VtkSMP: Task-based Parallel Operators for Accelerating VTK Filters

Mathias Ettinger, François Broquedis, Thierry Gautier, Stéphane Ploix, Bruno Raffin

VtkSMP: Task-based Parallel Operators for VTK Filters

Mathias Ettinger, François Broquedis, Thierry Gautier, Stephane Ploix, Bruno Raffin

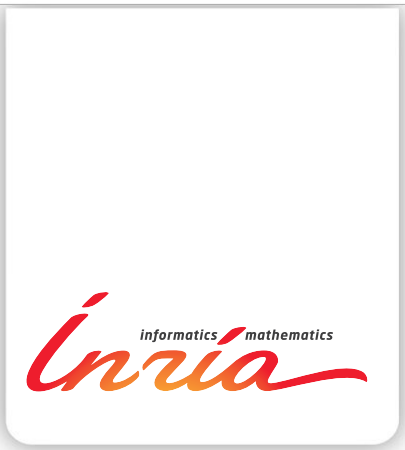
RESEARCH REPORT

N° 8245

February 2012

Project-Team Moais

ISSN 0249-6399 ISRN INRIA/RR--8245--FR+ENG



VtkSMP: Task-based Parallel Operators for VTK Filters

Mathias Ettinger*, François Broquedis†, Thierry Gautier*, Stephane Ploix‡, Bruno Raffin*

Project-Team Moais

Research Report n° 8245 — version 2 — initial version February 2012 — revised version December 2014 — 21 pages

Abstract: NUMA nodes are potentially powerful but taking benefit of their capabilities is challenging due to their architecture (multiple computing cores, advanced memory hierarchy). They are nonetheless one of the key components to enable processing the ever growing amount of data produced by scientific simulations.
In this paper we study the parallelization of patterns commonly used in VTK algorithms and propose a new multi-threaded plugin for VTK that eases the development of parallel multi-core VTK filters. We specifically focus on task-based approaches and show that with a limited code refactoring effort we can take advantage of NUMA node capabilities. We experiment our patterns on a transform filter, base isosurface extraction filter and a min/max tree accelerated isosurface extraction. We support 3 programming environments, OpenMP, Intel TBB and X-KAAPI, and propose different algorithmic refinements according to the capabilities of the target environment. Results show that we can speed execution up to 30 times on a 48-core machine.

Key-words: parallelism, parallel runtimes, work stealing, scientific visualization, VTK.

This work has been partly funded by EDF

* Inria
† Grenoble INP
‡ EDF

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

VtkSMP : un module parallèle pour VTK à destination des architectures multi-cœurs

Résumé : Les architectures NUMA multi cœurs sont potentiellement très puissantes, mais les exploiter pleinnement reste difficile étant donné leur complexité (grand nombre de cœurs, hiérarchie mémoire profonde). Un usage efficace des ces ressources est cependant impératif pour pouvoir traiter la masse toujours plus importante de données produites par les simulations numériques.

Dans ce papier, nous étudions les différentes environements de parallélisation qui ont été développés en visualisation scientifique, et tout particulièrement autour de la plateforme VTK. En complément des solutions existantes nous proposons une approche s'appuyant sur la programmation par tâches et le mécanisme d'équilibrage de charge par vol de tâches pour la parallélisation de filtres VTK. Nous expérimentons plusieurs schémas sur les filtres de transformation locale, d'extraction d'iso-surface directe et d'une version optimisée reposant sur une structure accélératrice arborescente de type min/max. Nous évaluons les performances avec 3 environements, OpenMP, Intel TBB et X-KAAPI. Les résultats montrent que l'on peut obtenir des accélérations significatives sur une machine à 48 cœurs.

Mots-clés : Programmation parallèle, vol de tâches, visualisation scientifique, VTK.

Contents

1 Introduction

The size of data produced by scientific simulations is growing at a steep rate. Postprocessing tools, including scientific visualization libraries, are urged to evolve accordingly to cope with these datasets.

Simple PCs as well as large supercomputers are today built around multi-core processor architectures, and future architectures are expected to host many-core processors, i.e. processors with hundreds of cores. Taking advantage of this processing power requires a core-level parallelization.

Though many parallel algorithms have been proposed to supplement sequential ones, many commonly used libraries are still not supporting efficient multi-core parallel executions. The VTK scientific visualization library is one of them. Beside the important effort required to revisit a large sequential code base, the lack of a standard parallel programming environment probably hampered parallelization efforts. Efficiently supporting large scale parallelism is a major challenge today [CGS+13].

The challenge is to provide a programming model that enables the programmer to swiftly shift from its sequential programming habits to parallel ones while enabling efficient executions. Task parallelism based on the MPI standard, is classically used for large parallel executions. But on shared memory machines it can lead to data replications that affect memory footprint and performance. GPUs can be very efficient for high throughput data parallelism. But even if some task parallelism is enabled at the

block level, it is often quite cumbersome to exploit. As a result, efficient implementations of irregular applications are difficult. The associated programming environments like Cuda or OpenCL are close to the GPU specific architecture, requiring to deeply rewrite sequential codes for GPU executions.

Work-stealing scheduling associated with task based programming is today emerging as sound approach for parallel shared memory programming. The user expresses potential parallelism though tasks, the scheduler taking care of balancing the workload between the participating compute resources. Parallelism is eventually extracted only if idle computing resources are available to steal part of these taks from busy resources. Environments like Cilk, TBB or KAAPI are based on this model. The OpenMP standard, Initially focused on loop parallelization, also support tasks (with dependencies since OpenMP 4), and implementation based on work stealing runtime provide high performance even at large core count.

In this paper we explore the potential of task based programming for VTK scientific visualization algorithms. This paper is an extended and updated version of the initial EGPGV 2013 paper [EBG+13]. Driven by genericity and code reusability, we identify some high-level parallel operators that can replace sequential patterns in VTK filter code. We study 3 different parallelization patterns. The first one targets loops with independent iterations (*foreach* loop) that produce independent data chunks of known size that can be directly written in a global data structure without concurrency-related issues. Next, we look at foreach loops that produce data with unknown memory footprint. A merge of partial results can thus be necessary to produce a compact data structure. Eventually, we propose to parallelize a tree traversal, a common accelerated data structure that shows an irregular access pattern. The abstraction level proposed enable to select different runtimes, like KAAPI, TBB or OpenMP, at compile time. Experiments performed on 48 core machine show the potential of such approach with different runtimes

This work led to the development of a new module, named *vtkFiltersSMP*, integrated into VTK version 6. Technical details are available through following wiki `http://www.vtk.org/Wiki/VTK/VTK_SMP`.

Section 2 discuss existing approaches. Sections **??** reminds the base concepts of work stealing and taks based programming. Our approach is presented in section 4, with experimental results in section 5. A discussion (sections 6) and conclusion (section 7) closes this paper.

2 Related Work

Message passing as popularized by the MPI standard, is a classical parallel programming model well adapted for distributed memory machines. It however requires to explicitly manage data distribution across processes. For stencil like algorithms needing a visibility of the state of neighbors, data on the border of the partitioned domains are duplicated to reduce the data exchange overhead. These ghost cells increase the complexity of the code and the memory usage, while on a shared memory system these data duplication could be avoided. Ahrens *et al.* [ABM+01] relied on MPI to propose a two level parallelization of VTK applications. Given that the input data can be partitioned (using ghost cells if necessary), their model supports data parallelism through a duplication of the processing pipeline. Additionally, when supported by the filters and the data, data blocks can be streamed down the pipeline, enabling concurrent executions of the different stages. It however does not support dynamics load balancing, relying on the user to define the partitioning policy.

Hyperflow [VOS+10, VOC+12] adopts similar goals but with a more modern context. They propose a thread level streaming and pipeline duplication strategy, thus supporting shared memory architectures. This thread level parallelization enables to avoid ghost cells, given that the necessary protection steps are taken to avoid race conditions. If the programmer provides a GPU implementation for some filters, the Hyperflow runtime is also able to defer computation to the GPUs available on the machine.

Similarly an early approach for shared memory machines have been proposed in [ALS+00]. The authors propose a thread based interface for VTK, hiding some details of thread handling. But directly

programming at a thread level is today recognized as a low level approach that is error-prone [Lee06].

VTK has been initially developed based on a sequential execution model, leading to intrinsic limitations when facing the parallelization at large scale. For instance the visualization pipeline is built by assembling filters. Pipe-line execution is under the control of executives. This abstraction level brings modularity to the application, frees the programmer from scheduling issues, and enables VTK runtime to make smart decisions for optimizing performance. However this model becomes too complex when parallelization at large scale must be enabled. Hyperflow developed a specific executive to enable more parallelism at the pipe-line level.

Some recent initiatives explored alternative models for scientific visualization frameworks designed from the ground for large scale parallelism. These frameworks are known as the VTK-m frameworks. The collective paper [SMM+12] presents an overview of these different approaches. We detail them in the following. Most of these frameworks are somehow interoperable with VTK.

Piston [StLA12] proposes a programming framework based on vector operators (data parallelism) as laid out in Guy Blelloch's seminal work [Ble90]. This approach requires to revisit all algorithms from this data parallel point of view. If data parallelism fits well several scientific visualization algorithms, it becomes more challenging as the level of regularity decreases. This model is well adapted to generate kernels for GPUs. Folding data parallel algorithms onto MIMD multi-core architectures is often easy, given that we do not attempts too aggressively to suppress the numerous synchronization barriers associated with data parallel model. For that purpose Piston is developed on top of the Thrust library [thr12]. Thrust offer parallel versions of operators on STL like vectors and list data structures for executions on GPUs through Cuda, but also on multi-core CPUs with OpenMP and Intel TBB. For performance purpose, Piston relies on simple sequences of operations and does not propose any specific high level model for ensuring some higher level of modularity when composing different opération as VTK does.

The Dax toolkit [dax, MAGM11] also adopts the data parallel model and abandons the visualization pipeline *à la* VTK in favor of a simple sequence of worklets calls. A worklet is a stateless sequential code to be executed on small elementary data. A given worklet is executed in parallel on all elementary data that constitute the input data set. A worklet does not directly manage memory allocation and layout, but rather accesses data through an abstraction layer, defering memory issues to the framework. However worklets derive from base worklets depending on the type of data access pattern and output generated. For instance `WorkletMapField` *performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location.* `WorkletGenerateTopology` *generates a cell connectivity. When invoked, the dispatcher is given an array containing the number of output cells derived from each input cell. Each invocation produces exactly one cell.* Dax was initially developed for the GPU, but now supports OpenMP via Thrust as well as Intel TBB. Also notice that in the initial paper [MAGM11] Dax relied on a data-flow model where worklets were assembled into a direct acyclic graph, the framework having the possibility to perform some optimizations when scheduling the different processing steps. But this approach was abandoned in favor of an explicit sequential pipe-line under the control of the user without further automatic optimization. Both Dax and Piston find that the traditional VTK pipe-line tends to prevent an efficient execution of the different steps on massively parallel architectures.

EAVL [eav, MAPS12] similarly focuses on data parallel operators. EAVL adopts a functor/iterator model where a functor encodes an operation and an iterator defines an execution pattern. EAVL main originality comes from the data model. Data layouts stay under the explicit control of the user, but it departs from the traditional VTK model by offering much more flexibility. This model is based on sets of coordinates objects and sets of cell objects. A separate logical structure object describes the basis of arrangement for the points or cells. Additionally a set of fields can be associated with some parts of the mesh. The goal of this data model is to enable to design memory efficient data structures according to the target architecture as well as to support non traditional data sets. EAVL also supports MPI parallelization and parallel file readers for formats like Silo, BOV and NetCDF. On these aspects though the information

we found did not enabled us to have a clear understanding of the approach proposed. Current EAVL implementation supports CUDA and OpenMP backends.

DIY [PRG+11] ("Do-It-Yourself" Parallel Analysis) is also part of the VTK-m suite, but takes a different approach. DIY focuses on large scale distributed memory parallelization relying on MPI as communication library. DIY does not address the fine grain thread parallelism that may be required at a node level. The goal of DIY is to provide patterns of data movements for supporting the parallel implementation of analysis and visualization algorithms. The user work in a familiar C/C++/Fortran environment with a MPI like view of its computing infrastructure. The data sets are partitioned by DIY (using the Zoltan library) into blocks respecting the requirements expressed by the user (need for ghost cells for in instance), DIY taking care of actually distributing these blocks to processes and performing some dynamics load balancing if required. Thus, the user does not need to explicitly map blocks to resources. He expresses his algorithm through operations on blocks and request communications between blocks rather than processes. DIY provides several collective block level communication routines like `DIY_Exchange_neighbors` or `DIY_Merge_blocks`. The implementation relying on modern algorithms like radix-k for tree-based merging. Notice that to identify the required communication pattern, Peterka et al. provide in this paper an analysis of the different classes of visualization algorithms in terms of data movements. Similarly the paper by Moreland et al. [MGMM13] propose a classification of visualization algorithms but with a thread level parallelization in mind.

3 The Work-Stealing Paradigm

Work-stealing is emerging as an alternative of choice for shared memory programming. Relying on a dynamic load balancing runtime, these libraries can cope with irregular applications and unsteady core availability. They also propose a programming model based on tasks and not threads to delimit the potential parallelism. The runtime thus takes care of distributing these tasks on the processing cores, relying on low overhead mechanisms, like a distributed task heap.

Let us remind the base concepts of task based programming and work stealing. The programmer expresses the potentiel parallelism in his code by delimiting tasks that can be executed concurrently. Tasks can be created dynamically. Each processing core maintains a list of tasks. When a core generates a task, it pushes it in its local list. A task of this list becomes ready to be executed once synchronization constraints have been resolved. When a core becomes idle (no local task left), it randomly selects another target core and steals part of the tasks ready to be executed in this core's task list. If no task can be stolen, an other victim is targeted. This scheduling algorithm has proven performance [BL99]. Today several parallel programming environments are task based like Cilk [FLR98], TBB [Rei07] or X-KAAPI [GBP07, LMDG11]. They also come with higher level constructions easing the parallelisation of common patterns (loop with independent iterations for instance). Their implementations ensure high performance on multi-core processors and shared memory machines.

The OpenMP standard, initially focused on loop parallelization, support tasks without dependencies since version 3 and with dependencies since version 4. Several OpenMP runtimes rely on work stealing schedulers. The OpenMP [Ope13] standard relies on code annotations that the programmer uses to identify possible sources of parallelism, e.g. indicating when the iterations of a loop are independent and can thus be executed concurrently. Then the compiler and runtime can extract parallelism. OpenMP 4.0 has also taken care of heterogeneous architectures, by adding the `target` construct, which enable offloading to accelerators, such as GPU or Xeon Phi. From a scheduling point-of-view, OpenMP gives the choice to use both static and dynamic scheduling, and also the possibility to use the runtime's default scheduler. Current tasks implementations are not totally mature but quite encouraging, as shown in [VBB+14]. OpenMP compatibility is also provided by some work-stealing libraries, such as X-KAAPI [BGD12] or StarPU [dt] which support OpenMP 3.1 and above.

Some scientific visualization frameworks like Dax or Piston indirectly rely on work stealing, but do not expose tasks in their programming model that fits to a data parallel model. Some papers like [TDR10] or [Wal12] have intended to fully take benefit of the task model for fine tuning some visualization algorithms.

Work-stealing has also been experimented for balancing the work load in between the different cores of a GPU for some specific algorithms like octree construction for particle sorting [CT08]. But the GPU not being fully programmable (scheduler in particular), no full fledged task based programming model and runtime has been developed on such architecture as far as we know. This is not the case on Xeon Phi accelerator that support Cilk and TBB for instance.

4 Parallelization of VTK Algorithms

Our goal is to identify patterns of code within VTK filters that appears to be good candidates for parallelization. The pattern we focused on are described in the following sections.

The first and most simple one is the loop with independent iterations. VTK filters mainly iterate over cells and/or points to compute their output. Since the computation for an iteration does not need results from previous ones, parallelization of this pattern is pretty straightforward. Indeed, each iteration can be seen as an independent task to compute. The runtime system is then responsible for efficiently schedule these tasks over all the available computing resources. We implement this pattern using a *ForEach* construct. This way, algorithms that can predict the size of their outputs can directly rely on *ForEach* constructs without any further effort.

Some filters produce an amount of output data that cannot be determined beforehand. In such cases we use a different approach : the strategy for these algorithms is to have each thread to perform its own computation in a private space, called Thread Local Storage, followed by a parallel merge operation. Since this operation does not exist in a sequential execution, it has to be very efficient in order to limit parallel overhead.

The last pattern we study in this paper concerns acceleration data structures. Such structures are often implemented as trees (binary trees, octrees, kD-trees...). We experimented a parallelization of a generic tree traversal. Given a few set of operations that a tree needs to implement, we guaranty its traversal in parallel with respect to its sequential depth first execution and branch cutting capabilities.

4.1 ForEach

Given some requirements, iteration over independent loop is very easy to parallelize. Indeed, using n as the size of the loop, one can see such loop as n independent tasks. Thus giving each core $\frac{n}{\text{numberofcores}}$ elements to compute is the easiest way to parallelize this pattern. However this approach can lead to load imbalance issues. If a few iterations take much more time than the others, some cores may finish their computation before the others and have to wait for them. Dynamic scheduling and work-stealing techniques can efficiently reduce these problems.

But giving work to each processor is not the only thing to care when dealing with independent loops. One has to make sure to allocate enough memory for output data structures and to use thread-safe methods while computing each element. Moreover, handling writing offset may be a non-trivial task. Thus these aspects are left to the user. Our plugin guaranties to divide the loop's iteration range the best way according to the chosen runtime.

Our plugin provides the *vtkSMPTools::For* operator. As for Intel TBB's *tbb::parallel_for*, one has to move the sequential body of the loop into a freshly build class (*e.g.* figure 1). It has the advantage to separate the algorithm (the pattern) and the computation. It also exhibits pieces of code that need to be carefully managed (memory allocation, thread safe methods, etc.). Writing the class is a matter of

```
1  struct VcsModificatorFunctor {
2    vtkDataArray* inVcs;
3    vtkDataArray* outVcs;
4    double (*matrix)[4];
5    void operator () ( vtkIdType begin,
         vtkIdType end )
6    {
7      double vec[3];
8      for( vtkIdType id = begin; id < end; id++)
9      {
10       inVcs->GetTuple( id, vec );
11       vtkSMPTransformVector( matrix, vec, vec
           );
12       outVcs->SetTuple( id, vec );
13     }
14   }
15 };
```

```
16
17 void vtkSMPTransform::TransformVectors(
18 vtkDataArray *inNms,
19 vtkDataArray *outNms)
20 {
21   vtkIdType n = inNms->GetNumberOfTuples();
22   this->Update();
23
24   VcsModificator myvectorsmodificator;
25   myvectorsmodificator->inVcs = inNms;
26   myvectorsmodificator->outVcs = outNms;
27   myvectorsmodificator->matrix =
         this->Matrix->Element;
28
29   vtkSMPTools::For( 0, n,
         myvectorsmodificator );
30 }
```

Figure 1: Implementation of the parallel *vtkTransform::TransformVectors(...)*.

checking the sequential behaviour. To call the operator one only needs the range of the computation and an instance of the aforementioned class.

This operator can be found in the `vtkCommonCore` module as well as every abstraction layer (in specific directories) for each supported runtime.

4.1.1 Grain Tuning

When the operator's work is very regular, we face a scalability issue due to steals. Indeed, each steal adds a little overhead to the computation, and generates transfers on the memory bus to retrieve suitable data. The number of steals can grow significantly as we get closer to the end of the computation.

These steals grabbing a small work load tend to induce more overhead than computation speed-up. To avoid the performance drops occurring with this behavior, the work-stealing scheduler defines a threshold representing the minimal range for a computation, *i.e.* the grain. The problem is thus to set the grain to its optimal value : if the grain is too coarse, load balancing becomes ineffective, and if the grain is too fine, load balancing induces too much overhead. Previous studies [PP95] showed that the maximum amount of parallelism reachable is when the grain is $\Theta(\sqrt{n})$. Indeed the number of tasks created is, at most, $\frac{n}{\text{grain}}$ and these tasks are limited to a range containing at least *grain* iterations. Thus the critical path (*i.e.* the sequential part of the computation) is $\Theta(\text{grain} + \frac{n}{\text{grain}})$. The optimum of this function appears when the grain reaches $\sqrt{n}$. Experiments confirm that a $\sqrt{n}$ grain leads to the best performances.

4.2 Thread Local Storage and Merge

Failing to meet the requirements for the *ForEach* pattern is often due to the impossibility of knowing the size of the output without actually performing the computation. For such algorithms, and those who fails at using thread-safe methods, there is still a possibility to use the *ForEach*.

The idea behind this second approach is to use private output data structures per core. This way, parallel execution is closer to the sequential one and restrictions for thread-safe methods are less strict. There is some drawbacks :

- these data structures need to be initialized efficiently;
- each of these data structures will contain partial results that will need to be merged afterward.

4.2.1 Initialization

These templated structures, called Thread Local Storage (TLS), provide a `Local()` method that creates a new object if needed and ensures that each thread accesses its own separate object. These structures are intended to be used within parallel operators to ensure that each thread writes data in separate memory space. Due to some runtimes limitations, using such structures outside of a parallel operator might not give the expected results.

We provide two classes that implements TLS : `vtkSMPThreadLocal` and `vtkSMPThreadLocalObject`. The former is intended to manage generic data and the latter is designed for `vtkObject`s. It takes care of reference counting, so one does not have to register the objects if he does not need to access them when the TLS is destroyed.

For performance reasons, objects for a particular thread within a TLS are not constructed until they are first accessed. There is no built-in way to tell if an access is the first one or not. Therefore if a code relies on initializing a per-thread data structure once, and using it assuming it is initialized, then it will have to keep track of whether or not such structure is initialized. Since it is so common to do so, functors can embed an `Init` and a `Reduce` method. If both methods exists, the `Init` one is executed once per thread before any computation, and the `Reduce` one is executed once the functor iterates over each element of the computation. Thus, with a well written `Init` method, the `operator()` can assume that every data structure used has been initialized properly.

Such capabilities are also useful in case of severe load from other jobs. When cores are so loaded that they could not perform a steal, they do not attempt to initialize their TLS, leading to time sparing.

Code for these TLS is also found in the `vtkCommonCore` module.

4.2.2 Merge

Once initialization is done and actual computation performs in parallel, each TLS contains a part of the final expected result. Since this result is often a mesh, our plugin provides capabilities to fuse partial meshes into a single one. This is performed in parallel for efficiency reasons.

This operation cannot mimic the behaviour of *vtkAppendFilter*, which sticks together the points and cells of several meshes, because of duplicates points at the border of each partial meshes. If we kept all those points in the output of the filter, the mesh would lose its manifoldness (if any). Results would give strange outputs when filtered through decimation or subdivision, for example.

To track and remove duplicates points of partial meshes, we used VTK built-in *Locators*. We extended the behaviour of *vtkMergePoints* to support parallel operations. Basically, this locator keeps track of existent points thanks to a spatially-guided hash table. The bounding box of the resulting mesh is regularly divided into a 3D grid and each voxel of this grid is mapped to an entry in the table. Each entry stores the indices of the points that belongs to this voxel. Knowing the index of the entry associated to a point requires simple mathematics involving point and bounding box coordinates.

The simplest use of our locator for a parallel merge of all partial meshes is, for each thread, to insert their points one by one into the locator. This solution obviously produces wrong results if no care is taken. Our solution is to use an hash table of mutexes, in addition to the one dedicated to points ids, as shown in figure 2. Each insertion of point is thus preceded by a lock operation. Since conflicts only occurs for points that lay on the boundary of a mesh, overheads are still low. However, memory footprint is twice as important as the sequential object.

Using only one locator in the case of filters that already need one per thread is not highly productive. So we used the idea from flat combining [HIST10] to benefit from these arrays of buckets[1]. With an iteration over each voxel, one thread can merge all equivalent buckets in a row. Locks that were protecting voxels from concurrent accesses are thus no longer needed. The only requirement that is still present is the

[1] A bucket represents a *vtkIdList* in a voxel.

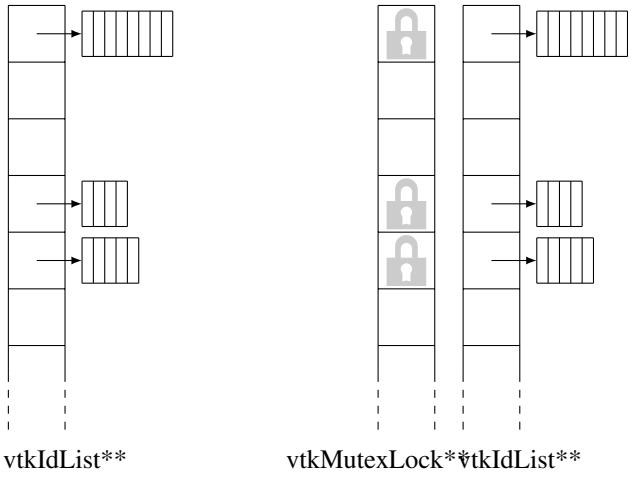


Figure 2: Inner representation of *vtkMergePoints* (left) and our first implementation of *vtkSMPMerge-Points* (right). Each *vtkIdList* is designated as bucket.

need of an atomic increment on the number of points in the output *vtkPoints*. But this approach imposes a constraint to our framework : a thread must be able to retrieve data from any other. Moreover our case-study shows that more than 90% of the buckets are empty. Taking care of these two particularities leads to initiating the merge of voxel's buckets only by a thread which actually has data in these buckets. To do so, each thread looks for the first non-empty bucket it owns. If no other thread already merged the buckets associated to the same voxel, it retrieves the data associated to this voxel from the other threads, and performs the merge in the output data structures. Otherwise it discards this bucket and look for the next one which contains data.

Our framework provides both merge capabilities. Filters that use a locator to produce their output points usually end up using thread local locator while parallelized. In such cases, the last technique can use the locator to perform point insertion bucket by bucket. Otherwise the function build a new locator and uses it as described for the first technique.

4.3 Acceleration Data Structures

Due to their inner complexity, several algorithms have an accelerated version. It mainly consists of the addition of a data structure that will ease computation. The purpose of this structure is to decrease the amount of data that needs to be analyzed. They are not automatically used because they may introduce overhead for their initialization, or increase the memory footprint. But what makes them efficient is that they speed up the computation of the algorithm each time they are used. Best results are thus obtained when data are requested several times, with different values.

In the case of our study we focused on trees, which are common acceleration data structures in the visualization field. Their aim is to reduce the size of the iteration over the end-elements (cells, pixels, objects, 3D space) by cutting unproductive content.

Use of trees can be accelerated at two different steps. The first one is when building the tree. But since this step is highly dependent on the type of the tree and its usage, it might be a per-implementation study. The second one is during the tree traversal. This one is much more easy to provide as a generic operator, since the traversal can be seen as a sequence of node indices to explore.

One cannot simply build this sequence and apply our *ForEach* on it : indeed each node must be treated after its ancestor has finished its computation. And in most cases, the computation of a node may result

in the choice of not going further in the exploration of its descendants.

A simple solution is to use tasks spawning on each node. That is :

- the computation of a node is considered as a task;
- each node creates a task per descendant that need to be explored;
- these tasks are spawned and the runtime is then responsible of an efficient scheduling policy;
- the tree traversal starts with the spawn of the root task.

First levels are quite slow to traverse since there are fewer tasks than available resources, but the number of tasks is rapidly increasing and they can be efficiently scheduled on the whole machine.

4.3.1 Adaptive tasks

Creating much more tasks than available computational resources is a good way to simplify the scheduling policy and balance the work load. But it has a cost : creating and managing tasks may become costly compared to the actual computation of a task, especially in visualization where nodes often perform only a few comparisons to decide whether or not they should spawn their descendants.

The answer of some work-stealing runtimes such as TBB or X-KAAPI is to provide a way for creating adaptive tasks. Such tasks have the particularity to be splittable on demand by an idle resource, thus limiting the number of created tasks to the number of cores actually executing them.

Using these tasks, we propose an adaptive tree traversal which behaves as follow :

- Create a task that traverses the entire tree and start the (sequential) execution;
- On a demand for splitting the task, find the top-most untreated node that would be the last computed in sequential depth-first order;
- Consider this node and its sub-tree as treated and create a task that traverse this sub-tree.

Newly created tasks behave like a fresh tree and their traversal follows the same rules as above. Thus they can also be split if a processor is idle. In this scheme, the number of splits that occurs (*i.e.* the total number of tasks created) is $\Theta(\text{number of processors})$ which may be much less than the number of traversed nodes.

5 Experimental Results

Following sections provides results for our operators based on the parallelization of two VTK filters.

5.1 Evaluation Platform

We conducted our experiments on a CC-NUMA machine made of AMD Magny Cours processors holding 12 cores each. Each processor is divided into two chips of 6 cores each. The machine has 4 sockets for a total of 48 cores. Figure 3 shows the internal organization of a socket obtained thanks to the HWLOC [BCOM+10] library. Each socket, or node, is made of two 6-core processors. Each core has access to 64 KB of L1 cache, 512 KB of L2 cache and 5 MB of L3 cache. The first two caches are private while the third one is shared between the 6 cores of a chip from a Magny Cours processor. A 32 GB memory bank is attached to each processor of this platform for a total of 256 GB of main memory. We will refer to this configuration as **AMD48** in the following of this section.

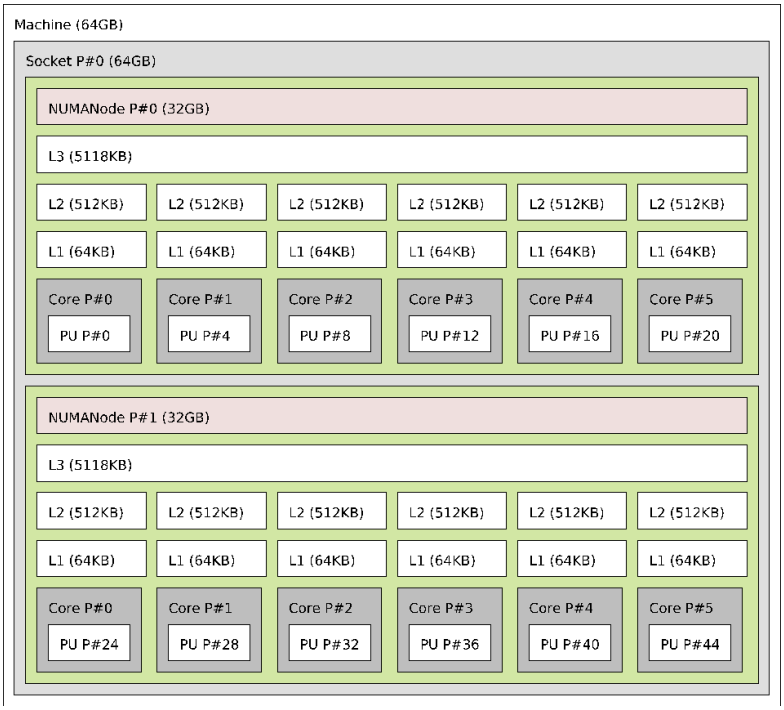


Figure 3: A socket from the AMD48 evaluation platform.

Our input dataset comes from the Stanford 3D Scanning Repository[2]. It is the Lucy mesh that contains 28 M cells (triangles) for 14 M points. We load it as a *vtkPolyData* object and provide it with point normals to test specific behaviour of *vtkTransformFilter*.

5.2 vtkTransformFilter

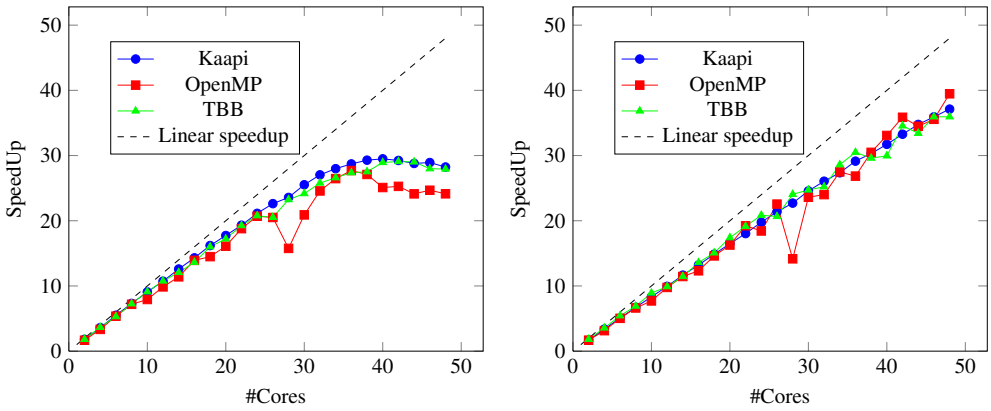
An example of VTK algorithm that performs a loop with independent iterations over cells and/or points is the *vtkTransformFilter*. It applies scales, translations and rotations on a mesh. The computation is performed by a *vtkTransform* that contains the description of the transformations to be applied. The *vtkTransform* is supplied to the *vtkTransformFilter*, and iterates on the input data to produce the output.

We modified *vtkTransformFilter* to perform the needed memory management before every computation, and we built the *vtkSMPTransform* class which mimic the behaviour of the *vtkTransform*, except that the sequential loop was turned into a parallel ForEach.

Figure 4a shows the performance obtained on our AMD48 machine. AMD48 is a NUMA machine with shared memory, so we additionally studied the influence of data placement. Since memory pages of output data are touched only after each point or cell transform computation, we put two *vtkTransformFilter* in a row in our visualization benchmark. As a consequence, the first filter load data from a unique memory node, leading to memory contention, and the second one has the memory pages of its input data interleaved on each memory node thanks to the first filter. Thus the second filter makes a better usage of memory bandwidth. Results are shown in figure 4b.

As expected, mapping the data close to their computing core enhances speed-up and drops memory contention. Results also show that work-stealing runtimes are at least as efficient as OpenMP for highly

[2] `http://graphics.stanford.edu/data/3Dscanrep/`



(a) First filter (input data on a unique NUMA node).

(b) Second filter (input data interleaved on all NUMA nodes).

Figure 4: Speedups for execution of parallelized loops within *vtkTransformFilter*. Comparison between several runtimes of the execution of the *ForEach* loop vs. the sequential `for` loop.

regular loops — which is known to be the stomping ground of the latter. Moreover, a side effect of work-stealing runtimes is that they tend to smooth the memory contention issue.

5.3 vtkContourFilter

A widely-used filter that cannot calculate the size of its output structure before the actual computation is the *vtkContourFilter*. Its purpose is to compute one or several isosurfaces on any kind of datasets. In its current implementation, *vtkContourFilter* is mainly a switch between several specific implementations depending on the type of the input dataset. The implementation parallelized in this study is the generic one, which is applied if the input dataset does not have a specific algorithm that handles it.

The algorithm works as follows:

- for each cell in the mesh, compare the scalar values associated to the points of this cell with the isovalue being computed;
- if the isovalue is either above the maximum or below the minimum of these values, do not compute anything;
- otherwise create the part of the output mesh that correspond to the values pattern.

Since the created portion of the output mesh depends on both the scalar distribution above and below the isovalue and the topology of the cell, it is not possible to know how many points and cells[3] will be created per input cell.

The parallelized version of *vtkContourFilter* thus makes use of TLS and fuse the partial pieces of mesh. Figure 5 shows the execution of the surface creation (with merge) on our AMD48 platform. Speed-ups are computed against the sequential `for` loop execution time. Since isocontouring is an algorithm that can often face load balancing issues, in this experiment we provided a very high load-imbalanced input dataset.

To do so we computed scalars values in such a way that only a few first cells can possibly contain points whose scalar values are both above and below the isovalues. Thus using a static partitioning of

[3] One or several points, one or two lines, one or two triangles...

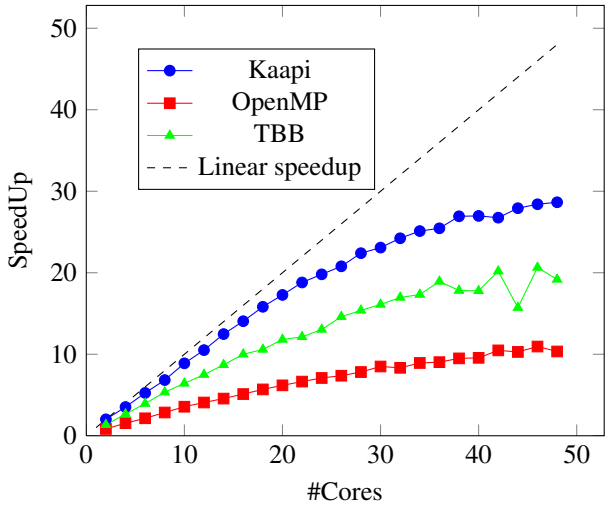


Figure 5: Speed-ups for execution of parallelized loop within *vtkContourFilter*. The difference between the runtimes is due to the high load of the first core.

the iteration range leads to the first core being the only one to create the output mesh. As expected, work-stealing runtimes have a better load balancing and outperforms the static partitioning of OpenMP. Moreover, dynamic scheduling tuned with the right chunk-size (grain) gives similar results.

5.3.1 Focus on the Merge Operator

The *vtkContourFilter* is a good place to experiment the behaviour of our *Merge* operator, since the iso-surface computation makes use of one locator per thread.

Figure 6 plots the execution time for both merge implementations. The method using one locator per thread is faster than the one using only one global locator. It also requires twice as less memory since the array of *vtkMutexLock* is not required in this case. There is one drawback though : all the parallelized filters that would make use of the *Merge* operator could not be able to use a thread local locator.

5.4 Accelerated vtkContourFilter

We tested our parallel acceleration tree with the classical min-max tree used for isosurface extraction. A min-max tree stores at each node the min and max values of all scalar values contained in its sub tree. The tree is built in a bottom-up fashion after choosing a size for leaves, i.e. the number of points associated to each leaf. If the isovalue is not included in the min-max interval of a given node, no further test is needed for all the points of the sub-tree.

As stated earlier, such data structure shows its power when reused several times on the same input. Indeed, extrema on each node do not depend on the isovalue being computed.

Results presented in figure 7 show the difference between the task spawning technique and the work-stealing one. In this experiment we computed 11 isovalues through our input mesh, with the accelerated version of the *vtkContourFilter*. The speed-up is for the parallel traversal and merge time, computed against the sequential traversal time.

Even if the overhead for the creation of one task is very low, our parallel tree traversal is slightly better. This is due to the number of tasks created, which depends on the number of computational resources and not on the size of the input.

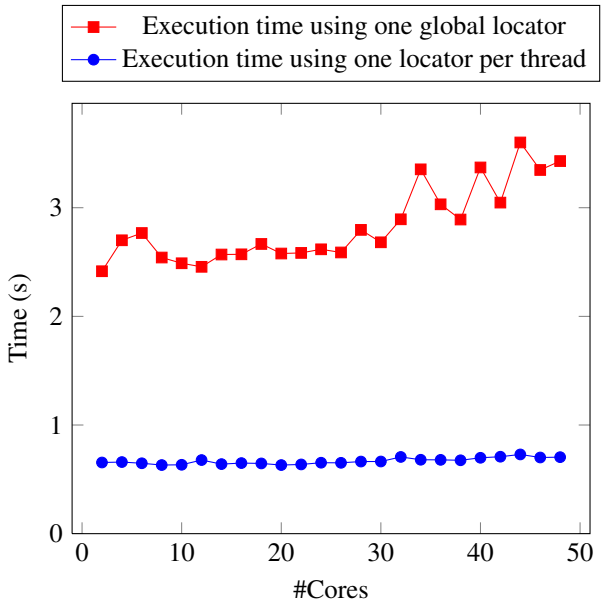


Figure 6: Comparison of our two merge implementations.

6 Discussion

6.1 The Pipe-line Model

Is the VTK pipe-line model not adapted to large scale parallelism ? We made some attempts to enable pipe-line parallelism for multi-block datasets. These blocks can safely be processed in parallel. By default, even if a filter is parallelized, it cannot start processing a block if all blocks have not been processed by the upstream filter. The filter needs to have the full data set available before to start. This creates an implicit synchronization between each filter. Our goal was to remove this synchronization to enable a filter to process a block as soon as available. A VTK feature prevented good performance though. The `vtkTimeStamp` class, used within every `vtkObject` is widely used during filters execution. Our first implementation using a global counter increment led to incorrect results. Using an atomic counter increment to solve this issue was inefficient due to the high number of operations involved in parallel. To address this issue, a solution was to deactivate the use of `vtkTimeStamp` during the executive parallel operation. But it can not be a global deactivation because some features (such as updating the bounding box of the output dataset) rely on it. The final solution was to deactivate it only for `vtkInformation` object because they were the one causing the most update on their `vtkTimeStamp`. But parallelizing `vtkCompositeDataPipeline` with this approach has an inherent weakness related to cache consistency. Each time a dataset reaches such parallel executive, it is processed, block by block, until all the data went though the filter. Not only it creates a synchronisation point that can lead to overheads if load balancing fails, but it also does not take into account the fact that each output block will be reused as an input by the next filter, ruining cache consistency.

The VTK-m approaches all went away from this model for a simpler one. One motivation is that encapsulation in opaque filters prevents an efficient control on the data movements between filters, or more precisely to correctly handle the task/data affinity. The model also tends to favor the sequence of short filters, while efficient parallelizations rather call for long parallel sequences with as few as possible synchronization, communication and sequential section phases. This is a major performance issue that

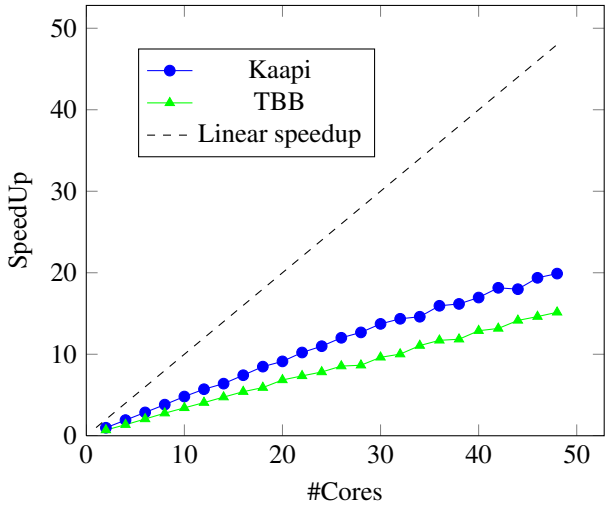


Figure 7: Execution time speed-up of *vtkContourFilter*, with our parallel tree traversal.

will very likely worsen on next generation architectures. The programming model should be designed such that the programmer be aware and able to master the main performance pitfalls.

6.2 Data Parallel versus Task Parallel Programming

The data parallel paradigm is adopted by DAX, EAVL and PISTON. The reasons are twofolds: many operations related to scientific visualization are data parallel [MGMM13]; and GPUs have demonstrated that this model could lead to high performance at a good flop/watt ratio.

We believe that task-based programming as developed for Cilk, TBB or OpenMP is a better option. First, they support all data parallel operations, but also enable to express less regular patterns with tasks that can be efficiently executed with work-stealing runtimes. One major limitation is that GPU's, as designed by NVIDIA or AMD, do not enable task based programming. The scheduler is not exposed and not programmable. The GPU processors have a double parallelism level associating a number of streaming multiprocessors that run asynchronously, each one executing synchronous threads. Thus, a task model could be supported by GPUsl as demonstrated by [CT08, TG12], but so far vendors did not follow that path.

An other point of view is to argue that next generation chips, being CPU or GPU based, will support multiple cores, each core executing vector/SIMD instructions. Even if compilation technologies have progressed to automatically fill vector units, efficient programming still needs human involvement. Programming directly with data parallelism enable to expose to the user only one level of parallelism. In opposite, task based programming needs to be complemented with vector programming inside tasks to reach high performance, as it is for instance the case with the Xeon Phi accelerator. Even if data parallelism efficiency and simplicity fade away for irregular applications (this includes many acceleration data structures used in computer graphics), wasting some compute cycles may not be so critical given that we will have an abundance of threads.

6.3 Memory Layouts and Data Movements

Often scientific visualization algorithms are data intensive. Data layout and data movements have in this case a significant impact on performance on NUMA architecture given the complexity of their memory

hierarchy. Similar issue occurs on GPUs where the user needs to explicitly manage most of the data movements between the different memory levels. Non of the VTK-m solution really addresses this problem in their model. EAVL offers a very flexible data structure model that can favor efficient mapping on the target architecture, but it does not expose some specific memory model. Work-stealing approaches, like TBB and Cilk, often consider that the underlying model is a uniform shared memory with a local cache per worker. Given the random nature of work stealing, the user does not know in advance who does what. Some attempts have been made to propose some architecture aware stealing policies [TDG+10, DBGR13] or alternative, more cache efficient, dynamics scheduling policies [BGM95]. For instance in [DBGR13] the authors propose a two level stealing policy with KAAPI where an idle worker attempts to first steal work from other workers running on the same socket, before stealing to a distant socket. But so far such ideas has not made their way into main stream programming environments.

In OpenMP, the `OMP_PLACES` environment variable gives the user the possibility to define a set of "places" corresponding to specific hardware threads. One can also use several abstract names, such as `cores` or `sockets`, if he wants places to respectively match cores or sockets on the target machine. For each parallel region, a team of threads is created to execute it, and the user can then use the `proc_bind` clause to specify how he wants the threads to be dispatched among the places (e.g. `close` to have the thread mapped as close as possible of the master thread, or `spread` to have a sparse distribution of the threads among the places).

6.4 I/Os

Through the various experiments we performed, we noticed that one bottleneck comes from reading the input data from files. VTK reader are single threaded. Given the default *first touch* policy Linux uses, all read data are thus stored in the memory bank attached to the processor that actually reads the file. The first filter operating on these data, even if efficiently parallelized, will need to have all its threads access the same memory bank, leading to some performance degradation. This effect can be mitigated by forcing a different policy (with a tool like *bind* for instance). We tried to parallelize a reader but encountered thread-safety issues. A pragmatic approach is to rely on the MPI parallelization of VTK, having one MPI process per socket and a thread-level filter parallelization operating on each socket. This brings some dynamics load balancing capabilities within each socket with the advantage of having all data local to each socket, i.e. to the local memory bank, while enabling parallel file reading between sockets.

In-situ processing brings a different perspective on this issue. In that case data are not read from disk, but are directly provided by the simulation processes running locally on the node. If some visualization algorithms are executed locally on each node, having load balancing capabilities may improve the use of the local resources shared by the simulation and the in-situ processings (given that an asynchronous approach is enabled).

7 Conclusion

We presented the basis of a framework for VTK that aims at creating parallel filters. It targets multi-cores platforms and takes advantage of reusability of sequential code. It relie on the task-based programming paradigm and on work-stealing runtimes.

This work led to the development of a new module, named *vtkFiltersSMP*, integrated into VTK version 6. Technical details are available through following wiki `http://www.vtk.org/Wiki/VTK/VTK_SMP`.

This work is concomitant with other initiatives exploring new frameworks for large scale parallel analytics and visualization like the VTK-m software suite. Most of these approaches adopt a data parallel approach, that fit well the GPU architecture. To our opinion a task-based approach is more flexible, and

today good runtimes, based on work-stealing, ensure a high performance. Such model is already fully supported on Xeon Phi accelerators. GPUs are made of asynchronous cores, but the exposed programming model does not enable to support properly and efficiently such model so far. In both cases, CPUs or GPUs, we believe that it is difficult to avoid a two level parallelization model, one for distributing work to cores and an other one to load the vector/SIMD units each core owns. Given the limited size of these vector units, the algorithms that are efficient at this scale are not the same than the one relevant at large scale.

Other issues remains, like developing a sound memory model that is not to complex for the programmer while still exposing enough key architecture features to lead to efficient programs. On this aspect the CPU and GPU differ deeply, making the development of a common layer difficult. Another issue is the distributed programming level, a world today dominated by MPI. We can underline some attempts to propose alternative models like the Swift/T [WAW+13] environment relying on a taks-based programming model with a work stealing strategy for work load distribution.

Aknowledgments

The development of VTKSMP has been partly funded by EDF R&D. This research report is an extension of the paper [EBG+13]. This extension work has been funded by CEA.

References

[ABM+01] James Ahrens, Kristi Brislawn, Ken Martin, Berk Geveci, C. Charles Law, and Michael Papka. Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications*, pages 34–41, 2001.

[ALS+00] James Ahrens, Charles Law, Will Schroeder, Ken Martin, Kitware Inc, and Michael Papka. A parallel approach for efficiently visualizing extremely large, time-varying datasets. Technical report, 2000.

[BCOM+10] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italie, February 2010.

[BGD12] François Broquedis, Thierry Gautier, and Vincent Danjean. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 102–115, Berlin, Heidelberg, 2012. Springer-Verlag.

[BGM95] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 1–12, New York, NY, USA, 1995. ACM.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[CGS+13] H. Childs, B. Geveci, W. Schroeder, J. Meredith, K. Moreland, C. Sewell, T. Kuhlen, and E.W. Bethel. Research challenges for visualization software. *Computer*, 46(5):34–42, May 2013.

[CT08] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 57–64, 2008.

[dax] Data analysis at extreme: The dax toolkit. http://www.daxtoolkit.org/.

[DBGR13] Marie Durand, François Broquedis, Thierry Gautier, and Bruno Raffin. An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines. In *International Workshop on OpenMP (IWOMP)*, volume 8122 of *Lecture Notes in Computer Science*, pages 141–155, Canberra, 2013. Springer Berlin Heidelberg.

[dt] StarPU development team. Starpu openmp runtime support.

[eav] Eavl: : The extreme-scale analysis and visualization library. https://github.com/jsmeredith/EAVL/wiki.

[EBG+13] Mathias Ettinger, François Broquedis, Thierry Gautier, Stéphane Ploix, and Bruno Raffin. VtkSMP: Task-based Parallel Operators for VTK Filters. In *Eurographics 2013 Symposium on Parallel Graphics and Visualization (EGPGV'13)*, May 2013.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, 1998.

[GBP07] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of PASCO'07*, New York, NY, USA, 2007. ACM.

[HIST10] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39:33–42, 2006.

[LMDG11] Fabien Le Mentec, Vincent Danjean, and Thierry Gautier. X-Kaapi C programming interface. Technical Report RT-0417, INRIA, 2011.

[MAGM11] K. Moreland, U. Ayachit, B. Geveci, and Kwan-Liu Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 97–104, Oct 2011.

[MAPS12] Jeremy S. Meredith, Sean Ahern, Dave Pugmire, and Robert Sisneros. Eavl: The extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 21–30. The Eurographics Association, 2012.

[MGMM13] Kenneth Moreland, Berk Geveci, Kwan-Liu Ma, and Robert Maynard. A classification of scientific visualization algorithms for massive threading. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, UltraVis '13, pages 2:1–2:10. ACM, 2013.

[Ope13] OpenMP Architecture Review Board. OpenMP application programming interface version 4.0, July 2013.

[PP95] Victor Y. Pan and Franco P. Preparata. Work-preserving speed-up of parallel matrix computations. *SIAM J. Comput*, 1995.

[PRG+11] T. Peterka, R. Ross, A. Gyulassy, V. Pascucci, W. Kendall, Han-Wei Shen, Teng-Yok Lee, and A. Chaudhuri. Scalable parallel building blocks for custom data analysis. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 105–112, Oct 2011.

[Rei07] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[SMM+12] Christopher Sewell, Jeremy Meredith, Kenneth Moreland, Tom Peterka, Dave DeMarle, Li-ta Lo, James Ahrens, Robert Maynard, and Berk Geveci. The sdav software frameworks for visualization and analysis on next-generation multi-core and many-core architectures. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 206–214. IEEE Computer Society, 2012.

[StLA12] Christopher Sewell, Li ta Lo, and James Ahrens. Piston: A portable cross-platform framework for data-parallel visualization operators. In *Eurographics Symposium on Parallel Graphics ans Visualization*, 2012.

[TDG+10] Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec, and Bruno Raffin. A Work Stealing Algorithm for Parallel Loops on Shared Cache Multicores. In *4th Workshop on Highly Parallel Processing on a Chip (HPPC)*, August 2010.

[TDR10] Marc Tchiboukdjian, Vincent Danjean, and Bruno Raffin. Cache-efficient parallel isosurface extraction for shared cache multicores. In *Eurographics Symposium on Parallel Graphics ans Visualization*, 2010.

[TG12] Julio Toss and Thierry Gautier. A new programming paradigm for gpgpu. In *Euro-Par*, pages 895–907, 2012.

[thr12] Thrust library. http://code.google.com/p/thrust/, 2012.

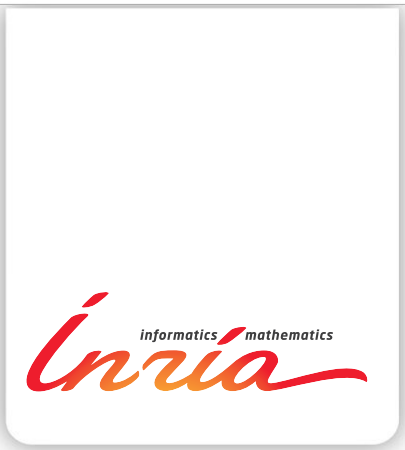
[VBB+14] Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of openmp dependent tasks with the kastors benchmark suite. In Luiz DeRose, BronisR. de Supinski, StephenL. Olivier, BarbaraM. Chapman, and MatthiasS. Müller, editors, *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pages 16–29. Springer International Publishing, 2014.

[VOC+12] Huy Vo, Daniel Osmari, Joao Comba, Peter Lindstrom, and Claudio Silva. Hyperflow: A heterogeneous dataflow architecture. In *Eurographics Symposium on Parallel Graphics ans Visualization*, 2012.

[VOS+10] Huy Vo, Daniel Osmari, Brian Summa, João Comba, Valerio Pascucci, and Cláudio Silva. Streaming-enabled parallel dataflow architecture for multicore systems. In Blackwell Publishing Ltd., editor, *Eurographics/IEEE-VGTC Symposium on Visualization*, pages 1073–1082, june 2010.

[Wal12] Ingo Wald. Fast construction of sah bvhs on the intel many integrated core (mic) architecture. *Visualization and Computer Graphics, IEEE Transactions on*, 18(1):47–57, Jan 2012.

[WAW+13] Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, and Ian T. Foster. Swift/t: Scalable data flow programming for many-task applications. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 309–310, New York, NY, USA, 2013. ACM.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

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

ISSN 0249-6399