PGAS languages in the exascale era

Extended Abstract

George Almasi, IBM Research

Description

The topic of this discussion is whether any aspects of Partitioned Global Address Space (PGAS) languages are likely to make it into the exascale era. I envision starting off the discussion with a presentation enumerating some of the best known programming practices surrounding PGAS languages. A discussion can ensue whether any of these are suitable for the highly hierarchical memory system that is likely to characterize exascale hardware.

Motivation

All but the most enthusiastic supporters of Partitioned Global Address Space (PGAS) languages will admit that the promise of PGAS has failed to materialize. PGAS languages have not taken over the world; they have even failed to make substantial changes in the state of high performance computing. Any single instance of a PGAS language has either failed spectacularly (e.g. HPF), found a niche with a few specialized applications written to it (e.g. Global Arrays) or has yet to prove itself and make a major impact (e.g. Chapel).

However, taken as a body, the ideas embedded into PGAS languages are slowly percolating into the consciousness of HPC programmers - and by that we mean specifically programmers of MPI codes. This might be entirely due to the influence of the PGAS languages themselves; but a much more likely explanation is that best practices are slowly improving the way MPI codes look, as well as being formalized into new languages.

The requirements of exascale architectures are likely to force a major change in HPC programming practices. This change is caused by the power requirements; while energy for floating-point operations might conceivably be found with the silicon technologies available to us at the end of this decade, all levels of the the memory subsystem are likely to suffer. Just naming a project ``exaflop'' condemns it to sacrifices in bandwidth and latency at the expense of floating point capability. Consequently the memory subsystems of exaflop machines are likely to be highly hierarchical and non-uniform. This is at odds with the 2-level hierarchy of MPI (``local'' vs ``remote'') or even OpenMP+MPI combinations. This might be the very opening for a new PGAS-like language (or languages).

The state of the art

Current high performance machinery is operated by a combination of OpenMP and MPI programming models, which nicely covers today's prevalent architecture - many nodes of relatively 100x- wide SMPs. One does not argue with success, but even supporters admit that using OpenMP and MPI together can be a detriment to productivity. Several of MPI's well thought out features, like modularity, are somewhat compromised by the approach. And of course, how is this combination going to take on a system with 5 or 6 levels of hierarchy?

The other contender for HPC at exascale level is NVidia's GPU approach. The advantages are obvious:

a clear separation of concern between the floating point intensive and communication rich codes; a clear separation of levels of abstraction; a reasonable (if not very attractive) programming model leveraging the strength of either side. For this, the programmer pays by having to re-think and re-write every application from scratch. Also, the programming model is fundamentally unsuitable to any programming model that looks like global shared memory, hamstringing the ``global view'', one of the more useful PGAS paradigms (more about this later).

Common features of PGAS languages

The collection of languages that call themselves PGAS has grown over the years. In this section we describe a number of common features that can each be found in a wide subset of PGAS languages. It is among these features that we should look for elements of the exascale programming model.

- Global View embodies one of the central tenets of partitioned global address space and sets it apart from MPI. The idea is that data structures are global and span the whole address space, not just individual tasks. It is easier for the programmer to think in terms of large aggregate data structures than in terms of their node-local definitions.

  Global View languages and libraries, such as ZPL, HPF and the HTA library, allow the user to express the global semantics of data structures, and operators on the whole data structure are defined, allowing concise and expressive programs to be written. The performance and scalability aspect is taken care of by allowing the user to define data layouts across the memory hierarchy of distributed systems. Distributed data layout primitives can become quite involved and need extensive compiler support to realize.

  Apart from the inevitable compiler complexity that springs up with this approach, its chief disadvantage is its limited power of expression.

- Parallel loops and loop distribution primitives are not unique to PGAS languages - OpenMP has them too - but they show up in most PGAS languages and nicely complement the Global View paradigm. Loop distribution primitives capture a large variety of static and dynamic work scheduling paradigms. However, they are not universal and cannot deal with highly dynamic scenarios with severe load balance issues. The Unbalanaced Tree Search (UTS) benchmark illustrates a situation where no parallel loop distribution primitive is known to work well.

- Collective communication primitives have come along with distributed programming tools like BSP, PVM and MPI. There is a global view aspect to collective communication that makes them attractive to PGAS languages (e.g. a global matrix transpose operation can be trivially expressed with an MPI-like Alltoall primitive).

  There are some uneasy intersection points between collective communication primitives and PGAS languages. For example, collectives in MPI only operate on private (not shared) memory; protecting the semantics of collective communication in the presence of shared memory that is concurrently accessed by third parties is a non-trivial task.

- Weak memory consistency. Distributed shared memory (as employed by PGAS machines) comes with the implicit assumption that reads and writes to remote memory are more expensive than local accesses by several orders of magnitude. In this situation an implementation of strict sequential consistency becomes prohibitively expensive in terms of performance. Almost all PGAS languages circumvent this problem by defining some form of weak memory consistency or by prohibiting situations in which memory consistency can become an issue.

Another concession to performance is the presence of split-phase transactions, employed either as a means to tolerate access latency or as a means of overlapping computation with communication. Split-phase transactions are not specific to PGAS (as exemplified by MPI3 non-blocking collectives), but are crucial for performance reasons.

The cost of weak memory consistency and split-phase transactions to programmer productivity is not to be underestimated. In our opinion these features of PGAS languages should be handled with care, either by expert programmers willing to spend time debugging their code, or by the compiler generating split-phase transactions when it recognizes an opportunity.

- An almost inevitable consequence of global shared memory is the presence of pointers to this memory. While superficially attractive, the presence of pointers to global objects has the potential to cause major programmability issues, due to the fact that pointers do very poorly at expressing memory hierarchy. One can think of the nastiest pointer chasing code written with C-style pointer arithmetic compounded by locality and hierarchy issues. Some PGAS languages, like CoArray Fortran, having recognized the essential unproductivity of pointers to shared objects, expressly forbid them.

- Asynchronous execution primitives, as found e.g. in Scala, Chapel or X10, are high productivity refinements of the idea of active messages - work and data shipped to a remote processor for execution. They are indeed suitable for very natural expression of some programs, and constitute a valuable tool in the arsenal of parallel programming. They also have some downsides, like the loss of the comfortable and popular SPMD paradigm and the difficulty of detecting when all remotely spawned computation has terminated.

- Shared memory-like synchronization primitives, like locks, mutexes and atomic sections, are also present in several PGAS languages. They are very popular because familiar to programmers. However, locks in a distributed system come with performance challenges, and their overuse often leads to poor performance.

PGAS languages in the exascale era

While in previous section we have related facts about PGAS languages, in this section we venture opinions about what features could and should make it into a programming model suitable for exascale.

- No programming model today can survive in the HPC era unless it can complement and succeed MPI. A *much* tighter integration with MPI is called for, even at the risk of losing some of the purity of design. The times where a language can stand on a single central idea are past.

- The exascale language cannot be owned by a single company or be IP encumbered in any way. Success cannot be declared until multiple companies implement a common specification (as is the case with OpenMP and MPI).

- Any PGAS language that purports to provide scalability, ease of programming and performance will have to do so by means of a compiler. Some of the PGAS features enumerated in the previous section carry heavy programmability penalties and are very difficult to debug. The compiler will have to do at least some of the semantic checking, if not all the code generation.

- The PGAS languages today all have different base languages - anything from Fortran to java. In the author's humble opinion the only compiled language that stands a chance at being the base of an exascale programming model is C++. Java is a wash because of perceived and real performance problems and its awkward native API; Fortran is dated and is losing ground on its own. A number of scripting languages, most prominently Python, could figure in supporting roles (system wide aggregation and so on).