An abstract object oriented runtime system for heterogeneous parallel architecture

Jean-Marc Gratien

▶ **To cite this version:**

Jean-Marc Gratien. An abstract object oriented runtime system for heterogeneous parallel architecture. 2013. hal-00788293

HAL Id: hal-00788293
https://ifp.hal.science/hal-00788293

Preprint submitted on 14 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An abstract object oriented runtime system for heterogeneous parallel architecture

Jean-Marc GRATIEN
Applied mathematics and computer sciences department
IFPEN
Rueil-Malmaison, France
jean-marc.gratien@ifpen.fr

***Abstract*—In our paper we present an abstract object oriented runtime system that helps to develop scientific application for new hererogenous architecture based on multi-node of multi-core processors enhanced with accelerator boards. Its architecture based on abstract concepts enables to follow hardware technology by extending them with new implementations modeling new hardware components, while limiting the impacts on existing application architecture or in the developpement of high level generative framework based on Domain specific language. We validate our approach with a multiscale algorithm to solve partial derivative equations that we have implemented with this runtime system and benchmarked on various heterogeneous hardware architecture.**

***Keywords*-High performant computing; Runtime system; GPGPU; Multi-core; Heterogeneous architecture;**

I. Introduction

The trend in hardware technology is to provide hierachical architecture with different levels of memory, process units and connexion beween ressources, using either accelerating boards or by the means of hybrid heterogeneous manycore processors. The complexity to handle such architecture has considerably increased. The heterogeneity introduces serious chalenges in term of memory coherency, data transfer between local memories, load balancing between computation units. Various approaches have appeared to manage the different levels of parallelism : different programming models, programming environnements, schedulers, data management solutions and runtime systems like Charm++[3], StarSS[1], StarPU[2] or XKaapi[4] that provide higher-level software layers with convenient abstractions which permit to design portable algorithms without having to deal with low-level concerns.

In scientific computing, new methods have emerged, like multiscale methods to solve partial derivative equations. These methods, when they are based on algorithms providing a great amount of independant computations, are good candidates to perform on new hardware technology. However, using often complex numerical concepts, they are developped by programmers that cannot deal with hardware complexity. Most of the existing programming approaches remain often too poor to manage the different levels of parallism. Runtime system solutions that expose convenient and portable abstraction to a high-level compiling environnements and to highly optimized libraries are interesting as they enable end users to develop complex numerical algorithms hiding the low level concerns of data management and task scheduling. Such layer provides a unified view of all processing units, enables various parallelism models with an expressive interface that bridges the gap between hardware stack and software stack.

In our paper we propose an abstract object oriented runtime system that enables to handle the variety of new hybrid architectures and to follow the fast evolution of hardware design. Its architecture is based on the abstract concepts like *Tasks*, *Data management*, *Scheduler* and *Executing driver* that enables various extentions by the mean of new implementations modeling new hardware components. We mean by abstract the fact that the concepts of our runtime system are defined as requirements on the `C++` types that represent them. The purpose is to allow developper to write programs with abstract types independantly of the underlying objects implementation. This solution has the advantage to limit the impact of the choice of the runtime system implementation on the application architectures, to clearly separate application evolution from hardware one. Finally, in the contrary of some existing runtime system solutions, it enable to enhance specific part of existing applications without needing to restructure the all application architecture and to re-write from scratch often complex algorithms. We validate our approach on a multiscale methods to solve partial derivative equations : we have developped them with our runtime system model and various implementations of its abstract concepts and benchmarked on various heterogeneous archictectures with multi SMP nodes, multi-core processors and with multi accelerated boards.

II. An abstract object oriented Run-Time System Model

A. Contribution

In order to enable scientific developpers to implement their methods in a transparent way, we propose a runtime system layer on top of which they can write source code that performs efficiently on new heterogeneous hardware architectures. Our approach is to provide an abstract object oriented runtime system model that enables developpers to handle, in a unified way, different levels of parallelism and

different grain sizes. Like for most existing Runtime System frameworks, the proposed model is based on:

- an abstract architecture model that enables us to describe in a unified way most of nowadays and future heterogeneous architectures with static and runtime information on the memory, network and computational units;
- an unified parallel model programing based on tasks that enables us to implement parallel algorithms for different architectures;
- an abstract data management model to describe the processed data, its placement in the different memory and the different way to access to it from the different computation units.

The main contribution with respect to existing frameworks is to propose an abstract archictecture for the model based on **abstract concepts** , where we define **Concept** as set of requirements for types of objects that implement specific behaviours. Most of the abstractions of our Runtime system models are defined as requirements for `C++` structures. Algorithms are then written with some abstract types which must conform to the concepts they implement. This approach has several advantages:

1) it enables to clearly separate the implementation of the numerical layer from the implementation of the RunTimeSystem layer;
2) it enables to take into account the evolution hardware architecture with new extensions, new concepts implementation, limiting in that way the impact on the numerical layer;
3) it enables the benchmark of competiting implementations of each concept with various technologies, which can be based on existing research frameworks like StarPU which already provides advanced implementation of our concepts;
4) it enables us to design a non intrusive library, which unlike most of existing framewok, does not constraint the architecture of the final applications. One can thus enhance any part of any existing applications with our framework, re-using existing classes or functions without needing to migrate the whole application architecture to our formalism. This issue is very important because often legacy codes cannot take advantage of new hybrid hardware because most of existing programming environnements make the migration of such applications painful;
5) finaly the proposed solution does not need any specific compiler tools and does not have any impact on the project compiling tool chain.

In this section we present the different abstractions on which the proposed framework relies. We detail the concepts we have proposed to modelize these abstractions. We illustrate them by proposing different types of implementation with various technolgies. We study how the proposed solution enables us to address seamlessly heterogeneous architectures and to manage the available computation ressources to optimize the application performance.

B. An abstract hybrid architecture model

The purpose of this abstract hybrid architecture model is to provide a unified way to describe hybrid hardware architecture and to specify the important features that enable to choose at compile time or at run time the best strategies to ensure performance. Such an architecture model has been already developped in the project **HWLOC** (Portable Hardware Locality)[5] which provides a portable abstraction (across OS, versions, architectures, ...) of the hierarchical topology of modern architectures, including NUMA memory nodes, sockets, shared caches, cores and simultaneous multithreading. It also gathers various system attributes such as cache and memory information as well as the locality of I/O devices such as network interfaces, InfiniBand HCAs or GPUs. We propose an architecture model, based on the HWLOC framework. An architecture description is divided into two part, a static part grouping static information which can be used at compile-time and a dynamic part with dynamic information used at run-time. The information is modelized with abstractions like `system` representing the whole hardware system, `machine` for a set of processors and memory with cache coherency, `node` modeling a NUMA node, a set of processors around the memory on which the processors can directly access, `cache` for cache memory (L1i, L1d, L2, L3,...), `core` for computation units or `pu` for processing unit ...The **static information** is represented by tag structures and string keys. They are organized in a tree structure where each node has a tag representing a hardware component, a reference to a parent node and a list of children nodes. Tag structures are used in the generative framework at compile time. For a target hardware architecture and with its static description, it is possible to generate the appropriate algorithms with the right optimisations. The **dynamic information** is stored, in each node of the tree description, with a property map associating keys representing dynamic hardware attibutes, to values which are evaluated at runtime, possibly using the `HWLOC` library. Theses values form the runtime information which enables to instantiate algoritms with dynamic optimization parameters like cache memory sizes, `stack_size` the size of the memory stack, `nb_pu` the maximum number of Process Units, `warp_size` the size of a NVidia WARP (NVidia group of synchronized threads) and `max_thread_block_size` the maximum size of a NVidia thread block executed on GP-GPUs, `nb_core` or `nb_gpu` the number of available physical cores of CPUs or GPUs,...Such runtime features, are useful parameters to optimize low level algorithms, in particularly for CUDA or OpenCL algorithms for GP-GPUs.

C. An abstract unified parallel programming model

We propose an abstract unified parallel programming model based on the main following abstractions:

- a **task abstraction** representing pieces of work, or an algorithm that can be executed on a core or onto accelerators asynchronously. A task can have various implementations that can be executed more or less efficiently on various computational units. Each implementation can be written in various low level languages (C++, CUDA, OpenCL) with various libraries (BLAS, CUBLAS) and various compilation optimizations (SSE directives,...). Tasks can be independent or organized in direct acyclic graphs which represent algorithms.
- a **data abstraction** representing the data processed by tasks. Data can be shared between tasks and have multiple representations in each local memory device.
- a **scheduler abstraction** representing objects that walk along task graphs and dispatch the tasks between available computational units.

These abstractions are modelized with `C++` concepts (defined in §II-A). This approach enables to write abstract algorithms with abstract objects with specific behaviours. Behaviours can be implemented with various technologies more or less efficient with respect to the hardware on which the application is executed. The choice of the implementation can be done at compile time for a specic hardware architecture, or at runtime for general multi-platform application.

A particular attention has been paid in the design of the architecture to have a non intrusive solution in order to facilitate the migration of legacy code, to enable the reusability of existing classes or functions and to limit the impacts on the existing application architecture. The purpose is to be able to select specific parts of an existing code, for example some parts which a great amount of independant works, then to enhance them by introducing multi-core or gpu optimisation without having to modify the whole of the code.

1) Runtime System Architecture: The proposed Runtime System architecture, illustrated in figure 1 is quite standard:

- Computation algorithms implemented by user free functions or classes are encapsulated in *Tasks* objects, managed by a centralized *task manager* ;
- The pieces of data processed by the task objects, represented by user data classes are encapsulated in *data handler* objects, managed by a centralized *data manager* ;
- The associations between tasks and the processed data handlers are managed by *DataArg* objects;

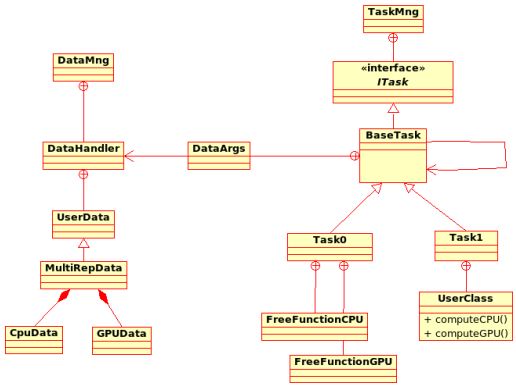


Figure 1. Runtime system architecture

Executing model

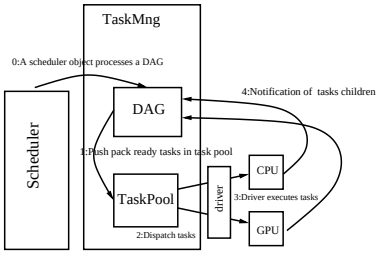


Figure 2. Executing model

- Tasks are organized in DAGs and processed by *scheduler* objects that dispatch them on devices to run them with executing *drivers*.

The executing model is illustrated in figure 2:

- A *Scheduler* object processes a DAG of tasks belonging to a centralized task manger;
- *Task* objects which are ready to be executed are pushed back in a *task pool*;
- The *scheduler* object dispatches ready tasks on available computation devices, with respect to a given strategy;
- *Tasks* objects are executed on a target device by a *driver* object, then they are notified once their execution is finished;
- A *DAG* is completely processed once the task pool is empty.

2) Task management: The task management of our Runtime System Model is modelized with the class `TaskMng` described in listing 1. The sub type `TaskMng::ITask` is an interface class specifying the requirements for task implementation. `TaskMng::ITask` pointers are registered in a TaskMng object that associates them to an unique integer identifier `uid`. Tasks are managed in a centralized collection of tasks and dependencies between tasks are created with their `uid`. The base class `TaskMng::BaseTask` in listing 3 refines the

`TaskMng::ITask` interface to manage a collection of uids of children tasks depending of the current task. Thus a **Directed Acyclic Graph** (DAG) (figure 3) is represented by a root task, and walking along it then consists in iterating recursively on each task and on its children.

Listing 1. TaskMng class

```
class TaskMng {
public :
  typedef int uid_type ;
  static const int undefined_uid = -1 ;
  class ITask ;
  TaskMng(){}
  virtual ~TaskMng(){}
  int addNew(ITask* task) ;
  void clear() ;
  template<typename SchedulerT>
  void run(SchedulerT& scheduler, std::vector< int > const& task_list) ;
};
```

Listing 2. Task class interface

```
class TaskMng::ITask
{
public :
  ITask() : m_uid(TaskMng::undefined_uid){}
  virtual ~ITask() {}
  uid_type getUid() const {
    return m_uid ;
  }
  virtual void compute(TargetType& type,TaskPoolType& queue) = 0 ;
  virtual void compute(TargetType& type) = 0 ;
protected :
  uid_type            m_uid ;
} ;
```

Listing 3. Task class interface

```
class TaskMng::BaseTask : public TaskMng::ITask
{
public :
  BaseTask() ;
  virtual ~BaseTask() ;
  void addChild(ITask* child) ;
  void clearChildren() ;
  void notifyChildren(TaskPoolType& queue) ;
  void notify() ;
  bool isReady() const ;
};
```

The *Task* concept enables to implement a piece of algorithm for different kinds of target devices. A specific type of target device, or computational unit is identified by a unique *Target* label. Task instances are managed by a *TaskMng* that associates them to an unique id that can be used to create dependencies between tasks. Each task manages a list of children tasks. *Directed Acyclic Graphs* (DAGs) can be created with task dependencies. They have one root task. Task dependencies are managed by task unique id. To ensure graphs to be acyclic, tasks can only be dependent on an existing task with a lower unique id. A task can have various implementations. They are associated to a *Target* attribute representing the type of computational unit on which they should be used.

3) Data management: Our runtime system model is based on a centralized data management layer aimed to deal with:

- the data migration between heterogeneous memory units;
- an efficient data coherency management to optimize data transfer between remote memory and local memory;

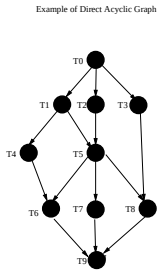


Figure 3. Example of directed acyclic graph

- the concurrency of tasks accessing to shared data.

Our data management is based on the `DataMng` and `DataHandler` classes (listing 9). `DataHandler` objects represent pieces of data processed by tasks. They are managed by a `DataMng` object which has a `create` member function to instanciate them. `DataHandler` objects have a unique `DataUidType` identifier `uid`. The `DataArgs` class is a collection of `std::pair<DataHandler::uid_type,eAccessMode>` where `AccessMode` is an `enum` type with the following values `W`, `R` or `RW`. The `DataHandler` class provides a `lock`, `unlock` service to prevent data access concurrency:

- a task can be executed only if all its associated data handlers are unlocked ;
- when a task is executed, the DataHandlers associated with a `W` or `RW` mode are locked during execution and unlocked after.

A piece of data can have multiple representations on each device local memory. The coherency of all representations is managed with a timestamp DataHandler service. When a piece of data is modified, the timestamp is incremented. A representation is valid only if its timestamp is up to date. When a task is executed on a specific target device, the local data representation is updated only if needed, thus avoiding unuseful data transfer between different local memories.

4) Task dependencies: Task dependencies can be created in three ways:

- **Explicit task dependencies** is based on task uids. The `addChild` member function enables to create dependencies between tasks. Only a task with a lower uid can be the parent of another one, thus ensuring that the created graph is acyclic;
- **Logical tag dependencies**, based on task tags create dependencies between a group of tasks with a specific tag and another group of tasks with another specific tag;
- **Implicit data driven dependencies** is based on the sequential consistency of the DAG building order. When a task is registered, if the DataHandler access is in:
  - `RW` or `W` mode, then the task implicitly depends on all tasks with a lower uid accessing that same DataHandler in `R` or `RW` mode,

– `R` or `RW` mode, then the task implicitly depends on the last task accessing that data in `RW` or `W` mode.

Once a task is executed, all its children tasks are notified. Each task manages a counter representing the number of parent tasks. When a task is notified, this counter is decremented . A task is ready when its parent counter is equal to zero and when all its dependent data handlers are unlocked. Its uid is then put in the queue of ready tasks managed by the scheduler that processes the DAG.

5) Scheduling and executing model: On heterogeneous architectures, the parallelism is based on the distribution of tasks on available computation units. The performance of the global execution depends a lot on the stategy used to launch independent tasks. It is well known that there is not a unique nor a best scheduling policy. The performance depends on both the algorithm and the hardware architecture. To implement various scheduling solutions adapted to different algorithms and types of architecture, we propose a `Scheduler` concept defining the set of requirements for scheduler types to represent scheduling models. The purpose of objects of such a type is to walk along task DAGs, to select and execute independent tasks on the available computation units, with respect to a given strategy. The principles for a scheduler object are:

1) to manage a pool of ready tasks (tasks which all parent tasks are finished and all datahandlers of its `DataArgs` attribut are unlocked);
2) to distribute the ready tasks on the different available computation units following a given scheduling stategy;
3) to notify the children tasks of a task once the task execution is finished;
4) to push back tasks that get ready in the pool of ready tasks.

The **TaskPoolConcept** defines the behaviour that must implement a type representing a `TaskPool`, that is to say the possibility to push back new ready tasks and to grab tasks to execute.

Listing 4. TaskPool

```
class TaskPoolConcept
{
public:
  template<typename TaskT>
  void pushBack(TaskT::uid_type uid) ;

  template<typename TaskT,typename TargetT>
  typename Task::ptr_type grabNewTask(TargetT const& target) ;

  bool isEmpty() const ;
};
```

A coarse grain parallelism strategy consists in executing the different independent ready tasks in parallel on the available computation units. We have implemented various schedulers like the `StdScheduler`, the `TBBScheduler` and `PoolThreadScheduler` described in §II-C7

Parallelism can be managed at a finer grain size with concepts like the **ForkJoin** and the **Pipeline** concepts.

ForkJoin: On multi-core architecture, a collection of equivalent tasks can be executed efficiently with technolgies like TBB, OpenMP, Posix threads. The ForkJoin concept (figure 4) consists in creating a DAG macro task node which holds a collection of tasks. When this node is ready, the collection of nodes is processed by a `ForkJoin Driver` in parallel. The macro task node is finished when all its children tasks are finished. The `ForkJoin Driver` is a concept defining the requirement for the types of objects that implement the fork-join behaviour with different technologies or libraries like TBB, Boost.Thread or Pthread.

Listing 5. Fork-Join driver concept

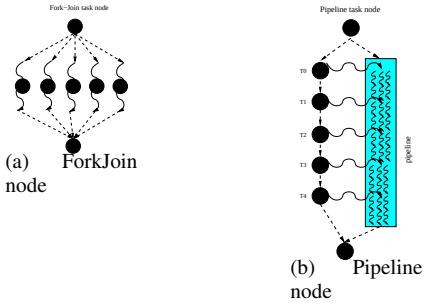
```
class ForkJoinDriverConcept
{
public:
  template<typename TaskT,typename TargetT, typename QueueT>
  void execForkJoin(std::vector< typename TaskPtrT::ptr_type > const& tasks,
                    std::vector< typename TaskPtrT::uid_type > const& uids,
                    TargetT& target,
                    QueueT& queue) ;

  template<typename TaskT,typename TargetT>
  void execForkJoin(std::vector< typename TaskPtrT::ptr_type > const& tasks,
                    std::vector< typename TaskPtrT::uid_type > const& uids,
                    TargetT& target) ;
};
```

Pipeline: On vectorial device or accelerator boards, the **Pipeline** concept (figure 4) consists in executing a sequence of tasks (each task depending on its previous one) with a specific internal structure of instructions. The `Pipeline Driver` is a concept defining the requirement for the types of objects implementing the pipeline behaviour. These objects are aware of the internal structure of the tasks and execute them on the computation device in a optimized way often with a thin grain size parallelism. This approach is interesting for new GPU hardwares which can execute concurrent kernels. It enables to implement optimized algorithms with streams and asynchrone execution flows that improve the occupancy of device ressources and lead then to better performance. For instance, for the computation of the basis functions of the multiscale model, we illustrate in §III how the flow of linear system resolutions can be executed efficiently on GPU device with the GPUAlgebraFramework layer.

Listing 6. pipeline driver concept

```
class ForkJoinDriverConcept
{
public:
  template<typename TaskT,typename TargetT, typename QueueT>
  void execPipeline(std::vector< typename TaskPtrT::ptr_type > const& tasks,
                    std::vector< typename TaskPtrT::uid_type > const& uids,
                    TargetT& target,
                    QueueT& queue) ;

  template<typename TaskT,typename TargetT>
  void execPipeline(std::vector< typename TaskPtrT::ptr_type > const& tasks,
                    std::vector< typename TaskPtrT::uid_type > const& uids,
                    TargetT& target) ;
};
```

Asynchronism management: On an architecture with heterogeneous memories and computation units, it is important to provide enough work to all available computation units and to reduce the latency due to the cost of data transfer between memories. The Asynchronism mechanism is a key element for such issues. The classes `class AsynchTask`



(a) ForkJoin node (b) Pipeline node

Figure 4. ForkJoin and pipeline task node

and `class Wait` parametrized by the types `TaskT` and `DriverT` implement the asynchronous behaviour:

- the `AsynchTask<DriverT,TaskT>` is a task node that executes asynchronously its child task of type `TaskT`;
- the `Wait<AsynchTaskT>` is a task node that waits for the end of the execution the child task of the previous node then notifies the children of this task.

The `Driver` concept specifies the requirement of the type of objects that implement the asynchronous behaviour. This behaviour can be easily implemented with threads. The child task is executed in a thread. The end of the execution corresponds to the end of the thread. For GPU device, this behaviour can be implemented using a stream on which is executed an asynchrounous kernel. The `wait` function is implemented with a synchronisation on the device.

The asynchronous mechanism is interesting to implement data prefetching on device with remote memory. Prefetch task nodes can be inserted in the DAG to load asynchronously data on GPU device so that they are available when the computational task is ready to run.

Listing 7. Data management

```
template<typename DriverT,typename TaskT>
class AsynchTask : public TaskMng::BaseTask
{
public:
  typedef TaskMng::BaseTask   BaseType ;
  AsynchTask(DriverT& driver,TaskT& task) ;
  virtual ~AsynchTask() ;
  virtual void wait(TargetType& type,TaskPoolType& queue) ;
  virtual void notify() ;
  virtual bool isReady() const ;
  void compute(TargetType& type) ;
  void compute(TargetType& type,TaskPoolType& queue) ;
  void finalize(TargetType& type,TaskPoolType& queue) ;
private :
  TaskT& m_task ;
};

template<typename AsynchTaskT>
class Wait : public TaskMng::BaseTask
{
public :
  typedef TaskMng::BaseTask   BaseType ;
  Wait(AsynchTaskT& parent) ;
  virtual ~Wait() ;
  void compute(TargetType& type,TaskPoolType& queue) ;
  void compute(TargetType& type) ;
  void finalize(TargetType& type,TaskPoolType& queue) ;
private :
  AsynchTaskT& m_parent ;
} ;
```

6) Example of application of the runtime system: With our runtime system abstractions, listing 8 illustrates how to write a simple program adding two vectors, which can be executed on various devices.

Listing 8. Simple vector addition program

```
class AxpyTask {
   void computeCPU(Args const& args) {
     auto x const& args.get<VectorType>('x').impl<tag::cpu>();
     auto y& args.get<VectorType>('y').impl<tag::cpu>();
     SAXPY(x.size(), 1.0, x.dataPtr(), 1, y.dataPtr(), 1);
   }
   void computeGPU(Args const& args) {
     auto x const& args.get<vector_type>('x').impl<tag::gpu>();
     auto y& args.get<vector_type>('y').impl<tag::gpu>();
     cublasSaxpy(x.size(), 1.0, x.dataPtr(), 1, y.dataPtr(), 1);
     cudaThreadSynchronize();
   }
} ;

int main(int argc, char **argv) {
   float vec_x[N], vec_y[N];

   /*(...)*/
   //
   // DATA MANAGEMENT SET UP
   DataMng data_mng ;
   VectorType x ;
   VectorType y ;
   DataHandler* x_handler = data_mng.create<VectorType>(&x) ;
   DataHandler* y_handler = data_mng.create<VectorType>(&y) ;

   //
   // TASK MANAGEMENT SET UP
   TaskMng task_mng ;
   /*(...)*/
   AxpyTask op ;
   TaskMng::Task<AxpyTask>* task = new TaskMng::Task<AxpyTask>(op) ;
   task->set<tag::cpu>(&AxpyTask::computeCPU) ;
   task->set<tag::gpu>(&AxpyTask::computeGPU) ;
   task->args().add('x',x_handler,ArgType::mode::R) ;
   task->args().add('y',y_handler,ArgType::mode::RW) ;

   int uid = task_mng.addNew(task) ;
   task_list.push_back(uid) ;

   //
   // EXECUTION
   SchedulerType scheduler ;
   task_mng.run(scheduler,task_list) ;
}
```

7) Elements of implementation of different concepts:

Data and task management concepts: The implementation of data and task management is based on the following principles:

- User Data are implemented by the mean of user `C++` classes or structures;
- User algorithms are implemented by the means of user free functions or member functions of user `C++` classes.

We have implemented `DataHandler` as a class that encapsulate any user classes or structures and which provides functions to retrieve the original user data structure, to `lock` or `unlock` the user data.

The `DataMng` is a centralized class that manages a collection of `DataHandler` objects and their integer unique identifier. This class enables to access any user data by the means of its unique identifier.

Listing 9. Data management

```
typedef enum {R,W,RW,Undefined} eAccessModeType ;

class DataHandler
{
public:
  typedef int      uid_type ;
  static const int null_uid = -1 ;
  DataHandler(uid_type uid=null_uid) ;
  virtual ~DataHandler() ;
```

```
    uid_type getUid() const ;
    template<typename DataT>
    DataT* get() const ;
    void lock() ;
    void unlock() ;
    bool isLocked() const ;
  };

  class DataMng
  {
  public :
    typedef DataHandler* DataHandlerPtrType ;
    DataMng() ;
    virtual ~DataMng();
    template<typename DataT>
    DataHandler* create() ;
    DataHandler* getData(int uid) const ;
  };
```

Tasks are implemented with the classes `TaskMng::Task0` and `class TaskMng::Task` (listing 10) that encapsulate any user free function or user class and member function, stored in a `boost::function` attribute. They implement the `TaskMng::ITask` interface that enables any scheduler to execute task objects on any target computation unit. They have a `set(<target>,<function>)` member function to define the implementation of the task for each target device.

Listing 10. Task class inplementation

```
class TaskMng::Task0 : public TaskMng::BaseTask
{
public :
  typedef ITask::TargetType                    TargetType ;
  typedef boost::function1<void,
                           DataArgs const&>    FuncType ;
  typedef std::map<TargetType,FuncType>        FuncMapType ;
  typedef typename FuncMapType::iterator       FuncIterType ;
  Task0() ;
  virtual ~Task0() ;
  void set(TargetType type,FuncType func) ;
  void compute(TargetType& type,TaskPoolType& queue) ;
  void compute(TargetType& type) ;
} ;

template<typename ComputerT>
class TaskMng::Task : public TaskMng::BaseTask
{
public :
  typedef ITask::TargetType                    TargetType ;
  typedef boost::function2<void,
                           ComputerT*,
                           DataArgs const&>    FuncType ;
  typedef std::map<TargetType,FuncType>        FuncMapType ;
  typedef typename FuncMapType::iterator       FuncIterType ;
  Task(ComputerT* computer) ;
  virtual ~Task() ;
  void set(TargetType type,FuncType func) ;
  void compute(TargetType& type,TaskPoolType& queue) ;
  void compute(TargetType& type) ;
} ;
```

Task execution: When a task is executed, data user structures are recovered with a `DataArgs` object that stores data handlers and their access mode. This data can be locked if it is accessed in a write mode when the user algorithm is applied to it. Modified data is unlocked at the end of the algorithm execution.

TaskPool concept: We have implemented the `TaskPoolConcept` with a simple parametrized class `template<TaskMng> class TaskPool` with two attributes: `m_uids` a collection of task uid and `m_mng` a reference to the task manager. The member function `pushBack(TaskMng::ITask::uid_type uid)` feeds the collection of ready tasks. The `Task::ptr_type grabNewTask(<target>)` grabs a `uid` from `m_uids` and returns the corresponding task with `m_mng`.

Scheduler concept: To implement a scheduler class, one has to implement the `exec(<tasks>,<list>)` function that gives access to a collection of tasks and a list of tasks, roots of different DAGs. Walking along these DAGs, the scheduler manages a pool of ready tasks: the scheduler grabs new tasks to execute, children tasks are notified at the end of execution and feed the task pool when they are ready. Some `Driver` objects can be used to execute tasks on specific devices, to modelize different parallel behaviours, to give access for example to a pool of threads that grab tasks to be executed in the pool of ready tasks. We have implemented the following scheduler types:

- the `StdScheduler` is a simple sequential scheduler executing the tasks of a TaskPool on a given target device;
- the `TBBScheduler` is a parallel scheduler for multi-core architecture implemented with the `parallel_do` functionnality of the TBB library;
- the `PoolThreadScheduler` is a parallel scheduler based on a pool of threads dispatched on several cores of the multi-core nodes, implemented with the `Boost.Thread` library. Each thread is associated to a physical core with an affinity, and dedicated to executed tasks on this specific core or on an accelerator device. The scheduler dispatches the tasks of the task pool on the threads which are starving.

ForkJoinDriver implementation: We have developped for multi-core architectures three implementations conforming to this concept:

- the `TBBDriver` is a multi-thread implementation using the `parallel_for` algorithm of the TBB library;
- the `BTHDriver` is a multi-thread implementation based on a pool of threads implemented with the `Boost.Thread` library;
- the `PTHDriver` is a multi-thread implementation based on a pool of threads written with the native posix thread library.

III. Application to multiscale basis functions construction

We have validated the RunTime System Model presented in §II implementing the basis function computation of a multiscale method. In [9], an interesting overview of such methods is done by by Kippe V., Aarnes J. E. and Lie K. A. Most of these methods are based on the computation of independant basis functions defined by partial derivated equations which leads to solve independant linear systems. The objectif here is to propose a generic way to implement these computations for various hardware configurations and various implementations of the runtime system using

muti-thread technology with TBB[6], Boost.Thread[7] or Posix.Thread library for multi-core platform or with the GPUAlgebraFramework layer, a library written with Cuda or OpenCL for node with GP-GPU accelerators aimed to perform efficiently on GP-GPUs, collections of small independant matrix-vector operations.

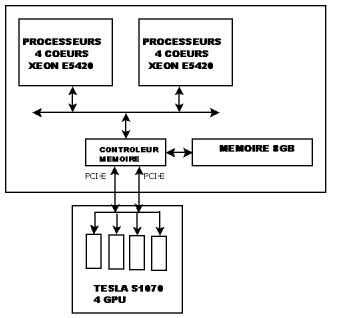
We have implemented a `BasisFunction` class with a standard implementation for CPU and a GPU implementation based on the GPUAlgebraFramework layer for GP-GPU devices. The algorithm to compute all the basis functions has been written as in listing 11 with the `Task`, `ForkJoin` and `Pipeline` concept, and with various fork-join driver implementations based on the TBB, Boost.Thread, pThread and with the pipeline driver based on the GPUAlgebraFramework library.

Listing 11. Basis computation algorithm

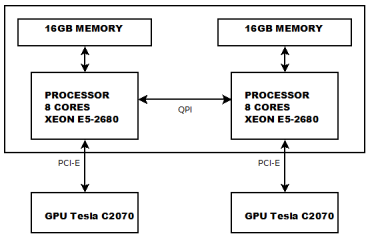
```
   template<typename SchedulerT,
            typename TaskMngT,
            typename ForkJoinDriverT,
            typename PipelineDriverT,
            typename DataMngT>
void computeBasis(std::vector<BasisFunction*>& basis)
{
  typedef typename TaskMngT::uid_type                          uid_type ;
  typedef typename TaskMngT::ForkJoinTask<ForkJoinDriverT> ForkJoinTask ;
  typedef typename TaskMngT::PipelineTask<PipelineDriverT> PipelineTask ;
  typedef typename TaskMngT::TaskNode                          TaskNode ;
  typedef typename TaskMngT::Task<Basis>                       TaskType ;

  //DATA MANAGEMENT
  DataMng data_mng ;
  DataHandlerType* solver_handler = data_mng.getNewData() ;
  solver_handler->set<ILinearSolver>(basis_solver) ;
  DataHandlerType* k_handler = data_mng.getNewData() ;
  k_handler->set<VariableCellReal const>(&k) ;

  //TASK MANAGEMENT
  std::vector< uid_type > dag ;
  TaskMng task_mng ;

  //DEFINE PARALLEL FORKJOIN FOR MULTICORE ARCHITECTURE
  ForkJoinDriverT forkjoin(/* ...*/) ;
  ForkJoinTask* forkjoin_task =
        new ForkJoinTask(forkjoin,task_mng.getTasks());
  uid_type fk_uid = m_task_mng.addNew(forkjoin_task) ;

  //DEFINE PIPELINE FOR GPU ARCHITECTURE
  PipelineDriverT pipeline(/* ...*/) ;
  PipelineTask* pipeline_task =
       new PipelineTask(pipeline,task_mng.getTasks()) ;
  uid_type pipeline_uid = m_task_mng.addNew(forkjoin_task) ;

  //DEFINE A DAG ROOT NODE WITH CPU AND GPU IMPL
  TaskNode* root = new TaskNode() ;
  root->set("cpu",fk_uid) ;
  root->set("gpu",pipeline_uid) ;
  uid_type root_uid = task_mng.addNew(root) ;

  //ADD ROOT TASK LIST AS A DAG ROOT NODE
  dag.push_back(root_uid) ;

  //DEFINE BASIS TASKS AND TASK DEPENDANCIES
  std::for_each(auto ibasis : basis)
  {
     TaskType* task = new TaskType(*ibasis) ;
     task->args().add("Solver",DataHandlerType::R,solver_handler) ;
     task->args().add("K",DataHandlerType::R,k_handler) ;
     typename TaskType::FuncType f_cpu = &BasisFunctionType::computeCPU ;
     task->set("cpu",f_cpu) ;
     task->set("gpu",f_gpu) ;
     Integer uid = m_task_mng.addNew(task) ;
     forkjoin_task->add(uid) ;
     pipeline_task->add(uid) ;
  }

  //EXECUTE THE DAG
  SchedulerT scheduler ;
  task_mng.run(scheduler,dag) ;
}
```

IV. Performance results

In this section we present some performance results of the basis functions computation of the multiscale method implemented with our RunTime System Model on a benchmark of the 2D version of the SPE10 study case inspired from the benchmark described in [8]. We compare different implementations and solutions run on various hardware configurations. We focus on the test case with a 65x220x1 fine mesh and a 10x10x1 coarse mesh which leads to solve 200 linear systems of approximatively 1300 rows. We apply the reducing bandwidth renumbering algorithm to all matrices and their bandwidth is lower than 65 for all them.



(a) Server 1



(b) Server 2

Figure 5. Servers architecture

A. Hardware descriptions

The benchmark test cases have been run on two servers (figure 5):

- the first one, Server 1 is a Bull novascale server with a SMP node 2 quad-core intel Xeon E5420 GPU tesla server S1070 with 4 GPU tesla T10 with 30 streaming processors with 8 cores, 240 computation units per processor, total of 960 for the server. 16 GB central memory;
- the second, Server 2 is a server with a SMP node with 2 octo-core processors Intel Xeon E5-2680 linked by a NUMA memory and with 2 GPUs Tesla C2070 per processor with a fermi architecture.

B. Benchmark metrics

In our benchmark we focus on the execution time in seconds of the computation of all the basis functions of the study case. This computation time includes for each basis function, the time to discretize the local PDE problem, to build the algebraic linear system, to solve it with a linear

solver and to finalize the computation of the basis functions updating them with the solution of the linear system.

To analyze in detail the different implementations, we also separately measure in seconds:

- t_{start} the time to define basis matrix structures;
- $t_{compute}$ the time to compute the linear systems to solve;
- t_{sinit} the setup time of the solver;
- t_{solver} the time to solve all the linear systems;
- $t_{finalize}$ the time to get the linear solution and finalize the basis function computation;
- t_{basis} the global time to compute all the basis functions.

The performance results are organized in tables and graphics containing different times in seconds which can be compared to the time of a reference execution on one core.

C. Results of various implementations executed on various hardware configurations

Multithread forkjoin and GPU pipeline implementation: In table 6 and figure 7, we compare the performances of:

- the forkjoin concept implementations `TBB`, `BTH` and `PTH` using respectively the `TBBDriver`, `BTHDriver` and `PTHDriver` drivers which are all thread based implementations for multi-core configuration.
- the pipeline concept implementation based on the GPUAlgebraFramework.

We study the following hardware configurations:

- `cpu`, the reference configuration with 1 core;
- `gpu`, configuration with 1 core, 1 gpu;
- `n x p core`, configuration with n cpus and p cores per cpu.

In figure 7, we compare three implentations of the fork-join behaviour with threads. The analysis of the results shows that they all enable us to improve the efficiency of the basis function computation taking advantage of the multi-core architecture. The `PTH` implementation, directly written by hand with Posix threads is the most efficient while the `PTH` one implemented with Boost threads the less. The `TBB` version efficency is between the two others. In the implementation of the pipeline behaviour for GPU, we can notice that only the solver part is really accelerated on the GPU. Nevertheless it enables to improve the efficiency of the basis function computation with respect to the standard version on one core. Finally all these results prove that we can handle various hardware architectures, with one or several cores, with or without several GPGPUs, with a unified code. That illustrates the capacity of the runtime system to hide the hardware complexity in a numerical algorithm.

Multi-core multi-GPU configuration: For multi-core and multi-GPU configuration, we study the performance

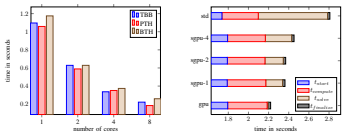
NbThreads	1	2	4	8	16
TBB	1.09	0.62	0.33	0.22	
Boost Thread	1.17	0.62	0.37	0.26	
Posix Thread	1.05	0.58	0.35	0.18	

(a) Basis functions computation time vs number of threads

opt	t_{start}	$t_{compute}$	t_{sinit}	t_{solver}	$t_{finalize}$	t_{basis}
cpu	1.73	0.36	0.	0.68	0.022	2.80
gpu	1.79	0.39	1.36	0.01	0.024	3.59

(b) Basis computation phase time for various solver configuration

Figure 6. Server 2: Multi-thread and GPU implementation results



(a) TBB,PTH,BTH (b) GPU

Figure 7. Performance analysis for multi-core configuration and GPU configurations

of a mixed MPI-GPU implementation with two levels of parallelism:

- the first level is a MPI based implementation for distributed memory;
- the second level is based on the `GPUAlgebraFramework` to solve the linear systems on GP-GPU devices.

We test different hardware configurations with different number of cores (1,2,4,8 and 16) sharing 1, 2 or 4 GPUs. In table 8 and figure 10 (respectively table 9 and figure 11) we present the performance results for the server 1 (respectively server 2).

The results show that the runtime system enable us to easily compare various hardware configurations: configurations where gpus are shared or not by cpus and cores, configurations with different strategies of connexion between gpus and cpus.

Analyzing the results of the different benchmarks, we have different levels of conclusions. the first level concerns the capacity of the Runtime system to hide the hardware complexity in a numerical algorithm. These benchmarks prove that we can handle various hardware architectures, with one or several cores, with or without several GPGPUs, with a unified code. The second level concerns the extensibility of the Runtime system. We could compare competing technologies with different implementations of our abstract concepts with few impacts on the numerical code. The third level concerns the capability of the Runtime system to really improve the performance of the numerical algorithm using the different levels of parallelism provided by hybrid architecture. With all the technologies tested

ncpu	1 gpu	2 gpus	4 gpus
1	1.95	1.95	1.95
2	1.22	1.04	1.04
4	0.98	0.76	0.66
8	0.63	0.37	0.45

(a) Computation times vs number of cpus and gpus

ngpu	2x2 cores	1x4 cores	1x8 cores
1	1.06	1.05	0.51
2	0.76	0.76	0.37
4	0.66	0.66	0.40

(b) Computation times vs number of cpus, cores per cpu and gpus

Figure 8. Server 1: multi-cores multi-gpu configuration

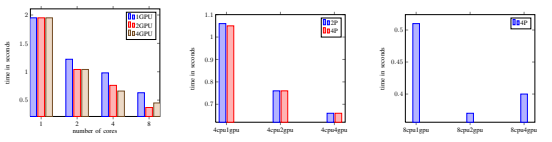
ncpu	1 gpu	2 gpus
1	0.75	0.75
2	0.44	0.38
4	0.25	0.24
8	0.12	0.12
16	0.05	0.06

(a) Computation times vs number of cpus and gpus

n x pe	2x2	1x4	2x4	1x8
1 gpu	0.53	0.67	0.57	0.50
2 gpu	0.46	0.45	0.37	0.37

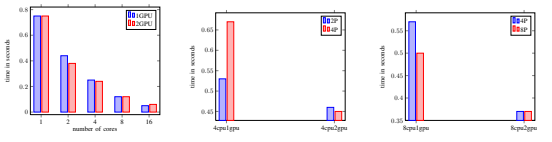
(b) Computation times vs number n of cpus, p cores per cpu and gpus

Figure 9. Server 2: multi-cores multi-gpu configuration



(a) Multi CPU, multi GPU (b) 4 CPU, multi GPUs (c) 8 CPU, multi GPUs

Figure 10. Server 1: multi-cores multi-gpu configuration



(a) Multi CPU, multi GPU (b) 4 CPU, multi GPUs (c) 8 CPU, multi GPUs

Figure 11. Server 2: multi-cores multi-gpu configuration

the performance of the computation has been improved compared to one computation executed on one core. The last level of conclusion is the fact that the runtime system enables to benchmark in a simply way the different hardware configurations parameters like the number of cores, the number of GPUs, the number of streams, the fact that a GPU is shared or not by several cores.

V. Conclusions and perspective

We have developped an abstract runtime system that enables to develop efficient numerical algorithms independantly of the hardware configuration. The results we have obtained implementing multi-scale methods with this runtime system have prove the interest of our approach to handle the variety of hardware technology with few impacts on the numerical layer. Nevertheless the solutions we have implemented are still to simple to get the maximum of the performance that can provide new heterogeneous architectures. We need to implement our different abstractions with advanced solutions as those existing in research runtime system solutions like StarPU or XKaapi. We plan also to benchmark different mechanisms that help to optimize data transfert between main memory and local accelerator memories and to measure the overhead of each solution with respect to the parallism grain sizes.

References

[1] D. Ayguade, R. M. Badia; F. D. Igual, J. Labarta,R. Mayo, and E. S.Quintana-Ort, *An Extension of the StarSs Programming Model for Platforms with Multiple GPUs*, Euro-Par 2009 Parallel Processing, Lecture Notes in Computer Science, 2009, Springer Berlin / Heidelberg SN - 978-3-642-03868-6

[2] Augonnet, C., Thibault, S., Namyst, R. and Wacrenier, P.-A. (2011), *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures*, Concurrency Computat.: Pract. Exper., 23: 187-198. doi: 10.1002/cpe.1631

[3] "*Charm++ Web page*", http://charm.cs.uiuc.edu/

[4] "*XKaapi Web page*", http://kaapi.gforge.inria.fr/dokuwiki/doku.php

[5] "*HWLOC Web page*", http://www.open-mpi.org/projects/hwloc/

[6] "*Intel Threading Building Blocks Web page*", http://hreadingbuildingblocks.org

[7] "*Boost library Web page*", http://www.boost.org/

[8] Tenth SPE Comparative Solution Project: A Comparison of Upscaling Techniques, M.A. Christie and M.J. Blunt, SPE Reservoir Simulation Symposium, Houston,Texas, 11–14 February 2001, Society of Petroleum Engineers

[9] V. Kippe and J.E. Aarnes and K-A Lie *A comparison of multiscale methods for elliptic problems in porous media flow*, Comput. Geosci., (2008), 12, 377–398