Software for HPCC Petaflops Architectures — A White Paper

December 18, 1996

Geoffrey C. Fox
gcf@npac.syr.edu
http://www.npac.syr.edu

Northeast Parallel Architectures Center
111 College Place
Syracuse University
Syracuse, New York 13244-4100

Abstract

We propose that Petaflops programming requires two key components. The first is research into new approaches to software and algorithms, which can handle memory hierarchy, latency and bandwidth, and its relation to machine and application geometric structure. This alone will not lead to a user-friendly programming environment. Here, we suggest a sophisticated Web technology-based "new generation" system supporting convenient flexible access to high-performance runtime libraries written in Fortran, C++, and Java.

Note, we only discuss a few issues—in particular, languages and overall environment. Other critical areas, such as tools, are not discussed.

I: Petaflops Architectures

For the purpose of this white paper, I consider the following Petaflops class architectures in a time frame of about 2007. We assume that individual memory chips will have a capacity of two Gigabytes.

A: Conventional Distributed Shared Memory Silicon Architecture

1. Clock Speed, 1 GHz
2. Four eight-way parallel complex C.P.U.'s per processor chip, giving a peak 32 Gigaflops performance per chip
3. 8,000 processing chips giving over 0.25 Petaflops peak performance
4. 32,000 2 Gigabytes memory chips, giving 64 Terabytes of memory

B: Superconducting Design

1. 200 GHz superconducting C.P.U. with negligible cache and simple architecture, giving 200 Gigaflops performance (this is probably conservative)

2. Conventional memory subsystem

3. 5,000 supercomputing C.P.U.'s and the same memory as option A; giving 1 Petaflops performance and 64 Terabytes of memory

C: Processor in Memory PIM

1. Maybe this would need to be fair to use "previous generation" half-Gigabyte chips.

2. One divides memory real estate into processor and memory using "simple" 250,000 transistor C.P.U.'s. Each memory chip, if divided equally in area between C.P.U. and memory could have 250 1 Gigaflops C.P.U.'s each with one Megabyte of memory.

3. 32,000 modified memory chips leads to 8 Terabytes of memory, and 8 Petaflops performance.

Superconducting technologies machines are characterized by dramatic disparity between C.P.U. and memory performance. The PIM architecture has good memory access for problems where the data can be laid out geometrically in a fashion (probably two dimensional) that matches the machine. The PIM architecture only has modest memory per CPU (which can be increased by using less than 50% of the silicon real estate for C.P.U.) which suggests it may need to be integrated with conventional (class A) nodes to handle sophisticated operating system functionality. This makes PIM machines like "attached processors" but with very flexible C.P.U.'s.

II. Programming Environments for Petaflops Architectures

1. Overview

The issues in Petaflops software fall into three classes: Fundamental, Engineering, Usability.

As fundamental, we view the linked combination of memory hierarchy—latency and bandwidth that express themselves differently in the various architectures. Each architecture also has different geometric structure (variation of these parameters with data location) for these fundamental parameters. As applications have different geometrical structure, and consequently different locality and bandwidth needs, the memory hierarchy-latency-bandwidth geometry tradeoffs are different for each application. Correspondingly, the various architectures have different performance characteristics on each application. PIM performs well in geometrical structured applications; "classic shared memory" on less structured dynamic non-local problems; superconducting systems do not obviously perform well on any applications with substantial memory bandwidth, and communication needs.

As engineering challenges, we first cite scaling. We term this engineering as most potential Petaflops applications do exhibit naturally, enough parallelism (100,000 to 1,000,000) to exploit Petaflops architectures. However, it requires careful architecture to produce systems that do not forget about parallelism in an area (say, input-output) and so are unable to deliver the natural parallelism. We claim that most applications are naturally massively (scaling) parallel, but not necessarily of a hierarchical structure (that matches hierarchy of hardware). Thus, memory hierarchy is a fundamental problem; scaling is engineering. The second obvious engineering issue is building a software environment that is relatively complete and works reliably.

I suggest that current HPCC software has addressed "fundamental" issues quite well (expressing geometric structure with HPF and more general decompositions with MPI). However, "engineering" issues have dominated the HPCC software with unreliable slow to appear HPF compilers, inadequate tools, and architecture blunders, such as Sequential I/O.

Under usability issues, we put those software features that are designed to make parallel systems easier to use. Portable software is an obvious theme and HPF, HPC++ are examples of systems that try to be more usable than explicit message passing MPI. If you target PetaFLOPS systems at the "marine corps" initial users, it is not clear how important usability issues are. Broad use of Petaflops demands usable software. So "Fundamental" software issues are those that allow initial users by hook or by crook to implement their applications and get whatever performance the architecture is capable of. "Engineering" issues ensure that we manage, design, and fund Petaflops software well enough so it robustly expresses and supports what we know. Usability issues allow a broad range of user access to Petaflops architectures.

If we confront the three software issues with the three prototypical architectures, the "engineering" and "usability" issues are roughly independent of the chosen architecture. However, the "fundamental" issues are extremely sensitive to the architecture as they are intrinsically entwined with the study of what algorithms and applications run on the machine and how they should be implemented.

2. Memory Hierarchy-Latency-Bandwidth-Geometry

Let us first discuss the memory hierarchy-latency-bandwidth-geometry "fundamental" issues. I believe these are clearest for PIM architecture (in our list of three architectures) for classic "geometric physical simulations." Most identified Petaflops applications are of this class even though they are more dynamic and irregular than today's such problems. Here, machine and problem structure are well matched, and conventional geometric decompositions should be effective. For all architectures, it is important to study data movement in classic algorithms—conjugate gradient, multigrid, FFT (etc.) and find efficient primitives. Although irregular dynamic problems will be technically harder to implement, I expect that study of the simpler regular problems will reveal essential issues. Good software for Petaflops machines will be built around efficient data movement primitives implemented as native runtime. HPF provides an example of a set of primitives, but it is unlikely these will be sufficient—except possibly for the PIM—as both HPF (and MPI) express just one level of memory hierarchy. As discussed at the PetaSoft meeting, research is needed on how to express the memory hierarchy, and the useful collective and point-to-point data movement and computation primitives (see Figure 1). This research should address both geometric problems, those like convolutions (FFT) with "long-range" structured data movement, and the presumably rather different parallel database, and Web server style applications.

An area which may have promise is extension of classic "load balancing" and data decomposition to memory management on Petaflops architectures. Many of the current powerful

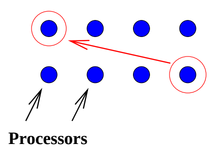

Figure 1a. Traditional MPI/HPF Data Movement Model

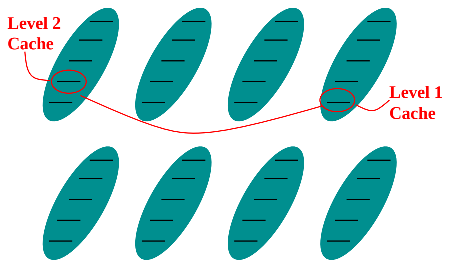

Figure 1b. Data Movement with Multiple Levels of Memory Hierarchy

Figure 1:

load-balancing algorithms can naturally deal with problem and computer hierarchy from either a computational graph (with different levels of contraction) or a physical (different resolution) point of view. One may be able to base adaptive memory movement on such algorithms.

The PetaSoft meeting exposed the need for a layered software model that presented a coherent virtual machine at of each level, but allowed user or system to “escape” into a lower more complex layer when needed for either performance or functionality.

3. Usability and Engineering in the Petaflops Programming Environment

We are perhaps pessimistic, but see no breakthroughs in usability. We expect that data parallel and message passing to be dominant forms of parallelism. We suggest that the “data parallel” approach could evolve with a different emphasis. One can view HPF as the language of identical operations on array elements—a limited concept as many parallel applications cannot be well expressed in this fashion. Rather, we view HPF as a high-level language that, through intrinsic functions, allows one to access a library of carefully tuned parallel algorithms. We believe this last view of HPF is the most promising and generalizable. Thus, our suggested usability model is through the greater use of interoperable parallel libraries. We assume that interpreted and/or graphical interfaces, such as APL, Matlab, or even Visual basic/VJ++ is nearer to the desired implementation than current HPF. In a layered view of the PetaSoft environment, we already mentioned the machine view with levels of the memory hierarchy supported by data movement software supporting “escaping” to lower levels. There is also a user’s layered view described below (see Figure 2).

a) Fully visual or scripted (interpreted) environment exhibiting domain specific functionality

This is optimized for user interface and only offers coarse grain access to capabilities in the fashion of AVS.

b) Partially scripted level offering

Portable flexible programming at some performance cost

c) Traditional compiled level

Offering a high-level language with few machine dependent features, and getting high performance—traditionally within about a factor of two of the peak performance possible on the particular algorithm.

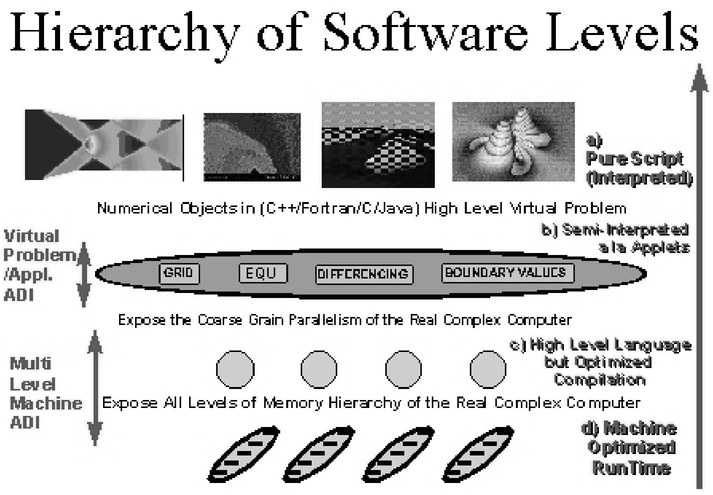
Hierarchy of Software Levels
a) Pure Script (Interpreted)
Numerical Objects in (C++/Fortran/C/Java) High Level Virtual Problem
b) Semi-Interpreted a la Applets
Virtual Problem /Appl. ADI
GRID
EQU
DIFFERENCING
BOUNDARY VALUES
Expose the Coarse Grain Parallelism of the Real Complex Computer
c) High Level Language but Optimized Compilation
Multi Level Machine ADI
Expose All Levels of Memory Hierarchy of the Real Complex Computer
d) Machine Optimized RunTime

Figure 2:

d) Traditional machine specific level

  Rarely used by application programmers or even those building (high level) tools. Clearly, allows user to obtain peak performance at the cost of a very inconvenient programming environment.

Examples in different domains of a) are Matlab (linear algebra, signal processing, etc.), JavaScript (document display), AVS (coarse grain dataflow), UNIX shell, and the growing number of Web interfaces. b) is illustrated by Java in Applet mode and Perl. c) is C, C++, Fortran, or compiled Java on sequential machines; such languages plus message passing or HPF on parallel machines. Looking at Petaflops machines, we see a critical problem that there is no natural correspondence between the hierarchy of machine levels (the virtual machine) and the hierarchy of problem levels (the virtual problem). The machine architecture affects programming levels a), b) and c). Level b) is the hardest with a key goal of designing a high-level language with minimal machine specific features—the analogies of HPF described in data parallel or MPI calls in message passing paradigm. I believe that it is unknown what the performance degradation (the hoped for worst case factor of two) will be obtained on Petaflops architectures with what user "directives" to optimize either/or parallelism or memory hierarchy. In fact, whereas it is almost certain that we can construct level a), it is not so obvious that the traditional high-level approach, b), will deliver effective performance. The "fundamental" research program outlined earlier is largely aimed at understanding possible approaches to levels a) and b).

The highest levels, c) and d), are less sensitive to the Petaflops architectures for several reasons. A simple observation is that often software at this high level will run on a client machine, and so by definition be of "conventional" architecture (of course, some "Petaflops" architectures are natural extrapolations of "conventional" architectures). The machines supporting levels c) and d) would be responsible for visualization with the user coding customized Java applets to analyze and display results computed on a Petaflops machine. We believe such a model is reasonable in general, and that implementation of levels c) and d) will be needed independent of the Petaflops initiative. However, there are two interesting points that link these high levels to Petaflops systems. Firstly the possible difficulties in designing and implementing a user friendly powerful level b), suggests that it may be particularly important to develop levels c) and d) into a relatively complete high-performance environment. This would access optimized libraries written by experts using relatively machine specific software at levels a) and b). This growing reliance on runtime libraries seems to me one of the few promising approaches to future HPCC software. Note that level a) and b) are not "assemblers" and "compilers," but rather "machine specific" and "largely architecture independent."

There is a second, perhaps very pessimistic, reason to link the design and implementation of levels c) and d) with the Petaflops initiative. Thus, to do this programming environment "right" requires, we think, a break with the past. However, there will be a natural tendency to "evolve" existing approaches in the commercial mainstream. Thus, we see that the Petaflops initiative gives the opportunity to build a "new generation" programming environment or HPCC-NG of the type described earlier.

The success of HPCC-NG will largely depend on making good engineering designs—from adequate funding to careful design. Our personal prejudice would be to build HPCC-NG in terms of Web technologies—linked Web servers and clients with excellent support for Java as both a primary programming language and as a wrapper for other languages. Although much research and experimentation is needed to fill in the details of a Web technology-based HPCC-NG, enough is understood to scope out and start such an initiative. The overall framework would be a coarse grain software integration from environment (called WebFlow by me in the past) of which the principles are clear. We have started a community initiative to understand the harder and, in our opinion, more profound issues associated with the use of Java as the basic language for science and engineering simulations. However, as massive (Petaflops!) parallelism issues in Java will (as far as we can see now) be quite similar to those in C++ and Fortran, we don't think that one needs to understand or even believe in Java as a computational science language to initiate design and implementation of HPCC-NG.

III. Overview of Petaflops Programming Environment

Let us summarize our picture. We propose a research and development program focusing on understanding memory hierarchy-latency bandwidth and geometry issues in Petaflops architectures. This should involve Fortran, C++, Java and other languages. Considering the three Petaflops architectures described at the start of this white paper, we expect that the PIM architecture to be the easiest on which to get good performance. We expect that it will be possible with both MPI (explicit message passing) and HPF to obtain good performance at the cost of more and more complex directives and primitives with somewhat more machine dependency. Thus, we anticipate an acceptable but relatively low-level programming environment. The research program should, of course, involve applications, algorithms, architectures, and software.

We do not expect that this high-performance programming environment will be very attractive to the general user and suggest focusing on a sophisticated high-quality high-level ((c) and d)) environment with both domain specific and general capabilities. These high-level interfaces will access runtime libraries typically written by experts in parallel (Petaflops) computing.

This approach to a more user-friendly HPCC environment will require re-engineering the basic software infrastructure, and we proposed that a new HPCC-NG activity be initiated to provide a new Web-based framework.