Apex-Map: A Synthetic Scalable Benchmark Probe to Explore Data Access Performance on Highly Parallel Systems

Erich Strohmaier and Hongzhang Shan

Future Technology Group, CRD, Lawrence Berkeley National Laboratory
One Cyclotron Road, Berkeley, CA 94720
{estrohmaier,hshan}@lbl.gov

Abstract. With the increasing gap between processor, memory, and interconnect speed, the performances of scientific applications on high performance computing systems have become dominated by the ability to move global data. However, many benchmarks in the field of high performance computing focus on measuring the achieved CPU speed in MFlop/s. In this paper, we introduced a novel benchmark, Apex-Map, which focuses on global data movement and measures how fast global data can be fed into computational units. Apex-Map is a parameterized synthetic performance probe and integrates concepts for temporal and spatial locality into its design. By measuring the Apex-Map performance for a whole range of temporal and spatial localities performance surfaces can be generated which can be used to study the characteristics of the computational platforms and which are useful for performance comparison. Results on a vector platform and two superscalar platforms clearly reflect the design differences between these two types of systems.

1 Introduction

Benchmarking of high performance computing has often focused on floating point performance. One prominent example of this is the Linpack benchmark, which is used to rank systems in the TOP500 Project [1]. However, the performance of Linpack is in general not a good performance indicator for real applications. On most platforms, Linpack can achieve over 70% of peak performance while on the same systems many real applications might only achieve substantially lower performances.

With the increasing gap between CPU speed and memory speed, the capability to load and store data locally and globally has become the dominant performance factor for many applications. System designers are spending enormous efforts to design complex memory systems and interconnect networks to increase the data transfer bandwidths and reduce latencies. However, we still lack a quantitative methodology to relate changes in computer architectures to improvements in application performances. There even still is no standard or widely accepted way to measure progress in our ability to access globally distributed data. STREAM [2] is often used to measure memory bandwidth but its use is limited to at the most a single shared memory node. Recently, the HPC Challenge benchmark [3] has included the RandomAccess benchmark, to measure the rate of integer random updates of memory. Unfortunately, this benchmark cannot easily be related to scientific applications and thus does not help much for applications performances.

In this paper, we introduced a novel synthetic memory access probe, called Apex-Map [4], to measure global data access performance. Apex-Map has three main parameters, the global memory size M used, the temporal locality α, and the spatial

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 114–123, 2005.

locality L. Our basic assumption is that an application's global memory access can be approximated by multiple data access streams, each of which can be characterized with the three parameters introduced above. The execution profile of Apex-Map can then be tuned by its set of input parameters to match the data access characteristics of a chosen scientific application. This allows us to use Apex-Map as a performance proxy for the actual codes. An advantage of our synthetic benchmark probe is that due to its simplicity it can easily be run by simulators. This allows its usage in the early stages of architecture design.

Another feature that distinguishes Apex-Map from many other benchmarks is that its input parameters can be varied independent of each other between extreme values. This allows generating continuous performance surfaces to explore the performance effects of all potential values of the characterizing parameters. By examining these surfaces, we can understand how changes in spatial or temporal locality affect the performances of applications and which factors are more important for performance. Moreover, we can compare these performance surfaces across different platforms and explore the advantages and disadvantages of each platform. Most current benchmark suits (HPCC, NAS [5], and SPEC [6]) only contain several application codes or their synthetic benchmarks have other features strongly limiting the scope of performance behaviors they can explore. The results of these application benchmarks provide very good indications how similar applications will perform on a specific platform. However, these benchmarks are not very helpful for other applications, as their performances cannot be related directly to them.

The design details of Apex-Map are described in Section 2. In Section 3, we analyze our results on our three test platforms, two superscalar platforms and one vector platform. We find that the Apex-Map performance results clearly reflect the design differences between the superscalar and the vector platforms. Finally, we analyze the scalability of these three platforms based on the Apex-Map results. Section 4 summarize our results and discusses our ongoing and future work.

2 Implementation

The parallel implementation of Apex-Map uses the same concept as the sequential version [7]. It has the same three main parameters, the global memory size M, the temporal locality α, and the spatial locality L. These parameters are related to our methodology to characterize application performances. Apex-Map assumes that the performance of a data access pattern of an application can be approximated by combining a blocked access to memory with length L with a non-uniform random address determined by α. In Apex-Map a global data-array of size M is evenly distributed across all processes as illustrated in Fig. 1. Data will be accessed in block mode, i.e., L continuous memory addresses will be accessed in succession and the block length L is used to characterize spatial locality. The starting addresses X of these data blocks are computed by using a non-uniform random address generator driven by a power function with the shape parameter α. A power function was chosen as generating function as a simple scale-invariant, one-parameter approximation for the behavior of real applications.

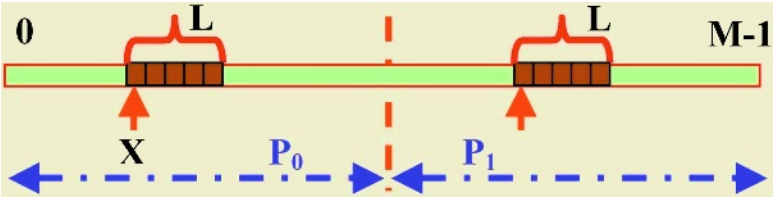

Fig. 1. Apex-Map Data Distribution and Data Access

Table 1. The flowchart of the Apex-Map implementation

Basic Parallel	MPI

Repeat N Times
Generate Index Array
CLOCK(start)
For each Index *i* in the Array
If (not local data)
Get Remote Data
End If
Compute
CLOCK(end)
RunningTime += end – start;
End Repeat
	Repeat N Times
	Generate Index Array
	CLOCK(start)
	For each Index *i* in the Array
	If (local data)
	Compute
	Else
	Generate Remote Request
	End If
	Serve Incoming Requests
	Process Replies
	CLOCK(end)
	RunningTime += end - start
	End Repeat
	CLOCK(start)
	Wait For Finish
	CLOCK(end)
	RunningTime += end – start

The basic flowchart of the plain parallel version of Apex-Map is shown in the left side of Table 1. The indices X are generated and stored in an index array first before the measurement starts. Then, for each index it is tested, if the addressed data resides in local memory in which case the computation proceeds immediately, or if it resides in remote memory in which case it is fetched into local memory first. Apex-Map is designed to measure the rate at which global data can be fed not only into the memory or into cache but into the CPU itself. Therefore, it is essential that an actual computation is performed in the Compute module, which currently is a global sum of all accessed array elements.

The pre-computed indices X are stored in an array of size I. The indices are generated based on a power distribution based random function, which is controlled by the parameters M, L, and α. Generated addresses are shifted so that each process accesses its own memory with the highest probability. The frequency with which remote data access occurs is determined by the temporal locality parameter α. For 256 processes and $\alpha = 1$, the data accesses follow a uniform random distribution and the percentage of remote access is 255/256 (=99.6%). With the increase of temporal locality, the percentage reduces to 0.55% for $\alpha = 0.001$.

The main output of Apex-Map is the average cycles per data access for one process and the aggregate bandwidth in MB/s for the given parameters. The results are directly comparable across different platforms. By running a set of parameters, such as

α = 0.001 to 1.0 and L = 1 to 16384 words, Apex-Map can generate a performance surface to explore the performance effects of temporal locality and spatial locality.

2.1 MPI Implementation

One major non-trivial issue that has not been discussed until now is how the remote access is carried out. The implementation could be highly affected by the available parallel programming paradigm and different programming styles. We assume that the operation for different indices is independent and multiple remote accesses can be executed on the fly at the same time. Our first version was developed using two-sided MPI since it is the most popular and portable parallel programming model available today.

Even if we only consider MPI, there are many implementations thinkable. One possibility is to aggregate the remote requests instead of sending them one by one. We explored several different strategies to do this in depth, but had to conclude, that we ended up only benchmarking our inventiveness for new algorithms to assemble and exchange these messages and our skills to implement them. This approach not only further complicates the code, but also conflicts with our locality concept. By extensively rearranging the order of data-accesses, the actual executed address stream will no longer show the intended features to achieve the given localities. In effect, such rearranging would substantially change the actual localities from the intended localities and would go contrary against our design principles. We therefore decided not to permit such message aggregation and to exchange messages for each remote access.

However, we permit multiple outstanding requests for data and out-of-order processing of the received data. Since in Apex-Map the process numbers for message exchanges are generated based on a non-uniform random access, non-blocking, asynchronous MPI functions are used to avoid blocking and deadlock. Given our non-deterministic random message pattern it was not clear if a scalable implementation of Apex-Map in MPI was possible. However, we succeeded with an efficient and scalable implementation, which shows increasing performance up to 1000s of processors.

Due to the unpredictable communication patterns, the flowchart becomes substantially more complex (see the right side of Table 1) and several MPI related implementation parameters have to be introduced. The first parameter is B, the number of receive buffers allocated, which are needed for each call of MPI_Irecv. It defines the maximum possible number of concurrent outstanding remote data requests per process. Another parameter is SMSG, the maximum number of outstanding send handles defined for MPI_Isend. The last parameter is NSER, with which we limit how many remote requests can be served at one time by our Serving Incoming Requests module. This parameter is especially useful when the remote request distribution is imbalanced. Without this parameter, a process may get completely stuck in serving remote requests for a long time and might not make any progress on its own local computation, which would cause a severe load-imbalance at the end of the global execution.

In summary, there are three kinds of Apex-Map parameters. The first category of parameters includes M, L and α, which are the characteristic parameters of interest. The second category includes general implementation related parameters, including the index array size I and the number of times N the experiment is repeated. The third category includes parameters related to the MPI implementation such as the number of receive buffer B, the number of send handles SMSG, and the maximum number of

served requests in one iteration NSER. Fortunately, experiments on several systems indicate that our default values for all implementation parameters work reasonably well on all of them. The "Wait For Finish" module is needed for MPI because even if a process has finished its own task, it may still need to provide data for other processes and hence cannot complete its execution.

3 Results and Analysis

In this section, we first introduce the three platforms we tested, two superscalar platforms and one vector platform. Then, we analyze the relation of the results of Apex-Map and the PingPong benchmark, as a traditional measure for global communication performance. Finally, we compare the Apex-Map results between the three platforms and examine how the Apex-Map results reflect their architectural differences.

Table 2. Some characteristics of the three platforms used

	CPU	**Memory Bandwidth**	**Network**
Seaborg	IBM Power3, 375 MHz	16 GB/s /node 1 GB/s /processor	IBM Colony-II, 1 GB/s /node
Cheetah	IBM Power4, 1.3 GHz	44 GB/s /node 1.375 GB/s /processor	IBM Federation, 4 GB/s /node
Phoenix	Cray X1, 400 MHz, (800 MHz for vector units)	25.6 GB/s/ MSP	Cray SeaStar 25 GB/s /node

3.1 Three Platforms: Seaborg, Cheetah, and Phoenix

Seaborg is currently the main computing platform of NERSC, a DOE Office of Science user facility at Lawrence Berkeley National Laboratory. It is an IBM Power3 based distributed memory machine. Each node has 16 IBM Power3 processors running at the speed of 375 MHz. The peak performance of each processor is 1.5 Gflop/s. Its network switch is the IBM Colony II, which is connected to two "GX Bus Colony" network adapters per node.

Cheetah is a 27-node IBM p690 system with the IBM Federated switch, where each node has 32 Power4 processors at 1.3 GHz. The peak performance of each processor is 5.2 Gflop/s. Phoenix is a Cray X1 platform consisting of 512 multi-streaming vector processors. Each MSP has four single-stream vector processors and a 2 MB cache. Four MSPs form a node with 16 GB of shared memory. The inter-connect functions as an extension of the memory system, offering each node direct access to memories on other nodes. These two machines are currently operated by the center for Computational Sciences at Oak Ridge National Laboratory. Table 2 lists some main characteristics of these three systems.

3.2 Relationship with PingPong Performance

The PingPong benchmark performance is a well-accepted performance number of parallel systems. In this subsection, we are going to examine the relationships between Apex-Map and PingPong on the above three platforms. The inter-node PingPong performance is measured with one process sending data while the other process is receiving them. The code used was obtained from the Pallas MPI benchmarks [8].

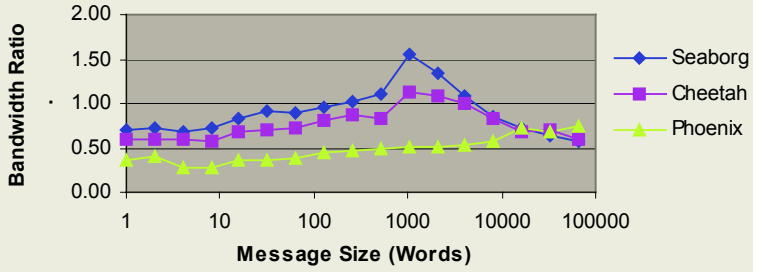

Fig. 2. The performance ratio between Apex-Map (α =1.0) and PingPong

We plot the relative performance of Apex-Map to PingPong in Fig. 2. The inter-node Apex-Map bandwidth per process is obtained with α =1.0 (uniform random data access) and M = 64 Mwords using two Apex-Map processes. Unlike PingPong, Apex-Map measures the performance of non-uniform random access. The communication pattern is unpredictable and the code overhead for it is substantially higher. These factors contribute to the lower performance of Apex-Map when the message size is small. With the increase of message size, the constant overhead becomes less and less important and the Apex-Map performance gets closer to that of PingPong. On Seaborg, Apex-Map performance becomes 60% better than PingPong when message size reaches 1024 words. If we only count the number of exchanged messages and of local memory accesses, Apex-Map should perform 200% better than PingPong since only 50% of the accesses are remote access when α = 1. However, beyond the message size of 1024 words, the performance ratio begins to drop. The main reason here is that Apex-Map measures how fast the data can be fed into the CPU. After remote data arrive in local memory, they further have to be brought into cache and registers for the global sum computation. The effect of this computation can be ignored for smaller messages but is more substantial for large messages on superscalar platforms such as Seaborg. The performance ratio on Cheetah is similar to Seaborg but the MPI-overhead seems to be more severe.

On Phoenix, the performance ratio of Apex-Map to PingPong for smaller messages is even smaller than on the IBM platforms. There also are further differences in the MPI implementations on these two different systems. On Phoenix, using multiple receive buffers in Apex-Map does not improve the performance at all while on Seaborg and Cheetah, the performances benefit substantially from using multiple buffers. Phoenix also does not exhibit the drop in the performance ratio for large messages. Experimental results indicate that the sum computation has only a minor effect on Apex-Map performance on this vector platform.

3.3 Apex-Map Performance

Different from other benchmarks, which usually provide only several performance points, Apex-Map can generate continuous performance surfaces over a whole range of temporal and spatial locality values. These surfaces can be used to study the effects of varying temporal and spatial locality and provide insight into architectural designs. Fig. 3 and 4 show the surface space for α = 0.001 to 1.0, L = 1 to 65536 words on 256

processors for M = 64 Mwords*256 on Seaborg and Phoenix. The Z-axis shows the achieved bandwidth per processes in log-scale.

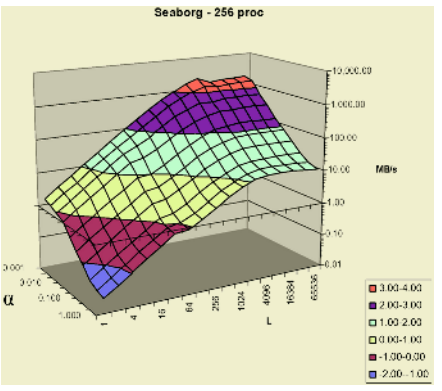

Fig. 3. The achieved bandwidth per process on Seaborg for 256 Processes

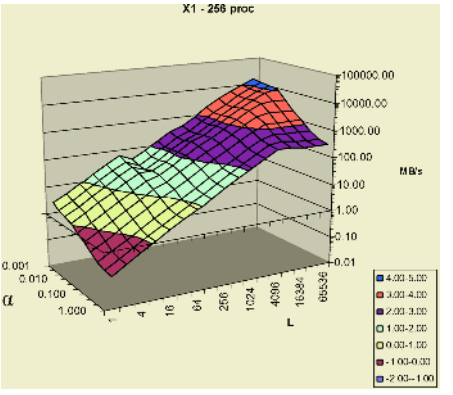

Fig. 4. The achieved bandwidth per process on Phoenix for 256 Processes

Fig. 3 shows that both temporal and spatial localities affect the bandwidth substantially. The worst performance is observed when α = 1 and L = 1, which are the lowest values for temporal and spatial locality. By increasing either the temporal locality or spatial locality, the performance improves. The best performance is obtained when α = 0.001 and L = 4096 Words. Further increasing L does not improve performance. This is mainly because the sum computation on this platform is less efficient for very large messages. Beyond L = 4096 spatial locality has only minor influence on performance while temporal locality α still has a large influence. If we look at an intermediate performance level such as 1 MB/s, we see that the temporal locality and spatial locality can be substituted by each other to some degree. To achieve 1 MB/s at high temporal locality of α = 0.005, a very low spatial locality of L = 1 is sufficient. With decreasing temporal locality (increasing α), a higher spatial locality of up to L = 85 is needed to maintain this performance. The performance characteristics of Cheetah are very similar to Seaborg.

Fig. 5 shows the performance ratio between Cheetah and Seaborg. From Table 2 we see that the ratio of processor speeds between these two systems is 3.47, the ratio of local memory bandwidth is 1.375, and of network bandwidth is 4. For high temporal locality or high spatial locality the performance ratio of 2-4 seems to be dominated by the ratio of the respective memory bandwidth. For low localities, the performance ratio between these two systems is in the range of 6-8 and thus higher than any ratio of simple architectural parameters. In this locality range, performance is dominated by a large number of very short messages. The details of the MPI implementation as well as the cross-section bandwidth of the interconnect can be expected to have a large influence on performance in this corner of low localities where it will be notoriously difficult to achieve high absolute performance.

Fig. 4 shows the performance surface for the Cray X1 for which the effects of increasing spatial locality are significant even for values of L beyond 4096. Spatial locality affects the performance in general much stronger. For example, on Cheetah, in order to maintain the bandwidth around 10 MB/s, if we reducing the temporal lo-

cality α from 0.001 to 1, the spatial locality needs to increase 128 times. On Phoenix, it only needs to increase 16 times. We also notice that when L changes from 32 to 64, the performance drops. This is an effect of the MPI implementation on the Cray X1. When the message size becomes larger than 32 words or 256 bytes, communication in MPI will switch from eager mode to rendezvous mode and the implementation overhead increases.

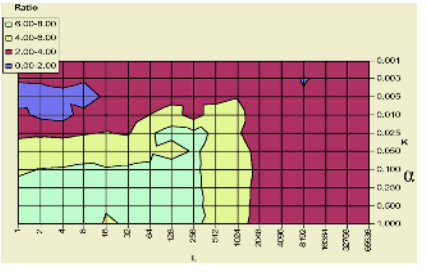

Fig. 5. The bandwidth performance ratio between Cheetah and Seaborg

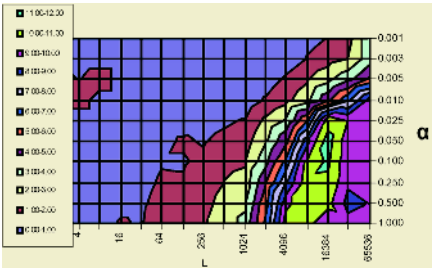

Fig. 6. The performance ratio between Phoenix and Cheetah

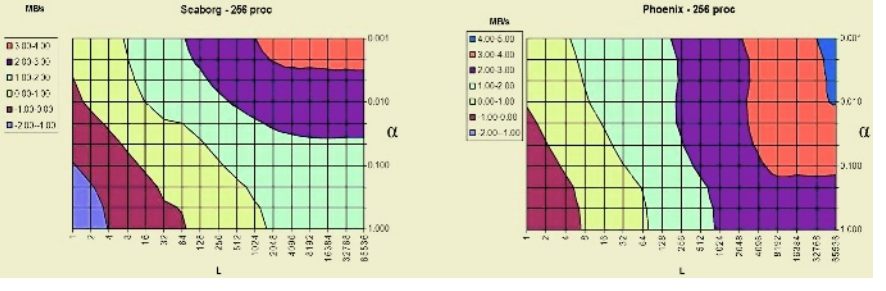

Fig. 7. Contour plots of the performance surfaces for Seaborg and Phoenix

To compare the performance surface for the superscalar IBM systems with the Cray vector system we put contour-plots of Seaborg and Phoenix next to each other in Fig 7. For the IBM systems, the area of highest performance is of rectangular shape and clearly elongated parallel to the spatial locality axis while for the Cray system it is elongated parallel to the temporal locality axis. The IBM system can tolerate a decrease in spatial locality more easily but is much more sensitive to a loss of temporal locality. This reflects the elaborate cache and memory hierarchy on the individual nodes as well as the global system hierarchy which also heavily relies on reuse of data as the interconnect bandwidth is substantially lower than the local memory bandwidth. The Cray system can tolerate a decrease in temporal locality much better but is sensitive to a loss in spatial locality. This reflects an architecture which depends very little on local caching of data and an interconnect bandwidth equal to local memory bandwidth. To see such a clear signature of the Cray architecture is even more surprising considering that we us an MPI based benchmark, which does not fully exploit the capability of this system. The lines of equal performance on the Cray system are in general more vertical than diagonal as with the IBM system, which further con-

firms our interpretation. These differences in our performance surfaces overall clearly reflect the different design philosophies of these two different systems and demonstrate the utility of our approach.

The performance ratio between Phoenix and Cheetah is shown in Fig. 6. Interestingly, when the spatial locality is poor or temporal locality is high, the vector processor X1 delivers less performance than the super-scalar processor Power4. In these cases, performance is dominated either by short MPI messages for which the Power 4 processor has the clear advantage of a much faster scalar processor or by very localized memory accesses for which the Power4 can effectively use its cache hierarchy. In this locality range, the Cray X1 can also not show its true potential with our current MPI based benchmark implementation. A shmem or UPC implementation might change this. The X1 shows the clearly better performance when spatial locality becomes high, especially in the area with poor temporal locality (the bottom-right corner). In the best case, it can deliver 12 times better performance than Power4 platform. Performance in this corner is dominated by the exchange of many long messages which requires an interconnect network with a large cross-section bandwidth.

4 Conclusion and Future Work

In this paper, we describe a novel synthetic performance probe, Apex-Map. It focuses on measuring the performance of global data movement and has three main parameters, the global data size M, the temporal locality α, and the spatial locality L. We assume that the performance of the data accesses of an application can be approximated by a generic, non-uniform random, block-access to global data defined by the parameters M, α, and L. We have run multiple experiments with Apex-Map on two superscalar platforms and one vector platform and have generated continuous performance surfaces, which enable us to study the effects of spatial and temporal locality on performance. The initial results on these platforms show that Apex-Map can be used to compare efficiency and scalability across different platforms and the performance surfaces generated by Apex-Map clearly reflect the design differences between these platforms.

Our first parallel implementation of Apex-Map is based on the most common parallel programming model, MPI. Currently we are implementing Apex-Map in other popular or emerging programming models, such as SHMEM and UPC, to study the effects of different programming paradigms and their relation to spatial and temporal locality. More importantly, we are also investigating methods to characterize parallel applications with the Apex-Map parameters. In our earlier work, we have successfully characterized several sequential scientific kernels [7] this way. Such a characterization allows us to use Apex-Map as a performance proxy for real scientific applications.

References

1. http://www.top500.org
2. STREAM: Sustainable Memory Bandwidth in High Performance Computers, http://www.cs.virginia.edu/stream/

3. HPC Challenge Benchmark, http://icl.cs.utk.edu/hpcc/
4. Apex-Map: Application Characterization-Memory Access Probe, http://ftg.lbl.gov
5. NAS Parallel Benchmarks, http://www.nas.nasa.gov/Software/NPB/
6. SPEC, http://www.spec.org/
7. E. Strohmaier, Hongzhang Shan, "Architecture Independent Performance Characterization and Benchmarking for Scientific Applications", International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. Volendam, The Netherlands, Oct. 2004
8. Pallas MPI Benchmarks, http://www.pallas.com/e/products/pmb/