Eurographics Symposium on Parallel Graphics and Visualization (2013)
F. Marton and K. Moreland (Editors)

VtkSMP: Task-based Parallel Operators for VTK Filters

M. Ettinger[1] and F. Broquedis[2] and T. Gautier[1] and S. Ploix[3] and B. Raffin[1]

[1]Inria, France
[2]Grenoble INP, France
[3]EDF, France

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.*

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.1]: Hardware Architecture—Parallel processing; Processor Architectures [C.1.2]: Multiple Data Stream Architectures (Multiprocessors)—Parallel Processors; Programming Techniques [D.1.3]: Concurrent Programming—Parallel Programming

1. Introduction

The size of data produced by scientific simulations is growing at a steep rate. Post-processing tools, including scientific visualization ones, 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 standard libraries are still not supporting efficient multi-core parallel executions. The VTK scientific visualization library is one of them. Beside the required effort to revisit a large sequential code base, another issue is probably the lack of a standard, yet efficient, parallel programming environment. Programming environments for multi-core architectures are facing two main issues: providing a programming model that enables the programmer to shift swiftly from his sequential programming habits to parallel ones, and a runtime system that ensures efficient executions even with moderate optimization efforts.

In this paper we focus on task based programming environnements. The programmer delimits potential parallelism through tasks, the compiler and/or runtime system taking care of computing a static or dynamics task scheduling. Dynamic scheduling, usually based on work-stealing, is particularly interesting in the context of scientific visualization where the computation load is often irregularly spread on the dataset. We propose a VTK plugin to support 3 different environments, namely OpenMP, Intel TBB and X-KAAPI.

We study 3 different parallelization patterns that can be reused in various VTK filters. They are implemented for the 3 environnements with various refinement levels. The first pattern targets loops with independent iterations (*foreach* loop) producing 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 producing data with unknown memory footprints. A merge of partial results is thus necessary to produce a compact data structure. Eventually, we propose to parallelize a tree traversal, a pattern relevant for several acceleration data structures. We compare performance results with various VTK filters on a 48 core machine. Work-stealing based runtimes appear to be more versatile to ensure a good resource usage on various kinds of problems.

DOI: 10.2312/EGPGV/EGPGV13/041-048

We discuss related works in section 2 and remind the work-stealing paradigms in 3. We present our approach in section 4 and experimental results on various VTK filters in section 5, before to conclude in section 6.

2. Related Work

MPI is one of the most popular parallel programming environment for distributed memory machines. A MPI programmer splits a parallel application in several processes managing their own data and communicating through message passing. For stencil-like algorithms that communicate with neighbor elements, 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 copies 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 pipeline, data can be partitioned in blocks that are streamed down the pipeline, enabling the concurrent execution of the different stages. This approach does not support dynamic load balancing, relying on the user to define the partitioning policy.

Hyperflow [VOS*10, VOC*12] implements the same approach in a more modern context. They propose a thread-level streaming and pipeline duplication strategy. This thread-level parallelization enables to avoid the use of ghost cells on shared memory architectures. The programmer can also provide a GPU implementation of some filters. The Hyperflow runtime is then able to execute the related computations on the available GPUs.

Ahrens *et al.* [ALS*00] also proposed an early approach for extending VTK for shared memory machines, which consisted in providing a thread-based programming interface that can be used to parallelize filters. Thread-level programming is today recognized as a low-level approach that can be error-prone [Lee06].

The OpenMP standard relies on high-level code annotations that the programmer uses to identify possible sources of parallelism, typically indicating when the iterations of a loop can be executed concurrently. While this model avoids the drawback of thread-level programming, OpenMP relies on static load balancing strategies. The performance is impaired when executing parallel programs with irregular work-load due to the nature of the algorithm or the unsteady availability of cores. Version 3 of OpenMP was extended with the concept of tasks [DFA*09], but current mainstream implementations still suffer from overheads as stated in [BGD12].

Work-stealing, through libraries like Cilk [FLR98], TBB [Rei07] or X-KAAPI [GBP07, LMDG11], is emerging as a good candidate to efficiently exploit nowadays shared memory machines. Relying on a dynamic load balancing runtime, these libraries can cope with irregular applications and unsteady core availability. They propose a programming model based on fine-grain tasks to express parallelism. The runtime system is responsible for 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 rely on these runtimes for parallelizing some scientific visualization filters [TDR10], and so far no generic approach has emerged. In the following section we remind the bases of work-stealing and we describe the related programming model.

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 completely redesigned to achieve a good performance.

The Piston approach [StLA12] proposes a programming environment for scientific visualization filters relying on the Thrust library [thr12]. Thrust offers 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. They obtain a good performance with parallel algorithms exposing a regular workload, but we can expect a performance drop for algorithms requiring a dynamic load balancing. Testing the TBB library would be very interesting to validate the benefits of work-stealing in the Thrust context.

The Dax toolkit [MAGM11] targets exascale architectures but expriment on GPUs meanwhile. Their approach is parallel-centric. Traditional approaches pipe filters that locally iterate over all elements. Dax proposes to pipe worklets operating on individual elements, exposing a massive parallelism at the outmost level. This approach enables to expose more parallelism and reduce the required synchronization points, but does not offer a soft transition for legacy code.

3. The Work-Stealing Paradigm

As stated by Lee [Lee06], low-level parallel programming directly using thread libraries is considered to be error-prone. Parallel environments provide high-level constructs that help designing both portable and efficient programs, and are naturally considered when it comes to parallelize large libraries like VTK. Standard solutions, leaded by OpenMP, propose parallelization patterns that rely either on a static partitioning of the parallel work or a dynamic load balancing relying on a centralized task list. While being well suited to applications exposing regular workloads, such approaches may result in disappointing performance on irregular, memory-bound problems, like many meshes manipulation filters.

3.1. Coupling Parallel Algorithm to the Work-Stealing Scheduler

Work-stealing is a well-known technique to improve the overall efficiency of parallel applications on modern multicore machines, especially when these applications expose irregular workloads. It consists in dividing a computation into fine-grain tasks. Each core manages a local list of task to process. Idle cores stole tasks from loaded cores, ensuring the workload balance. 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] environment developed in our group goes further, supporting tasks with dependencies 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 into its own workqueue. The task creation and the enqueue operation are designed to lead to minimal overheads. A running task can create children tasks. Some implementations like the X-KAAPI runtime system enable to express dependencies between tasks, based on the task input and output variables (data-flow dependencies). This model implements a valid sequential execution order. The runtime system only needs to compute data-flow dependencies when a thread reaches a task that has been stolen and not yet stamped as ready for execution. The successors of the stolen task depend on its completion. During task execution, if a thread encounters a stolen task, it suspends its execution and switches to the workstealing scheduler that waits for dependencies to be met before resuming the task. Otherwise, and because sequential execution is a valid order of execution [GBP07], tasks are performed in FIFO order without computation of data flow dependencies.

3.1.2. Adaptive Tasks for Parallel Algorithms

Writing efficient programs within the task programming model requires creating much more tasks than available computing resources. Then, the scheduler can efficiently and dynamically balance the workload. But tasks management leads to 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 achieve a good performance and scalability. A task becomes ready for execution once all its input variables have been produced. A task being executed cannot be stolen. To allow on-demand task creation, some runtimes extends this model: a task publishes a function, called *splitter*, that contain the logic for extracting part of its work load. On a steal operation, the splitter is called by an idle thread. A new task is handled to this thread based on the work the splitter extracted from the victim. The task and its splitter run concurrently and must be carefully managed as they both need to access shared data structures. The programmer is held responsible for writing correct task and splitter codes. To help him, the runtime system ensures that only one thief calls the splitter, extracting work load for itslef and all other idle thieves requesting work from this victim.

4. Parallelization of VTK Algorithms

Our goal is to identify code patterns in VTK filters that can be good candidates to be parallelized and to provide operators to ease this parallelization. The first pattern we study here is the loop with independent iterations. Many VTK filters use this pattern to iterate over cells and/or points. Because the computations for one iteration do not depend from the results of previous ones, the parallelization of this pattern is pretty straightforward. Each iteration can be embedded into an independent task. The runtime system is then responsible for efficiently scheduling these tasks over all the available computing resources. We implement this pattern using a *ForEach* construct. When each task can write its results independently from the others, this pattern does not require further effort.

On the other hand, some filters produce an amount of output data that cannot be *a priori* predicted. The strategy to tackle this situation is to have each thread performing 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 needs to be very efficient to limit the overheads that impair the parallelization efficiency.

The last pattern we study in this paper involves some acceleration data structures that are used to speed up the execution of serial VTK filters. Such structures are often implemented as trees (binary trees, octrees, kD-trees...), so we provide a parallel version of a generic tree traversal. We guarantee a parallel tree traversal that respects the sequential depth-first exploration scheme and the branch cutting capabilities.

The following sections develop each of these aspects.

4.1. ForEach

Parallelizing an independant loop is pretty straigthforward. Let n be the size of the loop, one can see such loop as n independant tasks. Assigning $\frac{n}{\text{number of cores}}$ tasks to each core of the platform happens to be the easiest way to parallelize this pattern. While being well suited to regular problems, this approach can lead to load imbalance issues when the work load varies from one iteration to another. To tackle such situations, the *ForEach* operator we provide rely on a dynamic scheduling and, when supported by the chosen runtime, work-stealing techniques.

Beside load balancing issues, we leave to the VTK programmer the responsability to ensure a proper memory allocation and the use of thread-safe methods.

```
struct VcsModificator : public vtkFunctor {
  vtkDataArray* inVcs;
  vtkDataArray* outVcs;
  double (*matrix)[4];
  void operator () ( vtkIdType id ) const
    {
    double vec[3];
    inVcs->GetTuple( id, vec );
    vtkSMPTransformVector( matrix, vec, vec );
    outVcs->SetTuple( id, vec );
    }
// Regular VTK overloaded methods (PrintSelf,
      constructors, ...)
};

void vtkSMPTransform::TransformVectors(
                      vtkDataArray *inNms,
                      vtkDataArray *outNms)
  {
  vtkIdType n = inNms->GetNumberOfTuples();
  this->Update();

  VcsModificator* myvectorsmodificator =
        VcsModificator::New();
  myvectorsmodificator->inVcs = inNms;
  myvectorsmodificator->outVcs = outNms;
  myvectorsmodificator->matrix = this->Matrix->Element;

  vtkSMP::ForEach( 0, n, myvectorsmodificator );

  myvectorsmodificator->Delete();
  }
```

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

Figure 1 shows an example of our *vtkSMP::ForEach* operator applied to a transform filter. As for Intel TBB *tbb::parallel_for*, the VTK programmer has to move the sequential body of the loop into a freshly built class, leading to separate the algorithm (the pattern) from the computation. Writing this class requires to check that the sequential code to be embedded in the task behaves correctly with respect to memory allocation and thread safety. Calling the operator is then straightforward, as it only requires to pass the range of the loop to execute and an instance of the corresponding class.

4.1.1. Work-Stealing for Independent Loops

Used unwisely, work-stealing can lead to poor scalability. Indeed, each steal operation adds a little overhead to the computation, loading the memory bus to retrieve the required data. The number of steal operations can grow significantly as we get closer to the end of the computation. To avoid the performance drops induced by this behavior, the work-stealing scheduler defines a threshold, called *grain*, representing the minimal number of iterations to be executed by a task. Setting this grain to its optimal value is crucial to achieve good performance. If the grain is too coarse, the load balancing becomes ineffective. If the grain is too fine, the scalability gets limited by task management overheads. Preparata and Pan [PP95] showed that $\Theta(\sqrt{n})$ was a good value for this grain, considering that the number of created tasks is, at most, $\frac{n}{\text{grain}}$ and that 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 grain set to $\sqrt{n}$ leads to the best performance.

4.2. Merging Parallel Contributions

In some VTK filters, the size of the output data generated from independent loops cannot be statically predicted. Using the *ForEach* construct on such loops requires the use of per-core private memory areas to store output data structures. This is usually implemented using a thread library feature called *Thread Local Storage*. This way, the parallel behavior is closer to the sequential one, and it loosens restrictions for thread safe methods. But:

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

The initialization of such structures should not slow down the computation. In other words, we do not want to wait for every thread to initialize its own private structure before starting to execute the *ForEach* construct. 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.

Once the initialization and the computations are performed, each Thread Local Storage contains a part of the whole expected result. Since this result is often a mesh, our plugin provides capabilities to fuse partial meshes into a single one in parallel. This operation cannot behaves like the *vtkAppendFilter*, which gathers together the points and cells of several meshes. We need to take care of the potential duplicate points at the boundary of each partial meshes. Keeping all those points would affect the mesh manifoldness (if any). The results would give unexpected outputs when filtered through decimation or subdivision for example.

To track and remove duplicated points over partial meshes, we use VTK 's builtin *Locators*. We extended the behavior of *vtkMergePoints* to support parallel operations. Basically, this locator keeps track of existing 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, called bucket, in the table. Each bucket stores the indexes of the points that belong to this voxel. Knowing the index of the bucket associated to a point is a simple arithmetic operation involving the point and bounding box coordinates.

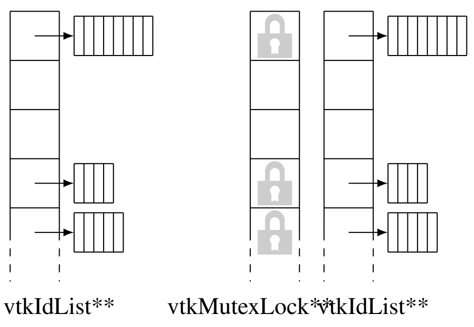


Figure 2: Internal data structure used by *vtkMergePoints* (left) and first implementation of *vtkSMPMergePoints* (right). Each *vtkIdList* coresponds to a bucket.

The locator we implemented is a helper class to perform a parallel merge of all partial meshes. It comes with a hash table of mutexes in addition to the one of point ids, as shown in figure 2. Each point insertion starts with a lock operation on the coresponding bucket. Overheads related to concurent lock acquisitions are limited as they only occur for the points that lay on the partial mesh boundaries. However, adding one lock per entry leads to roughly double the memory footprint of the data structure.

Current sequential implementations of some VTK filters already make use of locators, and thus need one locator per thread when running in parallel. In this case, we rely on the flat combining approach [HIST10] to efficiently merge the partial meshes and build a global locator. The points could be merged by assigning part of the voxels to each thread. A thread then gathers the buckets from all local locators for each voxel, removing duplicated points if any. But we can be more efficient by taking into account that usually more than 90% of the buckets are empty. The idea is to make one of the threads actually having data at a given voxel responsible for merging the points of this voxel. 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 this voxel, it retrieves the data associated to it from the other threads and merges them in the output data structure. Otherwise it ignores this bucket and looks for the next one that contains data.

Our framework provides both merge capabilities. If the sequential filter already uses a locator for its output points, it is more likely to use thread local locators in its parallel version. The second merge algorithm is then more relevant. Otherwise a new locator is created for the parallel version and used as described in the first technique.

4.3. Acceleration Data Structures

Several filters support a version relying on an acceleration data structure. The purpose of such structures is to decrease the amount of data that needs to be analyzed. However, they lead to some overheads mainly due to their initialization or their memory footprint. They are most of the time used when the data are requested several times with different values, thus amortizing the initialization cost.

In this paper we focus on tree based acceleration data structures. Their goal is to avoid iterating over all end-elements (cells, pixels, objects, 3D space) by providing a mean to identify quickly where relevant elements are located. Using trees to speed up the execution of a VTK filter is mainly driven by two factors: how long it takes to build the tree, and how expensive the traversal operation is. We focus on the second factor, providing a generic operator for parallel tree traversal. This pattern is pretty different from the previous ones as a node can only be processed after its parent. Moreover, the computation of a node may lead to decide to stop exploring its descendants, leading to an unpredictable work load. A simple solution consists in spawning one task per node:

- the tree traversal starts by spawning the root task;
- for the nodes, the task must spawn one task per descendant that needs to be explored;
- for the leaves, the task must apply the computation over the end-elements.

The traversal of the upper levels may result in limited speed-up as long as there are fewer tasks than available ressources. The tasks are dynamically spawned and their number increases rapidly, requiring a runtime able to efficiently schedule them on-line.

4.3.1. Work-Stealing for Tree Traversal

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. The overhead of creating and managing tasks may become important compared to the actual computation of a task, especially in visualization filters, where nodes often perform only a few comparisons to decide wether or not they must spawn their descendants. As described in section 3.1.2, the adaptive task model aims at reducing task related overheads. It is supported by TBB and X-KAAPI runtimes. Taking advantage of this mechanism, we propose an adaptive tree traversal behaving as follows:

- the tree traversal starts with one adaptive task that (sequentialy) traverses the entire tree;
- when a thief calls the splitter of its victim, this one extracts the topmost unprocessed ready node (*i.e.* a node whose parent have been processed);
- the thief stole this node and its sub-tree and starts its own adaptive task to 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 splitted if a processor is idle.

The non adaptative approach presented before would spawn as many tasks as nodes explored. In opposite, the total number of splits that occur here (*i.e.* the total number of tasks created) is Θ(number of processors), enabling to signficantly reduce the overheads as shown through the experimental results.

5. Implementation of Testing Filters

We made our framework compilable against OpenMP, TBB and X-KAAPI. It also provides mechanisms to ease supporting other runtimes. Results compare the execution of VTK filters for these 3 runtimes.

We conducted our experiments on a CC-NUMA machine made of 4 AMD Magny Cours processors holding 12 cores each. We will refer to this configuration as **AMD48** in the following of this section.

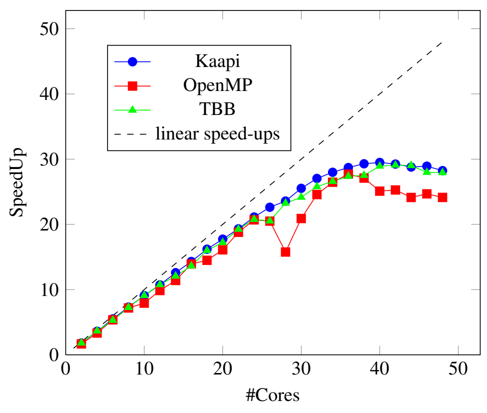
Filters are tested with Lucy mesh from the Stanford 3D Scanning Repository. This mesh contains 28 M cells (triangles) for 14 M points.

5.1. vtkTransformFilter

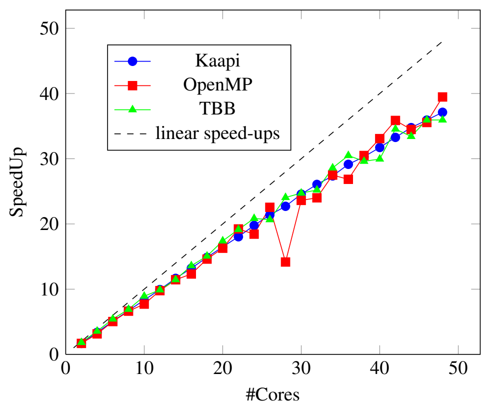
An example of VTK algorithm that performs a loop with independent iterations over cells and/or points is the *vtkTransformFilter*. It applies scales, translations and rotations on a mesh. The computation is performed by a *vtkTransform* that contains the description of the transformations to be applied.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

We modified *vtkTransformFilter* to perform the needed memory management before every computation. We built the *vtkSMPTransform* class that mimics the behavior of the *vtkTransform* one except that the sequential loop was turned into a parallel ForEach. Figure 3a shows the performances obtained on our AMD48 machine. As memory hierarchy of NUMA machines can lead to significantly slow down the execution, it is important to take into consideration the memory layout to maximise the use of the available bandwidth.

We put two *vtkTransformFilter* one after the other in our visualization benchmark, to compare the performance of two different page placements on the machine memory banks. The first filter loads data from a unique memory bank (memory layout resulting from the execution of the sequential mesh reader). This filter execution thus suffers from memory contention as all threads need to access the same memory bank. But, each thread stores the transformed mesh in pages located in the thread local memory bank. The resulting mesh is evenly stored in the various memory banks. The second filter transforms this mesh, and thus can benefit from a higher aggregated memory bandwidth (Fig. 3b).



(a) First filter (input data on a unique memory bank).



(b) Second filter (input data distributed over all memory banks).

Figure 3: Performance results (speed-up against the sequential `for` loop execution) of parallel *vtkTransformFilter* executions with different memory page layouts.

As expected, mapping data close to the core that uses them reduces the memory contention, leading to enhanced speedups. Results also show that the work-stealing runtimes TBB and X-KAAPI are at least as efficient as OpenMP for highly regular loops, which is known to be the stomping ground of the latter.

5.2. 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 dataset. In its current implementation, *vtkContourFilter* mainly switches 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.

For each cell in the mesh, the algorithm compares the scalar values associated to the points of this cell with the isovalue. The cell is skipped if the isovalue does not fit into the range of the scalar values. Otherwise a fragment of the isosurface is created for this cell. Since the created fragment depends on both the distribution of scalar values in the cell and the topology of the cell, it is not possible to know *a priori* how many points and cells (one or several points, lines, triangles. . .) will be created per input cell.

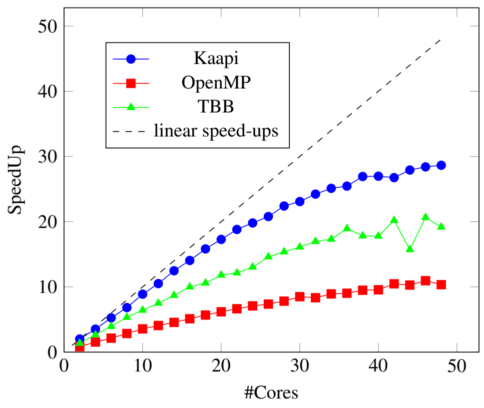


Figure 4: Speed-ups for the execution of the parallelized loop within the *vtkContourFilter* computed against the execution of the `for` loop of the sequential filter. The difference between runtimes is due to the high load of the first core.

The parallelized version of the *vtkContourFilter* uses a Thread Local Storage and must fuse the partial meshes. Figure 4 shows the execution of the surface creation (with merging) on our AMD48 platform. Since isocontouring is an algorithm that can often face load balancing issues, we provided a highly unbalanced input dataset in this experiment: we set scalars values in such a way that only the first cells can possibly contain points whose scalar values are both above and below the isovalues. Thus a static partitionning of the iteration range puts all the work load on the first core. As expected, work-stealing runtimes efficiently balance the work load and outperform the static partitioning of OpenMP. OpenMP can achieve a similar performance if using the dynamic scheduling parametered with the appropriate partition size (grain). Only the static OpenMP results are shown in figure 4.

5.2.1. Merge Operator

The *vtkContourFilter* is a good place to experiment the behavior of our *Merge* operator as the isosurface uses one locator per thread.

Figure 5 presents the execution time for both merge implementations. The method that takes advantage of the local locators is faster than the one that uses only a global one. It also requires twice as less memory since the array of *vtkMutexLock* is not required in this case. The optimized *merge* algorithm is signficantly faster than the first one, but this approach only makes sense for the filters that already use a locator in their sequential version.

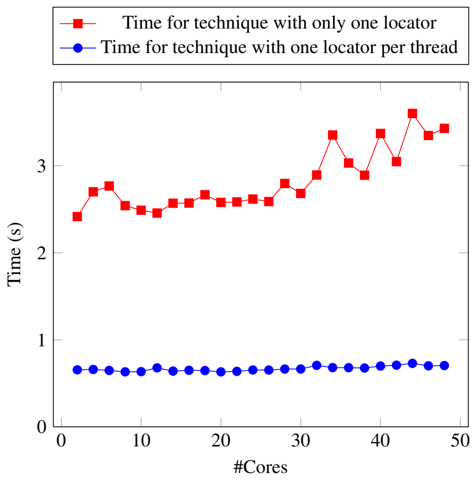


Figure 5: Comparison of our two merge implementations.

5.3. 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 cells associated to each leaf. If the isovalue is not included in the min-max interval of a given node, no further test is needed and all the nodes of the sub-tree are skipped.

Results presented in figure 6 show the differences between the classical task spawning technique (implemented with TBB) and the adaptive approach (implemented with X-KAAPI). Notice that the adaptative algorithm could also be implemented with TBB. We computed 11 isovalues on the input mesh with the accelerated version of the *vtkContourFilter*. We compared the parallel traversal and merge time 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 related 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

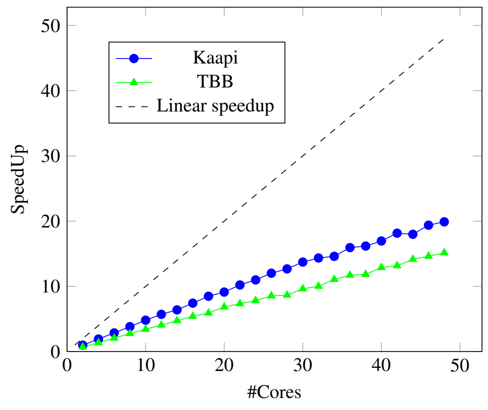


Figure 6: Execution of *vtkContourFilter* with our parallel tree traversal.

favors a smooth refactoring of the sequential code. Experiments show that we can acheive a good performance without advanced optimizations. The default behavior of our operators is efficient enough to be used directly without requiring an advanced expertise in parallel programming. Moreover, our operators enable to take advantage of the work-stealing runtimes such as X-KAAPI or TBB for an improved performance. These runtimes are well optimized. The overheads related to dynamic load balancing are small enough to obtain competitive execution times on regular applications compared to OpenMP static partitioning.

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 *vtkExtractEdge*. We did not target GPUs, but will consider the new multi-core architectures for accelerators like the Intel Mic [Ska10] that integrates all features necessary for an efficient support of work-stealing. Lastly, the parallel composition of filters may bring more parallelism for a better resource usage. The idea is to exploit an asynchronous filter execution as presented in HyperFlow [VOC*12], each filter being internally parallelized with our framework. We hope to maximize both the CPU and memory bandwidth usage without falling to the pitt of an over utilization of the memory bus.

References

[ABM*01] AHRENS J., BRISLAWN K., MARTIN K., GEVECI B., LAW C. C., PAPKA M.: Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications* (2001), 34–41. 2

[ALS*00] AHRENS J., LAW C., SCHROEDER W., MARTIN K., INC K., PAPKA M.: *A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets*. Tech. rep., 2000. 2

[BGD12] BROQUEDIS F., GAUTIER T., DANJEAN V.: 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* (Berlin, Heidelberg, 2012), IWOMP'12, Springer-Verlag, pp. 102–115. 2

[BJK*96] BLUMOFE R., JOERG C., KUSZMAUL B., LEISERSON C., RANDALL K., ZHOU Y.: Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing 37*, 1 (1996), 55–69. 3

[DFA*09] DURAN A., FERRER R., AYGUADÉ E., BADIA R. M., LABARTA J.: A proposal to extend the openmp tasking model with dependent tasks. *Int. J. Parallel Program. 37* (June 2009), 292–305. 2

[FLR98] FRIGO M., LEISERSON C. E., RANDALL K. H.: The implementation of the cilk-5 multithreaded language. *SIGPLAN Not. 33* (1998), 212–223. 2, 3

[GBP07] GAUTIER T., BESSERON X., PIGEON L.: 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. 2, 3

[HIST10] HENDLER D., INCZE I., SHAVIT N., TZAFRIR M.: Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2010), SPAA '10, ACM, pp. 355–364. 5

[Lee06] LEE E. A.: The problem with threads. *Computer 39* (2006), 33–42. doi:10.1109/MC.2006.180. 2

[LMDG11] LE MENTEC F., DANJEAN V., GAUTIER T.: *X-Kaapi C programming interface*. Tech. Rep. RT-0417, INRIA, 2011. 2, 3

[MAGM11] MORELAND K., AYACHIT U., GEVECI B., MA K.-L.: Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on* (2011), pp. 97–104. 2

[PP95] PAN V. Y., PREPARATA F. P.: Work-preserving speed-up of parallel matrix computations. *SIAM J. Comput* (1995). 4

[Rei07] REINDERS J.: *Intel threading building blocks*, first ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007. 2, 3

[RVK08] ROBISON A., VOSS M., KUKANOV A.: Optimization via reflection on work stealing in TBB. In *IPDPS* (2008). 3

[Ska10] SKAUGEN K.: *Petascale to Exascale: Extending Intel's HPC commitment*. Tech. rep., ISC keynote, 2010. 8

[StLA12] SEWELL C., TA LO L., AHRENS J.: Piston: A portable cross-platform framework for data-parallel visualization operators. In *Eurographics Symposium on Parallel Graphics ans Visualization* (2012). 2

[TDR10] TCHIBOUKDJIAN M., DANJEAN V., RAFFIN B.: Cache-efficient parallel isosurface extraction for shared cache multicores. In *Eurographics Symposium on Parallel Graphics ans Visualization* (2010). 2

[thr12] Thrust library. http://code.google.com/p/thrust/, 2012. 2

[VOC*12] VO H., OSMARI D., COMBA J., LINDSTROM P., SILVA C.: Hyperflow: A heterogeneous dataflow architecture. In *Eurographics Symposium on Parallel Graphics ans Visualization* (2012). 2, 8

[VOS*10] VO H., OSMARI D., SUMMA B., COMBA J., PASCUCCI V., SILVA C.: Streaming-enabled parallel dataflow architecture for multicore systems. In *Eurographics/IEEE-VGTC Symposium on Visualization* (june 2010), Ltd. B. P., (Ed.), pp. 1073–1082. 2