MHPM: Multi-Scale Hybrid Programming Model

A Flexible Parallelization Methodology

Nader Khammassi, Jean-Christophe Le Lann
Lab-STICC UMR CNRS 6285
ENSTA-Bretagne
29806 Brest Cedex 9, France
{nader.khammassi, jean-christophe.le_lann}@ensta-bretagne.fr

Jean-Philippe Diguet
Lab-STICC CNRS
University of South Brittany
56321 Lorient Cedex, France
jean-philippe.diguet@univ-ubs.fr

Alexandre Skrzyniarz
Domain Design Authority
Radar & Warfare Systems
Thales Airborne Systems
29200 Brest , France
alexandre.skrzyniarz@fr.thalesgroup.com

***Abstract*— The continuous proliferation of multicore architectures has placed a great pressure on developers to parallelize their applications accordingly with what such platforms can offer. Unfortunately, traditional low-level programming model exacerbate the difficulties of building large and complex parallel applications. High-level parallel programming models are in high-demand as they reduce the burdens of programmers significantly and provide enough abstraction to accommodate hardware heterogeneity. In this paper, we propose a flexible parallelization methodology, and we introduce a new task-based hybrid programming model (MHPM) designed to provide high productivity and expressiveness without sacrificing performance. We show that MHPM allows easy expression of both sequential execution and several types of parallelism including task, data and temporal parallelism at all levels of granularity inside a single structured homogeneous programming model. In order to demonstrate the potential of our approach, we present a pure C++ implementation of MHPM, and we show that, despite its high abstraction, it provides comparable performances to lower-level programming models.**

Keywords: Parallel Programming Model, Structured Parallelism, Skeleton, Execution Patterns, Parallel Constructs, Multicore

I. INTRODUCTION

With the rise of Chip Multicore Processor (CMP), parallel computing hardware is getting widely available at many scales: from personal computers to embedded systems to high performance supercomputers...[1,2,3,4]. While concurrent programming is still distant from the average sequential programmers, this proliferation of multicore architectures has placed a great pressure on mainstream developers to parallelize their applications as much as possible to take advantage of theses platforms. Parallel programming using the traditional thread-and-locks programming model remains a hard task for most of the programmers since it is time consuming, error prone and requires strong knowledge and skills. Consequently, programmers are facing a complex productivity-performance trade-off where they should extract enough parallelism to justify the use of a dedicated parallel programming library. Moreover, parallel hardware is becoming increasingly heterogeneous: a modern work station may includes two or more multicore processors with several manycore GPUs... In order to target such architectures, a programmer must have a deep understanding of the target hardware and should often use several disparate programming models making parallel programming harder and resulting into too poor productivity. Exploiting software parallelism on these emerging heterogeneous multicore architectures has become a great design challenge which outline the need for new technologies to make multicore processors more accessible to a larger community [18].

Due to this technological context, two major needs have been emerged: in one hand, a *high hardware abstraction* to hide details of the underlying platform providing portability , scalability and accommodating its heterogeneity. In the other hand, *programmability* improvement is in high-demand as it increase productivity and minimize programming complexity. These two needs should be satisfied without sacrificing *performance and forward scalability*. Programmability is achieved by minimizing parallel development cost in term of time, complexity and required tools in order to remain as close as possible to traditional sequential development. Parallel development overhead comes mainly from programming paradigms-related routines such as synchronization, communication, shared memory management, workload scheduling... These routines introduce a significant amount of extra-code related to parallel programming paradigms and not to the user application itself. Additional effects such as hard debugging and difficult performance tuning are also induced.

Skeleton-based programming, often referred as structured parallel programming [10,11], is a promising high-level approach which satisfy most of these requirements and attempts to replace the traditional low-level thread lock model with better abstraction and easier way to express parallelism through a collection of recurrent parallel patterns [6,9]. It aims mainly to provide a good trade-off between programmability, portability, reusability and performance increasing in order to improve programmer productivity by letting him focus on algorithms instead of hardware architectures. In this paper, we introduce a pattern-based hybrid programming model named MHPM (Multiscale Hybrid Programming Model) designed to provide high productivity without sacrificing execution efficiency. The philosophy behind its design is “Easing parallelism expression without loosing execution efficiency”. In MHPM C++ implementation, productivity is promoted through providing a friendly and intuitive programming interface allowing easy expression of both sequential execution and several types of parallelism, including task parallelism, data parallelism and temporal parallelism (pipelining), at multiples level of granularity inside a single homogeneous

and structured model through an extendable collection of execution patterns or algorithmic skeletons.

In order to illustrate the potential of MHPM and the simplicity of its programming interface, Fig. 1 shows an irregular hierarchical task graph representing multiple fork/join execution pattern at several level of granularity, and Fig. 2 show how parallelism in this relatively complex graph can be expressed through a single C++ line of code (line 5) without altering the original sequential code. Each task of this graph can be easily defined, at the cost of a single line of code, from a function, a class method or a lambda expression allowing direct reuse of sequential code without any constraints on function or class method prototypes and without modifying their code. Moreover, in MHPM, concurrent shared data access, and therefore potential "race condition", is detected transparently, then shared data is protected automatically at any level of granularity through critical sections relieving the programmer from managing shared data by himself such in near-all parallel programming model where programmer has to manage it manually using mutual exclusion primitives.

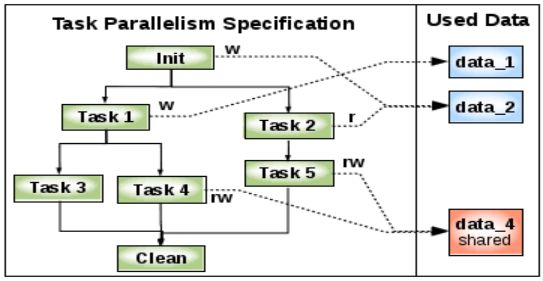

Figure 1. Example of task graph specifying task parallelism and task-data dependencies (r: read, w:write).

```
1 void main() {
2  task ta(function, data_1),      // task definition
3       tb(&o, cls::method, data_2), ...;
4  task_group * program;
5  program = parallel(sequential(ta, parallel(td,te)),
6                     sequential(tb, tc));
7  init();
8  program->run(); // 'data_4' protected automatically
9  clean();
10 }
```

Figure 2. Task parallelism expression in MHPM

MHPM is based on a powerful and rich intermediate program representation, named the *Hierarchical Task Group Graph* (HTGG). HTGG is the heart of MHPM which specify sequential execution and several types of parallelism at many levels of granularity disregarding the available amount of parallelism in the target program. HTGG is built from an extendable collection of nestable execution patterns which can be used hierarchically inside each other allowing progressive parallelization, better granularity control and data, task and temporal parallelism integration inside a single homogeneous and highly structured programming model. Fig. 3 shows how a sequential program can be parallelized through specifying parallelism in the HTGG and gives an overview of MHPM architecture which is mainly composed of :

- A C++-based programming interface which exploit C++ meta-programming capabilities [24] to ease expression of parallelism in the HTGG.
- A *Hardware Abstraction Layer (HAL)* which provides dynamically a description of the underlying architecture and accommodate hardware heterogeneity: HAL uses dynamic hardware exploration to detect available computing resources and their properties (available processing unit, their execution capabilities, processor cache topology …)
- An *Intelligent Run-time System (IRS)* which exploit information, extracted transparently from both hardware description and used execution patterns (task ordering and task-data dependencies) to perform efficient execution on the underlying architecture.

The MHPM is implemented as a C++ framework named XPU. This framework offers an intuitive, easy-to-use and light-weight programming interface to design parallel applications or parallelize sequential ones. At the opposite of many parallel programming models which introduce new languages, define compiler annotations or extends existing language and thus requires specialized compiler, extra-hardware, or virtual machines [4,18]... XPU is a pure software technology entirely based on the traditional standard ISO C++ language and requires nothing more than a standard C++ compiler to be used, and therefore, improve learning curve steepness and is easily portable to many systems.

In this paper we will focus mainly on the programming interface and the program parallelization methodology: we show how a sequential program can be progressively parallelized and represented as a HTGG, then we explains how several types of parallelism as well as sequential execution can be specified easily in the HTGG through the provided collection of nestable execution patterns. We discuss how our programming interface is able to extract transparently information on task-data dependencies and how this information can be exploited to improve productivity and provide dynamically efficient execution on the underlying multicore architecture. Finally we give a brief overview of achieved performances by XPU. Since hardware abstraction and task scheduling techniques are not the subject of this paper the HAL and IRS components will be discussed in dedicated papers.

II. RELATED WORKS

Structured parallel programming with deterministic patterns [6] is a high-level approach mainly based on a collection of recurrent parallel execution patterns, often referred as algorithmic skeletons [9,10,11] or parallel constructs, which abstract program description and hides low-level multithreading details and many complexities inherent in parallelism from the programmers [16,17]. These reusable patterns automate many parallel paradigm-related routines such as synchronization, communication, data partitioning or task scheduling... and handles them internally. For instance, many task-based structured programming models such as Thread Building Blocks [7] and Cilk++ [8] offer a set of execution patterns which handles transparently task scheduling, data partitioning and load-balancing. Unified Parallel C (UPC) Task Library HotSLAW [5] abstracts concurrent task management details and provides transparent data communication and dynamic load balancing [5,18].

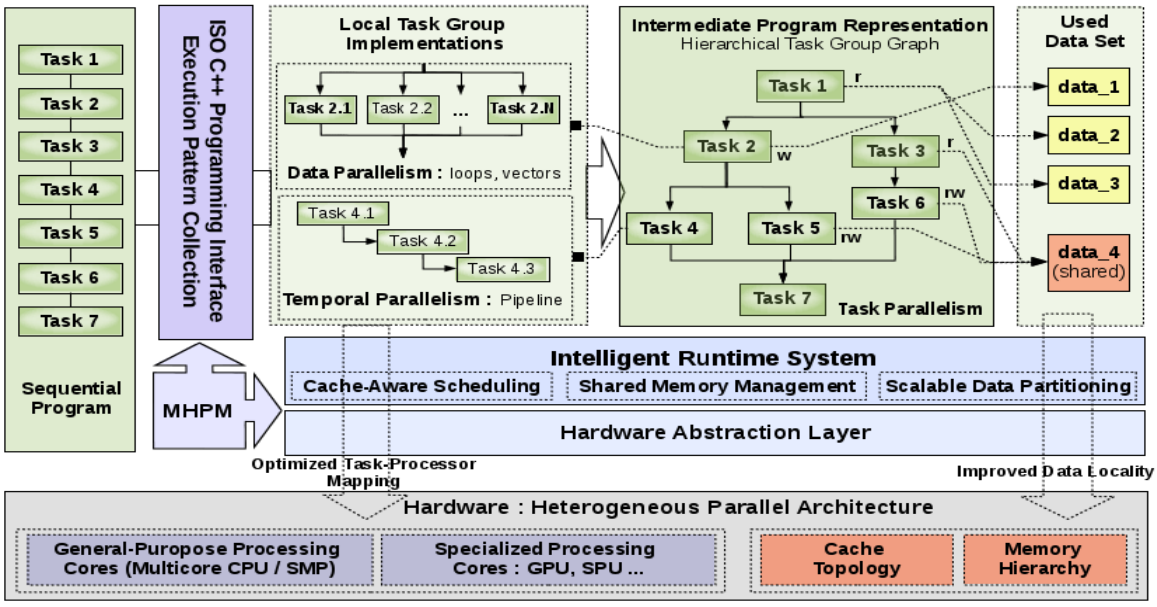

Figure 3. Overview of the intermediate program representation (HTGG) and MHPM architecture.

Sequoia [20] is another task-based programming model which offer transparent data management in deep processor's memory hierarchy including data allocation and communication through the memory tree [18].

By decoupling the programming model from the underlying architecture, pattern-based approach offer also a good hardware abstraction accommodating architecture heterogeneity, opening the path to more hardware support and allowing the programmer to focus on algorithms instead of hardware architecture. For instance MIT-LL is developing the Parallel Vector Tile Optimization Library (PVTOL) [14] in order to expand parallel programming constructs in Parallel Vector Library (PVL) [22] and VSIPL++ [12] to support both homogeneous and heterogeneous multicore architectures [18].

Despite their ability to express parallelism at the cost of relatively little amount of programming effort, most of task-based parallel programming models target specific application domains such as signal processing in the case of PVL, PVTOL and VSIPL++ and offer limited collection of execution patterns to express specific type of parallelism: for instances if Cilk++ allows easy expression of simple and nested task parallelism, its ability to express temporal parallelism, such in the pipeline execution pattern, is much harder and requires verbose restructuring of the code [27]. Finally, in spite of their high hardware abstraction, most of known task-based programming are not yet able to support heterogeneous multicore architectures requiring programmers to use one or more additional programming models, such as OpenCL[25] or CUDA[26] to support GPGPU for example, in conjunction with another task-based programming model to exploit multicore CPU and SMP platform, and perhaps a third programming models such as MPI to support distributed memory architecture... resulting into heterogeneous programming model hard to use and maintain and requiring multiple skills and deep understanding of different hardware and software components. These constraints results in a severe productivity loss and can be discouraging for the average sequential programmers.

Main-stream applications and general-purpose programs are "more-or-less" parallelizable depending on their nature: programs may expose a varying amount of parallelism and consequently different parallel-sequential ratio: : many known scientific simulations and signal processing problems are massively parallelizable however many other general-purpose applications are much less parallelizable and exposes much more sequential execution constraints such many video decoding algorithms [28] and compression algorithms [29]. This outlines the need to express both parallel and sequential execution into a single homogeneous hybrid programming model able to specify both sequential and parallel execution in order to provide a generic and non-domain specific programing model. Also, a program may exposes several types of parallelism often difficult to express using a single programming model so programmer uses several disparate programming models inside the same application resulting into ineffective uses of processors caches, poor load-balancing and potentially system overloading with many independent run-times.

MHPM try to bypass these limitations by allowing easy expression of both sequential execution and several types of parallelism at multiples level of granularity inside a single homogeneous model, so a programmer can parallelize its application as much as possible by using a single flexible programming model. Since it allows the specification of parallel execution as well as sequential one, MHPM target a wide range of programs from various application domains: from highly parallelizable applications to much less parallelizable ones and remains valid even for fully sequential ones. Since promoting productivity is one of our primary design goals, MHPM handle implicitly many parallel programming paradigm-related routines such as synchronization, communication, shared memory management … and therefore, hide many complexities

inherent in parallelism from users. The internal design of parallel and sequential patterns allows transparent extraction of valuable information on task-data dependencies enabling an intelligent run-time system to detect shared data between tasks and protect it transparently against conflictual accesses often referred as “race condition”.

III. A TASK-BASED PARALLELIZATION METHODOLOGY

Task-based programming is based on the decomposition of a program into a set of tasks which cooperates with each others to perform the main work of the application program. Tasks granularity can be controlled and specified by the programmer: a program is basically the main task which is split into several coarse-grain tasks which may be split, in turn, into finer-grain ones, and so on... until we reach the finest-grain allowed by the host programming language (cf. Fig. 4).

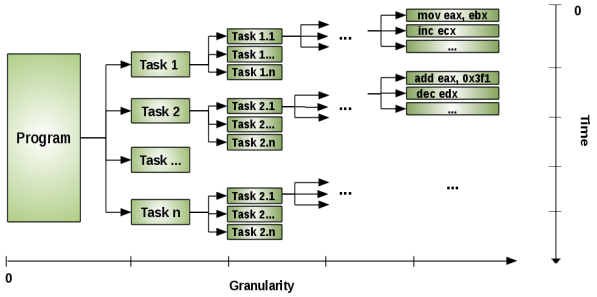

Figure 4. Program decomposition at many granularity levels

Each task of the application program performs a piece of work in which it may consumes or produces data, i.e., read or write private or shared data. In order to speedup optimally program execution on parallel computing architectures, we have to extract the maximum amount of parallelism. The ideal case, is the one in which all tasks, at the finest possible granularity level, doesn't exposes any data or control dependencies, so they can be executed simultaneously (cf Fig. 5). Unfortunately, real world programs are “more-or-less” parallelizable depending on their natures: while many scientific simulations exposes massive data parallelism and

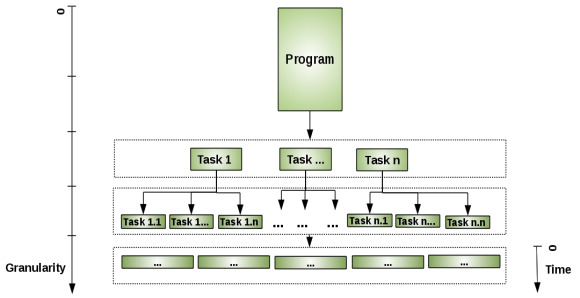

Figure 5. Ideal Parallel Program

thus are highly parallelizable , many other general-purpose applications, which represents the wide majority in the software landscape, are much less parallelizable due to data and control dependencies and explicit task ordering. Indeed, these algorithmic constraints introduce needs for synchronization and ordering to preserve memory coherency and algorithmic consistency. Consequently, each subset of the tasks composing the program can be executed either in parallel or sequentially depending on these constraints which define thus the parallel-sequential code ratio or the available parallelism in the target program. At the end of the parallelization process, we obtain a hybrid execution graph containing both sequential and parallel sections (cf Fig. 6). The available parallelism vary depending on applications natures, but the model remains usable for either highly and weakly parallelizable programs and even for fully sequential ones.

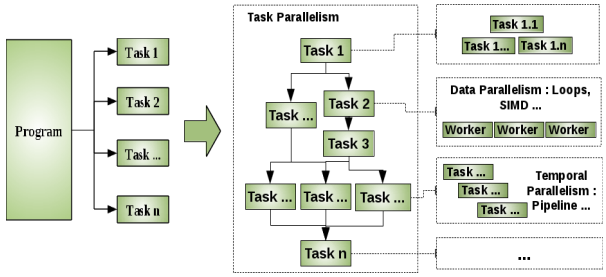

Figure 6. The Hybrid Programing Model specifies sequential execution and several types of parallelism at all level of granularity.

Tasks may expose locally several types of parallelism including nested task parallelism, data parallelism at thread through parallel loops or instruction level through vectorization or temporal parallelism through pipelined execution. These execution configurations can be specified into a collection of execution patterns. For the sake of simplicity, we use interchangeably “execution pattern”, “construct” or “skeleton” to indicate a structure storing a set of tasks and specifying their execution configuration.

A. *The Hierarchical Task Group Graph*

In order to accommodate execution patterns heterogeneity, so they can fit into a single homogeneous structure representing the program, we define a common abstract constructs named “task_group”. All our execution patterns implement this common interface: for example “sequential_tasks” are a group of tasks scheduled to run sequentially while “parallel_tasks” are a group of tasks scheduled to be executed simultaneously (a basic fork and join pattern) and “pipeline” is a group of communicating tasks running as a chain of overlapped processing stages... etc (cf. Fig. 7). These task group implementations can be easily extended to express more execution patterns and meet specific programmer needs in all applications domains. We note the “Task” is also, by design, a “task_group” containing a single task. Consequently most provided constructs are nestable and can be used hierarchically inside each other.

By expressing parallelism at several level of granularity using these patterns, we obtain a hierarchical structure composed from task groups of “task_group”, this structure is named HTGG. Task ordering is specified inside each constructs, so when a task group is called, it execute its sub-task groups following the specified execution pattern and each of these sub-task groups will, in turn,do the same with their sub-task group ...etc.

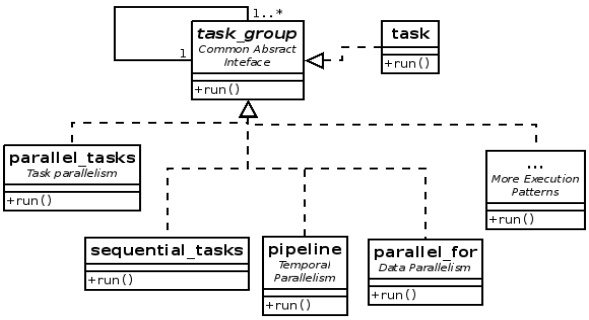

Figure 7. A Simplified overview of the internal software design accommodating constructs heterogeneity so they can fit inside a single homogeneous and hierarchical structure: The HTGG

IV. PROGRAMMING INTERFACE

The HTGG is a complex structure containing several heterogeneous constructs to express different execution patterns. It encapsulates not only the code of tasks but also specify task execution ordering and contains many other information about task-data dependencies and shared memory and provides an interface to specify the task-processor mapping. In order to build easily this complex structure, we tried to exploit C++ Meta-programming capabilities to offer an intuitive interface to build the HTGG at the cost of a little amount of paradigm-related extra code and to promote reuse of sequential code with the lowest possible modification/alteration. C++ Meta-programming techniques imply massive uses of templates which offer great compile-time optimization but also may make the code less readable and relatively verbose on programming errors. In our case, we used templates internally behind front-end polymorphic functions to relieve the user from specifying explicitly argument types and count when building execution patterns. Consequently, the resulting programming interface is easy-to-use and doesn't expose any template structure to the programmer. As we progress in this section we will mention the different meta-programming techniques used to simplify the programming interface and to provide advanced features such as transparent shared memory detection and protection.

A. Example of a multimedia application

In order to illustrate the potential of our programming model, we consider a simplified example of a multimedia application. In the begining, we introduce the sequential version written in C++ and we show how we can parallelize it progressively at many levels of granularity the collection of execution patterns provided by XPU.

1) Sequential version

Every frame of the input stream of our multimedia application contains sound and images. Our program process sound samples, and images before encoding them into a common compressed output bitstream. Fig. 8 and 9 gives a general overview of the sequential program algorithm and the associated skeleton of sequential c++ code. In the first place we don't discuss implementation details of each block of the algorithm, we consider simply a set of functions coded using the traditional sequential programming model, (They may be functions, class methods, or even remote function calls...), we give just their prototype without discussing their internal implementations to illustrate how we can reuse directly sequential code without significant modifications.

```
1  int load_input_stream(char * in_stream);
2  int extract_audio_samples(char * in_stream, char * audio_samples);
3  int process_audio_samples(char * audio_samples);
4  int encode_audio_samples(char * audio_samples);
5  int write_audio_to_stream(char * out_stream, char * audio_samples);
6  int extract_video_frames(char * in_stream, char * video_frames);
7  int process_video_frames(char * video_frames);
8  int encode_video_frames(char * video_frames);
9  int write_video_to_stream(char * out_stream, char * video_frames);
10 int write_output_stream(char * out_stream);
```

```
1  int main()
2  {
3    char * in_stream, * out_stream, * audio_samples;
4    image *  video_frames;
5    while (input_stream_available)
6    {
7      load_input_stream(in_stream);
8      extract_audio_samples(in_stream, audio_samples);
9      process_audio_samples(audio_samples);
10     encode_audio_samples(audio_samples);
11     write_audio_to_stream(out_stream, audio_samples);
12     extract_video_frames(in_stream, video_frames);
13     process_video_frames(video_frames);
14     encode_video_frames(video_frames);
15     write_video_to_stream(out_stream, video_frames);
16
17     write_output_stream(char * out_stream);
18   }
19 }
```

Figure 8. Sequential version of the multimedia application

2) Parallelization

In a first time we decompose our program into tasks simply by reusing the sequential functions as tasks. By looking to our sequential program as shown in Fig. 9, we can identify coarse-grain task parallelism between audio processing and video processing which can be executed simultaneously since they are independent so we can start by parallelizing our program through a large grain fork/join pattern. In a next step, and at a finer grain, if we analyze each task of both the audio and video processing task group, we can identify, locally, several parallelism types including data parallelism through parallelizable for loop such in “audio_encode_samples” task, temporal parallelism by using the pipeline pattern in the audio and video fileting tasks and finally massively parallel operations on large vector of data (massive SIMD operations) such as in the last stage of “process_video_frame” pipeline : the “multiply” task. We can go further and parallelize at a finer grains to extract the maximum amount of parallelism. Final parallel program structure is relatively complex, however, in the next paragraphs, we show how parallelism in this program can be expressed at the cost of a little amount of extra-code and thus little programming effort and we outline the flexibility of our model which allow progressive parallelization of the target program.

B. Task Definition

Decomposing a program into a set of pieces of code is the first step in the parallelization process in most of parallel programming models, in low-level thread-lock programming model, these pieces of code are called callbacks, in higher level task-based programming models this piece of code is called task. We outline the high programmability of our programming model by comparing it to lower level one (PThreads) and the high-level task-based programming model Threading Building Block (TBB).

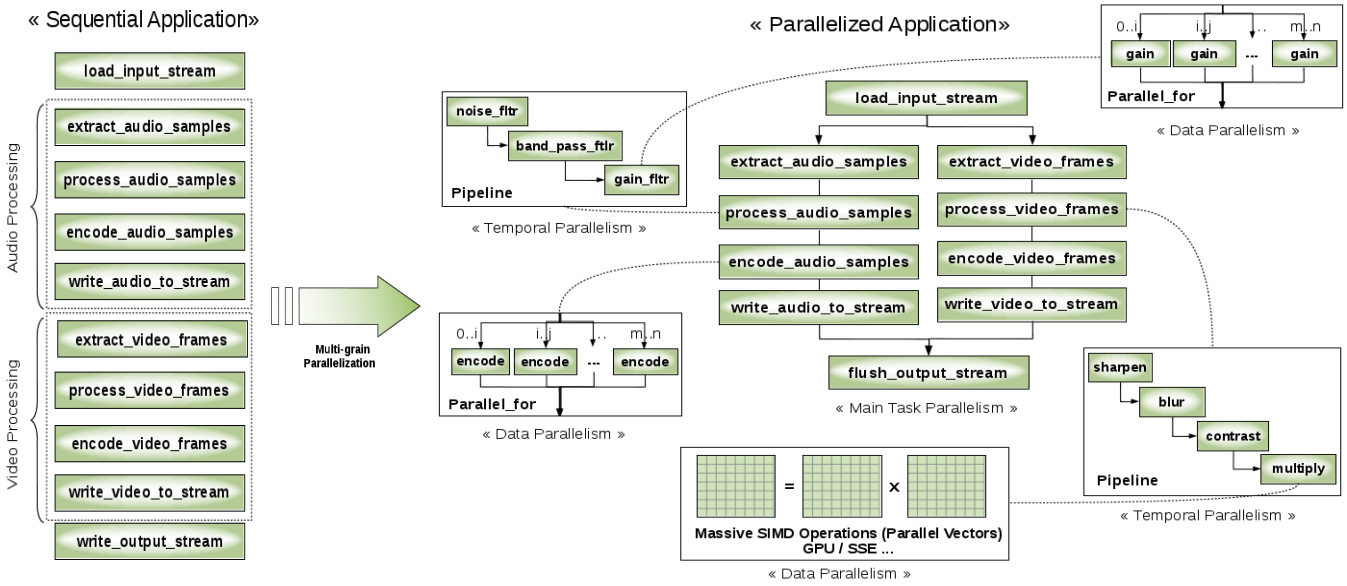

Figure 9. This multimedia application can be parallelized at several level of granularity: task parallelism between audio and video processing and local data and temporal parallelism inside several tasks can be expressed in the HTGG.

1) POSIX Threads Programing Model

In the traditional low-level thread-lock model, this piece of code is called callback and plays the role of tasks in task-based programming model and is the main component of multi-threaded applications. If we consider C++ language, the host-language of our programming model, sequential code is often severally altered since the targeted piece of code has to meet the native callback prototype "void * function(void *)" which imposes many restrictions to the programmer when parallelizing application or reusing sequential code : only static functions can be used as callback, dynamic class method can't be used directly, in addition, consumed and produced data should be stored in a common intermediate structure then extracted and restored to their original type through type casting. These constraints lead to many modification of the sequential code, usually a lot of programming paradigm-related extra-code and thus make the code less readable, error-prone and difficult to maintains. This lack of flexibility and programmability amplify the burden of the programmer dramatically and make the reuse of sequential code difficult.

2) Threading Building Blocks

TBB is a high level programming model which provides more abstraction and allows the reuse of sequential code in a less restrictive way. However, significant modification to sequential code are required: task code and its consumed or produced data should be encapsulated in a class respectively as class members and class methods with specific prototype. Consequently, sequential code can't be reused directly and has to be significantly transformed. This leads to verbose and less readable code and requires significant programming effort.

3) XPU

Since promoting the reuse of sequential code is one of the primary design goals of XPU, we tried to outcomes the previously enumerated limitations through a more flexible task design. In MHPM, by design, a task is basically an abstract callable piece of code which can be executed. This piece of code may consume or produce data. Data are passed in the form of arguments to each task. Fig. 10 show how a task can be created from a function or a class method disregarding its argument count or type and its return type.

A more advanced implementation of tasks for static and dynamic distributed systems are in development at the time of writing of this paper. We exploit C++ meta-programming capabilities to provide a friendly programming interface allowing simple and fast definition of tasks from existing code to promote reuse of code and improve programmer productivity. At the same time, by combining polymorphism and several C++ template programming techniques we take advantage of compile-time compiler optimization to produce efficient code. Moreover, meta-programming techniques allow us to investigate used data type ,through Compile-Time Type Identification CTTI [23,24], and therefore, detect transparently task-data dependencies at compile-time. This information are exploited to detect automatically concurrent accesses to shared data in the HTGG and protect it against potential race condition. In ongoing works, we try to exploit this same information to perform efficient execution on the underlying architecture dynamically by improving temporal and spatial data locality through cache-aware task scheduling.

When defining a task, data access is specified implicitly or explicitly through the passed argument. By default, argument passed by value are considered as a local read access data, arguments passed by pointer are considered as a potentially shared data accessed in write mode, argument passed through constant pointer are considered as a potentially shared data accessed in "read only" mode (it can be explicitly specified using the *__read_only( )* macro or simply passed as a constant pointer argument), finally, case of arguments passed by references is not yet treated due to value/reference ambiguity issue with some compilers.

```
1  int load_input_stream(char * in_stream);
2  int extract_audio_samples(char * in_stream,
3                            char * audio_samples);
4  int main()
5  {
6   task load_stream_t(load_input_stream, in_stream);
7   task extract_audio_t(extract_audio_samples,
8                        __read_only(in_stream),
9                         audio_samples);
10   ...
11   load_stream_t(); // or load_stream_t.run() to run task
12 }
```

```
1  class image
2  {
3    public:
4      int sharpen(int val);
5      int blur(...);
6    ...
7  };
8  int main( )
9 {
10   image img("img.jpg");
11   task sharpen_t(&img, &image::sharpen, 11);
12 }
```

```
1 int main( )
2  {
3   float * f;
4   xpu::task low_pass([](float * samples, int freq)
5   {...code... }, audio_samples, 7000);
6  }
```

Figure 10. Task definition from different pieces of code.

As chown in Tab. I, despite its high abstraction, task call introduce negligible execution overhead.

TABLE I. TASK CALL OVERHEAD (CLOCK CYCLES): COMPARISON BETWEEN CALLING TASK ENCAPSULATING FUNCTION CODE AND DIRECT FUNCTION CALL.

Compiler	C++ GNU v4.6.1			Intel v12.0.5		
Optimization	-O1	-O2	-O3	-O1	-O2	-O3
Function call	2358	2356	962	1778	966	966
Task call	2356	2358	1038	1782	968	968
Overhead	-2	2	76	4	2	2

V. TASK PARALLELISM

If we consider data dependencies in our multimedia application example, video processing and audio processing can be viewed as two coarse-grain independent tasks and can be executed in parallel. However input stream loading task should be executed before any stream processing. Finally, output stream writing task can't be executed before processing tasks are done. Task ordering and parallelism can be simply specified as in the Fig. 11 and expressed using XPU as in Fig. 16.

Task ordering can be specified recursively using two keyword "*parallel*" and "*sequential*" whatever the task graph complexity, irregularity or depth. These functions take as arguments, two or more "task" or "task_group". At the end of task ordering specification we obtain a HTGG which specify parallelism and sequentiality at all levels of granularity. Once the tasks graph constructed, tasks are ready to be executed in the specified order. Since tasks ordering specification is inherent in the HTGG, thread creation, task spawning and synchronization (fork and join) are managed transparently by the run-time scheduler which ensure the execution of tasks in the specified order. If we summarize the previous, expressing parallelism in our programing model can be performed, mainly, in three steps: tasks definition, HTGG construction and finally HTGG execution by calling the root task.

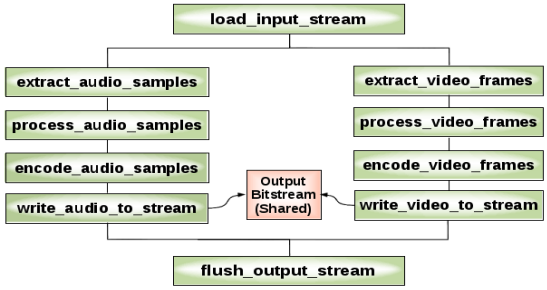

Figure 11. Intermediate program representation

```
1  int main()
2  {
3   ...
4   task_group * process_audio = sequential(extract_audio_t,
5                                           process_audio_t,
6                                           write_audio_t);
7
8   task_group * process_video = sequential(extract_video_t,
9                                           process_video_t,
10                                          write_video_t);
11  task_group * program ;
12  program = sequential(load_stream_t,
13                       parallel(process_video, process_audio),
14                       write_stream_t);
15
16  while (input_stream_available)
17        program.run();
18 }
```

Figure 12. We express task parallelism the multimedia application in two steps to have more readable code.

Separating task definition and graph construction from tasks execution allows our run-time system to exploit the valuable information about task-data dependencies encapsulated in the tasks as well as the explicit parallelism specification, inherent in the tasks graph, to perform several operations before executing the task graph. In order to promote productivity, speedup and simplify parallel programming MHPM automates many parallel-programming paradigm-related routines such as shared memory management, tasks synchronization and performance tuning. In the next paragraph we review briefly transparent shared data detection and protection feature.

A. Transparent Shared Data Detection and Protection

We can remark, in the definition of the task "write_video_stream" and "write_audio_stream", that both of these tasks write their output bitstream into the same common "output_stream" (cf. Fig. 8 and 11). When building task graph, the run-time system will check dynamically task-data dependencies to determine which task accesses which data and how it is accessed (write or read mode), then it will look for shared data in parallel sections at all granularity levels. Finally, if two or more concurrent tasks accesses shared data in write mode, these tasks will be transformed into critical sections by associating transparently a "lockable" (an abstraction of mutual exclusion mechanism: mutex and spinlock are an example of implementations of the "lockable" interface.) to the shared data so it will be protected against potential race-condition when executing the tasks. We note that this technique suppose no unsafe direct accesses from tasks to global variables since this type of accesses is out of the control of our run-time system which uses arguments list to determine task-data dependencies. Consequently, programmer is invited to specify explicitly used data in its argument list when defining tasks and should not hides pointers to shared data inside complex structures.

Specifying in/out data accesses for each task can be seen as a constraint, but also it can be seen as a good practice which make code easier to read, maintain and parallelize. Following this unique rule, the IRS will gantry transparent management of shared data and prevent potential "race-condition" automatically. Breaking a big grain task, such video and audio processing tasks in our example, into a sequence of finer grain tasks can be useful if this task share data with parallel tasks: since the task in this case is executed inside a critical section, dividing this task into a set of sub-tasks may reduce critical section size and maximize parallelism.

B. *Experiment: Programmability Evaluation*

In this experiment, we tried to evaluate XPU programmability in comparison with TBB. A traditional approach for quantifying programmablity or required programming effort is to compare the parallel program to its sequential version in term of the number of lines of code [31]. We consider a simple sequential application in which we call successively 7 functions, this application can be parallelized as shown in our first example in Fig. 1. We can define a task for each function when required. We note that "data_4" is shared between two parallel tasks (4 and 5). We tried to express task parallelism as specified in the task graph using XPU and TBB then we counted the required parallel paradigm-related extra-code line and the reused sequential code lines. We used "CLOC" [30] to count lines of code in each version, we removed all blank lines and comments, we verified we have exactly the same number of lines of code found by CLOC, then we used the classic "comm" tool, available in most UNIX systems, to determine the number of lines of both reused sequential code and required extra-code.

In the TBB version we define 6 task classes: 3 tasks to encapsulate function 1,2 and 3, one intermediate task hold these 3 tasks, another intermediate task to call successively function 2 and 5 and finally we define the root task that spawn these intermediate tasks. We use TBB task parallelism primitives: allocating, spawning and waiting for root and child tasks. A TBB mutex is used to protect shared data "data_4" from race condition [7]. XPU version require a single line to define a task for each function in the parallel section and another 2 lines to build the task graph and execute it. The sequential code of functions doesn't require any modification. The shared data is detected and protected automatically by the run-time system. As show in Fig. 13, while XPU version reuse 80 lines among the 86 lines of sequential code and requires only 10 extra-lines for parallelization, the TBB version reuse 42 lines and requires 140 lines of parallel paradigm-related extra-code.

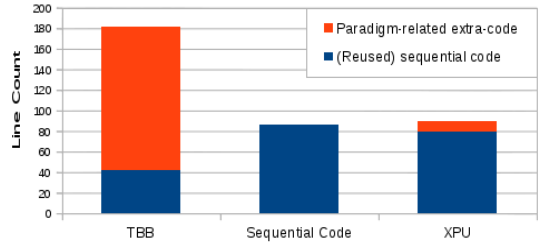

Figure 13. programming models in term of required programing effort: required extra-code and reuse of sequential code

VI. TEMPORAL PARALLELISM

MHPM enables the programmer to express temporal parallelism easily through the pipeline execution pattern. Pipeline pattern implements the abstract *«task_group»* interface and thus can be used at any level in the HTGG as the other *«task_group»* implementations. The *«pipeline»* pattern can be constructed from a set of tasks used as overlapped processing stages. As shown in Fig. 14, these stages are executed to process elements of a data container: each element is processed by the first stage or the "head" of the pipeline, when finished, this stage notify its successor that the element is ready to be processed by the next stage, then it moves to the next data element. This cycle is reproduced for each stage until we reach the tail of our pipeline. So sequentiality of the execution is satisfied at the element level, but parallelism is exploited at the data container level increasing therfore task throughput. Pipeline is a recurrent execution pattern in many application domains such as signal, image and more generally stream processing.

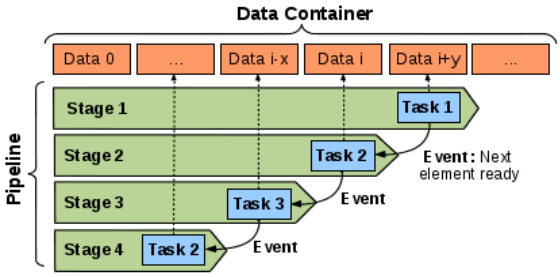

Figure 14. Internal Pipeline Architecture

In our example, pipeline pattern can be used in the image processing task "process_video_frame" in which many filters are applied to each frame of the incoming stream. Filters should be executed in the specified order for each image of the input stream. In order to satisfy filter ordering, filters can be used as pipeline stages so each image is processed by the first filter then can be immediately processed by the next filter without waiting until all images are processed. We note that the first function argument is used as an index of the element to be treated, it is updated by the pipeline constructs for each stage until all elements are processed by all stages. Fig. 15 show how we can build a 4-stages pipeline. This pipeline can be used to parallelize the task "process_video_frames" of our multimedia application.

```
1 void sharpen(int i, vector<image> * imgs) // i = frame index
2 { imgs[i]->sharpen(); }
3
4 void multiply(int i, vector<image> * imgs, image * mask)
5 { imgs[i]->multiply(mask); }
6
7 int main()
8 {
9   vector<image> frames(size);
10  ...
11  task sharpen_t(sharpen, 0, &frames),
12       blur_t(blur, 0, &frames),
13       multiply_t(multiply, 0, &frames, &mask);
14 task_group * process_image = pipeline(size, sharpen_t,
15                                        blur_t,
16                                        contrast_t,
17                                        multiply_t);
18  process_image->run(); // frame index "i" will be updated
```

Figure 15. An example of four stages pipeline

VII. DATA PARALLELISM

Data parallelism refers to scenarios in which the same operation is performed concurrently on elements of a data container [33]. Data parallelism can be specified at different levels of granularity and can be implemented at thread level or at the instruction level. In data parallel operations, data is partitioned so multiple threads can operate on different data partition concurrently. XPU enables the programmer to express data parallelism at thread level (TLP) through parallelized for loop , at instruction level (ILP) through a set of vectorized data types (SIMD) and at both through parallel vector.

A. Parallel loop

The "parallel_for" pattern is a task group implementation specifying parallel for loop. When defining a parallel loop (cf. Fig. 16), its main range will be partitioned transparently using a pseudo-fair partitioning algorithm into several sub-ranges depending on the available processing units. Partitioning is performed at the pattern construction stage minimizing overhead at construct execution and allowing us to implements various data partitioning schemes without caring about algorithmic complexity: for instance, we experimented several workload distribution techniques and data partitioning algorithms without modifying the «*parallel_for*» front-end due to its high abstraction of implementation details. The run-time system behind our programing interface, its data partitioning algorithms and exceeds the subject of this paper and will not be discussed in dedicated papers .

```
1  int process(int from, int to, int step, image* images) {
2    for (int i=from; i<to; i+=step)  ...
3  }
4  void main()
5  {
6    image * images = … ;
7    task process_t(process, 0,0,0, images);
8    task_group * pf;
9    pf = new parallel_for(0, image_count, 1, &process_t);
10   pf->run();
11 }
```

Figure 16. An example of parallel for loop definition

B. Parallel Vector: Massive data parallelism on heterogeneous multicore architecture.

Vector is a popular homogeneous data container which might contains a significant amount of data. This structure is particularly suited for massively parallel operations on different partitions of the target vector. Providing an abstract parallel vector interface allows the parallel processing of its element using different algorithms and on various multicore and many-core platforms: for instance Thrust [21] is a parallel algorithms library which offers a C++ standard template library-like vector interface and inter-operates internally with CUDA, TBB or OpenMP to run on multicore and many-core architectures including CPU and GPU. PVL [22], PVTOL [14] and VSIPL++ [12] uses high-level vector, matrix and tensor data structures in conjunction with task maps and data maps to parallelize data processing and distributes workload across available processors [18].

OpenCL is established as a standard for heterogeneous computing [25] and was explicitly designed with abstraction that are low-level, high performances and portability [18]. OpenCL programming interface requires explicit management of memory, kernels compilation and workload scheduling resulting into verbose code and requiring deep understanding of all software and hardware components. In our programing model we defined an abstract parallel vector interface in top of OpenCL enabling programmer to perform massively parallel operations on heterogeneous multicore architecture. Our vector interface uses the operator overriding features of the C++ language to translate transparently, at run-time, basic operations, such as addition or subtraction...etc, into corresponding OpenCL kernel then managing transparently memory transfers and workload scheduling. Fig. 17 shows an example of vector addition.

```
1 #define size 1000000
2 int main()
3  {
4    xpu::vector<float> A(size), B(size), C(size);
5    A = B + C; // Transparent addition on GPU/CPU/...
6 }
```

Figure 17. Parallel vector interface: OpenCL kernel is generated transparently and memory transfers is managed automatically.

C. Brief overview of XPU performances

In order to give an overview of the achieved performances by our data parallelism implementation, we used our framework to parallelize the popular "Black-Scholes" problem at thread level using the "parallel_for" construct and at a finer grain, at the instruction level, using the vectorization capability provided by XPU through a built-in vectorized type (vec4f) implemented in top of SSE to support SIMD. We used the sequential code of the "blackscholes" application as provided in PARSEC Benchmark Suite [13,32]. Main processing loop has been parallelized at the cost of 3 lines of extra-code. Vectorization has been introduced simply by replacing regular float type by the "vec4f" vectorized type and by setting increment step of "parallel_for" to 4 instead of 1. We compared achieved performance by our application to the five parallel version provided in the same benchmark suite: OpenMP, TBB, Pthreads, OpenMP/SSE and PThreads/SSE. We used the Intel C++ Compiler v12.0.5 and we executed our benchmark on several multicore platforms. In XPU, optimal thread count is determined automatically at run-time, for the other programming models, since thread count is fixed manually, we choose the thread count giving the best results for each platform. Figure 19 show the achieved performances for different data input size on a 16 Threads SMP platform. Fig. 20 give an overview of the scalability of each version on several platforms ranging from 2 to 16 hardware threads.

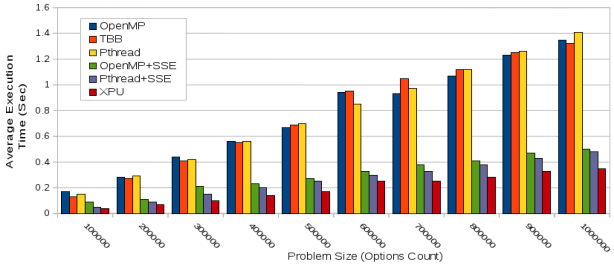

Figure 18. Execution time of the "blackscholes" application for different problem size on 16 hardware threads platform (SMP with 2 x two Intel Xeon E5620 at 2.4 GHz)

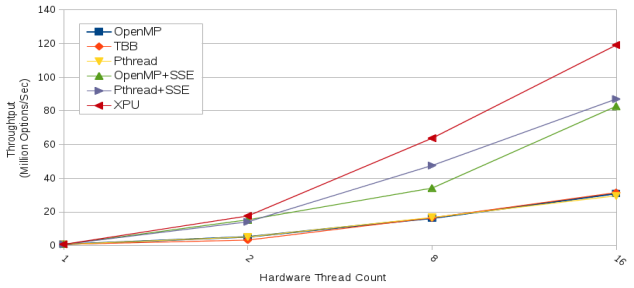

Figure 19. Achieved Throughput (Options/sec) .

VIII. Conclusion and Future Works

In this paper, we showed how several different types of parallelism can be easily specified at all level of granularity inside the HTGG through a friendly programming interface designed for high productivity and programmability. The HTGG is a rich intermediate representation of parallel programs encapsulating not only parallelism and task ordering specification but also task-data dependencies. At the time of writing this paper, we try to exploit information about task-data dependencies in conjunction with the provided description of the underlying architecture by a dynamic hardware explorer (processing unit count, execution capabilities, cache topology and sharing...) to design an efficient cache-aware scheduling algorithm to achieve efficient execution through improving spatial and temporal data locality and minimizing communication overhead. Since all this information become available just after the HTGG construction, task-processor mapping can be performed before execution reducing significantly the importance of the scheduling algorithm complexity since no overhead will be introduced when executing the constructs. Task-data dependencies can be also exploited to build a data-dependency graph automatically at run-time then translate it into a HTGG. This may allow us to avoid explicit specification parallelism through the keywords “parallel” and “sequential”. Instead, programmer can gives simply a tasks sequence which will be parallelized transparently at run-time using the data-dependency graph..

References

[1] G. Blake, R. G. Dreslinski and T. Mudge, “A Survey of Multicore Processors”, IEEE Signal Processing , vol. 26, n. 6, pp. 26-37 November 2009

[2] L. J. Karam, I. Alkamal, Alan Gatherer, G. A. Frantz, D. V. Anderson and B. L. Evans, “Trends in Multicore DSP platforms”, IEEE Signal Processing , vol. 26, n. 6, pp 38-49, November 2009

[3] W. Wolf, “Multiprocessor System-on-Chip Technology”, IEEE Signal Processing vol. 26, n. 6, November 2009

[4] H.Park, H. Oh and S. Ha “Multiprocessor SoC Design Methods and Tools”, IEEE Signal Processing vol. 26, n. 6, November 2009

[5] Seung-Jai Min, Costin Iancu, and Katherine Yelick, “Hierarchical Work Stealing on Manycore Clusters”, Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11), Oct 2011

[6] M. D. McCool, "Structured Parallel Programming with Deterministic Patterns", HotPar'10 Proceedings of the 2nd USENIX conference on Hot topics in parallelism, 2010

[7] Intel Corporation, Threading Building Blocks, Tutorial rev 1.6, “http://www.threadingbuildingblocks.*org*”, 2007

[8] Cilk++ Programmer's Guides, Cilk Art, Lexington, MA, Mar. 16, 2009

[9] M. Aldinucci and M. Danelutto, “Skeleton-based parallel programming: Functional and parallel semantics in a single shot”, Comput. Lang. Syst. Struct., 33(3-4). 2007, pp. 179-192

[10] M. Cole, “ Algorithmic Skeletons: structured management of parallel computations”, Pitman/MIT Press, 1989

[11] M. Cole, “Bringing Skeleton out of the closet: a pragmatic manifesto for skeletal parallel programming”, Parallel Computing , 30(3), pp. 389-406, March 2004

[12] J. Lebak, J. Kepner, H. Hoffman and E. Rutledge, “Parallel VSIPL++: An open standard software library for high-performance parallel signal processing” , Proc. IEEE, vol 93, no. 2, pp. 313-330, 2005

[13] C. Bienia, S. Kumar, J. P. Singh and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications”, Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, October 2008.

[14] H. Kim, E. Rutledge, S. Sacco, S. Mohindra, M. Marzilli, J. Kepner, R. Haney, J. Daly, N. Bliss, MIT Lincoln Lab., Lexington, MA, “PVTOL: Providing Productivity, Performance and Portability to DoD Signal Processing Applications on Multicore Processors”, in Proc. High Perfprmance Computing Modernization Program Users Group Conf. 2008, Seattle, WA, July 2008, pp. 327-333

[15] S. Mohindra, J. Daly, R. Haney, and G. Schrader, “Task and conduit framework for multi-core systems”, in Proc. High Performance Computing Modernization Program Users Group Conf., Seattle, WA, July 2008, pp. 506-513

[16] Horacio González-Vélez and Mario Leyton "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers" Software: Practice and Experience Volume 40, Issue 12, pages 1135-1160, November/December 2010.

[17] Mario Leyton, Jose M. Piquer. "Skandium: Multi-core Programming with algorithmic skeletons", IEEE Euro-micro PDP 2010.

[18] Hahn Kim and Robert Bond, “Multicore Software Technologies”, IEEE Signal Processing, vol. 26, no. 6, pp. 80-89

[19] Luis Mura E Silva and Rajkumar Buyya, “Parallel Programming Models and Paradigms”, Citeseer Cluster Computing, vol. 2, 1999, pp. 4-27

[20] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally and P. Hanrahan. “Sequoia: Programming The Memory Hierarchy” in Proc. Supercomputing 2006, Tampa Bay, FL, Nov. 2006

[21] J. Hoberock and N. Bell, “Thrust”, http://code.google.com/p/thrust/

[22] J. Kepner and J. Lebak, “Software technologies for high-performance parallel signal processing”, Lincoln Lab. J., vol. 14, no. 2, pp. 181-198, 2003

[23] H. Singh, “Introspective C++”, Thesis, Virginia Polytechnic Institute, 2004

[24] J. Koskinen, “Meta-programming in C++”, March 9, 2004

[25] A. Munshi, “The OpenCL specification version 1.1”, Khronos Group, January 6, 2011

[26] “NVIDIA CUDA Programming Guide Version 2.2.1”, NVIDIA Santa Clara, CA, May 26, 2009

[27] H.Vandierendonck, P. Pratikakis and D. S. Nikolopoulos, “Parallel Programming of General-Purpose Programs Using Task-Based Programming Models”, Proceeding HotPar'11 Proceedings of the 3rd USENIX conference on Hot topic in parallelism USENIX Association Berkeley, CA, USA 2011

[28] D. Lin, X. Huang, Q. Nguyen, J. Blackburn, C. Rodrigues, T. Huang, M. N. Do, S. J. Patel and W. W. Hwu, “The Parallelization Of Video Processing”, IEEE Signal Processing , vol. 26, n. 6, pp 38-49, November 2009

[29] S. T. Klein and Y. Wiseman, “Parallel Huffman Decoding”, Proceeding DCC '00 Proceedings of the Conference on Data Compression IEEE Computer Society Washington, DC, USA 2000

[30] Northrop Grumman Corporation - IT Solutions. CLOC - Count Lines Of Code v1.53, http://cloc.sourceforge.net

[31] C. Teijeiro, G. L. Taboada, J. Tourino, B. B. Fraguela, R. Doallo, D. A. Mallon, A. Gomez, J. C. Mourino and B. Wibecan, “Evaluation of UPC Programmability using class room”, Proc. of the 3rd Conf. on Partitioned Global Address Space Programing Models ACM, New York, USA 2009

[32] PARSEC Benchmark Suite v2.1, “http://parsec.cs.princeton.edu/”

[33] Microsoft Task Parallel Library, “http://msdn.microsoft.com/en-us/library/dd537608.aspx”