2011 IEEE International Parallel & Distributed Processing Symposium

Towards an Effective Unified Programming Model for Many-Cores

Ana Lucia Varbanescu*†, Pieter Hijma*, Rob van Nieuwpoort*‡ and Henri Bal*
*Computing Systems Group, Vrije Universiteit Amsterdam, The Netherlands
†Parallel and Distributed Systems Group, Delft University of Technology, The Netherlands
‡ASTRON, Dwingeloo, The Netherlands
e-mail: {analucia,rob,hphijma,bal}@cs.vu.nl

***Abstract*—Building an effective programming model for many-core processors is challenging. On the one hand, the increasing variety of platforms and their specific programming models force users to take a hardware-centric approach not only for implementing parallel applications, but also for designing them. This approach diminishes portability and, eventually, limits performance. On the other hand, to effectively cope with the increased number of large-scale workloads that require parallelization, a portable, application-centric programming model is desirable. Such a model enables programmers to focus first on extracting and exploiting parallelism from their applications, as opposed to generating parallelism for specific hardware, and only second on platform-specific implementation and optimizations.**

In this paper, we first present a survey of programming models designed for programming three families of many-cores: general purpose many-cores (GPMCs), graphics processing units (GPUs), and the Cell/B.E.. We analyze the usability of these models, their ability to improve platform programmability, and the specific features that contribute to this improvement.

Next, we also discuss two types of generic models: parallelism-centric and application-centric. We also analyze their features and impact on platform programmability. Based on this analysis, we recommend two application-centric models (OmpSs and OpenCL) as promising candidates for a unified programming model for many-cores and we discuss potential enhancements for them.

I. Introduction

Modern multi-core processors and many-core accelerators (simply, many-cores), such as graphics processing units (GPUs) and the Cell processor, offer unprecedented performance levels by exploiting hardware parallelism on a large scale. Inevitably, they are seen as a solution to the performance problems that arise in many applications. This assumption only holds if applications respond with comparable parallelism, a non-trivial task for most types of workloads.

Programming many-cores is difficult, as it is a problem with multiple constraints: we want applications to deliver great performance, to be easy to program, and to be portable between architectures. The perceived levels of these parameters - performance, productivity, and portability - combine into a measure of platform programmability, which is a good indicator for the success of an architecture.

As matters are only getting worse with the increased variety of many-cores (see Section II for a comprehensive overview), both vendors and academia provide platform-specific programming models and tools, aiming to improve platform programmability (i.e., to make these architectures easier to program). This paper presents a survey of programming models used for general purpose and high-performance computing on many-core architectures. To evaluate these models, we focus on the model features used to improve platform programmability (Section III).

Our discussion has two parts. First, we present a set of representative hardware-centric models (Section IV), which aim to make many of the complicated low-level architectural features of many-cores transparent to the users. These models tackle mostly application development (aiming to improve usability by offering an easy-to-use development environment), and do not provide enough support for parallel application design.

Further, we also analyze several representative generic programming models - i.e., models that can be successfully used for more many-core families (Section V). We discuss both parallelism-centric and application-centric models, and we show that these models offer enough support for application development, and provide additional support in expressing and exploiting parallelism at design time.

Each one of the twenty models we survey, with its qualities and drawbacks, has a positive impact on platform programmability. Still, most of these models are scarcely ever used. We believe this poor adoption of high-level programming models is due, to a large extent, to the multitude of models available. It is difficult for potential users to understand the limitations of each model, the differences between models, or the impact a model has on a specific workload. Therefore, programmers use the native models which, although cumbersome and low-level, guarantee full flexibility. To counter this approach, we propose to choose the most promising models and identify how can they be transformed into a generic application-centric model for many-core applications. Based on features, users-base, current and projected development status, we choose two good candidates: OpenCL[1] and OmpSs[2], [3]. We compare these models in more detail (Section VI), and we make a couple of recommendations on what they can improve in terms if programmability impact and generality. We conclude (Section VII) that using these "candidate" models pays off in (investments in) their faster paced improvement, and limits the use of the lower-level native and/or hardware-centric models.

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.210

TABLE I
KEY COMPUTING AND MEMORY PROPERTIES OF MANY-CORE PROCESSORS.

Platform	Cores / threads	Vectors	ALUs	Scheduling	Par levels	Space(s)	Access	Cache
Intel Nehalem EX	8 / 16	4-wide	64	OS	2	shared	R/W	transparent L1-3
AMD Magny Cours	12 / 12	4-wide	48	OS	2	shared	R/W	transparent L1-3
IBM Power 7	8 / 64	4-wide	256	OS	2	shared	R/W	transparent L1-3
Sun Niagara II	8 / 64	no	64	OS	1	shared	R/W	transparent L1-2
GF1000	(16x32)xSIMT=10K-100K	no	512	HW	3	shared;device;(host)	R/W;(transparent L1-2)	app-controlled shared store
HD5870	(20x16)xSIMT= 10K-100K	4-wide	1600	HW	4	shared;device;(host)	R/W;(transparent L1-2)	app-controlled shared store
Cell/B.E.	9 / 10	4-wide	36	App	5	SPE(local store);PPE	DMA;R/W	app-controlled local store

II. A COMPARATIVE ANALYSIS OF MANY-CORES

In this section we discuss three classes of many-core devices used for high performance computing: multi-core processors[1], graphical processing units (GPUs), and the Cell Broadband Engine (Cell/B.E.).

General purpose multi-cores (GPMCs) are replacing, since 2006, traditional single-core CPUs in both personal computers and servers. GPMCs are homogeneous platforms with complex cores, based on traditional processor architectures. They are typically shared-memory architectures, with multiple levels of caches. We emphasize the diversity of the spectrum of GPMCs by discussing multi-cores from different vendors: The Intel Nehalem EX (Xeon 7500 series), the AMD Magny Cours, the IBM POWER7, and the Sun Niagara 2 (UltraSPARC T2+). Many-core programming models should be retargetable to all these architectures.

Since the first HPC applications ran on a GPU tens of times faster than on the CPUs of the time (2007), GPUs are constantly increasing their computation abilities. Nowadays, state-of-the-art GPU architectures target HPC markets directly, by adding computation-only features to their graphics pipelines. GPUsare shared memory machines, with a complex memory hierarchy, combining different types of caches and scratch-pads at each level. GPUs are used as accelerators, which requires very low flexibility in the hardware; in turn, this allows for architectures that provide high memory throughput and computation capabilities. The PCI express bus is used to connect a GPU to a host system. We discuss GPUs from NVIDIA (the GF100 or Fermi architecture), and AMD/ATI (the Radeon HD 5870 or Cypress).

Finally, we discuss the Cell/B.E., a 9-core heterogeneous processor (1 PPE and 8 SPEs) with a very basic programming model, in which a lot of architecture-related optimizations must be done by the programmer. The eight SPEs are dedicated to high-speed processing, have their own local scratch-pad memories, and access the main memory by explicit DMAs. The main system memory is directly accessible to the PPE (which also has its private L1 and an L2 shared with the SPEs). Cell can be used both as a stand-alond processor and as an accelerator[2].

[1]We generically call all these platforms "many-cores" due to their relatively large numbers of hardware threads. However, we preserve the name "multi-cores" as traditional for general-purpose many-cores.

[2]Cell/B.E. is the main processor in PlayStation 3; RoadRunner (http://www.lanl.gov/roadrunner/) uses Cell as accelerators next to AMD Opteron's as main processors

Table I presents some key computing and memory properties of many-core platforms. Note the increase in the number of parallelism levels: programming models can handle these explicitly or implicitly, trading performance for programmability (see Section IV). Further, note that many-cores have increasingly complex memory and caching hierarchies. This happens because they have to compensate for the inherent decrease in memory bandwidth *per core* with the increase in the number of cores and ALUs. One of the key differences between multi-core CPUs on the one hand, and the GPUs and the Cell/B.E. on the other, is that the memory hierarchy is more exposed, and often explicitly handled in the latter. This has a clear impact on the programming effort that is needed to achieve good performance.

III. EVALUATING PROGRAMMING MODELS

We start from the assumption that programming models are built to improve platform programmability. Therefore, this section defines platform programmability and its components, and presents a list of features used to evaluate many-core programming models in terms of usability and programmability impact.

A. Programmability

Due to the multiple levels of parallelism many-core platform offer, their peak performance is only achievable if applications are able to extract and express enough layers of parallelism, at par with those offered by the hardware platform.

Platform programmability is a measure of how easy it is for (generic) applications to express enough parallelism to match the hardware offer.

Typically, the native programming model of a platform exposes its "bare" programmability, as it provides users with the means to express parallelism in a platform-specific form, and it has minor limitations on achievable performance. Higher level programming models aim to improve programmability, by (1) offering users easier abstractions for designing and building parallelism, and (2) building better back-end components (i.e., compilers and runtime systems) to minimize the impact on performance.

We judge the impact a programming model can have on platform programmability as a combination of its *productivity*, *portability*, and *performance*. Productivity is a measure of the development effort (typically, the time spent by the user when designing and developing the application). A model's portability indicates the potential re-usability of the solutions built using it. A model's performance indicates the achievable

performance of a solution; from the model's perspective, the performance potential is usually measured as efficiency (i.e., how much of the platform's peak performance is achievable when using the chosen model).

Note that productivity, portability and performance are strongly interconnected. For example, obtaining maximum performance might increase development time, thus decreasing productivity; similarly, a highly portable solution can use no hardware specific optimizations, thus limiting achievable performance. Therefore, a programming model has a positive impact on platform programmability if it can increase productivity and portability without (negatively) affecting the achievable performance.

B. Features

The features of a programming model are essential for its adoption: while the availability of features is not sufficient to guarantee success, the lack thereof is definitely a show stopper. There are three categories of features we use to evaluate the programming models: usability, design support, and implementation support. We further explain the features we consider representative for each category, and show how do they influence platform programmability.

Usability: We include here a set of practical features that programming models offer; these features are linked to the ease of use of a programming model, ultimately aiming to increase programmers' productivity.

Class: We separate models in three classes: parallelism-centric, hardware-centric, or application-centric. Parallelism-centric usually provide model-specific constructs to parallelize an application. To parallelize applications, they have to be altered in such a way that they use the parallel constructs. Hardware-centric models focus on providing a simplified interface for exploiting platform-specific parallelism. Finally, application-centric models help programmers to design an inherently parallel program. These models typically provide abstractions for parallelism.

Initial problem specification: Programming models require different ways of exposing the problem to be solved. We distinguish here models that start from the sequential code (typically enhanced with parallelism by the programmer), models that start from an algorithm and apply model-specific parallelization (this usually requires finding a new, parallel algorithm), and models that start from application specification. Note that for hierarchical models (e.g., models that use a host-accelerator(s) structure), it is typical that the problem specification differs between layers.

Actions: The actions to be performed to transform the problem specification into a parallel solution, as well as the way they are done (by the user or (semi-)automated) are essential in increasing productivity. Typical actions are specific parallelization, where the user parallelizes the given algorithm to fit the target model, loop-level parallelization, usually done by the compiler, kernel isolation and fine-grain parallelization, where the users need to isolate the highly parallel regions in the code and exploit loop-like parallelism within the model space, and data clustering, where users specify collections of data to be processed in parallel, and a compiler or runtime system uses these elements as concurrency units.

Impact: Problem specification is important for productivity because using the right initial description helps with correct solution design and minimizes the time spent in design, thus making the process more efficient. For example, if sequential code is not available for a given application, choosing a model that requires the sequential code algorithm to design the parallel version is counter-productive. Furthermore, models that require a detailed application specification and complex actions to be performed by the user have good performance potential, but their impact on productivity is negative; by contrast, models that rely on automated transformations of applications specification typically show both improved productivity and performance limitations. The class of a model are only indirectly linked to usability: users are responsible for choosing a model that suites their knowledge level (models based on known programming languages, or use familiar programming constructs, as well as models that use libraries of pre-optimized components are proven to be more productive than models that use new abstractions and languages).

Design support: There are four features we evaluate to determine if programming models offer support for parallel application design:

Algorithm view: The *algorithm view* [4] of a programming model can be fragmented or global. The parallelism constructs of a fragmented-view model are usually explicit, and interleaved with the processing constructs. In this case, the processing appears as fragmented, like in the classical example of MPI [5]. In contrast, a global-view model typically uses implicit communication and synchronization constructs, resulting in little interference with the processing, thus preserving the global view of the algorithm. A typical example is High Performance Fortran [6].

Parallelism: A model can support multiple *types of parallelism*. At a lower level, models can offer support for SIMD (single instruction, multiple data - typically known as vectorization) or SIMT (single instruction, multiple thread - also known as lock-step execution), targeting fine-grain parallelism. At a higher level, models can offer SPMD or MPMD [7] (single process/multiple process, multiple data) - targeting coarse-grain parallelism. Finally, at the highest level, models can offer one or several patterns for both SPMD and MPMD, such as divide-and-conquer, map-reduce, pipelining, or streaming.

Concurrency units and granularity: A programming model can define its own concurrency units and to provide mechanisms to control their granularity. Concurrency units can range from data items (in flat, data-parallelism oriented models) to functions and/or processes; models can also have hierarchies of different concurrency units. Furthermore, models that may alter the granularity of their concurrency units (more or less dynamic) are told to have "granularity control". Available models range from those which do not have abstractions for granularity control, other than changing the program

(the majority), to programming models that offer automatic and/or dynamic granularity control.

Data layout: A model can provide ways to explicitly specify a *data layout* and/or a *data distribution* among the concurrency units. This improves the control users have to limit unneeded communication and influence future mapping and scheduling decisions.

Impact: The overall design support of a model has an important impact on productivity, portability, and performance. The algorithm view influences portability and productivity: global-view algorithms are easier to reason about and simpler to port on different platforms than fragmented-view models; in turn, fragmented models might simplify debugging and implementation. The supported types of parallelism is directly linked to productivity: a good match of the application parallelism with the programming model parallelism leads to a programmability boost, while a mismatch typically requires a lot of empirical changes on the algorithm, decreasing productivity. The *control over the granularity of the concurrency units* can contribute to performance, but also to portability. A model that allows explicit granularity definition without changing the program, may contribute to performance. In addition, applications can be ported to an architecture which needs more fine-grained or coarse-grained parallelism. Finally, data distribution contributes to productivity and portability, as a model with explicit data distribution is easier to reason about, debug, and tune for different platforms.

Implementation support: We discuss here four features that offer implementation-level support and impact overall platform programmability.

Mapping and Scheduling: By mapping and scheduling, we refer to the way the concurrency units are "placed" on the platform resources and executed to improve concurrency. Models typically choose one of the following solutions: (1) require users to make an explicit mapping, (2) determine the mappings automatically or even dynamically (using their own runtime system), or (3) rely on either the Operating System (OS) or the hardware schedulers for a "default" mapping.

Data transfers: Due to their complex memory hierarchies, many-cores need *data transfers* between memory levels. Therefore, applications need to program transfers between concurrency units (and, eventually, concurrency layers). Programming models can choose to require transfers to be made explicitly or deal with them implicitly (i.e., transparent). As memory hierarchies vary a lot between platforms, it is likely that a hybrid approach - where some transfers are explicit, while the rest are taken care of by either the hardware (shared memories) or a runtime system - will prevail.

Communication and synchronization: An important part of application development deals with the communication and synchronization between concurrency units. The traditional alternatives are implicit (i.e., transparent to the user) and explicit. Implicit communication and synchronization is essentially solved by the model, thus avoiding typical parallelism problems (like deadlocks or race conditions). Models that choose for an explicit approach require users to program the communication and/or synchronization explicitly. Hybrid approaches are possible, and fairly quite common.

Optimizations: Programming models can simplify certain types of optimizations. If such optimizations can be performed automatically (without users tweaking the code), their positive influence on performance translates into a positive impact on programmability. However, optimizations are typically low-level and platform-specific (see memory coalescing for GPUs and SIMD extensions for the Cell/BE or the GPMCs), requiring user's intervention and diminishing solution portability and productivity. Among the models that require users' intervention for optimization, we those models that encourage the users to freely apply low-level optimizations (by simply altering the code) and those which limit or even obstruct this action - mainly because such interventions on code lower the ability of the model's analyzers to parse and extract other parameters and/or parallelization opportunities.

Impact: Programming models that offer enhanced implementation support for parallel applications on many-cores have to cover these features. Furthermore, the way these features are reflected by models impacts programmability. For example, using explicit mapping and scheduling increases solution complexity, lowering productivity; the implicit alternative typically affects performance. The way data transfers between concurrency units (and, eventually, concurrency layers) are done impacts both performance and productivity. Requiring data transfers to be made explicitly affects portability and increases the complexity of the solution, while making them implicit without performance penalties requires the programming model to know the concurrency units mapping.

IV. Hardware-centric Programming Models for Many-cores

Hardware-centric programming models aim to replace the native platform programming (typically supported by a low-level model) with higher-level, user-friendly solutions. In this section, we present a set of selected models designed to address the challenges posed by the three families of many-cores we target.

A. GPMC Programming Models

The native parallelism model of GPMCs is symmetrical multithreading, as we deal with homogeneous architectures. GPMCs target coarse grain MPMD or SPMD workloads. Programmers cannot control scheduling and mapping through the programming model - these are typically done by the operating system. Memory consistency and contention are additional problems: consistency may impact solution correctness, while memory contention often limits performance.

The hardware features with the most influence on platform performance are the multiple hardware threads, the caches, the memory hierarchy, as well as OS-based scheduler.

GPMCs are best suitable for coarse-grain parallel workloads, i.e., applications consisting on multiple complex, yet independent tasks, or massive data-parallel processing.

Intel Threading Building Blocks: Intel Threading Building Blocks (TBB [8]) is a C++ library with a strong focus on data parallelism. It centers around the concept of tasks instead of threads where tasks are performed on data in parallel. The library provides scalability by means of recursive subdivision of data and tasks that perform work-stealing. The library has three types of building blocks. It contains built-in constructs like `parallel_reduce` that can be composed recursively, container classes that are thread-safe, and locking classes (although the usage of these is discouraged).

TBB can be used to parallelize parts of an existing C++ program. It provides parallelism at a level of abstraction that is above threads, with support for concurrent container classes and reduction constructs. However, it is still a flexible framework where programmers can also use lower level constructs.

With the predefined constructs, TBB offers a global view, and ensures that algorithms remain general. TBB is rather flexible and offers both task and data parallelism, and good control over task creation, and granularity. TBB does not offer mechanisms to specify data layout, task mapping or data transfers, but it is possible to control task scheduling. It is also possible to perform communication and synchronization by hand but it is not recommended. The model allows other optimizations at a later stage.

Ct: C for throughput (Ct [9]) extents C++ with support for nested data parallelism. The programming model centers around a special throughput vector (TVEC) which allows more irregular data parallelism than flat data parallel models. The model targets GPMCs, and aims to be scalable when the number of cores on a chip increases. A TVEC represents a single assignment vector that can be constructed to hold values of various scalar types and perhaps in the future also structures. A TVEC can also contain nested TVECs, achieving nested data parallelism. TVECs are constructed in a separate memory space and garbage collected by the Ct Memory Manager.

The model is limited to nested data parallelism, but it offers a global-view of computation. Therefore, the algorithms that conform to this model are general enough to be portable to other families of architectures. The model does not have the concept of tasks, but it does allow to specify the data layout with help of the TVEC construct. Ct is high-level which means that it does not offer control over mapping scheduling, data transfer, communication and synchronization. It also obstructs other optimizations.

Ct has been discontinued as it has been re-used and re-branded in the ArBB model.

Intel Array Building Blocks: Array Building Blocks (ArBB) is a continuation of C for throughput (Ct [9]) with some features from RapidMind[10]. It extends C++ with a library, JIT compiler, and ArBB specific constructs such as special for and while statements. The programming model centers around special data containers with support for regular data (dense or sparse) and irregular data. Through the use of these data types, ArBB supports data and nested data parallelism. Operations, such as reductions on these data types are logically performed in a separate memory space and garbage collected to obtain a deterministic programming model.

ArBB targets GPMCs using vector instructions, and aims to be scalable when the number of cores on a chip increases or vector instructions become wider. The model is well suited to parallelize data intensive parts with numerical computations in an existing C++ program.

There are special copy in and copy out instructions to logically define the data in the ArBB memory space. This does not mean that the data is actually copied, but from the programming model point of view this results in a fragmented view of algorithms. The users have no control of task granularity as the dynamic compilation phase of ArBB operations takes care of this. Users have control over data layout by specifying their data structures in a different way.

Summary: All three models preserve the native parallelism models of GPMCs - ArBB and Ct focus on SPMD, as they focus on data-parallel workloads, while TBB allows for more generality by supporting both SPMD and MPMD, and focusing on task-parallel workloads. All models simplify both the data distribution and the communication/synchronization between threads (in the limits of the types of workloads they consider). None of the model has a clear definition of a concurrency unit and granularity - arbitrarily sizes data elements (ArBB) or functions (TBB) are used instead. None of the models deals with the memory hierarchy or cache optimizations - these are left to the user and/or the compiler. Mapping is offloaded to a run-time system (which maps the model's concurrency units on threads), while scheduling is offloaded to the OS (which maps virtual threads on hardware cores/threads). Overall, TBB offers more flexibility in expressing a parallel solution, but it has no support for parallel solution design.

B. GPGPU Programming Models

Programming GPUs combines coarse-level parallelism (offloading) and fine-grained parallelism (massive multi-threading): the host CPU offloads the data-parallel *kernels* as large collections of threads on the GPU. The native parallelism models for the GPUs self are SIMD/SIMT[3], with medium and low granularity. Note that due to the time-consuming offloading of the kernels from the host (CPU) to the GPU, too low-granularity kernels are only suitable for these architectures in large numbers.

For GPUs, the architectural features with the highest impact on programmability are: the very large number of threads and the dynamically partitioned register counts (i.e., registers are dynamically partitioned between running threads, resulting is a trade-off between using more registers per thread, or more threads with less registers per thread), the hardware-based mapping and scheduling, the complex memory hierarchy and its parameters (sizes, latencies, and bandwidths), and the different memory spaces (host and device).

[3]SIMT stands for "Single Instruction Multiple Threads"; it can be seen as a finer-grained version of SIMD.

GPUs are typically used for highly data-parallel workloads, where hundreds to thousands of threads can compute concurrently.

NVIDIA CUDA: NVIDIA's native programming model is called CUDA [11]. Based on C, CUDA uses language extensions for separating device (i.e., GPU) from host code and data, as well as for launching CUDA kernels. An advantage of NVIDIA hardware and CUDA is that the application does not have to do vectorization, since all cores have their own address generation units. All data parallelism is expressed by using threads. The programmer has to explicitly group threads in thread blocks. All threads in a block run on the same streaming multiprocessor. Thread blocks are in turn grouped in a grid.

While considered a fairly simple programming model, CUDA is still a low-level tool, and requires a lot of programmer's insight and experience to claim impressive performance results. With CUDA, one essentially explicitly subdivides the work over the streaming multiprocessors, and has to define correct and suitable grid configurations. In addition, the programmer has to consider many details such as memory coalescing, the texture cache, etc.

CUDA is global algorithm view model, where kernels need to be separated (using special constructs) from the host code and explicitly launched. The model uses kernels as the main concurrency unit for the overall application, and data elements for the SIMT/SIMD parallelization of the kernels themselves (finer granularity). The application data layout is also specified in two layers: the data structures used by the kernels are simply moved when and where they are needed; for the kernels themselves, the data layout results from the access patterns of the threads. Mapping and scheduling are performed by the hardware, and the data transfers from host to device are explicit. Low-level optimizations are left to the user.

Stanford Brook / AMD Brook+: In terms of workloads, ATI GPUs are targeting similar applications as NVIDIA's processors: highly data-parallel applications, with medium and low granularity. Therefore, choosing between the two becomes a matter of performance and ease of programming. For high-level programming, ATI adopted *Brook*, which was originally developed at Stanford [12]. ATI's extended version is called *Brook+* [13]. In contrast to CUDA, Brook+ offers a programming model that is based on streaming. Therefore, a paradigm shift is needed to port CPU applications to Brook+, making this a more difficult task than porting applications to CUDA.

With Brook+, the programmer has to do the vectorization, unlike with NVIDIA GPUs. Brook+ provides a feature called *swizzling*, which is used to select parts of vector registers in arithmetic operations, improving readability. In our experience, the high-level Brook+ model does not achieve acceptable performance. The low-level CAL model that AMD also provides does, but it is difficult to use. Recently, AMD adopted OpenCL as a high-level programming model for their GPUs, but also for the CPUs, and therefore Brook is most likely discontinued.

Brook is a fragmented view model, which uses explicit data transfers between host and device and implicit data layouts and transfers on the device itself. The concurrency units are kernels (coarse granularity) and stream elements (fine granularity). These are both controllable through the code, using language constructs. Mapping and scheduling are implicit. The model allows low-level optimizations.

PGI Fortran & C Accelerator Programming Model: Using PGI's Accelerator compilers[14], programmers can accelerate applications by adding OpenMP-like compiler directives to existing high-level Fortran and C programs. In this respect, PGI's programming model is similar to PathScale's. Compute intensive kernels are offloaded to the GPU, using two levels of explicit parallelism. There is an outer *forall* loop, and an inner synchronous SIMD loop level.

Based on sequential code reuse, PGI Accelerator is a global view model which uses pragmas to separate the potential kernels (its main concurrency units). Data layouts and transfers are both implicit, as kernels are automatically offloaded and parallelized. As the model uses CUDA as back-end, most of the implementation features - hardware-based mapping and synchronization, SIMT-based granularity, etc. - are inherited from CUDA. Still, the model does not allow for hand-tuning or architecture-specific optimizations on potential kernel code, as these interfere with the ability of the compiler to automatically parallelize the kernel loops.

Pathscale ENZO: The PathScale compiler company[4] has recently released a GPU software suite called ENZO. Although the programming model is device-independent, it initially targets NVIDIA GPUs and GPMCs, as well as hybrid combinations between these two. ENZO comes with it's own hardware device drivers, which focus on computing and do not support graphics. This way, PathScale expects to achieve better performance than CUDA.

With ENZO, programmers annotate their code with directives to indicate which code regions should be parallelized on the GPU. The C with annotations approach is similar to the OpenMP's and PGI's Accelerator model, preserving their advantages (e.g., portability, relatively high-level, and starting from sequential code), as well as their drawbacks (e.g., important architecture-specific optimizations cannot be expressed).

ENZO is a global view model, using pragmas for kernels' granularity control and mapping/scheduling. It relies on hardware-based mapping and scheduling at kernel level. Data transfers are automatically generated and performed, and there are no special constructs for data layouts. To compensate for the lack of low-level optimizations, the model uses pre-optimized libraries, code generators and auto-tuning to improve kernel performance.

Summary: All four GPU programming models are very close to the architecture; they all approach application parallelization by identifying and offloading the potential kernels to be accelerated. The way the offload is performed differs: CUDA and Brook require users to code this offload explicitly, while PGI and ENZO work by isolating kernels (with user-

[4]See http://www.pathscale.com.

inserted pragmas) from available sequential code. Further, the models tackle the massive parallelism in the kernels differently. PGI and ENZO use a compiler to extract and exploit the fine-grain concurrency units; CUDA requires the programmers to do so manually, while Brook uses streams to help the user identify the fine-grain concurrency. None of the models allows for specifying granularity requirements - these are guessed and tuned by the user. Data distribution is simplified, but not optimized - i.e., the models do not tune the virtual organization of threads (blocks and grids) to match the application requirements). Mapping and scheduling are left to the hardware (impossible otherwise). All models but CUDA (the native model) simplify the offloading procedures, minimizing the problem of different memory spaces. The effective use of the accelerator memory hierarchy remains the duty of the programmer.

C. Cell/B.E. Programming Models

Cell/B.E. programming is based on a simple multi-threading model: the PPE spawns threads that execute asynchronously on SPEs, until interaction and/or synchronization is required. The communication between the SPEs and the PPE is bidirectional and on-demand. All data distribution, task scheduling, communication, and processing optimizations are to be performed "manually" by the user/application. As a result, programming the Cell/B.E., and especially optimizing the code for Cell/B.E., are notoriously difficult.

The hardware features with the highest impact on programmability are: the heterogeneous cores, the inter-core communication and synchronization, the manual work- and data-distribution, the task scheduling (including thread management), the SIMD intrinsics, and the DMA operations.

Given that the architecture supports MPMD, SPMD, and SIMD parallelism, it is suitable for both coarse- and fine-grain parallel applications, and especially for workloads that exhibit nested parallelism (i.e., combinations of MPMD/SPMD and SIMD).

IBM Cell SDK: The SDK is the native Cell/B.E. programming model, developed to allow full flexibility for any workload. The model offers all the needed constructs to define the PPE and the SPE codes, and their interaction. With the SDK, programmers design and develop the main control flow on the PPE (using simple C or C++ code), and the computation kernels for the SPEs (using C and special intrinsics for optimized processing). Both SPMD and MPMD execution is supported for the SPE kernels. The SPE kernels are managed and coordinated by the PPE. Data layout is implicit - the SPEs and the PPE collaborate in data transfers. There are no special constructs for data layouts, as most of these are settled through programmed DMA accesses to the main memory.

To summarize, the SDK is a fragmented view model, with kernels as main concurrency units. Granularity is derived from code (no special constructs are provided), and concurrency is achieved by running different threads on the SPEs. Data transfers are all explicit, and programmed by the user using DMAs; so are synchronization and communication, which use special channels, but still require code to manage the protocol. Mapping and scheduling are performed by the user via the PPE code. Low-level optimizations (mostly vectorization and SIMD operations) are also explicitly performed by the user.

ALF: ALF (Accelerated Library Framework [15], [16]) provides a set of functions/APIs for the development of data parallel applications following an SPMD model on a (multi-level) hierarchical host-accelerators system (PPE-SPEs and/or host-Cell's). In ALF, the *host* runs the control task and the *accelerators* run the compute tasks. The same program runs on all accelerators at one level. ALF provides data transfer management, task management, double buffering support, and data partitioning. Further, the model uses three types of code: (1) accelerator optimized code (compute tasks optimized for a specific accelerator), (2) accelerated libraries (kernels and their interfaces, including the data management and buffering), and (3) applications (user-defined aggregation of compute tasks). Overall, ALF is an elegant high-level model which offers high productivity and, provided with the right libraries, also very good performance.

Feature-wise, ALF is also a fragmented view model, with coarse, task-level granularity. The model is hierarchical, but focuses on SPMD parallelism. Kernels are defined by the programmer as concurrency units. Kernel mapping and scheduling are solved transparently by the runtime system; the same holds for communication and synchronizations. Data layout can be pre-set for the SPMD tasks, and data-transfers, communication and synchronization are implicit. Mapping and scheduling are performed by the runtime system. Optimizations are typically performed by the user in the kernel code, but the model can use imported kernel from pre-optimized libraries.

Cell SuperScalar: Cell SuperScalar [5] (CellSS) is a pragmatic model, suitable for quick porting of existing sequential applications to the Cell/B.E. [17]. CellSS uses a compiler and a runtime system. The compiler separates an annotated sequential application in two: a PPE part (the main application thread), and the SPEs part (a collection of functions to be offloaded as SPE tasks). The runtime system maintains a dynamic data dependency graph with all active tasks. When the PPE reaches a task invocation, it requests the CellSS runtime to add a new task to the execution list. When a task is ready for execution (i.e., all its data dependencies are satisfied and there is an SPE available), the DMA transfers are transparently started (and optimized), and the task itself is started on the available SPE. Various scheduling optimizations are performed to limit communication overhead. Additionally, CellSS provides execution tracing, a mechanism included in the runtime system to allow performance debugging by inspecting the collected traces.

When starting from suitable sequential code, CellSS is a very productive global-view model for first-order implementations of Cell applications. As mapping and scheduling are dynamically optimized, the performance depends on the

[5]The model is based on the principles of Grid SuperScalar, a successful grid programming model.

kernels' performance. As kernels are generated from sequential user-written functions, low-level optimizations need to be performed by hand, and some manual (re)sizing might be needed to avoid task imbalance.

Summary: The three models presented here are quite different: the SDK is the native model, which provides the means to program the platform, but offloads all tasks to the user; ALF includes some rudimentary design elements (builds a hierarchy of tasks), and from there it derives additional optimization for mapping and scheduling; CellSS enables quick parallelization for the Cell, guided entire by the user. Except for the SDK, the presented models simplify (1) mapping and scheduling at the MPMD/SPMD level on the SPEs (based on a run-time system), (2) data distribution, including the complete DMA transfers, with reasonable optimizations (prefetching, double buffering), and (3) inter-core communication. Low-level optimizations are left to the user (with potential help from the compiler). The concurrency units and their granularity are arbitrarily chosen by the user, but none of these three models helps in properly estimating them.

V. Generic Programming Models

Our analysis on hardware-centric models shows that their design support is rather rudimentary. Rather, these models focus almost entirely on simplifying the user experience by tuning application parallelization to match a chosen platform. Given the size of the parallelization problem the software community is facing - i.e., all applications have to be parallelized, sooner or later, to make effective use of many-cores - using hardware-centric models becomes a non-scalable solution:

Alternatively, one can use generic programming models (i.e., models which run on more than one family of many-core processors), focusing first on the application parallelism, and only second on mapping the parallel solution on one platform or another. We split this "generic" models in two categories: parallelism-centric models and application-centric models. We briefly discuss each category.

A. Parallelism-centric models

Parallelism-centric models are built to allow users to express typical parallelism constructs in a simple and effective way, and at various levels of abstraction. The higher the level of abstraction is, the less (explicit) parallelism constructs are available, but also the less the flexibility and expressibility of the model are. Parallel-centric models are typically used to express complex parallel algorithms (i.e., the design of the parallel solution for the application and its implementation are decoupled).

Threads with shared memory: Threading libraries such as POSIX threads or the Java Thread class extend the sequential imperative programming model in a natural way to obtain parallelism. Functions are spawned as new threads that globally share data.

Threads provide mechanisms on the lowest level of abstraction of parallel programming and are very flexible. It is natural to spawn threads with different functions to obtain task parallelism, but threads can also be spawned in a loop, with the same function operating on different data, which results in data parallelism. There is extensive synchronization support, such as joins, barriers and condition variables. Threads offer a fragmented-view as programmers need to divide data among threads and join for the results. Algorithms expressed in this model are not portable to other architectures and there is no concept of tasks that can be sized or resized other than functions. Users have no control over mapping and scheduling and the model has no specific means to change the data layout. Threads do allow other low-level optimizations.

The flexibility of threads provides programmers lots of control over task creation and synchronization. Threads are well suited for coarse-grained tasks that need much synchronization. Because threads are so low-level, programmers have many opportunities to optimize on for example synchronization.

MPI: The Message Passing Interface (MPI)[5] targets both distributed memory systems and shared memory machines, but is normally used for distributed memory. An application consists of multiple processes that communicate with messages. MPI gives much control due to the strong separation of communication and computation and is not suitable for fine-grained parallelism.

MPI is the typical example of a fragmented-view programming model. The user is responsible for all communication and synchronization. Explicitly parallel algorithms are needed, which results in algorithms being not easily portable to another family of platforms, or even other platforms of the same family. There is no way to size MPI tasks other than changing the program. Data transfer and layout is explicit but only on a high granularity. Communication and synchronization is explicit by means of messages and the user has no control over how processes are scheduled. However, it allows other optimizations at a later stage. For example MPI can be mixed with OpenMP [18].

MPI is well suited for applications where the input and communication is static, for example a regular data structure that can be divided in regular coarse-grained blocks that each are computed on different processors.

Cilk: The language Cilk allows programmers to write parallel divide-and-conquer programs. It extends C and C++ with keywords such as `spawn` and `sync`. A `spawn` in front of a function call creates a non-blocking function call that is executed in its own thread and may spawn other (possibly recursive) functions. Multiple consecutive spawn function calls create parallelism in the program. The keyword `sync` blocks the calling thread until the results of the spawned function calls are available.

Cilk offers a global view of the algorithm. Syntactically, a sequential divide and conquer algorithm is very similar to a parallel divide and conquer algorithm. The language is limited to divide-and-conquer parallelism. The model gives no control over tasks. Programmers often control the granularity of the recursive task by manually choosing between a sequential version or parallel version based on the size of the data that

needs to be processed.

Data is communicated to threads by using parameters of spawn function calls. The model offers several ways to obtain more control over synchronization. For example, the `abort` keyword aborts other spawned threads. A typical use-case is a parallel search where one thread finds an item and aborts the others. Another example is the use of inlets that guarantee that results of spawned threads are treated atomically with respect to the other spawned threads.

The Cilk system dynamically schedules threads and dynamically maps them to hardware.

B. Application-centric models

Application-centric models tackle application parallelization from design to implementation. Some of them also include several generic optimizations. These models have less explicit parallelism constructs. Their goal is to help users to find an effective, (partially) platform-agnostic parallel solutions for their applications, and implement them using a limited set of concurrency, granularity, and parallelism constructs.

CHARM++ and the Offload API: CHARM++ is an existing parallel programming model adapted to run on accelerator-based systems[19], [20]. A CHARM++ program consists of a number of *chares* (i.e., the equivalents of tasks) distributed across the processors in the parallel machine. These chares can be dynamically created and destroyed at run-time, and can communicate with each other using messages.

For accelerators, Charm++ uses an Offload API: a chare can offload *work requests*, which are computation-intensive kernels to be accelerated by the accelerators (e.g., the SPEs on the Cell/B.E.); on the host side (e.g., the PPE on the Cell/B.E.), the Offload API manages each work request, coordinating data movement, execution, and completion notifications.

Charm++ is a fragmented view model, with coarse parallelism expressed by chares. The model supports both SPMD and MPMD. The granularity is controlled at design time, by defining the chares; data distribution is implicitly defined by the data usage of these chares and data transfers are automated. The dynamic mapping and scheduling, together with the highly optimized data transfers contribute to a high performance potential.

Sequoia: Sequoia requires the programmer to reason about a parallel application focusing on memory locality [21]. A Sequoia application is a tree-like hierarchy of parametrized tasks; running tasks concurrently leads to parallelism. A tree has two different types of tasks: inner-nodes, which spawn children threads, and leaf-nodes, which run the computation itself. The task hierarchy has to be mapped on the memory hierarchy of the target machine (not only Cell/B.E.) by the programmer. Tasks run in isolation, using only their local memory, while data movement is exclusively done by passing arguments (no shared variables). One task can have multiple implementation versions, which can be used interchangeably; each implementation uses both application and platform parameters, whose values shall be fixed during the mapping phase. For the Cell/B.E., all inner nodes run on the PPE, while the leaf nodes are executed by the SPEs. The PPE runs the main application thread and handles the SPE thread scheduling. Each SPE uses a single thread for the entire lifespan of the application, and it continuously waits, idle, for the PPE to asynchronously request the execution of a computation task.

As a generic model, Sequoia uses a fragmented algorithm view. Based on coarse-level granularity SPMD parallelism, the model requires applications to be designed using a divide-and-conquer approach. The granularity can be controlled at both compile time and runtime. Data transfers are implicit. The special feature of the model is its user-defined mapping and scheduling (by user-file), which also results in implicit, yet automated data layouts. Optimizations are allowed as different versions of the same kernel can be user interchangeably during the lifespan of the application.

Still, Sequoia has limited productivity, as the model is difficult to use for non divide-and-conquer applications. The application and machine decompositions are independently reusable. The manual application-to-machine mapping offers a flexible environment for tuning and testing application performance.

Pattern-based models: OPL: Pattern-based models allow users to focus entirely on application analysis. An application is built as a composition of nested patterns. Once all these patterns are implemented for a platform, their composition, also a pattern, leads to a complete application on that platform. Pattern languages use different classes of patterns, applied at different stages of application design.

First, the high-level application structure is described in terms of structural patterns (a graph of tasks) and computational patterns (the computation of each task). Next, the algorithm strategies patterns are employed: they identify and exploit application concurrency as exposed by the structural patterns. The way the program and data are organized is specified by implementation strategies patterns, which are ways of implementing each algorithmic pattern. Finally, the low-level parallelism support, matching both the application and the target architecture, is included in the so-called parallel execution patterns.

A pattern-based language is a promising abstract concept, being elegant, generic, systematic, and allow for feedback loops and incremental re-design. Implementing such a language, however, requires platform-specific pattern implementations, an effort that depends on the number of commonly-used patterns. Furthermore, the first and last categories of patterns - the structural and the parallel execution, are not trivial to apply to a new application. Structural mistakes can significantly affect performance, but the simplicity of the model and, eventually, the limited number of structural patterns should allow for extensive auto-tuning.

One example of a pattern-based language is OPL (Our Pattern Language), developed at Berkeley[6]). Details on the five categories of patterns the language offer can be found

[6]http://parlab.eecs.berkeley.edu/wiki/patterns/patterns

in [22]; however, there is no implementation of OPL so far, so any technical details are missing. Therefore, we consider that the the practical side of this solution is yet to be proven.

OmpSs: OmpSs[2], [3] addresses the programmability of heterogeneous architectures by allowing the user to exploit task-level parallelism [7] using a similar approach to that of OpenMP. Based on pragmas, the model uses a source-to-source translator to separate the code in dedicated programs for the different components of the heterogeneous system; furthermore, a the runtime system schedules tasks to execution, preserving and optimizing the dependencies among tasks.

The system is based on incremental parallelization of a single-source code, allowing step-by-step restructuring and optimization, and a separation of the implementation from the platform specific details (which are, of course, encapsulated in the runtime system). OmpSs is also portable, as the same code (typically, a sequential C or FORTRAN application with pragmas) runs on any machine where the backend is ported. Programmers may choose to apply platform specific optimizations (i.e., design and implement platform specific versions of the tasks), but they may also choose to ignore them, preserving portability at the expense of performance.

OmpSs is a global view model with coarse granularity, MPMD (and SPMD) parallelism, implicit data transfers, pragmas for data distribution, and runtime-based dynamic mapping and scheduling. Platform-specific can be applied "outside the model" - i.e., the model allows multiple kernel implementations to be plugged in.

OpenCL: OpenCL was proposed as a in 2008 (by the KHRONOS group) as a solution to the platform diversity problem. OpenCL proposes a common hardware model for all multi-core platforms. The user programs this "virtual" platform, and the resulting source code is portable on any OpenCL compliant platform[8].

The OpenCL platform model consists of a host connected to one or more OpenCL compute devices. A compute device is divided into multiple compute units (CUs); CUs are divided into multiple processing elements (PEs); PEs perform the computations. Each PE can either behave as a SIMD or as a SPMD unit: a kernel is executed concurrently on multiple processing elements, each with its own data and a shared program counter or each with its own data but its own program counter. SIMD execution implies that all PEs execute a strictly identical set of instructions, which cannot be always true for SPMD due to possible branching in a kernel. Further, the OpenCL platform has a multi-level shared memory model, featuring four distinct memory spaces: private, local, constant and global. Private memory can only be used by a single compute unit (like registers in a single compute unit). Local memory can be used by the work-items in a work-group (similar to on-chip shared memory). Constant memory can be used to store constant data for read-only access by all of the compute units in the device during the execution of a kernel. The host allocates and initialize the memory objects from constant memory. This is similar to the constant caches that are available on AMD GPUs. Finally, global memory is memory that can be used by all the compute units on the device. This is similar to the off-chip GPU memory that is available on AMD GPUs. Note that the mapping on real hardware depends on real memory subsystem, and it might require different memory spaces are to be collapsed together.

An OpenCL program has two parts: the compute kernels that will be executed on one or more OpenCL devices, and a host program that defines the context for the kernels, initiates and manages their execution. The "main" OpenCL application runs on the host, and submits commands from the host to be executed on the processing elements within a device. Kernels can run either in-order or out-of-order. Events allow checking the status of outstanding kernel execution requests and other runtime requests. The execution domain of a kernel is defined by an N-dimensional computation domain, which signals how large of a problem the user needs to solve. Each element in the execution domain is a work-item and OpenCL provides the ability to group together work-items into work-groups for synchronization and communication purposes.

VI. The Unified Model: OpenCL or OmpSs?

So far, we have discussed twenty programming models for many-core processors - and these are only the representative ones. Each one of these models has its qualities and drawbacks, and has a positive impact on platform programmability. Still, many of them are used only sporadically. We believe this poor adoption of high-level programming models is due to a large extent to the multitude of models available. It is difficult for potential users to understand limitations of each model, the differences between models, or the impact a model has on a specific workload. Therefore, programmers choose the native models which, although cumbersome and low-level, offer full flexibility.

We believe that the space of programming models needs to be pruned to only a few items: generic models and application-specific models. In this context, we extract two candidates for a unified many-core programming model - OpenCL and OmpSs - and compare them from this perspective. While we are not able to extract a clear winner, we can make several recommendations for each of the models to improve the potential impact on platform programmability.

A. Programmability Impact

We briefly analyze the impact of both OpenCL and OmpSs on platform programmability by looking at all its three components, as described in Section III-A.

1) Productivity: OpenCL supports application-centric programming, covering a good mix of both design and implementation features. Application parallelization starts from a specification and/or an algorithm. It is a kernel-based global

[7]Instances of the generic StarSs programming model include GRIDSs (for the Grid), CellSs (for the Cell B.E.), and SMPSs (for multi-core processors), GPUSs (for GPUs); Ss stands for Superscalar.

[8]Currently, ATI's and NVIDIA's GPUs, AMD's and Intel's multi-cores, ARM's embedded processors, and the Cell/B.E have hardware drivers and compiler back-ends for OpenCL.

view model: it preserves the algorithm design and offloads designated kernels to be accelerated. Kernel mapping and scheduling are not controllable, but they can be influenced by using asynchronous queues. Fine-grain data parallelism is well supported. The model has no definition of a task (and, implicitly, no control over sizing and composition), but kernels can be used to implement some degree of task parallelism, provided that the hardware platform supports it. There are virtually no limitations in expressing parallelism in OpenCL. However, the way this parallelism is translated from the virtual OpenCL platform to the real hardware platform is hidden behind vendor-specific drivers, and can be counterintuitive.

In contrast, OmpSs requires sequential code to produce a parallel application by offloading the kernels (specified by the user) and parallelizing them accordingly. OmpSs is also a global-view model. Despite its less developed parallel design, OmpSs allows for quick parallelization of available applications by user-placed pragmas. This approach leads to excellent productivity. Kernel-level optimizations are very similar to the ones in OpenCL (in fact, one of the back-ends of OmpSs is OpenCL). There can be limitations induced at design-level by a poorly written sequential application. In this case, a fall-back to a parallel solution designed from scratch is recommended (for which an implementation in OpenCL is an alternative). Finally, OmpSs is slightly friendlier for coding: the application is single-source (separated by the compiler in device code and kernels), and it is less verbose than the original OpenCL (the context setting code is generated).

The most difficult problem remains the parallelization from scratch. For OpenCL, applications need to fit the architectural model of the common middleware. These are workloads with large collections of fine-grain work items, very limited synchronization, and no need for user-controlled mapping and scheduling; the applications can only be driven/managed from the host, with no device-initiated communication. Essentially, applications with coarse tasks and a lot of interdependencies are unsuitable for OpenCL. OmpSs requires a good sequential implementation (or at least its control-flow skeleton together with hardware-specific kernels) to generate a good parallelization. This is a much quicker design, but the performance behavior will vary a lot on real hardware platforms.

Overall, OmpSs is more productive than OpenCL when sequential code is available. On the other hand, already parallelized codes (such as, for example, CUDA applications) are typically much easier to translate to OpenCL than (directly) to OmpSs.

2) Portability: The strongest point of OpenCL is its portability by construction. This is a result of using the *common platform model* as a virtual middleware. Further, this separates the design and implementation concerns: OpenCL's back-end targets one machine type only, and it is the responsibility of the hardware vendors to provide the OpenCL drivers; programmers are only concerned with designing a parallel application for a given platform model. Note that the portability of OpenCL (and its relative success in both the academia and the industry) is a good incentive for all processor vendors to produce high-performing OpenCL drivers for their platforms.

OmpSs is portable at source-level: each platform for which OmpSs is available has its own compiler and run-time system, fully optimized for the specific platform. While this is, in a sense, similar to OpenCL's portability provided by the drivers, the incentive and the driving force behind the model are significantly lower. Thus, we expect the number of devices supported by OpenCL and not by OmpSs to increase rapidly.

Overall, the two models are equally portable. However, the performance penalties that are inferred by this portability might differ significantly.

3) Performance: Finally, when it comes to the performance impact, OpenCL is expected to enable GPUs to achieve comparable results to their "native" models - there are no fundamental reasons for OpenCL to behave worse: parallel design is similar, low-level optimizations can be enabled, and CUDA-like execution can be emulated perfectly. Therefore, bad-performing drivers and/or compilers, as well as bad programming, could be the only causes for lower OpenCL performance. For the GPMCs and the Cell, the performance depends on (1) how well the application maps to the OpenCL platform, and (2) how well the hardware maps to the OpenCL platform. Overall, the few performance case-studies available so far real applications, are inconclusive [23], [24], [25], [26]: performance variations range between 10% and 90%. Our own preliminary evaluations show that, for fair comparisons, the gap is not larger than 10%.

For OmpSs, performance penalties come from two sources: the poor parallelization of the application (due to bad use of pragmas or unsuitable sequential code) and non-optimized kernels. The first one can be attributed to programmers, and it is impossible to eradicate. The second one is addressed by the model itself, which allows multiple implementations of the same kernel to be included in the program, and the system can choose, at run-time, which implementation is the most suitable one for the platform in use. However, there are also performance improvements that are specific to OmpSs: its run-time system provides dynamic mapping and scheduling, taking into account data locality and minimizing data transfers. Therefore, it is most likely that GPU-like applications will perform about the same in both models, while for the GPMCs and the Cell, the OmpSs will perform better.

Performance-wise, the systems are comparable; a more precise ranking can only be made on an application basis.

B. Recommendations

Based on the detailed analysis provided above, we conclude that both models have provide a mix of features and good impact on platform programmability to become generic models for many-core processors. Before that, more case-studies have to be developed, to assess if some of the theoretical assumptions we have made here hold in practice for real workloads.

Still, we recommend two improvements for each model. For OpenCL: (1) provide basic communication options to allow the computing elements to communicate (at least to the host) and

(2) allow some degree of control for mapping work items on the platform, enhancing support for the task parallelism. And for OmpSs: (1) find a higher level application specification to be used for cases when sequential code fails, and (2) enable an auto-tuning like technique to allow the automated selection (or even the tuning/generation) of the best available kernel implementation for a given platform.

VII. Conclusions and Recommendations

Multi-core processors are here to stay. With that, application parallelism has become mandatory. The software community is (suddenly) faced with a large problem: virtually every application will have to run on a parallel machine, rather sooner than later. And with multi-core complexity steadily increasing, addressing applications and machines as one pair at a time will be counterproductive.

Using hardware-centric programming models can speed-up the implementation process per platform, but the lack of portability will eventually lead to lower performance and/or lower productivity. Therefore, a more effective way to address the mass-parallelization problem is to focus on *application-centric programming*: first design a parallel solution for the problem, and then implement it on one/multiple platform(s). However, this is easier said than done: we analyzed twenty many-core programming models, and showed that most of them lack either the design or the implementation component.

Our analysis leads to three important conclusions. First, the perceived difficulty of many-core programming generates a lot of hardware-centric models, platform-specific and non-portable. Second, application-centric models can be used to improve platform programmability, as they offer increased productivity and portability with minor performance penalties. Third, although there is no unified model for efficient programming of many-cores, OpenCL and OmpSs are promising candidates for achieving such a consensus.

For the near-future, we have three suggestions. For hardware vendors: support OpenCL by developing drivers for your platforms, rather than building yet another hardware-centric programming model. For (third-party) programming model designers: support application-centric models by using OmpSs or OpenCL as a compilation target for your higher-level models. For programmers: use OpenCL and OmpSs for prototyping and development - a quicker adoption will speed-up their development, increasing the chances for a unified model for programming many-cores to emerge.

References

[1] OpenCL committee, "OpenCL 1.1 standard," http://www.khronos.org/opencl/, October 2010.

[2] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Optimizing the exploitation of multicore processors and gpus with OpenMP and OpenCL," in *LCPC2010*, October 2010.

[3] E. Ayguade, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez-Gonzalez, J. Labarta, L. Martinell, X. Martorell, R. Mayo, J. M. Perez, J. Planas, and E. S. Quintana-Orti, "Extending OpenMP to survive the heterogeneous multi-core era," *International Journal of Parallel Programming*, vol. 38, no. 5–6, pp. 440–459, june 2010.

[4] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[5] M. P. I. Forum, "MPI: A Message-Passing Interface standard," 1994.

[6] K. Kennedy, C. Koelbel, and H. Zima, "The rise and fall of high performance fortran: an historical object lesson," in *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. New York, NY, USA: ACM, 2007, pp. 7–1–7–22.

[7] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948 –960, 1972.

[8] J. Reinders, *Intel Threading building blocks*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2007.

[9] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and B. Chen, "Future-proof data parallel algorithms and software on intel multi-core architecture," *Intel Technology Journal*, vol. 11, no. 4, pp. 333–348, 2007.

[10] M. McCool, "Developing for GPUs, Cell, and multi-core CPUs using a unified programming model," http://www.linux-mag.com/id/6374, July 2008.

[11] *CUDA Programming Guide*, nVidia, 2007.

[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," in *ACM Transactions on Graphics, Proceedings of SIGGRAPH 2004*, Los Angeles, California, August 2004, pp. 777–786.

[13] Advanced Micro Devices Corporation (AMD), *AMD Stream Computing User Guide*, august 2008, revision 1.1.

[14] *PGI Fortran & C Accelerator Programming Model white paper*, version 1.2 ed., The Portland Group, March 2010, http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.2.pdf.

[15] *Cell/B.E. Programming Tutorial*, 2nd ed., IBM, December 2006.

[16] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca, "A rough guide to scientific computing on the playstation 3," ICL, University of Tennessee Knoxville, Tech. Rep. UT-CS-07-595, May 2007.

[17] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSS: A programming model for the Cell BE architecture," in *SC'06*. IEEE Computer Society Press, November 2006.

[18] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 404–418, 2009.

[19] D. Kunzman, "CHARM++ on the Cell Processor," Master's thesis, Dept. of Computer Science, University of Illinois, 2006.

[20] D. Kunzman and L. Kalé, "Towards a framework for abstracting accelerators in parallel applications: experience with cell," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[21] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *SC'06*. ACM Press, November 2006.

[22] K. Keutzer and T. Mattson, "Opl: Our pattern language," http://parlab.eecs.berkeley.edu/wiki/_media/patterns/opl-new_with_appendix-20091014.pdf, October 2009. [Online]. Available: http://parlab.eecs.berkeley.edu/wiki/_media/patterns/opl-new_with_appendix-20091014.pdf

[23] S. M. Cho, D. W. Im, O. Y. Jang, H. J. Song, B. D. Paulovicks, V. Sheinin, and H. Yeo, "OpenCL and parallel primitives for digital TV applications," *IBM Journal of Research and Development*, vol. 54, no. 5, pp. 1–14, September 2010.

[24] P. Du, P. Luszczek, and J. Dongarra, "Opencl evaluation for numerical linear algebra library development," in *SAAHPC '10*, June 2010.

[25] J. D. Sean Rul, Hans Vandierendonck and K. D. Bosschere, "An experimental study on performance portability of OpenCL kernels," in *SAAHPC '10*, June 2010.

[26] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of OpenCL programs," in *VecPar'10*, 2010.