Programmability Issues

Vivek Sarkar (Rice U.), Jesus Labarta (UPC), Mitsuhisa Sato (U. of Tsukuba),
Barbara Chapman (U. of Houston)

Programming models are central to our effort to address the exascale challenge. They are the key interface that will allow the separation of the programmers' concerns from those of system designers, potentially at different levels of granularity. Any such model must meet the extensive needs of application developers and be supported by the entire software stack. The programming and execution model interfaces are key to allowing programmers to focus on their algorithms while providing the mechanisms that will enable the compilers and run times to infer the information they need to optimize, automatically and dynamically, the use of system resources (cores, memory, bandwidth, power). Considerable research is needed to define and implement the programming and execution models for such systems. Whereas evolutionary approaches may best support the migration of existing application software, revolutionary models may be best suited to providing extreme-scale performance for new applications on emerging architectures. Both approaches should be explored.

Desirable properties of exascale programming models include the following:

- They should provide highest levels of **performance**. Most HPC programs are written for performance. Moreover, exascale programming languages should be **performance-aware**: they should provide an adequate abstraction of high performance parallel hardware platforms to enable the exploitation of their features, and some means to tune performance. The failure of automatically parallelizing compilers and HPF was caused not only by technical immaturity but also by a lack of an interface in the programming language for performance improvement. When the programmer finds a performance bug, he or she should have some mean to improve performance by modifying the program. The model should provide the necessary interfaces to allow tools (especially performance tools) to obtain information on the application's execution behavior.
- **Expressivity** is a key requirement. Exascale programming languages should provide a model and an interface to express the parallelism in programs. In functional programming languages and "old" dataflow languages, parallelism is implicit since the model of computation itself exploits the parallelism. In imperative languages, new constructs and mechanisms should be introduced to express the parallelism. From the application points of view, task parallelism must be able to support coupled multi-physics simulations at several levels for exascale systems. Applications will need to express massive amounts of potentially fine-grain parallelism, of asynchrony and locality. Dynamic application behavior will need to be supported. It should be possible to express hierarchical parallelism within the application. Latency hiding needs to be facilitated.
- They should enable **composability**. Composability is essential to support productive programming on exascale systems. Libraries and object-oriented approach help accomplish this in conventional sequential programming, but they

don't always work in parallel programming. For example, it is difficult to use parallel libraries with current OpenMP. Parallel object-oriented programming is sometimes useful, but has some problems.

- They should support **fault tolerance** and **error handling**. Fault tolerance is one of the most difficult issues faced on exascale systems. If faults are exposed to programmers, then some programming language support will be required to handle them. It must moreover be possible for an application to respond to faults and program errors gracefully rather than simply crashing.
- They need to support **massively parallel I/O**. An abstraction of I/O, including the file system, may help programmers handle the huge amounts of data that will have to be read and written.

Approaches to programming exascale systems should take the following into account:

- There is a need to provide a **smooth transition path** from existing practices and codes to future approaches. **Programming environments** will be needed that support this transition, as well as all phases of application development and tuning on exascale architectures under new and enhanced programming models.
- Approaches should provide **portability (functional and performance)** across platforms such that the porting effort can be amortized over the foreseeable variation of systems to appear from now till the exaflop era and beyond.
- **Incremental** parallelization/tuning of applications is a desirable property closely related to the above two issues.
- Initial approaches should address the **device, node and system level** programming. Proposals for hybrid programming should ensure clean interaction between the different levels and ensure that the synchronization semantics and scheduling decisions at one level do not imply restrictions on other levels.

Topics for detailed study include:

- **Address space structure**. Identify abstract levels of a structure that is simple enough for use by a programmer to express objects/ideas yet allows the run time flexibility regarding its mapping to the potentially varied physical structure.
- Flexible **work generation** (parallelism/task specification) **and synchronization structures** beyond pure fork-join approaches in order to support flexible parallelism and **high levels of asynchrony**. Ideas from data flow or functional programming may be revisited and smoothly integrated into current practices.
- **Latency tolerance**, being able to specify required data accesses with large lookaheads such that implementations (compiler or run time) can anticipate the required data transfers and schedule them appropriately.
- The issue of **hierarchy and heterogeneity**, providing mechanisms for **modular** designs with interchangeable implementations of tasks.
- **Separation of functionality and performance**, providing mechanisms for the programmer to provide hints that may help satisfy performance or power requirements, but are not required to provide functionality of the algorithms.

- **Malleability,** the ability of applications to dynamically adapt to the available resources which may vary during a job run. Programming and execution models should support/promote malleable programming practices by separating (virtualizing) the algorithmic structure of a program from the resources where it is executed.
- **Error handling and fault tolerance.** Providing the appropriate hooks for resilient applications.
- **Application development environments** that facilitate the migration of current codes and/or the development of new ones from scratch.

The evolutionary path aims to adapt existing programming models to needs of exascale computing, and facilitate task of creating and tuning potentially hybrid application codes. This could include work to enhance MPI, OpenMP, CUDA/OpenCL or other approaches to programming accelerators and SIMD units, as well as work to improve their interoperability. It might also include more effort to deploy the PGAS languages and ensure that they may interoperate with other programming interfaces. A revolutionary path might be based upon HPCS languages or might be a completely new path. It might be worthwhile to revisit old parallel programming models and languages to obtain new insights from the past, as is being done in the architecture community. Functional programming models used to programming the dataflow machines, such as Id, SISAL, ... could be interesting to evaluate. HPF was a great effort to develop a standard parallel programming language and is also worthy of re-examination. It is important to take their experience of failure into account for better future developments.

In such an open field, it is advisable to pursue a few alternatives and ensure there is sufficient sharing of experiences as well as **comparative studies** between them. These should be in terms of complexity/readability of the code and programming effort as well as performance (both actual measurements on common platforms as well as predictions for different potential targets). Although these types of studies are often difficult to perform, special efforts should be devoted to that. **Common sets of algorithms** should be used for evaluation by all the proposed models.

Finally, we should promote efforts to develop standard APIs between several levels and components of existing software in the IESP community. For programmers and end-users, candidates for standardization will include:

- PGAS languages (UPC and CAF, …)
- Global views models such as Chapel and HPF

For the system developer, the candidates are:

- One-sided communication APIs
- Fault tolerant model and APIs
- API for I/O on massively parallel system
- API for accelerators
- Performance profile API and data format such as OTF
- API for thread scheduler

The standard development effort is a key to "evaluation" which develops the community. It will be the basis for the next "revolution" of rich diversity for exascale computing.