An Introduction To Programming With X10

July 28, 2012

Copyright © 2010 The IBM Corporation
All rights reserved.
This work is part of the X10 project (http://x10-lang.org).

Contents

1 Why X10?

The vast and omni-present computer industry of today is based fundamentally on what we shall call the *standard model* of computing (also called by some as the *von Neuman* model). This is organized around the idea that a single stream of instructions operates on *memory* – discrete cells holding *values* – by *reading* these values, operating on them to produce new values, and *writing* these values back into memory.

In the last few years this model has been changed fundamentally and forever. We have entered the era of *concurrency* and *distribution*.

Moore's Law continues, unabated – making available twice the number of transistors every 18 months for the same silicon. Traditionally computer architects have used this extra real-estate to provide faster implementations of the standard model by increasing clock-speed and deepening the pipelines needed to speculatively execute many instructions in parallel. Thus, the standard model has needed no change. Computer users have come to rely on having their programs run faster (with essentially no change) merely by upgrading to a new generation of computers.

Unfortunately, clock speeds have now run up against the *heat envelope* – speed cannot be increased without causing the chips to run hotter than can be tolerated by current packaging.

What should this real-estate be used for, then? Computer manufacturers have turned to a radical idea – multi-core parallelism. Instead of using all the real-estate to support the illusion of a single thread of execution, they have divided up the real-estate to support *multiple* such threads of control. Complex new phenomena now becomes possible – multiple threads may read and write the same location simultaneously. Thus, the relative speeds of execution of threads can now affect the outcome of the execution, resulting potentially in different outcomes each time a program is run.

This change has been coupled with a second, less visible but equally powerful change. The widespread availability of cheap commodity processors and advances in computer networking mean that *clusters* of multiple computers are now common-place. Further, a vast increase in the amount of data available for processing means that there is a real economic need to deploy the tremendous computational power of clusters in service of analyzing this data.

Consequently the standard model must now give way to a new programming model. This model must support execution of programs on thousands of multi-core computers,

with tens of thousadns of threads, operating on peta-bytes of data. The model should smoothly extend the original standard model so that familiar ideas, patterns, idioms continue to work, in so far as possible. It should permit easy problems to be solved easily, and must allow sophisticated programmers to solve hard problems. The model should be practical and easily implementable on all existing computer systems.

Since 2004, we have been developing just such a new programming model. We began our work as part of the DARPA-IBM funded PERCS research project which set out to develop a peta-flop computer (capable of 10^15 operations per second), which could be programmed ten times more productively than computer of similar scale in 2002. Our specific charter was to develop a programming model for such large scale, concurrent systems that could be used to program a wide variety of computational problems, and could be accessible to a large class of professional programmers.

The programming model we have been developing is called the *APGAS* model, the *asynchronous, partitioned global address space* model. It extends the old standard model with two core concepts: *places* and *asynchrony*. The collection of cells making up memory are thought of as partitioned into chunks called places, each with one or more simultaneously operating threads of control. A cell in one place can refer to a cell in another – the cells make up a (partitioned) global address space. Four new basic operations are provided. An `async` spawns a new thread of control that operates asynchronously with other threads. An async may use an `atomic` (and the associated `when`) operation to execute a set of operations on cells located in the current place, as if in a single step (as far as other threads are concerned). It may use the `at` operation to switch the place of execution. Finally, and most importantly it may use the `finish` operation to execute a sequene of statements and wait for all asyncs spawned (recursively) during their exeution to terminate. These operations may be seen as orthogonal to each other. Importantly, all these operations can nest arbitrarily with few exceptions[1]. The power of the APGAS model lies in that many patterns of concurrency, communication and control can be effectively realized as appropriate combinations of these constructs.

Any language implementing the old standard model can be extended to support the APGAS model by supplying constructs to implement these operations. This has been done for Java (X10 1.5), C (Habanero C), and research efforts are underway to do this for UPC. Our group has also designed a new programming language, X10, organized around these ideas.

This manual presents an introduction to programming in X10. It covers the core constructs.

[1] `atomic` and `when` are not permitted to contain APGAS constructs

2 The APGAS model

3 X10 summary

4 Basic Idioms

We start with the Hello World program.

5 Advanced Idioms