VtkSMP: Task-based Parallel Operators for VTK Filters

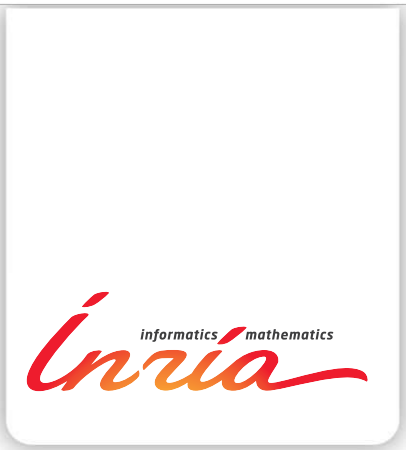
Mathias Ettinger, François Broquedis, Thierry Gautier, Stéphane Ploix, Bruno Raffin

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 — February 2012 — 16 pages

Abstract: Scientific simulations produce more and more memory consuming datasets. The required processing resources need to keep pace with this increase. Though parallel visualization algorithms with strong performance gains have been developed, there is a need for a parallel programming environment tailored for scientific visualization algorithms that would help application programmers move to multi-core programming.
Facing the challenge of genericity, standard solutions like Pthreads, MPI, OpenCL or OpenMP often fail to offer environments that are both easy to handle and lead to efficient executions.
In this paper, we study the parallelization of patterns commonly used in VTK algorithms and propose a new multi-threaded plugin for VTK that ease development of parallel algorithms. Focusing on code reusability, we provide tools that implements these patterns in a generic way. With little code modification, we experiment these patterns with a transform filter, brute force isosurface extraction filter and a min/max tree accelerated isosurface extraction. 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 : De nos jours, les simulations scientifiques produisent des jeux de données de plus en plus volumineux. Leur traitement nécessite donc des ressources adaptées. Et même si des algorithmes de visualisation des données fournissent de très bonnes performances en parallèle, une uniformisation de ces travaux reste nécessaire. Un environnement de programmation parallèle adapté à la visualisation scientifique aiderait les progammeurs à passer à une écriture d'applications à destination des architectures multi-cœurs.

Dans un contexte où une approche générique serait un atout, les solutions standard telles que Pthreads, MPI, OpenMP ou encore OpenCL réunissent rarement la facilité d'utilisation et les performances facilement accessibles.

Dans ce papier, nous étudions la parallélisation de schémas d'accès aux données présent dans la bibliothèque de visualisation scientifique VTK. Nous proposons un nouveau module pour VTK qui facilite l'écriture d'algorithmes parallèles. Ciblant la réutilisation du code existant, nous proposons des outils génériques qui implémentent ces schémas en parallèle.

Suite à de faibles modification de leur code, nous avons étudié nos outils sur un filtre transform, une extraction d'isosurface simple et une accélérée par un arbre de recherche min/max. Nos résultats montrent qu'une accélération d'un facteur 30 est obtenu sur une machine possédant 48 cœurs.

Mots-clés : parallélisme, environnements parallèles, vol de tâches, visualisation scientifique, VTK.

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. Taking advantage of their 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, an other issue is probably the lack of a standard parallel programming environment.

These kinds of programming environments are facing two main issues: providing a programming model that enables the programmer to swiftly shift from its sequential programming habits to parallel ones and a runtime system that ensures efficient executions even with a moderate optimization effort. Usual runtime systems generally lack one of these requirements.

Our research goal is to provide a fully functionnal plugin for VTK able to provide tools that are used to parallelize any VTK 's algorithm. Driven by genericity and code reusability, we provide high-level parallel operators that easyly replaces sequential pattern. Our plugin can handle differents runtime, choosen at compile time. The performances of our plugin are studied in a way that, depending on the parallel runtime it uses, the operators give the best of their capabilities.

In this paper 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 produces 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 which is a common accelerated data structure. Results show that work-stealing runtimes are better to ensure both efficiency and optimal ressources utilization on any kind of problems. They achieve speed-ups of 30 on a 48 cores machine for very irregular problems that expose severe load imbalance.

The following sections of this paper will present an overview of multi-threaded approaches for scientific visualisation and expose general guidances on work-stealing approaches (sections 2 and 3). Then we present our method of parallelisation based on data access patterns on section 4 and their efficiency on VTK filters on section 5. We conclude our paper in section 6.

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. Moreover MPI implementations on shared memory machines tends to suffer from high overheads. Ahrens *et al.* [ABM+01] relied on MPI to propose a two level parallelization of VTK applications. Given that the data to process can be partitioned (using ghost cells if necessary), their model supports data parallelism through a duplication of the processing pipeline. Additionally for each pipe-line data can be partitioned in blocks that are streamed down the pipe-line, 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].

The OpenMP standard relies on code annotations that the programmer uses to identify possible sources of parallelism, typically indicating when the iterations of a loop are independent and can thus be executed concurrently. Then the compiler and runtime can extract parallelism. This model avoids the drawback of thread level programming. But OpenMP relies on static load balancing strategies, impairing performance when the work-load vary dynamically due to the nature of the algorithm or a varying availability of cores. Version 3 of OpenMP was extended with the concept of tasks [DFA+09]. But current implementations rely on a centralized list scheduling that incurs high overheads.

Work-stealing, through libraries like Cilk [FLR98], TBB [Rei07] or X-KAAPI [GBP07, LMDG11], 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. To our knowledge, only a limited set of papers have been relying on these runtimes for parallelizing some scientific visualization filters [TDR10] and so far no generic approach as emerged. In the following section we remind the bases of work-stealing and the related programming model. 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]. It is also supported on the Intel Xeon Phi accelerator through Cilk and TBB. Task oriented programming relying on work-stealing scheduling thus appears as a possible unified paradigm for ensuring portability across CPUs and GPUs.

Though not supported through the aforementioned libraries, work-stealing has 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]. It shows that work-stealing could become a paradigm of choice for ensuring portability across CPUs and GPUs. Work stealing.

The GPU is also a target of choice for accelerating visualization filters. The programming model, based on Cuda or OpenCL, is SIMD oriented, often requiring significant programming efforts compared to a more classical sequential or multi-core programming approach. The OpenCL standard ensures a functional portability to different devices, including multi-core CPUs, but the code needs to be reworked to reach good performances.

The Piston approach [StLA12] proposes a programming environment for scientific visualization filters relying on 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. As far as we know, only Cuda and OpenMP have been tested in the Piston context. A good performance is reached for regular parallelism, but we can expect to face the performance limitations of OpenMP on CPUs when dynamics load balancing is required. Testing with TBB would be very interesting to validate the benefits of work-stealing in the Thrust context.

3 The Work-Stealing Paradigm

As stated by Lee [Lee06], directly programming at a thread level is considered to be error-prone. Thus the usage of parallel runtime is unavoidable if we aim at parallelizing a huge library such as VTK. Standard solutions, leaded by OpenMP, propose parallelization pattern that relies either on a static partition of

the parallel work or a centralized task list used to dynamicaly adapt to cores availability. Even if these approaches fit well problems that have either a regular workload or a non memory intensive data access schema, this is not the case of many meshes manipulation filters.

3.1 Coupling Parallel Algorithm to the Work-Stealing Scheduler

Work-stealing is a well-known technique to improve overall efficiency on modern multicore machines, especially when executing applications exposing irregular workloads. It consists in dividing a computation into fine-grain tasks that can be stolen from a loaded core to an idle one to dynamically balance the workload of a parallel application on any computing platform. Some popular parallel environments like Cilk [BJK+96, FLR98] and Intel TBB [Rei07, RVK08] have successfully implemented this technique providing mechanisms to efficiently deal with independent tasks. The X-KAAPI [GBP07, LMDG11, LMGD11] environment developed in our group goes further, providing portable ways of both expressing dependencies between tasks and scheduling them on large-scale heterogeneous parallel architectures efficiently.

3.1.1 The Execution Model

The work-stealing runtime system associates a *worker thread* to each core of the platform. Each worker thread is able to execute fine-grain tasks, and to steal tasks from other worker threads. A thread that creates tasks pushes them on its own workqueue. The creation of the task and the enqueue operation is garanted to be very fast in order to achieve performances. A running task can create child tasks. Dependencies amongst tasks and scheduling policies depends on the choosen runtime.

This model implements a valid, highly efficient sequential execution order, as the runtime system only needs to compute data flow dependencies when the thread execution scheme reaches a task that has been stolen and not completed yet. The successors of the stolen task depend on its completion. In order to keep fast stack-based execution without computation of data flow dependencies in X-KAAPI [GBP07], a thread suspends its execution when it reaches the first stolen task in its stack and calls the work-stealing scheduler to steal a new ready task.

3.1.2 Adaptive Tasks for Parallel Algorithms

Writing performance-portable programs within the task programming model requires creating much more tasks than available computing resources. Then, the scheduler can efficiently and dynamically balance the work load. But task management led to some overheads, even for the tasks that are not stolen.

Adapting the number of created parallel tasks to dynamically fit the number of available resources is a key point to reach high performance.

In the data flow model, a task becomes ready for execution once all its inputs have been produced. A task being executed cannot be stolen. To allow on-demand task creation, adapted runtimes extends this model: a task publishes a function, called *splitter*, to further divide the remaining work. The splitter is called on a running task by an idle thread during a steal operation. The task and its splitter are concurrent and must be carefully managed as they both need to access shared data structure. The programmer is held responsible for writing correct task and splitter codes. To help him, runtimes systems ensures that at most one thief performs splitter concurrently with the task execution. It allows for simple and efficient synchronization protocols.

4 Parallelization of VTK Algorithms

Our goal is to identify patterns of code within VTK filters that appears to be good candidates for parallelization. A simple first pattern 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.

On the other hand, some filters produce an amount of output data that cannot be a priori predicted. The strategy for these algorithms is to have each thread 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 studied 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 garanty its traversal in parallel with respect to its sequential depth first execution and branch cutting capabilities.

Following sections devellop each of these aspects.

4.1 ForEach

Given some requirements, iteration over independant loop is very easy to parallelize. Indeed, using n as the size of the loop, one can see such loop as n independant tasks. Thus giving each core $\frac{n}{\text{numberofcores}}$ elements to compute is the easiest way to parallelize this pattern. However such 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 other and wait them. Dynamic scheduling and work-stealing techniques can efficiently handle such problems.

But giving work to each processor is not the sole thing to care when dealing with independant loops. One has to make sure to allocate enought 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 then guaranty to divide the iteration range of the loop the best way regarding of the choosen runtime.

Our plugin provides the *vtkSMP::ForEach* 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 exhibit pieces of code that need to be carefully manage (memory allocation, thread safe methods, aso.) Writing the class is a matter of checking the sequential behaviour. Then calling the operator only need the range of the computation and an instance of that class.

4.1.1 The Work-Stealing Approach

When the *void operator() (vtkIdType id) const* operation is very regular we faced a scalability issue due to steals. Indeed, each steal adds a little overhead to the computation, loading 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 incur more overhead than computation speed-up. To avoid the performance drops induced by 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 the optimal value. If the grain is too coarse, load balancing becomes inaffective. Previous studies [PP95]

```
1  struct VcsModificator : public vtkFunctor {
2    vtkDataArray* inVcs;
3    vtkDataArray* outVcs;
4    double (*matrix)[4];
5    void operator () ( vtkIdType id ) const
6    {
7      double vec[3];
8      inVcs->GetTuple( id, vec );
9      vtkSMPTransformVector( matrix, vec, vec );
10     outVcs->SetTuple( id, vec );
11   }
12   // Regular VTK overloaded methods
         (PrintSelf, constructors, ...)
13 };
14
15 void vtkSMPTransform::TransformVectors(
16 vtkDataArray *inNms,
17 vtkDataArray *outNms)
18 {
19   vtkIdType n = inNms->GetNumberOfTuples();
20   this->Update();
21
22   VcsModificator* myvectorsmodificator =
         VcsModificator::New();
23   myvectorsmodificator->inVcs = inNms;
24   myvectorsmodificator->outVcs = outNms;
25   myvectorsmodificator->matrix =
         this->Matrix->Element;
26
27   vtkSMP::ForEach( 0, n, myvectorsmodificator
         );
28
29   myvectorsmodificator->Delete();
30 }
```

Figure 1: Implementation of the parallel *vtkTransform::TransformVectors(...)*. Base class for independant loop splits computation and parallel patterns.

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 containting 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 performance.

4.2 Thread Local Storage and Merge

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

The idea behind this second approach is to use privates output data structures per cores. This way, execution is closer to the sequential one and restrictions for thread safe methods are less strict. But:

- these data structures need to be initialized efficiently;
- each of these data structures will contain partial results that we need to merge afterwhile.

Initialization of such structures, called Thread Local Storage (TLS), must not slow down the computation. So we can not wait for all initializations to be achieved before starting the call to the *ForEach*. Instead we provide late stage initialization capabilities. The class used to enclose the loop body can define an *Init* function that will be called once (and only once) for each core before any iteration.

Such capabilities are also usefull 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.

Once initialization is done and actual computation performed in parallel, each TLS contain a part of the whole expected result. Since this result is often a mesh, our plugin provide capabilities to fuse partial meshes into a single one. And it performs it 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 builtin *Locators*. We extended the behaviour of *vtkMergePoints* to support parallel operations. Basically, this locator keeps track of existent points thanks to a spacially-guided hash table. The bounding box of the resulting mesh is regularly divided into a 3D grid and each voxel of this grid maps 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 is simple mathematics involving point and bounding box coordinates.

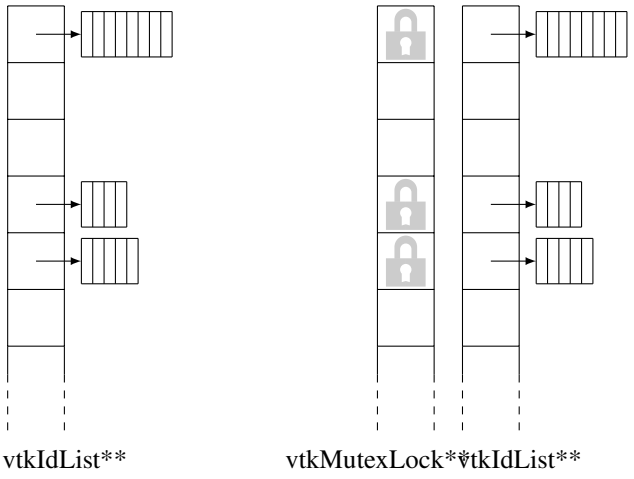


Figure 2: Inner representation of *vtkMergePoints* (left) and our first implementation of *vtkSMPMergePoints* (right). Each *vtkIdList* is designated as bucket.

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 produce wrong results if no care is taken. Our solution is to use an hash table of mutex in addition to the one of 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 lays 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 inspired 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 need of an atomic increase 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 initiate the merge of buckets corresponding to a voxel only by a thread that actually have data in such buckets. To do so, each thread looks for the first non-empty bucket that 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 that contains data.

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

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

4.3 Acceleration Data Structures

Due to their inner complexity, several algorithms have an accelerated version. It mainly consist 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 automaticaly used because they may introduce overhead for their initialization or their memory footprint. But what makes their efficiency 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 time — 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 un-productive content.

Usage of trees can be speed up in two different ways. The first one is by building the tree. But since this step is highly dependant on the type of the tree and its usage, it might be a per-implementation study. The second one is during the tree traversal. And 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 decision 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 availlable ressources, but the number of tasks is rapidly increasing and they can be efficiently scheduled on the whole machine.

4.3.1 The Work-Stealing Approach

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. And the cost of creating and managing tasks may become important compared to the actual computation of a task — especially in visualization where nodes often perform only a few comparisons to decide wether or not they must spawn their descendants.

The answer of some work-stealing runtimes such as TBB or X-KAAPI is to provide a way for creating adaptives tasks. Such tasks have the particularity to be splitable on demand by an idle resource, thus limiting the number of created tasks to the number of cores actualy executing them. For load-balancing purpose, splitted tasks may also be adaptive tasks.

Using these tasks, we propose an adaptive tree traversal which behave as follows:

- create an adaptive tasks that traverse the entire tree and start the (sequential) execution;
- on a demand for splitting the task, find the top-most un-treated node that would be the last computed in sequential depth-first order;
- consider this node and its sub-tree as treated and create an adaptive tasks that traverse this sub-tree.

Newly created tasks behaves like a fresh tree and their traversal follow the same rules as above. Thus they can also be splitted 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 Implementation of Testing Filters

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 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.

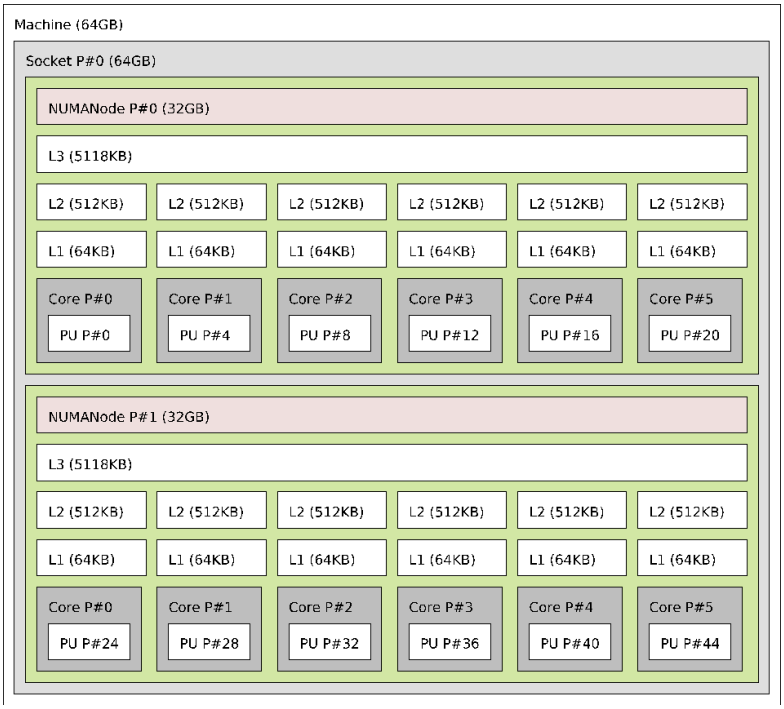


Figure 3: A socket from the AMD48 evaluation platform.

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

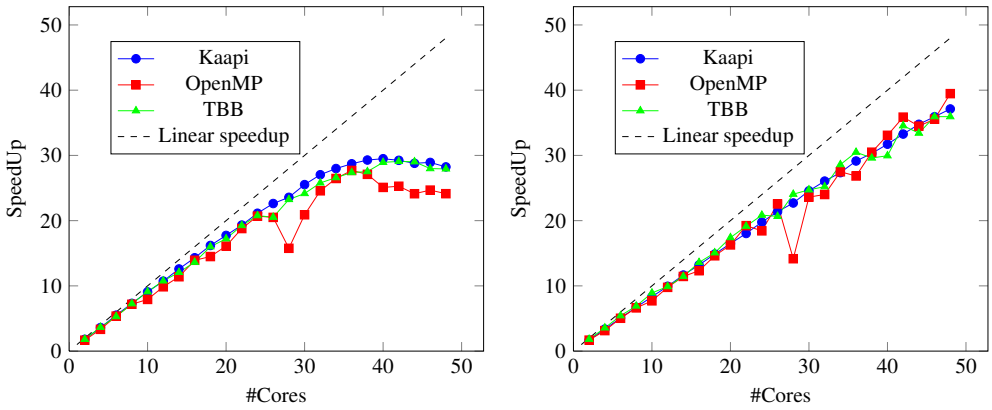
An example of VTK algorithm that performs a loop with independent iterations over cells and/or points is the *vtkTransformFilter*. It applies scales, translations and rotations on a mesh. The computation is

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

performed by a *vtkTransform* that contains the description of the transformations to be applied. The *vtkTransform* is supplied to the *vtkTransformFilter* and it is this *vtkTransform* that 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 that mimic the behaviour of the *vtkTransform* one except that the sequential loop was turned into a parallel ForEach.

Figure 4a shows the performances obtained on our AMD48 machine. Moreover, as we targeted both shared memory and NUMA machines, we have also 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* one after the other in our visualization benchmark. As a consequence, the first filter load data from a unique memory bank, leading to memory contention, and the second one have the memory pages of its input data interleaved on each memory banks thanks to the first filter. Thus the second filter make a better usage of memory bandwidth. Results are shown in figure 4b.



(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.

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 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, the one 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.

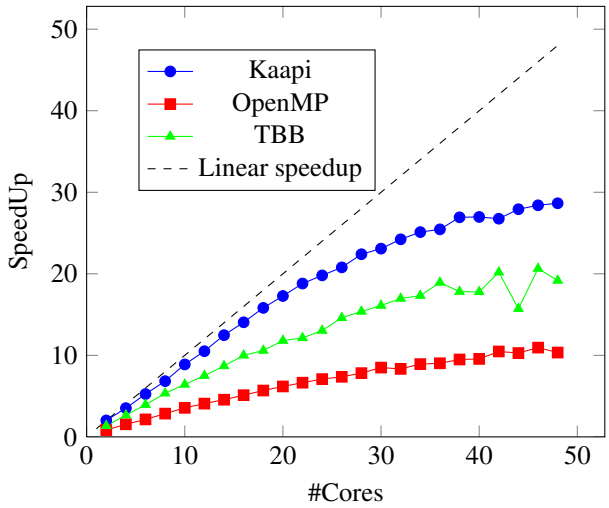


Figure 5: Speed-ups for execution of parallelized loop within *vtkContourFilter*. The difference between the runtimes is due to the high load of the first core.

The 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 compare to the `for` loop in the sequential algorithm. 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 partitionning of the iteration range leads to the first core being the only one to create the output mesh. As expected, work-stealing runtimes load balances efficiently and outperforms the static partitioning of OpenMP. Nevertheless, dynamic scheduling parametered with the right size for partition (grain) gives similar results.

5.3.1 Focus on the Merge Operator

The *vtkContourFilter* is a good place to experiment the behaviours of our *Merge* operator since the isosurface computation makes use of a locator per thread.

Figure 6 presents the execution time for both merge implementations. The method using all the locator is faster than the one using only one. It also requires twice as less memory since the array of *vtkMutexLock* is not required in this case. The only drawback is that not all the filters that would make use of the *Merge* operator in their parallel version can be parallelized using a thread local locator.

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

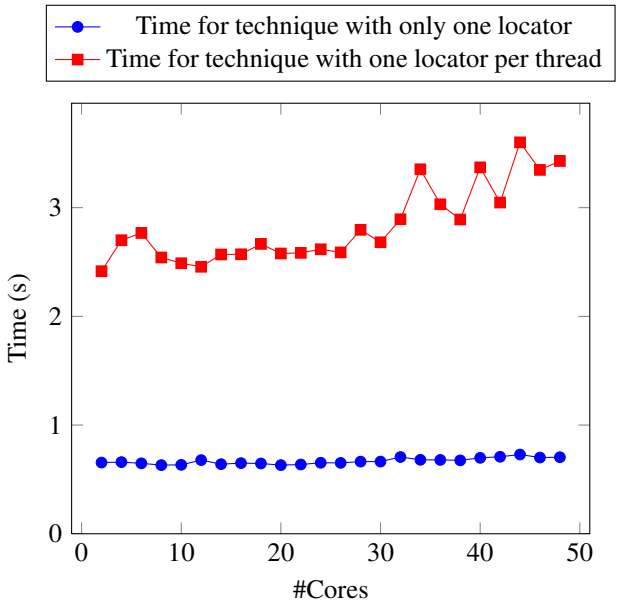


Figure 6: Comparison of our two merge implementations.

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 build 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*. We compared the parallel traversal and merge time to 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.

6 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.

Experiments showed guaranted performances without deep optimisation. Standard behaviour of our operators is efficient enough for someone with a limited knowledge in parallelism to exploit them well.

Moreover specific behaviour of work-stealing runtimes such as X-KAAPI or TBB can be exploited to reach better performances. They are also richly optimized so that the overhead of dynamic load balancing is small enough to be competitive with OpenMP static partitioning for regular applications.

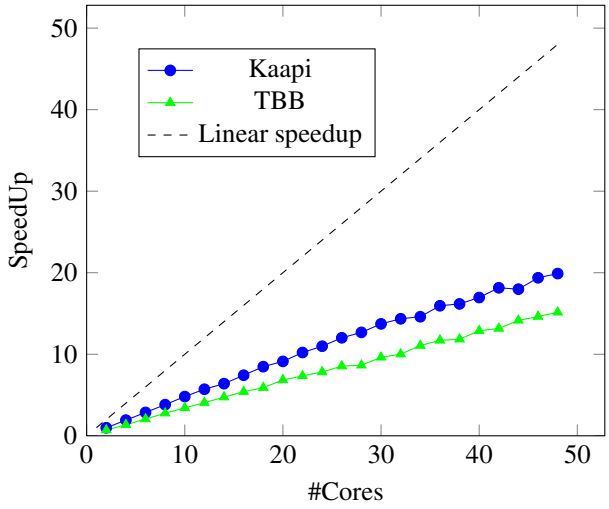


Figure 7: Execution of *vtkContourFilter* with our parallel tree traversal.

Our goal is to provide a full fledged parallel programming environnement dedicated to visualization filters. Next steps will focus on other classical algorithms like *vtkStreamTracer* and its subclasses and *vtkExtractEdge*.

We did not address the issue of parallelizing code on GPUs, load balancing still being delicate to perform efficiently for heterogeneous architectures. But this may change with new multi-core architectures for accelerators like the Intel Mic [Ska10] that integrate all features necessary for an efficient work-stealing implementation.

Lastly, parallel composition may lead to better resource utilisation. The idea is to exploit asynchronous filter execution presented in HyperFlow [VOC+12] and merge it with our multi-core framework. We hope to maximize both CPU and memory bandwith utilisation without falling to the pitt of an over utilisation of the memory bus.

Contents

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.

[BJK+96] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[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.

[DFA+09] Alejandro Duran, Roger Ferrer, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. A proposal to extend the openmp tasking model with dependent tasks. *Int. J. Parallel Program.*, 37:292–305, June 2009.

[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.

[LMGD11] Fabien Le Mentec, Thierry Gautier, and Vincent Danjean. The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming. Technical Report RT-0418, INRIA, 2011.

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

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

[RVK08] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *IPDPS*, 2008.

[Ska10] Kirk Skaugen. Petascale to exascale: Extending intel's hpc commitment. Technical report, ISC keynote, 2010.

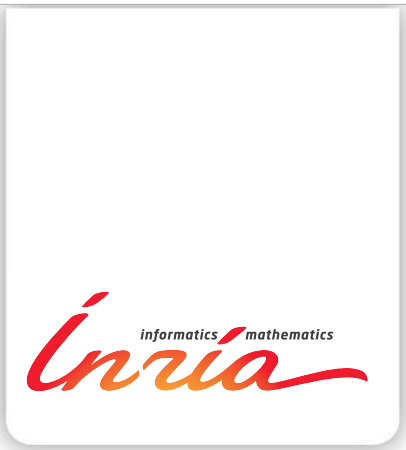
[StLA12] Christopher Sewell, Li ta Lo, and James Ahrens. Piston: A portable cross-platform framework for data-parallel visualization operators. In *Eurographics Symposium on Parallel Graphics ans Visualization*, 2012.

[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.

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

[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.



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