Resource Centered Computing delivering high parallel performance

Jens Gustedt
INRIA Nancy – Grand Est, France
ICube, Strasbourg, France
Email: Firstname.Name@inria.fr

Stephane Vialle, Patrick Mercier
SUPELEC Metz Campus, France
UMI Georgia Tech – CNRS 2958
Email: Firstname.Name@supelec.fr

AlGorille INRIA project team

***Abstract*—Modern parallel programming requires a combination of different paradigms, expertise and tuning, that correspond to the different levels in today's hierarchical architectures. To cope with the inherent difficulty, ORWL (*ordered read-write locks*) presents a new paradigm and toolbox centered around local or remote *resources*, such as data, processors or accelerators. ORWL programmers describe their computation in terms of access to these resources during *critical sections*. Exclusive or shared access to the resources is granted through FIFOs and with read-write semantic. ORWL partially replaces a classical runtime and offers a new API for resource centric parallel programming.**

We successfully ran an ORWL benchmark application on different parallel architectures (a multicore CPU cluster, a NUMA machine, a CPU+GPU cluster). When processing large data we achieved scalability and performance similar to a reference code built on top of MPI+OpenMP+CUDA. The integration of optimized kernels of scientific computing libraries (ATLAS and cuBLAS) has been almost effortless, and we were able to increase performance using both CPU and GPU cores on our hybrid hierarchical cluster simultaneously. We aim to make ORWL a new easy-to-use and efficient programming model and toolbox for parallel developers.

***Index Terms*—resource centered computing; read-write locks; clusters; accelerators; GPU; experiments; performance;**

I. Introduction and Overview

Recent years have seen an increasing diversification in computing architectures and software in the attempt to cope with parallelism. Parallelism is crucial to deliver the performance that medium and large scale applications nowadays require, but usually still demands a lot of design, programming and maintenance effort. Models and tools for parallel or distributed computing are multiple, most commonly used seem to be different forms of CPU threads (POSIX [1], OpenMP [2], TBB [3]), accelerator threads specific to NVIDIA GPU (CUDA [4]) or more generic (OpenCL [5], OpenACC [6]), and message passing (MPI [7]). Some low level implementations for vector processing units (SSE or AVX [8]) are also used by expert communities to develop vector computing libraries on CPUs. Less commonly used in performance critical contexts are functional programming approaches [9], modeling within the BSP and similar models [10], software or hardware transaction models [11], resource oriented architectures (ROA) [12], skeletons [13] or map reduce [14].

A current trend to facilitate parallel code development consists in using optimized scientific libraries and directive-based parallel programming languages. A greater number of developers is capable to design and to implement parallel applications using these paradigms. Sophisticated runtimes may then optimize each parallel run for a particular distributed target architecture. This approach aims to improve the ease of parallel developments, to reduce development times ([15], [16]) and to augment portability. However, the expressiveness of such development models can be insufficient for certain algorithms, and the performance of the runtimes can be too limited. In this case, lower level parallel programming tools are required to attempt to compensate these limitations (see Fig. 1 left). Then the risk is to provide too complex programming models to the developers, requiring great efforts to design and implement explicit task synchronization, resource sharing, communications and computations overlapping... We need programming models and tools for large communities of *developers*, not for few *high level experts in parallelism*, adapted to various kinds of problems and heterogeneous parallel architectures.

We present a new paradigm and toolbox called ORWL (ordered read-write locks) that changes the view on parallel and distributed architectures in that it centers around the *resources* that are necessary for a computation. Such resources can be local to the process or remote, may be data (input, output or intermediate results), memory, accelerators (GPU, FPU, co-processors), program code (compiled offline or in place) or network links and other communication features. Effectively, ORWL partially replaces the classical runtime level and offers a new API for resource centric parallel programming that complements classical multi-thread and many thread parallelization tools, see Fig. 1 right.

The aim is to have the programmer describe his computation in terms of access to these resources during *critical sections*, *i.e.* sections of code during which exclusive (write) or shared (read) access to the resource is granted. That description of the principal structure of the program is *explicit* and *static*, that is it is provided through syntax, namely through annotated blocks of the supporting programming language, here modern C, see [17].

The access to these critical sections (and thus the modeled resources) is regulated by simple rules: they combine a FIFO, read-write semantics and mapping into a unique tool. This uniform access to critical resources is a small restriction in terms of expressiveness of a parallel program, but we will show that this restriction is largely compensated by clear semantics (that allow for stringent proofs) and better optimization opportunities, in particular data prefetch and other computation/communication overlap.

This paper is organized as follows. After relating ORWL to previous work about parallel programming models and support tools (Section II) we introduce ORWL (i.e programming model and library interface) in Section III. That section more specifically explains the structural information about the application that is needed during execution startup and how an individual computation and communication phase of an application is organized. Then, in Section VI, as a principal example we present experiments with a classical block-cyclic matrix multiplication algorithm that runs on different categories of platforms ranging from small multi-core machines, over machines with accelerators (GPU in that case), a cluster of such machines, a multi-processor machine, and a large multi-core cluster. We then conclude in Section VII.

II. Related work

The inherent programming models that underlie existing parallel programming tools strongly characterize (or constrain) the ease or burden of programming with them. Generally there is a trade off between expressiveness and generality on one side and resource usage and performance on the other. In this section we will briefly review existing tools and compare their choices concerning such a trade off to the one we are proposing with ORWL.

Architectural hierarchy: Different parallel programming models and tools designed to hierarchical parallel architectures, consider different abstract hierarchical architectures: with more or less details, with more or less parallelism levels. Some models consider tasks spread on numerous levels in the parallel architecture, such as the GPH language [18], that refers to a set of clusters of multicore nodes including vector accelerators. At the opposite, some models as HiDP [19] or Chapel [20] consider tasks or a hierarchy of tasks running on a set of interconnected computing nodes and view them as a generic 1 or 2 level parallel architecture. ORWL can run on a multi-level hierarchical architectures, but considers only a set of multicore nodes and does not differentiate a cluster or a cluster of clusters. Moreover, it allows to run different sorts of compute kernels, designed for distinguished types of cores including different vector accelerators, but does not express vector parallelism or data transfers between CPU node memory and accelerator memory. We consider that these issues are relevant for compute kernel design and implementation, and might require specific programming tools (such as CUDA or OpenCL).

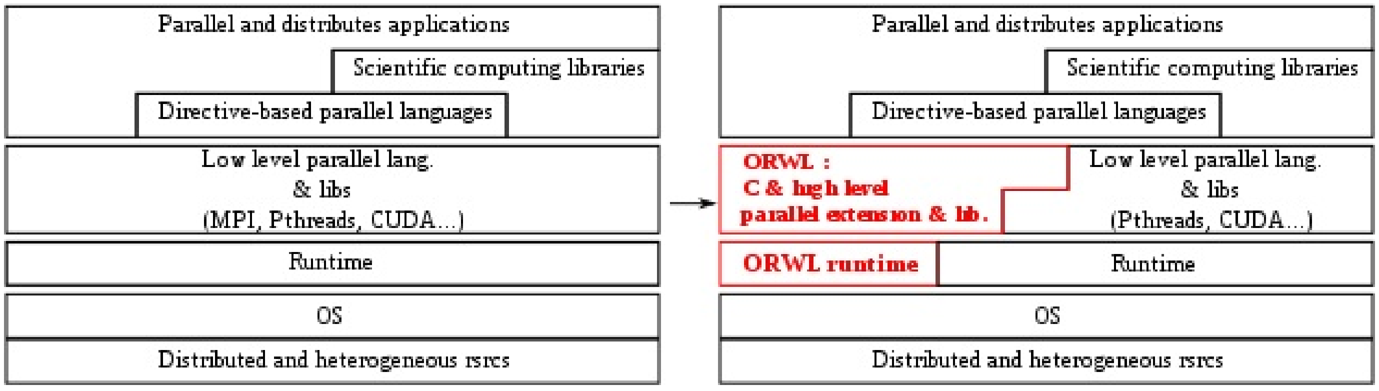
Architecture exposure: Some programming models aims to virtualize and hide the core architecture. Ambitious ones, such as HiDP [19], attempt to automatically generate efficient compute kernel codes from a high level source code. Others propose a comfortable interface with a kernel programming language considered as highly portable. See for example JavaCL[1], a Java extension interfaced with OpenCL. However, it is usually mandatory to redesign data storage and data access strategy to get high performance kernels on different core architectures, even with the same kernel programming tool [21]. Yet other models prefer to clearly expose the core architecture and encourage users to develop very optimized kernels, such as Sequoia [22]. ORWL follows this approach: users have to use adapted compute kernels for the different core architectures they aim to use, by either referring to available highly optimized kernels (such as ATLAS or CuBLAS) or by programming them directly with adapted low level programming tools (CUDA, intrinsics, or OpenCL) to track maximum performances.

Expressiveness for Parallelism: Modern parallel computing models create and manage *tasks*, but in very different ways.

- Chapel and Sequoia give a fine control of parallelism to their users.
  The programming model of Sequoia requires users to define explicit inner and leaf tasks, and to create an explicit task tree. An inner task uses a call-by-value-result mechanism (some kind of improved RPC) to call and run its subtasks in the tree, inducing vertical communications and synchronizations [22].
  Chapel's model expresses parallelism explicitly with `forall` loops and `cobegin` statements, and achieve synchronization with *future* variables (locking when read before to retrieve the result of a parallel computation) and *atomic regions* [20].
- At the opposite, GPH and HiDP attempt to hide more parallel programming details to users.
  GPH is a functional language able to achieve parallel or sequential evaluation of function arguments, and to define evaluation strategies. Then some parallel skeletons can be defined (like *parallel-map* and *divide-and-conquer*) associated to task distribution strategies, and become high level parallelism constructs [18].
  HiDP supports some *array operations* and requires users to define *map blocks*. They lead to an automatic exploitation of data-parallelism inside the map blocks, encapsulated by implicit synchronization barriers. However, some directives can be added to map clauses by users to guide the compiler and to improve the parallelization [19].
- Finally, most of modern parallel programming model designed for hierarchical parallel architectures, support and take advantage of nested parallelism.

ORWL first requires the user to create some *tasks* and identify some *resources* (data, hardware, software entities). Second, a planning of the resource accesses from the different tasks has to be explicitly described. Then a local synchronization is automatically deduced for each task, and task communications are automatically achieved on resources accesses at runtime (see section III). So, ORWL parallelism expressiveness remains explicit and based on tasks, but task

[1]https://code.google.com/p/javacl/



Parallel and distributes applications
Scientific computing libraries
Directive-based parallel languages
Low level parallel lang. & libs (MPI, Pthreads, CUDA...)
Runtime
OS
Distributed and heterogeneous rsrcs
Parallel and distributes applications
Scientific computing libraries
Directive-based parallel languages
ORWL : C & high level parallel extension & lib.
Low level parallel lang. & libs (Pthreads, CUDA...)
ORWL runtime
Runtime
OS
Distributed and heterogeneous rsrcs

Figure 1. Software suite in parallel and distributed systems, without ORWL (left) and with ORWL (right)

synchronization and communications are resource-centric and deduced from a *task-resource graph* description.

Data and computation locality: To minimize communications and achieve high performances, data and computation locality is critical in a hierarchical parallel architecture. Different programming models allow to express this locality in very various ways.

- GPH [18] allows to control the *distance* between a new task and its parent task in a multi-level hierarchical architecture.
- Chapel [20] requires to express the distribution of arrays on the local memories of the different computing units. Some distribution policies can be reused and improved by application semantic knowledge.
- Sequoia [22] uses a *task isolation* approach. It runs tasks by sending their data and retrieving their results . Users can reduce the impact of the induced communication by running tasks that work with local data or by using some available efficient hardware block transfer mechanisms. Locality is managed by the user.
- HiDP [19] runs on one node hosting CPU and GPU cores, and faces a more focused locality issue. Data locality management is also critic inside a GPU. HiDP automatically generates several CUDA compute kernels (for each data-parallel source routine) with different data locality management and generates code to benchmark these kernels, and finally point out the most efficient.

ORWL has been designed to identify *resources*, representing data entities, hardware or software components, and then to plan the accesses to these resources. ORWL allows to differentiate local resources (only defined on the node hosting the ORWL process of the ORWL task) from remote ORWL processes hosted on another node. In fact, access operations are presented identical to the programmer. Only during initialization a differentiation between local and remote resources is necessary. Then, at runtime, the library guarantees optimized access to data resources: data that is already present locally is directly visible in the address space of the task, no copy operation is performed. Only when remote data is requested, a network transfer is initiated in the background and the data is written into a buffer that is directly accessible to the task. Again, no additional user space copy operation is necessary.

III. Resource centered modeling with ORWL

In this section, we describe the foundations of the underlying synchronization model for ORWL. ORWL is based on four major concepts that compose its model of computation, *tasks* as units of program execution, *resources* as an abstraction of data or hardware on which tasks interact, *handles* as means of access of the tasks to these resources and *critical sections* to organize access to resources and computation. Major theoretical properties and proofs for them can be found in [23]; in particular, there it is shown how to construct iterative applications as a set of *autonomous* tasks such that any execution has guarantees of liveness and fairness.

Tasks: For this model we suppose that a given set of tasks $\mathcal{T}$ is to perform some computation and that data dependencies exist between these tasks. We also suppose that tasks may be recurrent, as part of an iterative application. Data dependencies are distinguish read and write operations that are not necessarily atomic. Therefore a dependency of task v from task w is modeled by v reading data that w has written previously. Hence, v may only be executed while w is not. Our model provides a way to control the execution order of tasks algorithmically based on their data dependencies. ORWL tasks run *concurrently*, they are controlled *autonomously* trough dependencies on resources (there is no centralized scheduler) and they may be *heterogeneous*, that is each task may run a different compute kernel designed to be executed on a CPU core, on GPU cores or on other accelerators. ORWL tasks and OS processes or threads are only loosely related. In fact, one OS process can realize one or several tasks, while several OS threads will be used by ORWL to realize a task.

Syntactically, an ORWL task is coded by the programmer as a *task data structure* (**taskType**, say) that holds relevant data for the task and a *task function* that describes the processing of the task. The ORWL library provides some syntactic sugar to ease task declaration (ORWL_DEFINE_TASK(**taskType**)) and creation (**taskType_create_task**), but we will not go into details of these tools, here.

Resources: To each task we attribute a fixed number of control data structures coined *resources*. To each such resource ORWL associates a FIFO that regulates the access to it. Such an access can be done by any task τ, regardless whether it is the local task to which the resource is attributed or any other remote task.

- Prior to an access, τ inserts a request in the FIFO of the resource.
- Access is then granted, once that request becomes the first element in the FIFO.
- In order to grant access to the following requests in the FIFO, τ releases the resource as soon as it may.

Such requests follow semantics of read-write locks: several adjacent read request are served simultaneously, write requests are exclusive.

By choice of the application, a resource is usually associated to a data object; callbacks can be plugged onto such resources in order to manage the correspondig data in heap memory (default), files or memory segments. Other resources can be associated to hardware entities such as CPUs, GPUs, or GPU data streams.

Handles: Other than for classical tools such as POSIX mutexes, semaphores or read-write locks, access to an ORWL resource is not granted to a process, thread or task ID but to a *handle*, a separate data structure. The access through a handle of the resources is bound to the FIFO ordering as given above; that is a task would use a handle to first place a *request* for a resource, eventually wait until the lock is *acquired*, and only then *map* the data of the resource into its address space. Each task may hold several handles that are linked to the same resource. By that a task that already has achieved access to a resource may at the same time insert another handle in the FIFO of the same resource for future access.

Critical Sections: With the help of handles, an ORWL application then organizes itself by means of *critical sections*. A critical section is a block of code that guarantees the desired access to a resource, either exclusively or shared (see Listing 1). The resource is locked on entry of the block and if necessary data is mapped in the address space of the task; on exit of the block, the lock on the resource is released and the mapping into the address space is invalidated.

Listing 1. a critical section for handle *myHandle*

```
ORWL_SECTION(myHandle) {
    ...
}
```

Syntactically, a critical section is just a normal C compound block that is prefixed by a macro that specifies the handle in question:

Expressiveness and usability: The emphasis of the ORWL programming model lies in its expressiveness of a *localized view* of computation. For each task we initially have to identify the data resources that it manages (see Listing 2) and then to relate the access to these resources to the access of other "neighboring" tasks, see Sec. IV-A below.

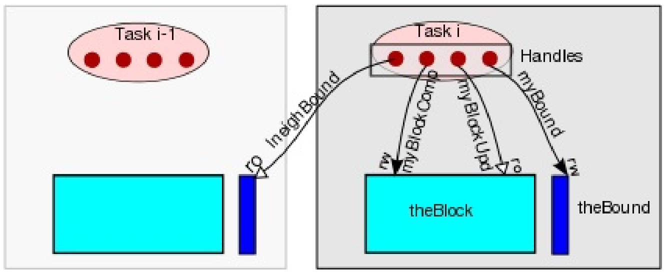


Figure 2. Structure of the example. One of the four handles refers to a remote resource. White arrow and *ro* represents a shared read access, black arrow and *rw* an exclusive write access.

Listing 2. a top level declaration of two ORWL resources per task

```
ORWL_LOCATIONS_PER_TASK(theBlock, theBound);
```

After that, programming of the effective computation phase (Sec. IV-B) is easy: ORWL guarantees that computation only takes place if data is available, routes computed data as it is needed by other tasks to them, and overall guarantees that the computation is dead-lock free and fair. For a more detailed explanation of these phases see below.

IV. An introductory example

With the above features, programming of parallel algorithms that achieves a *domain decomposition* and iterative computations is straightforward. ORWL allows to easily express an algorithm that:

- decompose the data domain into blocks with a well defined boundary relation between these blocks
- uses existing sequential code (or enhanced to use SIMD units, like AVX ones) for the computation on each block
- stitches the results of computations together on the boundary

Figure 2 illustrates this kind of algorithm by giving an ORWL implementation strategy. Each task is associated with two local resources, `theBlock` and `theBound`. In addition to its own local resources, each task accesses the resource `theBound` of the previous task.

For the example we will suppose that we have two applicative functions as follows:

Listing 3. Function interfaces provided by the application

```
void block_computation(size_t n, double data[n],
                       size_t m, double const oBound[m]);

void update_boundary(size_t m, double myBound[m],
                     size_t n, double const data[n]);
```

Here **block_computation** is supposed to be the compute kernel of the application. It receives the data block for computation in `data` and the boundary information that it needs from other tasks in `oBound`. **update_boundary** in turn, is supposed write an updated version of the boundary to myBound.

A. Initialization phase

Here we only describe a simplified usage mode that initializes the control structures (handles) once in the beginning of the application. Other modes are possible but their description

Listing 4. ORWL task code: initialization phase

```
 1 // Scale the data resources
 2 orwl_scale(sizeof(double[n]), theBlock);
 3 orwl_scale(sizeof(double[m]), theBound);
 4
 5 // Link handles to the needed resources
 6 orwl_handle2 myBlockComp = ORWL_HANDLE2_INITIALIZER;
 7 orwl_handle2 myBlockUpd  = ORWL_HANDLE2_INITIALIZER;
 8 orwl_handle2 myBound     = ORWL_HANDLE2_INITIALIZER;
 9 orwl_handle2 lneighBound = ORWL_HANDLE2_INITIALIZER;
10
11 // First (FIFO pos. 0) we will write our data block
12 orwl_write_insert(
13   &myBlockComp,                            //< handle
14   ORWL_LOCATION(orwl_mytid, theBlock),     //< resource ID
15    0);                                     //< FIFO position
16
17 // ... at the same time (FIFO pos. 0) we will read from
18 // our left neighbor, if it exist
19 if (orwl_mytid > 0)
20     orwl_read_insert(
21       &lneighBound,                            //< handle
22       ORWL_LOCATION(orwl_mytid-1, theBound),   //< resource
23        0);                                     //< position
24
25 // Then (FIFO pos. 1) we will have to update our bound
26 orwl_write_insert(
27   &myBound,                                //< handle
28   ORWL_LOCATION(orwl_mytid, theBound),     //< resource
29    1);                                     //< position
30
31 // ... request our own block for reading
32 orwl_read_insert(
33   &myBlockUpd,                             //< handle
34   ORWL_LOCATION(orwl_mytid, theBlock),     //< resource
35    1);                                     //< position
36
37 // Synchronize with the other tasks
38 orwl_schedule();
```

Listing 5. ORWL task code: iteration.

```
// Run the applicative computations
for (size_t orwl_phase = 0;
     orwl_phase < maxPhases;
     ++orwl_phase) {
  // computation operation
  ORWL_SECTION(&myBlockComp) {
    double* data = orwl_write_map(&myBlockComp);

    ORWL_SECTION(&lneighBound) {
      double const* lData = orwl_read_map(&lneighBound);
      // do the real computation here
      block_computation(n, data, m, lData);
    }
  }

  // update operation
  ORWL_SECTION(&myBlockUpd) {
    double const* data = orwl_read_map(&myBlockUpd);

    ORWL_SECTION(&myBound) {
      double* bData = orwl_write_map(&myBound);
      update_boundary(m, bdata, n, data);
    }
  }
}
```

is outside the scope of this paper, and this simplified mode is adapted to a large family of iterative computing algorithms. The idea of this simplified mode is that all resources and their access scheme are declared in an initial phase of the application. This access scheme will implicitly synchronize the tasks during the following computation phase. See Listing 4 for an example of such an initialization phase.

For this simplified initialization, a task specifies 4 parameters R, p, rw, it for each handle that it uses:

R	is the resource to which the handle refers,
p	is the initial *FIFO position*,
rw	is either *read-only* or *write* to describe the desired concurrency of the access, and
it	controls if a handle is used iteratively or not.

Syntactically,

it	is given through the type of the handle (orwl_handle or orwl_handle2),
rw	is coded in the name of the function as “read” or “write”,
R, p	are parameters. R is specified through the use of macro ORWL_LOCATION (lines 14, 22, 28, 34) which uses a task ID and the local resource name to identify a resource.

In the above example we can see how the concepts of local and remote resources and handles differ. In fact, here each task declares **two** local resources (Lst. 2) and during initialization (Lst. 4 line 1) scales them to the appropriate size. Then **four** handles (orwl_handle2, line 5) are used to access **three** different resources: its own two and one of a neighbor.

- The requested access for the “own” block of data is once for exclusive write (line 11) and once for a shared read (line 31).
- Access to the boundary block of the neighboring task is inclusive (line 17), only for reading.
- Access to the “own” boundary block is exclusive (line 25).

In our example, the initial FIFO position p has two values that correspond to the logical order in which a task will later (Section IV-B) access the resources. There is a first phase where a task reads data from a neighbor and updates its block (FIFO position 0), and a second phase (FIFO position 1) where the task then only needs to read its block and updates its Boundary.

Each task finishes that initialization phase by a call to orwl_schedule (line 37) which implements a global barrier for all tasks and ensures that the FIFOs of all resources of all tasks are initialized with requests that are consistent with the initial 4 parameters that were specified for all handles. In [23] it has been proven how such an initialization can be implemented such that the subsequent computation phases are guaranteed to be *deadlock free* and *fair*.

B. Computation phase

During the computation phase, tasks usually access the resources they specified during initialization within *critical sections* that restrict simultaneous access to a specified resource. This restriction is inclusive or exclusive as the insertion of the request was read or write, see Section III. Critical sections of tasks that access different resources may be processed concurrently without any additional restriction.

Syntactically the ORWL library implements a critical section as prefix ORWL_SECTION(handle) to a normal block of C code, see e.g Listing 5 line 6. Inside such a block it may request access to the data of the handle by mapping it into its address space, line 7. In the example we see two separate operations, the first for the core of the computation, starting line 5, the second for the update of the "boundary", starting line 16. In a more sophisticated implementation these two operations of the task could easily be run by different OS threads, but the presentation of that feature is beyond the scope of this paper.

Once the task reaches a critical section it is blocked until the resource is available. The associated data is accessible to the task after mapping it explicitly. As long as execution stays inside that block, this access may be simultaneously read-only shared with other tasks, or exclusive if requested during the initialization. Once execution leaves the section, the address to which the data is mapped is invalidated and the lock on the resource is released.

If the handle was declared to be used iteratively (using orwl_handle2 data type), special care is taken to allow the task to revisit the same critical section, again. Once the access is granted to such a handle, a new request is automatically appended to the FIFO of the resource. Thereby the system ensures that the access to that same resource is granted iteratively in the same initial FIFO ordering, and the iteration loop, starting in line 2, proceeds predictably.

V. A BENCHMARKING EXAMPLE

A. *Benchmark objectives and choices*

As example code for the benchmarks, we chose to implement the well-known block cyclic algorithm for dense matrix multiplication. The advantage of dense matrix multiplication with this algorithm is that there exist very performing libraries (for CPU and GPU) that can be used as subroutines to build parallel or distributed implementations on top.

Using that algorithm, we have implemented a *reference version* of a dense matrix product, using MPI, OpenMP and CUDA programming tools, and ATLAS (for CPU computations) and CUBLAS (for GPU computations) as scientific libraries for dense matrix multiplication. We have run both this reference program and the ORWL version on our cluster in order to measure the absolute and relative performances achieved by ORWL.

B. *Benchmark algorithm*

We implemented a dense matrix product framework with ORWL using a classical block-cyclic parallel algorithm. Each ORWL process stores some fixed horizontal slices of A matrix, some *circulating* vertical slices of B matrix, and computes some fixed horizontal slices of C matrix. Figure 3 shows the initial data stored in the first two ORWL processes. Node 1 stores the top slices of A and the left slices of B, and aims to computes the top slices of C. Node 2 stores the next slices of A, B and C. Then, each node is going to compute some subpart of its slices of C, corresponding to the available slice of B.

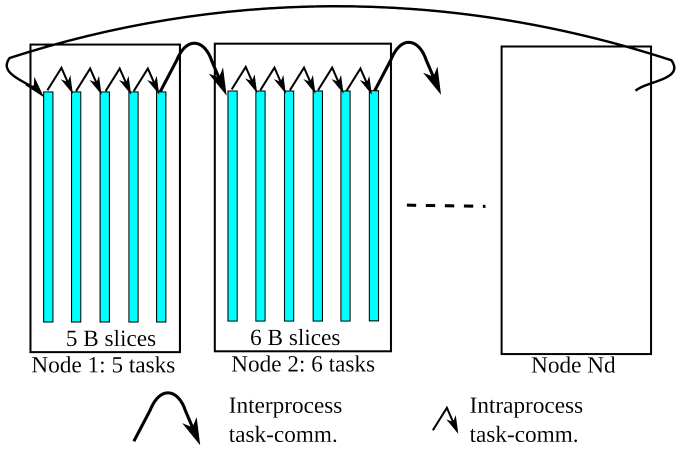


Figure 4. Circulation of the slices of matrix B between tasks, intra and inter processes.

But in fact, each node is composed of several computing resources (some CPU cores and sometimes an accelerator like a GPU), and each resource is exploited by one or several ORWL tasks (typically one task per resource). As ORWL programming model is *task-based*, it is better to consider that each task owns an horizontal slice of A with a thickness depending on its computing power: the task run on a GPU owns a larger slice than a task exploiting a CPU core. This partitioning strategy allows to achieve a static load balancing across all the computing resources. Each task owns also a vertical slice of B, but all these slices have the same thickness (see Figure 3). Finally, each task has to compute an horizontal slice of C, corresponding to its slice of A.

Then, slices of B are going to *circulate* across a ring of tasks, as illustrated on Figure 4. Task i sends its B slice to task $i + 1$ ans receives a B slice from task $i - 1$. Data circulation between tasks located on different ORWL processes are achieved by message passing on the interconnexion network, while data circulation between tasks located inside the same ORWL process are just some buffer pointer exchanges. However, when the task in exploiting an hardware accelerator, some data transfers are required to send the B slice in the accelerator memory and to retrieve the result from this memory (for example a GPU card memory). In our current version, these data transfers are not managed by ORWL, but are implemented and executed with the compute kernel devoted to the accelerator.

ORWL proposes to use the same semantic and syntax (see further) to express data circulation between tasks, while each achievement depends on the relative locations of the involved tasks. ORWL hides the exact implementation and execution of the task communications, and optimizes their achievement avoiding message passing and also memory copy when buffer pointer exchanges is enough.

Finally, when a task has hosted all B slices in its data circulation buffers, it has been able to compute all subparts

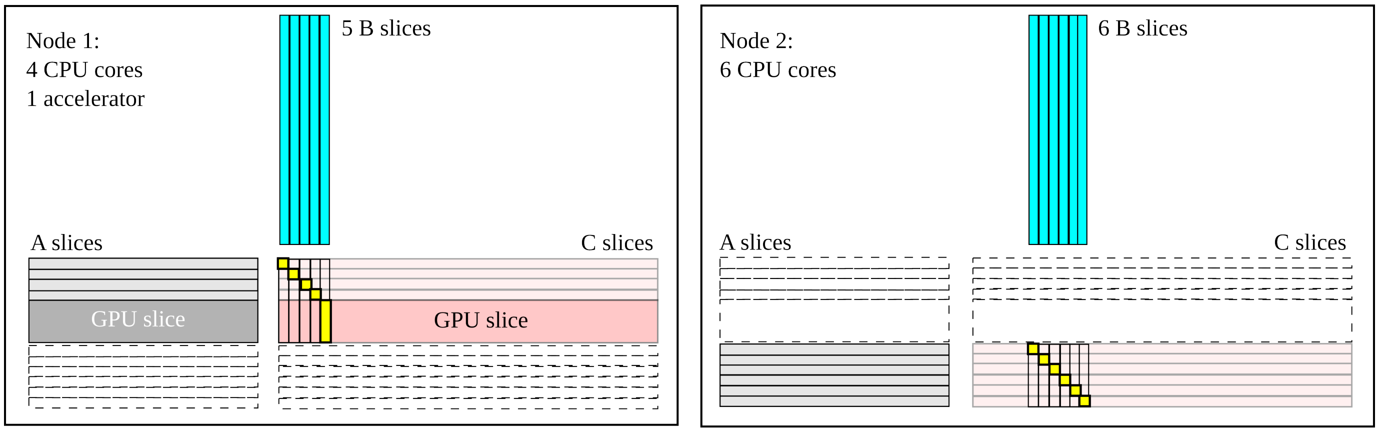


Figure 3. Two node view of a heterogeneous matrix partition on several nodes with different capacity. The column slices of B are all of equal size. The row slices of A are tuned to the capacity of the processing element (CPU core or accelerator chip).

of its slice of C. If we assume we have deployed a total of T ORWL tasks to compute the C matrix slices, then after T *computation-circulation* steps, C matrix computation is achieved and B matrix will be stored again in its initial state ready to be reused in another matrix operation).

C. ORWL implementation

We implemented the ORWL implementation of block cyclic matrix multiplication with an additional tool called orwl_circulate . In addition to the features of an orwl_handle2 as described previously, this automatically implements the circulation of the corresponding buffers under the hood. The simplified initialization of such a orwl_circulate is shown in Listing 6. Here the overall management of buffers becomes particularly simple: our algorithm only has *one* resource, the circulating buffer, to worry about. In a standard implementation as our reference implementation (using MPI, OpenMP and CUDA), several buffers (for current, previous and following phase) have to be maintained.

Listing 6. ORWL initialization for circular buffer.

```
/* A chunk of matrix B will circulate among the tasks. */
orwl_circulate circB = ORWL_CIRCULATE_INITIALIZER;

/* The initial scheduling is simple, here. We only use one
   location per task, through which the matrix B will
   circulate. */
orwl_circulate_insert(&circB, locationB);

/* Now synchronize to have all requests inserted orderly
   at the other end. */
orwl_schedule();
```

Listing 7 then shows the compute phase of the implementation. Similar to before, there is a ORWL_CIRCULATE_SECTION that marks the block of the critical section. Inside the critical section the application maps the data into the address space (orwl_circulate_map). Without the programmer having to worry, if possible this “mapping” accesses data that a neighboring task had just used in a previous phase through a pointer. If this is not possible (the task is located in a different process on a different node) the data is transferred behind the scenes.

Listing 7. ORWL matrix multiplication: iteration.

```
for (size_t phase = 0; phase < orwl_nt; ++phase) {
    /* Replace the data in B by the block that we received in
       the previous phase. */
    ORWL_CIRCULATE_SECTION(circ, allocB) {
        /* Obtain a pointer to the buffer in our virtual
           address space. */
        void const*restrict curB = orwl_circulate_map(circ);

        /* keep track of the current row of B that we
           handle in this phase. We start with orwl_mytid
           and then at each phase add one around the circle.
           */
        size_t iB = (orwl_mytid + phase) % orwl_nt;
        func->mult(n0, n1, n2, iB, orwl_nt,
                   C, A, curB, transBuffer, context);
    }
}
func->output(n0, n2, orwl_nt, C, context);
```

The call to the compute kernel is then done through func−>mult. Here this is a function pointer that can be used to trigger different versions of the kernel, e.g. versions that are adapted to a specific CPU or that use an accelerator. The arguments to the function are (1) size and index information about the matrices, (2) matrices C, A and curB, (3) an auxiliary buffer as it might be needed by some of the kernels, and (4) context information that might be needed to configure an accelerator.

Another function func−>output is then called after the main loop to retrieve the result of the computation. This function might be empty if the computation took place on the CPU or might perform a transfer from the accelerator RAM to main RAM, if necessary.

The following points ensure that ORWL is a good choice for such computations that are assembled using preexisting compute kernels:

- The straightforward implementation with ORWL of our block-cyclic parallel algorithm achieves a natural overlap of communications and computations.
- The ORWL runtime avoids superfluous copy operations as long as tasks are performed in the same address space.

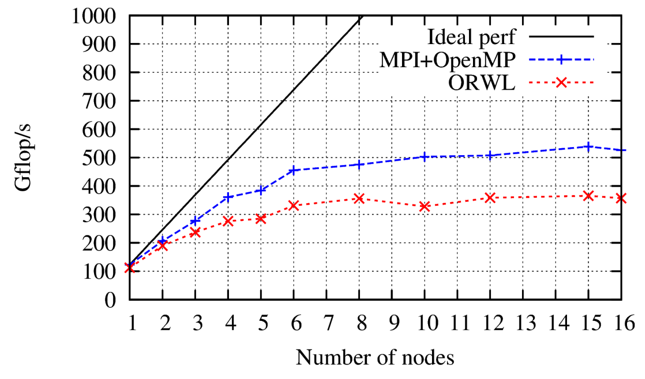


Figure 5. Gflop/s achieved on a constant size problem (5760 × 5760 matrices of double precision data) using only the CPU cores of *Cameron* cluster

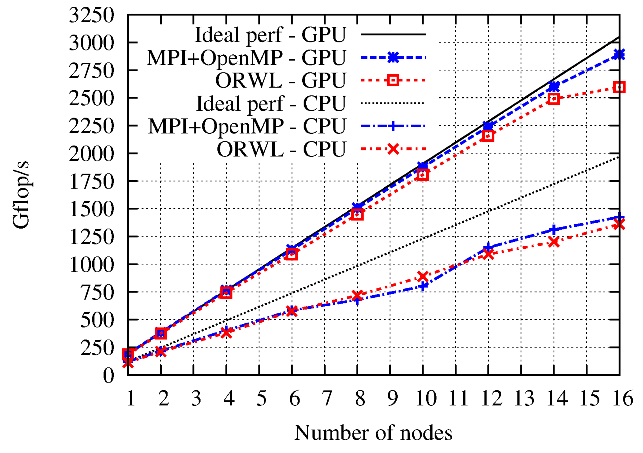


Figure 6. Gflop/s achieved on an increasing size problem: maximum size supported by the global amount of GPU memory of all *Cameron* nodes

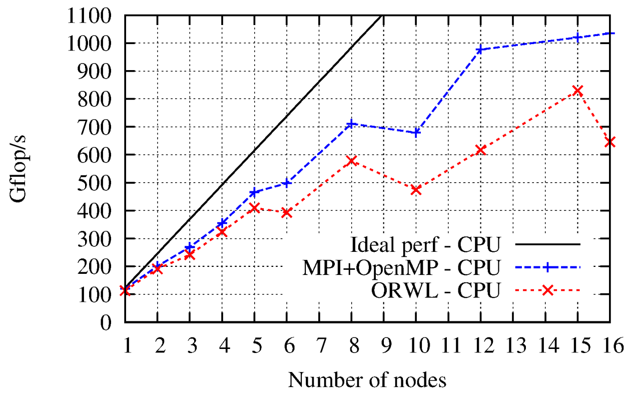


Figure 7. Gflop/s achieved on an increasing size problem attempting to keep constant the execution time while increasing the number of computing nodes, and using only the CPU cores of *Cameron* cluster

- The ORWL runtime ensures direct buffer to buffer communication between different compute nodes.
- Without difficulties, we have been able to reuse some *home-grown* compute kernel source codes.
- Similarly, we have easily called kernels of scientific libraries: BLAS/ATLAS on CPU, and CUBLAS on GPU.
- All these kernels can be used intermixed on the appropriate parts of the heterogeneous platform. In particular, CPU and GPU compute kernels can be run concurrently and use the full computing capacity of the dual hardware.

VI. Experiments

A. Testbeds

Most of our benchmarks we run on an experimentation cluster *Cameron* of SUPELEC, with 16 nodes that are interconnected across a 10-Gbit Ethernet switch, an OmniSwitch Alcatel 6900-X20-F, with up to twenty 10-Gbit/s ports. Each node has an Intel Xeon E5-1650 processor at 3.2 GHz, composed of 6 physical hyperthreaded CPU cores (12 logic cores), and is equipped with 8 GiB of global DDR3 RAM on a 1600MHz memory bus. Moreover, each node includes a NVIDIA GeForce GTX580 (FERMI architecture) with full double precision capacity, 512 CUDA cores and 1.5 GiB of memory that is connected to its CPU across a Gen3 x16 PCIe bus.

We used other platforms for additional tests, that represent different development or production context. First a laptop machine, with 2 physical cores plus 2 hyperthreaded cores, a multiprocessor-multicore machine *LeMans* with in total 24 cores at 800 MHz (located at the ICube lab) and a cluster *Pastel* with up to 60 processor and 180 cores on the Grid5000 platform.

B. Scalability experiments

In order to evaluate the performance of ORWL, we run some benchmarks on the *Cameron* cluster (up to 16 nodes, see Section VI-A) using CPU cores, GPU cores, or both (hybrid computing). All performance measurements are given with the achieved number of floating point instructions per second (flop/s), accounting two double precision instructions per multiply-add of a vector-vector product. We only show performance measures based on ATLAS for CPU and CUBLAS for GPU computations; evidently our home grown implementation of dense matrix multiplication has proven to be less performing than these highly optimized libraries. Nevertheless, their high performance is due to an extreme use of the available hardware and is bound to arithmetic conditions of the matrix sizes. In particular, some of the measurements below show fluctuations that we blame on the varying divisibility conditions when increasing matrix sizes.

Figure 5 shows performance measurements (in Gflop/s) that were achieved by our matrix product implementations (reference implementation and ORWL) on fixed size matrices, namely 5760 × 5760 double precision data. The MPI+OpenMP+Atlas code performs a bit better than ORWL and reaches approximately 500 Gflop/s on CPU cores whereas the ORWL code reaches only 360 Gflop/s. Both reference and ORWL codes achieve poor scalability and don't scale well beyond 8 nodes: a finer decomposition of the input matrix for a fixed sized problem leads to more and smaller message between the increasing number of nodes. We conclude that the parallelization overhead of the ORWL implementation remains more important than for the reference code, and can be improved, but that the scalability problem is the same for both.

For larger problem sizes, Figure 6 then shows close performances and good scalability achieved both by the reference and the ORWL codes. This benchmark processes problems with a data size that increases linearly with the number of nodes: on P nodes it processes the maximum problem size that can be stored in the total GPU memory of the P nodes. Performance on the CPU is still not so close to the ideal performance (see bottom part of Figure 6), but it increases steadily and *scales* up to 16 nodes. Performance on the GPU (upper part of Figure 6) is very close to the ideal performance: it only deteriorates slightly when using 16 nodes.

The difference in the scalability when using CPU and GPU cores can be explained by the communication overhead. When computing on GPU cores, the CPU cores remain available to perform the internode communication. Thus we can achieve a very efficient overlap of computations and communications. When computing on CPU cores, internode communications and computations compete for CPU and for memory accesses, leading to an imperfect overlap. We note that ORWL based code is as efficient as the reference code when processing large amount of data.

Figure 7 shows the performances achieved by ORWL based and reference code on a benchmark with an intermediate size increase. For a product of two matrices of size N performing $2 \cdot N^{1.5}$ floating point operations, a linear increase in $N = k \cdot P$ in the previous benchmark (Figure 6) leads to increase the amount of computation on each node ($(2 \cdot k^{1.5}) \cdot \sqrt{P}$ flop per node on P nodes), and so to a total increase of the execution time. The benchmark of Figure 7 keeps the amount of floating operations constant on each node, by adjusting the increase of the problem size to a sublinear function. A classical industrial use case for this kind of benchmark consists in running finer simulations on larger machines, but respecting the same deadline as before. Figure 7 shows the performances of ORWL based and reference codes. The performance increase for both is suboptimal and fluctuating, due the arithmetic conditions on the matrix sizes as mentioned above, and ORWL code remains slower than reference code. But both codes still succeed to increase their performances and to scale.

These three benchmarks show that the ORWL application has a performance behavior that is similar to the reference code: limited scalability and performance for fixed sized problems, increasing but sub-optimal performance for increasing sized limited to a fixed amount of computation per node, and strong scalability for saturating problem sizes. For later, ORWL achieves very similar performance than the reference code.

To complete the picture, Figures 8 and 9 present scalability experiments on a multi-core processor (*LeMans*, one process, up to 24 ORWL tasks) and on a cluster of multi-cores (*Pastel*, one process with 4 ORWL tasks per node), respectively.

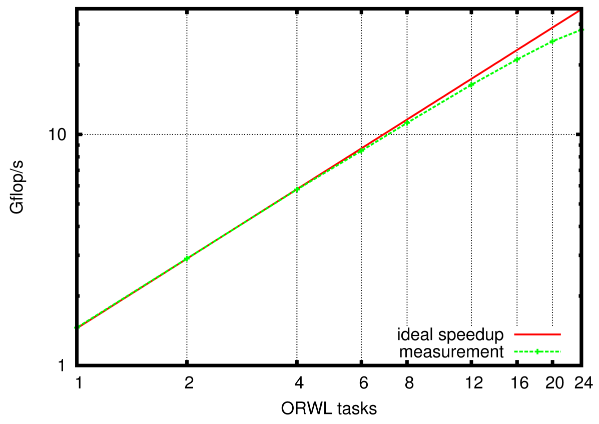


Figure 8. Gflop/s achieved by using an increasing number of ORWL tasks on *LeMans* with constant matrix size of 11520×11520 `double` and ATLAS `dgemm` as compute kernel. Also shown is the achievable limit with an ideal speedup.

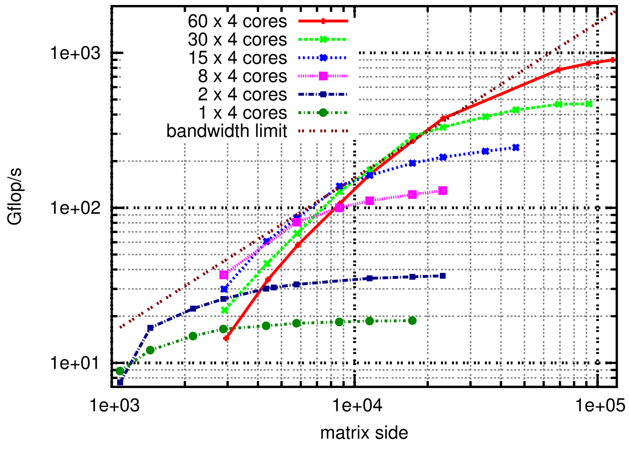


Figure 9. Gflop/s achieved on an increasing problem size on *Pastel* using ATLAS `dgemm` as compute kernel. Also shown is the achievable limit on that platform induced by the usable network bandwidth.

The first shows an almost perfect scaling, seemingly the architecture copes well by having one compute task per core.

The second also shows a typical behavior as we should expect on such a hierarchical architecture. If we take the "60×4" curve as an example we can identify three different performance ranges.

1) In a first range up to a matrix side of 10^4 elements for this curve, network latency dominates the computation. The parallelization doesn't pay off, in the contrary, this curve is below the one for "30×4" in that range. So doubling the number of cores slows down execution, here.
2) A second range (between 10^4 and $2 \cdot 10^4$ elements for the "60×4" curve) is restricted by the available bandwidth, in this case the total bandwidth of the network switch. The performance is basically similar to that of "30×4", the number of nodes that is used doesn't influence much.
3) Then, in a third range above $2 \cdot 10^4$ elements the

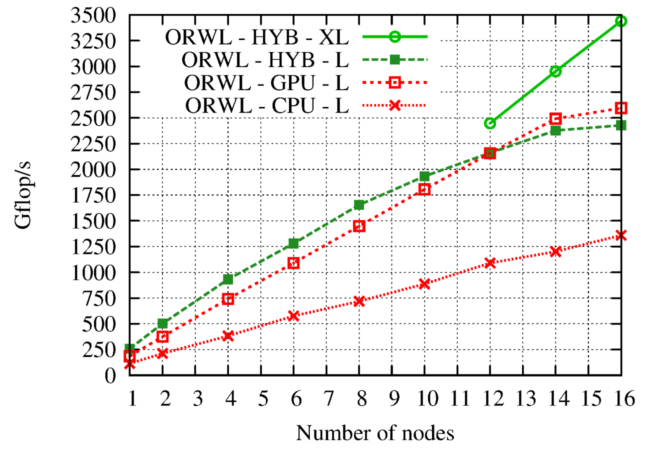


Figure 10. Gflop/s achieved on an increasing size problem: maximum size supported by the global amount of GPU memory of all nodes, using both CPU and GPU of each *Cameron* node

parallelism of the cores and the efficiency of the compute kernel kicks in. Each core has sufficiently large data to spend most of the time in computation. We approach a peak performance of approximately 1 Tflop/s.

C. Hybrid computing

Many computing nodes in modern parallel architectures are hybrid nodes, both with CPU cores and hardware accelerators (like GPU cores). Our cluster *Cameron* realizes such an architecture. The total computing power of all CPU cores of each node of our cluster is not negligible, compared to the computing power of the GPU of each node. Our benchmark reaches 180 Gflop/s on one GPU (using CUBLAS kernel), and 12 Gflop/s on each CPU core (using ATLAS kernel). With 6 CPU cores we have a potential close to 72 Gflop/s on the CPU, and a combined total for CPU and GPU close to 252 Gflop/s per node. So the maximum potential is 4 Tflop/s when combining the 16 nodes of the cluster.

Due to its modularity, the ORWL code has been easily extended to implement a hybrid matrix product; we just added static load balancing between the GPU and each CPU core. Then we conducted some experiments to determine the best number of ORWL tasks to run on our 6-core CPU nodes, and the best load balancing.

We have first benchmarked our hybrid matrix product on the same problem sizes we used on GPU only (see Section VI-B and Figure 6): the maximal problem size that can be stored on GPU memory of the used nodes. Our experiment has shown the most efficient solution was to run one ORWL task to control GPU computing and one ORWL task per CPU core: $1 + 6$ ORWL tasks per node. The best load balancing appears to depend on the number of used nodes and the problem size, so dynamic load balancing could be envisioned in the future to ease the parametrization of the execution. The best performances that were achieved by our hybrid implementation are shown in Figure 10: from 1 up to 10 nodes using both CPU and GPU cores leads to significantly better performances (see the green curves starting at 1 node). Beyond 12 nodes using GPU cores only is more efficient for the chosen problem size, that do not contains enough computations to efficiently use both CPU and GPU cores on more than 12 nodes.

However, when processing a very large problem ("XL" curve), even larger than could be stored in the GPU memory, the number of Gflop/s still increases. The second green curve (in the top right of Figure 10) reaches a peak performance of 3.44 Tflop/s on 16 nodes, about 800 Gflop/s (30 %) more than we achieved by using only the GPU processors and memory for computation.

D. Results synthesis

Our experimentation campaign has shown our ORWL matrix product application has achieved very good weak scalability, with similar performances to the reference MPI+OpenMP+CUDA implementation. But we have poor strong scalability, compared to the reference code. We need to improve our current implementation of ORWL for small problem size. Finally, our approach of hybrid architecture programming has been successful. However, a dynamic load balancing mechanism has to be designed and implemented, in order to avoid long tuning of each run.

VII. Conclusion and Future Work

We presented a computational model for parallel and distributed algorithm design (ORWL) that is centered around an abstract notion of *resources* that are meant to represent key concepts in the design, implementation and execution of an algorithm. They can represent blocks of data (input, intermediate or output), software (e.g a specialized function) or hardware (CPU, GPU, network card) entities. Resource access and sharing is granted in *critical sections* through simple FIFO strategies with read-write semantics.

A support environment and library for this model has been implemented that is compatible with known low level parallelization tools (OpenMP, CUDA) and allows to reuse existing optimized kernels like ATLAS or cuBLAS. A sequence of benchmarks with a block cyclic matrix multiplication has been presented. It shows very good scaling properties that are comparable to that of an optimized parallel reference implementation. The advantage of the ORWL implementation appears in particular when we use all available processing units (CPU and GPU) on a cluster equipped with heterogeneous hardware: without changing the implementation of the algorithm we are easily able to draw all available power from such a platform.

In order to reach our goal to make ORWL a new easy-to-use and efficient programming model and toolbox for parallel developers, we aim to experiment ORWL on parallel applications requiring more complex communication schemes. Simultaneously we will continue to investigate hybrid computing on hierarchical architectures including hardware accelerators. The unmodified benchmark used in this paper already shows promising results on Intel's Xeon Phi platform, and we are excited with the prospect to run it in on a cluster equipped with such accelerators in the weeks to come.

As we succeeded to use both CPU and GPU cores with a fine tuned static load balancing, the next step will consist in developing a dynamic load balancing mechanism, without increasing data transfers between CPU and GPU memories. We plan to use different task queues that group tasks that require common data. They will be pre-allocated to CPU or GPU, and support work stealing from processors having exhausted their task queues. Task queues will be modeled as computing *resources* that will be accessed across critical sections by ORWL.

Acknowledgment

Authors want to thank LORRAINE REGION for the constant support of SUPELEC research, and especially for the help to deploy a 10-Gbit/s Ethernet interconnection network inside the hybrid computing cluster used in this work.

This work has been partially supported by the INRIA exploratory action MultiCore, and some experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] Austin Group, "Threads," in *The Open Group Base Specifications*, A. Josey *et al.*, Eds. The Open Group, 2013, vol. Issue 7, p. chapter 2.9. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009695399/
[2] "OpenMP multi-threaded programming API." [Online]. Available: http://www.openmp.org
[3] "Threading building blocks," Intel Corp. [Online]. Available: http://software.intel.com/en-us/intel-tbb
[4] *NVIDIA CUDA C Programming Guide 4.0*, NVIDIA, June 2011. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
[5] Khronos OpenCL Working Group, *OpenCL Specification*, Khronos Group, 2012, version 1.2. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
[6] "The OpenACC^{TM} Application Programming Interface," OpenACC-Standard.org, Tech. Rep., Apr. 2013, version 2.0, draft 5. [Online]. Available: http://openacc.org/sites/default/files/OpenACC-2.0-draft.pdf
[7] "Message passing interface." [Online]. Available: http://www.mpi-forum.org/docs
[8] C. Lommont, *Introduction to Intel® Advanced Vector Extensions*, Intel Corp., 2012. [Online]. Available: http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/
[9] S. Peyton Jones, A. Gordon, and S. Finne, "Concurrent haskell," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '96. New York, NY, USA: ACM, 1996, pp. 295–308. [Online]. Available: http://doi.acm.org/10.1145/237721.237794
[10] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[11] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, A. J. Smith, Ed. ACM, 1993, pp. 289–300.
[12] R. Lucchi, M. Millot, and C. Elfers, "Resource Oriented Architecture and REST," European Commission Joint Research Centre, Tech. Rep. EUR 23397, 2008.
[13] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1989.
[14] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492
[15] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis, "Accelerating hydrocodes with OpenACC, OpenCL and CUDA," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, 2012, pp. 465–471.
[16] V. D. Fabien Le Mentec, Thierry Gautier, "The X-Kaapi's application programming interface. Part I: Data flow programming," INRIA, Inovallée, 655 avenue de l'Europe Montbonnot, 38334 Saint Ismier Cedex, Technical Report 0418, November 2011.
[17] JTC1/SC22/WG14, Ed., *Programming languages - C*, cor. 1:2012 ed. ISO, 2011, no. ISO/IEC 9899.
[18] M. Aswad, P. W. Trinder, and H.-W. Loidl, "Architecture aware parallel programming in Glasgow Parallel Haskell (GPH)," in *ICCS*, ser. Procedia Computer Science, H. H. Ali, Y. Shi, D. Khazanchi, M. Lees, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds. Elsevier, pp. 1807–1816.
[19] F. Mueller and Y. Zhang, "HiDP: A hierarchical data parallel language," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494994
[20] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442
[21] W. Kirschenmann, L. Plagne, and S. Vialle, "Multi-Target Vectorization with MTPS C++ Generic Library," in *PARA 2010 - 10th International Conference on Applied Parallel and Scientific Computing*, ser. Lecture Notes in Computer Science, K. Jónasson, Ed., vol. 7134. Reykjavík, Iceland: Springer Berlin / Heidelberg, 2012, pp. 336–346. [Online]. Available: http://hal.inria.fr/hal-00685159
[22] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188543
[23] P.-N. Clauss and J. Gustedt, "Iterative Computations with Ordered Read-Write Locks," *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 496–504, 2010. [Online]. Available: http://hal.inria.fr/inria-00330024/en
[24] J. Gustedt and E. Jeanvoine, "Relaxed Synchronization with Ordered Read-Write Locks," in *Euro-Par 2011: Parallel Processing Workshops*, ser. LNCS, M. Alexander *et al.*, Eds., vol. 7155. Springer, May 2012, pp. 387–397. [Online]. Available: http://hal.inria.fr/hal-00639289