Toward an understanding of the building blocks: constructing programs for high processor count systems

M H Reilly
SiCortex, Inc., Maynard, MA 01754 USA
E-mail: matt.reilly@sicortex.com

Abstract. Technology and industry trends have clearly shown that the future of technical computing lies in exploitation of more processors in larger multiprocessor systems. Exploitation of high processor count architectures demands a more thorough understanding of the underlying system dynamics and an accounting for them in the design of high-performance applications. Currently these dynamics are incompletely described by the widely adopted benchmarks and kernel metrics. Systems are most often characterized to allow comparisons and ranking. Often the characterizations are in the form of a scalar measure of some aspect of system performance that is a “not to exceed” number: the maximum possible level of performance that could be attained. While such comparisons typically drive both system design and procurement, more useful characterizations can be used to drive application development and design. This paper explores a few of these measures and presents a few simple examples of their application. The first set of metrics addresses individual processor performance, specifically performance related to memory references. The second set of metrics attempts to describe the behavior of the message-passing system under load and across a range of conditions.

1. Overview
The primary challenge in high performance systems design is not the development of new and powerful architectures, but in bridging the gap between potential system performance and the performance delivered by actual applications: that is, once a system is designed, how can users best make use of its performance features? To some degree, the recent anxiety around the introduction of multi-core processors and the end of the period of dramatic intergenerational improvements in single processor performance is an expression of this very problem. Users have expressed frustration at realizing they are tapping into just a small fraction of their system’s potential. In the past this frustration has been ameliorated by the knowledge that a new generation of processors would arrive in 18 months that would still deliver a fraction of its potential, but a potential that had doubled relative to the current generation: the surest software performance upgrade strategy was not tuning, but waiting for new hardware. The era of multicore designs has made that strategy obsolete.

If an application is to tap a greater portion of a multicore-multinode system, the application’s design must recognize the key performance characteristics of the underlying system. This is not to say that each application must be tailored to each target hardware platform, but awareness of the characteristics of some platform will at least improve performance in one instance. However,
few systems are characterized in such a way as to allow “architecture-aware” application design. Instead system characterization is driven most often by benchmark sets developed for pre-procurement comparisons. Where actual applications are used as benchmarks, their complex dynamics often obscure the underlying system characteristics to the point that the benchmark result sheds little light on just how a new application could be designed to best use the hardware.

Application design can and should be driven by an understanding of the behavior of the supporting hardware and system platform. Some common contemporary benchmarks are helpful in this effort but do not form a complete set. This paper explores a few new metrics for hardware behavior. First, we describe the SiCortex SC5832, the testbed for the concepts described here. Then we describe a set of metrics that can provide some insight into the design of an application. Then we review a few results in application or kernel design that have taken advantage of some of the new system metrics.

2. The System
The SiCortex SC5832 system comprises a cluster of 972 six-processor computing nodes connected via a high speed parallel point-to-point network. The network (fabric) topology is a Kautz graph that guarantees that the longest path through the network is logarithmic in the number of network nodes. In the case of the SC5832, the network diameter is 6 hops, with each node connecting three input ports and three output ports into the fabric [1], [2].

![SiCortex six-processor cluster node](image)

Figure 1. The SiCortex six-processor cluster node.

Figure 1 shows the single-chip, six-processor cluster node. Each processor is a 500 MHz low-power, 64-bit MIPS core and is described in table 1. All caches within the SMP node are coherent. The six processors share access to two independent DDR2 memory controllers, the DMA message handling processor, and a PCI Express controller.

An SC5832 is built from thirty-six modules, each containing twenty-seven cluster nodes, for a total of 972 nodes or 5,832 processors in the system. One node on each module connects to a dual Gigabit Ethernet controller. Three other nodes on each module connect to PCI Express Module slots to support external I/O. The SC5832 provides up to 324 PCI Express Module slots for connections to disk farms or external storage networks.
Table 1. SiCortex Node Chip Characteristics

| Characteristics            | Value          |
|----------------------------|----------------|
| Peak FP Rate               | 1000 MFLOP/s   |
| DP Matrix Multiply (DGEMM) | 730 MFLOP/s    |
| Stream TRIAD Bandwidth     | 350 MB/s       |
| Clock Frequency            | 500 MHz        |
| L1 DCache Size             | 32 KB          |
| L1 ICache Size             | 32 KB          |
| L2 Cache Size              | 256 KB         |
| Power Dissipation          | < 1 W          |

Table 2. HPCC communications characteristics for the SC5832

| Characteristics                | Value          |
|--------------------------------|----------------|
| Random Ring Bandwidth          | 50 MB/s        |
| Random Ring Latency            | 4 microseconds |
| Ping Pong Bandwidth            | 1.5 GB/s       |
| Ping Pong Latency              | 1.4 microseconds |
| PTRANS (Parallel Transpose) Bandwidth | 210 GB/s       |

The 972 nodes in the SC5832 are connected via a logarithmic diameter network described by Kautz almost 40 years ago [3]. The network offers low latency and high bandwidth communications and has the ability to route around failed nodes or links. Each node has three outbound fabric links and three inbound links as shown in figure 1. The peak bandwidth on each link is 1.6 GB/s. Table 2 presents the performance of the communications fabric as measured by the High Performance Computing Challenge benchmarks [4].

3. Memory performance

The STREAM benchmark set, and in particular the STREAM triad metric, is useful in predicting the cost in time of many data intensive operations. However, the STREAM tests stress sequential access to memory. Many large dataset operations exhibit distinctly nonsequential addressing patterns. Notable examples are sort routines and FFT operations.

In the case of sort routines, the address stream can be considered “random.” This random sequence can place significant stresses on the virtual memory translation unit, especially when the translation look-aside buffer (TLB) covers a region much smaller than the dataset to be sorted. Further, sort operations often look at a small part of the record. Such patterns exhibit no block level spatial locality, meaning that large cache blocks are often filled on a miss, only to have most of the data fetched into the block ignored. Even in cases where the list to be sorted is dominated by the key field so that several list elements fit in a single cache block, it is unlikely that more than one element in a cache block will be examined before the block is victimized.

In the case of FFT operations, the butterfly or bit-twiddle memory access pattern of the classic Cooley-Tukey algorithm presents a significant challenge to any prefetch logic in a compiler or a processor. For this reason, it is useful to consider two more metrics beyond the STREAM triad bandwidth for a processor. The first is the cost (load to use latency) of an isolated, unpredictable memory reference that misses in all levels of cache. In this case, we assume that the address translation is in the TLB so that the latency is entirely due to the time required to pass through the caches and extract the data from main memory (we consider here only systems with uniform access to local memory). Measuring this operation can be quite difficult, as good
compilers and clever reordering hardware can often frustrate attempts to measure real latency. The measurement technique is specific to each processor implementation. Nonetheless, such a number can be quite useful. Typical values are on the order of 100 ns, with some values reported as low as 35 ns.

A second interesting memory performance metric is the cost (again, load to use latency) of a load operation that results in a translation buffer miss where the page table entry misses in the cache and the resulting resolved target address also misses in all levels of the cache. Again, this factor can be quite difficult to measure. In the absence of any other knowledge, the cost can often be approximated as the latency of two sequential (non overlapping) references to main memory. Typical values for this metric are on the order of 200nS.

The final memory metric we consider is the “span” of the TLB: the size of the memory region it covers if all TLB entries contain mappings for a nominal sized page. Nominal here is used in the sense of “the page size used to contain most of the data that we are interested in.” While almost all modern processors support superpages and other methods of expanding the maximum span of the TLB for specially identified memory regions, it is important for this metric to use the actual page size used in practice. As an example, while the MIPS 5kf processor in SiCortex’ Ice9 node chip supports super pages as large as 16 MB, Linux support for variable page sizes is somewhat unsettled, so the dominant page size is 64 KB, making the span of the TLB 6 MB. Datasets larger than 6MB will cause capacity TLB misses. For long random-access sequences over very large databases, this can ensure that many such random accesses result in a TLB miss that causes a cache miss (to fetch the page table entry) and a cache miss on behalf of the original load instruction.

4. Communication metrics

4.1. Simple pairwise bandwidth

The HPCC benchmark suite has several interesting communication metrics that can be used to guide program design. However, all are scalar. For example, the ping-pong bandwidth is a measure of the maximum bandwidth available to a pair of communicating processors in an otherwise quiescent system. The PTRANS bandwidth is a good measure of how fast a large dataset can be exchanged among a set of processes. The random ring bandwidth is a good measure of what can be expected for process-to-process transfers of very large messages while the communications fabric is heavily loaded. Unfortunately these scalar measures offer little guidance for optimization as they obscure the relationship between delivered bandwidth and message size.

The ping-pong (zero-contention) and random ring (high contention) tests can be elaborated to allow insight into the dependence of bandwidth on message size.

The zero-contention bandwidth test is performed over a range of message sizes to measure the available transmit bandwidth under no contention. The test calculates the delivered unidirectional bandwidth between nodes of a randomly selected pair from the 5832 processor complex, while all other processors are idle. Figure 2 shows a scatter plot of measured bandwidth vs. message size. Each point in the plot represents a trial for a randomly selected pair and a message size. The peak of 1.4 GB/s is determined by the throughput of the node’s DMA engine. The spread over samples at the same message size is caused by differences in the network path length for each randomly chosen pair as it affects the latency of rendezvous response messages, and possibly by interfering low-level system events. Such a plot shows that the fabric becomes more efficient with increasing message size up to about 256 KB and that messages smaller than 2 KB are limited by the fixed cost of initiation. The spread also suggests that not all pairs are alike and even that we should expect system variability to affect delivered bandwidth. We see similar results on a 512-processor segment of the Lonestar Infiniband based cluster; see figure 3.

The random ring bandwidth test can be expanded in a similar manner (though in this case,
we do not use the random ring communication pattern, as it can confound attempts to measure communications behavior with operating system and benchmark synchronization effects). The available transmit bandwidth under high contention test, shown in figure 4, measures the average unidirectional message bandwidth across 2500 randomly assigned pairs of processors. The measurement is taken while all processors are exchanging messages to maximize fabric utilization.

The first region of the curve, for messages ranging from 4 bytes to 1 KB, is entirely determined by the round-trip delay and flow control overhead for the eager message exchange. Messages larger than 1 KB are transferred via a rendezvous protocol that eliminates the flow control overhead after an initial negotiation. Messages of 128 KB and larger trigger a transition to a multirail transmission scheme that was intended to optimize performance under low contention. Unfortunately the multirail distribution tends to increase network contention and reduce delivered bandwidth.

Note that good performance – in fact, nearly optimal performance – is obtained for message sizes ranging from 2 KB to about 100 KB. Further, the delivered bandwidth for small messages of 256 bytes is high enough to permit consideration of schemes that rely on small data transfers.
4.2. Collective metrics
Some applications or kernels rely on collective operations (such as the MPI_Allreduce operators) over the entire group of communicating processors. We have found that characterizing the collective operators relative to vector length and communicator group size has been useful in planning a spanning tree analysis program.

The SC5832 MPI library, based on MPICH, implements the AllReduce collectives using an algorithm requiring $O(\log p)$ message exchanges between the $p$ processes. The cost of the exchanges is $O(l)$, where $l$ is the number of elements in the vector [5]. Figure 5 shows the cost of an integer MIN operation over a range of vector sizes. For larger vectors, the cost is clearly proportional to vector length. For smaller vectors, the initiation cost of the operation dominates, and so we see nearly constant cost for vectors shorter than 128 bytes. The apparent “noisiness” in the result between 64 and 1K bytes may be caused by flow control dynamics in the eager message protocol, though we do not yet have any definitive evidence.

Figure 6 shows the variation in the cost of the AllReduce operation vs. the number of participating processes. The binomial behavior at the foundation of the implementation is clearly apparent in the scatter plot [5].

4.3. Work underneath nonblocking transfers
Frequently algorithms admit to some overlap of communications operations with actual calculation. In the context of MPI operations, it has been useful to know how much work can be fit between a nonblocking MPI message request (e.g., MPI_Irecv, MPI_Isend, MPI_Issend) and the corresponding MPI_Wait completion operation without adversely effecting the communications performance by more than one percent.

This test is more complex than the simple bandwidth measurements described so far. The algorithm for calculating the communication/work overlap for MPI_Isend is shown in figure 7. The “guess” step is driven by the history of s1 values. “Close” is defined such that s1 is within ±1% of s0. Half of the processors in the job execute this timing loop; the other half act as sinks, repeatedly calling MPI_Irecv.

Figure 8 shows the result of running the Isend work overlap test on two system configurations, the first using 128 processors on 22 nodes, the second using 4096 processors on 683 nodes. The plot shows the proportion of the communication time (the cost of the MPI_Isend and MPI_Wait
double findWorkTime(msg_size)
  s0 = Time(MPI_Send(msg_size)) // s0 is the time to complete an MPI_Send operation
  until (s1 is "close" to s0) do
    worktime = guess how much work I can do
    s1 = Time (MPI_Isend(msg_size); waste_time(worktime); MPI_Wait())
  end
  return s1

Figure 7. Communication overlap test for MPI_Isend.
The SiCortex MPICH implementation of MPI_Irecv is such that more computation/communication overlap can be found by calling MPI_Test or MPI_Iprobe at intervals while performing overlapped computation. So, the MPI_Irecv overlap test is modified by breaking the “work time” in figure 7 into 1, 2, 4, or 8 segments. Figure 9 shows the available overlap for 1 and 8 work segments. That is, in the “One Work Segment” case the test called “waste_time” just once and was able to overlap about 77% of the time for a 128 KB message. But if the test called MPI_Test 8 times during the overlap interval, more than 82% of the wait time was available for computation.

5. Experience with the Metrics
Few useful parallel programs are developed as leisure activities. While enjoyable, most parallel development is demand driven and often time constrained. The metrics presented here can assist both in forming algorithmic approaches and in guiding the developer’s attention toward areas promising improvement and identifying “good enough” solutions and hopeless cases.

The brief practical experience described here attempts to catalog the influence of some of these metrics on two recent and small development efforts.
5.1. TeraSort

The TeraSort test was inspired by the TeraByte Indy sort benchmark at http://www.hpl.hp.com/hosted/sortbenchmark/. TeraSort departs from the original in that it ignores the influence of IO system bandwidth by generating internally all the data to be sorted. This reduces the problem to the sort step alone. The object is to sort $10^9$ records of 100 bytes each.

The TeraSort program implements a simple bucket sort to distribute a portion of the ten billion records to each of the cooperating processes. At the start of the test, each of the N processes owns $10^9/N$ records with randomly generated keys. After building a global histogram of the distribution of key values, each process is assigned a contiguous section of the 10 byte keyspace of approximately $10^9/N$ records. Each record is then sent from the process that generated it to the process that is responsible for its range of the keyspace. Once all the records are distributed, each process sorts its portion using the quicksort algorithm.

The initial implementation of the parallel sort took 28.8 seconds to complete on a 5600-processor complement. (This time included 4 seconds to generate the random data.) The sort time was dominated by the exchange step, which took 11.4 seconds, and the local quicksort step, which took 8.78 seconds.

On average, each processor was required to sort about 2M records. The addressing sequence for the quicksort algorithm is somewhat problematic, as little use is made of spatial cache locality, and large record layouts can cause very poor TLB performance. Quicksort suggests that each processor would complete approximately 40M comparisons. Each comparison is between a record that was recently considered and one that was unlikely to have been accessed recently. Thus, over a record list of 2M entries, each of 100B, the second access is likely to result in both a cache miss and a TLB miss. (Note the span of the SiCortex SC5832 TLB is only 6 MB.) With a combined cost of about 200nS, the TLB miss traffic accounts for almost all of the 8 second sort time. (200nS per TLB miss · 1TLB miss per comparison · 40 · 10^6 comparisons = 8Sec) Clearly, we cannot expect to eliminate all of this time, as cache misses are inevitable given the list size and the 256 KB L2 caches. But we can reduce the number of TLB misses by building a shadow copy of the keylist and sorting the shadow copy. The shadow contained only the top 4 bytes of the key and a pointer to the actual record. Comparisons that showed the top 4 bytes as equal would then dereference the record pointer. This condensed the sort list from 200 MB down to 16 MB. While this was still larger than the span of the TLB, the reduction in the TLB miss rate from almost one per compared to about one for every three comparisons was a substantial reduction in cost. Further, with fewer active pages, the TLB miss was more likely to find the page table entry already resident in the L2, further mitigating the cost of a TLB miss.

As a result, with simple reformattting of the list and some small adjustments to the original qsort algorithm, the quicksort phase of the operation was reduced to 3.75 seconds.

Finally, the 11.4 second exchange time was far greater than the expected value. The algorithm had already been designed to exchange relatively large messages: approximately 10 KB. But the communication bandwidth vs. message size graphs encouraged a larger message size at the expense of some local buffer copies. Further, our expectation of a reasonable amount of overlap suggested an implementation that would perform as much of the buffer copy and list reformattting as possible underneath nonblocking receive operations. Finally, a knowledge of the expected exchange bandwidth between nodes (greater than 50 MB/s for the message size in question) suggested a closer look at the communications pattern: 11.4 seconds to perform a 200 MB exchange was clearly far from optimal. Primarily the most significant improvement came from detailed analysis of message exchanges that uncovered an implicit synchronization barrier. The final cost of the exchange, at 8 seconds, still suggests further improvement is possible.

After tuning the application, directed in part by our metrics, the resulting sort was completed in just 19.65 seconds on 5600 processors.
5.2. Three-dimensional FFT

Certain tomography computations depend on a large, three-dimensional FFT calculation over volumes of approximately $10^9$ points. Multidimensional FFTs are composed of single-dimensional FFT operations repeated in each axial dimension. Therefore, a 1040 x 1040 x 1040 point FFT would require about $3 \cdot 10^6$ 1040 point FFTs. (We chose 1040 points as a good fit for the prime-factor FFT. [6])

The primary focus of the effort was to distribute the calculation and data set among 65 processes and optimize the z-axis exchange prior to the FFT scan in the final (third) dimension. This step involved transposing the three-dimensional volume. The graphs of exchange bandwidth under contention vs. message size suggested that not much was to be gained from messages beyond 128 KB or so, which guided the decision to exchange slabs of the volume sized to fit the optimal message size. As a result, the exchange phase accounts for 4 seconds out of the 15.1 seconds taken for the complete 3D operation.

The overlap plots suggest that total operation time could be reduced as much as 20% by overlapping some operations with the exchange, but this has not yet been attempted.

Test programs

The test programs described here are available at http://www.BigNComputing.org/Big_N_Computing/Test_Programs.html.

References

[1] Reilly M, Stewart L C, Leonard J and Gingold D 2006 SiCortex Technical Summary (SiCortex, Inc.)
http://www.sicortex.com/press/sicortex-tech_summary.pdf
[2] Stewart L C and Gingold D 2006 A New Generation of Cluster Interconnect (SiCortex, Inc.)
http://www.sicortex.com/press/sicortex-cluster_interconnect.pdf
[3] Elspas B, Kautz W H and Turner 1968 Theory of Cellular Logic Networks and Machines (Tech. Report Stanford Research Institute AFCRL-68-0668)
[4] High Performance Computing Challenge Benchmarks 2008 http://icl.cs.utk.edu/hpcc
[5] Thakur R, Rabenseifner R and Gropp W 2005 Optimization of collective communication operations in MPICH International J. High Performance Computer Apps. 19(1):49–66
[6] Temperton C 1985 Implementation of a self-sorting in-place prime factor FFT algorithm J. Computational Phys. 58: 283–299