Combining Data and Computation Distribution Directives for Hybrid Parallel Programming

A Transformation System

Rachid Habel
TELECOM SudParis
habel.rachid@telecom-sudparis.eu

Frédérique Silber-Chaussumier
TELECOM SudParis
frederique.silber-chaussumier@telecom-sudparis.eu

François Irigoin
MINES ParisTech
francois.irigoin@mines-paristech.fr

Elisabeth Brunet
TELECOM SudParis
elisabeth.brunet@telecom-sudparis.eu

Abstract

This paper describes *dSTEP*, a directive-based programming model for hybrid shared and -distributed memory machines. The originality of our work is the definition and an implementation of a unified high-level programming model addressing both data and computation distributions, providing a particularly fine control of the computation. The goal is to improve the programmer productivity while providing good performances in terms of execution time and memory usage. We define a generic compilation scheme for computation mapping and communication generation. We implement the solution in a source-to-source compiler together with a runtime library. We provide a series of optimizations to improve the performance of the generated code, with a special focus on reducing the communications time. We evaluate our solution on several scientific kernels as well as on the more challenging NAS BT benchmark, and compare our results with the hand written Fortran MPI and UPC implementations.

The results show first that our solution allows to make explicit the non trivial parallel execution of the NAS BT benchmark using the dSTEP directives. Second, the results show that our generated MPI + OpenMP BT program runs with a 83.35 speedup over the original NAS OpenMP C benchmark on a hybrid cluster composed of 64 quadricores (256 cores). Overall, our solution dramatically reduces the programming effort while providing good time execution and memory usage performances. This programming model is suitable for a large variety of machines as multi-core and accelerator clusters.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords Distributed-memory, shared-memory, source-to-source transformation, OpenMP, MPI, Optimization

1. Introduction

Clusters of multi-core CPUs and hardware accelerators are currently the most powerful machines, as stated in the Top 500 classification [1]. These architectures offer very high theoretical performance peaks, but leveraging a sustained performance at a reasonable programming effort is the major challenge for end users. We propose dSTEP, a high-level directive based programming model with three goals:

1. offering good performances on hybrid parallel machines, compared to hand-written codes,
2. processing very large data sets,
3. simplifying programming by offering high-level directives.

The originality of our work is the definition and an implementation of a unified high-level programming model addressing both data and computation distribution with a fine control of the schedule constraints of the computation.

In this paper, we first study the related work on programming hybrid parallel machines in section 2. In section 3, we present the programming model of dSTEP. We detail our generic compilation scheme for dSTEP programs in section 4. In section 5, we present some optimizations focusing particularly on communications cost reduction. In section 6, we expose some experiments that validate our solution. We conclude and present our future work in section 7.

2. Related work

2.1 Libraries

MPI [2] is the standard API for programming distributed memory systems. It offers a rich support for process management, topology, collective and point-to-point communications as well as for one-sided communications since MPI-2. MPI is at a very low level of abstraction and requires an important programming and debugging effort. Global arrays [3] allow for the allocation of objects in the global space. Each processor can asynchronously read and update the global elements, and the necessary synchronizations for maintaining data coherency must be handled by the programmer. The

GASNet [4] library is used essentially as a runtime support for the PGAS languages. It focuses particularly on the efficient implementation of one-sided communications. StarPU [5] is a task-oriented library for hybrid programming. It schedules the computation efficiently on the different available computation resources. It offers functions on top of MPI for distributed memory programming and offers a support for data distribution in an element-wise fashion.

2.2 Programming languages

High Performance Fortran [6] (HPF) is the first standardization effort for programming distributed memory machines, constructed on top of the Fortran language. It offers a convenient set of directives for data distribution over virtual templates which are mapped to physical processors. The HPF compilers use the so-called *owner computes* rule to assign computation to processors. HPF compilers then automatically generate the necessary communications. The dHPF [7] compiler developed at Rice University brings many optimizations over former HPF compilers. It extends the owner computes rule to unions of *ON HOME* sets of arbitrary read or write references. dHPF also implements the multi-partitioning distribution together with optimizations like partial replication of local computation to avoid some communications. The HPF language did not meet the success intended by its designers for some of the reasons described by Kennedy *et al.* [8]. The *Partitioned Global Space Languages* (PGAS) aim at a better explicit programming of distributed memory systems at the language level. Co-Array Fortran [9] (CAF) is a PGAS based on Fortran 95 with the central notion of co-dimension. A co-dimension is the replication of the elements of the corresponding dimension on all the processors, called images. The replication results globally in a co-array which is accessible from any image. The elements on any specific image can be accessed by sub-scripting the co-array with the image identifier. A newer version of CAF, called CAF 2.0 [10] is developed at Rice University. It notably adds support for low level features inspired from MPI like sub-sets of images and topology. Unified Parallel C (UPC) [11] is another PGAS on top of the C programming language. Data objects are made globally addressable by declaring them as *shared*. Both CAF and UPC require the use of low-level synchronization constructs to ensure the coherency of the elements allocated in the global space. Chapel [12] and X10 [13] are two more recent PGAS. The Chapel language, developed by CRAY, uses the concept of *locale* to abstract the logical units of the parallel machine. Data can be defined on index domains of integers which are distributed over the locales. Chapel uses the concept of halo to declare replicated elements on distributed arrays. Computation distribution is expressed using the *forall* construct and can be controlled on each locale using the *on* clause. X10 offers support for both classic and user-defined data distributions over *places*. The distributed data is accessed by asynchronous activities launched on remote locales. While the PGAS languages offer an explicit control over data distribution, they put on the programmer the burden of synchronizing all the accesses to globally allocated data.

2.3 Compilation directives

OpenMP [14] is the *de facto* standard for programming shared memory parallel computers. It is a set of directives and runtime functions to exploit data parallelism essentially at the loop level. OpenMP 3.0 adds support for task parallelism. Even though OpenMP 4.0 adds support for accelerators programming in a distributed host/device environment, it does not allow to program distributed memory systems in general. Some research projects worked on the adaptation of OpenMP to clusters of CPUs like distributed OpenMP [15], Cetus [16] and STEP [17, 18] and even for clusters of GPUs like [19] and [20]. In these solutions, data are replicated on the compute nodes, limiting therefore the ability of processing very large data sets. OmpSs [21] is a task parallel programming model targeting several architectures. The implementation of OmpSs on clusters [22] is based on a centralized management of tasks and communications, resulting in a bottleneck. XcalableMP [23] offers separate directives for data and computation distributions, but requires from the programmer the use of explicit directives for communications. HMPP [24] is a programming model aimed at programming hardware acelerators which influenced the OpenACC programming model [25, 26]. These models target accelerators programming and do not handle distributed memory clusters.

2.4 Automatic solutions

The goal here is to hide the distribution to the programmer. *Software Distributed Shared Memory* (SDSM) solutions, like TreadMarks [27], emulate a logical shared memory over a physically distributed memory. It handles memory updtaes at the page level. These solutions suffer from important overheads due to page synchronization and updates. The PARADIGM [28] compiler tries to extract automatically the distribution of data of a program statically. However, automatically computing an optimum alignment and distribution of data has been proved NP-complete by Li and Chen [29]. Yuki *et al.* [30] use the polyhedral model to automatically distribute data by extending parametric loop tiling to distributed memory [31]. This solution is limited to static-control programs [32] and do not give hints on how to manage data distribution over several successive loop nests.

2.5 Our dSTEP solution

The existing solutions and programming models do not meet the goals mentioned in the introduction. Libraries are too low level and address generally only the distributed memory aspect. Programming languages put the burden of expressing and synchronizing distributed data accesses on the programmer. Compiler directives of current solutions address either shared-memory or accelerator targets. The extension solutions to distributed memory replicate the data. Transparent solutions like SDSM come with unpredictable performance, and automatic transformations based on the polyhedral model are restricted to static control programs and it is not clear how they handle the problem of data and computation distribution for an entire program.
In the next section, we introduce *dSTEP*, a new programming solution for better programming of hybrid parallel architectures.

3. Our programming model

In this section we present *dSTEP*, a directive-based programming model for dense scientific applications. We aim at distributing array data structures and loop nest computations. For this purpose, we propose the *distribute* directive, inspired by HPF, to distribute arrays and the *gridify* directive, inspired by HMPP, to distribute loop nest computations. The distribution of each array and each loop nest is made on a virtual grid of processors based on the available processors given at execution time.

3.1 Array distribution

The dSTEP distribute directive. It expresses a distribution type for each array dimension. In addition to the classical *block*, *cyclic*, *replicated* distribution types, we also propose the multi-partitioned distribution which allows blocks to be distributed along an array diagonal. A multi-partitioned distribution is indicated by the $diag$ qualifier. An example of such a distribution is given in example `e` of Figure 1.

The halo. In addition, we propose the definition of a *halo*. When defining a halo, the programmer defines some replicated data. A

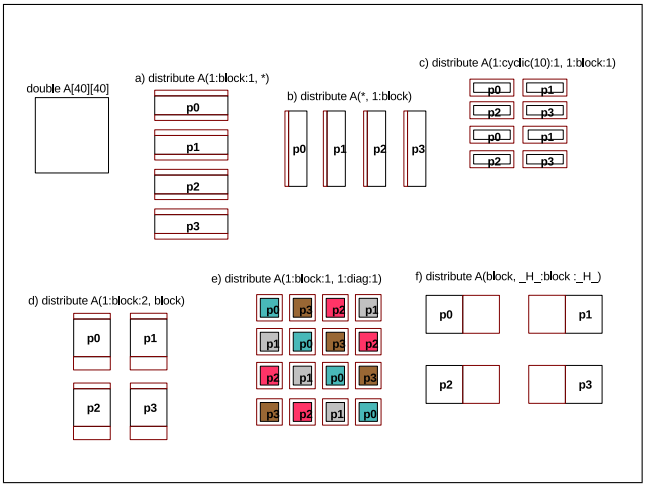

Figure 1. 2-D array distribution examples on 4 processors

Distribution type	Semantics
block	The loop iterations are divided into blocks of equal size, each block is assigned to a single processor for that dimension
cyclic(block_size)	The loop iterations are divided into blocks of equal size defined by the programmer and assigned in a round-robin fashion to processors
* (replicated)	All the iterations of the dimension are assigned to each processor in a single block
all(block_size)	The loop iterations are divided into blocks of equal size defined by the programmer. All the blocks are assigned to each processor.

Schedule type	Semantics
parallel	The iterations can be executed in parallel.
ordered	The iterations must be executed in the initial order.
owner	Only the processors on which the accessed elements are allocated execute the corresponding iterations.

Table 1. Semantics of loop distribution and schedule types

halo is composed of extra elements contiguous to a distributed dimension on each processor. The purpose of the halo is to satisfy the locality of all the data accesses of the program. Based on this halo our tool automatically generates the necessary communications when updates are necessary.
Listing 1 presents the syntax of the distribute directive. The optional quantity $hlow$ (resp. hup) is the lower (resp. upper) halo of an array dimension.

We define $\_H\_$ as a specific halo value that indicates a *total halo*: remaining elements of the dimension are spanned in the lower and upper directions (see example `f` of Figure 1).

```
1 #pragma dstep distribute A([hlow:]dist_type[:hup],
      ...)
```

Listing 1. dSTEP *distribute* directive syntax

3.2 Loop nest distribution

The dSTEP gridify directive. It offers the programmer a convenient way to express both the distribution type and the schedule type of any loop nest dimension. The *diag* qualifier indicates that the corresponding distribution is multi-partitioned.

```
1 #pragma dstep gridify(i1(dist=dist_type[,diag]; sched
      =sched_type),
2   i2(...), ...) [private(var_list)] [firstprivate(
      var_list)] [reduction(op:var_list)]
```

Listing 2. dSTEP *gridify* directive syntax

Table 1 defines the different possible values for the $dist$ and $sched$ attributes, as well as their semantics. The default values for $dist$ and $sched$ are block and parallel, respectively.

Differences with HPF. Unlike HPF, dSTEP does not use the notion of data owner. Array elements are distributed over processors and the halo introduces a certain level of replication. The computation distribution and scheduling are entirely expressed in the $gridify$ directive and each processor can read and update any element allocated locally. The dSTEP $owner$ schedule is different from the owner-computes rule too. In dSTEP, it means that for a given iteration slice only the processors on which the accessed elements are allocated do the computation and the others just skip it. For example, if a column of a matrix happens to be in the halo space on a processor, then this column is also allocated on another processor. An $owner$ update of this column results in the computation on both processors: there is no owner of the column, and there is no communication triggered along this dimension.

4. Compilation

4.1 PIPS

PIPS [33, 34] is an open source compilation workbench that implements powerful interprocedural, data-based program analyses and transformations. PIPS is used, among other applications, to parallelize scientific programs. One important building blocks of PIPS are the array regions [35]. An array region is a set of array elements described by a system of linear constraints defining a convex polyhedron. Array regions are used to represent program statements effects on array elements. Depending on the action upon the array elements, array regions can be of type *READ* or *WRITE*. To represent the flow of array elements through the program, two other types of array regions, *IN* and *OUT*, are defined in PIPS. For a given portion of code, an *IN* region is the *READ* region for this portion of code that is used before possibly being redefined. An *OUT* region is a *WRITE* region the elements of which are used at some future point of the program. In this work, we use *READ* and *WRITE* regions to compute the sets of accessed elements, necessary for the compilation of iterations slices and the *OUT* regions used to generate the communications.
Let us notice that these array regions analyses are limited to affine expressions in array subscripts.

4.2 Virtual grids of processors

Array elements and loop nests iterations are distributed over virtual grids of processors constructed with the functions defined below.

Simple grid of processors. Function id_in_grid maps a flat processor identifier in the set $\mathcal{P} = [0, nb\_procs[$ to a vector identifier in a virtual grid of processors defined by the set $\mathcal{P}'_{grid} = \{\vec{0} \leq \vec{p} < P'\vec{1}\}$ where P' is a diagonal matrix containing the number of processors along each dimension of the grid. P' and id_in_grid are defined using MPI topology functions.

Extended virtual grid of processors. Function $extend\_id$ (eq. 1) maps an identifier in a virtual grid and a virtual processor number to an identifier in an *extended* virtual grid of processors defined

by the set $\mathcal{P}_{grid} = \{\vec{p}|\vec{0} \leq \vec{p} < P\vec{1}\}$. The diagonal matrix P is constructed from the matrix P' by adding an extra dimension corresponding to the diagonalized dimension. The number of virtual processors along the diagonalized dimension if any, $vprocs$, is equal to the minimum of processors along the other dimensions of the grid, excluding units (corresponding to replicated dimensions). We define the set $\mathcal{V}$ as $\mathcal{V} = [0, vprocs[$. mod_v is the component-wise modulo operator for vectors.

If no dimension is diagonalized then the distribution is not multi-partitioned and we get $vprocs = 1$ and $extend\_id$ be the identity function. We handle at most one diagonalized dimension.

$$\begin{array}{rl} extend\_id: & \mathcal{P}'_{grid} \times \mathcal{V} \rightarrow \mathcal{P}_{grid} \\ & (\vec{p'}, v) \rightarrow (\vec{p'} + v\vec{1}) mod_v P\vec{1} \end{array} \quad (1)$$

We illustrate the construction of the virtual grid of processors using the id_in_grid and $extend\_id$ functions in Figure 2. Th example shows a multi-partitioned distribution of a three-dimensional array. The *distribute* directive indicates that the third dimension of array A is diagonalized. If the program is executed on four processors, a 3-dimensional extended virtual grid of processors is constructed from the flat four processors as illustrated on the left of Figure 2. In this example, we have $vprocs = min(2, 2) = 2$. On the right part of the figure, we show how the elements of array A are distributed on each processor.

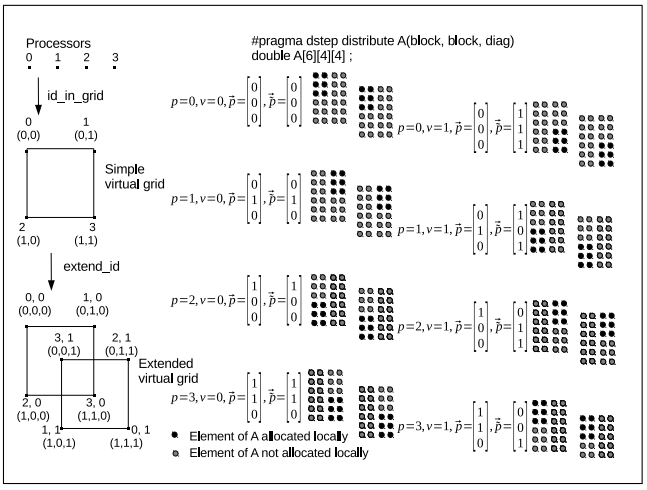

Figure 2. Example of array distribution over an extended virtual grid of processors using a multi-partitioned distribution

4.3 Distribution of array elements

We use a linear algebra model inspired by the work of Coehlo *et. al* [36]. Compared to their model, we do not use templates but we introduce the halo and the multi-partitioning support. For an array x, the diagonal matrices B_x, P_x and $Hlow_x$ contain respectively the block sizes, the processors of the virtual grid and the lower halos. Equation 2 maps an element in the distributed domain, $(\tilde{p}_x, c_x, l_x)$, to an element a in the sequential domain. It defines the $array\_element$ function. The first term of the equation defines the cycles offset, the second term defines the blocks offset, the third one is the lower halo correction and finally the $\vec{l_x}$ vector is the local offset within the local memory.

$$\vec{a} = B_x P_x \vec{c}_x + B_x \vec{\tilde{p}}_x - Hlow_x \vec{1} + \vec{l}_x \quad (2)$$

4.4 Distribution of loop iterations

The distribution of loop iterations is defined with a similar model, where the diagonal matrices B_{ln} and P_{ln} define the blocks and the processors in a virtual grid for the loop nest ln. The diagonal matrix L_{ln} (resp. U_{ln}) contains the lower (resp. upper) bounds of the iteration space of ln. Equation 3 defines the $loop\_nest\_iteration$ function.

$$\vec{i} = L_{ln}\vec{1} + B_{ln}P_{ln}\vec{c}_{ln} + B_{ln}\vec{\tilde{p}}_{ln} + \vec{l}_{ln} \quad (3)$$

4.5 Other definitions

The following definitions will be used in the compilation scheme. Function $iteration\_slice$ computes the set of iterations for an extended processor identifier and a cycle of a loop nest. The loop nest ln is represented by the set $ln = \{\vec{i}|L_{ln}\vec{1} \leq \vec{i} < U_{ln}\vec{1}\}$. $\mathcal{C}_{ln} = \{\vec{c}|\vec{0} \leq \vec{c} < \lceil (U_{ln} - L_{ln}) B_{ln}^{-1} P_{ln}^{-1} \vec{1} \rceil\}$ is the set of cycles of the loop nest.

$$\begin{array}{rl} iterations\_slice: & \mathcal{P}_{ln} \times \mathcal{C}_{ln} \rightarrow \wp(ln) \\ & (\vec{\tilde{p}}_{ln}, \vec{c}_{ln}) \mapsto \{\vec{i}| \\ & loop\_nest\_iteration(\vec{\tilde{p}}_{ln}, \vec{c}_{ln}, \vec{0}) \leq \vec{i} \wedge \\ & \vec{i} < loop\_nest\_iteration(\vec{\tilde{p}}_{ln}, \vec{c}_{ln}, B_{ln}\vec{1}) \wedge \\ & L_{ln}\vec{1} \leq \vec{i} \wedge \vec{i} < U_{ln}\vec{1}\} \end{array} \quad (4)$$

Function $belongs\_to$ finds the cycle and the virtual processor to which a rectangular region of array elements belong. Function $extend\_replicated$ extends the iteration slice along the replicated dimensions to the original extent in ln. Function $owner\_dimensions$ constructs the set of array dimensions accessed by loop nest indexes with an owner schedule. Function $project$ eliminates from a vector the dimensions contained in a set of dimensions. Function *pos* computes the position of a vector in a lexicographically ordered set of vectors.

4.6 Representation of a loop nest

Algorithm 1 represents of a loop nest with a *body* statement containing references to distributed arrays x represented by syntactic objects X. The m^{th} reference to array x is represented by a function ref_x^m of the iteration vector $\vec{i}$.

Algorithm 1 Syntactical representation of a loop nest annotated with the *gridify* directive

#pragma dstep gridify(...) [private(var_list)] [firstprivate(var_list)] [reduction(op:var_list)]
for $\vec{i} \in \{\vec{i}|L_{ln}\vec{1} \leq \vec{i} < U_{ln}\vec{1}\}$ with increment $\vec{inc}$ **do**
  $body(..., X[ref_x^m(\vec{i})], ...)$
end for

4.7 Data allocation

From an original array x of dimension d and of sizes expressed in a diagonal matrix D, we generate the distributed array allocation as defined by Algorithm 2. The generated declaration of a distributed array is augmented by two extra dimensions corresponding to the cycles and the virtual processors to handle multi-partitioning.

Algorithm 2 Data allocation for a distributed array

Input: $D, B, Hlow, Hup, \mathcal{C}, \mathcal{V}$
Output: The replacement of the original declaration by a distributed one
1: $generate\_allocation(X[D_{0,0}]...[D_{d-1,d-1}]) \equiv$
2: **for** each dimension k of array x **do**
3: **if** dimension k has a total halo **then** $L_{k,k} = D_{k,k}$
4: **else**
5: $L_{k,k} = Hlow_{k,k} + B_{k,k} + Hup_{k,k}$
6: **end if**
7: **end for**
8: X[$|\mathcal{C}|$][$|\mathcal{V}|$][$L_{0,0}$]...[$L_{d-1,d-1}$]

4.8 Computation mapping

We distinguish the compilation of a parallel loop nest, which has no ordered dimension and the compilation of an ordered loop nest, which has at least one ordered dimension.

Parallel loop nest. The compilation of a parallel loop nest is described in Algorithm 3. We first enumerate and compile the iteration slices by scanning the virtual processor and the cycle dimensions. We generate the code corresponding to the $send$ communications (see Section 4.9.1) of the iteration slice just after its compilation. After that, we generate the code for distributed reductions on scalar variables if any. Finally, we generate the *receive* communications (see Section 4.9.2) and the completion of the send and receive communications.

Algorithm 3 $compile\_parallel$

Input: $ln\_stmt, body, \mathcal{V}_{ln}, \mathcal{C}_{ln}$
Output: ln_stmt compiled for a hybrid target

```
1:  p = rank(); p⃗_ln = id_in_grid_ln(p)
2:  for v_ln ∈ V_ln do                                   ▷ No order constraint
3:      p⃗_ln = extend_id(p⃗_ln, v_ln)
4:      for c⃗_ln ∈ C_ln do                               ▷ No order constraint
5:          compile_iteration_slice(ln_stmt, body, p, p⃗_ln, c⃗_ln)
6:          generate_sends(ln_stmt, p, p⃗_ln, c⃗_ln)
7:      end for
8:  end for
9:  if reduction then
10:     generate_parallel_reductions(op, var_list)
11: end if
12: generate_recvs(ln_stmt, p, ∅)
13: generate_completes(ln_stmt)
```

Ordered loop nest. The compilation of an ordered loop nest presented in Algorithm 4 is more complicated because it has to ensure the conservation of the sequential order of the ordered dimensions. First, we generate *recv* communications for iteration slices which have some other precedent iteration slices, defined by the *Before* set (line 10). Here, we generate a blocking receive to ensure that the computation does not proceed until the complete reception of the dependency data. We then save the precedent iteration slice to not be reconsidered in the remaining *recv* communications. The compilation of each iteration slice and the generation of the corresponding *send* communications is identical to the parallel case. After the loop nest, we generate the code for distributed reductions if any and the code for *recv* communications for the iteration slices of remote processes which were not considered in the blocking phase of the $recvs$ (lines 21-23). Finally, the communications completes are generated.

Compilation of an iteration slice. The compilation of an iteration slice (see Algorithm 5) consists of first finding the locally accessed memory for each array. This is done by calling the $belongs\_to$ function. If the accessed elements are not allocated locally, than we raise a bad distribution exception. In presence of a loop nest owner dimension, we check that all the accessed elements for each array are either allocated locally or not allocated at all. We then perform a domain change, by translating the sequential references into the distributed domain (line 28). We note that each processor executes its assigned iterations (line 27).

Exploiting the different cores on a shared memory node. The iteration vector $\vec{i}$ combines all the distributed dimensions of the loop nest. We insert an *omp parallel for* directive at the first parallel dimension of vector $\vec{i}$. If there is an enclosing ordered dimension, we insert a *firstprivate* clause containing the appropriate variables, particularly the enclosing loop indexes. A *private* clause is automatically created for all the indexes specified in the *gridify* directives

Algorithm 4 $compile\_ordered$

Input: $ln\_stmt, body, \mathcal{V}_{ln}, \mathcal{C}_{ln}$
Output: ln_stmt compiled for a hybrid target

```
1:  p = rank()
2:  p⃗_ln = id_in_grid_ln(p)
3:  for v_ln scan_v V_ln do        ▷ Enumeration constrained by the ordered dimensions
4:      p⃗_ln = extend_id(p⃗_ln, v_ln)
5:      for c⃗_ln scan_c C_ln do    ▷ Enumeration constrained by the ordered dimensions
6:          iteration_set = iteration_slice(p⃗_ln, c⃗_ln)
7:          for each array X written in ln_stmt do
8:              p⃗_x = id_in_grid_x(p)
9:              for (p⃗'_ln, c⃗'_ln) ∈ Before(p⃗_ln, c⃗_ln) do
10:                 accessed_x = read_region(x, body, iteration_set) ∪
    write_region(x, body, iteration_set)
11:                 (c⃗_x, v_x) = belongs_to(p⃗_x, accessed_x)
12:                 generate_sync_recv(stmt_ln, p⃗_x, c⃗_x, p⃗'_ln, c⃗'_ln)
13:                 add(Received, x, (p⃗_x, c⃗_x), (p⃗'_ln, c⃗'_ln))
14:             end for
15:         end for
16:         compile_iteration_slice(body, p, p⃗_ln, c⃗_ln)
17:         generate_sends(ln_stmt, p, p⃗_ln, c⃗_ln)
18:     end for
19: end for
20: if reduction then
21:     generate_parallel_reductions(op, var_list)
22: end if
23: generate_recvs(ln_stmt, p, Received)
24: generate_completes(ln_stmt)
```

Algorithm 5 $compile\_iteration\_slice$

Input: $body, p, \vec{p}_{ln}, \vec{c}_{ln}$
Output: the code of an iteration slice for a hybrid target

```
1:  iteration_set = iteration_slice(p⃗_ln, c⃗_ln)
2:  for each owner dimension k in loop nest ln do
3:      extend(iteration_set, k)
4:  end for
5:  computes = true
6:  skips = true
7:  for each array x referenced in ln_stmt do
8:      p⃗_x = id_in_grid_x(p)
9:      accessed_x = read_region(x, body, iteration_set) ∪
    write_region(x, body, iteration_set)
10:     (c⃗_x, v_x) = belongs_to(p⃗_x, accessed_x)
11:     if element_undefined((c⃗_x, v_x)) then
12:         if there is no owner dimension then
13:             Abort("Bad distribution")
14:         else
15:             computes = false
16:         end if
17:     else
18:         p⃗_x = extend_id(p⃗_x, v_x)
19:         shift_x = array_element(p⃗_x, c⃗_x, 0⃗)
20:         skips = false
21:     end if
22: end for
23: if !(computes ⊕ skips) then                          ▷ Xor
24:     Abort("Bad distribution")
25: end if
26: if computes then
27:     for i⃗ ∈ iteration_set with increment inc⃗ do
28:         body(..., X[pos(c⃗_x, C_x)][v_x][ref^m_x(i⃗) − shift_x], ...)
29:     end for
30: end if
```

and for any innermost indexes which are not specified. In the presence of a *reduction* clause, the same clause is generated with the corresponding OpenMP directive.

4.9 Communication Generation

The generation of the communications uses the halos declared for each array and the PIPS *WRITE* and *OUT* regions. A *send* is a point-to-point communication message which allows a processor to propagate the updates of array elements to another processor which uses the value of these elements in a future computation. A *receive* communication is the symmetric operation of a send communication.

4.9.1 Generating send communications.

The generation of send communications is described in Algorithm 6. For each iteration slice, we compute the written elements which are forwardly exposed in the program based on the OUT regions (line 12). We then scan the allocated memory on the other processors and compute the intersection between the locally live region and the remote memory for each array. In fact, we scan only the processors with common replicated elements for each array, called *neighbours*, known thanks to distribution information. For non-empty intersections, a non-blocking MPI send communication is generated.

Algorithm 6 $generate\_sends$

Input: $ln\_stmt, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln}$
Output: ln_stmt with send communications inserted

1: $generate\_sends \equiv$
2: $\vec{p}_{ln} = id\_in\_grid_{ln}(p)$
3: $iteration\_set = loop\_nest\_iterations(\vec{\tilde{p}}_{ln}, \vec{c}_{ln})$
4: **for** each owner dimension k in loop nest ln **do**
5: $\quad iteration\_set = extend(iteration\_set, k)$
6: **end for**
7: **for** each array x written in ln_stmt **do**
8: $\quad owner_x = owner\_dimensions(ln\_stmt, x)$
9: $\quad \vec{p}_x = id\_in\_grid_x(p)$
10: $\quad write_x = write\_region(x, body, iteration\_set)$
11: $\quad out_x = out\_region(x, body)$
12: $\quad live_x = write_x \cap out_x$
13: $\quad$ **if** $live_x \neq empty\_region$ **then**
14: $\quad\quad (\vec{c}_x, v_x) = belongs\_to(\vec{p}_x, live_x)$
15: $\quad\quad$ **if** $element\_defined(\vec{c}_x, v_x)$ **then**
16: $\quad\quad\quad \vec{\tilde{p}}_x = extend\_id(\vec{p}_x, v_x)$
17: $\quad\quad\quad$ **for** $p' \in \mathcal{P}$ **do** ▷ scanning other processors
18: $\quad\quad\quad\quad \vec{\tilde{p}'}_x = id\_in\_grid_x(p')$
19: $\quad\quad\quad\quad$ **for** $v'_x \in \mathcal{V}_x$ **do**
20: $\quad\quad\quad\quad\quad \vec{\tilde{p}'}_x = extend\_id(\vec{\tilde{p}'}_x, v'_x)$
21: $\quad\quad\quad\quad\quad$ **if** $project(\vec{\tilde{p}'}_x, owner_x) \neq project(\vec{\tilde{p}}_x, owner_x)$ **then**
22: $\quad\quad\quad\quad\quad\quad$ **for** $\vec{c'}_x \in \mathcal{C}_x$ **do**
23: $\quad\quad\quad\quad\quad\quad\quad to\_send = live_x \cap \{\vec{a} | array\_element(\vec{\tilde{p}'}_x, \vec{c'}_x, \vec{0}) \leq \vec{a} < array\_element(\vec{\tilde{p}'}_x, \vec{c'}_x, L_x\vec{1})\}$
24: $\quad\quad\quad\quad\quad\quad\quad$ **if** $to\_send \neq \emptyset$ **then**
25: $\quad\quad\quad\quad\quad\quad\quad\quad \vec{shift}_x = array\_element(\vec{\tilde{p}}_x, \vec{c}_x, \vec{0})$
26: $\quad\quad\quad\quad\quad\quad\quad\quad async\_send(X[pos(\vec{c}_x, \mathcal{C}_x)][v_x], p', to\_send, \vec{shift}_x)$
27: $\quad\quad\quad\quad\quad\quad\quad$ **end if**
28: $\quad\quad\quad\quad\quad\quad$ **end for**
29: $\quad\quad\quad\quad\quad$ **end if**
30: $\quad\quad\quad\quad$ **end for**
31: $\quad\quad\quad$ **end for**
32: $\quad\quad$ **end if**
33: $\quad$ **end if**
34: **end for**

4.9.2 Generating receive communications.

The generation of receive communications is symmetric to the generation of send communications. There are two main differences:

1. We pass as a parameter to the *generate_recvs* function the *Received* set, which contains the remote iteration slices the contribution of which have already been considered (see the compilation of an ordered loop nest in Algorithm 4),
2. In the blocking version of the recv communication generation, we indicate explicitly for which local memory block to consider the contribution of a specific remote iteration slice.

5. Optimizations

5.1 Extension of the loop iteration domain

Iteration space boundaries and subscript expressions may result in accesses which require a greater halo than expected. This situation is illustrated in Figure 3 for an execution of the code of Listing 3 on three processors. The access functions for arrays A and B will require a halo greater than 1. This leads to additional communications as the modified elements are forwardly exposed. We can avoid these communications by applying an extension of the iteration space of dimension i. We introduce the optional *ext* clause to indicate, for each dimension of a loop nest, an extension to the lower and upper bound of the dimension. The goal of this optimization is to adjust the iteration space for a better fit with the distributed elements of the accessed arrays. This optimization does not imply any redundant computation; it yields to a smaller halo and minimizes the induced communication. In our example, the use of the *ext* clause (line 9) reduces the needed halo for array B (line 2), and totally eliminates the needed halo and thus the communication on array A (line 5).

```
1  #pragma dstep distribute A(2:block:2)
2  //#pragma dstep distribute A(block)
3  double A[15];
4  #pragma dstep distribute B(3:block:3)
5  //#pragma dstep distribute B(1:block:1)
6  double B[15];
7
8  #pragma dstep gridify (i)
9  //#pragma dstep gridify (i(ext=3,1))
10 for(i=3; i<14; i++)
11    A[i] = B[i-1] + B[i+1];
12 ...
```

Listing 3. Using the *ext* clause to adapt the iteration space extent

5.2 Static adaptation of the halo

The purpose of this optimization is to reduce the visible halo of an array at some points of the program to reduce the induced communication on that array. At the declaration of the array, the programmer declares, for each dimension, a halo which is supposed to satisfy the locality of all the accesses to that array in the program. The replicated elements must be updated each time they are modified and are forwardly exposed. The aim of this optimization is to control the amount of halo which is really necessary at some points of the program by *hiding* the unnecessary halo. We apply a def-use analysis on the halos. If in a path on the call graph, an halo is defined and killed several times, its value is set to zero until the next point before a possible use. For the time being, this optimization is applicable for array dimensions: 1) which are accessed by loop indexes spanning an iteration space identical to the array dimension extent, either directly or by applying the iteration space extension optimization, and 2) which are accessed by array subscripts of the

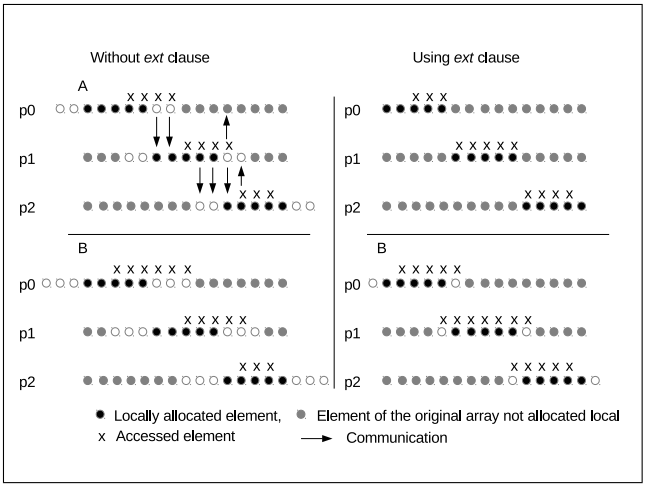

Figure 3. Iteration space extension optimization

form $a * i + b$ where $a = 1$. In addition, the possible computed values for the halo are either the original value or the zero value. We are working on extending this optimization to find finer results in the range $[0, original\_halo\_val]$.

5.3 Loop interchange

In the presence of enclosing ordered dimensions, the first parallel dimension if any is annotated with an *omp parallel for* directive with the enclosing indexes passed in a *firstprivate* clause. Teams of OpenMP threads are created and destructed for each enclosing ordered dimension. Based on the direction vector of a loop nest [37], it is always legal to exchange the first parallel dimension with the outermost ordered dimension, because the dependence will still be carried by the ordered dimension on each process. As a result, the OpenMP team of threads is created only once, saving a great overhead. As a counterpart, the loop interchange introduces some locality penalty if strided accesses are introduced. The actual benefits of this optimization depend on the code.

5.4 Overlapping communication and computation

By construction, the generated code of dSTEP overlaps the computation and communication of different iteration slices of a single loop nest. In addition we provide two communication completion strategies: explicit and implicit. The explicit strategy is as described in the compilation scheme above. In the implicit strategy, we do not generate code for communication completions at the end of a loop nest; the completions are done by the $belongs\_to$ function, which is called before any computation. Theoretically, the latter strategy allows for a maximum overlap by delaying any completion to the very point before the data are used, but communication engines do not always give the expected gains (see [38] for more details).

5.5 An example of generated code

In Listing 4 we show the use of the dSTEP *gridify* directive to distribute the computation of the $y\_backsubstitute$ of the NAS BT program using a multi-partitioned distribution for dimension k. The i and j dimensions are distributed by blocks. The j dimension is ordered, i and k are parallel.

The generated code for the $y\_backsubstitute$ function is showed in Listing 5. In line 1, the information describing the loop nest distribution is computed and cached using a generated loop nest identifier (64 in this case). In line 3, the virtual processors are scanned. Since the distribution is not cyclic, there is a single cycle for the loop nest (line 4). In line 6 to 8, the bounds of the

```
1 #pragma step gridify(i, j(dist=block; sched=ordered), k(dist=block, diag;
      sched=parallel))
2   for (i = 1; i < grid_points[0]-1; i++) {
3     for (j = grid_points[1]-2; j >= 0; j--) {
4       for (k = 1; k < grid_points[2]-1; k++) {
5         for (m = 0; m < BLOCK_SIZE; m++) {
6           for (n = 0; n < BLOCK_SIZE; n++) {
7             rhs[i][j][k][m] = rhs[i][j][k][m]
8               - lhs[i][j][k][CC][m][n]*rhs[i][j+1][k][n];
9           }}}}}
```

Listing 4. Distribution of the $y\_backsubsitute$ function of BT

current iteration slice are computed. Lines 11 to 26 correspond to the *Pre-recvs* on array rhs imposed by the ordered dimension j. The membership to the $Before$ set, which defines the iteration slices that must be executed before the current one is implemented using the predicate in line 19. Lines 31 to 41 show the computation corresponding to the iteration slice in the distributed domain. The cycles dimensions of arrays lhs and rhs are always accessed with a zero subscript since the distribution of these arrays is not cyclic. Lines 42 to 43 correspond to the generated send communications. The not shown *Post-recvs* phase is similar to the *Pre-recvs* and is done for iteration slices which are not in the $Before$ set. Finally, the communications on array rhs are completed (line 49).

6. Experiments

We present several experiments validating the different aspects of our dSTEP prototype. Matrix Multiplication and Jacobi illustrate the halo. Polybench Adi and NAS BT both compare standard block distribution and a multi-partitioned distribution. On the NAS BT, we show the benefit of the optimizations described in section 5. We generate the hybrid distributed and -shared memory programs and measure the execution times on a cluster of the Grid5000 platform. We use 32 bi-processor quadri-core Intel Xeon E5520 nodes connected with an Infiniband network.

6.1 Matrix multiplication

We consider the naive algorithm of matrix multiplication of the form $C = C + A * B$. There is an initialization phase of the matrices A, B and C and a final phase which computes and displays the trace of matrix C. We use a two-dimensional distribution for matrices A, B and C and use a total halo for the second dimension of matrix A and for the first dimension of matrix B to satisfy the accesses of the innermost dimension of the multiplication loop nest, k. For computation distribution we evaluate two strategies: 1) the distribution of the first two parallel dimensions i and j by blocks, and 2) in addition to the distribution of the i and j dimensions, we distribute the k dimension using an $all(block\_size)$ distribution. We obtain better performance for the second strategy over the first. With matrices of size 2048×2048, we obtain a speedup of 151.2 on 256 cores using a block size of 16. With matrices of size 9196×8196, the speedup on 256 cores is of 181.83 using the first strategy and of 1632 using the second one. This spectacular speedup is due to the temporal locality obtained by blocking the innermost k dimension.

6.2 Jacobi Kernel

The Jacobi kernel iterates over a 2-D stencil computation pattern until the convergence of the norm, which is computed using a parallel reduction. This program contains two 2-D matrices which are distributed using a 2-D block distribution and using a halo of 4 for both matrices on each dimension. The computation is distributed using a 2-D block parallel distribution. Figure 5 shows the execution times for 8196×8196 matrices. We obtain a speedup of 192.4 on 256 cores.

```
1    DSTEP_DISTRIBUTE_LOOP(64, _C, 2, 3, DSTEP_INT_DIV(grid_points
        [0]-1-1+1-1, _P[0], 1), DSTEP_INT_DIV(grid_points[1]-2+1-0, _P[1], 1)
        , DSTEP_INT_DIV(grid_points[2]-1-1+1-1, _P[2], 1));
2    _v = DSTEP_FIRST_VPROC(_P, _p, _vprocs, 1, -1);
3    for (___v = 0;___v<_vprocs;___v++) {
4       int _c[3] = {0, 0, 0};
5       DSTEP_EXTEND_ID(3, _P, _p, _v, _t_p);
6       DSTEP_LOOP_BOUNDS(3, _LOW, _UP, _v, _P, _c, _t_p, _L, _U, _INCR);
7       int DSTEP_i_LOW = _LOW[0], DSTEP_i_UP = _UP[0], DSTEP_j_LOW =
8       _LOW[1], DSTEP_j_UP = _UP[1], DSTEP_k_LOW = _LOW[2], DSTEP_k_UP = _UP
        [2];
9       int _ACCESSED_rhs[4][2] = {{DSTEP_GENERIC_MAX(2, DSTEP_i_LOW, 1),
        DSTEP_i_UP}, {DSTEP_GENERIC_MAX(2, 0, DSTEP_j_LOW), DSTEP_j_UP+1}, {
        DSTEP_GENERIC_MAX(2, DSTEP_k_LOW, 1), DSTEP_k_UP}, {0, 4}};
10      DSTEP_BELONGS_TO(rhs, _DIMS_rhs, &_c_rhs, &_v_rhs, _c_VECT_rhs,
        _shift_rhs, _ACCESSED_rhs, 0, &_computes, 1);
11      //Pre recvs
12      DSTEP_NEIGHBOURS(rhs, &_NB_N_rhs, &_N_rhs);
13      for (_n_rhs = 0;_n_rhs<_NB_N_rhs;_n_rhs += 1) {
14         __rank = _N_rhs[_n_rhs];
15         DSTEP_PROC_ID(64, __rank, 3, __p);
16         for (__v = 0;__v<_vprocs;__v++) {
17            int __c[3] = {0, 0, 0};
18            DSTEP_EXTEND_ID(3, _P, __p, __v, __t_p);
19            if (_t_p[1]<_P[1]-1&&__t_p[1]==_t_p[1]+1&&(__t_p[0]-_t_p
        [0]>=-1&&__t_p[0]-_t_p[0]<=1)&&(__t_p[2]-_t_p[2]>=-1&&__t_p[2]-_t_p
        [2]<=1)) {
20             DSTEP_LOOP_BOUNDS(3, _LOW, _UP, _v, _P, __c, __t_p, _L, _U,
        _INCR);
21             int DSTEP_i_LOW = _LOW[0], DSTEP_i_UP = _UP[0],
22             DSTEP_j_LOW = _LOW[1], DSTEP_j_UP = _UP[1], DSTEP_k_LOW =
23             _LOW[2], DSTEP_k_UP = _UP[2];
24             int __recv_rhs[4][2] = {{DSTEP_GENERIC_MAX(2, DSTEP_i_LOW,
        1), DSTEP_i_UP}, {DSTEP_GENERIC_MAX(2, DSTEP_j_LOW, 0), DSTEP_j_UP},
        {DSTEP_GENERIC_MAX(2, DSTEP_k_LOW, 1), DSTEP_k_UP}, {0, 4}};
25             DSTEP_SYNC_RECV(rhs, __rank, __recv_rhs, sizeof(double),
        _OWNERS_rhs, _v_rhs, 0);
26          }}}
27      int _ACCESSED_lhs[6][2] = {{DSTEP_GENERIC_MAX(2, DSTEP_i_LOW, 1),
        DSTEP_i_UP}, {DSTEP_GENERIC_MAX(2, DSTEP_j_LOW, 0), DSTEP_j_UP}, {
        DSTEP_GENERIC_MAX(2, DSTEP_k_LOW, 1), DSTEP_k_UP}, {2, 2}, {0, 4},
        {0, 4}};
28      DSTEP_BELONGS_TO(lhs, _DIMS_lhs, &_c_lhs, &_v_lhs, _c_VECT_lhs,
        _shift_lhs, _ACCESSED_lhs, 0, &_computes, 1);
29
30      if (_computes) {
31        for(i = DSTEP_i_LOW; i <= DSTEP_i_UP; i += 1)
32 #pragma omp parallel for private(j, k, m, n) firstprivate(i)
33           for(j = DSTEP_j_UP; j >= DSTEP_j_LOW; j += -1)
34              for(k = DSTEP_k_LOW; k <= DSTEP_k_UP; k += 1)
35                 for(m = 0; m <= 4; m += 1)
36                    for(n = 0; n <= 4; n += 1){
37 rhs[0][_v_rhs][i-_shift_rhs[0]][j-_shift_rhs[1]][k-_shift_rhs[2]][m] =
38 rhs[0][_v_rhs][i-_shift_rhs[0]][j-_shift_rhs[1]][k-_shift_rhs[2]][m] -
39 lhs[0][_v_lhs][i-_shift_lhs[0]][j-_shift_lhs[1]][k-_shift_lhs[2]][2][m][n]*
40 rhs[0][_v_rhs][i-_shift_rhs[0]][j+1-_shift_rhs[1]][k-_shift_rhs[2]][n];
41                 }
42        int __send_rhs[4][2] = {{DSTEP_GENERIC_MAX(2, DSTEP_i_LOW, 1),
        DSTEP_i_UP}, {DSTEP_GENERIC_MAX(2, DSTEP_j_LOW, 0), DSTEP_j_UP}, {
        DSTEP_GENERIC_MAX(2, DSTEP_k_LOW, 1), DSTEP_k_UP}, {0, 4}};
43        DSTEP_SEND(rhs, _rank, _NB_NODES, _c_rhs, _v_rhs, _c_VECT_rhs,
        __send_rhs, sizeof(double), _OWNERS_rhs);
44      }
45      _v = (_v-1+_vprocs)%_vprocs;
46   }
47   //Post recvs
48      ...
49   DSTEP_COMPLETE_COMM(rhs);
```

Listing 5. dSTEP generated code for the $y\_backsubsitute$

6.3 Polybench Adi

The Polybench Adi [39] presents an alternate line and column sweep pattern. We use a 2-D distribution for arrays and loop nests, with two distribution strategies: 1) the two dimensions are distributed using block distribution (standard distribution), 2) the first dimension is distributed by blocks, and the second one is distributed using a diagonalized distribution, resulting in a multi-partitioned distribution. Figure 6 shows the execution times for a large 16000×16000 data set. We observe that until 128 cores, the multi-partitioned distribution outperforms the standard one. On 128 cores, the speedups are 39.4 and 32.26 and on 256 cores the speedups are 51.6 and 41.08 for the two strategies respectively.

6.4 NAS BT

The NAS Parallel benchmarks [40] are a set of scientific programs designed by the NASA to evaluate the performance of supercomputers. The NAS BT application solves three-dimensional Euler/Navier-Stokes equations by sweeping alternatively along the x, y and z dimension. The BT program contains a succession of

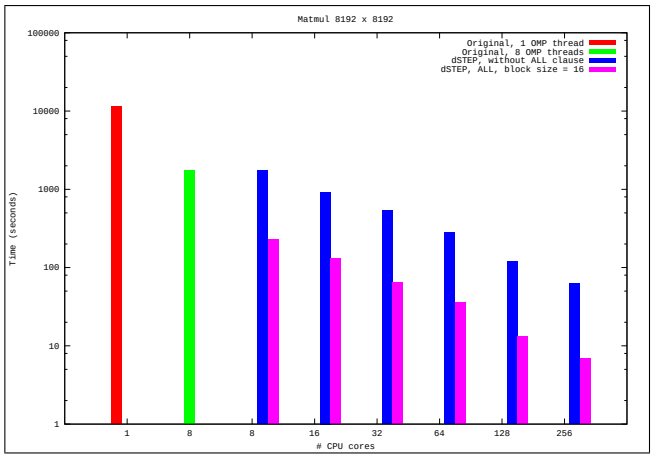

Figure 4. Matrix multiplication execution times

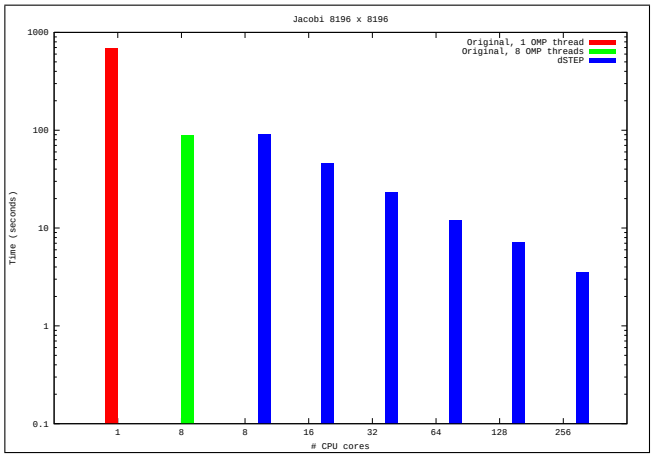

Figure 5. Jacobi execution times

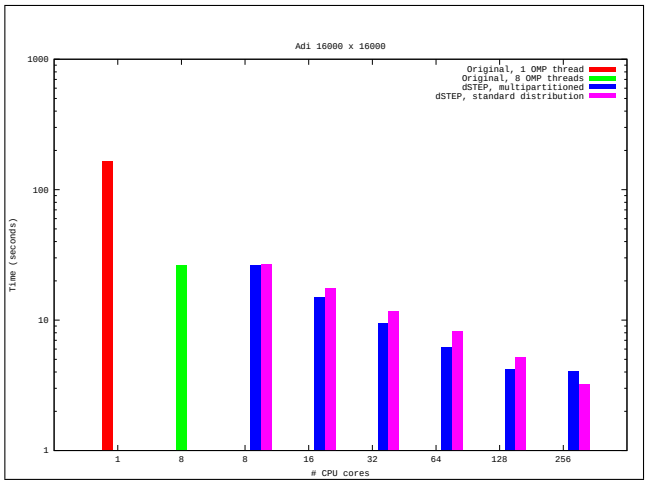

Figure 6. Adi execution times

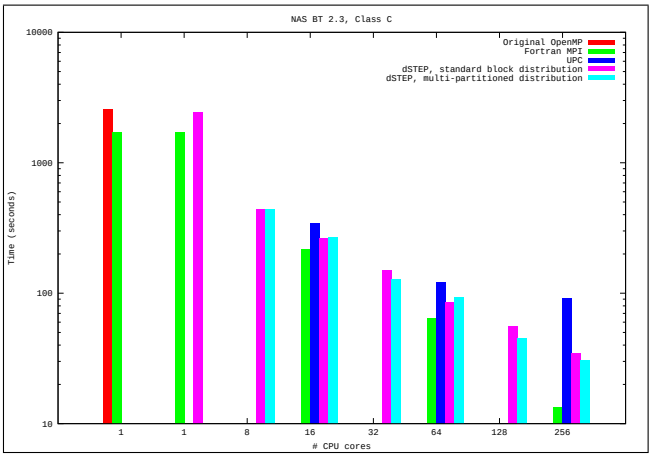

Figure 7. BT class C computation times, comparison between *dSTEP*, MPI Fortran and UPC.

loop nests where the three first dimensions are not always all parallel. Each time the system is solved along one direction, the corresponding dimension is ordered. In addition, some parts of the arrays are accessed separately by isolated iterations. We distribute the first three dimensions of the arrays and of the loop nests of the program. We use, like for the Polybench Adi kernel, a standard and a multi-partitioned distribution. For the loop nests, we use, as needed, parallel, ordered and owner schedules for the distributed dimensions.

Optimizations. Table 2 shows the performance gains on 64 cores using the loop interchange and the halo adjustment optimizations for the standard distribution. We obtain a gain of 4.5% with loop interchange and 16.8% when combining both optimizations. For all the remaining experiments, we apply the *loop interchange* and the *halo adjustment* optimizations.

No optimization	Loop interchange	Loop Interchange + *set_halo*
102.96 sec.	98.4 sec.	85.7 sec.

Table 2. Loop Interchange and *set_halo* gains on 64 cores

We compare the execution times of four versions of the BT application: the official MPI Fortran manual implementation, the UPC manual implementation using MPI and the generated dSTEP code with the standard and the multi-partitioned distribution. We notice that the baseline for the Fortran MPI version is better than the C version used as input to dSTEP. The execution times of the MPI Fortran are always lower than those of both the UPC and the generated dSTEP codes. The multi-partitioning implementations in the Fortran MPI and UPC versions requires a square number of processors to be executed. With dSTEP however, we can execute the generated code on any number of processors. The dSTEP code performs better than the UPC code, for which we don't have the results on 1 and 8 cores because of memory errors. The multi-partitioned version of dSTEP performs better than the standard one, except for 16 and 64 cores because of the grid shape. For instance, the 64 cores execution corresponds to the launch of 8 MPI processes with 8 OpenMP threads on 8 nodes of bi-quadricore processors. In this case, the grid for each array and loop nest has the shape $4 \times 2 \times 2$, which makes only half the processors active when sweeping the x dimension. This imbalance combined with the communication overhead of the multi-partitioned distribution makes it perform worse than the standard distribution on 64 cores. In Table 3, we compute the speedups for the MPI Fortran code

# cores	8	16		32	64		128	256	
MPI Fortran	-	7.89		-	26.95		-	128.8	
speedup, % / MPI Fortran	s	s	%	s	s	%	s	s	%
dSTEP standard	5.77	9.77	123%	17.14	29.88	110%	46.12	73.62	57%
dSTEP multi-partitioned	5.77	9.5	120%	19.99	27.28	101%	56.76	83.35	65%

Table 3. BT compared performance

and for both versions of generated dSTEP code. The speedups are computed relatively to the specific baseline of each version of the code. We than compare the ratio between the dSTEP speedups and the Fortran MPI speedups. We can see that on 16 and 64 cores, the dSTEP speedups outperform the MPI Fortran speedups. On 256 cores however, the speedup of the dSTEP multi-partitioned code is 65% of the Fortran MPI code.

Conclusion from the experiments. We obtained very good performance for matrix multiplication by combining the total halo and the *all* distribution. We obtained a fairly good performance for the Jacobi kernel. For the Polybench Adi, the performance did not scale very well due to a low computation/communication ratio. On the BT program, we showed good performance using both a standard block and multi-partitioned distribution. On 16 and 64 cores, we outperformed the MPI Fortran manual implementation but we noticed a degradation of the performance on 256 cores. On N nodes, the memory used on each node is $S/N + H$ for each array of size S and halo H, allowing to handle very large data sets when the halo is not very large. Finally, we showed that the performance of the dSTEP generated programs come with a low investment effort from the programmer, consisting on inserting *distribute* and *gridify* directives.

7. Conclusion and future work

In this paper we presented dSTEP, a high-level programming model for hybrid distributed and -shared memory systems for dense scientific programs. We proposed a generic compilation scheme and implemented our solution in the PIPS source-to-source compiler together with a runtime system. Our transformation system maps the computation to distributed and shared memory architecture and automatically generates the necessary communications. We unify the standard distribution types, the multi-partitioning and the use of the halo in a single model. We offer a series of optimizations aiming essentially at reducing the amount of communicated elements and to overlap computation and communication. Our solution handles the problem of data and computation distribution interprocedurally for a whole program. We validate our solution on several scientific kernels and on the more challenging NAS BT application for which we outperform the Fortran MPI manual implementation on 16 and 64 cores.

Our ongoing work is on the adaptation of the dSTEP programming model to clusters of GPUs. In our solution, the inter-node communications are already handled and can be readily reused for GPU clusters. Beside that, the host/device communication on each node is obtained by instrumenting the send and recv functions: at each communication call, we have the description of the array region to communicate; we insert a synchronous device-to-host transfer of the corresponding elements before each send communication and we insert a synchronous host-to-device transfer after each complete of a recv operation. The missing part is the mapping of the computation to the specific target language like CUDA.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Top500, "Top 500 supercomputer sites," http://www.top500.org/, 2014.

[2] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 3.0," Tech. Rep., September 2012.

[3] "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers," *The Journal of Supercomputing*, vol. 10, no. 2, 1996.

[4] D. Bonachea, "GASNet Specification, v1.1," Berkeley, CA, USA, Tech. Rep., 2002.

[5] Augonnet, Cdric and Thibault, Samuel and Namyst, Raymond and Wacrenier, Pierre-Andr, "StarPU: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2.

[6] High Performance Fortran Forum (HPFF), "High Performance Fortran Language Specification," 1997.

[7] Mellor-Crummey, John M. and Adve, Vikram S. and Broom, Bradley and Chavarra-Miranda, Daniel G. and Fowler, Robert J. and Jin, Guohua and Kennedy, Ken and Yi, Qing, "Advanced Optimization Strategies in the Rice dHPF Compiler." *Concurrency and Computation: Practice and Experience*, vol. 14, no. 8-9, pp. 741–767, 2002.

[8] K. Kennedy, C. Koelbel, and H. Zima, "The Rise and Fall of High Performance Fortran: An Historical Object Lesson," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III, 2007, pp. 7–1–7–22.

[9] Numrich, Robert W. and Reid, John, "Co-Array Fortran for Parallel Programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998.

[10] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin, "A New Vision for Co-Array Fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, ser. PGAS '09, 2009, pp. 5:1–5:9.

[11] UPC Consortium, "UPC Language Specifications, v1.2," Lawrence Berkeley National Lab, Tech Report LBNL-59208, 2005.

[12] Chamberlain, B.L. and Callahan, D. and Zima, H.P., "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[13] Charles, Philippe and Grothoff, Christian and Saraswat, Vijay and Donawa, Christopher and Kielstra, Allan and Ebcioglu, Kemal and von Praun, Christoph and Sarkar, Vivek, "X10: an Object-Oriented Approach to Non-Uniform Cluster Computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[14] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[15] J. Merlin, D. Miles, and V. Schuster, "Distributed OMP: Extensions to OpenMP for SMP Clusters," in *EWOMP 2000, Second European Workshop on OpenMP*, 2000, pp. 14–15.

[16] A. Basumallik and R. Eigenmann, "Towards Automatic Translation of OpenMP to MPI," in *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005, pp. 189–198.

[17] Millot, Daniel and Muller, Alain and Parrot, Christian and Silber-Chaussumier, Frédérique, "STEP: A Distributed OpenMP for Coarse-Grain Parallelism Tool," in *OpenMP in a New Era of Parallelism*, ser. Lecture Notes in Computer Science, R. Eigenmann and B. Supinski, Eds. Springer Berlin Heidelberg, 2008, vol. 5004, pp. 83–99.

[18] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier, "From OpenMP to MPI: First Experiments of the STEP Source-to-source Transformation Tool." in *PARCO*, 2009, pp. 669–676.

[19] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization," *ACM Sigplan Notices*, vol. 44, no. 4, pp. 101–110, 2009.

[20] F. Silber-Chaussumier, A. Muller, and R. Habel, "Generating Data Transfers for Distributed GPU Parallel Programs," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1649–1660, 2013.

[21] A. Duran, E. Ayguad, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A Proposal For Programming Heterogeneous Multi-Core Architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[22] in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds., 2011, vol. 6852.

[23] Nakao, Masahiro and Lee, Jinpil and Boku, Taisuke and Sato, Mitsuhisa, "Productivity and Performance of Global-View Programming with XcalableMP PGAS Language," in *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012, pp. 402–409.

[24] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-core Parallel Programming Environment," in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[25] "The OpenACC Programming Interface," *http://www.openacc-standard.org*.

[26] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC - First Experiences with Real-world Applications," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 859–870.

[27] Amza, C. and Cox, A.L. and Dwarkadas, S. and Keleher, P. and Honghui Lu and Rajamony, R. and Yu, W. and Zwaenepoel, W., "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[28] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM Compiler for Distributed-Memory Multicomputers," *Computer*, vol. 28, no. 10, pp. 37–47, 1995.

[29] J. Li and M. Chen, "Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays," in *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, 1990, pp. 424–433.

[30] T. Y. S. R., "Parametrically Tiled Distributed Memory Parallelization of Polyhedral Programs," Colorado State University Technical Report CS13-105, Tech. Rep., June 2013.

[31] D. Kim, "Parameterized and Multi-level Tiled Loop Generation," Ph.D. dissertation, Fort Collins, CO, USA, 2010, aAI3419053.

[32] P. Feautrier, "Dataflow analysis of array and scalar references."

[33] M. Amini, C. Ancourt, F. Coelho, F. Irigoin, P. Jouvelot, R. Keryell, P. Villalon, B. Creusillet, and S. Guelton, "PIPS is Not (Just) Polyhedral Software," in *International Workshop on Polyhedral Compilation Techniques (IMPACT11), Chamonix, France*, 2011.

[34] Irigoin, François and Jouvelot, Pierre and Triolet, Rémi, "Semantical Interprocedural Parallelization: an Overview of the PIPS Project," in *Proceedings of the 5th international conference on Supercomputing*, ser. ICS '91. New York, NY, USA: ACM, 1991, pp. 244–251.

[35] Creusillet, Béatrice and Irigoin, François, "Interprocedural Array Region Analyses," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer Berlin Heidelberg, 1996, vol. 1033, pp. 46–60.

[36] Corinne Ancourt and Fabien Coelho and François Irigoin and Ronan Keryell, "A Linear Algebra Framework for Static High Performance Fortran Code Distribution," *Scientific Programming*, pp. 3–27, 1997.

[37] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 26, no. 6, pp. 30–44, May 1991.

[38] F. Trahay, E. Brunet, A. Denis, and R. Namyst, "A Multithreaded Communication Engine for Multicore Architectures," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–7.

[39] L.-N. Pouchet, "PolyBoench/C, The Polyhedral Benchmark suite," http://www.cse.ohio-state.edu/ pouchet/software/polybench, 2014.

[40] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.