Find Regularity in Irregularities

Hiroshi Nakashima
(Kyoto University)

Irregularity in Big-Data Computing
Among various technical issues in extreme-scale big-data computing, here I am pointing out the necessity of finding out some good *regularity* from *irregular* computation, data-structures and interactions between them. Though big-data computing cannot be categorized into a particular type of computation, we have to expect that its significant portion has irregularities which we have not seen in typical HPC programs; such as a set of dynamically created small data pieces for lists, trees, graphs, etc. rather than (semi-)statically allocated arrays; random and indirect accesses to those data pieces instead of straight-line direct accesses to arrays; and open-bound *while*-loops (or recursions sometimes) with many conditionals contrastive to simple (nested) *for*-loops of fixed bounds.

High-Performance Processors Need Regularity
These irregularities are not friendly to current high-performance architecture especially that of cores of manycore processors whose source of high-performance is in a wide SIMD mechanism supported by a wide memory access path up to 64-byte now but 128-byte or wider in near future. You might claim that such concerns about architectural friendliness to processors of high floating-point performance is irrelevant to integer-dominant big-data computing, but we have to remember such processors are the most efficient means of *any* kind of high-throughput computation. The other claim might be that we should design big-data oriented processors with, e.g., many but simpler integer-oriented cores, but even such processors should rely on wide memory access paths to cope with the memory wall which we must expect at least as high and thick as we are seeing in typical HPC programs.

List in General but Array in Practice
A way to fit such a big-data program onto the mechanisms of a high-performance processor is to find some good regularity in its execution and to present the regularity to the mechanisms by a program transformation. This idea is based on the expectation that, while the program may have irregularity *in general*, the irregularity is not necessary to be *in practice* in the majority of its executions or in the major part of each execution. For example, the set of data pieces which the program repeatedly scans may have to be represented as a linear list because it must be dynamically created, but once the list is created it may be fixed throughout the program execution to allow us to convert the list into an array (or a set of them) for *cdr coding*. Even in the case that the list can grow or shrink during an execution, we may stick to the array representation with some margin in its size if additions or deletions of its elements are infrequent and/or the amount of them is small. Moreover, even if the growth of

the list cannot be bounded *in general*, we may still use an array with very occasional *realloc() and free()*.

List v.s. Array in HPC
In fact such in-general-irregularity can be found even in HPC programs. For example, in particle-oriented simulations we often need to hold the set of particles in each small *cell* in the simulated space domain. A straightforward implementation of such sets is to use a (double-linked) list for each set, because the additions and deletions of set members due to the movement of particles are easily implemented. However, if only a few particles move to other cells as we see in most of such simulations, it is irrational to use a list for the easy implementation of the movement of such minority particles sacrificing the performance of other operations on majorities. Therefore, we have to use an array for each set to have good performance, often to obtain an improvement of more than one order of magnitude[1], of the most frequent operations with a tough coding effort to take care of less frequent ones including that on array overflow.

Trees and Graphs
Finding regularities in the operations on non-linear data structures such as trees and graphs should be more challenging. However, if such a structure or some part of it is traversed many times, we might transform the structure into an array-based form in which the components of the structure, e.g., vertices of a graph, are ranked in a order convenient for repetitive traversals. This data structure conversion can be applicable to HPC programs in which sparse matrices are scanned many times. For example, sparse linear solvers repeatedly perform multiplication of a sparse-matrix and a vector (SpMV) in which the matrix and the product vector are accessed directly and sequentially while the accesses to the multiplier vector are random and indirect *in general*. However it is not impossible to reorder the matrix and vectors so that these three structures are mainly accessed directly and sequentially while (hopefully) a small portion of them needs exceptional random and indirect accesses.

Programming *Regular* Computation
Of course finding regularities and exploiting them are very challenging, not only because it should be tough to make a implementation really efficient, but also because the implementation itself is prohibitively complicated if we force programmers doing it by themselves abandoning the familiar and convenient way to manipulate irregular but flexible data structures. However we have to remember that our target in a big-data computing or non-trivial HPC is neither a list nor a one-dimensional array but a *set*, or neither a pointer-rich structure nor a CRS matrix but a *graph* or a mathematical matrix derived from the graph. Therefore, giving a proper abstraction of sets, graphs, etc. together with providing convenient operations upon them hiding the efficient *implementation* of them from programmers, we will have powerful, convenient and efficient means for big-data computing as well as some complicated HPC programming.

References

[1] Hiroshi Nakashima, Manycore Challenge in Kyoto: What We Learned from HPC Programming with KNC, In *ESAA 2014*, September, 2014.