Pipeline Patterns on top of Task-Based Runtimes

Enes Bajrovic, Siegfried Benkner, Jiri Dokulil

Faculty of Computer Science, University of Vienna, Vienna, Austria

enes.bajrovic, siegfried.benkner, jiri.dokulil@par.univie.ac.at

Abstract. Task-based runtime systems have gained a lot of interest in recent years since they support separating the specification of parallel computations from the concrete mapping onto a parallel architecture. This separation of concerns is considered key to coping with the increased complexity, performance variability, and heterogeneity of future parallel systems and to facilitating portability of applications across different architectures. In this paper we present our work on a programming framework that enables the expression of pipeline patterns at a high-level of abstraction by adding pragma directives to sequential C++ codes. Such high-level abstractions are then transformed to a runtime coordination layer, which utilizes different task-based runtime systems including StarPU and OCR to realize efficient parallel execution on single-node multi-core architectures. We describe the major aspects of our approach for mapping pipeline patterns to task-based runtimes and present experimental results for a real-world face-detection application indicating that a performance competitive with low-level programming approaches can be achieved.

Keywords: Parallel Programming, Runtime Systems, Multicore Architectures,

1 Introduction

Many recent research efforts have focused on the use of dynamic task-based runtime systems (e.g., StarPU [1], HPX [2], OCR [3]) to realize efficient applications on increasingly complex parallel architectures. Commonly, these approaches represent an application as a directed acyclic graph (DAG) where nodes represent computational tasks, and edges represent dependences between tasks. From such a representation, a runtime system can dynamically determine tasks that can be executed in parallel and use sophisticated scheduling algorithms to map these tasks to available execution units of the target system. With such an approach, programmers usually only specify what can be potentially executed in parallel, while deferring to the runtime how parallel execution is organized on a specific parallel architecture. Separating the specification of parallelism from its implementation enables the runtime system to better adapt an application to a specific parallel architecture taking into account changing performance characteristics of execution units or dynamically changing workloads.

While task-based runtime systems offer the required flexibility to deal with emerging and future architectures, they are often at a very low level of abstraction and consequently very complex and difficult to use directly for application development. Usually programmers are required to rewrite their applications in a task-parallel way, identifying the individual tasks, describing dependences between them and selecting a suitable scheduling strategy to achieve efficient execution.

In our work we aim at raising the level of abstraction when dealing with task-parallel programming of modern hardware. We have developed a programming framework that supports expression of high-level pipeline patterns within sequential C/C++ codes while automatically generating a low-level task-parallel program for efficient execution on different parallel architectures using either the StarPU or OCR runtime system. Initially we have developed the support for high-level pipeline patterns within the European PEPPHER project [4][15], which relied on a direct transformation of annotated C++ code to the StarPU runtime system. In our recent work we have redesigned our system to support different low-level task-based runtime systems through the introduction of an intermediate runtime coordination layer (RCL) which decouples our programming system from the concrete underlying low-level runtime.

Specifically this paper makes the following contributions: (1) a programming system that maps high-level pipeline patterns to a task-based runtime system hiding the details of parallel execution from the user while providing some means for influencing the degree of parallelism; (2) an intermediate runtime coordination layer (RCL) that utilizes application-specific knowledge to control how tasks are submitted to a lower-level runtime system; (3) a discussion of the major issues in targeting different runtime systems like StarPU and OCR from the RCL and the major differences between those runtimes; and (4) an experimental evaluation of our framework using a real-world face-detection application showing that performance comparable to much more complex, low-level programming approaches can be achieved.

The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 provides an overview of our programming framework and outlines pipeline patterns. Section 4 describes the runtime coordination layer and presents the main issues in targeting StarPU and OCR, respectively. Section 5 presents experimental results followed by a conclusion and discussion of future directions in Section 6.

2 Related Work

Addressing the challenges of programming increasingly complex parallel architectures, a variety of dynamic task-based runtime systems and programming approaches have been developed [1][2][3][5][6][7][8][9][10]. We discuss some of these systems below.

StarPU [1] is an asynchronous task-based runtime system and programming library for heterogeneous systems. StarPU introduces *codelets* that abstract *tasks* and allow programmers to provide different implementation variants of a computational kernel (e.g. for CPUs or GPUs). Moreover, StarPU provides data management facilities that automate the data transfer between processing units. StarPU automatically determines data dependences by analyzing task arguments and supports various scheduling strategies for deciding when and where to execute tasks taking into account historic performance data, resource readiness, and data availability.

The Open Community Runtime (OCR) [3] is an open specification for a low-level, event-driven, asynchronous, task-based runtime system designed for extreme scale

parallel computing. A parallel application in OCR is expressed as a DAG where nodes represent tasks (or events) and edges dependences. Event-driven tasks (EDTs) are decoupled from their actual loci of execution, which are determined dynamically by the runtime system. Events are used to control the execution and synchronization of tasks. The data in OCR is organized in *data* blocks, which describe data independently from their actual physical memory location. This allows OCR to dynamically relocate tasks and data in order to improve locality, optimize performance and support fault tolerance. As opposed to StarPU, which automatically infers dependences between tasks, in OCR all dependences between tasks need to be explicitly specified.

OmpSs [5] extends OpenMP with support for asynchronous task-parallel execution. The OmpSs memory model assumes a shared address space to enable automatic data movement across the system. It provides support for multiple implementation variants of tasks when targeting heterogeneous systems. Many features introduced in OmpSs have been adopted by the OpenMP standard [6].

Legion [7] is a data-centric, task-based programming system that supports dynamic hierarchical data partitioning based on the concept of logical regions. Tasks are bound to regions and may access regions with different privileges and subject to different data coherence modes. Legion provides a mapping interface that enables programmers to control the mapping of tasks and data regions in order to optimize an application for a specific architecture.

The HPX runtime system [2] is a C++-based implementation of a task-based programming model within an active global address space, which supports migration of objects between the nodes of clusters.

Most of the mentioned runtime systems are being used directly by application programmers, leading to a low-level programming methodology similar or even more complex than the dominating MPI model. While for some runtime systems also high-level approaches exist (e.g., the Regent language targeting Legion, or OpenMP tasking extensions), none of the discussed runtimes currently supports high-level patterns comparable to our pipeline patterns.

Thread Building Blocks (TBB) [8] is a task-based C++ template library. It provides support for parallel patterns (including pipelines) and concurrent data structures that hide some of the complexity of parallel programming. TBB's task-based approach utilizes work-stealing task schedulers to map tasks and perform load balancing among processing resources. While TBB is a template library and provides an integrated scheduler, our approach relies on annotating sequential C++ codes and allows targeting different low-level runtime systems transparently.

3 High-Level Parallel Pipeline Patterns

We provide a set of preprocessing directives that enable expression of pipeline patterns at a high level of abstraction in sequential C/C++ codes and a transformation system and runtime layer for executing pipelines on top of task-parallel runtime systems.

3.1 Overview of the Programming Framework

Figure 1 provides an overview of our programming framework. We support applications written in C/C++ with extensions for high-level pipeline patterns. A source-to-source compiler translates sequential C++ code with pipeline patterns to an explicitly parallel representation that utilizes the *Runtime Coordination Layer* (RCL) for managing execution on top of a task-based runtime system (OCR or StarPU).

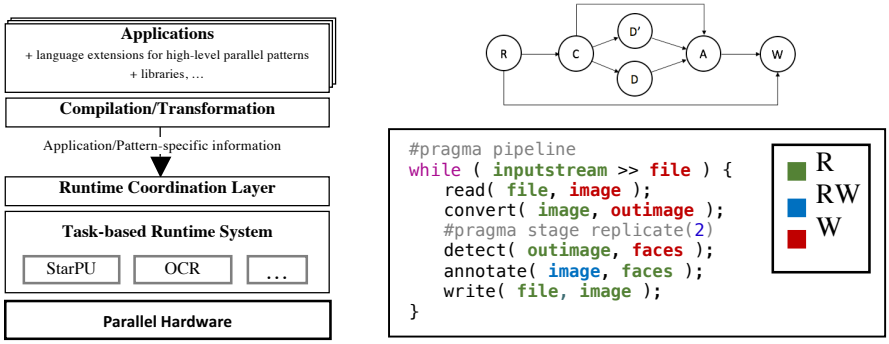

Figure 1. Programming framework (left), face detection pipeline (right)

Currently our framework targets single-node multicore systems. Support for heterogeneous architectures (e.g., CPU/GPU systems) is currently restricted to the StarPU runtime. The framework also provides hooks for interfacing it with an external autotuner [16] to support tuning of application-specific parameters like stage replication factors and runtime-specific parameters like the number of worker threads or the scheduling strategy [11].

3.2 High-Level Pipeline Patterns

A pipeline consists of several interconnected stages, where data items flowing through the pipeline are processed at every stage. Usually, stages are connected via buffers from which stages access the data. While buffered pipelines require additional memory, they allow to decouple stages, to mitigate relative performance differences, and to better control parallel execution and pipeline throughput. In our approach buffers between stages are hidden from the user and generated automatically by the compiler.

A pipeline is constructed with the *pipeline* pragma directive from *while-loops*. The *stage* directive is used to indicate that one or more function calls correspond to a pipeline stage and may be omitted if a stage contains only a single function call. For each stage function multiple implementation variants may be provided (e.g., a CPU or a GPU variant). An external XML-based descriptor has to be provided for each stage function comprising its interface specification, the intent of function arguments (*read, write, readwrite*), and information about available implementation variants.

Within the stage directive a *replication factor R* may be specified, indicating that *R* instances of a stage should be generated, operating in parallel on different data items.

The specification of replication factors enables users to control the degree of parallelism. The stage directive may also be used to merge multiple function calls into a single stage in order to increase the computational granularity of stages.

Figure 1 shows a face detection pipeline. Images are read from disk (*read* stage), converted to a grayscale (*convert* stage) and analyzed for human faces (*detect* stage). The detect stage, which is the computationally most intensive one, is replicated 2 times. In the *annotate* stage a rectangle is drawn around each detected face. Finally, the resulting image is written to the disk in the *write* stage. External XML descriptors for stages are not shown.

4 Runtime Coordination Layer

The primary objective of the RCL is to provide an abstraction of the underlying low-level runtime system in order to simplify the compilation process and to facilitate retargeting the programming framework to different task-based runtime systems.

The RCL utilizes an object-oriented representation of patterns and aims at exploiting pattern-specific knowledge available in the high-level code in order to optimize task-parallel execution. The main objectives of RCL are: (1) to manage the details of submitting tasks to the low-level runtimes, (2) to manage data transfers between pipeline stages, and (3) to control the degree of parallelism by restricting the number of tasks concurrently submitted to the low-level runtime.

4.1 Representation of Pipeline Patterns in RCL

During the transformation process, the compiler analyzes each pipeline and generates code that constructs a corresponding object-oriented representation of the pipeline. The compiler automatically derives the stage interconnections by analyzing the data flow between stages and the intent (*read*, *write*, or *readwrite*) of the arguments of each stage function. Stage interconnections provide the basis for automatically generating buffers between stages and for setting up explicit task dependences as required by OCR.

For each type of data item that is being processed in a pipeline, a *data descriptor* is generated, comprising the details of how these items are represented (e.g., size and location in memory) and managed. From these data descriptors corresponding data handling code for the different low-level runtimes is generated.

Similarly, since for each invocation of a stage function a task needs to be submitted to the low-level runtime system, the RCL generates corresponding *task descriptors*. A *task descriptor* comprises all information needed to create a concrete task for a low-level runtime system. Besides information about data consumed and produced by a pipeline stage, the task descriptor captures the stage dependences, and contains pointers to one or more implementation variants of the stage function. Task descriptors are then used by RCL classes to dynamically generate tasks for the concrete underlying runtime system.

4.2 Execution of Pipelines with RCL

The compiler generates for each pipeline RCL code that executes the pipeline in an asynchronous fashion. For each pipeline stage the following *stage mechanism* is executed by a so-called *runner*: (1) if all required input data of the stage are available, the data are acquired from the corresponding buffers; otherwise the stage mechanism waits (sleeps) until data becomes available; (2) using the information in the associated task descriptor, a task is generated and submitted asynchronously to the underlying runtime system (StarPU or OCR); (3) when the task finishes execution, the runner is notified, resumes execution, and the generated output is pushed to the corresponding output buffer(s). If the output buffer of a stage is full, the runner will wait (sleep) until there is a free slot in the target output buffer.

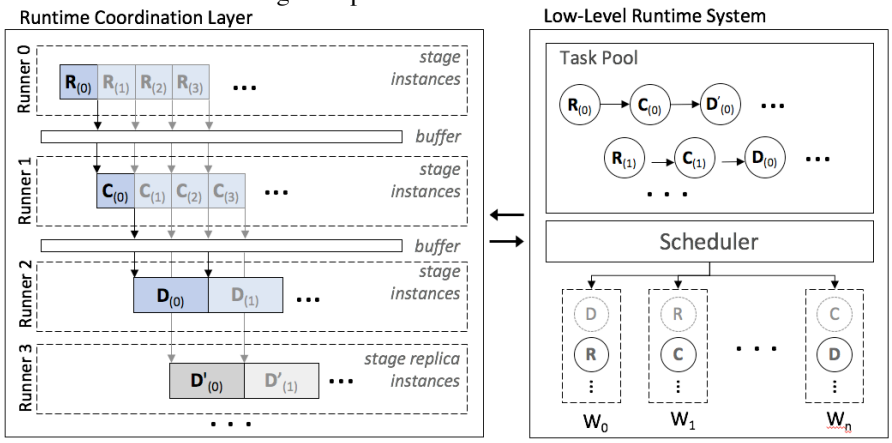

Figure 2. RCL and Low-Level Runtime System

Figure 2 sketches how the first three stages of the face detection pipeline are managed by the RCL. For each stage, the stage mechanism is executed by a *runner*, submitting for each instance of a stage a task to the low-level runtime system. In case of a replicated stage (e.g., the *detect* stage), a runner is generated for each stage replica. The low-level runtime system schedules the execution of tasks to worker threads.

The number of tasks submitted to the low-level runtime at the same time (and potentially running in parallel) is controlled by stage replication factors and buffer sizes. If a stage is replicated *R* times, then *R* stage objects are generated, executing the above-mentioned stage mechanism in parallel. Consequently, for replicated stages task are submitted in parallel to the underlying runtime system. As all replicas of a stage share input and output buffers, buffers have been implemented such that they can be accessed efficiently in parallel while maintaining ordering of data items to preserve the original semantics of the application. RCL buffers have been implemented on top of concurrent queues supporting *fifo*, *filo*, and *priority* ordering.

Another way to control the degree of parallelism is via the sizes of buffers. If the output buffer of a stage is full, the stage will wait (sleep) until there is a free slot in the target output buffer. Consequently, the size of buffers not only determines the memory footprint but also the number of tasks that are being submitted to the runtime.

For each type of data item processed corresponding data representations for the low-level runtime systems are derived based on the associated data descriptor. Both OCR and StarPU require that data consumed or produced by a task is contiguous in memory and that the access mode (intent) is explicitly specified (*read, write, readwrite*). We avoid explicit copying of data and handle all data transfers between stages only by means of manipulating pointers (i.e., data items in buffers are represented by pointers to corresponding data descriptors).

The RCL provides special classes for dynamically generating tasks from task descriptors associated with the stages. Depending on whether the targeted low-level runtime system relies on an *explicit* or an *implicit task dependence mechanism*, the dynamic conversion of task descriptors to concrete tasks is different. While OCR requires that all task dependences be explicitly specified using events, StarPU implicitly determines task dependences by analyzing the dataflow between tasks.

Also, the stage mechanism in RCL needs to be aware of when a low-level runtime task has finished, so it can push results to the output buffer. For this purpose, StarPU provides either the *starpu_task_wait()* routine to wait for a specific task to finish, or *callback* functions to be executed after the task has finished. In the current version of the RCL we rely on the first mechanism. In OCR no such functionality is provided, and additional OCR tasks are used to inform RCL about task execution state.

Finally, the stage mechanism (i.e., the *runners)* is also implemented differently. While for StarPU runners are implemented as separate C++ threads, for OCR the runners are realized as OCR helper tasks since OCR routines are not safely callable from outside of OCR tasks. Consequently, when OCR is targeted, the generated RCL program will start with the OCR *mainEdt()* and not with the standard C++ *main()*.

4.3 Task Generation for StarPU

For each instance of a stage, RCL generates a task for StarPU. A StarPU task is described by a *codelet* which usually comprises a pointer to the stage function, a set of handles for all data items consumed and produced, and information about performance models used for scheduling decisions.

In order to enable StarPU to analyze data dependences between tasks, all the input and output data of a kernel has to be registered with StarPU using data handles generated from RCL data descriptors. If the access mode of a stage argument is *read* or *readwrite,* RCL passes a pointer to the argument to StarPU, otherwise it acquires a pointer from StarPU. Registering data also allows StarPU to automatically manage data transfers across different execution units, e.g., from CPU to GPU memory.

For each stage the RCL uses a separate C++ thread (runner) to execute the stage mechanism. Once all input data are available, data handles are registered with StarPU and the generated task is submitted asynchronously to StarPU by calling *starpu_task_submit()* followed by *starpu_task_wait()* to wait until the task has finished. Upon task completion, input data of the task are deregistered and output data pushed to the output buffer(s) allowing subsequent stages to continue executing.
The StarPU runtime system provides different scheduling strategies and for the experiments reported in this paper we have used the HEFT scheduling strategy [1], which relies on performance models built from historic execution data of tasks.

4.4 Task Generation for OCR

The OCR task-parallel model relies on *event driven tasks* (*EDTs*), which are connected via *events* with other tasks, to represent a parallel application as a task graph (DAG) that is dynamically scheduled for parallel execution by the runtime system. EDTs operate on contiguous, relocatable blocks of data, called *data blocks*, which are managed by the runtime system. An EDT may have multiple pre-slots for input data and one post-slot for the output. Instances of EDTs are created from EDT templates, which comprise information about the function an EDT executes, its parameters, and its dependences.

As opposed to StarPU where dependences between tasks are automatically determined, OCR requires all dependences to be explicitly specified. Once all data blocks an EDT depends on are available, a task becomes runnable and will eventually be executed on some execution unit selected by the runtime and run to completion regardless of the behavior of other tasks. The RCL provides the *OCRRuntime* class that comprises a set of EDT templates used to orchestrate the execution of pipelines, as well as for calling *ocrShutodown()* when an application terminates.

Due to restrictions of OCR (OCR owns the *main()*) the generated RCL program does not start with the standard main function but with the OCR *mainEdt()* function. Also, because calling OCR routines from outside of OCR tasks is not possible, we utilize for each stage instance an OCR *helper task* to run the stage mechanism and to submit tasks for executing the stage function. The helper task creates OCR data blocks for all data items (by calling *ocrDbCreate()*), an *executor* EDT (*ocrEdtCreate()*) to execute the stage function, and it sets up dependences (*ocrAddDependence()*) between the data blocks and the *executor EDT*. It then calls corresponding OCR routines to satisfy all the dependences of the *executor EDT,* which then becomes runnable and is eventually run by the OCR runtime executing the stage function. Upon completion, another *helper* task will be generated managing the execution of the next stage instance.

5 Evaluation

For evaluation we use the face detection application outlined previously in Figure 2. The application utilizes routines from the OpenCV library [12], which have been slightly reengineered to conform to the tasking model. For the measurements, the application processes a set of 500 images of 360p resolution, each containing an arbitrary number of faces.

We present speedup measurements on a machine with 2 octa-core Intel Xeon E5-2650 CPUs (2.0 GHz, 128GB RAM). We compare our high-level pipeline code using either StarPU or OCR runtime to a sequential C++ version, hand coded StarPU and OCR versions, and a TBB version that uses the pipeline algorithm of TBB. Additionally, we evaluated the impact of stage replication factors on the overall pipeline performance. We have used GCC 5.3.0 compiler, OpenCV version 3.3.1, StarPU 1.2.4, TBB 4.2, and our own OCR implementation [13][14].

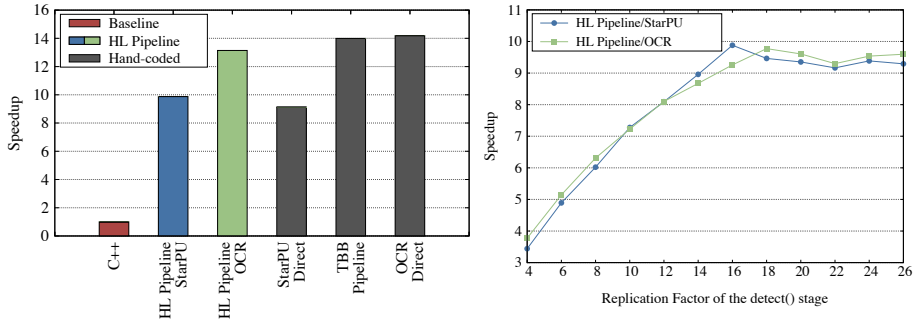

Figure 3. Speedup of face detection pipeline over sequential C++ on 16 cores (left) and impact of replication factors (right)

Figure 3 (left) shows the speedup on 16 cores for different versions over the sequential C++ version (red bar), which is the same as the high-level pipeline code without pragma directives. Our high-level pipeline code on top of StarPU (blue bar) achieved a speedup of 9.9 and on top of OCR (green bar) a speedup of 13.1. Interestingly, the highest performance was achieved for OCR when replicating the *convert* and *annotate* stage 5 times each and *detect* stage 16 times, while for StarPU when replicating only the *detect* stage 16 times.

The hand-coded StarPU and OCR versions (StarPU/OCR Direct), which have been parallelized manually, achieved speedups of 9.1 and 14.2, respectively. The TBB version, which uses the TBB pipeline algorithm achieved a speedup of 14. These comparisons show that the performance achievable with our high-level patterns is very close (in case of StarPU even better) to hand-coded versions that utilize the low-level runtimes directly or to TBB, which is a highly optimized template.

Our high-level pipeline code has 50 lines of code (4 lines/pragmas more than the sequential C++ version), the TBB version has 85 lines, the hand coded StarPU version 294 lines, and the hand coded OCR version 209 lines. These line counts do not include the used OpenCV functions, which were the same for all these versions.

Figure 3 (right) shows how replicating only the *detect* stage affects the overall performance. As can be seen, in this configuration our high-level pipeline code performs best with a replication factor of 16 using StarPU and of 18 using OCR.

6 Conclusions and Future Work

In this paper we have presented our work on a programming framework that supports high-level pipeline patterns within sequential C++ codes, which are transformed by our source-to-source compiler to an intermediate runtime coordination layer (RCL) that realizes task-parallel execution on top of the StarPU or OCR runtime system. Our experiments with a face detection application show that performance competitive with much more complex low-level programming approaches can be achieved.

Our future work will focus on improving the tuning support for our framework (e.g. tuning of stage replication factors) and on support for heterogeneous systems.

Acknowledgement: The work was supported in part by the Austrian Science Fund (FWF) project P 29783 Dynamic Runtime System for Future Parallel Architectures.

References

[1] C. Augonnet, S. Thibault, R. Namyst and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice & Experience - Euro-Par 2009 (Feb. 2011) 187-198

[2] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey. HPX - A Task Based Programming Model in a Global Address Space. PGAS 2014: The 8th International Conference on Partitioned Global Address Space Programming Models (2014)

[3] R. Cledat, Tim Mattson. OCR, The Open Community Runtime Interface. OCR Specification 1.2.0 (2016)

[4] S. Benkner, S. Pllana, J. L. Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, V. Osipov. PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. IEEE Micro 31, 5, 28-41 (2011)

[5] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, J. Labarta. Productive Programming of GPU Clusters with OmpSs. Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International (2012)

[6] OpenMP Architecture Review Board, OpenMP Application Programming Interface v4.5, (2015)

[7] M. Bauer, S. Treichler, E. Slaughter and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, Utah (2012)

[8] C. Pheatt. Intel® Threading Building Blocks. Journal of Computing Sciences in Colleges, Volume 23 Issue 4 (April 2008) 298-298

[9] M. Robson, R. Buch, L. Kale. Runtime Coordinated Heterogeneous Tasks in Charm++. ESPM2 Workshop, in conjunction with SC16, Salt Lake City (2016)

[10] D. Majeti, V. Sarkar. Heterogeneous Habanero-C (H2C): A Portable Programming Model for Heterogeneous Processors. 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (2015)

[11] E Bajrovic, S. Benkner. Automatic Performance Tuning of Pipeline Patterns for Heterogeneous Parallel Architectures. The 2014 International Conference on Parallel and Distributed Processing, Techniques and Applications (2014)

[12] B. Gary, K. Adrian. Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library. O'Reilly Media (2016)

[13] J. Dokulil, M. Sandrieser, S. Benkner. OCR-Vx - An Alternative Implementation of the Open Community Runtime. International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15. Austin, Texas, (November, 2015)

[14] J. Dokulil, M. Sandrieser, S. Benkner. Implementing the Open Community Runtime for Shared-Memory and Distributed-Memory Systems. In 24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Heraklion, Greece, February 2016, IEEE Computer Society

[15] S. Benkner, E. Bajrovic, E. Marth, M. Sandrieser, R. Namyst, S. Thibault. High-Level Support for Pipeline Parallelism on Many-Core Architectures. In Proc. European Conference on Parallel Computing, Euro-Par 2012, Rhodos, Greece, Aug. 27-31, 2012, LNCS 7484, pages 614-625, Springer Verlag

[16] M. Gerndt, E. Cesar, S. Benkner. Automatic Tuning of HPC Applications - The Periscope Tuning Framework (PTF), Shakar Verlag, 2015