Towards Fully User Transparent Task and Data Parallel Image Processing

Jan Lemeire, Yan Zhao, Peter Schelkens
Dept. of Electronics and Informatics (ETRO), Vrije Universiteit Brussel (VUB) -
Interdisciplinary Institute for Broadband Technology (IBBT), Brussels, Belgium

Steve De Backer
Visionlab, Antwerp University - Interdisciplinary Institute for Broadband Technology (IBBT), Belgium

Frans Cornelissen, Bert Torfs
Janssen Pharmaceutica, Turnhoutseweg 30, B-2340 Beerse, Belgium

Abstract

This paper reports on the integration of parallel image processing in the ITK library and on improvements to the state-of-the-art of user transparency. In our approach, image processing tasks are wrapped into objects which are passed to the parallel engine. The engine is able to exploit data and task parallelism when executing the tasks on multicores, clusters and/or GPUs. All features necessary for efficient parallel processing are specified by the task objects. The engine can figure out most of the features itself, and is able to check the correctness of the features provided by the user. Interoperation optimization is attained by efficient scheduling of the tasks. The task dependency graph is automatically created at runtime. This is possible by delaying the execution of the tasks and by the intrinsic ITK pipeline updating mechanism. The low-level functions are also made available for the user, as well as a library-independent version.

1 Introduction

Processing power is nowadays abundantly available, as well as libraries and tools for efficient parallel processing. But there is still a high threshold associated with the use of high-performance computing architectures. Our goal is to maximally hide the parallel aspects for the non-expert. Our strategy is to create domain-specific solutions and to integrate them into domain-specific libraries. Domain-specific assumptions allow us to reach a higher level of user-transparency. Here we consider image processing and integrated our parallel solution into the ITK library. This work is part of an ongoing project [2], in which the ITK library (`http://www.itk.org`) was chosen. ITK is an open-source, cross-platform system that provides developers with an extensive suite of software tools for image analysis.

The key extensions to the state-of-the art [6, 13, 7, 11] of user transparent parallel image processing are the following:

1. Our solution provides 3 interfaces, called levels, which are integrated in ITK and 2 library-independent levels.

2. We opt for object-oriented generic programming instead of skeleton programming.

3. All forms of parallelism are exploited: data and task parallelism on the application level; multi-threading, message-passing and the use of GPUs on the system level.

4. The parallel engine can automatically extract the tasks' parallelization features and check for their correctness. This serves as guidance of the non-expert user.

5. The task dependency graph is automatically generated at runtime which is used for interoperation optimization. This overcomes the main limitation of a library-based approach [11]. In our approach, image operations are specified at a higher level than for example in [7]. We call the operations tasks: a task transforms two images into a new image, for example. The transformation itself can be a set of instructions.

The following section presents the philosophy behind our approach. The library is given in Section 3. Section 4 explains how efficient task parallelism is attained. Section 5 makes an assessment of our approach and Section 6 presents experimental results.

2 Approach

Fig. 1 shows the philosophy of the solution. The user 'concretizes' image operations by wrapping them into task objects. The application is implemented as a sequence of tasks and images which are passed to the parallel engine. The parallel engine automatically tracks the dependencies between the tasks. These are described by the task dependency graph. The engine then executes the tasks in the most efficient way utilizing the available processing power.

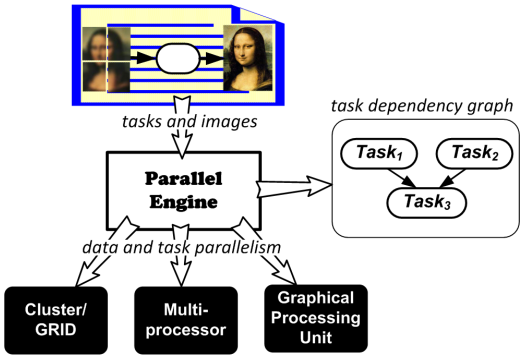

Figure 1. Scheme of the Parallel Engine.

The idea is that the user should simply pass the tasks and images to the parallel engine. The engine will make the optimal decisions about executing the tasks. All information the parallel engine should know about the tasks is given by the task object. This is explained in the next subsection.

2.1 Generic Algorithm Design with Object-Oriented Techniques

Generic parallel solutions are mostly designed through skeletons. Skeletons are algorithmic abstractions which encapsulate a form of parallelism common to a series of applications. The intricacies of parallelization are hidden behind the library's API. Skeletons are most often programmed in functional programming languages. A similar but more modern approach is using object-oriented programming. The idea is that a generic solution is provided by a *template method* in which problem-dependent parts are filled in by *hook methods* which are supplied by the library user [10]. The hook methods correspond to the function parameters in skeletons. It must be noted that ITK is a full object-oriented library.

Applied to image processing, the user implements the hook methods in a class which inherits from an abstract class, called `ImageTask`. The template methods are implemented in what we called the parallel engine. Each specific image operation is encapsulated in an object which provides all information that is required by the engine. In this way the tasks become *tangible*. They can be run sequentially or in parallel, can be send to other processors, saved in memory, the result of the parallel execution can be compared with that of the sequential run, etc.

Object-oriented programming has already been used for the design of generic parallel programs [4, 14]. But, to our knowledge, the idea of encapsulating tasks into objects has never been investigated in the way here proposed. Also functional skeleton programming ensures the separation of the tasks from their parallel implementation. The main difference is code organization and reusability. In skeleton programming, the user passes variables and functions as parameters. In object-oriented programming, all these parameters are grouped in a class definition, which is reused by all object instantiations of that class. Passing objects enhances code readability by their encapsulation capacities. One passes one object instead of a possibly long list of parameters. Moreover, it is possible to create new classes by inheriting from other ones.

The concrete class design is given in the next section. First we discuss the functionalities of the parallel engine.

2.2 Parallel Engine

For minimizing the execution time, the engine tries to maximally exploit all available parallelism. The engine first utilizes task parallelism by scheduling the tasks among the available processors. The tasks are run concurrently taking into account precedence constraints specified by the task dependency graph. This will be explained in more detail in Section 4. If there are more processors available than tasks, data parallelism is used to run a single task in parallel.

Data parallelism is supported for image tasks for which the result comes from an operation which can be independently applied on each pixel of the input images. The result can be a new image or a global result which is calculated from all subresults. Most image operations, such as point, local neighborhood and global operations belong to this category. Parallel execution of these tasks is achieved by splitting and distributing the image and applying the operation on the subimages [8]. For local neighborhood operations, the destination pixel is a function of the source pixel and the value of the pixels in the neighborhood surrounding it. The neighborhood is characterized by the *border size*. For global operations, the result is a function of the whole image. The border size is then set to the size of the image. Of course this limits the benefits of data parallelism since the whole image should be transferred for parallel execution.

The user has the possibility to specify routines for special-purpose processing elements, such as Graphical Processing Units (GPUs). If these processing elements are available, the parallel engine can decide to run the task on them. For GPUs, the user has to rewrite the image task as a highly-optimized SIMD (Single Instruction Multiple Data) routine. In the context of this project, we work with NVIDIA graphics cards and write the routines with the CUDA library (`http://www.nvidia.com/cuda`).

Besides the efficient execution of the tasks, the parallel engine offers the following functionalities:

1. Comparison of the parallel result with the result of sequential execution of the tasks. This allows the verification of the correctness of the parallel execution.

2. Returning a performance analysis. The engine gives the speedup and efficiency of the execution and a quantification of the different overheads.

3. Rescheduling tasks in cases of failure, for example by a sudden unavailability of some processing units.

4. Figuring out the border size of neighborhood operations. It is calculated by applying the task on a restricted region of a random input image and check the

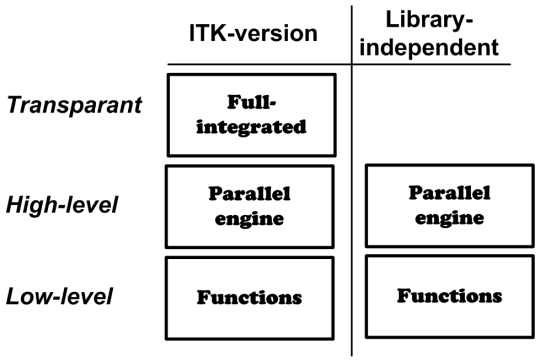

Figure 2. The 3 levels and 2 versions of the library.

pixels which were modified. In this way, a check of the border size as specified by the user can be performed.

5. Checking the object serialization as specified by the user.

3 Interface of Libraries

Fig. 2 shows the different parts of the library. At the top level is the version which is completely integrated in ITK. At the second level, the user creates and passes task objects to the parallel engine. There is an ITK version, based on the ITK Image class and a library-independent version, based on very general definitions of images (an array of pixels), borders etc. At the third level, the low-level functions used by the parallel engine are made available. They constitute functions to communicate and partition images, test functions, etc. One level can quite easily be implemented using the lower level.

We first define the high-level interface and then show how this can be integrated in ITK.

3.1 High-level Interface

A generic solution should properly define the class of sequential algorithms for which it applies. The following `ImageTask` class defines an operation on one image which is the result of a operation which can be independently applied on each pixel of the image. The result is an image and/or another set of variables. The following abstract methods have to be filled in by the user. A † indicates that the method should be implemented by the user. A ‡ denotes that it is optional. For reasons of clarity, some details are left out.

- `ImageRegion* execute(ImageRegion* input)`†: the operation that should be applied on the image regions of the input images.
- `void describe()`†: with this function the user describes all attributes of the class. This function is used for all serialization tasks, such as packing of objects into messages, unpacking messages, writing/reading objects to/from file. The mechanism is similar to that of the BOOST serialization library (`http://www.boost.org`).
- `int taskClassID()`†: a unique identifier of the tasks class. Is used to remotely create task objects.
- `int border()`‡: returns the size of the neighborhood that should be considered when applying the task on a region.
- `void recombine(ImageTask* partialResults)`‡: method which combines the partial results to calculate the global result. The function is used whenever that task should return a global result different than an output image.
- `Image executeOnGPU(Image& image)`‡: routine to run the task on a GPU.
- `float computationalComplexity()`‡: returns a number denoting the computational complexity of the task. It is used by the engine to optimize the scheduling. The unit can be chosen by the user, since it is only the relative value which is important for the engine.
- `void update()`: should be called by the user before accessing the result of the image task. This method is necessary for being able to delay the effective execution of the task, as explained in Section 4.

The `ImageTask` class defines the most general image task which accepts multiple input images and returns an output image and a globalized result. More specific classes can be derived from it, such as a filter class which transforms one image into another one. Note that the user specifies the parameters and results of the image task as attributes of the class.

The `ParallelEngine` class defines the main object of the library to which the tasks are passed:

- `Image* execute(Image* images, ImageTask* imageTask)`: method by which a task is passed to the engine for execution.
- `ImgTask* create(int taskClassID)`†: an abstract method that should be filled in by the user. Defines an object factory by which tasks can be created according to their class ID.

For the ITK version, the `Image` class of ITK is used. For the library-independent version, a simple Image class is used which is defines as an array of pixels, a height and width.

Next, we introduce the essentials of the ITK library before explaining how the above is integrated in ITK.

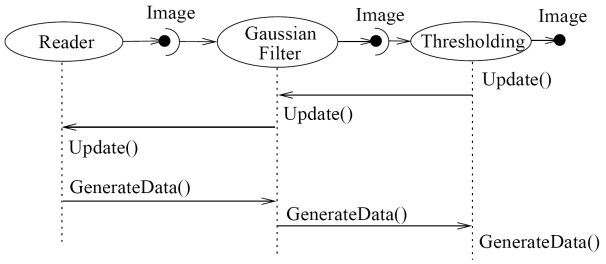

Figure 3. Sequence of the ITK data pipeline updating mechanism.

3.2 The ITK Library

ITK provides a powerful set of classes (such as for defining regions and iterators) so that efficient image processing algorithms can be implemented for 2 or multi-dimensional images. ITK inherently supports multithreading. One of the most common image operations is given by the `ImageToImageFilter` class, which transforms a given image. The following methods are relevant in the context of parallel processing:

- `GenerateData()`[‡]: performs the image processing sequentially.
- `ThreadedGenerateData(ImageRegion& regionForThread, int threadId)`[‡]: when this function is implemented by the user, the filter can be run multithreaded. The user specifies how the filter transforms the given image region. In that case, the `GenerateData()` method must not be implemented.
- `SetNumberOfThreads(int)`: with this method the user specifies in its application how many threads should be used when applying the filter.
- `Update()`: this causes the filter to be effectively executed. Actually, this invokes the data processing pipeline, as explained below.

Other image operations are defined similarly.

An important feature of ITK is the possibility to define data pipelines. Fig. 3 shows the pipeline execution of 3 image processing operations in ITK. The rationale is that an operation is not executed before an `Update()` method is called. At first, the user specifies the chain of operations by linking the output of operation with inputs of succeeding operations. In this way, the data objects and process objects are tied together. The pipeline is invoked whenever an `Update()` is called for one of the tasks along the chain. The `Update()` is passed along the chain and the `GenerateData()` is called.

3.3 Integration in ITK

Comparison of the interfaces shows that an `ImageToImageFilter` can easily be converted to an ImageTask class. The `ThreadedGenerateData` method gives the `execute` method. The following methods are added to the ITK `ImageToImageFilter` class and have to be implemented:

- `describe()`[†].
- `int border()`[‡].
- `Image executeOnGPU(Image& image)`[‡].

The filter objects should not be explicitly passed to the engine, this is done in the `Update()` method. The `SetNumberOfThreads(int)` method will also be called by the engine. If `ThreadedGenerateData` is not implemented, only the `GenerateData()` method, the parallel engine will only employ task parallelism.

3.4 Low-level Functions

The low-level library constitute functions for the following: image partitioning (taking into account an image border for neighborhood operations), image communication, scheduling algorithm, performance measurement and correctness checks.

4 Task Parallelism

When applying task parallelism, the tasks must be distributed among the available processors. Each task is mapped on one or more processors. This can happen dynamically, by scheduling the tasks one-by-one during the execution of the application. A more efficient mapping can be calculated if one knows in advance the tasks. One can then minimize communication. This is called interoperation optimization [12]. The task dependency graph is, however, not always a priori available. For some applications, tasks depend on the result of previous tasks. Then, initially, the task dependency graph can only be partly constructed. It will grow dynamically when new tasks are scheduled as a result of other tasks.

The dynamic creation of the task graph is done by the parallel engine as it receives tasks. For finding the most efficient schedule the engine should try to look as far as possible into the future. The parallel engine will therefore delay the execution of the tasks. When a task is passed to the engine, it is not executed immediately but used to create the graph. Only when the result of a task is really needed for determining the future course of the application, the pending tasks are executed. The execution is triggered by the `update()` method. When using the fully integrated ITK version of the library, the ITK pipeline mechanism can be exploited. This mechanism inherently delays the execution of the tasks, as was explained in Section 3.2. The task dependency graph is constructed during the execution of the backward update chain.

We used the algorithm described in [5] for homogenous multiprocessor scheduling. A series-parallel reduction of the graph is applied whenever possible [9, p. 30].

5 Discussion of Approach

In this section we will assess our approach. Seinstra [11] gives the following requirements for user transparent parallel image processing:

- *Low threshold, easy accessible*: parallelism is fully integrated in the popular ITK-library. Our approach fully exploits ITK's structure.
- *Maintainability*: the library must be extensible and easily maintainable. Therefore we defined 3 levels together with a library-independent version. The low-level functions are made available, so that high-level parallel solution can easily be built from them.
- *Availability*: applicable to commonly available parallel computers. Our approach supports multicores, clusters and GPUs.
- *Portability*: C/C++ is used in combination with MPI. The user does not have to learn a new parallel language.
- *Efficiency*: image processing algorithms have inherent concurrency which is relative simple to exploit.

Skeleton programming is extensively discussed in [1, 3]. They cite the following requirements for building successful generic solutions:

- *Propagate the concept with minimal conceptual disruption*: skeletons must be provided within existing programming environments without actually requiring the programmers to learn entirely new programming languages.
- *Accomodate diversity*: provide mechanisms to specialize skeletons. We made available the low-level functions of the library.
- *Show the payback*: a performance analysis is automatically performed and the attained speedup is returned to the user.
- *Support code reuse*: allow programmers to reuse with minimal effort existing sequential code. The definition of the image task class inherently allows the reuse of the sequential code. Instead of specifying the operation for an entire image, the user should specify the operation for a region of the image.
- *Handle target architecture heterogeneity*: our parallel engine can handle heterogeneous computing resources.
- *Handle dynamicity*, such as sudden unavailability of processing elements. Currently, the master regularly checks the state of each slave. If a slave does not respond, the tasks assigned to that slave are redistributed among the other slaves.

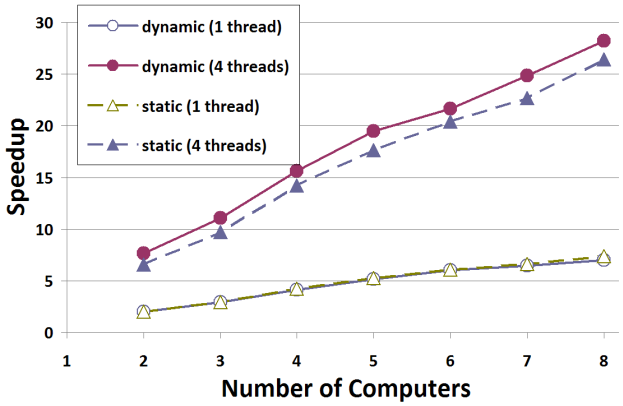

Figure 4. Speedup of parallel execution of the mosaicing algorithm: dynamic versus static scheduling and running with 1 or 4 threads.

6 Experiments

The proposed framework is developed in the context of a project dedicated to mosaicing of images of preclinical research [2]. In-vivo scanning of small animal subjects is performed by moving a fiber optic probe which is connected to a laser scanning unit along the animal's skin. By moving the probe, images are continuously captured, allowing the visualization of surfaces larger than the field of view of the probe. The mosaicing algorithm constructs a single static image from the multiple overlapping frames.

The algorithm consists of three steps. At the preprocessing step, the frames are denoised. Basically, this step constitutes of 9 separable convolutions. Secondly, succeeding frames in the sequence are matched (1 with 2, 2 with 3, etc.). This gives an estimation of the global position of each frame. In a third step, the global position is refined by considering other overlaps. The probe is moved in a zigzag way, so that image 1 overlaps with for example image 7 and 8. The stitching of 2 images, performed in the second and third step, needs 10 to 100 iterations, dependent on how fast the match is found. The computation time of the stitching is therefore variable. Each iteration consists of 2 separable convolutions and an interpolation. Note that the tasks in the third step depend on the results of the second step.

The experiments were performed on our cluster of 8 quadcores (Intel Core 2 Quad Q6700 2.66 GHz processors). Fig. 4 shows the speedup of running the mosaicing algorithm by the parallel engine. The mosaicing was done for 100 images of 518 by 426 pixels. The following run times were compared with that of single-threaded execution on one processor. The algorithm was executed on 2 to 8 processors, single-threaded and with 4 threads. Furthermore, we compared the 'static' scheduling algorithm explained in Section 4 with a 'dynamic' scheduling algorithm taking into account the unpredictability of the computation times of the tasks in step 2 and step 3 of the mosaicing algorithm. While the static algorithm calculates a mapping of the tasks

onto the processors which is then executed invariably, the dynamic algorithm simply passes tasks to idle processors. This latter is also called a farmer approach. In the former, the communication of images is minimized, while not in the latter; the images are passed together with the tasks and the results are collected each time by the master processor. So, the communication overhead is minimized for the static scheduling algorithm, while the idle time due to load imbalances is smaller for the dynamic scheduling algorithm.

Fig. 4 shows that the mosaicing algorithm is suited for efficient parallel processing; the overheads are low. The execution time is reduced from 90 minutes on a single core to 3.2 minutes fully exploiting the 8 quadcores. The dynamic scheduling algorithm is able to overcome load imbalances due to the unpredictable nature of the tasks' computation time. The idle time due to load imbalances outweighs the communication overhead so that the dynamic algorithm clearly performs better.

7 Conclusions

We present extensions for the current state-of-the-art on user-transparent parallel image processing. Our approach integrates with the ITK library. It maximally exploits the design of the library, such that only a little effort is demanded from the non-parallel expert user. The generic parallel algorithm is designed using object-oriented patterns instead of algorithmic skeletons. A parallel engine is constructed which is able to efficiently executing the image tasks on the available processing elements, exploiting task and data parallelism. Except for serialization[1], the engine can itself track the tasks' attributes which are necessary for parallelization. The task dependency graph is generated automatically during the execution of the application. Experimental results show that efficient parallel processing is attained.

The philosophy of our approach is the concretization of tasks by objects and the enhancement of a parallel engine with 'parallel intelligence'. In the future, we want to add a performance monitor to the engine which uses performance data of previous runs to optimize new runs of tasks and applications.

8 Acknowledgements

DMOBISA is an IBBT-project is cooperation with the following companies and organizations: ETRO, VisionLab, Janssen Pharmaceutica, SkyScan and DCILabs. IBBT is an independent multidisciplinary research institute founded by the Flemish Government to stimulate ICT innovation. Peter Schelkens is supported by a postdoctoral fellowship with the Fund for Scientific Research Flanders (FWO).

[1] Note that languages such as java support built-in object serialization.

References

[1] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming", *Parallel Computing*, 30(3), 2004, pp. 389–406.

[2] F. Cornelissen, S. D. Backer, J. Lemeire, B. Torfs, R. Nuydens, T. Meert, P. Schelkens, and P. Scheunders, "Fibered fluorescence microscopy (ffm) of intra epidermal nerve fibers–translational marker for peripheral neuropathies in preclinical research: processing and analysis of the data", In *Proc. of Applications of Digital Image Processing XXXI, part of SPIE Symposium on Optical Engineering and Applications*, 2008.

[3] M. Danelutto, "Second-generation skeleton systems", In *Proc. of the International Conference ParCo*, 2005.

[4] A. S. Dhrubajyoti Goswami and B. R. Preiss, "From design patterns to parallel architectural skeletons", *Journal of Parallel and Distributed Computing*, 62(4), 2002, pp. 669–695.

[5] K. Jansen and H. Zhang, "Scheduling malleable tasks with precedence constraints", In *17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2005)*, 2005.

[6] M. Moore, J. Sztipanovits, G. Karsai, and J. Nichols, "A model-integrated program synthesis environment for parallel/real-time image processing", In *SPIE Conf. on Parallel and Distributed Methods for Image Proc.*, 1997.

[7] P. J. Morrow, D. Crookes, T. J. Brown, G. McAleese, D. K. Roantree, and I. T. A. Spence, "Efficient implementation of a portable parallel programming model for image processing", *Concurrency - Practice and Experience*, 11(11), 1999, pp. 671–685.

[8] C. Nicolescu and P. Jonker, "A data and task parallel image processing environment", *Parallel Comput.*, 28(7-8), 2002, pp. 945–965.

[9] G. N. S. Prasanna and B. R. Musicus, "The optimal control approach to generalized multiprocessor scheduling", *Algorithmica*, 15(1), 1996, pp. 17–49.

[10] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[11] F. Seinstra and D. Koelma, "User transparancy: a fully sequential programming model for efficient data parallel image processing", *Concurrency and Computation: Practice and Experience*, 2004, pp. 611–644.

[12] F. Seinstra, D. Koelma, and A. Bagdanov, "Finite state machine-based optimization of data parallel regular domain problems applied in low-level image processing", *IEEE Transactions on Parallel and Distributed Systems*, 15(10), 2004, pp. 865–877.

[13] J. Serot, D. Ginhac, and J. pierre Derutin, "Skipper - a skeleton-based parallel programming environment for real-time image processing applications", In *Proc. of the Int. Parallel Computing Technologies Conference, St-Petersburg*, 1999.

[14] J. L. Sobral, "Skelj: Skeletons for object-oriented applications", In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *PPAM*, volume 4967 of *Lecture Notes in Computer Science*, pp. 1114–1121. Springer, 2007.