On the run-time cost of distributed-memory communications generated using the polyhedral model

Ana Moreton-Fernandez
Dept. Informática,
Universidad de Valladolid
Campus Miguel Delibes, s/n
Valladolid, Spain
Email: ana.moreton@alumnos.uva.es

Arturo Gonzalez-Escribano
Dept. Informática,
Universidad de Valladolid
Campus Miguel Delibes, s/n
Valladolid, Spain
Email: arturo@infor.uva.es

Diego R. Llanos
Dept. Informática,
Universidad de Valladolid
Campus Miguel Delibes, s/n
Valladolid, Spain
Email: diego@infor.uva.es

***Abstract*—The polyhedral model can be used to automatically generate distributed-memory communications for affine nested loops. Recently, new communication schemes that reduce the communication volume have been presented. In this paper we study the extra computational effort introduced at run-time by the code generated to manage the communication details across distributed processes. We focus on the most sophisticated communication scheme so far introduced (the FOP scheme). We present an asymptotic cost study of the FOP scheme in terms of two main run-time parameters: The problem size, and the number of processors. Based on this study, we identify scalability limitations in current implementations of these techniques, and propose a simple implementation alternative to eliminate one of them. Experimental results are presented, showing the potential impact on performance of these implementation limitations when using these codes in large parallel systems.**

***Keywords*—Polyhedral model, distributed-memory, run-time, complexity**

I. Introduction

The polyhedral model has been proved to be a useful tool to transform and generate parallel programs for codes with affine nested loops [1]. It can also be used to automatically generate code for distributed-memory platforms. The dependence analysis supported by the model allows to generate code that identifies which values should be communicated across processes, packing/unpacking the data, and executing the proper communication operations [2]. Griebl [5] presents a model to use the polyhedral model over distributed systems. However, his technique produces many redundant communication. Recently, several communication schemes has been presented in order to reduce the volume of data communicated [3], [6]. The automatically generated codes are capable of coordinating the computation and communication across heterogeneous devices. This allows the exploitation of parallelism in heterogeneous clusters with GPUs or other accelerators, which is the current trend to build huge parallel systems [8]. The scale of the machines and problems that can be currently faced, grows by several orders of magnitude comparing with those found in most performance evaluations done previously with distributed-memory polyhedral generated codes. And it will continue growing up, with exascale computing being an important research focus.

In this paper we study the codes generated by the most sophisticated communication scheme introduced so far (the FOP scheme [6]). We present a study of the extra cost introduced at run-time by the generated codes to manage the communications. We do an asymptotic complexity analysis in terms of two main run-time parameters: The problem size N, measured as the number of data elements to be processed, and the number of processors P. Our complexity model highlights some potential scalability limitations in terms of the problem size N, and the number of processors P, in the current implementations of these techniques. Specifically, we focus on the implementation included in the current Pluto compiler framework [4]. We identify and isolate one of this limitations, related to the application of the distribution policy used to schedule the iterations of a parallelized loop. We discuss that for deterministic distribution policies, a simple implementation alternative, previously exploited in Hitmap [7] (a run-time library for management of distributed hierarchical tiling arrays), eliminates this specific scalability problem.

We present three cases of study: 1-D and 2-D Jacobi solvers, and Floyd-Warshall algorithm. The reference codes of these three examples are included in Polybench [9]. Our experimental results show the potential high impact of these scalability limitations on the performance of the application, for huge data sizes and clusters. We also show how the proposed implementation alternative highly alleviates the performance problems, as predicted by our complexity model.

The rest of the paper is organized as follows: Section 2 summarizes the main features of FOP-scheme generated codes. Section 3 presents the cost model. Section 4 discusses our implementation alternative. Section 5 describes the application of the model to one case of study. Section 6 presents the experimental study and results. Section 7 concludes the paper.

II. The communication scheme

The FOP communication scheme is a model for the automatic generation of communication code for distributed-memory parallel programs, in the context of polyhedral code transformations. It is based in dependence analysis across tiles, for parallel programs that distribute the iterations of tiled loops. It has been designed to reduce the volume of data communicated across processors, compared with other state-of-the-art systems to automatically generate communication code for affine-loop nests. It has been also proved that it provides good performance for small and medium sized data sets, and small number of distributed-memory processes [6]. In that work, the authors describe the conceptual approach and the solution design in detail. An implementation of this scheme is included in the current version of the Pluto compiler [4]. In this section, we summarize the main features of the FOP scheme, and some design and implementation decisions of the generated codes.

The code dedicated to calculate and execute communications is inserted at the end of distributed loops which originate a communication need. The analysis of RAW dependences is done separately for each different data structure (array variable) involved in a distributed loop nest. For a given iteration of a distributed loop, the FO (flow-out) set is defined as the set of data generated/written during the iteration, that is required/read during the execution of other iterations. At compile time, the FOP scheme determines a *dependences partition*. A partition of the flow-out set in terms of subsets of target iterations that can be located in different tiles. Each part of the set is treated independently, leading to a different piece of communication code. This partition is application dependent. The generated code for a given partition can have two different flavors. *Multicast* operations that send all the data in a partition to every processor that requires data from it. And *Unicast* operations that issue a different communication for each iteration that requires data from this partition. To avoid sending more than one time the same data to the same processor, unicast operations are only chosen when it is possible to determine, at compile or run-time, that the receiving iterations are all scheduled at different processors. The authors of FOP propose some rules to determine when it is safe to introduce *unicast* operations, *multicast* being the default choice. In this work we will focus on the default and less complex multicast operations.

For multicast operations, FOP introduces one piece of code for each distributed loop, part of the dependences partition, and array variable. The code uses several auxiliary data structures. One data buffer per processor involved in the computation, to store the data to be sent. One single receive buffer to store all the data received from other processors. Two counters per processor, to store the amount of data to be sent or received to/from a remote processor. Each piece of code contain three stages:

1) Pack: Pack data while identifying target processors. The iterations space of the distributed loop assigned to the local processor is traversed again. For each iteration a function generated with application specific information (σ) is used to identify which other iterations (and thus, processors) require data from this partition and iteration. The data is packed (copied) into the corresponding output buffers and the counters that measure the data to be sent to each other processor are updated.
2) Coordination and communication: Interchange of communication sizes across processors, and issue the required point-to-point communications. The coordination step is done with the standard all-to-all MPI collective operation. Each processor sends the value of each output-buffer counter to the corresponding receiving process. This interchange avoids the need to traverse the iteration space scheduled on any other processor, doing the same analysis as for packing, only to obtain the sizes of data that we expect to receive from each other processor. After the coordination step, asynchronous send and receive operations are issued for each processor with a non-zero value in the corresponding counter. With all the receive counters available, it is possible to compute displacements to use one single buffer for all the receive operations.
3) Unpack: Unpack received data. The whole iteration space of the distributed loops is traversed identifying which iterations are scheduled on remote processors for which we have received data. For each one of these iterations it is tested if the local processor is one of the receivers of the data for this partition and iteration (again with a function specifically generated for this application). In that case, the data is unpacked from the buffer to the actual array variable.

III. Cost model

In this section we present a cost model for the run-time computational effort of the communication management code introduced by the FOP scheme [6]. It is the scheme for polyhedral model computations with less communication volume introduced so far. We use as reference for specific design decisions the codes introduced by the current implementation of Pluto. Our model measures the asymptotic cost of the calculations needed to issue the communications in terms of two run-time parameters: *Number of processors* (P), and *Problem size* (N), measured as the number of data elements to be processed. The model does not take into account the actual cost of the communications, which is dependent on external factors related to the platform and the communication topology. We model only the extra costs introduced by the automatically generated code to prepare and launch the communication activities (calculations to pack/unpack, and other local coordination activities). The objective is to find scalability limitations introduced when applying the scheme, that could be eliminated by design or implementation changes.

As commented in the previous section, in this work we will focus on the cost of the lower complexity multicast operations. Unicast operations, that may introduce further limitations, will be covered in an extended future work.

An excerpt of the code generated for a 1D Jacobi parallel program included in the Polybench [9] benchmark is shown in Fig. 1. It will be used as example while discussing the cost model.

A. General cost for a distributed loop

For simplicity of the discussion, let us consider a single distributed loop L, with an iteration index t. Let $D(t) \subset \mathcal{P}(\mathbb{Z})$ be the *Domain* of t, the subset of iterations that are traversed by the loop index. Let $O(L^{\circ})$ be the upper-bound of the run-time cost of calculating the communications needed for all the iterations of $D(t)$ scheduled to a given processor. From now on, constant factors that are application dependent, and not affected by run-time parameters will be denoted with c_{name}, where the *name* will be a single lower-case letter.

Each combination of distributed-loop, variable, and part of the dependences partition, leads to one *Instance* of communication code. In Fig. 1, lines 9 to 35 are the code generated for one communication instance associated to array variable b. Lines 36 to 61 contain a similar code, generated for a second communication instance associated to array variable a.

Let c_z be the number of combinations of different array variables, and parts of the dependences partition introduced by the FOP scheme for the loop L, and $O(y^{\circ})$ the upper-bound of the cost of one communication instance.

$$O(L^{\circ}) = c_z \times O(y^{\circ})$$

The cost of one instance of communication y° is the sum of the costs of its three consecutive stages previously described: packing, coordination, and unpacking. In the following sections we model the execution of one instance of code for a generic iteration of the outer loops. We will focus on the outer loops iterations that lead to maximum parallelism potential of the distributed loops.

B. Problem size and number of iterations

The loops parallelized by the polyhedral model tools represent a transformed space of the original loops. The cardinality of the iterations set of a distributed loop is a function of the problem size $|D(t)| = f(N)$, that can be determined in terms of the transformations applied. We are mainly interested in the loops where the cardinality grows asymptotically with N, allowing to exploit more parallelism for bigger problem sizes. Some constants are introduced by the transformations that reduce the overall cost. For example, when tiling is applied, the tile size c_t appears as a divisor $|D(t)| = f(N/c_t)$, with an asymptotic upper-bound still in the order of N: $|D(t)| \in O(N)$.

C. Distribution policy

A *Distribution Policy* function $\Pi : D(t), \mathbb{N} \rightarrow \mathcal{P}(D(t))$ is used to determine the subset of a domain $D(t)$ that is scheduled on a processor rank $p \in [0, P-1]$. The *Inverse Distribution Policy* function, $\pi : \mathbb{Z} \rightarrow [0, P-1]$, maps each index of the domain to the corresponding processor rank. In general, distribution policies try to obtain a good load balance. Thus, we assume that the number of iterations scheduled on each processor is similar: $\forall p \in [0, P-1], |\Pi(D(t), p)| \simeq f(N)/P$. The run-time cost of applying these functions is denoted with Π°, and π° respectively.

The function Π is used to compute the iterations of the loop scheduled to the local process. In the example code of Fig. 1, lines 6 and 7 calculate the lower and upper limits of the iteration space to be distributed *_lb_dist* and *_ub_dist*. These are the inputs for the Π function implemented in the *polyrt_loop_dist* function. The outputs, *lbp* and *ubp*, are the lower and upper limits of the locally scheduled iterations. The cost of this function is associated to the parallelization of the algorithm. Thus, we do not consider it as a specific cost introduced by the communication calculations.

D. Packing stage

The packing stage traverses the subset of locally scheduled iterations. See the loop in lines 9 to 15 in Fig. 1, that traverses iterations from *lbp* to *ubp*. It has two main contributions to the overall cost that are computed for each iteration considered.

First, each iteration applies a function $\sigma(i)$, specifically generated for each application, to obtain the list of receiving processors. The sigma function contains a constant number of conditionals c_c, dependent on the application source code. Each conditional potentially applies π to obtain the rank of the processor that has a target iteration. Thus, obtaining the target processors for all the iterations scheduled to a processor, is done in $f(N)/P \times c_c \times \pi^{\circ}$.

Second, each iteration traverses the list of processors to detect the ones that should receive data from the local process. This is done in $O(P)$, with a very small constant c_s, as it executes a simple conditional. See line 12 in Fig. 1. The actual packing operation is done only for the processors detected as receivers (condition evaluated to true). The code for packing data in the output buffers is application dependent and only traverses the data that is going to be sent. However, data are packed (copied) in a different buffer for each receiving processor. Thus, there could be multiples copies of the same data. In the worst case, all processors should receive the same data. This is dependent on the communication structure of the application. For example, neighbor synchronization communications have $O(1)$ number of processors involved for each data subset, while some communications in LU reductions result in $O(P)$ processors involved. Let us model the cardinality of the number of communications with an *h-relation* function $h(P)$. Let c_v be the mean volume of data to be sent by one

```
if ((N >= 1) && (T >= 1) && (N >= 4)) {
   for (t2 = -1; t2 <= floord (3 * T + N - 4, 32); t2++) {
      /* Sequential Code */
         .....
      /* End sequential code */
      _lb_dist = max (ceild (2 * t2, 3), ceild (32 * t2 - T + 1, 32));
      _ub_dist = min (min (floord (2 * T + N - 4, 32), floord (64 * t2 + N + 60, 96)), t2);
      polyrt_loop_dist (_lb_dist, _ub_dist, nprocs, my_rank, &lbp, &ubp);
      for (t4 = lbp; t4 <= ubp; t4++) {
         clear_sender_receiver_lists (nprocs);
         sigma_b_1_0 (t2, t4, T, N, my_rank, nprocs);
         for (__p = 0; __p < nprocs; __p++) {
            if (receiver_list[__p] != 0) {
               send_counts_b[__p] = pack_b_1_0 (t2, t4, send_buf_b[__p], send_counts_b[__p]);
      }  }  }
      if (t2 <= floord (3 * T + N - 5, 32)) {
         MPI_Alltoall (send_counts_b, ..., recv_counts_b, ...);
         req_count = 0;
         for (__p = 0; __p < nprocs; __p++)
            if (send_counts_b[__p] >= 1)
               MPI_Isend (send_buf_b[__p], send_counts_b[__p],... );
         for (__p = 0; __p < nprocs; __p++)
            if (recv_counts_b[__p] >= 1)
               MPI_Irecv (recv_buf_b + displs_b[__p], ...);
         MPI_Waitall (req_count, reqs, stats);
         for (__p = 0; __p < nprocs; __p++) {
            send_counts_b[__p] = 0;
            curr_displs_b[__p] = displs_b[__p];
      }  }
      for (t4 = _lb_dist; t4 <= _ub_dist; t4++) {
         proc = pi_0 (t2, t4, T, N, nprocs);
         if ((my_rank != proc) && (recv_counts_b[proc] > 0)) {
            if (is_receiver_b_1_0 (t2, t4, T, N, my_rank, nprocs) !=0) {
               curr_displs_b[proc] = unpack_b_1_0 (t2, t4, recv_buf_b, curr_displs_b[proc]);
      }  }  }
      for (t4 = lbp; t4 <= ubp; t4++) {
         clear_sender_receiver_lists (nprocs);
         sigma_a_1_0 (t2, t4, T, N, my_rank, nprocs);
         for (__p = 0; __p < nprocs; __p++) {
            if (receiver_list[__p] != 0) {
               send_counts_a[__p] = pack_a_1_0 (t2, t4, send_buf_a[__p], send_counts_a[__p]);
      }  }  }
      MPI_Alltoall (send_counts_a, ..., recv_counts_a, ...);
      req_count = 0;
      for (__p = 0; __p < nprocs; __p++)
         if (send_counts_a[__p] >= 1)
            MPI_Isend (send_buf_a[__p], ...);
      for (__p = 0; __p < nprocs; __p++)
         if (recv_counts_a[__p] >= 1)
            MPI_Irecv (recv_buf_a + displs_a[__p],...);
      MPI_Waitall (req_count, reqs, stats);
      for (__p = 0; __p < nprocs; __p++) {
         send_counts_a[__p] = 0;
         curr_displs_a[__p] = displs_a[__p];
      }
      for (t4 = _lb_dist; t4 <= _ub_dist; t4++) {
         proc = pi_0 (t2, t4, T, N, nprocs);
         if ((my_rank != proc) && (recv_counts_a[proc] > 0)) {
            if (is_receiver_a_1_0 (t2, t4, T, N, my_rank, nprocs) !=0) {
               curr_displs_a[proc] =  unpack_a_1_0 (t2, t4, recv_buf_a, curr_displs_a[proc]);
      }  }  }
}  }
```

Figure 1.   Excerpt of the generated communication code for a 1D Jacobi solver using the FOP scheme

iteration for the array variable considered. This is typically a constant determined by the application and transformations applied. Thus, the cost of this second part of the packing stage can be estimated with: $f(N)/P \times (c_s \times P + c_v \times h(P))$. The overall cost of the whole packing stage is estimated as:

$$pack^\circ = f(N)/P \times (c_c \times \pi^\circ + c_s \times P + c_v \times h(P))$$

E. Coordination and communication stage

The coordination stage includes several actions, see lines 16 to 19 in Fig. 1. It starts with an MPI all-to-all collective communication operation to interchange counters. In general, this type of all-to-all communications are assumed to be done in $O(P)$. Then, the actual point-to-point communications needed are launched traversing the processor ranks in $O(P)$. The actual cost of the communications is not modelled for this work, only the preparation and launching activities. Finally, a last loop is executed that also traverses the processor ranks in $O(P)$ for simple bookeeping operations. We model the overall cost of this stage (without actual communication costs) by:

$$coord^\circ = P$$

F. Unpacking stage

The data received from a processor has been packed in iteration order. Thus, they should be unpacked in the same order. See lines 30 to 35 in Fig. 1. This stage traverses the whole iteration space of the distributed loop (from *_lb_dist* to *_ub_dist* in the example code), using the π function to determine which iterations are scheduled in remote processors. The cost of this operation is modelled with $f(N) \times \pi^\circ$.

A second part of the cost appears only for iterations on remote processors from which data has been received at the local process during the communication stage. In the worst case this condition check, for a given iteration, can be satisfied for all the rest of P processors. But we can model again the number of iterations that are going to be detected as valid across the whole space with the h-relation function $h(P)$ of the application. Each locally scheduled iteration produces a mean of $h(P)$ communications received from other iterations. For these set of valid iterations, a second check is done with a application tailored function that contains one or more pieces of code (a constant number c_d of them, dependent on the source code) and internally applying the π function. Finally, the actual unpack operation is done only once for each data element, and the cost directly depends on the volume of data communicated v. The overall cost of the whole unpacking stage is modelled by:

$$unpack^\circ = f(N) \times \pi^\circ + f(N)/P \times h(P) \times (\pi^\circ + c_d + v)$$

G. Total cost

Our final cost model is dependent on two functions, and some constants, that should be determined for each application: $f(N)$, $h(P)$, v, c_c, c_d. As we are mainly interested in the asymptotic behaviour, it should not be difficult to determine the order of the functions in terms of N and P. The constants only give us a rough idea of the weight of each part of the formula, but they cannot be considered alone for a really precise model, as the amount of arithmetical operations generated by the loop transformations to access the data elements, pack/unpack them, and similar operations has not been considered.

The overall cost of calculating a generic communication instance y°, can be estimated as the accumulation of the three stages: $y^\circ = pack^\circ + coord^\circ + unpack^\circ$.

$$\begin{aligned} y^\circ = f(N)/P \times (c_c \times \pi^\circ + c_s \times P + c_v \times h(P)) \\ +P \\ +f(N) \times \pi^\circ + f(N)/P \times h(P) \times (\pi^\circ + c_d + v) \end{aligned}$$

After multiplicative constant factors elimination, and some simplification the asymptotic upper-bound can be modelled as:

$$O(y^\circ) = O(f(N) \times \pi^\circ + f(N)/P \times \pi^\circ \times h(P) + P)$$

IV. Proposal: Implementation alternative

As it can be observed in the cost model formula, a key operation is the identification of the processor that owns an iteration of the distributed loop, using the inverse distribution policy function π. It appears several times in the cost model, as a multiplier factor.

Given an unknown distribution policy function Π, a simple way to build π is to execute a loop that applies Π to each processor rank, checking if the iteration parameter is in the resulting set. See pseudocode in Fig. 2 (left). The implementation solution previously presented, makes the π function independent on the Π policy implemented, as far as the policy returns a block of contiguous iterations. The run-time Polyrt library version included in the current Pluto distribution, contains only one Π function: A classical block distribution policy. See pseudocode in Fig. 2 (middle).

With the current Pluto's implementation, the cost of the functions is: $O(\Pi^\circ) = O(1)$, and $O(\pi^\circ) = O(P)$. For more generic partition policies the cost may increase, because checking if an index is inside a block range can be done in $O(1)$, but for a generic set of n indexes the search cost is at least $O(\log n)$ if it is sorted, or $O(n)$ if it is is not. In this last case the cost of π could go up to $O(\pi^\circ) = O(P \times N)$.

We propose to use a different approach previously used in Hitmap [7], a run-time library for distributed hierarchical tiling arrays management. In Hitmap, the programmer of the distribution policies is forced to develop plug-ins that include both the direct and the inverse distribution policies functions. In Hitmap, the classical partition policies (block, cyclic, etc.) have implementations where the cost of Π° and π° is quite similar, and it is always $O(1)$. This solution can be exploited for any deterministic distribution policy based on an invertible function. For non-invertible functions the programmer may

```
function pi(Dom d,int i,int P)
  do p = 0, P-1
    d' = PI(d,p)
    if i in d' then return p
  enddo
```

```
function PI(Dom d,int p,int P)
  if (p < |d|%P)
    r.lb = d.lb + (|d|/P)*p + p
    r.ub = r.lb + (|d|/P)
  else
    r.lb = d.lb + (|d|/P)*p + |d|%P
    r.ub = r.lb + (|d|/P) - 1
  endif
  return r
```

```
function pi_Alt(Dom d,int i,int P)
  off = i - d.lb;
  lim = (|d|/P + 1)*(|d|%P)
  if ( off < lim )
    return off/(|d|/P + 1)
  else
    return (off-lim)/(|d|/P) + |d|%P
  endif
```

Figure 2. Pseudo-codes of the original π (left) and Π (middle) functions, and our alternative implementation proposed for π (right). $Dom < lb, ub >$ represents a tuple with the lower and upper bound of a contiguous 1-dimensional iteration space. $|d| = d.ub - d.lb + 1$ represents the domain cardinality.

chose to pay the extra run-time cost factor, or pay an extra memory cost. It is always possible to store in an array the index of the assigned processor for all the elements in the iteration space, keeping the $O(1)$ run-time cost for the π function.

We have introduced in Polyrt (Pluto's runtime helper functions) a direct implementation of the inverse distribution policy for block partitions, eliminating a multiplier factor of P in several stages of the communication calculation. See pseudocode in Fig. 2 (right).

The asymptotic impact of this change can be seen in the cost model. After substituting the costs of the π function derived from the current implementation, the result is:

$$O(y^{\circ}) = (f(N) \times P + f(N) \times h(P) + P)$$

With the alternative implementation, multiplier P factors coming from the π function disappear:

$$O(y^{\circ}) = (f(N) + f(N)/P \times h(P) + P)$$

It is specially remarkable that in the original implementation, the size problem is multiplied by the number of processors during the unpacking stage. In the following sections we present empirical evidence of the impact of creating a specific π function for each distribution policy Π, directly implementing the inverse function with a cost bounded by $O(1)$.

V. Case study: 1-D Jacobi

To show how to apply the cost model, we have chosen as case study the 1-dimensional Jacobi program. This application is a good example to study because the code produced by Pluto includes only one distributed loop with multicast operations, it has a simple neighbor synchronization communication structure, and it is very easy to find proper approximations for the application dependent functions.

A. Cost model parametrization

The code has been generated using the default tile sizes in the Pluto example ($c_t = 32$ iterations for any tiled loop). The function that computes the number of iterations in the transformed parallel loop, has two input parameters: $f(N, T)$. Where N is the array size, and T is the number of iterations of the original sequential code before transformations. The formula used to compute the limits of the distributed loop index $t4$ depend on the value of the outer loop index $t2$ (see lines 6 to 8 in Fig. 1). These loops create a pipelined execution. During the application progress, the amount of distributed iterations of the $t4$ loop grows, it keeps stable for a while, and then decreases. The maximum degree of parallelism obtained in the stable phase is related to the problem size parameters, and can be approximated with: if $(3T \geq N)$, then $f(N, T) \simeq 0.01N$; if $(3T < N)$, then $\lim f(N, T)_{T \to \infty} = 3.125T$. Thus, $f(N, T)$ grows linearly with the problem size parameters $O(f(N, T)) = O(\min(N, 3T))$. For simplicity, let us assume that T is always big enough to obtain the maximum degree of parallelism for a given input array size. Thus, $O(f(N)) = O(N)$.

There are two communications instances, one for array a, and one for array b. Thus, $c_z = 2$. The h-relation function $h(P)$ is typically $O(1)$ in neighbor synchronization applications. Indeed, experimental measures with the generated code for the 1-D Jacobi program show that the mean values of the h-relation across iterations and processors are: $\bar{h}(P) \simeq 1$ for the code instance generated for the array a, and $\bar{h}(P) \simeq 0.25$ for the code instance generated for the array b. The data volume c_v communicated by each distributed iteration has been also measured: $c_v = 188$ data elements for a array, and $c_v = 63$ data elements for b array. Inspecting the generated code, we observe that the other constant values are the following. For the a array $c_c = 7, c_d = 7$, and for the b array $c_c = 1, c_d = 1$.

For an asymptotic behaviour study, we can nevertheless ignore the application constants, and simplify the resulting model for the overall cost of the communications needed for one iteration of the outer loop as:

$$O(L^{\circ}) = O(N \times \pi^{\circ} + N/P \times \pi^{\circ} + P)$$

After substituting the costs of the π function derived from the current implementation, the result is:

$$O(L^{\circ}) = O(N \times P + N + P)$$

With the alternative implementation, multiplier P factors coming from the π function disappear:

$$O(L^{\circ}) = O(N + N/P + P)$$

B. Simulation study

Doing real experiments for big data sizes, and large number of processors, may require a huge amount of computation time in critical supercomputer infrastructures. Fortunately, we

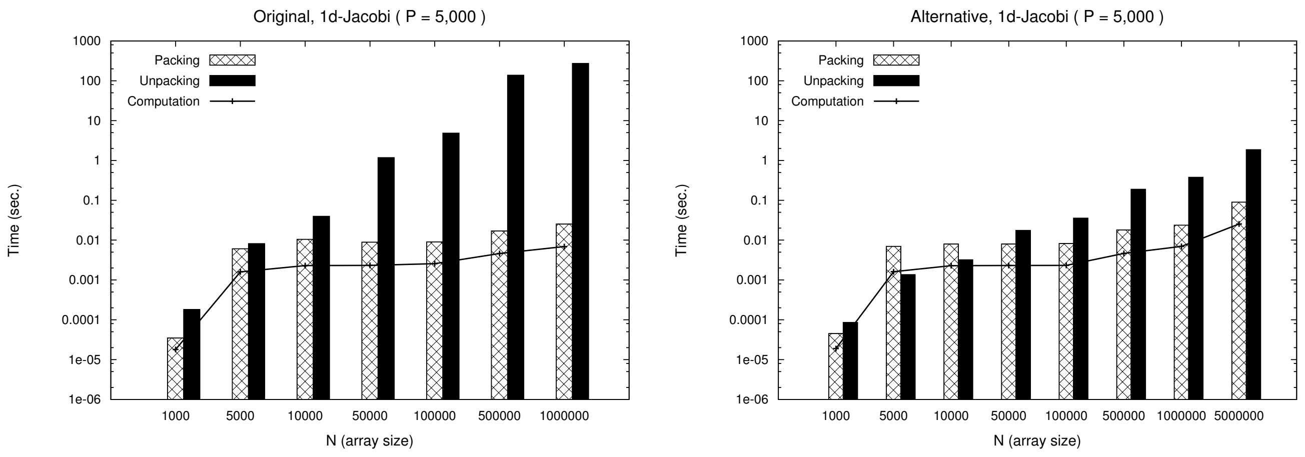

Figure 3. Execution times with the original and alternative π function with different problem sizes N

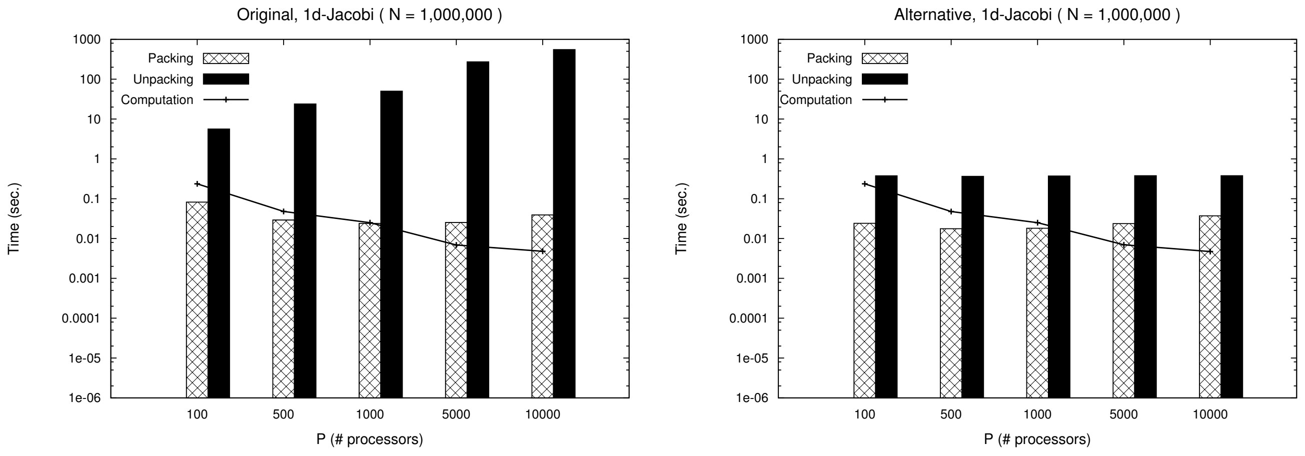

Figure 4. Execution times with the original and alternative π function with different number of processes P

can modify the codes generated by Pluto to simulate a given amount of the outer loop iterations in a chosen processor, with the desired N and P parameters, using a reduced amount of memory. This allow us to perform an empirical study to investigate the effects of scaling the N and P parameters to sizes that resemble high-end supercomputers. Experimental results in a smaller real case and machine are presented in Sect. VI.

The modifications need in the generated code of the 1D Jacobi example include: (1) Adding some code to read parameters for the chosen limits for the outer loop ($t2$ index); (2) change the declarations of the a and b arrays to have a small fixed size (4096 elements); (3) modify all array accesses to use the resulting index modulo 4096 to stay into the fixed arrays boundaries; (4) eliminate the MPI calls; (5) at the start of each $t2$ iteration, locally compute the send counters for all the remote processors, in order to simulate the all-to-all MPI communication eliminated, that coordinates the communication sizes across processors. This is done out of the code sections that are measured with time counters.

We preserve the same time measuring mechanisms included in the original code, for the computation section, and for each one of the three communication calculation stages. The data results produced by this simulation are not correct. The communication codes pack and unpack dummy values in the buffers, and in the constricted arrays. But all the communication preparation calculations, and packing/unpacking operations, are done exactly as in the original code. Thus, the time measures are consistent with the real case, except for the actual communication costs which are intentionally not included or considered in the study.

We discuss results obtained using the simulation program in an PC machine with an Intel-i3 M370 (2.4 GHz) CPU, running a Linux 3.2.29 kernel. The native compiler used is GCC v4.7.1, with the optimization flag $-O3$. We have compiled two versions of the simulation code: One using the original implementation of the π function (Org); and one using the alternative implementation of the inverse function (Alt). The programs are executed with a large range of problem sizes ($N \in [10^3, 10^6], T = N/3$), and number of processors

($P \in [10^2, 10^4]$). The simulation starts at the first iteration of the outer loop $t2$, where the maximum range of the distributed loop $t4$ is achieved. Then, the code runs 100 consecutive iterations of $t2$. Measures have been replicated with arbitrary processor numbers $p = 4, 17, 29, ...$, obtaining the same results.

Figure 3 and 4 shows the measured cost for 100 iterations of the communication code stages of the original (Org) program and the alternative code (Alt), when fixing one of the run-time parameters (N or P). The execution times of the sequential part of the code, that do the actual computation, are also shown with a line. Notice the logarithmic scale on y-axis.

We can observe with the original π function implementation how fast the calculations associated with communication code exceed by orders of magnitude the computation time, when the parameters grow. The product of N and P in the unpacking code due to the cost of the π function dominates the cost, growing to more than one minute of clock time for big problem sizes, or a high number of processors.

With out proposed alternative implementation, the P multiplier introduced by the π function disappears. It can be seen in the right of figure 4 how the unpacking part of the code is no more affected by it. When the number of processors P grows, the amount of work to be done by each local process is proportionally reduced. Nevertheless, the communication code cost is still dependent on the overall problem size. In our experiments, it exceeds the cost of the computation in one order of magnitude for enough number of processors. We can see in the figure 3 how the cost of the unpacking function grows faster than the computation effort for big problem sizes.

VI. Experimental Study

In this section we discuss a real experimental study performed to verify that the asymptotic behaviour of real codes executed in real machines follows the same behaviour as the simulation results, and can be predicted using the proposed cost model.

A. Experimental environment

We have chosen three study cases included with Pluto compiler as examples, and also included in the Polybench benchmarks. The first one is the already discussed 1D Jacobi program. The second one is a 2D Jacobi program, and the third one a Floyd-Warshall's algorithm implementation. This programs represents examples of the classes of programs in Polybench that generates communication code. Linear algebra examples do not derive in actual communications because Pluto transformations assume that the whole data structures are not distributed, but replicated on each processor, deriving in empty sets of flow-out dependences across processors.

We have compiled two versions of each generated program. One using the original implementation of the π function (Org); and one using the alternative implementation of the inverse π function proposed (Alt). The experiments were executed in a shared-memory machine (Heracles), a Dell PowerEdge R815 server, with 4 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores each, adding up to 64 cores in total. For this experimentation using a real platform, we have limited the problem size N, and the number of processors P to the maximum supported by the target machine.

B. Results

Figure 5 show the experimental performance measures obtained for the three study cases. We can observe the same predicted results than in the simulation study in Sect. V-B, but in a smaller scale due to the smaller N and P parameter values.

The impact on the performance of changing the original π function by our proposed alternative is more noticeable in some problems than in others. It depends on the ratio between sequential computation and communications times. For the three cases of study, the most noticeable effect appears for the Floyd-warshall case, where the packing/unpacking cost is almost 30% of the total execution time, as reported in [3]. This is due to: (1) A higher $h(P)$ factor of this algorithm comparing with the neighbor synchronization structure of the Jacobi programs; and (2) a higher number of communication instances in the loop. This effect is also predicted by the proposed cost model.

We can also observe that, as predicted by the model, even after applying our proposed alternative implementation of the π function, there is still a proportional increment of the communication calculation cost at run-time, with the problem size N. The FOP scheme relays on checking if communications are needed for the whole space of distributed tiles, on each processor.

Our results show that the cost model is an useful tool to predict the asymptotic behaviour of the code introduced to manage the communication. It can be used to locate scalability limitations, in order to take design or implementation decisions to avoid them. We also show how our proposed alternative for the implementation of the π function leads to the elimination of one of these scalability problems.

VII. Conclusion

This paper presents a model for the run-time cost of the codes generated by a state-of-the-art polyhedral-model technique (FOP scheme), for communication management in a distributed-memory environment. The model allows to study the asymptotic behaviour of the performance of these parts of the code, in terms of the problem size N, and the number of processors P. It highlights potential scalability limitations, helping researchers to identify them and possibly eliminate them in future designs and implementations.

This study shows how the model is used to detect scalability limitations. We also propose and alternative way of imple-

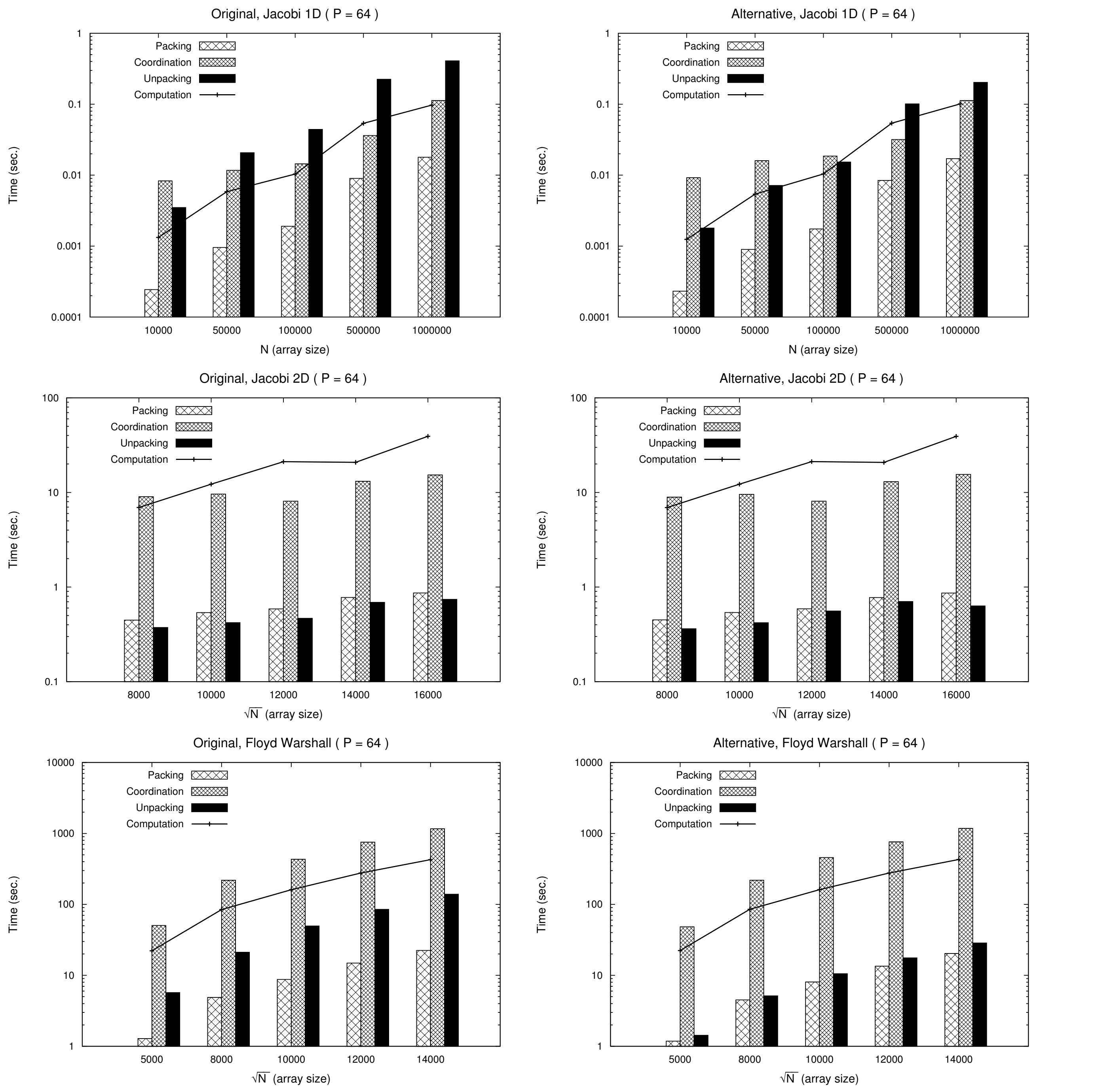

Figure 5. Execution times of the codes generated using the FOP scheme, with the original and the alternative π function implementation, for the three study cases: 1D Jacobi, 2D Jacobi, and Floyd-Warshall's algorithm.

menting the functions associated to the distribution policies, that eliminates a P factor from several parts of the cost model.

We present a case study, and experimental results with three study cases that confirm that the model is useful for asymptotic performance prediction of these communication management codes. The results show how the alternative implementation proposed highly alleviates one of the scalability problems.

This paper only covers the FOP *multicast* communication operations. Future work will present an extension of the model for both multicast and unicast operations, and a further study of the behaviour of more complex applications.

Acknowledgment

This research has been partially supported by the Ministerio de Economía y Competitividad (Spain) and the ERDF program of the European Union: CAPAP-H5 network (TIN2014-53522-REDT), MOGECOPP project (TIN2011-25639); the Junta de Castilla y León (Spain): ATLAS project (VA172A12-2).

References

[1] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. PACT'04*, pages 7–16. ACM Press, 2004.

[2] U. Bondhugula. Automatic distributed memory code generation using the polyhedral framework. Technical Report IISc-CSA-TR-2011-3, IISc, 2011.

[3] U. Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *Proc. SC'13*. ACM Press, 2013.

[4] U. Bondhugula. Pluto: An automatic parallelizer and locality optimizer for multicores. WWW, May 2013. on http://pluto-compiler.sourceforge.net.

[5] M. Claßen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 7–pp. IEEE, 2006.

[6] R. Dathathri, C. Reddy, T. Ramashekar, and U. Bondhugula. Generating efficient data movement code for hetetogeneous architectures with distributed memory. In *Proc. PACT'13*, pages 375–386. IEEE Press, 2013.

[7] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. Llanos. An extensible system for multilevel automatic data partition and mapping. *IEEE TPDS*, Mar 2013. doi:10.1109/TPDS.2013.83.

[8] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 supercomputer sites. WWW, June 2013. on http://www.top500.org/.

[9] L.-N. Pouchet. Polybench/c: the polyhedral benchmark suite. WWW, Mar 2013. on http://www.cse.ohio-state.edu/~pouchet/software/polybench/.