A Polyhedral Approach for Auto-Parallelization using a Distributed Virtual Machine

Damien de Montis, Jean-Baptiste Besnard, Christophe Alias

A Polyhedral Approach for Auto-Parallelization using a Distributed Virtual Machine

Damien de Montis, Jean-Baptiste Besnard, Christophe Alias

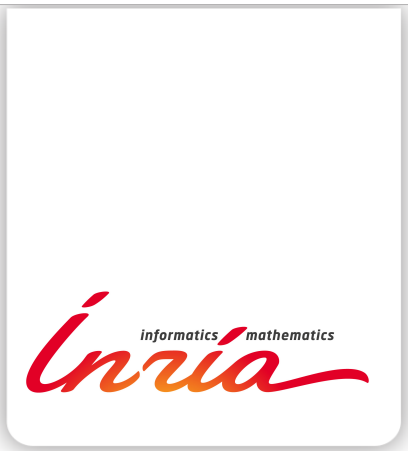
RESEARCH REPORT

N° 9432

October 2021

Project-Team Cash

ISSN 0249-6399 ISRN INRIA/RR--9432--FR+ENG



A Polyhedral Approach for Auto-Parallelization using a Distributed Virtual Machine

Damien de Montis*, Jean-Baptiste Besnard†, Christophe Alias‡

Project-Team Cash

Research Report n° 9432 — October 2021 — 25 pages

Abstract: As parallel systems have to undergo an unprecedented transition towards more parallelism and hybridization, we propose to discuss the consequences on programming models. In particular, MPI and OpenMP may face some complexity barriers due to the added complexity required by such hardware. We propose to build from the ground up a new way to program a parallel system, relying both on a distributed runtime, unifying multiple nodes in a coherent ensemble, and on advanced tools from the Polyhedral model. We first describe the Distributed Virtual Machine (DVM) runtime establishing a data-flow environment suitable to the polyhedral transformations. We present and identify what we have seen as key components in such a system, transposing loop nests to a distributed set of machines thanks to a pragma notation combined with an automatic tiling approach. Eventually, while presenting results we open the discussion of remaining challenges and some potential mitigation.

Key-words: Compilation, automatic parallelization, polyhedral model, runtime scheduling

* ParaTools SAS
† ParaTools SAS
‡ Inria/ENS-Lyon/UCBL/CNRS

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Une Approche Polyédrique de la Parallélisation Automatique Via un Machine Virtuelle Distribuée (DVM)

Résumé : Les systèmes parallèles devant subir une transition sans précédent vers plus de parallélisme et d'hybridation, nous proposons d'en discuter les conséquences sur les modèles de programmation. En particulier, MPI et OpenMP peuvent être confrontés à un manque d'expressivité en raison de la complexité supplémentaire requise par un tel matériel. Nous proposons de construire à partir de zéro une nouvelle façon de programmer un système parallèle, en s'appuyant à la fois sur un runtime distribué, unifiant plusieurs nœuds dans un ensemble cohérent, et sur des outils avancés du modèle polyédrique. Nous décrivons d'abord le runtime du Distributed Virtual Machine (DVM) établissant un environnement de flux de données adapté aux transformations polyédriques. Nous présentons et identifions ce que nous avons vu comme des composants clés d'un tel système, en transposant des nids de boucles à un ensemble distribué de machines grâce à une notation à base de pragmas combinée à une approche de tuilage automatique. Finalement, tout en présentant les résultats, nous ouvrons la discussion sur les défis restants et certaines mesures d'atténuation potentielles.

Mots-clés : Compilation, parallélisation automatique, modèle polyédrique, ordonnancement dynamique

1 Introduction

Programming a supercomputer has never been a simple task, being a specialized undertaking requiring multi-disciplinary knowledge at the frontier between computer science and the target domain of application. In this report, acknowledging the increasing pressure on parallel applications to put up with increased node-level parallelism, we ask the simple question of what if we could start everything from scratch given current hardware. Indeed, HPC software has a non-negligible legacy aspect, structurally imposing some technologies on the long-term. For example, MPI and somehow OpenMP are both long-running technologies shaping machines' programming for decades. Given the efforts put in specialized HPC applications, MPI and OpenMP are then part of the requirement for new machines as current codes should run at all costs. In this report, we wonder what could be achieved in terms of auto-parallelization using *polyhedral technologies* that may provide some components to build an alternative programming model.

Polyhedral model is a dynamic field providing a wide range of tools and models to capture how increasingly complex sets of loops are behaving. This model offers a rich set of operations atop loop nests enabling both capture program's behavior while manipulating its resulting expression using complex transformations – not even possible manually. Such technology is productively used for synthesis on Field-Programmable Gate Arrays (FPGAs). In the rest of this report, we propose to build upon this previous experience to define how a whole supercomputer could approach the layout of an FPGA, and therefore, as we will elaborate, how Polyhedral models unfold programs targeting the corresponding runtime. Thanks to a source-to-source transformation mechanism leveraging the Poco analyzer [1], we map loop nests on whole machines. Such an approach should eventually rely on pragmas extracting some loops to run in a transversal manner between distributed processes, requiring a combination of static analysis and runtime mitigation as presented in the rest of this document.

2 Overview

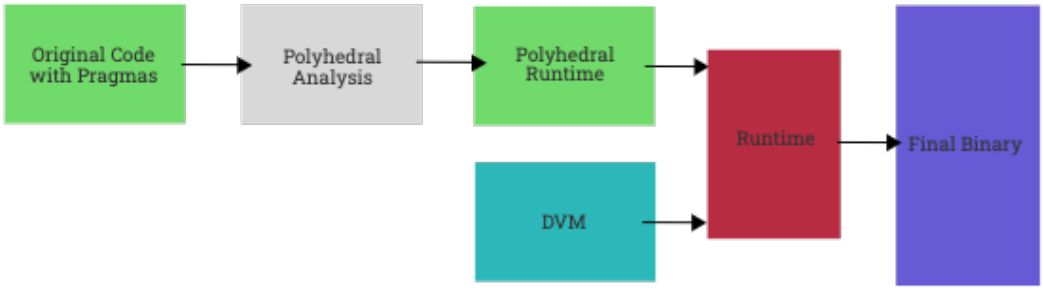


Figure 1: Overview of the Distributed Virtual Machine pipeline

In this report, we introduce the Distributed Virtual Machine (DVM) which is a data-flow environment for HPC systems. After, detailing this alternative programming interface aimed at gathering a whole HPC-class machine in a single "virtual" re-configurable system, we focus on programming such new abstraction. Building up from approaches tailored for Field Programmable Gate Arrays (FPGAs) which are examples of such re-configurable systems, we unfold possibilities proposed by our new runtime. In a second time, we explore polyhedral synthesis for the DVM to tile a set of representative loop kernels. To do so, as presented in Figure 2 we

present a semi-automated parallelization pass as our initial implementation of this programming scheme. Eventually, after presenting results linked to the expression of such kernels on the DVM, we conclude with some potential enhancements for this model, outlining remaining challenges and future-work addressing them.

3 Related Work

In this section, we propose to first discuss how supercomputers are mostly programmed as of today. In a second time, we will cover some less spread alternative models. Then, we will consider current techniques for automatic parallelization and how they relate to our approach. Eventually in the following section, we will summarize our contribution in the light of this existing work.

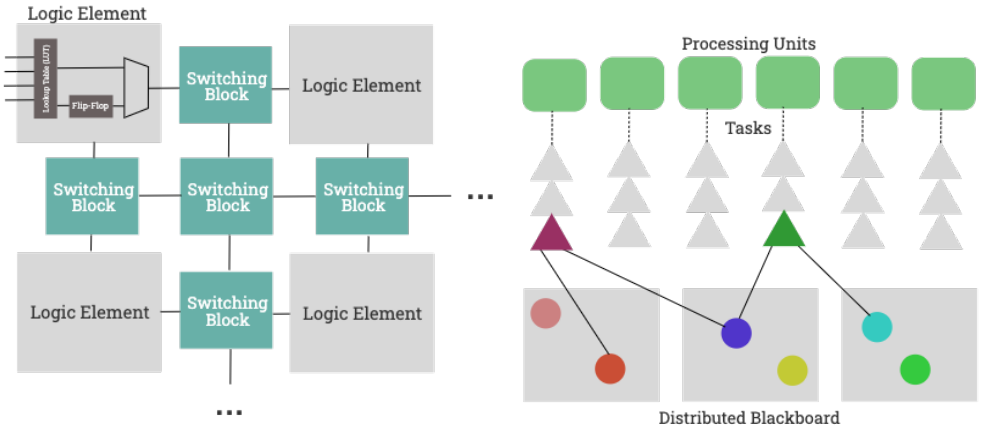
3.1 Established HPC Programming Models

If you connect to any supercomputer in the world you are likely to find at least an Implementation of the Message Passing Interface [9] (MPI) and one of OpenMP [5]. Indeed, most of the HPC code corpus is relying on a combination of this programming interfaces [3]. Such interfaces while not only providing the exact features needed by the codes of their time also kept evolving to accommodate for new hardware, this while maintaining an aggressive backward compatibility scheme. Therefore, it is no surprise if important computing centers are more likely to rely on these proven technologies which, for computer-science artifacts, are defying time. Proven reliability and how these interfaces provide transitive evolution for applications – not rewriting from scratch – are probably important factor in this success. Yet the pressure is increasing on these interface due to important hardware changes [11]. Indeed, what was previously pure MPI became in the 2000s $MPI + X$ due to increased shared-memory parallelism [13] and is yet to become $MPI + X + Y$ as GPUs are key components in upcoming Exascale systems often featuring converged architectures. Current approach to target GPUs seem to feature OpenMP targets [12] which despite elegantly integrated in the OpenMP syntax may lead to increasing progammative complexity. Indeed, Amdahl's law is never far and not using all transistors at once is bonding the speedup [4], therefore impressionist touches of nested parallel regions generally impact performance. Indeed, stacking models on each other, MPI plus OpenMP and punctually CUDA [8], for example, leads to increased program complexity and cannot be done in a complete manner by definition of the transitive model stacking at stake. This is what motivates some alternative approaches for paralellism that we present in the following sections.

3.2 Alternative Programming Models

HPC is not limited to only MPI and OpenMP, naturally there as several alternatives. For example, and close in some aspects to what we develop in this report, there are Partitioned Global Address Spaces (PGASs) [6,16]. Such programming model provide abstractions to expose a distributed memory as a global shared one. This enables programming multiple machines globally, however, the control-flow (e.g. the program) is still split manually in an MPMD fashion. As we will further develop, our approach is similar from a data standpoint, enabling global access, however, it is very different in terms of processing as we also enable them to be moved just as data, practically removing the MPMD constraint while enabling reconfiguration.

There are also programming languages dedicated to HPC, either proposing PGAS extensions such as Co-Array [10] Fortran or UPC [2]. Some other languages such as Chapel, Fortress or X10 [15] provide new ways of expressing parallelism thanks to extended semantics. Still



(a) FPGA architecture (b) DVM architecture

Figure 2: Architectural comparison between the DVM and an FPGA

such approach requires a full rewrite of any existing program to take advantage of these new capabilities.

4 Distributed Virtual Machine (DVM)

One of the key component in our approach is to abstract the set to nodes to allow it to behave *programatically* as a single machine. Currently, programming a supercomputer is practically giving instructions to N machines often identified by their MPI rank. All operations are generally located using this rank which in turn determines the set of instructions to be issued. Among such instructions there are messages which are here to account for data-exchanges between distributed memory areas. In this section, we consider not a mapping of logical processing in interaction but on the contrary we plan to map processing on an abstract view of a given system. Main difference being that initially, neither data nor processing instructions are present inside remote processes, unlike standard MPMD computations. It means the runtime is able to scatter computations just as data, meaning it is possible to send a full computation (operands + data) to any element of the system on the fly.

This reconfiguration capability is a key aspect of our runtime, as acting as a Distributed Virtual Machine (DVM) enables computational mitigation which are not practical when considering a set of collaborating machines. Indeed, in our approach we not only move data but we also move the computation. Practically, it means that computation is no longer tied to a fixed-set of resources, tasks of arbitrary complexity can move between nodes to account for dynamic load changes or computing constraints. As such layout leads to re-configurable system, similar in some aspects with what is seems with Field Programmable Gate Arrays (FPGAs) we will first compare the two resulting architectures. In a second time, we will present the DVM interface and how it is programmed. Eventually, we will outline how it is possible to issue simple program on this DVM.

4.1 On Similarities with FPGAs

FPGAs are example of re-configurable architectures where the polyhedral analysis plays an important role as doing the routing on such hardware involving thousands of component is far from being a trivial operation – requiring powerful abstractions. As presented in Figure 2a, an FPGA is macroscopically made of Logical Elements (LE) interconnected with a switching network, it means it is constituted at lower scale of re-configurable elements connected to each other depending on the program which determines the circuit layout interconnecting all the components. Data then flows between each elements to be computed, and practically *flows* between these logical elements, being often *streamed* as the processing goes. Here is an important aspect of FPGAs, their systolic nature, meaning that computation of arbitrary complexity can be performed by setting a given input and waiting for the result on the output as it propagates through the logical circuit. Comparatively, when a regular computing unit on a computer would have to scan each cell of an array for example to produce a result, applying a given operation to fill a new cell, the FPGA can have multiple cell as its input (if not the whole array) and directly yield the result as output. It is then a form of vectorization where a given processing is applied at once to multiple data, and where FPGAs stand out is that it is the hardware which shapes itself to match the operation to be applied to the data-set and not the opposite with data being laid out on computing units as done is most HPC MPMD programs.

Conversely, we now consider the Distributed Virtual Machine architecture in Figure 2b. First, note the similarity with the FPGA layout. In place of Logical Elements (LE) we have each CPU core which is much more capable than any FPGA LE. And as the interconnection network we simply have common HPC interconnect, shared-memory when suitable and messaging in distributed memory. Such messages are one-sided in nature, being remote procedure calls (RPCs). As far as the layout is concerned, a program is made of instructions to be run on each unit, and data to be streamed between these units. Just as with FPGAs where each LE was programmed to do instructions and fed with a bus of data to process. Main difference in this case is the abstraction level, Indeed for an FPGA you work at bit level with very simple operations (logical and mathematical) whereas, in the DVM, data can be of any size (blob of data) and operations of arbitrary complexity as a general purpose CPU is much more capable than a single LE. As far as the interconnection network is concerned, data are input to each processing to generate new data themselves contributing to other processing, this also leads to a data-flow between CPU cores as what was issued a lower level between Logical Elements in the FPGA.

Overall, the DVM is the transposition of an FPGA architecture to a whole distributed system. Expressing progressing as punctual tasks of arbitrary complexity and data movements as input-output links between these processing, we yield an approaching behavior at higher scale (more complex operations, arbitrary data-blobs). This symmetry in terms of architecture, as we will unfold in the rest of this report, also advocate for using similar tools to generate interaction, reason why we considered polyhedral analysis in our approach. Moreover, doing so means abstracting machine layout thanks to automated synthesis, point which starts to be limiting on new HPC architecture which tend to hybridize. In the following subsection we will focus on the DVM Implementation itself, outlining the interface which allows to express aforementioned data-flow.

If the DVM what to be compared to an existing programming abstraction it would be called a distributed blackboard architecture [7]. In such model of expert-system, data are transversely available to an arbitrary set of *Knowledge-Systems (KS)* freely processing these data while generating new elements on the blackboard, all these elements being organized by the *Control Shell (CS)* in charge of coordinating all participating components. Initially, such blackboards were introduced to mimic collaborative behavior in problem resolution. In fact, it is in several as-

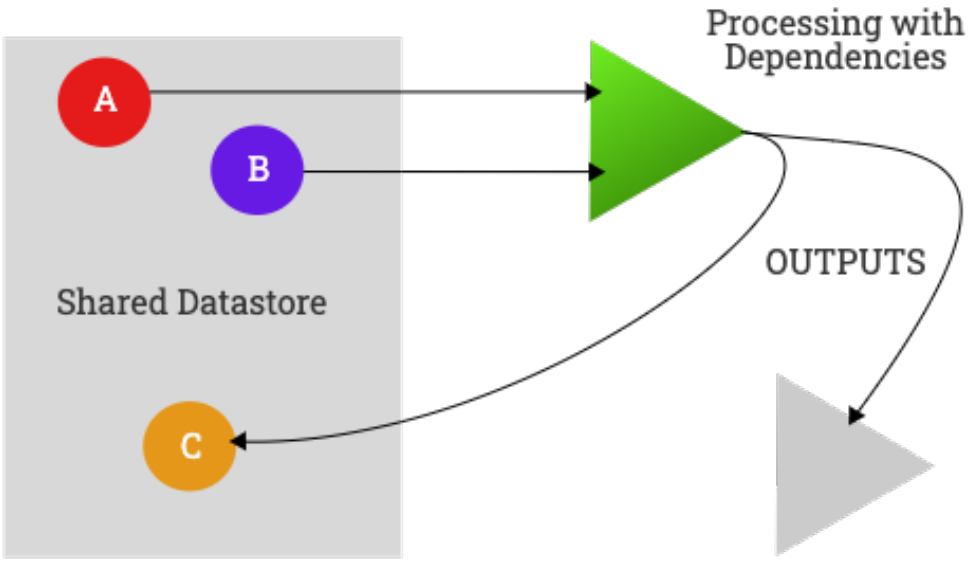


Figure 3: Illustration of DVM components.

pects a multi-agent system where the blackboard is the "world" connecting agents for problem resolution. Overall, the key change in such system is how data is now transverse, it means a lesser sensitivity to initial data-layout and the possibility of *spreading* processing arbitrarily to (1) reshape initial data before (2) applying arbitrary processing anywhere inside the distributed "world". It means that data-movement is now solely a data dependency for a given processing, unlike for example what happens for MPI programs where it is also inherently part of the program as matching collectives and sends and receives. In other words, such approach removes the data-movement burden from the computational expression, delegating it to an external agent, the *blackboard* (or the "world"). Naturaly, this shift of things, moves the complexity to expressing the computation in a data-flow manner, while it is initially done as a set of loops iterating on a given set of arrays – it is where polyhedral analysis will come at play.

4.2 DVM Implementation

As presented in Figure 3, the DVM is built as a simple programming interface. It manages two main classes of elements, *data* and *processing* which express how such data are to be processed. A processing can generate both new processing and new data. Therefore, a program on the DVM is initiated with the creation of initial data-elements and corresponding processing and then transitions to a data-flow oriented phase before completion. practically, due to the data-flow nature of such computation, the parallel DVM phase starts with a data-splitting phase analog to a scatter, goes on with parallel processing of individual data-stripes and concludes with the reconstitution of the initial array from striped data. The main reason for this approach is programmability, as we eventually want to include parallel DVM phases in regular programs just as, for example, OpenMP parallel regions. It means the data layout has to be converted and restored to enable transparent compatibility.

4.2.1 Resource layout

The DVM sees the machine as a collection of resources, each resource is attributed coordinates in a n-dimensional space such as the vector distance matches the actual device distance. Practically, it means that we use HWLOC to discover computing resources and map each Processing Unit (PU) to corresponding coordinates doing a breadth-first search in the global topology. This process takes place during the `dvm_init` call which also starts a pending execution thread bound to each PU. As-far as inter-process discovery is concerned we rely on the PMI just as the collocated MPI implementation may do to generate an unique rank accounting for each UNIX process. This done, the DVM features a non-linear space accounting for all PUs on the system. Process layout does not consider shared-memory separation, meaning that a rank will *always* point to an unique PU and not for example as in the case of MPI to a process hosting multiple PU. Yet, a single UNIX process could host multiple ranks in shared-memory to speedup launch time and allow shared-memory programming facilities for node-local operations. Therefore, a rank is always pointing towards a resource, and such number is simply topologically describing the machine layout.

4.2.2 Data management

Data management is a key part of the DVM, each data-segment is attributed with and unique identifier composed of two parts, the first part is the node number and the second par an incremental ID. Thanks to this numbering each segment can be retrieved from anywhere, the identifier carrying the memory region holding the data. Moreover, each data-entry is initialized with a reference counter which is decremented each time the data is provided as input for a processing. Retrieving a remote data is practically a remote procedure call, moreover there are caching mechanism to account for data possibly retrieved multiple times using a trivial Last Recently Used (LRU) cache mechanism flushed when more memory is needed. Data are created using two means, first mean is simply allocating new data with a given ref-counter, the DVM then frees them when no reference are left – the classical garbage-collector model. The second approach is devised to provide interface with existing codes, a given memory region can be registered and then provided with a new data-id and an optional callback is called when freed. Using such IDs and registration, the DVM then manages to convert local data-segments to a global address-space approaching what can be achieved with PGAS programming models. It means, each parallel DVM region will require a proper data-registration phase in order to convert from the in-memory model to the data-flow model considered by our approach, this therefore requires some a-priori knowledge on data-layout in order to generate pack and unpack methods to carry the computation. One may argue that this leads to data-duplication, indeed, from a first approach it means replicating the array, but ideally the initial array should be freed as the initial data are split over the DVM, in fact the compact representation is solely needed for regular loops and not for the individual DVM processing which will take advantage of smaller blocks. It means that we switch from a single static array to a symbolically linked set of buffers as tiled by the computational requirements, and therefore this will have a cost that should naturally be reduced.

4.2.3 Re-configurable processing

Processing is defined as the operation to be applied to some data. The instance of a processing plus its inbound data is called a task as in other programming models. Prior to the computation processing have to be registered in the DVM. This registration, as data, yields IDs which are tied to a given node, it means that is is possible knowing a given processing ID to retrieve the

remote machine code to proceed to the execution locally. This is done copying the function, considering architectures are compatibles, eventually, we are willing to expand this processing relocation capability to hybrid systems (GPUs) at it would allow dynamic scheduling outside of the boundaries of a given kind of computing units, enabling for example, spilling scheduling algorithms on the machine where computation spreads with data, taking advantage of a complete reprogammability of the DVM. Note that a processing can register other processing and data to feed new operations. Moreover, in most cases, processing should be a relatively static set of operations, data being the source for most of the parallelism. Still, such approach allows to combine all types of parallelism as various processing can be collocated on the same node by nature of the blackboard approach.

4.2.4 Programming the DVM

```
/***********
 * RANKING *
 ***********/

/**
 * @brief These functions provide rank information
 *
 */

#define DVM_MAX_RANK_DEPTH 8

/**
 * @brief This defines the rank array statically
 *
 */
typedef uint64_t dvm_rank_t[DVM_MAX_RANK_DEPTH];

/**
 * @brief Retrieve the dimension of each rank component
 *
 * @param dimension a number storing the dimension of each rank component
 * @return int DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_rank_dimension(int *dimension);

/**
 * @brief Get the rank of current executor
 *
 * @param rank_array output array
 * @param rank_array_size the size of the output rank array
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_rank_get(dvm_rank_t rank_array, int rank_array_size);

/**
 * @brief Get the size of each ranking component (note this size can change)
 *
 * @param size_array the output size array
 * @param size_array_size the output array size
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_rank_size(dvm_rank_t size_array, int size_array_size);
```

The Distributed Virtual Machine is practically a simple runtime. It is built around execution lists pinned to all CPUs in a parallel system. The runtime starts by scanning the whole machine to label each processing unit with an unique identifier. When combined with the node identifier this creates an unique global identifier for each processing list. The execution for each list is provided with a POSIX thread consuming jobs in order. When the DVM is fully started, the whole system (each PU) is seen as a list of unique identifiers. Moreover, a distance matrix is generated, also thanks to HWLOC to enable fine-grain scheduling (see later discussions). One aspect which is of interest is how we adopted a hierarchical rank, instead of a linear ID, we have a tree enabling distance computation.

```
/******************
 * DATA MANAGEMENT *
 ******************/

/**
```

```
 * @brief These functions implement the garbage collector
 *
 */

typedef uint64_t dvm_data_id_t[2];

/**
 * @brief Register an existing memory region in the DVM
 *
 * @warning The buffer is allocated with a refcounting of 1
 *          data are * not copied * buffer should remain until
 *          free_cb is called (meaning no reference left)
 *
 * @param addr the source addr to be registered
 * @param size the size to be registered
 * @param free_cb this function (if not NULL) is called when the buffer is ↩
     removed from the DVM
 * @param id the output UNIQUE identifier of this data
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_data_register(void *addr, size_t size, void (*free_cb)(void *↩
    addr, size_t size), dvm_data_id_t *id);

/**
 * @brief Allocate a new region be managed by the DVM
 *
 * @param size size of the new region
 * @param id corresponding ID
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_data_allocate(size_t size, dvm_data_id_t *id);

/**
 * @brief Increment atomically the refcounting of a given buffer
 *
 * @param id identifier of the buffer to acquire
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_data_acquire(dvm_data_id_t id);

/**
 * @brief Decrement the refcounting of a given buffer
 *
 * @note if the refcounting reaches 0 the buffer is freed (if allocated with ↩
     @ref dvm_data_allocate)
 *       or the free_cb from @ref dvm_data_register is called to notify release
 *
 * @param id identifier of the buffer to be relaxed
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_data_relax(dvm_data_id_t id);
```

For the data-management part, a simple allocator with support for garbage-collection was implemented. It can either register existing data-segments or allocate new memory region. Each data-segment is given a 64 bits identifier, first 32 bits of the identifier are the node identifier, and the last 32bits are a sequentially incremented number. A consequence of this layout is that each data-segment is uniquely identified machine wide (i.e. including between nodes). Therefore, if data-dependencies are encountered later on during the data-flow execution phase, it is possible to retrieve data from where they originate. Note that this is made possible thanks to polyhedral analysis which allows to compute how many tasks will need a given input data – enabling the

correct initialization of the ref-counter.

```
/**************
 * PROCESSING *
 **************/

typedef uint64_t dvm_processing_id_t[2];

/**
 * @brief These functions implement processing function registration
 *
 */

/**
 * @brief Register a new processing function in the DVM runtime
 *
 * @param callback pointer to the function to be called
 * @param number_of_args number of arguments to be passed to the function (of ↩
     type dvm_data_id_t)
 * @param description Short description of this function
 * @param id output identifier of the processing function
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_processing_register(int (*callback)(),
                                      int number_of_args,
                                      char *description,
                                      dvm_processing_id_t id);

/**
 * @brief Register a precompiled function as part of the current DSO
 *
 * @warning Local symbol resoltion with dlsym requires "-rdynamic"
 *
 * @param function_name name of the function to be looked up locally
 * @param number_of_data_args number of arguments to be passed
 * @param id output processing function id
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_processing_register_by_name(char *function_name,
                                              int number_of_data_args,
                                              dvm_processing_id_t id);

/**
 * @brief Remove a processing function from the DVM
 *
 * @param id the identifier to be unregistered
 * @return dvm_retcode_t
 */
dvm_retcode_t dvm_processing_unregister(dvm_processing_id_t *id);
```

One point that we wanted to explore in the DVM was reconfiguration. Indeed, we consider a system which is not pre-configured and consequently, we want to be able to spread computing as distributed system often spread data. In the interface, this materialize as computing identifiers which are also using a per-node plus local identifier numbering to allow their remote retrieval. For now computings are defined with a function pointer, and therefore enabling their mobility would require some binary introspection to identify the boundaries of the matching instructions. Note that this will naturally constrain target architecture to matching bytecode. On this area, having some intermediate representation could help moving computing to a wider range of devices. This leaves us with a fully reconfigurable system, scheduling a processing is pushing *both* a processing and its input data to produce new data and possibly new processing.

```
/**************
 * SCHEDULING *
 **************/

/**
 * @brief Run a given processing on a given rank
 *
 * @param rank the target rank to execute onto
 * @param function the function to be called
 * @param ... a variadic list of dvm_data_id_t which shall match
              the number of args in the function
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_sched_onto(dvm_rank_t rank,
                             dvm_processing_id_t function,
                             ...);

/**
 * @brief Get the number of pending tasks on a given rank
 *
 * @param rank target rank to query
 * @param load the number of pending tasks
 * @return dvm_retcode_t DVM_SUCCESS if all OK
 */
dvm_retcode_t dvm_sched_load(dvm_rank_t rank, uint64_t *load);
```

Eventually, executing work is simply matching a processing with its data and targetting it at a given processing unit, identified with its rank. This processing is allowed itself to register new data and new processing, enabling self-deploying computing. Thanks to global identifiers for both data and processing, if a given task (data + processing) is to be moved including in-between nodes it will always be able to find its input elements by querying the source node. This aims at symplifying load-balancing and gives more freedom to the underlying runtime. In addition, it enables one PU to generate work for the full system, not only spreading the data but also the code to execute.

A first version of the aforementionned interface has been implemented during the training. We relied on C++ with simple vectors as list. We have identified during this work that it was crucial to generate tasks inside a task, as otherwise, there was a non-negligible sequential time spent filling the lists before even executing the first useful instruction. In addition, the runtime was very simple and will benefit from further optimizations, using C and avoiding allocations as much as possible (which is sometimes not trivial in C++). From this first iteration, we are willing to leverage more advanced runtimes to achieve improved performance.

5 Generating Code for the DVM

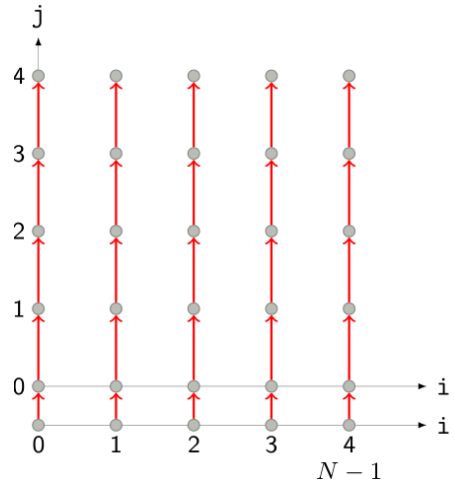
We have presented how the runtime could be used to express data-flow oriented computation in the context of the DVM. In this section, we will now detail how we can do source-to-source transformation to convert an existing loop to the DVM runtime model. After introducing the role of polyhedral analysis, we will present how we leveraged Poco [1] to perform ad-hoc tiling and scheduling for the DVM.

5.1 Polyhedral Model

The polyhedral model is a particular way to see loop nests. It aim to express all constraints with linear functions. We can easily see it with the illustration below. The right graph is the

representation of the left loop nest. The axes represent the count of i and j. Let's call S and T the two statements in the loop nest. Every point of the graph is a single operation. Their position is provided by the value of i and j. In this example w"e can see two abscesses. The lower one stand for S. Indeed, this statement represent the initialization of the resulting vector c. Last but not least elements are the red arrows. They represent the dependencies between iterations. The polyhedral model is very large and can handle many optimizations problems as index change for example. But in our problem, we just want to tile and schedule the loop nest.

```
for (i = 0; i < n; i++) {
S:c[i] = 0;
  for (j = 0; j < n; j++) {
T:   c[i] = c[i] + a[i][j] * b[j];
  }
}
```



Dependencies There are 3 types of dependencies. *Flow* dependencies (Read-After-Write) happen when an instruction depends on the result of a previous one. *Anti* dependencies (Write-after-Read) happen when an instruction requires a value that is updated later. *Out* dependencies (Write-After-Write) happen when the ordering of instructions will affect the final output value of a variable.

Given an operation (ie., an instance of some statement in the program) ω, we write read(ω) (resp. write(ω)) the set of addresses read (resp. written) by ω. There exists a *dependence* from an operation s to an operation t iff $s \prec_{seq} t$, both operations access the same address and one access is a write. When write(s) $\cap$ read(t) $\neq \emptyset$ (resp. read(s) $\cap$ write(t) $\neq \emptyset$, write(s) $\cap$ write(t) $\neq \emptyset$), we have a *flow* dependence and we write: $s \longrightarrow^{\text{FLOW}} t$ (resp. *anti*: $s \longrightarrow^{\text{ANTI}} t$, *output*: $s \longrightarrow^{\text{OUTPUT}} t$).

Dependencies are usually represented by a *reduced dependence graph* $Gc = (Sc, \Delta)$, whose nodes are the program statements; and edges $S \xrightarrow{\Delta_{ST}}{}^{\ell} T$ are labeled by $\Delta_{ST} = \{(vi, vj) \mid \langle S, vi\rangle \rightarrow^{\ell} \langle T, vj\rangle\}$, where $\ell \in \{flow, anti, out\}$.

Scheduling Scheduling aim to give an execution date to each iteration of a loop nest. In the polyhedral model this is described as follow.

A *schedule* θ_S assigns each operation $\langle S, \vec{i}\rangle$ with a timestamp $\theta_S(\vec{i}) \in (\mathbb{Z}^d, \ll)$. Intuitively, $\theta_S(\vec{i})$ is the iteration of $\langle S, \vec{i}\rangle$ in the transformed program. A schedule is *correct* if $\langle S, \vec{i}\rangle \rightarrow \langle T, \vec{j}\rangle \Rightarrow \theta_S(\vec{i}) \ll \theta_T(\vec{j})$, the lexicographic order ensuring that the dependence is preserved. The lexicographic order is the order defined by a finished list of elements in a given set. Here this is generalized by the dictionary order.

Tiling This is a re-indexing transformation which groups iteration into tiles to be executed atomically. There are many variants of this transformation. *Rectangular tiling* re-indexes any iteration $\vec{i} \in \mathcal{D}_S$ to an iteration $(\vec{i}_{block}, \vec{i}_{local})$ such that $\vec{i} = \mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local})$, with $\mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) =$

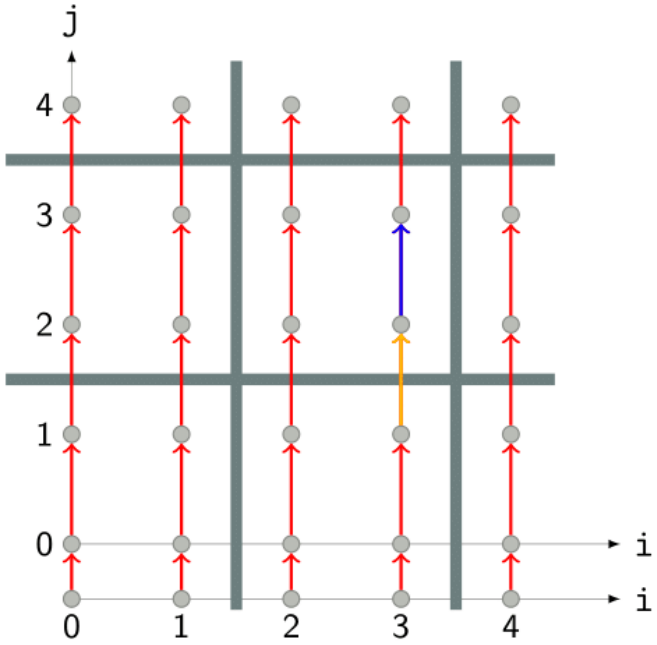


Figure 4: Graphical representation of a tiled loop nest

(diag $\vec{s}$) $\vec{i}_{block} + \vec{i}_{local}$, $0 \leq \vec{i}_{local} < \vec{s}$ where $\vec{s}$ is a vector collecting the tile size across each dimension of the iteration domain. $\vec{i}_{block}$ is called the *outer* tile iterator and $\vec{i}_{local}$ is called the *inner* tile iterator. The companion schedule associated to the tiling $\theta_S(\vec{i}_{block}, \vec{i}_{local})$ orders $\vec{i}_{block}$ first to to ensure the execution tile by tile. Figure 4 give an example of rectangular tiling with $\vec{s} = (2, 2)$. To enforce the atomicity (avoid cross dependencies between two tiles), it is sometimes desirable to precede the tiling by an injective affine mapping ϕ_S. The coordinates of $\phi_S(\vec{i})$, for $\vec{i} \in \mathcal{D}_S$ are usually called *tiling hyperplanes*. In that case, the transformation $\mathcal{T}_S^{-1} \circ \phi_S$ for some statements S is called an *affine tiling*. Note that rectangular tiling is a particular case of affine tiling where ϕ_S is the identity mapping.

Tiling is alike changing granularity. That mean we are now talking about groups of operations represented by tiles. Since we reason on tiles, we need to change each dependence between *operations* $\langle U, \vec{T}, \vec{i} \rangle \rightarrow \langle V, \vec{T}', \vec{i}' \rangle$ into dependence between *tiles* $\langle U, \vec{T} \rangle \rightarrow \langle V, \vec{T}' \rangle$. This is done by an orthogonal projection $(\vec{T}, \vec{i}, \vec{T}', \vec{i}') \mapsto (\vec{T}, \vec{T}')$, using the PoCo library [1].

5.2 Polyhedral Analysis

The polyhedral analysis abstraction is at the core of most automatic parallelization framework. Thanks to hyperplanes describing loop behaviors, regular loop nests can be expressed in a graphical fashion. Such representation, not only providing an in-depth vision of loop behavior can be used to *transform* loops as we will further detail in the rest of this section.

Starting from a regular loop expressed sequentially, our tool first tries to extract data-parallelism, unfolding dependencies and computing tiles on the data-path. Such tiles will be registered later on in the DVM – associating data to computing. Moreover, it is not only tiles which are computed but also data-dependencies in terms of overall size between tiles. This way, we are not only able to describe the computation in a set of multiples tasks automatically, but we

also have an heuristic to minimize data-movement when embedding the scheduling graph inside the actual machine topology.

The rest of this section will present how we leveraged polyhedral techniques to generate the aforementioned information to achieve proper scheduling on the DVM.

5.3 Loop Tiling

Tiling is the process of splitting a given loop in multiple data-parallel instances. To do so, we parametrize the loop with steps in its indices and run a polyhedral analysis to compute (1) loop boundaries for each loop instance and (2) a full parametrization of the input data-set for each instance. During our experiments, we observed that it was generally too expensive to try to build for a generic tile-size and therefore had to rely on static tile size during source-to-source synthesis to ease the work of the polyhedral analysis tool. This is a limitation which means that changing the tile size requires a new synthesis of the glue between the serial code and the DVM runtime calls. We plan to address this limitation later on by integrating tiling in an automatic manner in the source-to-source chain.

5.4 Pack and Unpack

The first step when running the DVM is identifying data block to be passed to the various parallel instances of the loop as it is being parallelized. This process generally starts from a static array (which is potentially distributed or simply not loaded in main memory) and then yields a tiling rule for this large data-set. It is important to stress that such data-set is too large for the memory available on a single node. Therefore, the first step we generated using polyhedral transformations is simply data preconditioning expressing how an array can be loaded in a distributed set of tiles to prepare for later data-resolution. Preparing data for the computing phase is close to simulating a scheduling of the so-called computing phase to determine a resource breakdown, in turn constraining a given data-layout.

Then, either the existing data-block (for example a matrix) is split in sub-elements moved to the right nodes or simply data are loaded from the storage subsystem to prepare all the tiles which will be later computed. Note that in the first model, the one generally present when the parallel DVM loop is inserted in an existing code, there may be data-duplication, at least the time for the array to be fully split in sub-blocks this can be a limitation factor. We developed a mitigation for this case, allowing data-buffers to be registered with a simple pointer but still, the case of non-contiguous elements (for example a matrix column in C) is non trivial to describe in place. We are considering higher level data descriptors (data-types with strides for example) but for now only matrix rows were able to take advantage of this in-place description, columns being generally packed with polyhedral codes. Note in addition, that this is a simplification of the issue, indeed, polyhedral analysis can outline much more complex dependencies schemes, including overlapping regions, for such cases, duplication is always a safe approach, moreover, one advantage of such model is at least improved cache efficiency, packing data and their operands.

5.5 Estimating Data-Exchanges

One last important aspect when scheduling a given loop in a distributed environment is data-locality. Indeed, unlike when running in shared-memory the distributed memory case may lead to data exchanges. For example, if we consider a pre-spliting of data due to single-node memory limitations, it is crucial to schedule the corresponding computations where they have the most dependencies, as close as possible from their inputs. To do so, we also rely on polyhedral analysis,

we have a pass analyzing for each input which are the dependencies, listing them all. Using a simple scoring mechanism, we tie processing to the PU holding the more input data. If this input is scattered among multiple nodes / PUs, the weighted-barycenter is used. Note that this process is also applied inside the nodes as there might be some Non-Uniform Memory Accesses at play, as far as data-block allocation is concerned it is for now a simple round-robin on the PUs of a given process (or node when running a single process per node).

The computation of these dependencies is done by first listing all the dependencies and then in a second time, summing up the size of their individual contributions while comparing with the initial data-layout. One point which is important to consider it is that we also use a dependency analysis to map data over the various nodes as discussed in previous section, it means that by nature processing units should have a correct mapping. Similarly, if a processing is to be moved towards another node, the same analysis is done with additional memory constraints, trying to transfer data to a close node.

5.6 Polyhedral Code Transformation Outline

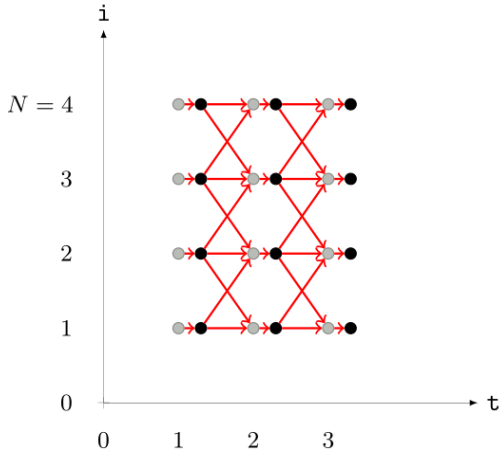
This section summarizes how the polyhedral transformation is integrated in the development workflow. In our approach we do source to source transformation. It means, we have to prepossess the code to replace it with modified instructions. Currently, we focused our approach on manually extracted loop kernels but we are planning to include a pragma-based notation to automate this loop outlining in a way similar to OpenMP. It means that we could then replace the loop instance by a function call, giving us freedom to compile a replacement translation unit. Our work in this report then starts from the source code of the outlined loop, we rely on a polyhedral compiler to extract loop hyperplanes and to compute the various in-variants required by the DVM (1) tiling to do work-splitting, (2) pack and unpack functions to enable transfers between nodes and (3) data-volume between tiles to improve overall scheduling.

This process is currently semi-automated in the sense that we have to feed-in the loop nest, however, tools such as PoCo could be leveraged in a compiler pass to automate this transformation. This analysis yields multiple functions in charge of describing tiling, and resulting operations. such functions are generated by the polyhedral compiler and may be relatively complex, they generally consist in a set of conditionals defining boundaries conditions.

From a DVM standpoint, registering tasks is looping on all tiles, computing dependencies, registering corresponding data-blocks and then registering the corresponding tasks. If the tile-space is too large, the tile generation can be inserted into an independent task, running as the program goes, to avoid having to unfold statically the whole computation before beginning the computation. One aspect that is still limiting us is that we need to compute the tile size statically at code-generation time. Indeed, keeping this parameter in the polyhedral model makes it too complex and prevents the polyhedral analysis to yield actionable code. This generates an undesirable dependency with the data-set that we avoided by generating the code for several tile sizes, enabling dynamic switching between them depending on program parameters.

5.7 Example of code generation : Jacobi 1D

```
for (t = 0; t < T; t++) {
  for (i = 0; i < N; i++) {
S:  b[i] = a[i-1] + a[i] + a[i+1];
  }
  for (i = 0; i < N; i++) {
T:  a[i] = b[i];
  }
}
```



During the internship, I worked on Jacobi method applied on a 1D array. We can see the above the algorithm. It involve 2 statements S and T. The first one is the computation requiring 3 elements from *t-1*. The second one is the copy of the result on the correct case on the current *t*. Graphically, S are the gray points while T are the black ones. We also can see the dependencies on the graph. We can express them as follow. $\langle S,t,i\rangle$ depends on $\langle T,t-1,i-1\rangle$, $\langle T,t-1,i\rangle$ and $\langle T,t-1,i+1\rangle$. Note that the lexicographic order guarantee the correct execution order of S and T.

One of the main objective was to tile this loop nest. To do so I applied the formulas seen before (5.1). Thanks to poco [1] I computed the Hyperplanes which will allow us to tile the loop nest. I found $\Phi S(t,i) = (t, 2t+i)$ & $\Phi T(t,i) = (t, 2t+i+1)$. On the figure 5 we can see that we changed both axis (abscissa, ordinate) to be parallel to the hyperplanes. We can see the domain transformed after Φ, on the tiling hyperplane. What's interest us is to know the dependencies between tiles. Indeed tiles are groups of statement which can be executed atomically. To do so we used ISCC [14], which is a polyhedral calculator to prototype polyhedral transformations. We use it to do a projection with the linear expressions of the dependencies between tiles.

Listing 1: expression of a dependence

```
#(T,t-1,i-1) --> (S,t,i), 0 <= t-1 <= T-1 /\ 1 <= i-1 <= N-2
    D := [T,N,T1,T2]->{[T1_source, T2_source,t,i]: exists r1,r2,r3,r4:
    0 <= t-1 <= T-1 and 1 <= i-1 <= N-2 and

    #define T1,T2 in terms of (t,i)
    t = 4*T1 + r1 and 0 <= r1 < 4  and
    2t+i = 4*T2 + r2 and 0 <= r2 < 4 and

    #define (T1_source,T2_source) in terms of (t,i)
    (t-1) = 4*T1_source + r3 and 0 <= r3 < 4  and
    2*(t-1)+(i-1)+1 = 4*T2_source + r4 and 0 <= r4 < 4};
```

The previous listing show us how To feed ISCC. We basically give the affine constraints that define the dependencies between tiles as seen before (5.1). These constraints are stored into the variable D. For the example, I chose here to show you the dependence $\langle T,t-1,i-1\rangle \rightarrow \langle S,t,i\rangle$.

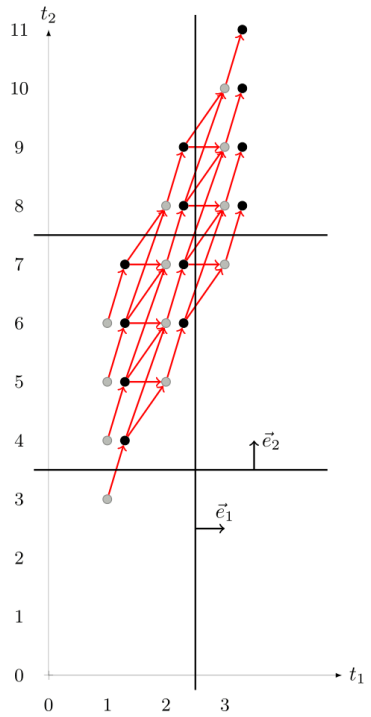


Figure 5: tiled Jacobi 1D

Listing 2: Operations with ISCC

```
proj := {[T1_source, T2_source,t,i] -> [T1_source, T2_source]};
tiled_D := proj(D);
codegen tiled_D;

card D;
```

Once we got the expression of the dependencies, we can compute a projection as the shown above (*proj*). ISCC can also generate the loop nest in C according the projection (*codegen*). *card* compute the carnality of the dependence D. It is useful to know the weight of data to communicate.

6 Conclusion

In this report, we have presented an approach for auto-parallelism starting from scratch. Relying on polyhedral models applied to the Distributed Virtual Machine (DVM) which is practically a distributed blackboard system, we developed a source-code transformation model aimed at parallelizing loops on multiple machines. Using established code analysis tools such as PoCo, we were able to extract loop hyperplanes to generate an efficient tilling. This tiling was then expressed using a code-generator to call the underlying runtime. The DVM itself is a relatively simple task engine with the particularity of being able to move data in a global manner, satisfying the transverse data-management propriety present in blackboards. Each tile is a set of data coupled with a given function (generated) which is to be called to generate the final result.

Working on examples such as a matrix product we explored the possibility of automating such transformation, our final goal being to expose a pragma like syntax outlining candidate loops in order to map them on multiple nodes. In this process, we focused ourselves here on the code transformation itself and on its expression in the DVM. We were able to unfold such loops, transforming them into data-flows, more suitable for parallel scheduling. Meanwhile, the DVM was launched on large nodes, validating its scalability for up to one hundred cores.

Overall, our parallelization technique outlines that tools for auto-parallelism are rapidly evolving supporting this challenging task. Still, code inertia, requiring strong analytical capabilities to extract a single section of a given program are still very complex. It means that to be practical, we will need to work on capturing loop candidates in a more exhaustive manner. Moreover, our code generation was based on a single problem size as keeping data-size and tiling size as a variable was not manageable in terms of complexity by the polyhedral compiler. It means that practically, our method requires some preconditioning to generate suitable code, this is a strong limitation that would be interesting to mitigate in the future. Yet we are convinced that parallel systems are putting an increasing pressure on programming models up to a point where a programmer will need abstractions to express sufficient parallelism – data-flows being a convenient expression for such parallelism. The questions is then to find the most suitable conversion mechanism to transition from imperative languages to more data-oriented models.

7 Future Work

We have seen that generating parallel code automatically was not a trivial undertaking. In particular, the polyhedral model despite being a powerful tool has some limitations. Having too many parameters or complex boundaries conditions can lead to an unmanageable polyhedral model, generating very complex code, or simply failing to be analyzed. Therefore, finding an expression of the initial code favoring code expressivity and avoiding to guess several parameters

is crucial for the eventual feasibility of such work. On a second aspect, our current experiences were limited to a single node, having one processing unit per core unfolding the data-flow, having our distributed data-model ready, it would be interesting to integrate Remote Procedure Calls (RPCs) in the loop to create a truly distributed data-flow.

Indeed, a lot of the complexity we have to face is associated with the fact that we need to extract information from an existing C program. This is a point that we see a important as we cannot ask all HPC programs to be fully rewritten. Yet, there is clearly some room to explore some component oriented approach to couple an existing application and a piece of code as the one we tried to generate in this work. Right now it is a pragma like mode with implicit barriers at the end, we need to look at this relationship in a more general manner to create some general scenario for automatic parallelization. We think that ad-hoc data structures and on the-fly function replacement could be a way. Otherwise, a DSL or code scaffolding tool could abstract further what is inside the program to enable its re-targeting prior to code generation. This later would have the advantage of simplifying test-case handling which we have seen to be a big contributor in the polyhedral complexity.

From the runtime side, we would benefit relying on existing runtimes instead of developing similar features again. Moreover, for the distributed aspect, we identified RPCs as good candidates to support the features of our runtime which are mostly one-sided (request). In addition, RPCs will provide a simple way to execute remote code materializing the data-flow we try to implement on the whole system in the DVM.

A last point which needs some research is scheduling under data-constraints. In this work we were considering the case where the DVM parallel region was launched inside an MPI program already running. It means there would be an a-priori data-splitting. We then need to be able to express this splitting during the polyhedral transformation to limit data-transfers. This relationship between processing and data is also tightly linked with how we could dimension task-level parallelism, to adapt the load to multiple kind of devices (CPU + GPUs, for example).

Overall, this training yield several questions and opened the way for more in-depth research around the idea of expressing tiling towards a distributed runtime (DVM) mimicking what the polyhedral model excels at, generating code for systolic architectures in the form of data-flow. There is clearly more to explore, in particular when it come to defining bridges between this kinds of computations.

References

[1] Christophe Alias. *Contributions to Program Optimization and High-Level Synthesis*. Habilitation à diriger des recherches, ENS de Lyon, May 2019.

[2] Michail Alvanos, Montse Farreras, Ettore Tiotto, José Nelson Amaral, and Xavier Martorell. Improving communication in pgas environments: Static and dynamic coalescing in upc. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 129–138, 2013.

[3] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. A survey of mpi usage in the us exascale computing project. *Concurrency and Computation: Practice and Experience*, 32(3):e4851, 2020.

[4] J. B. Besnard, A. D. Malony, S. Shende, M. Pérache, P. Carribault, and J. Jaeger. Towards a better expressiveness of the speedup metric in MPI context. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 251–260, Aug 2017.

[5] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[6] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, pages 1–3, 2010.

[7] Iain D Craig. Blackboard systems. *Artificial Intelligence Review*, 2(2):103–118, 1988.

[8] Rob Farber. *CUDA application design and development*. Elsevier, 2011.

[9] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[10] Robert W Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM New York, NY, USA, 1998.

[11] Dominik Reinhardt, Udo Dannebaum, Michael Scheffer, and Matthias Traub. High performance processor architecture for automotive large scaled integrated systems within the european processor initiative research project. *SAE Tech. Paper 2019-01*, 118, 2019.

[12] Lukas Sommer, Jens Korinth, and Andreas Koch. Openmp device offloading to fpga accelerators. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 201–205. IEEE, 2017.

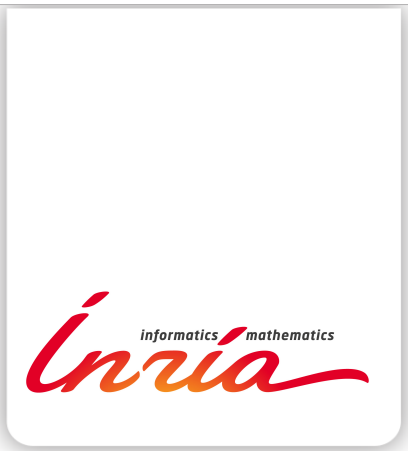
[13] Herb Sutter et al. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[14] Sven Verdoolaege. Counting affine calculator and applications. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11), Charmonix, France*, 2011.

[15] Michele Weiland. Chapel, fortress and x10: novel languages for hpc. *EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706*, 1, 2007.

[16] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. Upc++: a pgas extension for c++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114. IEEE, 2014.

Contents



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399