Solving The Software Productivity & Parallel Processing Problem

July 9, 2012

DEFINING THE PROBLEM

There are two major problems facing the U.S. Software Industry. They are not new. The first, programmer productivity, has been a major concern since the earliest programmable computers in the 1950s. The second, using parallel processors to gain speed, has been addressed since the 1960s. The solution to both of these problems must be embedded in the design of the programming language used to solve them. This paper addresses the barriers to solution of both problems and provides basic insights on how both can be solved.

After reviewing the history of programming languages, it is clear that Henry Ledgard’s article in the year 2000, The Emperor with No Clothes, [1], addresses a major barrier. He quoted W. Edwards Deming (father of quality control) who stated “If you can’t measure it, you can’t improve it.” Ledgard’s message: “Without measures from repeatable experiments, software is not a science.” This same point was made 10 years earlier by David Parnas, [2]. Considered by most as the father of Computer Science, Parnas said: “most CS PhDs are not scientists; they neither understand nor apply the methods of experimental science.” These views are hard for the software world to accept. Yet both authors are clearly at the top of the list of people knowledgeable in programming language design.

At the very top is Grace Hopper, who wrote the first compiler while working at Univac in 1952. In 1959, after the CODASYL conference started the formal development of COBOL, Hopper’s programming group at Univac spearheaded the language design based upon her own FLOW-MATIC language, see Wikipedia, and [3]. Hopper's belief that programs should be written in a language that was close to English rather than in machine code or languages close to machine code (e.g., assembly language) was captured in the new language, and COBOL would go on to be the most ubiquitous data system language to date. Hopper went on to develop CMS-2, a language for the U.S. Navy that added math and scientific facilities to COBOL. CMS-2 provided the same hierarchical data and hierarchical instruction syntax that contributes huge productivity gains and applies directly to parallel processing. As shown below, software productivity and parallel processing are intertwined.

Software Productivity

Looking at the history of productivity in software, the field went quickly from coding in 1’s and 0’s to an English-like language, dramatically improving speed as well as productivity. However, software productivity has gone downhill since the early 1980s. As shown in the charts below, productivity in software has been declining - faster than any other industry surveyed.

From a 1995 Business Week issue, [4], that surveyed productivity in 25 industries, one can derive the percent productivity change over the previous five year period. The results are shown in Chart 1. Productivity changes in the Chip industry were at the top of the list (a gain of 153%). Software was dead last (a loss of 11%).
Independently, in February 1995, the Standish Group published a report, [5], on the software industry supporting the negative productivity findings and describing software failure statistics. When discussing these negative results with higher-level managers responsible for funding software projects, the managers agreed with the data, saying it matched their experience.

A December, 2004 article by Robert Groth, published in IEEE Software, [6], showed the percent productivity gain of various major industries over the 1998-2003 period, see Chart 2. Again, over this period, computer chips (up 95%) had the most gain in productivity. Software was again last on the chart with a decline (down 5%). The general productivity issue is discussed in Groth’s article, where different views of this dilemma are offered.

CHART 1. Data From Business Week - January 9, 1995

The bottom line is that the high cost of building and maintaining U.S. software, and the large number of project failures faced by developers has forced companies to reduce the size of their software projects and put many projects on hold. Large companies are now outsourcing their software overseas, to India, China, and similar countries.

Parallel Processing

The problem of developing software for parallel processors was addressed in the 1960s. Mathematical problems using vectors and matrices were approached using special hardware designs. Most of the work done in using large numbers of parallel processors still continues down this path, with hardware designs making up for the lack of a software environment to deal with the mainstream problem. Even when handling these special problems, processor utilization efficiency is typically down around 10%. The current state of affairs is best exposed by the following statements extracted from EE Times, [7], and Reuters, [8]. Although the latest date offered is 2008, no significant changes have been identified since. The people quoted are those on top of the field in judging problems versus solutions.

EE Times: ESC Fall 2007 Preview: Multi-cores, software's Gordian Knot [7(A)]
To fully utilize the hardware parallelism inherent in embedded multi-core designs, they say, will require a shift to a more implicitly parallel programming language and methodology. However, many, including researchers at Microsoft, believe that it will take at least ten years for the industry to shift to a new parallel programming framework.

EE Times: Intel CTO presses software developers to keep pace [7(B)]
BENGALURU, India — Software development and delivery have failed to keep pace with advances in computer hardware, according to Intel Corp.'s CTO. --- As hardware technology approaches the terascale level on the desktop, software has fallen further behind. --- One result has been a lack of parallel programming applications to leverage dual-and multi-core processing technology. Intel is looking for "new languages for programming in parallel," (Justin) Rattner told the India Semiconductor Association.

EE Times: Industry seeks a model for next-gen multicore CPUs [7(C)]
"The industry is in a little bit of a panic about how to program multi-core processors, especially heterogeneous ones," said Chuck Moore, a senior fellow at Advanced Micro Devices trying to rally support for work in the area. "To make effective use of multi-core hardware today you need a PhD in computer science. That can't continue if we want to enable heterogeneous CPUs," he said. --- The challenge in the parallel world is finding a dynamic and flexible approach to schedule parallel tasks from these modules across available hardware in complex heterogeneous multi-core CPUs.

EE Times: Multicore puts screws to parallel-programming models [7(D)]
Leaders in mainstream computing are intensifying efforts to find a parallel-programming model to feed the multicore processors already on chip makers' drawing boards. --- Developers need to expand the current software stack in fundamental ways to handle a coming crop of processors that use a variety of cores, accelerators and memory types, according to the company. --- Both AMD and Intel have said they will ship processors using a mix of X86 and graphics cores as early as next year, with core counts quickly rising to eight or more per chip. But software developers are still stuck with a mainly serial programming model that cannot easily take advantage of the new hardware. --- Thus, there's little doubt the computer industry needs a new parallel-programming model to support these multicore processors. But just what that model will be, and when and how it will arrive, are still up in the air.
Reuters - Craig Mundie, Microsoft Corp's chief research and strategy officer, is sure he has a good handle on where technology is going. When is another story. [8]

The computer industry has taken its first steps toward parallel computing in recent years by using "multi-core" chips, but Mundie said this is the "tip of the iceberg." --- To maximize computing horsepower, software makers will need to change how software programmers work. Only a handful of programmers in the world know how to write software code to divide computing tasks into chunks that can be processed at the same time instead of a traditional, linear, one-job-at-a-time approach. --- A new programming language would be required, and could affect how almost every piece of software is written. --- "This problem will be hard," admitted Mundie, who worked on parallel computing as the head of supercomputer company Alliant Computer Systems before joining Microsoft. "This challenge looms large over the next 5 to 10 years."

The comments by Justin Rattner (Intel CTO), Chuck Moore (AMD Senior Fellow, Technology Group CTO, and Chairman of the Technology Advisory Board), and Craig Mundie (Microsoft Chief Research & Strategy Officer) indicate the level of urgency toward solving the problem while at the same time talking about the number of years it will likely take based upon the history. Of this group, Chuck Moore (recently deceased) clearly had the best understanding of the problem and would likely have best appreciated the solution described below. In any event, it is apparent from these articles that a barrier to language improvement must be overcome. What these articles do not describe are the multiple barriers against changing the “language”.

The Language Barriers

The promotion of C-based languages (e.g., C, C++, C#, Java, etc.) ushered in the departure of a scientific approach to software as described by Ledgard and Parnas. AT&T spent approximately $3B promoting UNIX, and C went along for the ride. SUN spent on the order of $1B promoting Java. The result, languages taught in the academic environment are generally C-based, with virtually no scientific comparisons to other languages as occurred during the 1970s. This supports the claim that software (Computer Science) is not based on science. This has presented a barrier to technologies that could dramatically improve both productivity and the use of parallel processor chips. Because of this barrier, the most knowledgeable people in software language design have departed the field in frustration.

In fact there are multiple barriers to accepting a new language adding up to a huge resistance to change. Some of these are typical barriers to a disruptive technology, a well documented phenomenon.

Overcoming The Resistance To A Disruptive Technology

The general topic of breakthrough or disruptive technologies is well described by Christianson, [9] and Kuhn, [10]. They discuss the factors affecting the rejection of such technologies, delaying their use for many years.
The reasons for resistance generally fall into the following broad categories.

- **Not Invented Here** - Known as the NIH factor, this is harbored in exclusive private, government, or academic research laboratories where people are getting paid large sums to solve critical technology problems.

- **Job Security** - End users have spent a significant amount of time learning, using and becoming adept at the current technology. Whereas the new technology eliminates the need for their expertise, and removes the corresponding security barriers provided them by the existing technology, a major resistance to change is naturally created.

- **Financial Competition** - Huge investments exist in the current technology, and these are at risk of being wiped out. It is not unusual in such cases to find political intervention against the new technology.

There are additional barriers that may be more difficult to overcome than those described above. These are introduced below.

**LANGUAGE DESIGN TRADEOFFS**

There are a number of tradeoffs to be considered when designing a language. From the standpoint taken here, these must be measured by productivity improvement and parallel processor utilization efficiency (the same application must run N times faster). As described by Anselmo & Ledgard, [11], the key factors affecting software productivity are **Understandability** and **Independence** - in both the development and support phases. Understandability is measured by the ease with which one programmer can understand - and successfully modify - another’s code. Independence is a property of software modules, measured by the ability to change a module without affecting other modules (this is practiced in hardware).

Independence is also measured by data sharing: Two modules are independent if they share no data. Both properties - understandability and independence - directly affect the ease with which one can build software that takes advantage of parallel processors.

**Understandability**

A major part of controlling large complex software systems is knowing where functions are performed. This depends on how things are organized. The military is an excellent example of organizing to maintain control in complex dynamic situations. This starts with well-defined hierarchies. If military personnel were only identified by name, organizations would be hard to control - especially for someone new to the organization. As another example, simple databases may be organized alphabetically (no hierarchy). Anyone who has worked with complex databases knows this does not work. Hierarchies are critical.

Engineering drawings of physical systems provide a good example of modular hierarchies. Controlling the design of a huge airliner without hierarchical drawings is impossible. Grace Hopper understood this principle in the design of languages. The ease with which one can create and use hierarchical data structures and hierarchical rule structures is one of the main reasons why COBOL is the most productive data system language.
Hopper also used the combination of hierarchical structures and well designed syntax to greatly simplify the understanding of complex conditional statements. This included setting conditions as well as testing them. Finally, mathematical statements in both FORTRAN and COBOL are compatible with mathematical texts. But when language design is driven by requirements to keep the compiler small and easy to write, none of the above logic applies, see [12].

The next tradeoff is the time it takes to type code versus the time it takes another programmer to understand that code. When building complex software in a competitive environment, minimizing coding keystrokes becomes equally non-productive. However, considering the current popular software languages and apparent desire to produce spartan code, this tradeoff warrants further investigation. There are numerous articles supporting the above tradoffs, see for example [13], [14], [15], [16], [17], [18], and [19].

**Language And Information Theory**

The following quote was stated by Bjarne Stroustrup, the inventor of C++: “English is arguably the largest and most complex language in the world (measured in number of words and idioms), but also one of the most successful,” see [20]. It dominates the world of free trade. Considering the small size of the islands where it originated, its survival is attributed to its reliability. To understand this, consider the military motto, “information is power” - the more information one has to make a decision, the more likely a good outcome. If the information is misunderstood, the outcome is likely to be unfavorable. So what are the rules that ensure reliable transfer of information and understanding?

The two major objectives in communications are reliability and speed. Fast and reliable transfer of information is the goal of information theory, as evolved by Shannon, [18] and others. We start by noting that: Reliability of information transfers is increased by adding redundancy (i.e., additional data). This may be as simple as sending the same message twice, or using additional words, such as articles, adjectives, or adverbs. Redundancy is used when writing and reading computer memory. Bits are added to the data being stored to decrease the probability of error when reading it back. In wireless communications, it is not unusual to double the size of the original data stream (redundancy) to ensure reliable transfers.

English is considered to have a high degree of redundancy compared to most other languages, implying it is more likely that information is transferred reliably - and key to the survival of its users. Studies comparing interactive languages have shown that errors increase as statements move from good English to a more terse form, [14]. Comparisons of COBOL, FORTRAN and C-based languages will typically derive the following programmer reactions: COBOL is verbose; FORTRAN is fair; C-based languages are terse.

The typical misperception is that verbose correlates to slower run-times. In fact, these properties are totally unrelated. Since source language is translated to machine language, the burden is on the machine translator. Reliability and speed can be improved simultaneously. If we are after reliability, verbose is best. If we are after speed, COBOL is still clearly the fastest at handling data. FORTRAN still inverts large matrices faster than C-based languages.
The Problem With Symbolic (Terse) Programming Languages

Some languages encourage minimization of keystrokes with their spartan syntax and symbology. This is tantamount to minimizing internal documentation and therefore understandability of the code. Such languages may reduce the time to type a line of code; but this is a small part of the time spent on a module. Anyone who has had to work with another author’s spartan syntax will confirm that the time lost trying to understand, test and debug complex algorithms far exceeds that spent saving keystrokes. Our experience in working with complex software algorithms is that terse symbology is the wrong direction.

One can argue the alternative of producing documentation inside code with large blocks of comments. In fact, this is a good indicator of a poor language - one that has to be heavily annotated to be understood by the reader. Good languages minimize the need for comments while providing a high degree of understandability directly from the code. COBOL required very few comments, FORTRAN about 50% of a routine. It is not unusual to find well documented C-based language programs where 70% or more of the code is comments.

Grace Hopper understood this. That is why she maintained that programs should be written in a language that was close to English, and why COBOL has been the outstanding leader in the productivity race.

ARCHITECTURE

Another trade-off is pictures versus words. “A picture is worth a thousand words” stresses the need for understandability. In other fields, whether designing machines, ships, or buildings, architects produce engineering drawings. These drawings are time invariant, not flow charts, nor “approximate” block diagrams. They are precise engineering specifications of how specific elements and modules are connected together. They are used directly in production.

Drawings of software are generally abstractions or flow charts, an aid in the design and coding process, but not final elements of the actual production. More specifically, they do not specify precise connectivity, defining what routines or modules share what data. This is done in the code. As a result, the term “software architecture” has been used rather loosely.

THE SOLUTION

To develop software for parallel processors, one must be able to create modules that can run concurrently. This implies that they must be independent. To be independent, they must not share any data. This requirement led to the Separation Principle, [22], the rule that separates data from instructions at the language level. This allows one to track software module independence, and automatically allocate processors to processes at run-time on a large parallel processor.

Equally important, the separation principle provides the basis for engineering drawings of software, with a one-to-one mapping from the drawings to the code, a true form of software architecture. These facilities are embodied in a CAD system for developing software and simulations, see [22]. Separate languages that define the data and instructions are read easily - like English - by non-programmers, a requirement for engineers building simulations and for subject area experts validating complex models.
Looking at Figure 1, an instruction icon (rectangle) has access to a data icon (oval) when connected by a line. This provides a one-to-one mapping from drawings to code. It allows architects to visualize hierarchies of software modules as well as their independence properties. Software is controlled by engineering drawings that provide for architectures of module hierarchies with the ability to edit the code right on the drawing (see Figure 1). Using this CAD facility, scope rules are implemented by the architecture, a direct visualization.

If there is inherent parallelism in the system being designed, it can be represented in the architecture by creating independent modules. These are modules that have a special interface to share data so they can run concurrently on separate processors. This new technology makes it easier to develop software for parallel processors than C-based approaches for single processors.

Having used this system, it becomes obvious that architecture is as important for software design as for hardware design, especially when designing architectures for parallel processors. Without the Separation Principle, software architecture does not exist.

**SUMMARY**

The two major problems facing the U.S. Software Industry today are programmer productivity and using parallel processors. The solution to both of these problems is inherently embedded in the design of the programming language used to solve them. This paper describes both problems, why they have not been solved, and finally the solution.

Quotes from the top people in programming language design emphasize the need to put software on a scientific basis using measures from repeatable experiments. When the properties of understandability and independence are embedded in a language, high productivity and effective use of parallel processors can be achieved. Contrary to most existing literature, today’s C-based languages lack these properties, causing productivity in software to decline faster than any other U.S. industry. This has forced companies to outsource their software overseas.

A new CAD system dramatically improves software productivity using English-like languages that make complex algorithms easy to understand and maintain. Hierarchies inside these languages help make software easy to design and understand, as well as run much faster than the current technology.

With this new technology, parallel processor utilization efficiency is considerably higher. Hardware designers can save precious chip space otherwise wasted by the current paradigm used to build software. The CAD approach described here eliminates the need for thread synchronization and cache coherency in hardware, leaving chip space for more memory to improve speed.

This technology makes it easy for designers to take maximum advantage of the inherent parallelism in systems while dramatically improving processor utilization efficiency. This increases parallel processor speed multipliers while using fewer processors. Finally, this CAD approach makes it easier to develop software for parallel processors than current approaches on a single processor.
Figure 1. Engineering drawing of software.
REFERENCES

[7] EE Times Articles: (A) Bernard Cole, 09/13/07; (B) K.C. Krishnadas, 01/17/08; (C) Rick Merritt, 02/13/08; (D) Rick Merritt, 02/15/08.
[8] Reuters Article, Daisuke Wakabayashi, Seattle, 03/13/08.