Processor Evolution: What to Prepare Application Codes For?

Orap Mini-Workshop
François Bodin, Henri Calandra, Alain Refloc’h
Migrating application codes may a lot look like

*In an old house, when replacing one wall tile you may end with getting the wall tore down.*

*(Thomas Guignon)*
Mini-Workshop Attendees

- Alimi Jean-Michel, Observatoire de Paris
- Colin de Verdière Guillaume, CEA DAM
- Courteille François, Nvidia
- Dinh Quang, Dassault Aviation
- Dolbeau Romain, CAPS entreprise
- Fournier Yvan, EDF
- Grigori Laura, Inria Rocquencourt
- Guignon Thomas, IFPEN
- Kern Michel, Inria (External contributor)
- Meurdesoif Yann, CEA
- Namyst Raymond, Inria Bordeaux
- Petiton Serge, Université de Lille 1, Sciences et Technologies
- Ricoux Philippe, Total
- Thierry Philippe, Intel
Context

• The Power Wall has led to the design of new parallel multi-core and many-core architectures
  – Achieving scaling is the challenge
  – Application codes (will) have to exhibit massive parallelism at the node level

• The evolution of processors is very strongly and deeply impacting
  – Code development
  – Maintenance practice
  – Numerical methods

• Bill Harrod from the Department of Energy (DOE): "Uncertainty threatens future of computing, the world has changed; technology is changing at a dramatic rate; The IT marketplace is also changing dramatically, PC sales have flattened, Handhelds dominate growth, HPC vendor uncertainty; Data volume and variety explosion"
Code Main Matters

• Code validation
• Surviving at least 4 generations of (very different) machines
• Homogeneous programming is at an end
• Sophisticated runtime techniques required
• Data structure organization
• IO sub-systems / data management
• Application development process

• Very likely that in some cases achieving scalability will require going back to basics,
  – i.e. physics, instead of forcefully trying to adapt legacy codes
Warning

It should be noted that the first factor driving an application development is its ecosystem and its deployment constraints
[R1] Validation

• Implemented as a continuous process as recommended
  – Validation must ensure reproducibility of the result \textit{without imposing a bit to bit comparison of the result}
  – For instance implemented by frameworks such as Hudson

• This issue is a primary consideration when migrating or designing a code
[R2] Data Locality

• We are now very very far away from the 24 Bytes per Flop of the CRAY1
  – Complex memory hierarchies

• Internal data structures must be designed in order to facilitate adaptation to the architecture of the computer
  – E.g. array of structures versus structure of arrays

• Serializable data structures are preferable since at one point they may have to migrate to a different storage unit
  – E.g. use of accelerators, saving data structures in NVM, …

• C++ templates can be one of the implementation technique
  – But this topic is controversial due to the complexity in code maintenance
[R3] Scalability

• Collective operations (e.g. reduction) do not scale well enough
  – When possible they should be avoided

• Neighbor data exchanges are to be preferred

• Choosing the solver is crucial and must be carefully considered
[R4] Programming Languages

• Programming language choice
  – Must first be performed according to developers background
  – Fortran can still be a good choice
  – Mixing languages (e.g. Fortran and C++) is an option to cover functional needs
  – Code architectures more important than the language

• Engineering consideration should be the one guiding the choices here
  – E.g. portability, persistence, efficiency, libraries, …

• Combining languages in a given application requires careful thinking
  – e.g. Libraries, build complexity, compiler adherence, …
[R5] Domain Specific Approaches

- May provide a level of abstraction appealing to scientists
  - Complexity of standard programming is likely to distract scientists from their scientific discovery goals

- Embedded Domain Specific Language in a general purpose host language (e.g. Fortran)
  - Embedded DSL provides extra semantic information and/or code generation strategy
  - E.g. OpenACC

- Embedded DSL in some cases are not enough to provide a high level algorithmic abstraction
  - The data model is the one of the host language and does not carry the high level semantic required (e.g. an array is not a matrix)
[R6] Development Process and Infrastructure

• The cost of a bug increases with its late discovery

• This is not new, but implementing massive parallel execution make code development extremely complex
  – Pro-active techniques needed
  – Applications embed bug detection, traces generation, ...

• The development process must integrate software engineering best practices such as
  – Control using automatic tools (e.g. continuous integration)
  – Configuration management
  – Performance measurement integrated inside the code
  – Unitary test
  – Non-regression testing
  – ...
[R7] Fault Tolerance

• Fault tolerance as envisioned for Exascale systems is not presently an important consideration

• But the increase in the cost of IO pushes
  – To consider strategies that minimizes the volume of data needed for implementing a checkpoint restart technique

• Applicative specific techniques are the most promising
  – It is believed that they are the only one that may scale in the long run
  – Does not mean that system/runtime support is not necessary, on the contrary
[R8] I/O

• Data management and I/O performance will strongly influence the design of applications
  – Requires a global view on the data life cycle

• Designing the applications requires finding tradeoffs between
  – In-situ vs. ex-situ processing
  – Selecting data format
  – Access policy
  – Data relocation
  – Format changes, etc.
Pre and Post Processing Integration

• Pre and post processing may frequently become the application bottlenecks
  – In-situ analysis to decrease the volume of data to transfer out of the machine

• The scientific discovery process must be particularly well understood to design a long-term solution
[R10] Code Architecture

• The code architecture
  – Must be mastered and **enforced**
  – Flaws in the architecture and **its dissolution** over time have expensive consequences

• Goal of the software architecture
  – Modular structure
  – Help re-writing, modifying the fast changing part of the code
  – Help mixing technologies and competencies of the application development teams
  – Provide external APIs to improve usability
  – **Internal API** to enforced structured development and best practices
  – Allow the development of a machine specific with more lasting generic versions

• Allow to plug-and-play solvers so they can be chosen according to the execution platform
  – This consideration is not only related to code architecture, it must be consistent with numerical schemes
[R11] Coding Rules

• They are a set of rules to guide developers
  – Specific to each application and ecosystem
  – Their use aims at preserving code efficiency, maintenance and evolution capabilities
  – Specifies preferred code writing style (e.g. naming conventions, …)

• For the HPC domain
  – Typically the rules would include code structures that favors vectorization, data locality, efficient parallel execution, etc.
[R12] Libraries

• The use of external libraries is recommended but must be chosen with care

• Use as much as possible native libraries (or open source ones)
  – Very well optimized

• Don’t use old algorithms that have been designed for sequential execution with no memory hierarchy
  – E.g. 1986 Numerical Recipes

• Choose long lasting libraries or easily replaceable ones to limit adherence to a given platform
[R13] Runtimes

• Runtimes provide intermediate resource management services not directly provided by the operating systems
  – E.g. StarPU, X-Kaapi, MPC, …
  – Adapt to run / machine configurations

• Growing number of threads
  – Hierarchical techniques needed to avoid high thread management overhead
  – However, this evolution may lead to less accurate scheduling with more workload unbalancing
[R14] Debugging

• Debugging is a sensitive issue difficult to integrate from the start to project
  – Usually a post-mortem technique however ease of debugging is usually a consequence of the development methodology
  – Integration in the application of the right observation tools (e.g. tracing capabilities, visualization of code data structure)

• The use of tools such as Valgrind etc. is recommended even if the code execution incurs a large slowdown (x10)
  – Detection of memory leaks, …
Vector and Data Parallelism

- Vector capabilities contribute to a large part of the performance of current CPU
  - E.g. 80% on an Intel Xeon Phi
  - Data parallelism to use accelerators (i.e. SIMT) is important

- The use of such parallelism cannot be left to compilers alone
  - Code writing rules can greatly help to achieve automatic vectorization by compilers
  - Use of vector intrinsics is not advised
[16] Technological Watch

• Anticipating hardware evolution has a cost
  – Especially since current uncertainty generates multiple tracks

• Avoid the temptation of everything that’s new but do not procrastinate
  – Technological watch is key to make the right decisions
  – Unfortunately it is always easier to justify new user oriented features rather than a technology evolution
[R17] Gathering Competencies

• International and national initiatives are such as ”Maison de la Simulation”

• And other competence centers should be used as much as possible
Conclusions and Perspectives

• Processor evolution toward massive parallelism questions the codes evolution

• Migrating or a designing a new code for the decade to come
  – An extremely challenging tasks
  – Will requires to make multiple algorithmmic and technological choices