Small or medium-scale focused research project (STREP)

Short proposal

ICT FET Open Call
FP7-ICT-2011-C

[PortabilityStack: A Software Stack for Productivity and Performance-Portable Parallel Applications]

[PStack]

Date of preparation: 1/3/2012

Version number: 2

Type of funding scheme: Small or medium-scale focused research project (STREP), short proposal

Work programme topics addressed: ICT-2011.9.1, FET Open

Proposal Abstract:

The productivity and efficiency of the global ICT industry, and by implication European ICT and its diverse customers, is under threat from the challenge of transitioning to heterogeneous, highly parallel hardware as the mainstream technology. The core need is a means to write parallel software once, productively, then have it automatically run high performance on all available hardware platforms, especially heterogeneous ones.

Attempts to provide this have been too narrow in scope, focused on a single language or a single run-time, or sub-set of hardware targets. The problem is larger than any one group can solve, requiring instead a new software stack, from the hardware-interface up to the programming tools, which is cooperatively provided by many players, each adding a small piece. In this way, many languages are supported, and the effort of supporting a language on a machine can be reused by other languages, which makes such a stack practical.

Our proposal addresses this challenge by delivering the seed of such a stack, including the interfaces and the process for independent players to update it with their own additional pieces. This will enable application sources to be compiled and executed, without source modification, and with good performance across the full range of current and, foreseeable, future hardware.

We are aware that this is a bold claim, which is considered beyond reach by many. However, this project takes a new, collective, approach. We provide structure that makes it easy and clean for players across industry to each supply one small piece of the solution, and our structure integrates them into a collective solution. To accomplish this, we will exploit two key advances: a recent theoretical model, used to design a set of portability constructs to be incorporated into languages, and a recent promising parallel hardware abstraction that provides an organizing principle and enables reuse. These are further augmented by a toolchain management approach, which makes it practical for 3rd parties to add new performance tools to the stack.

The seeds of each piece of the stack have been demonstrated by one partner or another. In the project, we will tackle the vision of creating an integrated whole from them, in a way that each part relies on and supports the others, and encourages diverse groups add to the stack independently, organically growing the solution.

Proposal

Section 1: Scientific and/or technical quality, relevant to the topics addressed by the call

The Need, which is to be Solved by the Proposal

The transition to highly parallel hardware as the mainstream technology has caused software to fall ever further behind the increasingly diverse array of parallel hardware available. It is too costly and too difficult to rewrite code for high performance for each hardware target. This slows progress in every segment of industry, given that all segments now have parallel hardware, from embedded devices to mobile devices to desktops to super-computers. As a result, European industry as a whole is harmed by the growing gap between new hardware and the ability and cost for applications to take advantage of it.

Despite the need to rewrite applications to efficiently use the new parallel hardware, this work has been delayed in the hope that new languages and tools would emerge to lower the number and difficulty, of rewrites. But economic pressure is forcing rewrites to happen anyway, using inefficient and non-portable techniques. The result is both increased cost and a slowing of software improvement, across industries, which harms the economy and delays important advances that would otherwise benefit European citizens. The problem only intensifies as new chips such as the 10 core Pentium, SCC, and MIC are released by Intel, while Tilera releases its 64-core chip, and AMD its new Fusion architecture.

The Nature of the Problem

The gap is due to infrastructure failing to adapt to the shift from single-processor to multi-processor and heterogeneous hardware. Previously, source code could take advantage of a new single-processor chip just by running the code through the new processor's development infrastructure (called a toolchain, and consisting principally of a compiler). But that isn't currently possible for parallel software. There exists no equivalent development infrastructure that can automatically optimize code for the wide array of new parallel hardware.

On the surface, the problem is that application code exposes features of the hardware, so it has to be tuned for each hardware target. But deeper down, the issue lies in the infrastructure, which supports parallel applications. Previously, the toolchain (compiler) was helped with the tuning process, because the hardware enforced its own abstraction, making all processors look similar. An equivalent abstraction doesn't exist for parallel hardware. In addition, the tuning of parallel applications requires more kinds of information than sequential languages to be captured. Hence, the infrastructure is starved of necessary input.

Productivity for parallel programming is widely considered to be provided by domain-specific languages. But language development and research is hampered by each group having to supply its own infrastructure for each language. The complex nature of parallel computation requires complex infrastructure, so this is non-trivial effort. To be practical, infrastructure needs to be reused across languages. But no standard exists, nor has even been proposed, for interfaces or other modularization that allows such reuse across languages and hardware.

Insights Relevant to the Problem

One insight as to how such infrastructure might be achieved is revealed by a survey of techniques used in hand-tuning parallel code. The study suggests the existence of an invariant set of manipulations, which, if captured inside the infrastructure, would automate most non-algorithmic performance tuning.

In addition, a recent theoretical model has been discovered that exposes the relationship among application features, scheduling, and hardware features. It identifies the underlying purpose of each type of manipulation, and bounds the type and amount information needed from an application.

Together, these give a good understanding of the structure underlying specializing code to particular parallel hardware. From that understanding, increasing clarity has been gained, which suggests a canonical set of information plus manipulations exists. These would enable near the best performance to be extracted, from a particular algorithm, on any of the known classes of parallel architecture.

Many of those required techniques already exist, but only in isolated tools that apply just to a narrow set of applications, or a narrow set of hardware. Each of these tools fills-in a small portion of applications-times-hardware combinations, so very few are actually used because of the impracticality of the dozens to hundreds of individual tools with different formats and steep learning curves that have to be manually applied. If all were collected within an automated system, that would be a significant step towards the sought-after infrastructure. However, there currently exists no means for them to inter-operate. That would require an organizing infrastructure.

The Solution

What is needed, to realize the sought-after infrastructure, is first a way to capture the extra parallelism-related information from the application; second an analogous hardware-abstraction that makes different parallel hardware look similar; and third, standard interfaces to integrate and interoperate independently developed tools and techniques. An infrastructure that embodies these will isolate application code from hardware, and encapsulate specialization of the application to hardware.

Such infrastructure will naturally take the form a software stack. At the top lie constructs to capture information, below are the tools that use the information, and at the bottom is the hardware abstraction that genericizes hardware. The tools layer uses the information, to map the application code onto the abstraction.

One twist, though, is the high diversity of parallel hardware. To keep high performance, the abstraction can't hide performance-relevant aspects of the hardware. This forces more than one abstraction to exist: a separate one for each class of parallel architecture. Fortunately, only major architectural features need to be exposed, such as shared memory vs distributed memory, memory hierarchy, vector capabilities, data layout requirements, and so on. From initial studies, the final number of abstractions needed appears to be reasonable, from 8 to 10 in total, due to composability, which covers heterogeneous hardware and multiple levels of hardware hierarchy.

Adopting multiple abstractions, for the multiple hardware-classes, adds another layer to the software stack, made up of the runtimes. This layer acts as a switch-board, which connects the programming languages above to the hardware abstractions below. A given language has a runtime for each class of hardware (kind of abstraction). Such runtimes encapsulate part of the specialization of source to hardware, because each runtime takes advantage of performance-related hardware features exposed in the abstraction it connects to. Recent experiments have shown that the runtime alone can contribute several orders of magnitude of effect on performance.

The Key Elements of the Solution

The keys to such a portability stack are the portability constructs at the top and the hardware-abstraction at the bottom, both supported by insights from the theoretical computation model. None of these elements exist in current and past attempts to achieve performant-portability.

The constructs at the top fulfill the needs of transforms in the toolchain, as well as the needs of scheduling and load-balancing in the runtime. Hence, all levels of the stack use information and patterns captured by the portability constructs. Each hardware's toolchain and runtime uses these to make hardware-specific transform and scheduling choices. For example, advanced high-performance compiler techniques, such as polyhedral, need properties of the code being manipulated, and those properties are captured by the portability constructs.

An organizing principle is needed to give structure to the stack, and is supplied by the abstraction at the base of the stack. Its job is to give a uniform view of the hardware, modularize runtime implementation, and standardize communication between layers. It should expose only performance-relevant features, and provide standard services to support synchronization, communication, and scheduling. The abstraction should make runtime construction simple, fast, and modular, in order to minimize the work of connecting a language to new hardware, as well as make practical the reuse of runtime technology between languages.

Differences from Other Approaches.

At first blush, something like posix-threads or TBB seems to also be a hardware-abstraction, and an established standard one, at that. However, they are more properly viewed as languages embedded into C. They don't provide an organizing principle for a stack built above them, nor do they give language runtimes intimate control over hardware, as VMS, to be used in this proposal, does. Finally, they have no features to reduce the difficulty of runtime implementation, nor to modularize the runtime to make reuse practical. VMS excels, in a unique way, at both those goals.

Additional Benefits of the Proposed Solution

Such a software stack reduces the disruption of introducing new hardware, and reduces cost for all players in the software eco-system. When a new parallel chip comes out, the application is not again modified, because the information needed for tools to specialize to the new hardware has already been captured by the portability constructs. This assertion is supported by results from the theoretical computation model. Only layers of the software stack get modified, to take advantage of the new hardware: the hardware-abstraction at the bottom is implemented for the new hardware; above that, new runtimes might be added to connect languages to the new hardware; above that, the languages may tweak their toolchains to take advantage of specific hardware features. This work takes place below the application, so it is reused by all applications. More importantly, it is done by specialists, so application programmers are insulated from hardware-related performance issues.

The organization encourages an eco-system, which collectively provides the pieces of the software stack and maintains it. The layered arrangement isolates different expertise for each layer, while the standard interfaces, provided by the VMS abstraction, and appropriate toolchain management, allow different players to easily cooperate, with only minimal real-world coordination.

In the eco-system, each entity provides on expertise and one piece, which collectively form the stack. Hardware manufacturers implement the abstraction on their own hardware, which collectively provides the bottom layer. Language-development entities and third parties contribute runtimes, relying on the interfaces above and below to isolate themselves from each other. Language developers then each focus on a certain application domain, providing a domain-specific language that includes the PStack portability constructs, and toolchains. In addition, performance-tool providers add specialized pieces to the toolchain layer. The modularity, standard interfaces, and toolchain management allow any one piece to drop-in. This way third-parties, like research groups, can freely add specialized tools to improve the toolchain and runtime under specific conditions.

The key to the viability of such an eco-system of independent real-world entities is the organizing principle of the VMS-based abstractions, the standard interfaces derived from the VMS approach, and automated management of the toolchains. These same things enable reuse of key runtime technology across languages, a natural way for narrow focus but high performance tools to gain wide exposure and low-barrier to acceptance. They will be hidden inside the toolchain layer, eliminating the practical difficulties currently blocking wide adoption.

This Project's Proposal

The aim of this project is to provide a working prototype of the portability software stack, as depicted in Figure 1. The portability constructs at the top will be refinements of initially demonstrated versions, and take advantage of recent theoretical models of parallel computation and compiler techniques to refine and expand on those. In the toolchain layer, automated specialization to target hardware will be managed by the BLIS approach. The middle and bottom layers will take advantage of a new hardware abstraction, called Virtualized Master-Slave (VMS), which has demonstrated the required properties. Partners with deep language experience will extend existing languages with the new portability constructs, allowing development in existing base languages, to minimize changes to existing code.

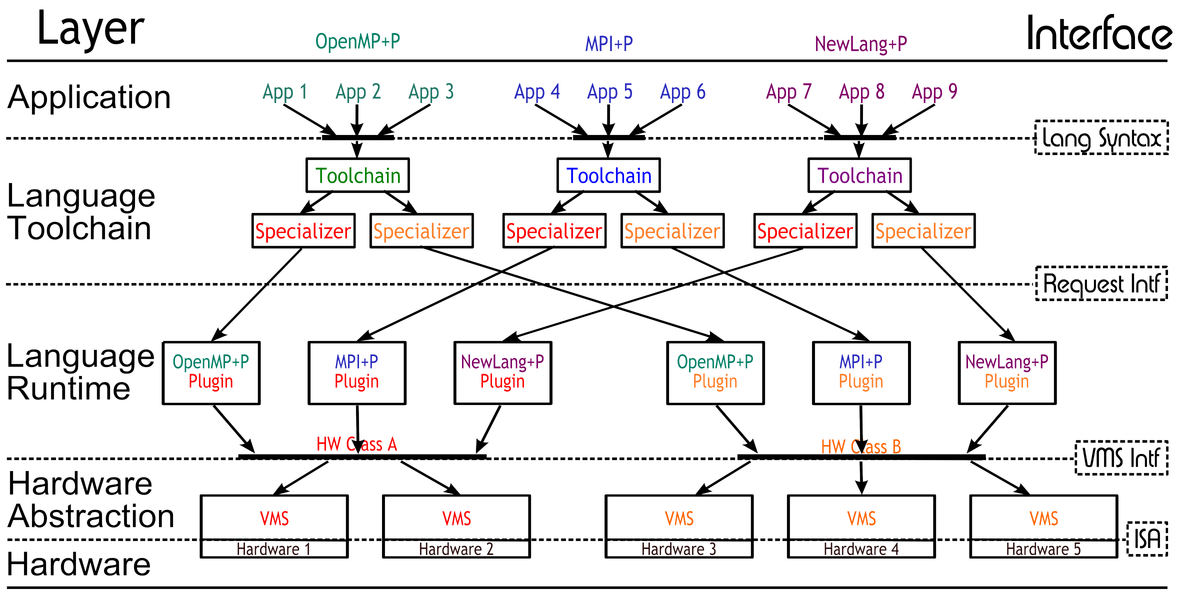

Fig 1: The layers and interfaces of the portability software stack. Applications at the top are modified once, to use the new portability constructs, then remain constant, while the stack adapts the source code to the hardware. The languages are modified by adding portability constructs, as indicated by the "+P" designator in the language name. The constructs enable the toolchain layer to extract application information and code-manipulators. It uses these to transform the application, then passes it to the runtime layer. Each runtime further uses the information to fine-tune code to hardware characteristics, and to schedule work onto the hardware resources. The abstraction hides low-level hardware details, allowing similar hardware platforms to reuse the same runtime. The abstractions are designed to allow the same top-level portability constructs to work across different memory-consistency models and hierarchies.

The research is ambitious, but the effort will be kept manageable by staging the effort, focusing first on the abstraction and runtime layers, then advancing to toolchain efforts near the end, once a solid foundation is established. This first-demonstration will aim for good performance, while establishing the path for toolchains to be independently added, as the route to ever higher performance. The nature of the portability constructs, which work together with the VMS interface, allow significant tuning to take place in the runtime and VMS itself, making this a reasonable approach. Toolchain techniques will be investigated late in the project, including a survey of existing high-performance toolchain and auto-tuning techniques. A report will be generated on ways to integrate those into the portability stack, to encourage the originating groups to perform the integration.

Selling Points of the Project

Such a software stack will be a radical departure from current research into portability. Researchers tend to focus on solutions they can provide within a single group, normally with only a single language or narrow family of hardware targets. The multi-entity eco-system approach of this proposal has not been previously investigated. Yet, it makes the most sense, and the new breakthroughs on the theoretical model, initial portability constructs, and hardware abstraction give it a serious chance of succeeding.

The project is ambitious, but tries to balance this by reducing risk by building on previous work. At the top, the portability constructs start with the foundation laid by DKU, and will be informed by The Holistic Model. Below that, the toolchain-layer employs the BLIS approach to fold-in advanced compiler techniques. It has previously demonstrated automation of toolchains that specialize single-source to multiple hardware targets, including a heterogeneous hierarchy of machines. Below that, the middle-layer is made up of runtimes implemented as VMS plugins. For the bottom-layer hardware abstraction, the proposal uses the Virtualized Master-Slave (VMS) abstraction recently published.

We believe this goal falls outside a standard project, and that the subject is considered too far-reaching to appear in a standard call, which suggests applying for the FET-Open call. We believe that despite its lofty goal that it is sufficiently grounded to expect that it is indeed possible. With the broad impact and dramatic rewards, we would be remiss if we failed to ask for funding to explore such a promising starting point.

1.1 Targeted breakthrough and its relevance towards a long-term vision

The project will deliver a working proof-of-concept portability software stack consisting of 6 languages augmented with portability constructs, a toolchain layer automated by BLIS, a middle-layer that connects each language to each hardware class, and VMS implemented on each hardware target. In addition, two hardware platforms will be delivered: a distributed memory platform reusing existing work and an implementation of a low power GPU currently being developed in another EU project. A benchmark suite will be delivered to test the stack, covering both common code patterns and particularly difficult ones.

The languages delivered will be a Java based domain-specific language for semantic analysis, OpenMP+P (where "+P" represents addition of portability constructs), MPI+P, Skeleton+P, SSR+P, and ResearchLang+P. Each language will be delivered with toolchains that run within BLIS, and connect to the VMS plugins. A plugin will be delivered for each pairing of language with hardware class, which will, in total, connect all languages in the project to all hardware platforms, allowing all benchmarks to run on all hardware.

A diverse set of benchmarks will be chosen, according to an analysis using the theoretical computation model, and the expertise of the project partners. Most benchmarks will already have been implemented in a base language, then during the project the portability constructs are added. Each benchmark will be demonstrated running on all hardware platforms, without further change to the source code, and with good performance.

The hardware targets will include both current and future-looking platforms, plus all connected together as a *heterogeneous cluster*. The "current" platforms will be coherent shared-memory multi-core in several variations, GPGPU co-processors, and the Cell Broadband Engine. Future-looking platforms will include the Tilera 64 core chip, the Intel SCC chip, a distributed-memory many-core, and a heterogeneous distributed system. Lastly, all those will be combined into a hybrid machine tied together via ethernet, as a separate hardware platform. This demonstrates hierarchical heterogeneous portability using the stack.

1.2 Novelty and foundational character

The novelty is: the concept of providing an industry-wide, shared, portability infrastructure that enables practical cooperation of many partners who each contribute a piece; the combination of a toolchain management tool (BLIS) with a hardware-abstraction (VMS) where both are fed tuning information about the application by co-designed portability constructs; the use and enhancement of the new theoretical model of parallel computation (The Holistic Model) to develop portability constructs; the new set of portability constructs; and the application of the new hardware abstraction (VMS) as the organizing principle of a portability software stack.

Comparing to other projects for portable applications, they fall into one of three general approaches:

- Portability through a single language + toolchain: Sequioa, OpenCL, OpenMP, Galois, StarSs
- Portability through libraries plus runtime: SETJits, TBB
- Portability through static tools in combination with runtime: CARP, CnC

These others have a narrow focus, with either a narrow range of applications, or narrow range of hardware targets, or both. They focus on only one part of the portability problem, with a stand-alone offering that doesn't interface with other approaches. They don't modularize the specialization process, nor span multiple points in an application's life-line. And they don't offer an organizing principle for multi-entity cooperation (eco-system). None offer all three of: a very wide variety of applications supported, a very wide variety of languages/programming models offered, and very diverse array of hardware targets, all at the same time. Instead, each of these other approaches can be implemented in terms of Pstack, and would benefit greatly from it.

Specifically, the two main elements other approaches lack are a hardware abstraction with VMS's benefits, and an equivalent general set of portability constructs, which supply specialization information. Some do have a form of abstraction, but limited. Most have constructs intended to aid portability, but none have been guided by a theoretical model with the power of the Holistic Model, and so their constructs fall short of being general.

Most importantly, no work has been published or funded on *general* infrastructure whose purpose is to act as portability glue, organizing multiple languages onto multiple hardware, with pieces provided by multiple entities. The existing and known proposed portability work takes the approach of providing a stand-alone system, for one language, library, or runtime system. This severely hinders reuse of the work. This vertical approach also hinders an eco-system of cooperating players, because it was designed as a single system, without general interfaces for interaction between separate pieces.

Confusingly, early work claimed to be portability infrastructure, such as Globus, PVM, Tempest, and Grid. The actual purpose of these approaches was to simplify the hand-tuning of parallel applications. Although they act as a type of hardware abstraction, they only cover communication, without the kind of organizing principle needed to be the base of a portability software stack. The hardware abstraction at the base must be confluent with the portability constructs at the top, and be designed in accordance with basic portability principles, as opposed to those referenced, which just provide convenient hardware patterns.

1.3 S/T methodology

Background on Starting Point Technologies

Portability constructs: The starting point for portability constructs is DKU, published in 2010. It provides hierarchical division of work, and cleanly identifies units of work (tasks). They were demonstrated with automated specialization, using BLIS, which inserted hardware-specific runtimes. Both multi-core and a heterogeneous distributed cluster were targeted, achieving near ideal speedup, without modification of the source.

Additionally, construct work will start with constructs suggested by analyses performed with the Holistic Model of Parallel Computation. It is a new theoretical model that is currently being researched. It indicates that a canonical and finite set of information both exists and also fulfils the requirements for portability. It also gives evidence that toolchains or runtimes alone, without such constructs to provide information, are unable to achieve wide portability. It demonstrates which application-patterns must be captured in the language and used in scheduling decisions, to gain portable high performance.

Toolchain layer: In order to gain the highest performance, advanced toolchain techniques must be incorporated, such as profile-driven optimization, auto-tuning, multi-versioned kernels, and hardware-targeted data-layout transformations. BLIS is the starting point for this layer, having demonstrated running hardware-specific toolchains and managing the multiple executables generated. It was published in 2010 with the DKU constructs.

Middle-layer: The middle-layer starting point is the VMS plugins, which have already been demonstrated connecting 4 languages to 3 hardware platforms.

Base-layer: The base layer starts with the VMS abstraction, published in 2011.

Full Stack: The starting point is the eco-system concept published in 2011, based on the VMS abstraction.

Addressing the gap between starting point technologies and the deliverables

Portability Constructs: DKU constructs exist for platforms with, coherent memory multi-core, distributed memory, and heterogeneous systems. They enable general, hierarchical division of tasks, when the tasks don't communicate. These will be extended to add communication among tasks, and constructs for prediction of data footprint, execution time, and manipulation of data layout. The extensions will be guided with theoretical results from the Holistic Model to ensure applicability to future architectures.

Toolchain-layer: Will start with existing BLIS automation, and be extended to cover the six languages in the project. Simple tools will be added to make BLIS more robust and convenient. It specialize both by passing portability construct information to the runtimes, and by invoking toolchains, which use portability construct information to manipulate code.

Middle-layer: Will use existing plugins for SSR, Cilk, and HWSim as models to guide plugins for the project languages. Each hardware target will have a plugin made for it, including distributed-systems, GPUs, and the Java Virtual Machine (JVM), which has support for the portability constructs defined during the project. In the later stages of the project, once all are working, will explore performance enhancements to the runtimes.

Base-layer: Start with existing VMS for coherent memory multi-core hardware. Define additional VMS interfaces to cover distributed-systems, GPUs, the JVM, and hierarchies. In the later stages of the project, explore performance-enhancing extensions to the interface, such as dialogue between plugin and VMS during scheduling decisions and task-tuning decisions.