This is a postprint version of the following published document:

García-Blas, J., Del Río Astorga, D., Carretero, J., García, J. D. (2020). Towards enhanced MRI by using a multiple back end programming framework. *Future generation computer systems*, 112, pp. 467-477.

DOI:https://doi.org/10.1016/j.future.2020.05.039

© 2020 Elsevier B.V. All rights reserved.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Towards enhanced MRI by using a multiple back end programming framework

Javier Garcia-Blas[a], David del Rio Astorga[a], Jesus Carretero[a], J. Daniel Garcia[a]

[a] *Universidad Carlos III de Madrid, Spain*

Abstract

In recent years, on-line processing of data streams (DaSP) has been established as a major computing paradigm. This is due mainly to two reasons: first, more and more data that are generated in near real-time need to be processed; the second reason is given by the need of efficient parallel applications. However, the above-mentioned areas expose a tough challenge over traditional data-analysis techniques, which have been forced to evolve to a stream perspective.

In this work, we apply a novel multiple back end programming framework for stream data and task based parallelism to a multi-staged diffusion magnetic resonance imaging (MRI) ~~application~~ toolkit, named *pHARDI*. The results demonstrate the benefits of using our framework in terms of performance and memory usage. The evaluation carried out also depicts that the speed-up of our parallel framework increases with the problem size.

Keywords: Data stream processing, GrPPI, Task-level parallelism, Data-level parallelism, MRI reconstruction

1. Introduction

High-level parallel programming models can help application developers to access and use resources without the need to manage low-level architectural entities, as a parallel programming model defines a set of programming

Email addresses: `fjblas@inf.uc3m.es` (Javier Garcia-Blas), `david.rio@uc3m.es` (David del Rio Astorga), `jesus.carretero@uc3m.es` (Jesus Carretero), `josedaniel.garcia@uc3m.es` (J. Daniel Garcia)

abstractions that simplify the way the programmer structures and expresses an algorithm.

Typically, both analysis and data processing have been carried out in a batch fashion (i.e. the different computing iterations were applied over a set of stored data). However, in recent years, domains that need a set of constantly refreshed information have gained greater importance. For instance, this is the case of application fields such as scientific computing research [1], environmental research by means of sensor networks, social network analytics, and many others. In these research areas, batch techniques cannot be applied since those techniques do not meet their computing needs.

The lack of effective tools for approaching this kind of problems has been solved by evolving traditional paradigm to the *stream processing* paradigm. In this paradigm, a constant flow of information retrieved from a set of sources needs to be analyzed/processed and the results must be reported in near real-time. Another outstanding need of this kind of problems is the urgency for retrieving the computed results because they either are used in decision-making processes, or are meaningful only during a limited time frame. The trend of continuous data flow processing under time constraints has been named *near real-time stream processing*. Moreover, the industry and the academic community have developed High Throughput Computing (HTC) techniques to provide solutions for these scenarios. HTC is the data-intensive variant of HPC.

Fast data processing is a fundamental requirement in stream processing. To achieve a high processing throughput, there is a need to replace the batch-like typical approach to novel strategies. One of these new strategies is pipeline processing, consisting in reading the incoming data and processing them through a sequence of user-defined stages.

In a previous work [2], we presented a comparative study of the generic interface for parallel patterns, namely GrPPI, under a real use case. By using performance and hardware metrics, we compared a stream-based diffusion magnetic resonance imaging (MRI) ~~application~~ toolkit, named pHARDI, under different aspects such as CPU time, cache efficiency, and memory consumption.

The main contribution of this paper is to present the benefits of using a novel data and task-based runtime for data streaming below MRI applications. The proposed solution aims to increase productivity by employing well-known parallel patterns that encapsulate algorithm features in parallel applications. The presented approach permits to execute the application under different parallel platforms with minimal modifications of the source code. In addition, we present an extended evaluation that compares this new execution mode with other existing ones, studying the impact of the size for two different datasets.

The rest of the paper is structured as follows. Section 2 introduces streaming parallel patterns as well as GrPPI, a generic interface to those patterns. Section 3 presents our task-based parallelism approach. Section 4 introduces pHARDI, our diffusion magnetic resonance imaging toolkit. In Section 5, we discuss about the experimental evaluation carried out. Section 6 revisits related works that intersect with the contributions presented in this paper. Finally, Section 7 presents major concluding remarks of this work.

2. Background

In this section, we introduce the pipeline and farm streaming parallel patterns as well as the GrPPI interface to those patterns.

2.1. Streaming parallel patterns

In general, DaSP applications can be seen as data-flow computations in the form of directed acyclic graphs (DAG), where the source node (*producer*) gets items from some input stream, intermediate nodes perform some transformation on them, and a sink node (*consumer*) dumps transformed items to an output stream. To accelerate these applications, the nodes can be executed in parallel as long as data item dependencies are preserved. As observed in the literature [3, 4], a common and simple DAG construction in DaSP applications is the *pipeline*, where the nodes in a topological ordering have data item dependencies only with the previous node. Another common construction is the farm pattern, where the transformation in a node is replicated n times to increase its throughput. This allows for multiple stream items to be computed in parallel.

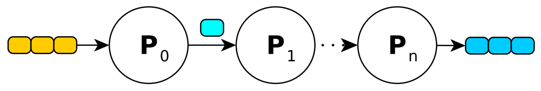
The formal definitions of the pipeline and farm patterns are the following:

- Pipeline. Items from the input stream are processed in several sequential stages. Each stage processes data produced by the previous stage and delivers results to the next one. Provided that the i-th stage in a n-staged pipeline computes the function $f_i : \alpha \to \beta$, the pipeline delivers the item x_i to the output stream applying the function

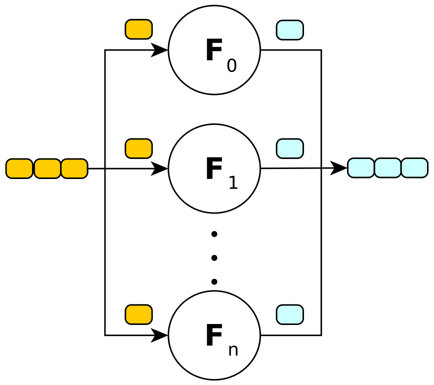
$f_n(f_{n-1}(\ldots f_1(x_i)\ldots))$ (i.e. function composition). The main requirement of this pattern is that the transformations in the stages should be independent from each other, i.e., they can be computed in parallel without side effects. The parallel implementation of this pattern is performed by using a set of concurrent entities, each of them taking care of a single stage. Figure 1(a) shows a pipeline diagram.

- Farm. Transformation $f : \alpha \to \beta$ is computed in parallel over all items from the input stream. Two items x_i and x_j can be transformed in parallel producing $f(x_i)$ and $f(x_j)$. Consequently, transformation of items need to be completely independent from each other. Figure 1(b) shows a farm diagram.

Both pipeline and farm patterns can be composed to produce more efficient applications. Basically, the compositions supported between the pipeline and farm patterns are those in which the pipeline stages can be parallelized individually using the farm pattern. Thus, if a pipeline stage corresponds with a pure function, this can be computed in parallel following a farm construct. Throughout this paper, we denote the sequential stages of a pipeline with "`p`", the farm stages with "`f`" and the communication between two stages with the symbol "`|`". For instance, a pipeline comprised of 4 stages, where the second and the third are farm stages, is represented by "`(p|f|f|p)`".



(a) pipeline pattern.



(b) farm pattern.

Figure 1: pipeline and farm pattern diagrams.

2.2. GrPPI*, a generic parallel pattern interface*

In order to provide a generic interface to parallel patterns we have defined GrPPI[1], a generic and reusable parallel pattern interface for C++ applications [5]. This interface takes full advantage of modern C++ features, metaprogramming concepts, and generic programming to act as switch between the OpenMP, C++ threads, Intel TBB, and FastFlow parallel programming models. Its design allows users to make use of the aforementioned execution frameworks from a unified and compact interface, hiding away the complexity behind the use of different implementation mechanisms. Furthermore, the modularity of GrPPI permits to easily integrate new patterns, while combining them to arrange more complex constructs.

GrPPI accommodates a layer between the application programmer and the different programming models. In this way, GrPPI can be used to implement a wide range of existing stream-processing and data-intensive applications with relatively small effort, having as a result portable codes that can be executed on multiple platforms.

Currently, GrPPI provides the following parallel programming patterns for the aforementioned back-ends:

- *Data patterns*: Express computations over a data set.
    - map, reduce, map/reduce, stencil.
- *Task patterns*: Express task composition.
    - divide/conquer.
- Streaming patterns: Express computations as a (possibly unbounded) data set.
    - pipeline.
    - Specialized stages: farm, filter, reduction, iteration

Listing 1 shows the pipeline and farm GrPPI C++ pattern generic functions. We make use of generic programming with *variadic templates* and *forwarding references* to take multiple callable entities (e.g. a function, a function object, or a lambda expression) as transformations. Note as well

[1]`https://github.com/arcosuc3m/grppi`

that the first parameter specifies the execution policy that shall be used to execute the operators. This allows to separate the programming model used as a back-end from the specific configuration of the computations.

Listing 1: Pipeline and farm interface.

```
template <typename E, typename G, typename ... Ts>
void pipeline(const E & execution, G && generator, Ts && ... transformers);

template <typename T>
farm_t<T> farm(int replicas, T && transformer);
```

Listing 2 depicts a pipeline where the first stage invokes `read_item` to get the next item in the stream. The second stage applies to each item the transformation `f()` with four parallel replicas. The third stage applies to each item the transformation `g()`. Finally, the fourth stage invokes `write_item` which writes each item to an output stream.

Listing 2: Pipeline and farm usage example.

```
parallel_execution_omp ex;
pipeline(ex,
  read_item,
  farm(4,[](auto && x) { return f(x); }),
  [](auto && x) { return g(x); },
  write_item
);
```

3. Generic task-based parallelism for multiple back-ends

In this paper, we present a novel back-end that follows a task-based parallelization approach integrated into GrPPI. The main difference with respect to the already implemented back-end using ISO C++ threads is that, instead of employing a thread per stage, this back-end leverages a pool of threads that executes the set of generated tasks. Each task represents the execution of a stage function to a single data item. Therefore, instead of moving the data across the different thread stages, the tasks are assigned dynamically to each worker. This way, the proposed back-end leads to improved resource exploitation in unbalanced scenarios.

This back-end is comprised of four different components:

- Task queue: With the aim of enabling communication and synchronization mechanisms, we provide a First-In-First-Out (FIFO) queue implemented as a circular bounded buffer that supports concurrent accesses for multiple producers and multiple consumers inspired by the

Michael and Scott's lock-free queue [6]. In such a way, multiple farm threads can concurrently access to the queue. Execution order of each task is determined by the Task generator.

- **Scheduler**: The scheduler is in charge of distributing the tasks among the different workers. This scheduler contains a task queue that stores the pending tasks and assigns them to a free worker in the pool of threads. In this paper, we have implemented a FIFO scheduler but, following the flexibility ideals of GrPPI, it can be modified using other scheduling policies by modifying a single argument on the policy construction. For instance, the FIFO scheduler can be extended using multiple task queues, one per NUMA-node in the platform to improve data locality by prioritizing the execution of tasks on the corresponding queue to that threads that are running on a given NUMA-node. Listing 3 shows an example of constructing a parallel execution task policy using the proposed FIFO scheduler.

Listing 3: Task back-end construction.

```
parallel_execution_task ex{fifo_scheduler{}};
```

- **Data manager**: The data manager is in charge of receiving the data items and storing them onto main memory providing a data location ID. This way, the worker thread can access easily to the data related to the task that is executing. In this version, we only make use of main memory storage but it can be extended in a future version to support the use of long-term storage.

- **Task generator**: The task generator is in charge of generating the initial tasks of a given parallel pattern. We distinguish two data sources: i) *data source*: data are available and the number of elements is known prior pattern execution, e.g. in a container, where the different data items can be also accessed in a parallel way, and ii) *stream source*: the number of elements is not known and requires an initial sequential access to the data items. Then, depending on the source type, all tasks are generated at the beginning of the pattern execution (data source) or they are generated as soon as the data items become available (stream source). It is important to highlight that the task granularity is user-defined, considering the data item size read from the source (generator function). The current implementation does not support failure recovery of tasks, since it is currently designed for shared memory systems, and therefore, its behaviour is similar to other solutions such as Intel TBB or OpenMP. In the future, we plan to support this functionality in a similar way to Big Data frameworks such as Apache Spark or Apache Flink, in where tasks are re-scheduled in presence of failures.

Algorithm 1 Task generation algorithm.

```
$Id = 0$
**if** $Generator = Data\_Source$ **then**
  **for all** $DataItem \in Data\_Source$ **do**
    $Task = \{DataItem.location, Id\}$
    $Scheduler.produce\_task(Task)$
    $Id{+}{+}$
  **end for**
**end if**
**if** $Generator = Stream\_Source$ **then**
  **while** $(DataItem = Stream\_Source.read()) \neq Empty$ **do**
    $Task = \{DataItem.location, Id\}$
    $Scheduler.produce\_task(Task)$
    $Id{+}{+}$
  **end while**
**end if**
$Scheduler.communicate\_end()$
```

Algorithm 1 describes the GrPPI task generation process depending on the data source type. As observed, in the case of a data source, where the data are available and the number of items is known prior to pattern execution, the generator creates a task per data element contained in the data source and send them to the scheduler as they appear. Otherwise, if the number of data items is unknown, i.e. data are read from a data stream, the generation task produces a task per data item that is acquired from the data source and it sends them to the scheduler. In any case, the scheduler will process all tasks until the end of the stream is reached (in case of a stream source) or all tasks are finished (data source). Finally, after creating the last task, the generator will indicate to the scheduler that it will not generate more tasks and it can finalize its execution as soon as all the pending tasks complete their execution.

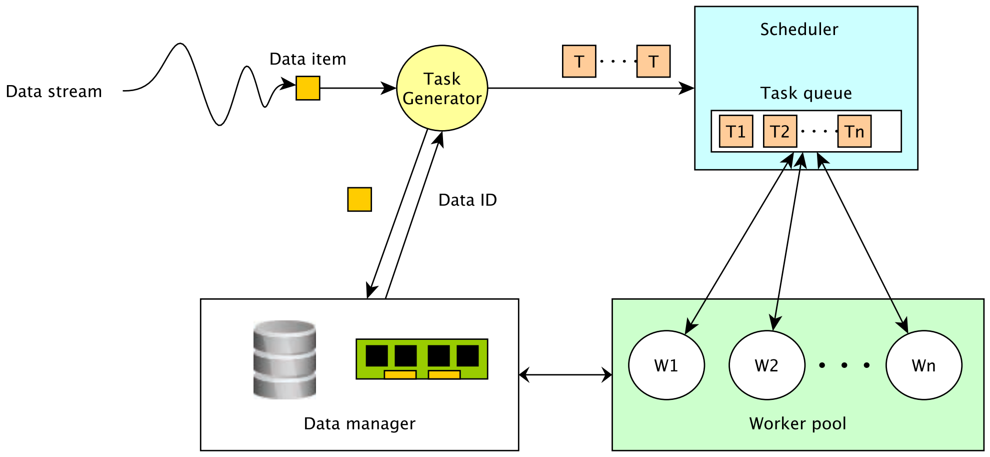


Figure 2: GrPPI task back-end schema.

Figure 2 represents the general schema of the proposed task back-end using a data stream as data source. As observed in the figure, each time a new data item arrives at the input stream, the task generation stores the read data onto the main memory through the data manager. Afterwards, the data manager returns an ID to identify the data location and the task generator produces a task which is stored into a task queue. Simultaneously, the workers ask to the scheduler for new tasks. When a worker obtains a given task, it accesses to the data manager and obtains the data using the data location of the task. Finally, if the finished task is not a sink, the worker stores the resulting data through the data manager and generates the next task, otherwise, it notifies the scheduler that the data item has been consumed and asks for a new task.

4. Use case: diffusion magnetic resonance imaging toolkit

Diffusion magnetic resonance imaging is a non-invasive technique capable of quantifying the diffusion process of water molecules in living biological tissues. Its main application is the study of the local geometry and wiring pattern of the human brain white matter. A large number of neuroscience research studies and clinical applications have been conducted in the last decade [7]. Many of these studies are based on different intra-voxel (volumetric pixel) models of molecular diffusion, which in turn, require different sampling schemes to collect the data and fitting algorithms.

In order to facilitate the widespread use of the molecular diffusion technique, in a previous work [8], we have developed a novel toolkit called **pHARDI** and a porting of the Matlab application to high-performance C++ codes: CPU/GPU-accelerated spherical deconvolution of diffusion MRI data. The purpose of pHARDI[2] is twofold: (1) to provide in a single toolkit an extensive and diverse set of reconstruction methods for different sampling protocols, and (2) to accelerate the reconstruction process by means of high quality linear algebra libraries. The toolkit has a layer-based design allowing to parallelize the computations via multiple accelerators in a wide range of devices, including co-processors, multi-core CPU, and GPU devices. Including new methods in pHARDI is facilitated by the modular design of the toolbox. Experimental evaluation showed that pHARDI attains, on average, a speed-up of 8× over equivalent Matlab implementations.

Figure 3 shows the five stages in which the pHARDI implementation is structured:

- Stage 1: an initial transformation from 4D volume represented in NIfTI [9] to a matrix format.
- Stage 2: this stage reduces the computation by applying a mask over the white matter region of the brain. After that, resulting voxels (3D volumetric pixels) are transformed into a matrix, which considers the directions of each track. The size of this matrix determines the computational cost for future tasks.
- Stage 3: it is the most time consuming part. This stage reconstructs each slice of the volume in all the directions provided in the input files.
- Stage 4: this stage aggregates the partial results (i.e., slices) into a final reconstructed volume. Each slice can be included in the final volume without interference (embarrassing parallelism).
- Stage 5: this final step is in charge of transforming the resulting matrix into a final NIfTI representation.

[2]Available at `https://github.com/arcosuc3m/phardi`

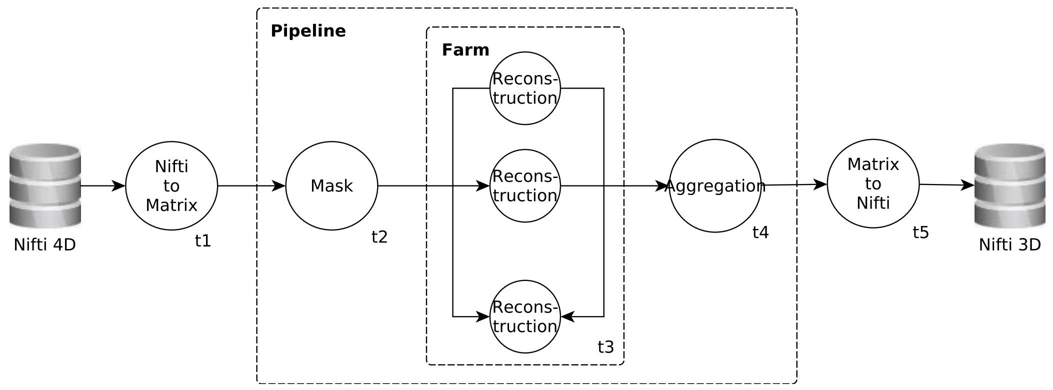


Figure 3: Parallel patterns identified in pHARDI.

Stages 2, 3 and 4 can be represented as a *pipeline* worflow, in which each stage of the pipeline pattern can be executed independently. Additionally, Stage 3 can be accelerated using a *farm* pattern as it is computationally more expensive than the others. Figure 3 also shows how Stage 3 is executed in parallel by multiple tasks using the Farm parallel pattern. In this work, we demonstrate the advantage of stream processing for both data and task-based parallelism using a multi-staged algorithm by using GrPPI.

In case of task-based parallelism, the chosen granularity corresponds with each slide of the brain. We assume that this granularity level enables a better performance and also reduces the necessary lines of code for using GrPPI. Each slide represent an independent piece of work.

With the aim of showing the changes done to migrate a version from OpenMP to a GrPPI-based implementation, we compare the modified source code in the following link[3]. As can be seen, the main for loop is migrated to a pipeline pattern, which orchestrates the execution of Stages 2, 3 and 4.

5. Experimental evaluation

The evaluation has been carried out by running the pHARDI MRI use case on a machine consisting of two multi-core Intel Xeon E5-2630 v3 processor with a total of 8 physical cores running at 2.40 GHz, hyper-threading activated, equipped with 128 GB of RAM, and executing Linux Ubuntu 18.10

[3]`https://github.com/arcosuc3m/phardi/commit/ed4229c2a9f39750ddf96119d909053211643e5f#diff-25108bf27823f3ed747c76deef718afe`

x64. The compiler used is GCC 7.0. After that, the source code has been compiled using both `-O3` and `-DNDEBUG` flags. In all the cases, we show the metrics for an execution with up to 64 threads. The source code of the modified version of pHARDI using GrPPI is publicly available[4].

In order to validate the obtained output, in each iteration we printed and compared the SNR (signal-noise ratio) metric of each slide with the baseline solution.

5.1. Dataset selection

We run the toolkit using a real diffusion MRI dataset[5] acquired from healthy subjects. Specifically, whole-brain HARDI data were acquired in a 3T Philips Achieva scanner (Sant Pau Hospital, Barcelona) with a 8-channel head coil along 100 different gradient directions on the sphere in q-space with constant $b = 2000\,\mathrm{s/mm}^2$. Additionally, $1b = 0$ volume was acquired with in-plane resolution of $2.0 \times 2.0\,\mathrm{mm}^2$ and slice thickness of 2 mm. The acquisition was carried out without undersampling in the k-space (i.e., $R = 1$). We use two dataset to evaluate the behaviour of our programming framework: 1) small dataset composed by 101 volumes of $128 \times 128 \times 60$ voxels/pixels, resulting in a file of 117 MiB; 2) large dataset composed by 288 volumes of $145 \times 174 \times 145$ voxels/pixels, resulting in a file of 5 GiB. We have executed pHARDI with those datasets with the aim of studying the impact of the problem size.

5.2. Performance metrics analysis

This section summarizes the performance metrics analysis carried out for the pHARDI ~~application~~ toolkit using the small dataset. In this section multiple comparative metrics are shown. We compare the sequential baseline implementation with a manual accelerated version implemented by us using OpenMP (OpenMP in tables) and GrPPI under different back ends, such as sequential (SEQ), OpenMP (OMP), Intel TBB (TBB), C++ native threads (THR), Task-based, and FastFlow (FastFlow).

We have employed *perf* [10] for collecting both execution time and hardware performance counters. Table 1 summarizes the metrics collected for this work with a small description.

[4]pHARDI using GrPPI can be found at `https://github.com/arcosuc3m/phardi.git` under branch 'gppi'

[5]`https://doi.org/10.5281/zenodo.1194253`

Table 1: Performance and hardware counters employed is this study.

Metric	Description
PPRTI	Elapsed real (wall clock) time used by the process.
PPSTI	Total number of CPU-seconds used by the system.
PPUTI	Total number of CPU-seconds that the process used directly in user mode.
PPMRS	Maximum resident memory of the process.
PPVCS	Number voluntary context switches.
PPICS	Number involuntary context-switches.
PPCRF	Number of cache references.
PPCMS	Number of cache misses.

5.2.1. Execution time

We observe from the metrics related to execution time that GrPPI versions are as competitive as the OpenMP-based version. Besides, there are also project-based speed metrics such as PPRTI, PPSTI and PPUTI, which were measured with *time* utility from the Linux platform. The results obtained from 20 measurements were averaged (the method was the same for every other run-time metrics as well). The final values of the metrics are shown in Figure 4.

PPRTI clearly indicates a reduction in execution time for the parallel part of the code as well as for the whole application. PPSTI and PPUTI refer to the total time consumed by the process. We also highlight the obtained value for TBB-based back end, which is motivated by the fact that Intel TBB is task based and its scheduler maps those tasks to available hardware threads with a work stealing policy.

5.2.2. Memory usage

In Table 2 we compare the total memory consumption of the pHARDI ~~application~~ toolkit under different back ends and OpenMP. We noticed that the parallelized versions need roughly the same amount of resources compared to the baseline version, according to the process maximum resident set size. This means that parallelization does not really influence memory usage for this use case.

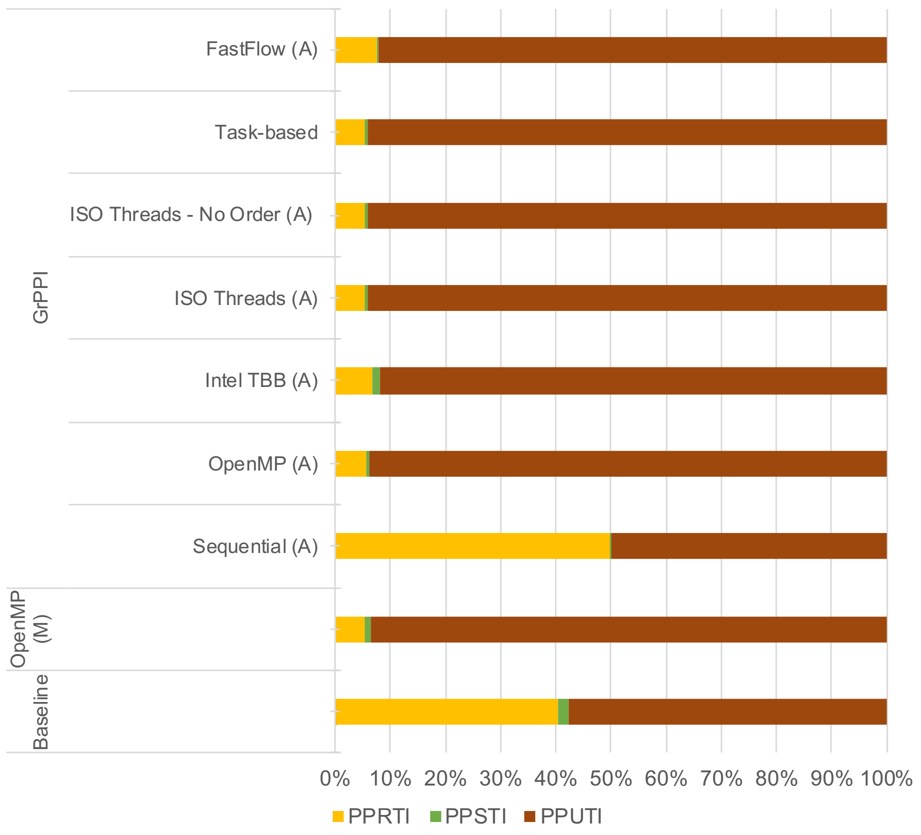


Figure 4: Metrics related to the execution time. **M** represents manual parallelization while **A** represents that parallelization has been carried out in an automatic way by using GrPPI.

Table 2: Metrics related to the RAM usage.

Implementation type	$128 \times 128 \times 60 \times 101$ **voxels PPMRS [Kilobytes]**
baseline	3,181,896
OpenMP (M)	2,917,100
GrPPI + SEQ (A)	1,876,960
GrPPI + OMP (A)	3,763,740
GrPPI + TBB (A)	**3,092,020**
GrPPI + THR (A)	3,741,500
GrPPI + THR No order (A)	3,734,364
GrPPI + Task-based (A)	3,120,440
GrPPI + FastFlow (A)	4,467,348

The less memory consuming version is the pipelined sequential version based on GrPPI. This is motivated by the fact that this version does not require queues for forwarding data between stages. We also conclude that the most memory consuming approach is GrPPI on top of FastFlow. The implementation of the back end of FastFlow requires more temporal copies than the other back ends. We have to highlight the efficient memory management implemented in Intel TBB, which is provided by Intel TBB scalable allocator (*tbbmalloc*) [11]. It is also important to note that the proposed task-based back end implemented in GrPPI obtain a similar memory consumption compared with Intel TBB. This is due to the memory saved in the pool of threads of the task-based back end.

5.2.3. Context switching

The results related to context switching show that parallelization causes an increase in PPVCS and PPICS metrics as it was expected (see Table 3). As shown, the solution with large amount of context switches is the baseline version of the applications, being two orders of magnitude larger.

Table 3: Metrics related to context switching. The table shows the number of voluntary (PPVCS) and involuntary (PPICS) context switches.

Implementation	$128 \times 128 \times 60 \times 101$ **voxels**	
type	**PPVCS**	**PPICS**
baseline	2,178,064	4,115,107
OpenMP (M)	12,690	104,981
GrPPI + SEQ (A)	1,678	246,959
GrPPI + OMP (A)	**4,488**	**286,909**
GrPPI + TBB (A)	10,979	332,388
GrPPI + THR (A)	6,570	294,722
GrPPI + Task-based (A)	6,900	412,007
GrPPI + THR No order (A)	6,818	468,235
GrPPI + FastFlow (A)	9,258	367,618

We can observe from the table that GrPPI reduces significantly the amount of context switching. This fact can affect positively the execution of parallel applications in cloud-based environment, where multiple applications can coexist in the same physical node (i.e., virtual environments).

5.2.4. Cache and instructions

We measured cache performance and instructions per cycle as summarized in Table 4.

Table 4: Metrics related to cache performance. The table shows the number of cache references and the number of cache misses, respectively.

Implementation type	$128 \times 128 \times 60 \times 101$ **voxels**	
	PPCRF	**PPCMS**
baseline	5,836	1,948
OpenMP (M)	11,816	4,027
GrPPI + SEQ (A)	10,635	2,826
GrPPI + OMP (A)	12,395	4,309
GrPPI + TBB (A)	8,472	**3,275**
GrPPI + THR (A)	12,291	4,285
GrPPI + THR No order (A)	12,317	4,306
GrPPI + Task-based (A)	9,894	4,295
GrPPI + FastFlow (A)	**8,170**	3,197

Given the results, we remark that GrPPI-based versions increase the number of cache misses (PPCMS) metric. However, given the number of cores employed, it is not a significant amount, even in presence of thread oversubscription.

5.3. Performance analysis

In this subsection, we evaluate the overall execution time and speed-up reached by a refactorized version of the ported implementation of pHARDI. It is important to mention that in case of Intel TBB and FastFlow, we have turned off the unneeded cores at system level [6]. However, it is difficult to predict the interference between both pipeline and farms threads.

[6] *echo 0 > /sys/devices/system/cpu/cpu0/online*

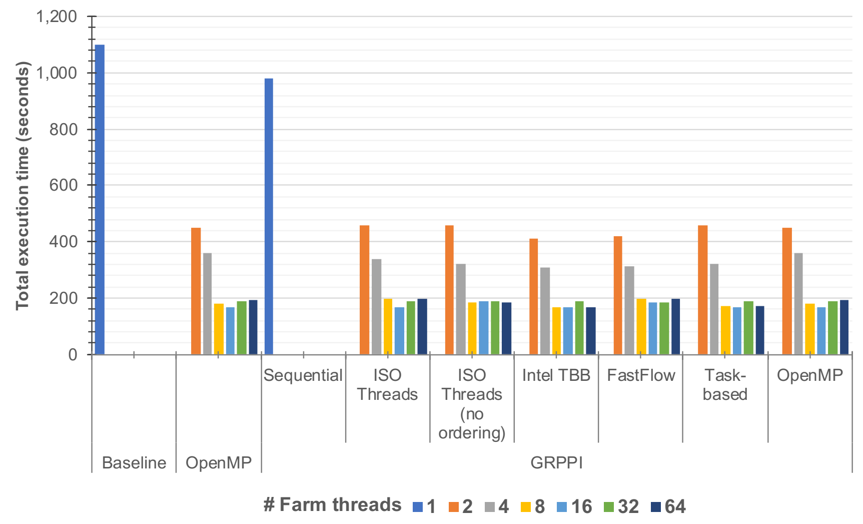


Figure 5: Performance evaluation over five GrPPI's back ends and a manually implemented version of pHARDI using OpenMP. Small dataset composed by 101 volumes of $128 \times 128 \times 60$ voxels.

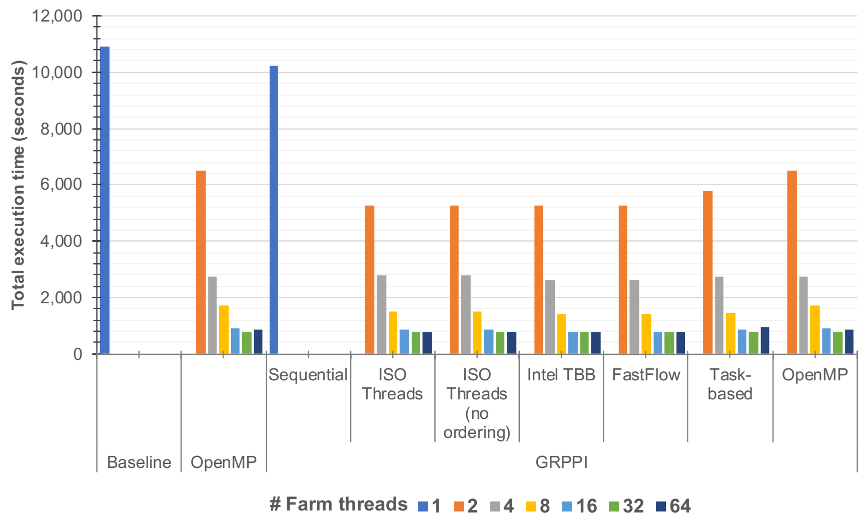


Figure 6: Performance evaluation over five GrPPI's back ends and a manually implemented version of pHARDI using OpenMP. Large dataset composed by 288 volumes of $145 \times 174 \times 145$ voxels.

Figures 5 and 6 plot the total execution time of pHARDI, comparing different back ends of GrPPI and OpenMP, over a small and big dataset, respectively. We evaluate an increasing number of farm threads of Stage 3 of the pipeline. We can observe that GrPPI scales with an increasing number of threads, reaching its minimum execution time with 16 threads. This number corresponds with the number of physical cores of the employed machine.

Equations 1, 2 and 3 depict the theoretical sequential and parallel execution times. N corresponds with the number of volumes. K represents the number of active cores involved in the parallel execution. $t1$ to $t5$ correspond with the stage in the defined pipeline in terms of computation time.

Given the limitations of transforming NIfTI-based data, Stages 1 and 5 run sequentially in both sequential and parallel implementations. Due to the use of the pipeline parallel pattern, we have experimentally characterized the execution times of Stages 2 and 4 and we conclude that those stages can be ignored.

$$T_{SEQ} = t_1 + N \times (t_2 + t_3 + t_4) + t_5 \tag{1}$$

$$T'_{SEQ} = t_1 + t_5 + N \times max(t_2 + t_3 + t_4) \simeq t_1 + t_5 + N \times t_3 \tag{2}$$

$$T_{PAR} = t_1 + t_5 + N \times max(t_2 + \frac{t_3}{K} + t_4) \simeq t_1 + t_5 + \frac{N}{K} \times t_3 \tag{3}$$

$$S = \frac{t_1 + t_5 + N \times t_3}{t_1 + t_5 + \frac{N}{K} \times t_3} \tag{4}$$

Figures 7 and 8 plot the achieved speed-up of pHARDI under different back ends compared with the baseline implementation based on sequential C++. The figure shows the theoretical speed-up calculated by using Equation 4. We highlight that most of the back ends outperform the theoretical maximum speed-up. This is mainly due to the cache behavior under the parallelized scenario. GrPPI's back ends are not affected by the increment of concurrent threads, maintaining the values of metrics like number of L1[7] data cache misses. Furthermore, the increase of hit ratio in L2 and L3 in the cache motivates the performance over the theoretical speed-up ~~(black line)~~. We also notice that the obtained speed-up increases with the problem size, reaching an acceleration near to 15× over the baseline version. This is due

[7]CPU cache is divided into three main "Levels", L1, L2, and L3. The hierarchy here is again according to the speed, and thus, the size of the cache.

to the utilization factor of the cores during Stages 2, 3 and 4 of the executed pipeline (as represented in Figure 3).

We can observe that the reached speed-up of the OpenMP version is close to the theoretical metric. We also observe that both OpenMP and GrPPI-based versions outperform this theoretical bound due to a higher hit rate (in both L2 and L3 cache levels). However, when we employ more threads than physical cores, this phenomenon disappears due to NUMA locally effects [12].

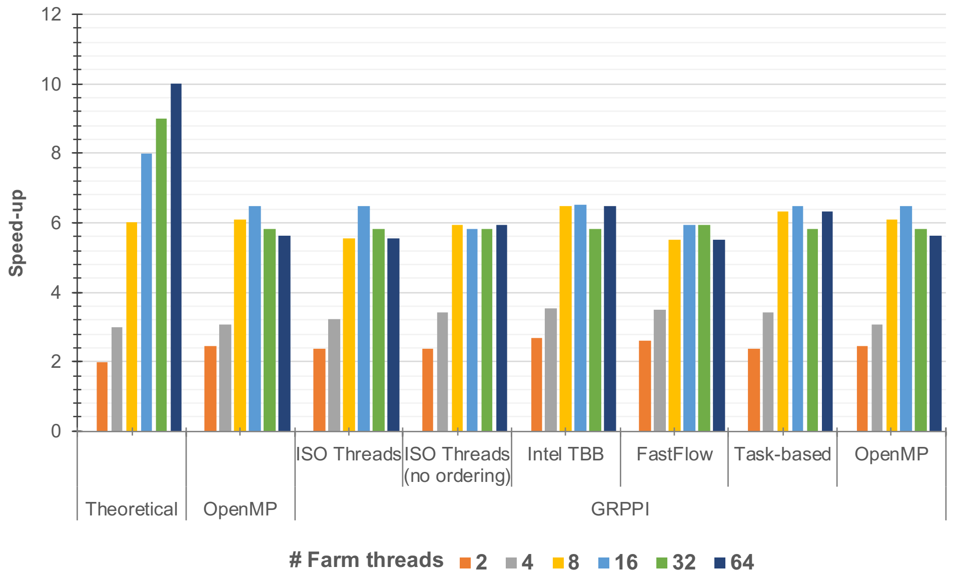


Figure 7: Performance evaluation a small dataset over six GrPPI's back ends of pHARDI and its speed-up compared with the baseline solution. Small dataset.

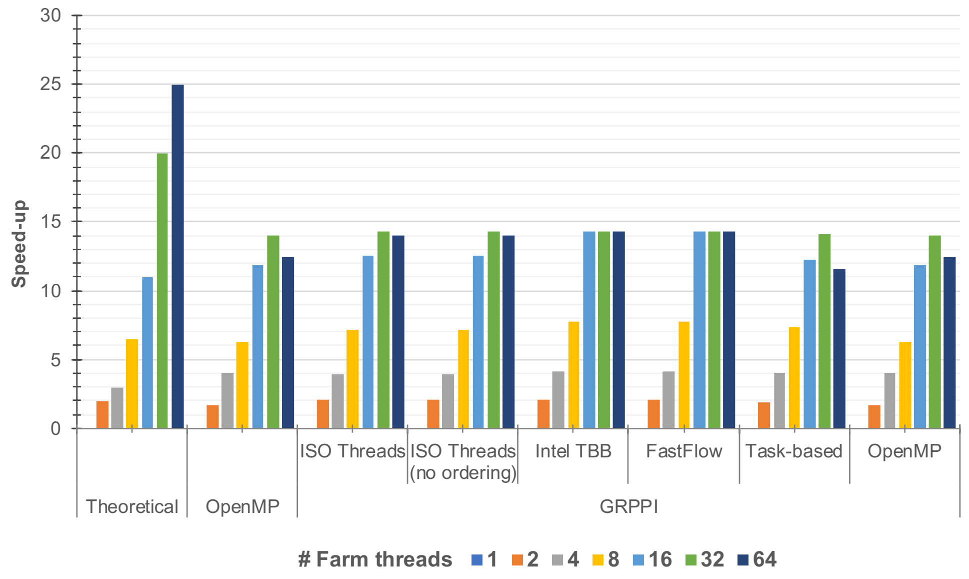


Figure 8: Performance evaluation of a large dataset over six GrPPI's back ends of pHARDI and its speed-up compared with the baseline solution.

In a nutshell, we conclude that the pattern implementations offered by GrPPI help improving both productivity and maintainability of DaSP parallel applications while obtaining the same performance as using directly the supported programming frameworks.

6. Related work

In this section, we show some of the most relevant works published that are related to our work.

6.1. Parallel Patterns

Due to the complexity of parallel programming, the advantages of using design patterns for the development of parallel applications were soon identified [13, 14, 15], so as their importance as parallelism enablers [16, 17, 18]. As a result, numerous efforts of parallel patterns targeted to modern architectures for developing data stream applications have been proposed. These efforts can be classified into the two following categories: *i)* data stream processing engines and *ii)* pattern-based parallel programming frameworks.

In recent years, many engines for shared-memory and distributed Autonomic Solutions for Parallel and Distributed Data Stream have been developed, such as Storm [19], Apache Spark [20], Apache Flink [21] and StreamIt [22]. Basically, applications implemented using these engines are represented as directed flow graphs, where nodes represent operators and the edges the stream flow. According to how the operators, or nodes in the graph, and the edges are defined, different and complex operations for filtering, splitting, and joining streams can be performed. Another example of streamed-data computing is Dispel [23]. Dispel is a strongly-typed imperative programming language used to construct logical descriptions of streamed-data workflows to be executed on distributed architectures. We differ from Dispel in that we use the existing C++ language, which is widely used in the industrial and scientific community, instead of a custom DSL. Additionally, Dispel addresses parallel platforms based on distributed memory architectures. Finally, a similar approach to the proposed in this paper is the presented by Pop et al. [24], which present a stream-computing extension to OpenMP. This approach takes advantage of OpenMP-based *pragma* notations to expose data, task, and pipeline parallelism. This solution orchestrates the execution on parallel sections defining the input/output dependencies of each parallel block. In terms of syntax, this OpenMP extension employs *input* and *output* clauses while GrPPI employs C++ callable entities or lambdas, in which both *input* and *output* items correspond with the function parameters. As GrPPI includes a OpenMP back-end, this extension could be employed, significantly improving the performance. As far as authors know, this approach is not publicly available.

On the other hand, a number of parallel-pattern interfaces have emerged oriented to multi-core processors and heterogeneous architectures. Threading Building Blocks (TBB) [25] offers a generic interface that includes stream parallelism. However, GrPPI ~~requires~~ provides more powerful type deduction capabilities which allows for simpler user code. Additionally, TBB requires linking with a specific library while GrPPI is a header only library. FastFlow [26] provides a more classic object oriented interface with less generic programming capabilities. Additionally, its implementation is thread oriented instead of allowing task parallelism as TBB or GrPPI allow. RaftLib [27] is specifically focused on parallel stream processing however its programming interface requires that application programmers define their own types that need to combine both generic programming and dynamic polymorphism making it some cases that user code is hard for the average programmer. In contrast GrPPI does not usually require the application programmer to define their own types and making use of simple lambda expressions is enough. Kanga [28] also provides an object oriented interface requiring more application code to express parallelism. One approach to address heterogeneity is SkePU [29]. However, two important limitations of SkePU are the need of a custom source to source precompiler and the additional constraints on the code that can be used in a computational kernel. In contrast, GrPPI is capable to handle any source code that is C++14 compliant. The Münster Skeleton Library (Muesli) [30] and CnC [31] both address distributed environments. While this is a future direction of GrPPI, it is currently focused on shared memory architectures.

Simultaneously, standardized interfaces are being progressively developed. This is the case of ISO C++ Standard Parallel STL published as technical specification [32] and now part of C++17 [33]. Similar implementations to the parallel STL can also be found as third-party libraries, as HPX [34].

6.2. MRI reconstruction techniques

Neuroscience research studies and clinical applications using MRI have become very popular in the last decade, including a wide range of applications such us brain tumor [35], autism [36], or Alzheimer [37].

Many of these studies are based on different intra-voxel models of molecular diffusion, which in turn, require different sampling schemes to collect the data (i.e., Diffusion Tensor Imaging (DTI) [38], High Angular Resolution Diffusion Imaging (HARDI) and Diffusion Spectrum Imaging Imaging (DSI)) and different fitting algorithms. There are several intra-voxel reconstruction methods like Constrained Spherical Deconvolution (CSD) (39) and those evaluated in (40), as well as fiber tracking algorithms to delineate the brain's white matter tracts (41). In order to facilitate the widespread use of this technique, various software solutions have been developed, such as MRtrix [42], FS [43], Dipy [7], and BrainVISA-Diffuse [44].

The implementation of those methods strongly benefits from parallelization and usage of accelerators to speed-up the reconstruction. Techniques have been presented in several works using different techniques such as automatic regularization [45] or GPUS [46, 47]. However, most of those applications were written from scratch to solve a specific algorithm or problem. In a previous work [48], we presented a preliminary prototype of pHARDI, an accelerated reconstruction toolkit to estimate the white matter fiber geometry from diffusion MRI data.

7. Conclusion

Stream processing frameworks are currently a trend in the data processing field. An evidence of this fact is the growing number of solutions in both industry and academia. In this paper, we have presented an extension of the GrPPI programming framework for supporting both streaming data and task parallelism. We have shown the benefits, not only in performance but also in productivity, of using the GrPPI generic API in a real world medical image processing application. Our approach enables the use of multiple backend in a single implementation. The evaluation results demonstrate the acceleration obtained after applying automatic refactoring techniques, clearly outperforming the baseline version. GrPPI allows to provide scale-in using the available cores and memory into the computing nodes transparently to the users. This has allowed us to accelerate the processing time by a factor around 15×.

Although GRPPI covers a large number of node-level backends, we are working on portable execution on GPGPU platforms, such as CUDA. We are also currently working in a distributed version of GrPPI that will support parallel patterns with horizontal scalability. This will allow to deploy pipelines in distributed memory systems and SPMD in a easy way. We aim to extend GrPPI to support distributed memory architectures by using a MPI+X model, which will take advantage of large scale systems.

8. Acknowledgments

This work was supported by the EU project "ASPIDE: Exascale Programming Models for Extreme Data Processing" under grant 801091 and project TIN2016-79637-P "Towards unification of HPC and Big Data Paradigms" from the Spanish Ministry of Economy and Competitiveness of Spain. This research was partially supported by Madrid regional Government (Spain) under the grant "Convergencia Big data-Hpc: de los sensores a las Aplicaciones. (CABAHLA-CM)" Ref: S2018/TCS-4423.

9. References

[1] R. Stoica, M. Frank, N. Neufeld, A. C. Smith, Data Handling and Transfer in the LHCb Experiment 55 (1) (2008) 272–277.

[2] J. Garcia-Blas, D. del Rio Astorga, J. D. Garcia, J. Carretero, Exploiting Stream Parallelism of MRI Reconstruction Using GrPPI over Multiple Back-Ends, in: 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2019), 2019, pp. 631–637. doi:10.1109/CCGRID.2019.00081.

[3] C. Misale, M. Drocco, G. Tremblay, M. Aldinucci, Pico: A novel approach to stream dataanalytics, in: D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, J. Weidendorfer (Eds.), Euro-Par 2017: Parallel Processing Workshops, Springer International Publishing, Cham, 2018, pp. 118–128.

[4] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, M. Danelutto, Bringing parallel patterns out of the corner: The p3 arsec benchmark suite, ACM Trans. Archit. Code Optim. 14 (4) (2017) 33:1–33:26. doi:10.1145/3132710.
URL `http://doi.acm.org/10.1145/3132710`

[5] D. del Rio Astorga, M. F. Dolz, J. Fernández, J. D. García, A Generic Parallel Pattern Interface for Stream and Data Processing, Concurrency and Computation: Practice and Experience 29 (2017) e4175–n/a.

[6] M. M. Michael, M. L. Scott, Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms, in: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '96, ACM, New York, NY, USA, 1996, pp. 267–275.

[7] E. J. Canales-Rodríguez, A. Daducci, S. N. Sotiropoulos, E. Caruyer, S. Aja-Fernández, J. Radua, J. M. Y. Mendizabal, Y. Iturria-Medina, L. Melie-García, Y. Alemán-Gómez, et al., Spherical deconvolution of multichannel diffusion MRI data with non-Gaussian noise models and spatial regularization, PloS one 10 (10) (2015) e0138910.

[8] J. Garcia-Blas, M. F. Dolz, J. D. Garcia, J. Carretero, A. Daducci, Y. Alemn, E. J. Canales-Rodrguez, Porting Matlab applications to high-performance C++ codes: CPU/GPU-accelerated spherical deconvolution of diffusion MRI data, in: ICA3PP: 16th International Conference on Algorithms and Architectures for Parallel Processing, 2016, pp. 630–643.

[9] M. Larobina, L. Murino, Medical Image File Formats, Journal of Digital Imaging 27 (2) (2014) 200–206. doi:10.1007/s10278-013-9657-9.
URL `https://doi.org/10.1007/s10278-013-9657-9`

[10] S. Browne, J. Dongarra, N. Garner, K. London, P. Mucci, A scalable cross-platform infrastructure for application performance tuning using hardware counters, in: Supercomputing, ACM/IEEE 2000 Conference, IEEE, 2000, pp. 42–42.

[11] A. Kukanov, M. J. Voss, The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks, Intel Technology Journal 11 (4) (2007) 309 – 322.

[12] A. Muddukrishna, P. A. Jonsson, M. Brorsson, Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors, Scientific Programming 2015 (2015).

[13] S. Siu, M. De Simone, D. Goswami, A. Singh, Design patterns for parallel programming., in: PDPTA, Vol. 96, 1996, pp. 230–240.

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

[15] T. G. Mattson, B. Sanders, B. Massingill, Patterns for parallel programming, Pearson Education, 2004.

[16] H. González-Vélez, M. Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers, Software: Practice and Experience 40 (12) (2010) 1135–1160.

[17] M. McCool, J. Reinders, A. Robison, Structured parallel programming: patterns for efficient computation, Elsevier, 2012.

[18] D. B. Kirk, W. H. Wen-Mei, Programming massively parallel processors: a hands-on approach, Morgan kaufmann, 2016.

[19] S. T. Allen, M. Jankowski, P. Pathirana, Storm Applied: Strategies for Real-time Event Processing, 1st Edition, Manning Publications Co., Greenwich, CT, USA, 2015.

[20] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65.

[21] S. Papp, The Definitive Guide to Apache Flink: Next Generation Data Processing, 1st Edition, Apress, Berkely, CA, USA, 2016.

[22] W. Thies, M. Karczmarek, S. Amarasinghe, Streamit: A language for streaming applications, in: R. N. Horspool (Ed.), Compiler Construction, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 179–196.

[23] P. Martin, G. Yaikhom, Definition of the DISPEL Language, The Data Bonanza: Improving Knowledge Discovery in Science, Engineering, and Business (2013) 203–236.

[24] A. Pop, A. Cohen, A Stream-Comptuting Extension to OpenMP, in: International Conference on High Performance and Embedded Architectures and Compilers, Heraklion, Greece, 2011.
URL `https://hal.inria.fr/inria-00551507`

[25] J. Reinders, Intel threading building blocks - outfitting C++ for multi-core processor parallelism, O'Reilly, 2007.

[26] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: high-level and efficient streaming on multi-core, Programming multi-core and many-core computing systems, parallel and distributed computing (2014).

[27] J. C. Beard, P. Li, R. D. Chamberlain, RaftLib: A C++ Template Library for High Performance Stream Parallel Processing, in: Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '15, ACM, New York, NY, USA, 2015, pp. 96–105. doi:10.1145/2712386.2712400.

[28] D. Kist, B. Pinto, R. Bazo, A. R. D. Bois, G. G. H. Cavalheiro, Kanga: A Skeleton-Based Generic Interface for Parallel Programming, in: 2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2015, pp. 68–72.

[29] J. Enmyren, C. W. Kessler, SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems, in: Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications, HLPP '10, ACM, New York, NY, USA, 2010, pp. 5–14. doi:10.1145/1863482.1863487.

[30] S. Ernsting, H. Kuchen, Data Parallel Algorithmic Skeletons with Accelerator Support, International Journal of Parallel Programming 45 (2) (2017) 283–299. doi:10.1007/s10766-016-0416-7.

[31] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, S. Taşirlar, Concurrent Collections, Sci. Program. 18 (3-4) (2010) 203–217. doi:10.1155/2010/521797.

[32] ISO/IEC, Programming Languages – Technical Specification for C++ Extensions for Parallelism, ISO/IEC TS 19570:2015 (Jul. 2015).

[33] ISO/IEC 14882:2017 – Programming Languages – C++, International standard, ISO/IEC (Dec. 2017).

[34] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, HPX: A Task Based Programming Model in a Global Address Space, in: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS '14, ACM, New York, NY, USA, 2014, pp. 6:1–6:11.

[35] N. Gordillo, E. Montseny, P. Sobrevilla, State of the art survey on mri brain tumor segmentation, Magnetic resonance imaging 31 (8) (2013) 1426–1438.

[36] R.-A. Müller, P. Shih, B. Keehn, J. R. Deyoe, K. M. Leyden, D. K. Shukla, Underconnected, but how? a survey of functional connectivity mri studies in autism spectrum disorders, Cerebral cortex 21 (10) (2011) 2233–2243.

[37] D. Wang, S. CN Hui, L. Shi, W.-h. Huang, T. Wang, V. CT Mok, W. CW Chu, A. T Ahuja, Application of multimodal mr imaging on studying alzheimer's disease: a survey, Current Alzheimer Research 10 (8) (2013) 877–892.

[38] M. W. A. Caan, F. M. Vos, A. H. C. van Kampen, S. D. Olabarriaga, L. J. van Vliet, Gridifying a Diffusion Tensor Imaging Analysis Pipeline, in: 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing 2010, 2010, pp. 733–738. doi:10.1109/CCGRID.2010.99.

[39] J. D. Tournier, C. H. Yeh, F. Calamante, K. H. Cho, A. Connelly, C. P. Lin, Resolving crossing fibres using constrained spherical deconvolution: Validation using diffusion-weighted imaging phantom data, NeuroImage 42 (2008) 617–625. doi:10.1016/j.neuroimage.2008.05.002.

[40] A. Daducci, E. J. Canales-Rodríguez, M. Descoteaux, E. Garyfallidis, Y. Gur, Y. C. Lin, M. Mani, S. Merlet, M. Paquette, A. Ramirez-Manzanares, M. Reisert, P. R. Rodrigues, F. Sepehrband, E. Caruyer, J. Choupan, R. Deriche, M. Jacob, G. Menegaz, V. Prckovska, M. Rivera, Y. Wiaux, J. P. Thiran, Quantitative comparison of reconstruction methods for intra-voxel fiber recovery from diffusion MRI, IEEE Transactions on Medical Imaging 33 (2014) 384–399. doi:10.1109/TMI.2013.2285500.

[41] Y. Iturria-Medina, E. Canales-Rodríguez, L. Melie-García, P. Valdés-Hernández, E. Martínez-Montes, Y. Alemán-Gómez, J. Sánchez-Bornot, Characterizing brain anatomical connections using diffusion weighted {MRI} and graph theory, NeuroImage 36 (3) (2007) 645–660. doi:https://doi.org/10.1016/j.neuroimage.2007.02.012.

[42] I. Aganj, C. Lenglet, G. Sapiro, E. Yacoub, K. Ugurbil, N. Harel, Reconstruction of the orientation distribution function in single- and multiple-shell q-ball imaging within constant solid angle, Magnetic Resonance in Medicine 64 (2010) 554–566. doi:10.1002/mrm.22365.

[43] A. Barmpoutis, B. C. Vemuri, A unified framework for estimating diffusion tensors of any order with symmetric positive-definite constraints, In Proceedings of ISBI10: IEEE International Symposium on Biomedical Imaging (2010) 1385–1388.

[44] L. Brun, A. Pron, J. Sein, C. Deruelle, O. Coulon, Diffusion MRI: Assessment of the Impact of Acquisition and Preprocessing Methods Using the BrainVISA-Diffuse Toolbox, Frontiers in Neuroscience 13 (2019) 536. doi:10.3389/fnins.2019.00536.

URL https://www.frontiersin.org/article/10.3389/fnins.2019.00536

[45] F.-H. Lin, K. K. Kwong, J. W. Belliveau, L. L. Wald, Parallel imaging reconstruction using automatic regularization, Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 51 (3) (2004) 559–567.

[46] S. S. Stone, J. P. Haldar, S. C. Tsao, B. Sutton, Z.-P. Liang, et al., Accelerating advanced mri reconstructions on gpus, Journal of parallel and distributed computing 68 (10) (2008) 1307–1318.

[47] D. S. Smith, J. C. Gore, T. E. Yankeelov, E. B. Welch, Real-time compressive sensing MRI reconstruction using GPU computing and split Bregman methods, International journal of biomedical imaging 2012 (864827) (2012).

[48] J. Garcia-Blas, Y. Aleman-Gomez, E. J. Canales Rodriguez, J. Carretero, J. D. Garcia, pHARDI: accelerated reconstruction toolkit for estimating the white matter fiber geometry from diffusion MRI data, Tech. rep. (2018).
URL {https://infoscience.epfl.ch/record/234474}