The Case for an Integrated Platform for Parallel Software:

The Dependencies Among Software Platform Elements

by Sean Halle

UC Santa Cruz

website: codetime.sourceforge.net
email: seanhalle@yahoo.com

Nov 26, 2006

Abstract

We present and discuss a graph of the dependencies among the elements involved in parallel software's life-cycle, from authoring, through distribution, to installation and run. The graph is rooted with the three goals of 1) high programmer productivity 2) write once, run high-performance anywhere 3) wide acceptance. The structure of the graph suggests that a single, real-world, non-profit "stewarship" entity should define and enforce a standard that specifies the elements and the interfaces between them. We define the collection of all the elements to be a "platform", which includes development tools, an intermediate format, and run-time support, among others.

1 Introduction

Parallel Programming is entering the mainstream due to the exponentially increasing number of cores on each successive generation of processor chip. The industry needs a solution to creating parallel software that allows similar levels of productivity to current programming practices, and allows a program to be written once then run on future generations of processors as well as on past generations of processors. The industry needs this solution to also be a widely accepted standard, so that software developers can write to one standard OS interface and distribution format, and their programs will run on any current hardware, while hardware manufacturers can write adaptors so that their machine will run any out-of-the-box program that conforms to the standard.

These goals, together with the peculiarities of parallel software, lead to a web of inter-dependencies. A means for creating parallel software which achieves all three goals will have to respect all of the dependencies.

We present a graph of many of these dependencies, then describe why each dependency-link is in the graph and what the presence of the link implies.

We draw the conclusion that the collection of dependencies implies that an entire software platform should be defined as a coherent comprehensive standard. The platform should cover the interfaces between every component, from development tools, through a packaging tool, through a computation model and intermediate format to an install-time compiler, an OS interface and a run-time system. Further, we suggest that a non-profit stewardship entity should be created that guides development, writes the standard, tests against a reference platform, and legally enforces conformance to the standard.

The stewardship entity would use branding and a certification mark to enforce conformance [Halc]. Only products which had passed the certification process could use the mark. The need for wide acceptance implies that the entity should be non-profit but charge for the certification process, in order to support its various activities.

2 The Dependency Graph

Each of the following numbered statements explains one or more of the dependencies depicted in fig 1.

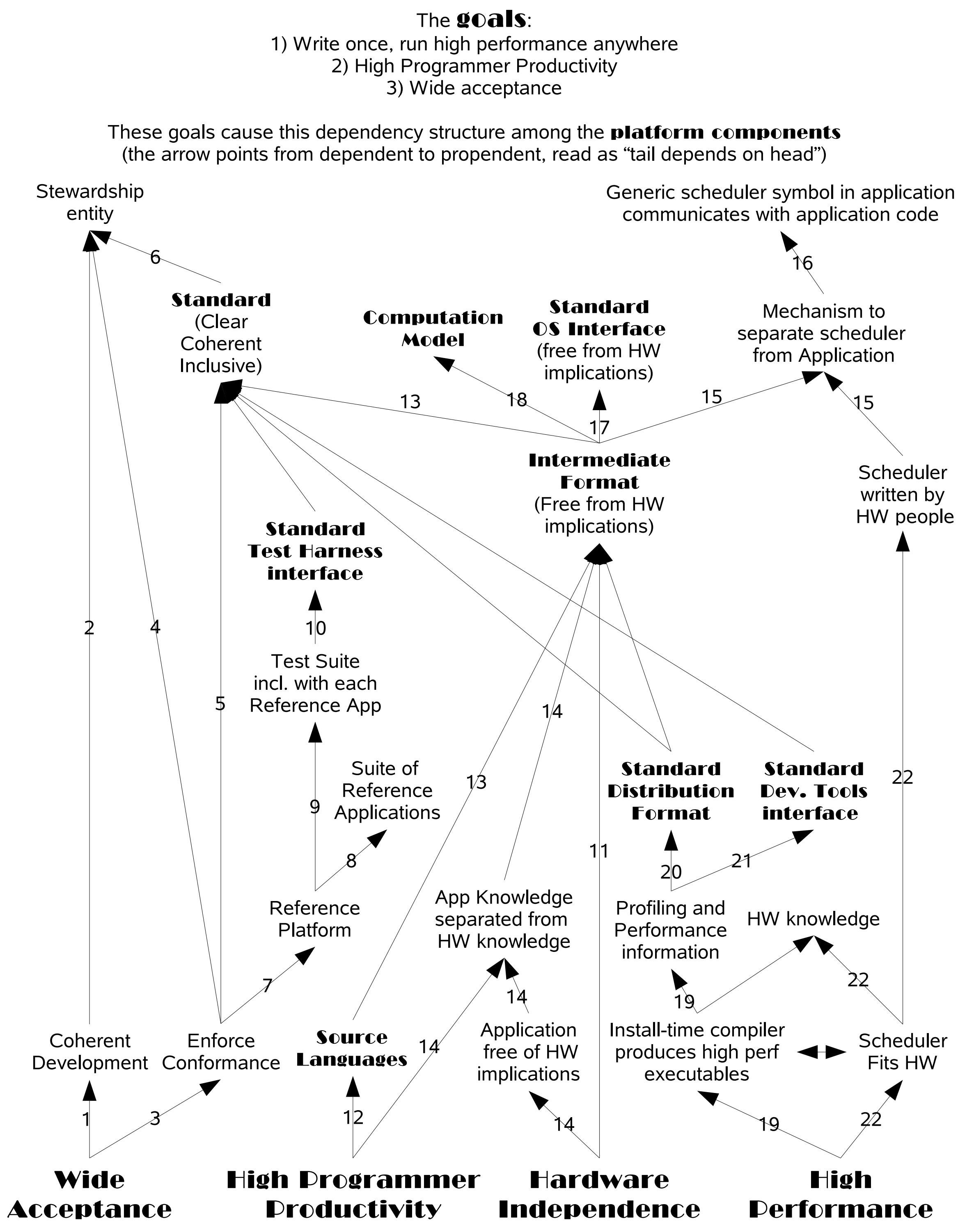
The goals:
1) Write once, run high performance anywhere
2) High Programmer Productivity
3) Wide acceptance
These goals cause this dependency structure among the platform components
(the arrow points from dependent to propendent, read as "tail depends on head")
Stewardship entity
Generic scheduler symbol in application communicates with application code
Standard (Clear Coherent Inclusive)
Computation Model
Standard OS Interface (free from HW implications)
Mechanism to separate scheduler from Application
Intermediate Format (Free from HW implications)
Scheduler written by HW people
Standard Test Harness interface
Test Suite incl. with each Reference App
Suite of Reference Applications
Standard Distribution Format
Standard Dev. Tools interface
Reference Platform
App Knowledge separated from HW knowledge
Profiling and Performance information
HW knowledge
Coherent Development
Enforce Conformance
Source Languages
Application free of HW implications
Install-time compiler produces high perf executables
Scheduler Fits HW
Wide Acceptance
High Programmer Productivity
Hardware Independence
High Performance

Figure 1. The roots of the dependency graph are at the bottom, where the goals are stated in bold. Each platform element is also stated in bold. From each goal emanates a number of arrows, each indicating something that goal depends directly upon. Every arrow indicates a dependency, where the arrow-tail is dependent upon whatever the arrow points to. The number on an arrow indicates which numbered-statement in the text explains that dependency

1. Many different groups and 3rd parties participate in software development, packaging, distribution, creation of hardware, creation of operating systems, and so on. Wide acceptance means that a

large portion of these groups use the same standard and the piece that each produces inter-operates with the pieces produced by the others. Interoperation is a synonym for wide acceptance. It needs, in part, that the development of interfaces between pieces be done in a coherent way.

2. Coherent development can be achieved in a number of ways, but many case studies and anecdotal evidence suggest that it is best achieved when a single entity is in charge of development.

3. Coherent development and hence wide acceptance can be derailed if a rogue entity implements non-conformant platform pieces and uses market manipulation or other tactics to force those non-compliant pieces into the community. Therefore some mechanism is needed to enforce conformance to the standard. A paper by Halle details one possible approach to ensuring conformance [Halc].

4. Standards created by committee have been effective in a few cases but implementations generally suffer from slightly varying interpretations of the standard. The committee has no means to test the various implementations nor to create consequences for non-conformance. Some entity with the financial means to perform the conformance testing and pursue legal protections is needed in order to have effective enforcement of a standard. For-profit examples include Sun's enforcement of the Java standard and Microsoft's enforcement of the windows standard. Non-profit examples include the foundation for Python [Pyt], and Linus Torvald's stewardship of the Linux kernel.

5. Before a standard can be enforced, a standard must exist. Because so many goals and platform elements depend upon having a standard, that standard should be clearly written, unambiguous, coherent, and inclusive of the desires of all of the members of the community that are affected by the standard (which biases towards a non-profit).

6. Because the entire community is affected by the standard and the standard needs to be clear and coherent, a non-profit entity that is friendly toward commerical 3rd parties needs to be created and charged with being a "steward" of the standard, the development process, and the reference platform. Such a non-profit entity is minimally biased, enhancing the probability of inclusiveness, is compact and centralized, enhancing coherence and clarity, and would charge only the minimal fees needed to support the development and enforcement processes.

7. The steward entity will need a means to test conformance as part of enforcing it. This requires a reference platform. Any community member wishing to get their platform element certified as compliant gives that element to the stewardship entity which unplugs the reference element and plugs in the element-under-test in place. It then runs a standard set of applications with known test results.

8. The reference platform has to include applications, which are run to test their interactions with the rest of the platform.

9. In order to get an application certified compliant, it must be tested on the reference platform, so it must be accompanied by a suite of tests. To be practical, the tests need to run automated.

10. To have automated tests come with every application, and have those tests run automatically on the reference platform, there must be a well-defined standard interface to the test harness. This interface must be one of the elements of the standard defined by the stewardship entity.

11. Hardware independence requires having a format, in which to represent programs, that gives no hint about what the hardware might look like. It cannot include the notion of a processor, as details in the application would then imply a particular range of the number of processors, which is a hardware implication. To gain extra performance, the distribution format will include optional performance-info which does expose HW details, but does not affect the answer computed, only the rate at which that answer is arrived at, so it can be safely ignored on alternative HW.

12. High programmer productivity requires having one or more reasonably high level source languages. Additionally, and not shown, is the dependence on quality development tools such as a code-aware editor, preferrably one using a parse tree to detect syntax, type, and grammar errors during editing; an integrated compile process to check run-time behavior; an integrated debugger; and so on.

13. To have multiple source languages, and a HW-independet format as stated in 11, and an install-time compiler as stated in 19, the languages must compile down to a common format, in which applications are distributed. Thus, the format a source language is translated to must be stated, and for wide-acceptance reasons, that format must be stated in the standard.

14. High programmer productivity also depends upon freeing the programmer from hardware concerns. The most complicated portion of writing parallel programs is related to the scheduling process, which maps tasks onto hardware resources and so is hardware specific. For example, two of the most difficult paralell-programming activities are getting locks and synchronizations right, and/or transforming the problem to one that maps well onto the hardware-friendly parallel constructs included in parallel languages. Both of these tasks are related to scheduling. The purpose, or effect, of locks and syncs in an application is to implement a portion of the scheduling process. The purpose of special parallel constructs in languages like HPF[KLSJ93], Titanium[Hea], and so on, is to provide easy-to-automatically-schedule on parallel-hardware constructs. Therefore, for both high programmer productivity and for hardware independence reasons, the application should not implement any portion of the scheduling process, nor should its choice of data structure or algorithm be affected by details of the scheduling process (such as having to map onto a limited number of scheduler-friendly constructs).

15. As explained in 14, to support high programmer productivity, the application must be ignorant of implementation details of the scheduling process, however, as stated in 22, high performance requires that the scheduling be able to change the sizes of application-defined data structures. This requires communication between the scheduling process and application code. The scheduler states how many pieces, which it decides based upon hardware concerns, and the application supplies the code which cuts data into pieces, gets a result for each piece, and puts together the individual results into a composite result for the original data. Having this communication, and remaining hardware independent, requires placing some mechanism into the intermediate format that allows the communication but keeps scheduling process details hidden from the applicaiton.

16. As explained in 15, a mechanism is needed which both hides scheduling details and allows communication between the scheduler, written by HW people, and the application, written separately by application people. One mechanism which provides this is a generic scheduler symbol in the source code that communicates with the rest of the program. This place-holder symbol uses a generic carrier to hand data plus a choice of number of pieces to an application-defined "scissors" that makes the pieces that are passed to a "body" that computes the answer for each piece then to a "tape-together" process that combines the piece-results into a composite result of the original data. Two papers by Halle include sections discussing one possible way to provide this mechanism[Halb][Hala]. Sample code for a Matrix Multiply program shows code for a scissors, body, and combiner [Hale].

17. The intermediate format, to be complete, must include a standard interface to operating system services. To be free from HW details, that OS interface must be, in turn, free from all HW implications. Thus, details like format of machine network address, location of data files, and so on must be removed from OS interactions, while still providing all the services needed by an application. A paper by Halle describes one way to do this [Hald].

18. An intermediate format embodies a computation model. So, the semantics of that computation model must be defined as part of the platform standard. A paper by Halle gives the semantics of one way to define a computation model that is free of hardware implications [Halb].

19. High performance relies upon tuning things to specific hardware details. To both have this and hardware independence in applications, a second compilation step is needed, which goes from the HW independent intermediate format to the machine format of specific processors. The more sophisticated this compiler, the higher the performance of the code it produces. Unfortunately compiling during a run doesn't allow time for complex techniques. One solution is to perform the hardware-specific compilation as part of the install process. This can be left to run for long periods, even adjustable by the user installing the code. Such a compiler can also be written to each specific platform, including details of the pipeline of the particular processor and cache configurations.

20. High performance optimizations depend upon common program behavior. Thus, profiling information is needed. Thus, the collection of profile information should be part of the standard. This implies that the test harness should automatically collect profile information. It also implies that the distribution format should be standardized and should specify the format of profile information.
    In addition, for scientific computing on multimillion dollar machines and other applications intended for particular hardware targets where performance is paramount, the platform should include a migration path. First the code is developed and tested correct, and profiled, then annotations are added in key places which state hardware-specific knowledge to improve run speed. This requires that the distribution format include a section for performance-enhancing information. It should be flexible enough to allow source language developers to work with particular install-time compiler writers to come up with whatever works between them. This info must be ignorable by other install-time compilers and still arrive at the correct answer. Thus, a flexible format for performance info should be part of the standard distribution format.

21. As indicated in 20, the collection of profile information should be part of the standard. Thus, the test harness should automatically collect profile info and communicate it to the tool that packages up the distribution (a distibution consists of source-compiled code, content data the application uses, performance info, and so on). This depends upon having a standard interface between the testharness and the packager tool. A standard interface is also needed between the source-compiler and the distribution packager.
    Also, much work has been done that suggests that programmers may be more productive in a visual environment, this implies the authoring tool, source compiler, debugger, and test-harness share one visual environment. To make this standard, the platform needs to specify interfaces between these tools. Then, one of the tools can be unplugged from the reference platform and a 3rd party tool plugged in, after which the entire process of source compiling and packaging will still work (it must work in order for the 3rd party development tool to receive certification).

22. High performance relies upon being able to define tasks that are of the optimal size and abundance for the particular hardware (a paper by Halle discusses this aspect of scheduling in more detail [Half]). In order for a scheduler to tune task size to fit the hardware, it must be written with hardware knowledge, preferrably by people with expert knowledge of that hardware. Thus, scheduling should be embedded into hardware-specific portions of the platform, such as the install-time compiler and/or a run-time system that the compiler compiles to.

Bibliography

[Hala] Sean Halle. The base codetime language. `http://codetime.sourceforge.net/content/CodeTime_BaCTiL.pdf`.

[Halb] Sean Halle. The big-step operational semantics of the codetime computational model. `http://codetime.sourceforge.net/content/CodeTime_Semantics.pdf`.

[Halc] Sean Halle. The codetime certification strategy. `http://codetime.sourceforge.net/content/CodeTime_Certification.pdf`.

[Hald] Sean Halle. A hardware independent os. `http://codetime.sourceforge.net/content/CodeTime_OS.pdf`.

[Hale] Sean Halle. Homepage for the codetime parallel software platform. `http://codetime.sourceforge.net`.

[Half] Sean Halle. A mental framework for use in creating hardware-independent parallel languages. `http://codetime.sourceforge.net/content/CodeTiime_Theoretical_Framework.pdf`.

[Hea] Paul Hilfinger and et. al. The titanium project home page. `http://www.cs.berkeley.edu/projects/titanium`.

[KLSJ93] C. H. Koelbel, D. Loveman, R. Schreiber, and G. Steele Jr. *High Performance Fortran Handbook*. MIT Press, 1993.

[Pyt] The python software foundation mission statement. `http://www.python.org/psf/mission.html`.