevertheless, a reasonable rule of thumb is to add the number of communication edges to the nested loop depth to get an idea of the analysis complexity increase. With our approach, which adds no communcation edge but rather triggers restarts from the flow graph EntryBlock, the complexity effect is very similar. Local restart can occur once for each individual channel, so that the “depth” is increased by the number of channels. Not surprisingly, an increased number of channels may yield more accurate analysis results, but may increase the analysis time. In practice, this slowdown is quite tolerable. To be totally honest, local restart incurs some redundant propagation compared to [10]: since restart is done from the EntryBlock rather than from the destinations of communication, it goes uselessly through all blocks between the EntryBlock and these destinations. However, this does not change the degree of complexity.

For propagation through the call graph, though, the number of times one given procedure is analyzed does not change with message-passing and still depends only on the structure of recursive calls. The restarts are local to each flow graph, and do not change the behavior at the call graph level. To summarize, the local restart method introduces an extra complexity only into the data-flow analysis of procedures that call message passing communication, directly or indirectly. However,

after implementation of the local restart into all the data-flow analyses of the AD tool `Tapenade`, we observe no particular slowdown of the differentiation process.

6 Choosing a good set of channels

Channel extraction depends on the message-passing communication library, in our case we use the `MPI` library [4, 8]. Collective communication functions such as broadcast do not need channels as all message-passing communications are done in one function call. We just have to focus on point-to-point communications functions.

We first define a test to match send's with receive's. For `MPI` point-to-point communication, this matching uses the source and destination, plus when possible the "tag" and "communicator" arguments of the message-passing function calls. If the communicators are identical, if the source and destination processes correspond, and if finally the tags may hold the same integer value, then the send and the receive match, which means that a value may travel from the former to the latter. The quality of this matching, i.e. the lowest possible number of false matches found, clearly depends on the quality of the available static constant propagation. Expressed in terms of channels, a match just means that there must be at least one defined communication channel that is common to the send and the receive.

Unfortunately, this matching depends on many parameters, and these are often computed dynamically in a way that is untractable statically, even with a powerful constant propagation. Static detection of matching send and receives will most often find too many matches, and we'd better resort to the user's knowledge of the code. This is done with a directive that the user can place in the code to designate explicitly the channel(s) affected by any communication call.

This preliminaries done, we end up with a bipartite graph that relates the send's to the matching receive's. We shall use Fig. 2 as an illustration. The question is to

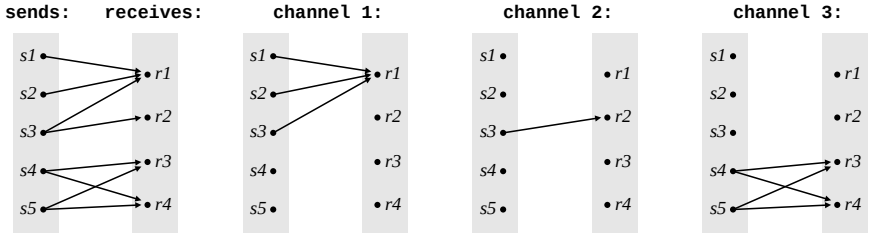

Fig. 2 A minimal biclique edge cover of a communication bipartite graph

find a good set of channels that will exactly represent this communication bipartite graph:

- First, a good set of channels must not introduce artificial communication. On Fig. 2, we see we must not use a single channel to represent communications between $s1, s2, s3$ to $r1, r2$, because this would imply e.g. a spurious communication from $s2$ to $r2$. The rule here is that the bipartite subgraph induced by nodes that share a given channel must be complete.
- Second, a good set of channels must be as small as possible. We saw that the number of channels conditions the extra complexity of the analyses. In particular, the trivial choice that assigns one new channel for each edge of the bipartite graph is certainly correct, but too costly in general. On Fig. 2, we could introduce two channels for the two edges $(s1, r1)$ and $(s2, r1)$, but one channel suffices.

This question is already known as the "minimal biclique edge cover", a known NP-complete problem. We have thus a collection of available heuristics to pick from. On Fig. 2, 3 channels suffice.

Even when all channels were specified by the end-user by means of directives, it is good to run the above minimization problem. The user may have in mind a "logical" set of channels that may be reduced to a smaller set. On Fig. 2, suppose the user defined two channels $c4$ and $c5$, corresponding to send's $s4$ and $s5$ respectively, and that receive's $r3$ and $r4$ can receive from both channels. It turns out that channel minimization will merge $c4$ and $c5$ into a single one, because this captures the same communication pattern.

In general, there is not a unique minimal biclique edge cover. Different solutions, although yielding the same number of channels, may imply a marginally different number of iterations in the analyses. On Fig. 3, we have two minimal covers of a

Fig. 3 Two different minimal covers. Channels shown between parentheses

communication bipartite graph. The cover on the left has a send node labelled with two channels. If a forward data-flow analysis reaches this node first, then both channels are affected at once and no other fixpoint iteration will be necessary when later reaching the other send nodes. Conversely, the cover of the right is more efficient for a backward data-flow analysis, as the node with two channels is now a receive node.

There is an unfortunate interaction between this channel mechanism and the choice of generalization during data-flow analyses. If the code is such that native `MPI` calls are encapsulated into wrapper procedures, then attaching the channel to the native `MPI` calls may leave us with only one channel, as there is only one textual `MPI_SEND` present. On the other hand, we probably want to attach different

channels to different wrapper calls, as if the wrapper procedures were the primitive communication points. We did not address this problem, which would either require to attach the channel to the wrapper call, or the possibility to opt for specialization instead of generalization for the analysis of each wrapper call, which means that a wrapper procedure will be analyzed once for each of its call sites.

7 Implementation and outlook

We have implemented a prototype native treatment of `MPI` communication calls in `Tapenade` following the ideas of this paper. Implementation amounts to the following:

- define the basic properties of `MPI` procedures in `Tapenade`'s standard library.
- make the tool recognize the `MPI` calls as message-passing calls, identify them as send, receive, collective ... and distinguish in their arguments those defining the channel and those containing the communicated buffer.
- implement flow graph local restart into the single parent class of all data-flow analyses.
- adapt each individual data-flow analysis at the point of propagating data-flow information through one message-passing call

We also updated tangent mode AD to introduce differentiated communication when the communication channel is active. Notice that this also introduces a notion of differentiation for parameters such as "tag", "request", and error "status". For instance, the "tag" of the differentiated communication call must be distinct from the original call's to make sure the receives are done in the correct order. Similar remarks hold between the "request" of nonblocking communication, and also for error "status".

We obtained correct data-flow information on a set of representative small examples, for all data-flow analyses.

We extended validation to a much larger CFD code called AERO, which implements an unsteady, turbulent Navier-Stokes simulation. The code is more that 100,000 lines long, and SPMD parallelization is necessary for most of its applications. Message-passing is done with `MPI` calls. In addition to point-to-point nonblocking communication `MPI_ISEND`, `MPI_IRECV`, and `MPI_WAIT`, the code uses collective communication `MPI_BCAST`, `MPI_GATHER`, and `MPI_ALLREDUCE`.

Given the current stage of development in `Tapenade` about message-passing communication, we could only apply tangent differentiation on the code. The resulting derivatives were validated by comparison of the parallel tangent code with divided differences between two runs of the original code, each of them parallel. At the source level, 10 of the 32 calls to `MPI` were detected active, causing 10 differentiated message-passing calls. On a relatively small test case, average run time per processor of the tangent code was 0.49 second, compared to an original run time per processor of 0.38 second. This increase of 30% is in line with what we observe on sequential codes.

The adjoint mode is still under development. However, we plan to validate soon an adjoint built semi-automatically, using the data-flow information which is already available, and hand-coding the appropriate adjoint communication calls.

We foresee a few extra difficulties for the adjoint mode of AD. As the adjoint differentiation model we have devised [11] exchanges the roles of paired `MPI_ISEND` or `MPI_IRECV` on one hand, and `MPI_WAIT` on the other hand, we need a way of associating those. A solution might be to wrap the `MPI_WAIT`'s into special-purpose `MPI_WAIT_SEND`'s or `MPI_WAIT_RECV`'s containing all the necessary parameters. Another manner would be to run another static data-flow analysis. Matching `MPI_ISEND` or `MPI_IRECV` to `MPI_WAIT` is local to each process, unlike matching `MPI_ISEND` to `MPI_IRECV`. Therefore all we need is a local analysis, akin to data-dependence analysis on the "request" parameter. Considering that `MPI_ISEND` or `MPI_IRECV` write into their "request" parameter, and that `MPI_WAIT` reads its "request" then resets it, the two will match when there is a true dependency between them. User-directives may also be of help as a fallback option.

This work was not done with the one-sided communications of MPI-2 in mind. Although its new synchronization primitives may prove difficult to handle, we believe the remote memory of one-sided communications can be treated like a channel.

References

1. Bronevetsky, G.: Communication-sensitive static dataflow for parallel message passing applications pp. 1–12 (2009). DOI http://dx.doi.org/10.1109/CGO.2009.32. URL `http://dx.doi.org/10.1109/CGO.2009.32`
2. C.Faure, P.Dutto: Extension of odyssée to the mpi library -reverse mode. Rapport de recherche 3774, INRIA, Sophia Antipolis (1999)
3. Gopalakrishnan, G., Kirby, R.M., Siegel, S., Thakur, R., Gropp, W., Lusk, E., de Supinski, B., Schulz, M., Bronevetsky, G.: Formal analysis of mpi based parallel programs: Present and future. Communications of the ACM (2011)
4. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd edition. MIT Press, Cambridge, MA (1999)
5. Hascoët, L., Naumann, U., Pascual, V.: "To be recorded" analysis in reverse-mode automatic differentiation. Future Generation Computer Systems **21**(8), 1401–1417 (2005). DOI 10.1016/j.future.2004.11.009
6. Kreaseck, B., Strout, M.M., Hovland, P.: Depth analysis of mpi programs. ANL/MCS-P1754-0510 (2010)
7. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
8. Pacheco, P.S.: Parallel programming with MPI. Morgan Kaufmann Publishers Inc. (1996)
9. Shires, D., Pollock, L., Sprenkle, S.: Program flow graph construction for static analysis of mpi programs. In: Parallel and Distributed Processing Techniques and Applications, pp. 1847–1853 (1999)
10. Strout, M.M., Kreaseck, B., Hovland, P.D.: Data-flow analysis for mpi programs. In: Proceedings of the International Conference on Parallel Processing (ICPP) (2006)
11. Utke, J., Hascoët, L., Heimbach, P., Hill, C., Hovland, P., Naumann, U.: Toward adjoinable MPI. In: Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering, PDSEC-09 (2009). DOI http://doi.ieeecomputersociety.org/10.1109/IPDPS.2009.5161165