SAND2007-2904C

Bringing Up Red Storm: Lessons to Remember

Robert A. Ballance
John P. Noe
Sandia National Laboratories
Albuquerque, New Mexico
May 4, 02007
{raballa@sandia.gov, jpnoe@sandia.gov}
SAND XXXXXXXXXX

Abstract

Capability computer systems are designed, developed, and operated to provide the computational power to investigate and solve some of the most difficult and compelling problems facing humanity. These unique resources can be extremely helpful in scientific endeavors, but present exciting challenges for the operators and customers who utilize their power. While many of the difficulties in getting capability jobs run are inherent in the job itself, others intrude by way of operational issues. Consider Red Storm: in Jumbo mode, its largest configuration, it provides 13,280 dual-core processors; over 30,000 links; 3,700 embedded Linux sensors; 88 RAID controllers; 50 10GigE network connections; plus cooling, power, and environmental sensors. For each component, the only definitely known state is "down." This overview looks at capability computing from the aspect of operations, but draws many conclusions about the effects of system design on the effectiveness of system operation.

Overview

Sandia National Laboratories has been building large systems for 15 years now. Prior to that, the Labs had been a purchaser and operator of systems like the Paragon, Thinking Machines CM-2, and Cray vector systems. During that time, Sandians have designed, deployed, and operated the first Teraflop system (ASCI Red), several of the first large commodity clusters (CPLANT), and this week's #2 system Red Storm. The first author joined Sandia several years ago, after working with the University of New Mexico and NCSA in standing up Los Lobos, one of the first Linux clusters to break the Top 100. (Sandia's CPLANT was another; HPTI built the third.) Today, Sandia also operates one of the largest Linux clusters in the world, Thunderbird with 8,960 processors.

Figure 1 maps several of many sets of influences on our thinking and on the subject of today's talk. Solid lines indicate direct lines of descendants, from the MPP's like the nCube and Paragon through ASCI Red to the Cray XT4 line, and from the first Beowulf clusters through commercial offerings like Los Lobos to Thunderbird. Dashed lines indicate influences.

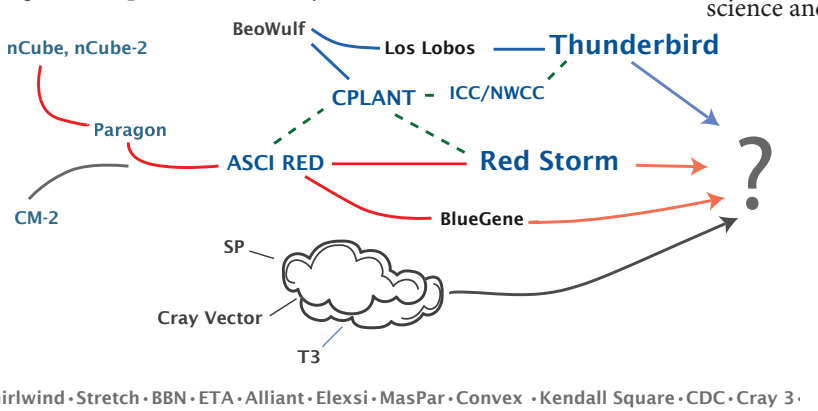

Figure 1: Systems from a Sandian's Perspective

Figure 2: ASCI Red, Installed 01997

Operating systems also play key a role, with the red lines trending the use of small, lightweight kernels, and the blue lines following variants of Linux. In 02007, Cray is working to blend the red and blue lines, which might bring us to something like Cray Purple, but that name has already been taken!

The lower tier of Figure 1 reminds us that many systems have been designed, fewer have been built, and even fewer have become ongoing commercial successes!

The systems named, of course, are only a handful of those involved in a great conversation. The topic concerns Exaflops for science and engineering: how to achieve them, how to deliver them, how to apply them, and how to fund them. In 02007, the next clear step is to a Petaflop, and from a theoretical Petaflop to a sustained, production, delivered, multi-petaflop system. There is no doubt that any of several vendors can construct a system with a PetaOp peak. What we've found, however, is that getting from the PetaOp peak to the sustained, production mult-petaOp system will take sustained effort from a broad-based team having diverse talents.

Isn't it interesting that we don't have a unit for sustained human effort? We need a unit that would signify honest attempts and would harmonize with Petaflops and Exabytes, something like Giga-tries

Figure 3: ASCI Red, 02006

or Peta-d'oh's. One of the motivations for this paper, and one of the missions for organizations like CUG and LCI, is to reduce the number of wasted human cycles in building ever-more-complicated systems.

But, lest we get too optimistic, let's consider the following four very different pictures. Figure 2 shows ASCI Red. Red was notable for several reasons. It successfully deployed a Lightweight Kernel on compute nodes called Cougar. Cougar descended from SUNMOS — the Sandia/UNM OS — that ran on the nCube, and which became PUMA on the Paragon. Red also had an integrated service (RAS) network that provided for system health and maintenance checks for all nodes. The RAS design on Red influenced both CPLANT and Red Storm. Finally, Red provided a simple mechanism to switch compute nodes from Unclassified to Classified processing and back gain. This red/black switching procedure allows Sandia to deploy capability computing on either network by sharing the compute partitions. The cost is relatively low: additional service and I/O nodes, and a mechanism for the switch. The duplicated I/O partitions have an additional benefit to operations, as well as a cost. The benefit is that during major upgrades, you can keep a portion of the system in production while other portions are being worked. Add to terms like scalability, reliability, and operability another: versatility. The cost, of course, is the overhead of maintaining two installations. In that sense, Red Storm is like having twins.

Figure 4: Clock of the Long Now

Figure 3 shows the end of ASCI Red. Red was #1 on the Top 500 list for 3.5 years; 7 lists! Red was decommissioned last year, not because it wasn't useful, but because we could no longer afford to keep it running. When it was turned off, there were still jobs on the mesh and more jobs in the queue. ASCI Red served production cycles for over 9 years from its initial operation until its retirement. That's a long time in supercomputer years! Of course there were upgrades along the way which took advantage of the rapid pace of technology maturation. Processors, memory, network interconnect, and disk storage were all upgraded.

The Clock of the Long Now

"I want to build a clock that ticks once a year. The century hand advances once every one hundred years, and the cuckoo comes out on the millennium. I want the cuckoo to come out every millennium for the next 10,000 years. If I hurry I should finish the clock in time to see the cuckoo come out for the first time."

Figure 4 shows a prototype of the Clock of the Long Now — an attempt to build a computer, and a very engineered artifact — that will continue to deliver cycles and be both maintainable and understandable for 10,000 years [4]. The cuckoo did come out the first time, seven years ago, and the clock struck twice: once for each millennium C.E. You might also be interested in Brian Eno's study of the bells for the clock; they are based on a 10-digit permutation so that each day the clock will sound a unique bell sequence! Of course, that sequence will eventually wrap around, in a bit over 10,000 years.

The Tacoma Narrows Bridge

Figure 5 shows the Tacoma Narrows Bridge, or "Galloping Gertie". Bridge builders have had a history of building new structures that creatively used new materials and new designs that pushed the envelope of bridge engineering. Just as regularly, on a well-documented 30-year cycle, bridges were built and collapsed. This cycle was originally noticed by Sibley, and is documented in Henry Petroski's "Engineers of Dreams" [12].

The amazing thing about the Tacoma Narrows Bridge is not that it was built, or that it fell down, but that the original architecture was carefully studied and engineered, and then built even when there were open questions about its stability. The width to

Figure 5: Tacoma Narrow Bridge

Figure 6: Red Storm 02006

span ratio of the deck was 1:76, while the next most comparable bridge (the Golden Gate) at the time had a width to length ratio of 1:46.5. Tacoma Narrows was truly a "stretch goal".

Bridges have a lot in common with supercomputers; they cost a lot, take a long time to build, carry a lot of traffic, and are only missed when they are gone. Like bridges, building a supercomputer often starts with a need and a vision. From those seeds begins lengthy rounds of funding negotiations, politics, further negotiations, and redesigns. Stretch goals and new technologies often determine the design of the system. Finally, if the team is lucky, the system is built and jobs start running.

Computer engineers tend not to lose the same number of workmen to death, accident, or caisson disease as the bridge builders, but we may lose the same number of systems in the end. How many of our prototypes become commercially successful? How many are maintainable very far into the future? And what can we do to make our efforts more successful?

The role of disaster in engineering needs to be acknowledged. Petroski, and many others, are far more articulate on this point than time allows. Stretch goals always involve risk, and at the edge of the known, the risk can be considerable. How then, can we continue to stretch while mitigating the risks and avoiding truly catastrophic disasters? "To boldly go where no one has gone before," to quote Gene Rodenberry.

To paraphrase Petroski (by substituting computers for bridges) *Must we thus expect, if not allow, a bridge (supercomputer) failure to occur now and then? The history and promise of bridges (supercomputers) suggest that we must, for reasons that have to do with neglect of the past and its relevance for the future. Neglect of the past is often embodied in a short-term historical memory, thinking, with hubris, that one's own generation's engineering science and technology have progressed so far beyond what they were a generation or two earlier that the bridges (supercomputers) of one's professional progenitors, and even ones mentors, make pretty pictures but not examples or models for modern engineering.*

The Principles

Looking back over our past successes (and failures!), Sandia teams have come up with the following principles for building reliable platforms that can serve both today's users and tomorrow's designs. These principles have been drawn from numerous systems, including those mentioned in Figure 1. Naturally, our attention is presently drawn to the most recent systems: Thunderbird and Red Storm. Where possible, we have tried to frame the lessons so that they might be memorable.

	Red Storm	Thunderbird
Stance	Red + Black	Black
Service Nodes	320 + 320	
Compute Nodes	12,960	4,480
Compute Cores	25,960	8,960
Segments	3360/6240/3360	
Interconnect	SeaStar Mesh	Infiniband
Disk	170TB + 170TB	46TB + 420TB shared
OS	Linux + Catamount	Linux

Seek First to Emulate

- Learn from the past
- Simulate the future

Learn from the past

This first principle is a restatement of Petroski's arguments for the use of historical precedent in the design of new structures. It was phrased succinctly in a quote by John Guale:

> *"A complex system that works is invariably found to have evolved from a simple system that works."*

Figure 7: Thunderbird

Both Thunderbird and Red Storm are complex systems evolved from simpler systems. Thunderbird, a 4,480 node InfiniBand Linux Cluster from Dell, is a straightforward extension of the Linux clusters that came before. Its scale, and to some extent the choices dictated by budget and time frame posed the newer problems. The design question that faced the Tbird team was not whether it could be made to work, but how to get it to work efficiently at scale. This was not a trivial problem; the system debuted at 38.27 TF, and through system tuning and changes to the software stack it delivered 53 TF a year later. What is interesting about this result is that TBird only managed to hold on to its #6 ranking, even with a 38% speedup!

Simulate the Future

Red Storm is based on the architecture proven in ASCI Red. In fact, a design goal of the Red Storm team was to ensure that applications which ran on Red would run on Red Storm with only a recompile. Red Storm has many differences: including a richer mesh, more complex I/O systems, and a new interconnect. Yet the basic architecture of the MPP remains constant. This architecture has since propagated into 16 other sites and almost 30 systems around the world, of varying sizes and interconnect topologies.

Looking forward, it will also be important to emulate in sense of computer architectures: we need to spend more time simulating new architectures in order to build larger machines that have well-understood characteristics. With technology moving as fast as it is, there's little time available to figure out how a system works after it has been built. Once operational, the clock of

obsolescence is ticking. Simulation, at all scales, is a partial view that can help to predict the behavior of the system before (and as) it is constructed. Work at Sandia looks at simulation both at the hardware [15] and software levels [13], enabling design teams to test alternative designs efficiently and with low cost.

The big bang only worked once[1]

- Nobody ever builds just one system, even when deploying just one system
- Globalize agility; localize fragility
- Deploy test platforms early and often
- Only dead systems never change

Nobody ever builds just one system

The nice thing about the big bang is that you only have to accomplish it once, and then the Universe is up and running. There is no such luck with large systems. Even when they are ostensibly off-the-shelf, each installation ends up with unique requirements. On top of that, as long as the software or hardware is improving, you will end up rebuilding the system.

This fact has several important implications. First, the administrator, will end up reconfiguring and reinstalling software. Any process that is time-consuming, awkward, or error prone once, will over time, become totally unwieldy. Error prone processes are especially disastrous in this situation, since the rebuild process is almost always executed under stress.

Second, any configuration process should support some form of consistency checking. All systems have some basic rules for configuration; being able to validate a configuration against those rules insures against some forms of error. During the early bring-up of Los Lobos, the management tool provided was XCAT. XCAT drove off a set of flat text files, but there was notion of consistency checking. It was easy to change one file, not change another, and induce a mysterious failure.

In the absence of consistency checking, one can instead adopt something like "pair programming" for system administration. One of the major effects of pair programming is that the code being written is undergoing immediate and constant review. This technique can easily be applied to system administration, if only the system administrators could be convinced to adopt it. The problem, though, is that consistency checking should be both automated and on-line so that any errors are recognized in real-time.

Third, some form of revision control over configurations is required, both for debugging and for disaster recovery. And fourth, there needs to be some form of regression testing after each configuration change, just to find the places where things break.

The authors continue to be amazed at the optimism of designers who expect to build a system (once) and have it work pretty much out of the box. Hardware designers are good at prototypes. The Clock of the Long Now is on its third working prototype, but then they have a schedule that allows time to learn. Software developers know what it means to see a prototype crash. In fact, you might claim that the software developers only have prototypes!

"Globalize Agility; Localize Fragility"

This quote comes from Tom Hunter, the President of Sandia. It certainly applies to the creation and deployment of large systems!

For starters, its hard to predict just how the system will work. It is harder, still, to predict just how the system will be configured when it is broken, or during an upgrade, or whenever. Building systems that can run fast, but still be effectively understood and debugged, is still an art.

An example of globalizing agility comes from tools that are built to handle non-uniform deployments. Many management tools, for instance, will assume that every node in a cluster will run the same kernel. Yet with specialized nodes, and occasionally inconsistent driver sets, this is a very optimistic assumption. Just because you can run different kernels doesn't mean that you should; inconsistency in large systems is our nemesis. However, the conscientious admin will always want the capability to change a kernel sometimes.

Deploy test platforms early and often

Test platforms allow for many activities, both before the main system is delivered and after. ASCI Red had numerous supporting systems used for software test and checkout, configuration testing, and application support. These activities are often mutually exclusive: its really hard to test a new scheduler configuration when the system software team is trying out the latest kernel, and the application teams get side-swiped in either case. Hardware testing, and testing at 'semi-scale' are also important. Teams are now focussing on Petascale systems and beyond; but as operations people we'd recommend starting smaller — say only 100 TF!

Test platforms give you space to run those regression tests mentioned above, as well as the place to develop, tune, and augment the regression testing. Right now, on Red Storm, much of our regression testing is by hand. Thunderbird, on the other hand, uses a more automatic testing model, especially when repairing hardware nodes.

Only dead systems never change

Once you do get it working, and running, and in production, what comes next? With Red Storm its a software upgrade, or a hardware upgrade, or an expansion. Sometimes you get all three, simultaneously. This implies that the operations team has to start all over again, using similar tools and techniques as the prior round. Or, you might have reached a point where new tools and techniques have to be invented.

For Red Storm, this happened when the system was upgraded from 41 TF to 124 TF. New dual-core processors replaced the single-cores. New networking hardware required a software upgrade. A fifth row was added. The system became twice as fast injecting packets into the mesh. The Lustre file systems grew to use all available I/O disk nodes. Same system? Same paint, though.

Only after the initial sanity tests did a deep problem surface: the I/O was stumbling due to networking issues. It took Cray and Sandia several weeks to discover that the original routing algorithms were no longer sufficient for the speed and volume of data being passed. Part of that time went into developing new tools to access deep data previously hidden in the system, including some very handy traffic counters. Access to those counters is now available for all XT administrators.

The new tools deployed as a result of this marathon debugging session remind us that systems have to grow smarter as they continue to operate. Systems learn because administrators and

1 Once in theory, for this Universe.

vendors continually add bits of code to ensure that, once recognized, problems and responses are not forgotten. A fundamental reason for architecting malleability into a system is to make it as simple as possible for this kind adaptation to take place. Building extension mechanisms into the software means that the learning can occur, and acts as a reminder that no one person or organization can ultimately predict all of the useful ways that the system might be used (or might fail!).

Moral: be prepared. When change ceases to happen, your system is ready for its next phase.

Build descalable scalable systems

- Don't forget that you have to get it running first
- Build scaffolding that meets the structure
- Leave the support structures (even non-scalable ones) in working condition

Don't forget that you have to get it running first

Scalability is a key to building systems like the largest clusters, or Purple, or Red Storm. It is also an elusive feature. At 100,000 components, a minor variation may be enough to trigger an imbalance. Moreover, any statistical argument will rapidly be stress tested due to the large numbers of events in the system. Once in a thousand years for a single component failure translates to once a month for Red Storm.

Conscientious designers, focussed on stretch goals, will compromise operability for scalability. Without a focus on scalability, the full system will not be scalable. While one must design for absolute speed, efficiency, and scalability, if you don't also build in structures that allow you to test, inspect, debug, and reconfigure the system, that day of full-scale, flat-out production gets postponed. And postponed again. We must find ways to build systems that can be "gracefully descaled" when the ultimately scalable solution falls over. Its also necessary to make sure that the descaling will work when needed.

In the quest for ever faster HPC systems, it is easy to rationalize away many intrusive debugging structures in the quest for absolute efficiency. Sandia's approach to MPP systems exacerbates the problem because Sandia deeply believes that less is more, especially when it comes to the amount of code in the main line. In return, it is common to assume that hardware and lower layers will work as specified.

What happens when the lower levels fail? At that point, one needs to have had all of those hooks installed so that we can turn back on the debugging code that was probably needed in the first place. And if there's a timing condition, and a dreaded Heisenbug appears, then one gets new insight as well.

The difficulty with building a scalable system is that you can so easily break the scaling aspects. For example, a minor piece of code that works well for small numbers of processors can affect the scalability of the application for large numbers of processors. Such failures typically appear as the system size increases by a power of two plus 1 (2^N+1).

Tom Gardiner, one of the Cray computational scientists working on Red Storm, uncovered the following behavior with Alegra, a Sandia-written application for computing magneto-hydrodynamics. The code scaled well out to about 8000 processors, but around 8000 processors there was a sharp drop in efficiency. The Alegra team rounded up all the usual suspects: I/O, MPI collectives, etc. But as one of my collaborators would say, "*Blessed are they who read the code.*" Tom found a piece of debugging code that was properly wrapped in conditional compilation directives everywhere except in the MPI rank 0 node. Rank 0 would obligingly pause and poll each process in the job, not once, but twice, for debugging output. The other nodes, of course, had no data to report. At 8,000 processes, that quick check would put rank 0 several seconds behind the rest of the processes.

Assumptions about global knowledge or behavior can also affect the scalability of the system, as can any built-in constants. Linear lookups and $O(n^3)$ algorithms hurt!

Build scaffolding that meets the structure

Two key questions before any acquisition are:

- Is the build/test/benchmark infrastructure in place first?
- Will it effectively support the installation team, the users, and operations?

The scaffolding that you need to support debug and test needs to meet both internal and external requirements. The internal requirement is the need to debug in terms of the inherent design abstractions and functions. This may seem obvious, but quite often the development and debugging teams end up on hardware or software that is not the same as the production system.

Externally, the scaffolding needs to meet the constraints of the operational environment. For example, it may not be possible to perform all debugging on an unloaded system. Or it may be necessary to allow multiple workers onto a single system. The early Cray XT systems had a very strong design constraint of only having a single software install on a single hardware installation. This complicated life both for developers (whose install gets used?) and for debugging (how to install a different release for a day?). Cray software is becoming more flexible in this regard.

Until recently, though, it was difficult to divvy up a single XT3 cabinet for multiple users, a capability that the initial lab systems provided, but that the production systems have lost. This turns out as a good example of useful scaffolding that has fallen into disrepair.

Leave the support structures in working condition

Of course, there are many examples of bit rot that we can all cite; code not exercised is always buggy. Support code, when not executed for long periods, will be the most out of compliance with the current code base. This means that one should regularly test the support code even though such code maintenance appears to be overhead. Programmers are at heart optimists. Support code is (or should be) dead code, after all.

You'll need to debug someday.

Like yesterday.

Make the lights green, then recheck the connections

- Software reports reality as it sees it
- Parallel tools for parallel systems

Software reports reality as it sees it

Software systems that rely on a single, internal, coherent view of their world are doomed to failure. For example, how many of you would volunteer to administer a file system without the equiva-

lent of fsck? But how many of us have been delivered tools that assume that their internal structures are correct, consistent, and which can't be compared to the existing system?

One of the key aspects of Jim Laros' work on management systems [20] is that the software is always capable of three important tasks:

- It can explore the system to see what is out there, and make that information part of the internal view. For example, Red Storm's disk management tool can map the controllers and determine attributes like model numbers and firmware levels that can become part of the tool's data base.
- It can make the system reflect its internal view. The same disk tool can be used to update firmware revisions by in order to make the controllers reflect the internal data.
- It can also provide us with a comparison of the internal structures and the external reality.

These are not wishful goals; they are the bedrock of managing and debugging a system that consists of many more components than human operators can recall. Yet, why do we trust the software? For that matter, why do we trust the hardware? As voltages lower, and as chips become denser, the probability (inevitability?) of soft logic errors increases.

One firm recommendation is that any software that maintains state about an operational system needs a way to compare that state to the current situation, and have ways to recover when the view diverge. It will also be important to provide online/real-time consistency checking.

Parallel tools for parallel systems

Engineers of parallel systems sometimes neglect to use parallelism themselves. For example, Red Storm has an integrated RAS network that is powered by network of embedded Linux processors: 3715 altogether in the Jumbo configuration. It is true that this hierarchical network has a single root. But this is really no different than a … Linux Cluster. It seems just right for exploiting locality, multiprocessing, parallelism, and all sorts of well-known techniques. Unfortunately, our experience has been that many tools are designed to run on a single processor. Sometimes with multiple threads, but still a single processor.

Version 1 of the Cray management tool for Red Storm was delivered with a simple facility to dump the state of the machine. It uses the RAS network. Of course, when we needed it, the system was in Jumbo mode: 10,368 compute nodes. System had crashed; advice was to gather a dump. Our estimate was that it would have taken over 3 days to complete the dump.

As a counter example, both Jim Laros and Ballance have written libraries that fan out work to available processors in a Linux Cluster. Jim's was for CPLANT, Ballance's for LOS LOBOS. These libraries make it simple to start a task on any node (and especially on an admin node), and then use the power of the cluster to do the work. Many others have written similar tools. These techniques, together with an extreme focus on scalability, have to be the cornerstone of any large machine.

Even Tiger Woods has a coach

- Don't assume you know/understand it all
- Observers help
- Never underestimate your blind spots

Don't assume you know/understand it all

One of the issues with designing/building/running a large system is that a complete understanding of all the pieces is beyond the scope of any single individual, and possibly beyond the scope of many teams. System interactions can be as small as the electron level within a chip or cable, and as large as the local power grid or an ensemble of 12,000 nodes. Simple, replicated components make this easier. Complexity is a cost.

A good end-to-end example is the I/O subsystem in Red Storm, which involves Lustre file systems with 161 Linux-based servers, 41 raid controllers, fibers, and lots of moving disks. Being able to trace a performance problem through the multiple layers, and across the many sibling processes, requires the full attention of many experts: CFS, CRAY, DDN, OS specialists, and sometimes even the routing experts.

During the upgrade of Red Storm to Version 2.1 SeaStar chips, our test plan exposed a routing issue that had probably been present in Red Storm from its first deployment. The growth of the Lustre file systems, their distribution across the I/O sections, and the new speed of the network exposed a routing failure that resulted in extreme network congestion. It took several weeks of intense effort, and two new routing algorithms, to iron out the issues.

One of the good things about the distributed team was that one member was working from England. The time zones were in our favor: we could run tests and ship results during the daytime in North America while that team member slept, and in return the New Mexico team would awake to a new patch for testing and logging.

Observers help

This effort also illustrates the second point: observers help. The Red Storm project depends extensively on e-mail, e-mail lists, and teleconferences. It was a newcomer to a teleconference who suggested the definitive test which isolated the routing problem.

Another excellent example of openness has been the ongoing series of Red Storm Quarterly meetings. These meetings started as a way to communicate about the project and design to its stakeholders. Along the way, it has also grown to include prospective owners and operators of XT systems. Today, CUG and other groups are taking over this communication arena. But at the onset of the project, it served us well to expose the designs and to publicly discuss the issues and the trade-offs.

Transparency of process is important: the open source movement has long held this as an axiom. This is true not only of code, but hardware and system design. The rise of Web 2.0 augments this process by providing tools to support distributed collaborative relationships. Right now, the tools are available; and many teams use them both for code development and systems operations. The capacity management teams at Sandia are a good example; using Jabber and Trac, they have moved most of their configuration management discussions into Web 2.0. What we need, though, are for the vendors to open up their own processes into similar forums.

Never underestimate your blind spots

Declan Rieb of Sandia calls this *"Playing with mental blocks."*

"The most important benefit of the end to end arguments is that they preserve the flexibility, generality, and the openness of the Internet." [5]

Blind spots are the inverse of the false positives of the green lights; you can't see, or reason about, what you aren't measuring. Even when you collect data, it takes careful analysis — visual and statistical — to draw the conclusions. In a sense, we are asking of systems the same questions that were defined by Kaplan and Garrick [9] for any risk analysis situation:

- What can go wrong?
- How likely is it to happen?
- What are the consequences?

Let's add a fourth:

How will we know it has happened?

End-to-End arguments apply

- Within large systems
- Within teams

The original phrasing of the end-to-end argument had to do with the placement of functions in a network. Simply stated, the end-to-end argument states that one should not build into the network core any functionality that cannot be completely implemented within the network [5].

End-to-end arguments are similar to the arguments for RISC: building a complex function into a network implicitly optimizes the network for one set of uses while substantially increasing the cost of a set of potentially valuable uses that may be unknown or unpredictable at design time. Reducing the complexity of the core allows for future flexibility while reducing cost. The generality in the network that comes from avoiding over commitment to a specific set of functions allows for new solutions to arise. And applications don't have to rely upon, or work around, complicated core structures that are themselves potentially unreliable.

End to end arguments do not imply a blind allegiance to placement dogma, but to a careful consideration of the impacts of placing functionality within a network.

Within large systems

Is a supercomputer a network? No doubt.

Another example from Red Storm: complication and inflexibility in the core. We've recently had the experience of being flooded by error messages when a component in the system fails. In our case, it was triggered by Lustre. The flood can easily create

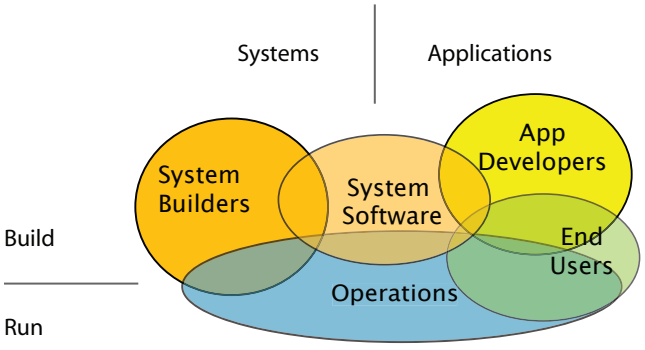

Figure 8: The Role of Operations

millions of messages and gigabytes of output. Cray's initial implementation had the following characteristics:

- No way to throttle messages; all or none
- No way to tune the message handling

The end-to-end placement failed in each direction. What could have been core network functionality (flow control of log messages) was not present in the network core. What could have been edge-defined behavior (message selection and processing activities) was implemented in the core, and so we could not adequately influence the behavior of the system.

Within teams

Does the team working on a supercomputer comprise a network? It does, and the management of the communications in the network is a worthy study. Review your communications mechanisms, and also the role of each individual in the overall communications. Revisit your own role. (Play with those mental blocks: see below) Does your team's communications infrastructure support their tasks and interaction styles? Does everyone have access to necessary information sources? Multi-hundred Gigabyte log files transmit slowly over the Net. Decisions can bottleneck at the decider.

Successful technology transitions require people transformations

Have you ever tried to teach your spouse or your parents to use PC, or a cell phone, a Palm, Blackberry, or even a VCR?

Jared Diamond, in *Guns, Germs, and Steel* [7] points out many times in the history of humanity that groups have successfully adopted a new technology, and other times when technologies were rejected. An example of the first was the adoption of horses by the Native Americans. Horses came from the Old World via the Spanish, but within a century their use, and their care, had spread throughout the West. Their owners changed in the process of course: they learned to ride and fight on horseback; they learned to care for large domesticated animals; they survived and developed immunities to new diseases.

Conversely, guns spread to Japan in 01453 CE, but after 01600 firearms were systematically rejected by the ruling Samurai class. The culture reverted to a gun-free society until 01853.

Closer to home, Sandia experienced some of the same issues in exporting the ASCI Red architecture to Cray. Overall, the transfer was successful, but turbulent. We might have improved the transfer process by finding ways to draw the communities into closer contact, up to and including co-locating some of the ASCI Red veterans with the design teams at Cray. This strategy worked well at Bell Labs in the 01970's when experienced telephone switching center operators were brought into the development teams in order to infuse the No. 1ESS project with a perspective for operations.

With any new technology, it is easy to transfer the physical or digital artifacts. What's hard to transfer is the insight and understanding of how to best use the new tools. This where the sensei arrives. There are several useful roles that need to be filled:

- Philosophers understand the end-to-end issues, the functioning of the big system, and where a specific subsystem fits in. Several of those Bell Telephone advisors stayed on with the project for many years, Several of them remained

with the project for multiple generations of designers, and ended up as senior architect-philosophers.

- Historians understand how the previous system evolved, and can cite not just the working solutions, but the dead ends involved. Evolution, after all, proceeds by exploration. All those dead ends are interesting, sometimes as points to avoid, and sometimes as starting places when the technology changes and makes an old approach viable once more.
- Tilt meters (another phrase owed to Michael Levine of PSC) are the canaries in the coal mines. However, their goal is not to die, but to say "What?". They are continually comparing the evolving design of a system against its goals, and try to ensure that the path to success remains open. A single person can, of course, fulfill many of these roles.

If there is a theme in these comments, it is that the architect must understand the end-to-end issues in the system, and communicate those issues to the specialists who are building components. Understanding the end-to-end issues depends on experience with prior systems, and depends as well on the ability to forecast behavior as systems scale to previously untested sizes.

Begin with the End in Mind

"Beginning with the end in mind" is the only viable approach to system design and deployment at this level.

The cast of contributors to developing any new high-end system is shown in Figure 8. Five groups of "personality types" appear. A sixth category, the funding officers and agencies, does not appear directly on this map. Like all maps, this one simplifies in order to make a point. In reality, many individuals migrate among the quadrants or play different roles within their career. In some ways, these categories correspond to "actors" in object-oriented design [6].

- System builders are drawn from computer scientists, computer engineers, and system architects. This group lives to build new, innovative, and much-needed systems.
- System software and library implementors are drawn largely from the same backgrounds as the builders. As a group, they implement the lower-level software required by applications. Many of the people included in this category are trained software developers.
- Application developers, unlike the system software developers, tend to have a strong scientific focus, with skills in software development derived from practice rather than extensive training. They also overlap with the end user segment of the population.
- End users actually use the high-end systems to achieve their goals. End users value predictable systems, job throughput, high operation rates, memory bandwidth, and high-bandwidth, low-latency networks. To the degree that they are also application developers, they also value ease of programming. However, not all end-users are application developers; many are users of applications developed by other groups.
- System operators and engineers get to make it all work. Sometimes, as with vendor-supported systems, they have the help of the vendor. Other times, as with commodity clusters, they have only their community of like-minded system administrators. They are the first to hear about and deal with user confusion and user dissatisfaction, and often are driven to create the tools they need to keep a system operational and in production.

What is the Apgar score?

"One minute — and again five minutes — after your baby is born, doctors calculate his Apgar score to see how he's doing. It's a simple process that helps determine whether your newborn is ready to meet the world without additional medical assistance.

This score — developed by anesthesiologist Virginia Apgar in 1952 and now used in modern hospitals worldwide — rates a baby's appearance, pulse, responsiveness, muscle activity, and breathing with a number between zero and 2 (2 being the strongest rating). The numbers are totaled, and 10 is considered a perfect score." [2]

High-end system development typically proceeds clockwise, beginning with the system builders. By the time an implementation reaches the lower half of the diagram, many design decisions are irrevocable and many resources are already committed. By the time the system reaches operations we are down to PERL scripts, SNMP, human ingenuity, and the (tested) good will of the system administrators.

But what does this development model imply for a system on which an application can run reliably, for long periods of time, on hardware that has intermittent failures or interrupted service? The application is now running far longer than the system will remain up; and might generate more data than can be stored locally. How do you manage this application? How can you manage the hardware? What does it mean for high-end computing to become a utility?

Operations are going to see it all in any case. The advantage to having operations involved from the start is that they often have the skills, tools, and stamina to travel over rough terrain to figure out the right direction of travel. In other words, they're like scouts for the army. Send them out early, and maybe they can help the army to assess its position, understand its terrain, and perhaps even be successful without a major battle (in this case, with the end users!)

Operability, like scalability, is not a feature that can be added to a system after it is designed.

Mind the Long Term

How is it possible to think about the long-term, when the lifetime of a system is only a few years? Is it worthwhile? Is it possible? The next big system is not, after all, the Clock of the Long Now.

Our limits to our vision come from three sources. First, it always seems like a miracle that the bridge will be built at all, especially given funding constraints and organization politics! The work of getting the system into place can be overwhelming. Red Storm had a several year deployment, starting from drafting the RFC to the transition to General Availability. Thunderbird, on the

THE HONORARY AWARD (Statuette). This award shall be given to honor extraordinary distinction in lifetime achievement, exceptional contributions to the state of motion picture arts and sciences, or for outstanding service to the Academy. [1]

other hand, provided no time to plan; it was all execution. From conception to first slap was about 6 months!

Second, teams don't build enough systems to get it right. Fred Brooks in the *Mythical Man Month* [3] has a wonderful comment about the third system being the right one; the first system is all learning, and then second time around the designer tries out all the stuff that got thrown out of the first.

Third, we focus so hard on design-order-build-install-run (5-4-3-2-LINPACK)! that the lifetime of useful service is hard to envision. Yet, it is during that post-HPCC phase that the effort continues, and the real work of HPC gets completed.

HPL-LINPACK, after all, is like an Apgar score. The Apgar is a simple triage to ascertain the level of care required by a newborn human. It is a really important number for about 24 hours. The HPCC suite improves the situation by incorporating more factors, but it is still primarily a birth-time assessment.

What is needed is a score less like the Apgar and more like an Academy Awards Lifetime Achievement: some set of numbers that will summarize the total productivity, usefulness, and costs of a system over its lifetime.

NERSC has a candidate in their ESP benchmark [11,16]. This benchmark "is designed to evaluate systems for overall effectiveness, independent of processor performance. The ESP test suite simulates 'a day in the life of an MPP' by measuring total system utilization. Results take into account both hardware (PE, memory, disk) and system software performance."

The problem is to generalize from a day to a lifetime; this is left as a goal for one of you. What is the measure of a machine over its lifetime? Does it have a progressive solution (e.g. first 6 months, year, two years, ...) And if you solve this, how do you get the vendors to start measuring it?

Closing

To restate the primary lessons:

- Seek first to emulate
- The big bang only worked once
- Build descalable scalable systems
- Make the lights green, then recheck the connections
- Even Tiger Woods has a coach
- End-to-end arguments apply
- Successful technology transitions require people transformations
- Begin with the end in mind
- Mind the long term

The design principles for the Clock of the Long Now [4] prove to be excellent guidelines for building large systems as well. Briefly stated they are:

- Longevity: Display the correct time for ten millennia
- Maintainability: with Bronze-age technology if need be.
- Transparency: obvious operational principles.
- Evolvability: improvable over time.
- Scalability: the same design should work from tabletop to monument size.

All worthy goals — suitably re-framed — for the next Petaflop system!

Acknowledgements

This paper could not have been contemplated without the opportunities afforded us to build, learn, reflect, and build again. Discussions and examples have been drawn from many conversations, and several employers.

Special thanks to Ron Brightwell, Bill Camp, Sophia Corwell, Michael Hannah, Tram Hudson, Steve Johnson, Sue Kelly, Ruth Klundt, Jim Laros, Rob Leland, Geoff McGirt,John Naegle, Kevin Pedretti, Rolf Riesen, Jon Stearley, Jim Sundet, Jim Tomkins, John Van Dyke, and Lee Ward for their insights, their stories, and their often elegant turns of phrase. Bob Ballance would also like to acknowledge Joe Davison, Frank Gilfeather, Barney Maccabe, Michael Mahon, Brian Smith, and Michael Van De Vanter for many fruitful interactions.

Finally, thanks to all the system administrators who are keeping them all running, even as we speak.

References and Readings

1. Academy Awards Web Site 2007, http://www.oscars.org/78academyawards/rules/index.html
2. BabyCenter.com, http://www.babycenter.com/refcap/3074.html
3. Brooks, Frederick P., *The mythical man-month: essays on software engineering*, 20th anniversary edition, Addison-Wesley, 01995
4. Brand, Stewart, *The clock of the long now*, Basic Books, 01999
5. Clark, David D., Blumenthal, Marjory S, "Rethinking the design of the Internet: the end-to-end arguments vs. the brave new world," *ACM Transactions on Internet Technology* (TOIT) Volume 1, Issue 1, pages 70–109, August 02001.
6. Cooper, Alan, *The inmates are running the asylum*, Sams Press, 02004.
7. Diamond, Jared, *Guns, germs, and steel: The fates of human societies*, W. W. Norton & Company, 01999.
8. Hillis, Danny, "The Millennium Clock," *Wired Magazine*, 01995 "Scenarios" issue.
9. Kaplan, Stanley, and Garrick, John B., "On the quantitative definition of risk," *Risk Analysis*, 1(1), 01981, pp 11–27.
10. Laros, James H, The Cluster Integration Toolkit, http://www.cs.sandia.gov/cit
11. NERSC ESP Website, http://www.nersc.gov/projects/esp.php
12. Petroski, Henry, *Engineers of dreams: great bridge builders and the spanning of America*, Alfred A. Knopf, 01995.
13. Riesen,Rolf , "A hybrid MPI simulator", *IEEE International Conference on Cluster Computing (CLUSTER'06)*, 02006.
14. Saltzer, J. H., Reed, D. P. and Clark, D. D., "End-to-End arguments in system design," *ACM Transactions on Computer Systems,* pages 277–288, 01984
15. Underwood, Keith D., Levenhagen, Michael, and Rodrigues, Arun, "Simulating Red Storm: challenges and successes in building a system Simulation," *Proc. International Parallel and Distributed Processing*, March 02007, IEEE
16. Wong, Adrian T., et al, "ESP: A system utilization benchmark, *Supercomputing 200*0.