Static Approximation of MPI Communication Graphs for Optimized Process Placement

Andrew J. McPherson[1], Vijay Nagarajan[1], and Marcelo Cintra[2]

[1] School of Informatics, University of Edinburgh
[2] Intel
{ajmcpherson, vijay.nagarajan}@ed.ac.uk, marcelo.cintra@intel.com

Abstract. Message Passing Interface (MPI) is the de facto standard for programming large scale parallel programs. Static understanding of MPI programs informs optimizations including process placement and communication/computation overlap, and debugging. In this paper, we present a fully context and flow sensitive, interprocedural, best-effort analysis framework to *statically* analyze MPI programs. We instantiate this to determine an approximation of the point-to-point communication graph of an MPI program. Our analysis is the first pragmatic approach to realizing the full point-to-point communication graph without profiling – indeed our experiments show that we are able to resolve and understand 100% of the relevant MPI call sites across the NAS Parallel Benchmarks. In all but one case, this only requires specifying the number of processes. To demonstrate an application, we use the analysis to determine process placement on a Chip MultiProcessor (CMP) based cluster. The use of a CMP-based cluster creates a two-tier system, where inter-node communication can be subject to greater latencies than intra-node communication. Intelligent process placement can therefore have a significant impact on the execution time. Using the 64 process versions of the benchmarks, and our analysis, we see an average of 28% (7%) improvement in communication localization over *by-rank* scheduling for 8-core (12-core) CMP-based clusters, representing the maximum possible improvement.

1 Introduction

Message Passing Interface (MPI) is the de facto standard for programming large scale parallel programs. Paradigm-aware static analysis can inform optimizations including process placement and communication/computation overlap [8], and debugging [27]. Fortunately, message-passing lends itself effectively to static analysis, due to the explicit nature of the communication.

Previous work in MPI static analysis produced several techniques for characterizing communication [5,24,25]. Common to these techniques is the matching of send and receive statements, which while potentially enabling interprocess dataflow analyses, can limit coverage. More importantly, the techniques are limited in their context sensitivity, from being limited to a single procedure [5,24], to only offering partial context sensitivity [25]. Therefore, the existing techniques do not provide tools applicable to determining the full communication graph.

In comparison to static approaches, profiling can be effective [7], but is more intrusive to workflow. As Zhai et al. [28] note, existing tools such as KOJAK [20], VAMPIR [21], and TAU [23] involve expensive trace collection, though lightweight alternatives e.g., mpiP [26] do exist. *The main question we address is whether a static analysis can provide comparable insight into the MPI communication graph, without requiring the program to be executed.*

Tools for understanding MPI communication have several applications. For example, one can consider the running of an MPI program on a cluster of Chip Multiprocessors (CMP). Here, there exists a spatial scheduling problem in the assignment of processes to processor cores. In MPI, each process is assigned a *rank*, used to determine its behavior and spatial scheduling. For example, OpenMPI [10] supports two schedules, **by-rank** – where processes fill every CMP slot before moving onto the next CMP, and **round-robin** – where a process is allocated on each CMP in a round-robin fashion. Without intervention, there is no guarantee that the communication is conducive to either schedule. This may lead to pairs of heavily communicating processes scheduled on different nodes. Communication between nodes, using ethernet or even Infiniband, can be subject to latencies significantly larger than in intra-node communication. This inefficient scheduling can cause significant performance degradation [2,19,29]. Prior analysis allows intelligent placement to alleviate this issue.

In this work, we propose a fully context and flow sensitive, interprocedural analysis framework to statically analyze MPI programs. Our framework is essentially a forward traversal examining variable definitions; but to avoid per-process evaluation, we propose a data-structure to maintain context and flow sensitive partially evaluated definitions. This allows process sensitive, on-demand evaluation at required points. Our analysis is best-effort, prioritizing coverage over soundness; for instance we assume global variables are only modified by compile-time visible functions. Underpinning our analysis is the observation that for a significant class of MPI programs, the communication pattern is broadly input independent and therefore amenable to static analysis [5,6,11,22].

We instantiate our framework to determine an approximation of the point-to-point communication graph of an MPI program. Applying this to programs from the NAS Parallel Benchmark Suite [4], we are able to resolve and understand 100% of the relevant MPI call sites, i.e., we are able to determine the sending processes, destinations, and volumes for all contexts in which the calls are found. In all but one case, this only requires specifying the number of processes.

To demonstrate an application, the graph is used to optimize spatial scheduling. An approximation is permissible here, as spatial scheduling does not impact correctness in MPI programs. We use the extracted graph and a partitioning algorithm to determine process placement on a CMP-based cluster. Using the 64 process versions of the benchmarks, we see an average of 28% (7%) improvement in communication localization over *by-rank* scheduling for 8-core (12-core) CMP-based clusters, representing the maximum possible improvement.

The main contributions of this work are:

- A novel framework for the interprocedural, fully context and flow sensitive, best-effort analysis of MPI programs.
- A new data structure for maintaining partially evaluated, context and flow sensitive variable representations for on-demand process sensitive evaluation.
- An instantiation of the framework, determining optimized process placement for MPI programs running on CMP-based clusters.

2 Related Work

2.1 Static Analysis of MPI Programs

Several techniques have been proposed to statically analyze MPI programs. However, they have limitations that prevent their application to the problem described. Noted by multiple sources are the SPMD semantics of MPI [5,18,25]. The SPMD semantics are important as they largely define the methods that can be, and are, used to perform communication analysis.

MPI-CFG [24] and later MPI-ICFG [18,25] annotate control-flow graphs (CFGs) with process sensitive traversals and communication edges between matched send and receive statements. Backward slicing is performed on the pure CFG to simplify expressions that indirectly reference process rank in the call parameter. *The lack of full context sensitivity prevents these works being applied to the problem described.* However, they do highlight the need to use slicing to determine process sensitive values and the need for an interprocedural approach.

Bronevetsky [5] introduces the parallel CFG (pCFG). It represents process sensitivity by creating multiple states for each CFG node as determined by conditional statements. Progress is made by each component until they reach a communication, where they block until matched to a corresponding statement. Communication is then modeled between sets, providing a scalable view of communication. The complex matching process is limited to modeling communication across Cartesian topologies. Due to their proof requirements, wildcards cannot be handled [5]. pCFG tuples are comparable with the data structure proposed in this work, but as detailed in Section 3 we dispense with sets, and with matching, achieving the data representation by different means. *Most importantly, pCFG is intraprocedural and therefore ineffective with real programs.*

2.2 Profiling and Dynamic Analysis of MPI Programs

Profiling and dynamic analysis techniques have also been applied to MPI programs [20,21,23,26]. Targeting the same optimization as this work, MPIPP [7] uses the communication graph, extracted via profiling, to optimize process placement. *This approach would compare unfavorably to a static approach achieving similar coverage, given the cost of repeated executions on potentially scarce resources.*

Recognizing the burden of profiling, FACT [28] seeks to understand communication by only profiling a statically determined program slice. *While reducing*

the cost of profiling, the authors of FACT note that the slicing may alter the communication pattern in non-deterministic applications.

Dynamic approaches include Adaptive MPI [15,16], which provides a runtime system, capable of automatic communication/computation overlap, load balancing, and process migration. These techniques allow it to take advantage of communication phases in the program. *Given the cost of migration and need for a runtime system, the methods described are required to overcome further overhead to achieve better speedup.* For programs that lack distinct temporal phases of communication, this may not be possible.

3 Our Approach

In this section we explain the key elements of our approach in terms of design decisions, data structures, and present an overall analysis algorithm. To motivate our approach we examine a sample MPI program, presented as Figure 1.

3.1 General Principles

The basic aim of a static approach to approximating the point-to-point communication graph is to understand *MPI_Send* calls (as in line 22 of our example). There are four elements to this, the **source** - which processes make the call, the **destination** - to which processes do they send data, the **send count** and the **datatype** - from which the volume of bytes can be calculated.

```
#include <mpi.h>
int my_rank, comm_size, indata, outdata;
MPI_Status stat;

int main (int argc, char **argv) {
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &comm_size);
  indata = comm_size + 4;
  if (my_rank < 5)
    communicate ();
  if (my_rank < 6)
    indata = indata + my_rank;
  if (my_rank > 7)
    communicate ();
  MPI_Finalize ();
  return 0;
}
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

```
void communicate () {
  if (my_rank % 2 == 0 && my_rank < comm_size - 1)
    MPI_Send (&indata, 1, MPI_INT, my_rank + 1, 0,
        MPI_COMM_WORLD);
  else
    MPI_Recv (&outdata, 1, MPI_INT, MPI_ANY_SOURCE,
        0, MPI_COMM_WORLD, &stat);
  indata = 0;
}
```

```
20
21
22
23
24
25
26
27
28
```

Fig. 1. Example of a simple MPI program

As we can see from line 10, the call to *communicate*, which contains the *MPI_Send* can be conditional. On this basis we can say that an interprocedural approach is essential, as an intraprocedural approach fails to capture the fact that any process with a *rank* greater than 4 would not make the first call to *communicate* and therefore not reach the *MPI_Send* in this instance.

Accepting the need for full context sensitivity, there are two basic approaches that could be employed. One could use some form of interprocedural constant

propagation [12], within a full interprocedural dataflow analysis [14], to determine the relevant parameter values (*destination*, *send count* and *datatype*). However, such an approach is not without issue. Significantly, the SPMD nature of MPI programs means the path through the program may be process sensitive (as seen in our example). Therefore, a constant propagation approach would require complete evaluation of the program for each intended process to determine the processes communicating (*source*) at each call site. Also, even with flow sensitivity, the coverage achieved by such a rigorous approach may not be enough to provide an approximation of the communication graph.

The alternative basic approach is a static slicing, based on a partial data flow analysis [13], that identifies the *MPI_Send* and then evaluates at the program point before the call, for each of the contexts in which the call is found. While such a technique is possible and requires potentially less computation than the previous approach [9], it suffers from the same weaknesses, with regard to strictness and full reevaluation to determine the *source*.

Due to these issues, we choose to follow a composite approach based largely on a forward traversal to establish interprocedural context without backtracking. This traversal walks through the CFG of a function, descending into a child function when discovered. This is analogous to an ad-hoc forward traversal of the Super CFG [3], but with cloned procedures. To avoid full reevaluation, we do not treat process sensitive values as constants and instead leave them partially evaluated in a data structure introduced in Section 3.3. Therefore, we progress in a process insensitive manner, only performing process sensitive evaluation for calls and MPI statements, using our data structure to perform on-demand slicing. To enable broader coverage, we make the approach best-effort, applying the assumption that global variables are only modified by functions visible to the compiler. While this renders our evaluations strictly unsound, this is required to achieve even fractional coverage.

3.2 Context, Flow, and Process Sensitivity

Focusing on the *MPI_Send* in our example, we see that establishing definitions with our approach requires understanding two elements; which processes enter the parent *communicate* function (context sensitivity) and of those processes, which reach the call (flow sensitivity). Due to the SPMD semantics, process sensitivity (which processes reach a certain program point), is derived from the context and flow sensitivities. These are handled using two related techniques.

To understand which processes call the parent function and therefore potentially make the *MPI_Send*, we introduce the "live vector", a boolean vector to track which processes are live in each function as we perform the serial walk. The length of the vector is the number of processes for which we are compiling, initialized at the main function as all true. Requiring the number of processes to be defined entails compiling for a specific scale of problem. However we do not believe this is a significant imposition, given the typical workflow of scientific and high-performance computing. Notably, this requirement also applies to profiling, where a new run is needed for each change in the number of processes.

The live vector is a simplification of the context of the call for each process. This allows for, at a subsequent assignment or call, evaluation using the live vector and flow information, rather than repeated reevaluations within the context of the entire program. When a call is found, we generate a live vector for that function before descending into it. This "child live vector" is generated from the live vector of the parent function of the call and is logically a subset of those processes that executed the parent function. The evaluation of which processes are live in the child live vector uses the flow sensitivity technique, described next.

Within a function, which processes make a call depends on the relevant conditions. We examine the CFG in a Static Single Assignment form where the only back edges are for loop backs, all other edges make forward progress. A relevant condition is defined as one meeting three requirements. Firstly, the basic block containing the condition is not post-dominated by the block containing the call. Secondly, there are no blocks between the condition block and the call block that post-dominate the condition block. Thirdly, there exists a path of forward edges between the condition block and the call block.

The evaluation of relevant conditions is done with regard to their position in the CFG and the paths that exist between them. This ensures that calls subject to interdependent conditions, as seen in line 21 of our example, can be evaluated correctly. The definitions for the condition and its outcome can be process sensitive, so the evaluation of the relevant conditions must be performed separately for each process. The method by which this and the evaluation of MPI arguments is achieved is introduced in the next section.

3.3 On-demand Evaluation

To evaluate the conditions and the arguments of the *MPI_Send* as detailed above, we implement a tree-based representation to hold the partially evaluated variables as our approach requires. Our representation provides the ability to perform on-demand static slicing, sensitive to a particular process, without repeated analysis of the program. In fact, since only a fraction of the variables influence the communication graph, most will not need evaluation.

For each assignment or ϕ-node encountered, a new node of our representation is created, or if the variable already exists, its node is modified. These nodes are stored in either the global or the local hash tables allowing efficient lookup and discarding of out of scope definitions that are unreferenced by any in scope.

Each node is of one of eight types, representing all the cases that arise. **Constant** - representing a number. **SPMD** - for definitions generated by operations with process sensitive results, e.g., a call to *MPI_Comm_rank*. **Expression** - represents an arithmetic expression and contains an operator and pointers to nodes upon which to apply it. **Many** - handles repeated definitions to the same variable, allowing context, flow, and process sensitive resolution. **Builtin** - required for built in functions (e.g., square root), contains an operator and pointer to the node upon which it is to be applied. **Iterator** - identical to Constant, but specially controlled for loop operations. **Array** - for handling array definitions, see Section 3.4. **Unknown** - for when a definition is unresolvable.

The node type used is defined by the node types of the operands of the defining statement and whether a definition already exists. ϕ-nodes are treated as multiple definitions to a variable, resulting in a **many** node.

To better convey the operation of this data structure we present Figure 2, which shows the state of *indata* by the end of the program described in Figure 1 (line 16). By the end of the program, *indata* has been defined multiple times, but not all definitions apply to all processes. For this example, we assume the program has been compiled for 12 processes.

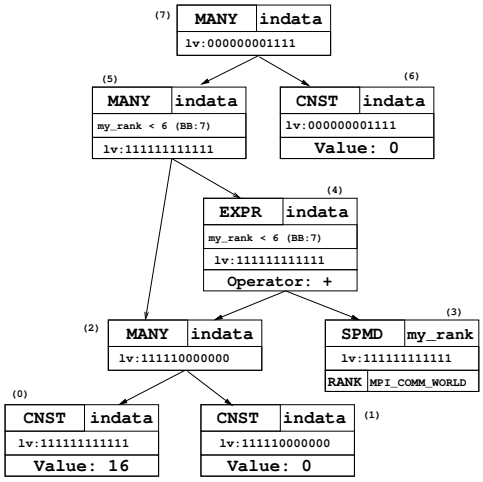

Fig. 2. The representation of *indata* at line 16 in Figure 1. **lv** represents live vector.

The first definition (line 9), is to add *comm_size* to the constant 4. While *comm_size* is an SPMD value, because it is the same for all processes this expression can be reduced to a **constant** (marked (0) in Figure 2). Then after descending into *communicate* for the first time, *indata* is redefined in line 27. Since *indata* has already been defined, as well as creating a new **constant** definition (marked (1)), a **many** (marked (2)), copying the live vector of the new definition is also created, as the new definition does not apply to all processes. Definition (2) is now the current definition stored in the hashtable. Were *indata* to be evaluated at this point, processes with a rank of less than 5 would take the right branch (to the newer definition) and evaluate *indata* as 0, whereas all others would use the previous definition.

Upon returning to the parent function, *indata* is redefined again (line 13). This time as its previous definition plus the rank of the process. Since the components are not both of type **constant**, an **expression** is created (marked (4)). This **expression** will combine the evaluation of the child **many** (marked (2)) with the rank for *MPI_COMM_WORLD* for the particular process (an **SPMD** marked (3)). Again because this variable has been defined before, a **many**

(marked (5)) is created, linking the old and new definitions. Note that we do not need to copy the old definition, merely including it in the new definition with appropriate pointers is sufficient. Note also that this new definition is subject to a condition, the details of which are also associated with both the **expression** and the **many**. The association of conditional information allows for differentiation between multiple definitions where the live vector is the same, i.e., the difference is intraprocedural. Finally, the program descends again into *communicate*, creating another definition (marked (6)) and **many** (marked (7)).

3.4 Special Cases

There are a few special cases that merit further explanation:

Arrays - Viewing elements as individual variables, there is a complication where the index of an array lookup or definition is process sensitive. Operating on the assumption that only a small fraction of elements will actually be required, efficiency demands avoiding process sensitive evaluation unless necessary. Therefore, an array is given a single entry in the hash table (type **array**), that maintains a storage order vector of definitions to that array. A lookup with an index that is process sensitive returns an **array** with a pointer to this vector, its length at the time of lookup, and the unevaluated index. Evaluating an element then requires evaluating the index and progressing back through the vector from the length at time of lookup, comparing (and potentially evaluating) indices until a match is found. If the matched node doesn't evaluate for this process, then taking a best effort approach, the process continues. This ensures that the latest usable definition is found first and elides the issue of definitions applying to different elements for different processes.

Loops - Again we take a best effort approach, assuming that every loop executes at least once, unless previous forward jumps prove this assumption false. At the end of analyzing a basic block, the successor edges are checked and if one is a back edge (i.e., the block is a loop latch or unconditional loop), then the relevant conditions are resolved without respect to a specific process. This determines whether the conditions have been met or whether we should loop. This means that when an iterator cannot be resolved as the same for all processes, the contents of the loop will have been seen to execute once, with further iterations left unknown. These loops are marked so that calls inside them are known to be subject to a multiplier. For more complex loops with additional exits, these are marked during an initial scan and evaluated as they are reached.

The choice to only resolve loops with a process insensitive number of iterations does potentially limit the power of the analysis. However, it is in keeping with our decision to analyze serially. Parallelizing for the analysis of basic blocks and functions inside a loop would complicate the analysis to the point where it would be equivalent to analyzing the program for each process individually. As we see in Section 4, this decision does not have a negative impact on our results with the programs tested.

Parameters - Both pass-by-value and pass-by-reference parameters are handled. In the case of pass-by-value, a copy of the relevant definition is created to prevent modifications affecting the existing definition.

3.5 Overall Algorithm

Combining the elements described, we produce an algorithm for the analysis of MPI programs, presented as Listing 1.1. The only specialization of this framework required to create an analysis of point-to-point communication, is the generation of graph edges based on the evaluation of *MPI_Send* statements. This is achieved by evaluating the *MPI_Send*, in the same manner as other functions, to determine for which processes graph edges need to be generated. Then for each of these processes, the relevant parameters (**send count**, **datatype**, and **destination**), are subjected to process sensitive evaluation.

```
1  global_defs = ∅
2
3  walker (function, live_vector, param_defs) {
4    local_defs = ∅
5    for basic block in function
6      for statement in basic block
7        if is_assignment (statement)
8          [ record to global_defs or local_defs as appropriate
9        else if is_call (statement)
10         child_live_vector = live_vector
11         for live_process in live_vector
12           [ evaluate relevant conditions to this call, in the context of each process, marking false in the
13           [ child_live_vector if the process won't make the call or the conditions are unresolvable
14         if is_mpi (call)
15           [ evaluate as appropriate
16         else if has_visible_body (call)
17           [ Generate parameter definitions based on the variables passed to the child function
18           walker (call, child_live_vector, child_param_defs)
19     [ If loop back block or additional exit, analyze conditions and adjust current basic block as appropriate
20 }
```

Listing 1.1. Algorithm for process and context sensitive traversal

3.6 Scalability

Scaling the number of processes results in a worst case $O(n)$ growth in the number of evaluations. This is due to the worst case being where all evaluations are process sensitive, with the number of evaluations increasing in line with the number of processes. A caveat to this is if the length of the execution path changes with the number of processes. Specifically, if the length of the execution path is broadly determined by the number of processes then the scalability would be program specific and unquantifiable in a general sense. However, in such a situation one would often expect to see better scalability than the stated worst case, as a fixed problem size is divided between more processes, reducing the length of the execution path.

To improve upon the worst case, process sensitive and insensitive evaluation results are stored for each node. This includes all nodes evaluated in the process

of evaluating the requested node. These results are then attached to the relevant nodes. This means that reevaluation simply returns the stored result. While storage of these results requires additional memory, it prevents reevaluation of potentially deep and complex trees. Since we find only a fraction of nodes need evaluating, this does not pose a great memory issue. As we will show in Section 4.4, we achieve far better than the worst case for all the benchmarks.

3.7 Limitations

There are a few limitations to the technique, some are fundamental to the static analysis of MPI, others particular to our design.

Pointers - The use of pointers in a statically unprovable way, with particular reference to function pointers, can lead the analysis to miss certain definitions. Again we prioritize coverage over soundness, neglecting the potential impact of statically unresolved pointer usage.

Recursive Functions - We take no account of recursive functions, which could lead to non-termination of the algorithm. Subject to the previous caveat, recursiveness can be determined by an analysis of the call graph or as the algorithm runs. The simple solution would be to not pursue placement if recursion is detected, but it is perhaps possible to allow some limited forms.

Incomplete Communication Graphs - If the complete communication graph cannot be resolved, it could produce performance degradation if placement or other optimizations are pursued. However, as we see in Section 4.2, certain forms of incompleteness can be successfully overcome. Automatically dealing with incompleteness in the general case remains an open problem.

4 Results

The primary goal of our experiments is to evaluate the efficacy of our framework in understanding communication in MPI programs. To this end, we evaluate our coverage – in terms of the percentage of sends we are able to fully understand. Next we investigate the improvements in communication localization that are available from better process placement, guided by our analysis. This is followed by an evaluation of the performance improvements available from improved process placement. Finally, we explore the scalability of the technique.

We implemented our framework in GCC 4.7.0 [1], to leverage the new interprocedural analysis framework, particularly Link Time Optimization. Experiments were performed using the 64 process versions of the NAS Parallel Benchmarks 3.3 [4], compiling for the *Class A* problem size. We tested all NAS programs that use point-to-point communication (BT, CG, IS, LU, MG and SP).

Spatial scheduling is considered as a graph partitioning problem. To this end we applied the k-way variant of the Kernighan-Lin algorithm [17]. It aims to assign vertices (processes) to buckets (CMPs) as to minimize the total weight of non-local edges. As the algorithm is hill-climbing, it is applied to 1,000 random starting positions, and the naive schedules, to avoid local maxima.

4.1 Coverage Results

Table 1. Coverage results and comparison with profiling for NAS Class A problems using 64 MPI processes.

	Profiling		Analysis	
	No. Call Sites	No. Bytes	No. Call Sites Correct	No. Bytes
BT	12	8906903040	12	$58007040 + n(44244480)$
CG	10	1492271104	10	1492271104
IS	1	252	1	252
LU	12	3411115904	12	$41035904 + n(13480320)$
MG	12	315818496	12^3	$104700416 + n(52779520)^3$
SP	12	13819352064	12	$48190464 + n(34427904)$

We quantify coverage by two metrics: the number of *MPI_(I)Send* call sites that we can *correctly* understand, and the the total *number of bytes* communicated. An *MPI_(I)Send* is said to be understood correctly if we can identify the calling process, the destination process, and the volume of data communicated *in all the circumstances under which the call is encountered* – as seen in Figure 1, the same call site can be encountered in multiple contexts. In addition to this, each of the sends can repeat an arbitrary number of times, necessitating that the analysis resolves relevant loop iterators. To quantify this, we measure the total number of bytes communicated.

The coverage our analysis provides is shown in Table 1, with profiling results for comparison. With the exception of MG, each *MPI_(I)Send* call site is being automatically and correctly evaluated in all contexts for all processes. This means that our analysis is correctly identifying the calling processes, the destination and the volume of data for every *MPI_(I)Send*.

In CG and IS the number of bytes communicated also matches the profile run. For these programs, the relevant loops could be statically resolved by our framework. However, in BT, LU, MG and SP an unknown multiplier n exists. This occurs when the iteration count of a loop containing send calls cannot be statically determined; in the case of the four benchmarks affected, the iteration count is input dependent. As will be seen in the following section, this has no impact on the schedule, and hence the communication localization.

In contrast, simple analysis of MG fails to determine the point-to-point communication graph. Our analysis correctly determines the sending processes (**source**) and the **datatype**, for each call site. However, the **destination**, **send count**, and number of iterations are input dependent. In the case of MG, the **destination** and **send count** depend on four input variables (nx,ny,nz,and lt). If these variables, which determine the problem scale, are specified, then our analysis is able to correctly evaluate each call site. With programs such as MG where the input is partially specified, one could specify the whole input (including the number of iterations), but this is not necessary.

The case of MG highlights the issue of input dependency and how it can blunt the blind application of static analysis. For programs where the communication pattern is input dependent, analyses of the form proposed in this work will never be able to successfully operate in an automatic manner. However, by supplying input characteristics (as would be required for profiling), it is possible to determine the same communication graph that profiling tools such as mpiP observe. Crucially, unlike profiling, this is without requiring execution of the program. For the following sections, we will assume that the four required input variables have been specified for MG, with results as shown in Table 1.

4.2 Communication Localization

In this section, we evaluate the communication localized by applying the partitioning algorithm to the communication graph generated by our analysis. We compare our localization with four other policies. *Round-robin* and *by-rank*, the two default scheduling policies; *random* which shows the arithmetic mean of 10,000 random partitionings; and *profiling* in which the same partitioning algorithm is applied to the communication graph generated by profiling.

As described in the previous section, four of the programs (BT, LU, MG and SP) have an unknown multiplier in the approximation extracted by analysis. To see the impact of this, communication graphs for each of these benchmarks were generated using values of n from 0 to 1,000. Partitioning these graphs yielded the same (benchmark specific) spatial schedules for all non-negative values of n. Therefore we can say that the optimal spatial schedules for these programs are insensitive to n (the only difference in coverage between profiling and analysis).

Figure 3 shows partitioning results for the NAS benchmarks on 8-core and 12-core per node machines. One can see from these results that of the naive partitioning options *by-rank* is the most consistently effective at localizing communication, better than *round-robin* as has previously been used as a baseline [7]. In fact we see that *random* is more effective than *round-robin* for these programs. Confirming our coverage results from the previous section, and our assertion of the null impact of the unknown multipliers, we see that our *analysis* localization results match the *profiling* localization results for each of the programs tested, as the same schedules are generated.

At 8-core per node we see improvement in 4 out of the 6 benchmarks. On average [4] we see 28% improvement over *by-rank*. We also see that *round-robin* performs equivalently to *by-rank* in 3 cases (BT, LU and SP), in the others it performs worse. For 12-core per node systems we see improvement in 5 out of the 6 benchmarks. On average we see 7% improvement over *by-rank*. Again *round-robin* significantly underperforms other strategies. In fact in 4 cases it fails to localize any communication.

As Figure 3 shows, it is not always possible to improve upon the best naive scheduling (*by-rank*). This occurs when the program is written with this schedul-

[3] Requires partial input specification, see Section 4.1

[4] Geometric mean is used for all normalized results.

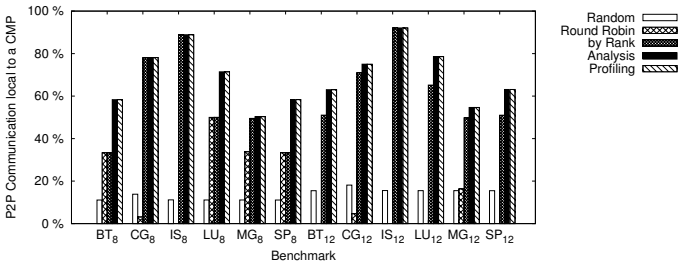

Fig. 3. Percentage of point-to-point communication localized to a CMP.

ing in mind and the underlying parallel algorithm being implemented is conducive to it. However as the results show, analysis of the communication graph and intelligent scheduling can increase the localization of communication.

4.3 Performance Results

While our main focus is on developing an accurate static analysis that matches the results of profiling, we performed a number of experiments to confirm the impact of improved spatial scheduling observed by others [7]. We used a gigabit ethernet linked shared use cluster which has both 8-core and 12-core nodes available. We found that the impact of improved spatial scheduling was greater on the 12-core nodes. In this configuration, the best result was with CG, where the improved spatial scheduling resulted in 18% (8%) execution time reduction over *round-robin* (*by-rank*). On average, across all benchmarks, the improved schedule resulted in 5% (2%) execution time reduction over *round-robin* (*by-rank*).

4.4 Scalability Results

To confirm our assertions in Section 3.6, we compiled the benchmarks for different numbers of processes. Figure 4 presents the results by comparing the total number of nodes of the data structure evaluated during each compilation. Note that a reevaluation returning a stored result still adds 1 to total count.

As Figure 4 shows, we achieve notably better than the $O(n)$ worst case. This demonstrates the effectiveness of the optimizations described in Section 3.6. With particular reference to IS and MG, we can also see the impact of the reduction in work per process, manifesting as a reduction in the number of evaluations, as the process specific program simplifies. Overall the scalability results are positive for all programs, with significant improvement over the worst case.

5 Conclusions

In this work we proposed a novel framework for the interprocedural, fully context and flow sensitive, best-effort analysis of MPI programs. This framework

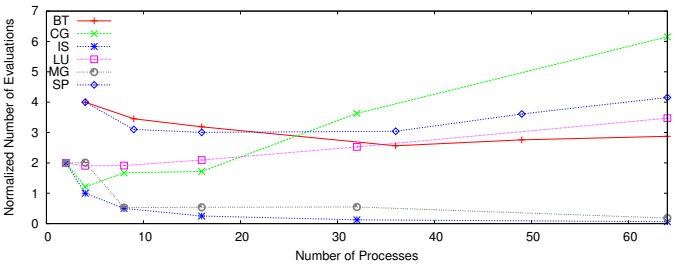

Fig. 4. Normalized total number of evaluations at each usable number of processes. BT and SP are normalized to 4 processes as they only support square numbers.

leverages a new data structure for maintaining partially evaluated, context sensitive variable representations for on-demand process sensitive evaluation. We instantiated this framework to provide a static method for determining optimal process placement for MPI programs running on CMP-based clusters.

Our analysis is able to resolve and understand 100% of the relevant MPI call sites across the benchmarks considered. In all but one case, this only requires specifying the number of processes. Using the 64 process versions of the benchmarks we see an average of 28% (7%) improvement in communication localization over *by-rank* scheduling for 8-core (12-core) CMP-based clusters, which represents the maximum possible improvement.

6 Acknowledgements

We thank Rajiv Gupta, Michael O'Boyle and the anonymous reviewers for their helpful comments for improving the paper. This research is supported by EPSRC grant EP/L000725/1 and an Intel early career faculty award to the University of Edinburgh.

References

1. GCC: GNU compiler collection. *http://gcc.gnu.org*.
2. T. Agarwal, A. Sharma, A. Laxmikant, and L. V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *IPDPS*, 2006.
3. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*, pages 906–908. 2006.
4. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, and L. Dagum. The NAS parallel benchmarks. *Int. J. High Perform. Comput. Appl.*, 1994.
5. G. Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *CGO*, pages 1–12, 2009.
6. F. Cappello, A. Guermouche, and M. Snir. On communication determinism in parallel HPC applications. In *ICCCN*, pages 1–8, 2010.

7. H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *ICS*, pages 353–360, 2006.
8. A. Danalis, L. L. Pollock, D. M. Swany, and J. Cavazos. MPI-aware compiler optimizations for improving communication-computation overlap. In *ICS*, pages 316–325, 2009.
9. E. Duesterwald, R. Gupta, and M. L. Soffa. Demand-driven computation of interprocedural data flow. In *POPL*, pages 37–48, 1995.
10. E. Gabriel et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *PVM/MPI*, pages 97–104, 2004.
11. A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In *IASTED PDCS*, pages 724–729, 2002.
12. D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *PLDI*, pages 90–99, 1993.
13. R. Gupta and M. L. Soffa. A framework for partial data flow analysis. In *ICSM*, pages 4–13, 1994.
14. M. W. Hall, J. M. Mellor-Crummey, A. Carle, and R. G. Rodríguez. FIAT: A framework for interprocedural analysis and transfomation. In *LCPC*, pages 522–545, 1993.
15. C. Huang, O. S. Lawlor, and L. V. Kalé. Adaptive MPI. In *LCPC*, pages 306–322, 2003.
16. C. Huang, G. Zheng, L. V. Kalé, and S. Kumar. Performance evaluation of adaptive MPI. In *PPOPP*, pages 12–21, 2006.
17. B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970.
18. B. Kreaseck, M. M. Strout, and P. Hovland. Depth analysis of MPI programs. In *AMP*, 2010.
19. G. Mercier and E. Jeannot. Improving MPI applications performance on multicore clusters with rank reordering. In *EuroMPI*, pages 39–49, 2011.
20. B. Mohr and F. Wolf. KOJAK - a tool set for automatic performance analysis of parallel programs. In *Euro-Par*, pages 1301–1304, 2003.
21. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12:69–80, 1996.
22. R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan. Using MPI communication patterns to guide source code transformations. In *ICCS*, pages 253–260, 2008.
23. S. Sameer S and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.
24. D. R. Shires, L. L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of MPI programs. In *PDPTA*, pages 1847–1853, 1999.
25. M. M. Strout, B. Kreaseck, and P. D. Hovland. Data-flow analysis for MPI programs. In *ICPP*, pages 175–184, 2006.
26. J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *PPOPP*, pages 123–132, 2001.
27. R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker. MPIWiz: subgroup reproducible replay of MPI applications. In *PPOPP*, pages 251–260, 2009.
28. J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng. FACT: fast communication trace collection for parallel applications through program slicing. In *SC*, 2009.
29. J. Zhang, J. Zhai, W. Chen, and W. Zheng. Process mapping for MPI collective communications. In *Euro-Par*, pages 81–92, 2009.