THE FUTURE OF CONCURRENT PROGRAMMING: HPCS LANGUAGES

AUTOR: LOZOVANU MIHAELA
Conducător ştiinţific: lect. superior Ciorba Dumitru
Universitatea Tehnică a Moldovei
E-mail:mihaela.lozovanu@gmail.com

Abstract: *Concurrent programming has been around for quite some time, but it was mostly accessible to the highest ranks of the programmer. This changed when, in the 2000s, concurrency entered prime time, prodded by the ubiquity of multicore processors. The industry can no longer afford to pay for hand-crafted hand-debugged concurrent solutions. The search is on for programming paradigms that lead to high productivity and reliability. Recently DARPA created its HPCS, High Productivity Computing Systems, and is funding the research that leads to the development of new programming languages that support concurrency: CHAPEL, X10, FORTRESS.*

Cuvinte cheie: *High Productivity Computing Systems, X10, Chapel, Fortress, global view.*

1. Introduction

In 2002 Defense Advanced Research Projects Agency (DARPA) initiated the “High Productivity Computing Systems” project, with the goal of accelerating both the performance of the largest parallel computers and their usability[1]. It was recognized that a significant barrier to the application of computing to science, engineering and large-scale processing of data was the cumbersomeness of developing software that exploits the power of new architectures. As part of the HPCS project, three computer vendors – Cray,IBM,Sun – have competed not only in the area of hardware design to address DARPA’s performance goals, but also in language design to address the software development productivity goals. In this paper is represented a snapshot of the language design efforts[2].

2. Three HPCS Languages

Yesterday’s supercomputers are today’s desktops and tomorrow’s phones. So it makes sense to look at the leading edge in supercomputer research for hints about the future. There is a well-funded DARPA program, HPCS, to develop concurrent systems. There were three companies in stage 2 of this program, and each of them decided to develop a new language:

- Cray: Chapel
- IBM: X10
- Sun: Fortress

All three languages are very similar. Three independent teams came up with very similar solutions– that can’t be a coincidence. For instance, all three adopted the shared address space abstraction. The languages are supposed to cover a large area: from single-computer multicore programming to distributed programming over huge server farms[3]. It was considered that they would use message passing, which scales reasonably well between the two extremes. They do, but without exposing it to the programmer. Message passing is a hidden implementation detail. It is considered too low-level to bother the programmer with.

3. Base language

The HPCS languages use different sequential bases. X10 uses an existing object-oriented language, Java, inheriting both good and bad features. It adds to Java support for multidimensional arrays, value types and parallelism and it gains tool support from IBM’S extensive Java environment. Chapel and Fortress use their own, new object-oriented languages. An advantage of this approach is that the language can be tailored to science(Fortress even explores new, mathematical character sets), but the fact that a large intellectual effort is required in order to get the base language right has slowed development and may deter users.

4. Parallelism in HPCS languages

Any parallel programming model must specify how the parallelism is initiated. All three HPCS languages have parallel semantics. All of them have dynamic parallelism for loops as well as tasks and encouragement for the programmer to express as much parallelism as possible, with the idea that the compiler and runtime system will control how much is executed in parallel.

The use of dynamic parallelism is the most significant semantic difference between the HPCS language and other languages with their static parallelism model. Having dynamic thread support along with data parallel operators may encourage a higher degree of parallelism in the applications and allows the simplicity of expressing their parallelism directly rather than mapping it to a fixed process model in the application.

Suppose we have to process a big 2-D array and have 8 machines in your cluster. You would probably split the array into 8 chunks and spread them between the 8 locales. Each locale would take care of its chunk (and possibly some marginal areas of overlap with other chunks). If you're expecting your program to also run in other cluster configurations, you'd need more intelligent partitioning logic. In Chapel, there is a whole embedded language for partitioning *domains* (index sets) and specifying their *distribution* among locales.

Domains are used, for instance, for iteration. When you iterate over an unmapped domain, all calculations are done within the current locale (possibly in parallel, depending on the type of iteration). But if you do the same over a mapped domain, the calculations will automatically migrate to different locales.

This model of programming is characterized by a very important property: separation of algorithm from implementation. You separately describe implementation details, such as the distribution of your data structure between threads and machines; but the actual calculation is coded as if it were local and performed over monolithic data structures. That's a tremendous simplification and a clear productivity gain.

4. Global View vs. Fragmented View

The era of the mighty single-processor computer is over. Now, when more computing power is needed, one does not buy a faster uniprocessor—one buys another processor just like those one already has, or another hundred, or another million, and connects them with a high-speed communication network. Or, perhaps, one rents them instead, with a cloud computer. This gives one whatever quantity of computer cycles that one can desire and afford. Then, one has the problem of how to use those computer cycles effectively. Programming a multiprocessor is far more agonizing than programming a uniprocessor. One can use models of computation which give somewhat of the illusion of programming a uniprocessor. Unfortunately, the models which give the closest imitations of uniprocessing are very expensive to implement, either increasing the monetary cost of the computer tremendously, or slowing it down dreadfully. One response to this problem has been to move to a fragmented memory model.

Multiple processors are programmed largely as if they were uniprocessors, but are made to interact via a relatively language-neutral message-passing format such as MPI, message passing library. This model has enjoyed some success: several high-performance applications have been written in this style. Unfortunately, this model leads to a loss of programmer productivity: the message-passing format is integrated into the host language by means of an application-programming interface (API), the programmer must explicitly represent and manage the interaction between multiple processes and choreograph their data exchange; large data-structures (such as distributed arrays, graphs, hash-tables) that are conceptually unitary must be thought of as fragmented across different nodes; all processors must generally execute the same code etc. These approaches results in *fragmented view* of the problem.

One response to this problem is *global view*[4]. This is exactly what HPCS languages offer. You write your program in terms of data structures and control flows that are not chopped up into pieces according to where they will be executed. Global view approach results in clearer programs that are easier to write and maintain.

5. Synchronization among Threads and Processes in HPCS languages

The most common synchronization primitives used in parallel applications today are locks and barrier. Barriers are incompatible with the dynamic parallelism model in the HPCS languages, although their effect can be obtained by writing for a set of threads to complete. X10 has sophisticated synchronization mechanism called "clocks", which can be thought of as barriers with attached tasks. Clocks provide a very general form of global synchronization that can be applied to subsets of threads.

In place of locks, which are by many as error prone, all three languages support atomic blocks. Atomic blocks are semantically more elegant than locks, because the syntactic structure forces a matching begin and end to each critical region and the block of code is guaranteed to be atomic with respect to all other operations in the program(avoiding problems of acquiring the wrong lock or with deadlock). The support for atomics is not the same across the three HPCS languages. Fortress has abortable atomic sections and X10 limits atomic blocks to a single place, which allows for a lock per place implementation.

The languages also have some form or a "future" construct that can be used for producer-consumer parallelism. In Fortress if one thread tries to access the result of another spawned thread, it will automatically stall until the value is available. In X10 there is a distinct type for the variable in which one waits and the contents, so the point of potential stalls is more explicit.

In Chapel, you not only have access to atomic statements (which are still in the implementation phase) and barriers, but also to low level synchronization primitives such as sync and single variables– somewhat similar to locks and condition variables. Chapel has the capability to declare variables as "single"(single writer) and "sync"(multiple readers and writers). The reasoning is that Chapel wants to provide multi-resolution concurrency support. The low level primitives let you implement concurrency in the traditional style, which might come in handy. The high level primitives enable global view programming that boosts programmer productivity.

However, no matter what synchronization mechanism are used, if the language doesn't enforce their use, programmers end up with data races–the bane of concurrent programming. The time spent debugging racy programs may significantly cut into, or even nullify, potential productivity gains. Fortress is the only language that attempted to keep track of which data is shared (and, therefore, requires synchronization), and which is local. None of the HPCS languages tried to tie sharing and synchronization to the type system in the way it is done in other languages.

References

1. http://en.wikipedia.org/wiki/High_Productivity_Computing_Systems
2. Lusk E.,Yelick K., *Languages for high productivity computing,* World Scientific, 2007
3. http://bartoszmilewski.wordpress.com/2010/08/02/beyond-locks-and-messages-the-future-of-concurrent-programming/
4. Vijay S., Bloom B., Peshansky I.,*X10 Language Specification,* 2011.