The Application Perspective - Seeking Productivity *and* Performance -

David Barkai

Abstract—In this note we propose two projects: (1) Creating a hierarchical programming model from current models, and (2) Extracting application primitives from the "13 dwarfs". The first topic addresses the need for a unified and manageable framework for very large scale concurrent execution. This is the productivity part - less complexity will drive better mapping of algorithms to architecture; which will also contributes to better performance. The second topic focuses mostly on the processor and the node with the aim of laying the groundwork for software and silicon optimized kernels. While it is understood that applications primitives are outside the scope of IESP, the motivation for introducing it here is that it is a companion issue and that increasing the efficiency of each processor provides high return for science - at all levels of system size.

Index Terms—programming model, manycore, multicore, clusters, applications, HPC, application primitives

1 Setting the Stage

WHILE the "moonshot" goal in front of us is preparing for systems with peak exaflops, we must not lose sight of the fact the all this is done so science can accomplish more through computations. To this end it is best to take the application perspective, and look for ways to help the scientist or application developer get more out of a given very large system. In this note we suggest to take on the two "P's" - Productivity and Performance (leaving out the third "P" - for Power; though with higher efficiency, another way of saying 'performance', a given computation gets done as fast on a smaller system - and consumes less power).

There is a fortunate synergy now between the need to address programmability on petascale and exascale systems and these three drivers that are now central to the future of high-performance computing (HPC):

- Almost universal adoption of clusters as a 'standard' architecture.
- Manycore processor chips in our future.
- Emergence of heterogeneous computing on or near the processor chip.

The synergy derives from the fact that a standard model that fits the above also suggests a hierarchical view of the system; a view that offers hope for a more manageable approach to dealing with the very high level of concurrency, of order $10^7 - 10^8$, required for a full use of an exascale system in circa 2018.

The ideas presented here are also influenced by the work commonly recognized now as the "view from Berkeley" [1], both with regard to extracting a cohesive programming model and in providing a framework for addressing performance through a set of application primitives.

• *David Barkai is with Intel Corporation, HPC division of the Digital Enterprise Group, Hillsboro, Oregon 97124*
email: david.barkai@intel.com

Revised April 17, 2009

2 The case for a consistent and layered model

THE advent of multicore in all of our platforms presents an opportunity, and motivation, to take a fresh look at our programming model. Looking ahead we have a 3-layer architecture from the user's perspective: the chip - with multiple cores, caches, and, potentially, attached accelerators; the node - multiple processor chips sharing memory; and the system of nodes governed by its distributed memory.

Today we have, essentially, two approaches to parallelizing applications: one for shared-memory systems (OpenMP, for example), the other for distributed memory systems (where MPI is the most popular tool). Multicore on the chip adds another layer, but also impacts the application's choice of algorithm in that the way to increase the performance from one generation to the next is only through finer parallelism as the number of cores on the chip increases, whence preference for algorithms that scale better.

The time is right for a community-wide initiative that will include the application writers, the software providers, and the hardware vendors, with the goal to define a programming model that will be integrated, consistent, and seamless across the three architectural layers, scalable from the node to the petascale and beyond, and allow for application driven expression of concurrency that will extend to dataflow and multitasking, as well as parallel computations.

The discussion is framed with a strong emphasis on the application's perspective, as we believe this will lead the application designer taking more responsibility to map the implementation to the system, resulting in higher productivity, and allowing the system and tools

software to do a better job in mapping the hardware. In short, we will be closer to a desired balanced between scientists' productivity and a reasonable performance relative to theoretical peak.

The desired programming model should comprehend partitioning details at a finer level than just assigning processes and threads to cores. It should allow visibility to on-chip or socket-attached interactions. The convergence to a single architecture makes it a good time for the HPC community to take a fresh look at the programming model when designing new implementations of numerical and data-intensive applications. A typical cluster is made up of high-volume off-the-shelf components for processors, memory, boards, interconnect, storage, file systems, etc. This is not central to the discussion here, but for the fact that it provides greater motivation for a 'standard' programming model.

There are two other challenges that large system users have been struggling with and that have not been resolved yet:

- Scaling of applications effectively as they increase in complexity, use higher resolution with larger datasets, and run on an ever-increasing number of processors and cores is, so far, a rare occurrence.
- Productivity - both in terms of the programmer's time, and in terms of output from the compute system is still a panacea.

A holistic, integrated and consistent programming model, constructed and presented from the application writer's perspective might help us move forward with regard to the two challenges above.

3 What might the model look like

This is an abbreviated version of a longer discussion, and, therefore, statements may seem too blunt. My apologies to the reader.

Discussions of programming models almost always turns to languages for expressing parallelism and tools to support parallel programming. We are skeptical that any new language will gain a wide acceptance, and believe the best course of action is to build on the tools that current applications are most invested in. That would be MPI used by Fortran and C/C++.

The need for hierarchical model, to better map the application to the underlying architecture and for better manageability of concurrency, led to various experimentations in "hybrid" implementations - combining MPI with OpenMP or other shared memory schemes. These met with varying degrees of success (see [2], [3], [4], [5]). It is stipulated that the use of OpenMP would not have been required if we had 'layered-MPI' to define such a hierarchy to help manage the decomposition of the application.

A layered model is also necessary in order to have any hope of managing the level of concurrency that will be in the 100's of millions in the future exascale system.

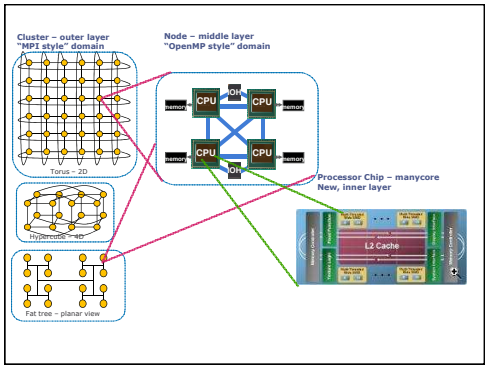

Fig. 1. Hierarchical view of the model

That said, we propose considering a hierarchical model (conceptually drawn in Figure 1), that can be built upon the following guiding principles:

- **Do no harm.** The expression of parallelism according to a new or modified model should not invalidate the huge investment put into existing codes. This principle forces us to look at extensions to, or evolution of, MPI. Given the much lesser use of shared memory models it seems more natural to build an integrated model from MPI.
- **Balance productivity and performance** - as expressed by the Berkeley team [1]. For productivity the model is to present the application view, be expressed in terms comprehended by a high level language and in terms relevant to the scientist and engineer. Let the compiler system (see below) deal with the details - which also vary from one system to another. And for performance's sake give the programmer the tools to associate computations with data, and to specify flow and communication patterns.
- **The application writer knows best** about how the application works and there will be no automatic parallelization any time soon. This has two important implications: (1) The programming model has to have 'hooks' into all the architectural layers and components. (2) The application writer can do a better job partitioning the data and computation than the compiler or middleware. Let the tools be there to offer the help the system software will need.
- **Integrated, layered model.** It would have a set of one or more MPI ranks per node, each may be split into a set of MPI processes, preferably optimized for shared memory, and allowing for each of those to further split into a set of 'fibers' to be executed on the same socket. It is this last, lowest, layer that can be used to interact with special-function units or an attached accelerator.
- **Extensible.** For large systems it may well be useful to allow for some kind of system-level partitioning, in addition to the layers described above. This will divide the highest level into regions of MPI environments working in tandem.

- **Coherency.** The implementation of the model has to be adaptable to various degrees and regimes of coherency. These may be dictated by the system in use, or be a choice to be managed by the user.
- **Robust runtime compiler system.** When the system is a cluster, compilers and runtime libraries that are local-node-aware are not optimum. A complement to any program such as the one outlined here has to drive a considerably more runtime-robust compiler system. A system that will pick up the allocated resources (the cluster or a part of it) and execute to it. This may include, for example, MPI operations that might be presented as directives or pragmas. This will allow skipping them when the job runs within a single node.

The benefits of the vision expressed above are fairly obvious: Common and comprehensive basis for applications design. Potential, and expected, higher performance due to the integration of the support for distributed, shared, and on-chip operations.

The next two sections deal with aspects of performance at the node level, likely to be dealt with in other forums.

4 Application primitives - key to performance

THE topic raised here is not specific to exascale systems, but very relevant to scaling and delivered performance for real applications. We will have processor chips with billions of transistors, and looking out towards the 2018 timeframe we can ask how best to use them.

The model we propose to follow is that of signal and image processing. Compact application primitives were identified such that great performance improvement was achieved with a combination of special function silicon and libraries. Despite the greater complexity, diversity, and dependence on bandwidth and latency of data access, can such methods not be applied to HPC?

At the very least, this is worth investigating. A starting point can be the first seven of the "13 dwarfs" taxonomy defined in the "View from Berkeley" [1], as they are the ones corresponding to numerical simulations. To remind the reader, these seven are dense and sparse linear algebra, spectral and N-body methods, structured and unstructured grids, and Monte Carlo. The problem is that the broad brush definition of the application categories is not actionable as it stands. To be able to act on the taxonomy it would be most useful to identify:

1) The algorithms that are the most important (sparse or N-body, for example, may employ a number of different algorithms and methods).
2) The relative weight of the category/algorithm within the general (high end?) scientific workload.

Setting aside the tasks above, for now, we can assess the problem with another source. The NAS Parallel Benchmarks (NPB) [6] are composed of several common computational procedures. They are sure to feature in several of 13-dwarfs categories. A nice feature of NPB is that it reports the MOPS (millions of operations per second) score, which for the numerical tests we discuss here is, essentially, the rate of floating point calculations. This allows us to measure the "efficiency" of the benchmarks compared to the ideal case where all data access can be hidden or overlapped. To make a point five are chosen: Multigrid (MG), Conjugate Gradient (CG), FT (FFT), LU (Lower-Upper decomposition), and BT (Block Tridiagonal). These were run, using the NPB 3.3 version, on 8 cores (a 2-socket node) of Intel's recently launched microprocessor, which has far superior memory bandwidth compared to previous generations of x86 architecture. Even with these very competitive times the calculated efficiency, listed below, ranges from just over 4% to under 20%, averaging less than 12%.

MG	CG	FT	LU	BT
11.1%	4.5%	11.8%	12.5%	19.3%

These findings are not a great revelation to the HPC community, but it gives us an idea of where to start looking for improvements. We must not overlook the fact that the performance efficiencies given above, due to data access and communication between processes, are prior to any effects of the network. These measurement were done on a single (shared-memory) node.

5 Can we do better? - sample ideas

OF course, the easy answer is to say "increase memory bandwidth and cache size and lower latency", when the code is data access bound (as is true for most codes), and "give us more floating point functional units" when the code is compute-bound, as is for dense matrix operations, for example. The latter is relatively easy, but not particularly impactful. The former is hard. We suggest, instead, to go back to the image/signal/graphics processing analogy and look for ways to optimize kernels, or what we might call "numerical operators" - though we don't forget the real challenge is in data access. Here are some partial, tentative, and somewhat random ideas.

Consider the computations derived from a stencil representation after discretization. Figure 2 shows a simple 6-point stencil.

To compute a given grid point we need a set of values which are not consecutive in memory. A cache line is loaded for one or two useful values. But if we computed along the index that is stored consecutively (say, the "i" index) then all the values brought in with the cacheline will be needed for computing the following grid points. The programmer or the compiler can direct the order of stepping through the grid. But there is no guarantee that the needed cachelines will not be replaced. Will it make sense to define a "stencil operator" as a macro instruction, allowing for parametrized number of stencil

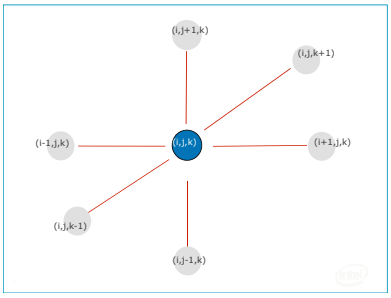

Fig. 2. A simple 3D stencil

points, and setting aside a buffer where these loaded 'vectors' can be kept? - this will result with a streaming of, say, 8 or 16 sets of computations before the buffer will have to be re-loaded. This is not necessarily practical, but illustrates how software and hardware can collaborate to provide higher performance for a useful and common sequence of operations.

Conjugate Gradient scores very low on efficiency mostly due to repeated passes through all the data with relatively small amount of computations done at each such pass. Here there might be a simple programming/algorithmic remedy. "Unroll" each iteration to perform 2 or 4 iterations through simple substitutions. We have done so in the past when the array did not fit in memory. We can do it now because the processors got so much faster. It is expected that this procedure will increase the efficiency by 2-4 times. Can a compiler be taught to unroll a loop in this iterative manner?

FFT performance is dominated by long distance communications, clearly noticeable even within a node (see FT above). The communication patterns are well structured and deterministic, though. Are there look-ahead ideas that, with some specially reserved buffer space, can reduce the shuffling of data, thus dramatically improve performance?

These are just sample random thoughts. A more structured approach is needed to provide cost-benefit analysis of where to place our efforts. An example of drilling in from the applications, to the common kernels, and to potential hardware support is the question of how best to approach the need of Gather/Scatter for HPC in a cache-based architecture.

Clearly, this short note does not do justice to the topic. More work is needed.

6 Conclusion: Proposed Actions

The desired outcome of this discussion note is to get the community at large to engage in creating a cluster-based holistic, integrated, backward compatible, application-based programming model. Whether the ideas and directions suggested here are followed is far less important than the getting together of all stakeholders to address the need for such "standard" programming model.

This is a call to the community - applications writers, software providers, and hardware vendors - to come together to define and implement a 3-layer (cluster, node, chip) programming model that:

- Extend MPI to allow layered, hierarchical, framework to express parallelism on a very large cluster. A single specification that defines the convention for the integrated model, and possibly adds directives that, for example, allow compilers to generate the calls to MPI routines.
- Adds mechanism for expressing interactions among cores within the processor chip. Allow extensibility to attached accelerators (OpenCL?).

The goals above are broad and directional in nature. A possible start can be to test the approach outlined here using a (crude?) prototype of the model on a couple of simple applications that span a cluster.

Just as important is setting goals for achieving higher performance out of each of the nodes that make up the total system. This, too, requires the HPC community to work with the Industry to -

- First, define and prioritize encapsulated computational kernels.
- Second, work jointly to come up with creative ways to combine software techniques, hardware capabilities, and architectural features that will enable a significantly higher efficiency of scientific codes.

The ideas presented here are far from even a proof of concept. Their intent is to encourage the community to create a more complete and more consistent framework for coding on our future HPC systems.

Acknowledgment

The author wishes to acknowledge and thank several Intel Corporation colleagues for helpful discussions and constructive feedback: Henry Gabb, Tim Mattson, Andrey Naraikin, and Rob van der Wijngaart.

References

[1] K. Asanovic et. al., *The Landscape of Parallel Computing Research: A View from Berkeley*, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] Y. He, C. Ding, "Hybrid OpenMP and MPI Programming and Tuning", Lawrence Berkeley National Laboratory, 2004.

[3] M. Su, I. El-Kady, D. A. Bader, S-Y. Lin, "A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelism", University of New Mexico and Sandia National Laboratory, 2004. http://ieeexplore.ieee.org/ielx5/9250/29349/01327945.pdf

[4] E. Lusk, A. Chan, "Early Experiments with the OpenMP/MPI Hybrid Programming Model", Argonne National Laboratory and University of Chicago, 2006.

[5] H. Gabb, "Hybrid Parallelism: where's the benefit?", LCI Conference on High Performance Clustered Computing, 2008. [contact henry.gabb@intel.com]

[6] NAS Parallel Benchmarks: http://www.nas.nasa.gov/Resources/Software/npb.html