Apex-Map: A Global Data Access Benchmark to Analyze HPC Systems and Parallel Programming Paradigms

Erich Strohmaier, Hongzhang Shan

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

Abstract

The memory wall and global data movement have become the dominant performance bottleneck for many scientific applications. New characterizations of data access streams and related benchmarks to measure their performances are therefore needed to compare HPC systems, software, and programming paradigms effectively. In this paper, we introduce a novel global data access benchmark, Apex-Map. It is a parameterized synthetic performance probe and integrates concepts for temporal and spatial locality into its design. We measured Apex-Map performance for a whole range of temporal and spatial localities on several advanced processors and parallel computing platforms and use the generated performance surfaces for performance comparisons and to study the characteristics of these different architectures. We demonstrate that the results of Apex-Map clearly reflect many specific characteristics of the used systems. We also show the utility of Apex-Map for analyzing the performance effects of three leading parallel programming models and demonstrate their relative merits.[1]

1. Introduction

During the last decades the memory wall between the peak performance of microprocessors and their memory performance has become the prominent performance bottleneck for many scientific application codes. Despite this development many benchmarking efforts in scientific computing have in the past focused on measuring the floating-point computing capabilities of a system. One prominent example is the Linpack benchmark, which is used to rank systems in the TOP500 Project [1]. These benchmarks can provide guidance for the performance of some compute intensive applications, but fail to provide reasonable guidance for any memory-bound applications. This situation has increased the interest in new concepts and benchmarks for describing and measuring the data access capabilities of modern parallel computer systems.

In this paper, we introduced a novel synthetic memory access probe, called Apex-Map [2], to measure global data access performance. Apex-Map is designed based on parameterized concepts for temporal and spatial locality and generates a global data access stream according to specified levels of these measures of locality. In our current approach we focus solely on the performance effects of data access and ignore any potential situations where the details of the actual computation or data dependencies determine overall performance. We also assume that the execution of an application can be broken down in different phases with different characteristics. Further, we assume that combining multiple data access streams, each of which can be characterized independently, can approximate the global memory access of each computational phase. The characterization currently used for Apex-Map has three descriptive parameters, the global memory size M used, a measure α for temporal locality, and a measure L for spatial locality. Ideally the execution profile of Apex-Map can be tuned by this set of input parameters to match the data access char-

SC|05 November 12-18, 2005, Seattle, Washington, USA

acteristics of a chosen scientific application. This would allow using Apex-Map as a performance proxy for the actual application code.

Other synthetic benchmarks measuring data movement include STREAM [3], which measures sustainable local memory bandwidth. It focuses on highly regular data access patterns for sequential processes, which limits its use to single shared memory nodes. Using Apex-Map with asymptotically large L parameters on single processors should generate equivalent performance numbers to STREAM. Recently, the HPC Challenge benchmark [4] has included the RandomAccess benchmark, to measure the rate of integer random updates of memory. RandomAccess focuses on uniform random global data access and could be seen as an opposite data access benchmark compared to STREAM. Using α=1 and L=1 for Apex-Map should allow us to generate a very similar data access pattern, as it is also a uniform random single word access stream. The main difference is, that RandomAccess uses global write operations and Apex-Map uses only global read operations. MAPS [5] also measures the bandwidth for a variety of kernels on a specific machine but only for single processors.

One 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 to generate continuous performance surfaces which can be used 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 performance and which factors are more important on a specific system. Moreover, we can compare these performance surfaces across different platforms and explore the advantages and disadvantages of each platform. Most current benchmark suites (HPCC [4], NAS [6], and SPEC [7]) only contain several application codes which strongly limits the scope of performance behaviors they can explore. The results of these application benchmarks provide very good indications how similar applications will perform. However, these benchmarks are not very helpful for other applications.

With the increasing complexity and concurrency levels of parallel systems the utility and performance of different parallel programming models have become of major interest. Parallel programming paradigms differ from each other substantially in their ability to exploit the potential performance of the underlying hardware and in their implementation overheads. Apex-Map is based on a synthetic problem and can easily be implemented in different parallel programming models. So far we have implemented Apex-Map with MPI, the most popular parallel programming model, and two less used but promising programming models, SHMEM and UPC. In MPI, each process has its own address space. The communication between processes is carried out by send-receive pairs. In SHMEM, each process also has its own address space, but they are symmetric and communication is one-sided. UPC provides a shared address space to the programmer. The shared data can be accessed by regular load and store operations. In this paper, we are going to examine how the achievable performance in Apex-Map is affected by these programming models relative to temporal locality and spatial locality.

In the following section, we describe the details of both the sequential and the parallel implementations of Apex-Map. The performance of several widely used microprocessors is analyzed using Apex-Map results in Section 3. Section 4 focuses on the performance comparison of several currently deployed high performance computing platforms, showing the strength and weakness of each platform. In section 5 we examine the performance effect of different programming models. Finally, we summarize our results and discuss our ongoing and future work.

2. Apex-Map Implementation

2.1 Principals

The synthetic memory access probe Apex-Map is designed based on parameterized concepts for temporal and spatial locality. It uses a blocked data access to a global array of size M to simulate the effects of spatial locality. The block length L is used as measure for spatial locality and L can take any value between 1 (single word access) and M. A non-unifom random selection of starting addresses for these blocks is used to simulate the effects of temporal locality. A power function distribution is selected as non-uniform random distribution and non-uniform random numbers X are generated based on uniform random numbers r with the generating function $X=r^{1/\alpha}$. The characteristic parameter α of the generating function is used as measure for temporal locality and can take values between 0 and 1. A value of α =1 generates uniform random numbers while small values of α generate random numbers centered towards the starting address of the global data array.

Apex-Map uses the same three main parameters for both the sequential version and parallel version. These are the global memory size M, the measure of temporal locality α, and the measure of spatial locality L. These three parameters are related to our methodology to characterize applications. For the parallel version, an important question is whether the effects of temporal locality and process locality should be treated independent of each other by implementing different parameters and executions models for them, or if a global usage of the temporal locality model can provide a sufficient first approximation of the data access behavior of scientific application kernels. For the parallel execution of a single scientific kernel any method to divide the problem to increase process locality should also be usable to improve temporal locality on a sequential execution. At the same time any algorithm with good temporal locality should on a global parallel implementation in turn exhibit good process locality. Thus the only cases where process and temporal localities can differ substantially, would be algorithms for which the problem solved in each process is different from the problem between processes. Examples would be e.g. the embarrassingly parallel execution of a kernel with low temporal locality by running multiple copies of the individual problem with different parameters at the same time and thus generating high process locality. For simplicity reasons we therefore decided to treat temporal and process locality in a unified way by extending the sequential temporal locality concept to global memory access in the parallel case. One important implementation detail here is that we access the global array only with load operations, which avoids any possibility of race conditions for memory update operations. Comparisons between APEX-Map performance and other known benchmarks such as Ping-Pong can be found in [8].

Temporal locality in actual programs can be caused by different reasons. In some cases (e.g. a blocked matrix-matrix multiplication) variables are not used for long periods of time but once they are used they are reused in close time proximity multiple times. Overall, however, all variables are accessed with equal frequency. In other codes (e.g. matrix-vector multiplication) some variables are simply accessed more often than others (in our example the elements of the original vector). Exploiting these different flavors of temporal locality in the sequential case requires different caching strategies such as dynamic caching if our first example and static caching in our second example. In practice dynamic caching is used almost exclusively as it tends to work reasonable well also in many (but not all!) situations, which conceptually would require static caching. APEX-Map clearly uses more frequent accesses to certain addresses to simulate the effects of temporal locality. In the parallel case the difference between these flavors becomes more important as placement (and possibly sharing) of data and their affinity to processes becomes a performance issue. In APEX-Map we assume that each process accesses certain variables more often and that these variables can be placed in memory closer to this process. We do not address the different question how to address and exploit temporal locality in cases where overall all variables are accessed with equal frequency and thus data placement is an ineffective strategy for exploiting temporal locality. APEX-Map also assumes that sharing of variables is not a performance constraint, as we are only reading global data but do not modify them. More discussions about the rational of the APEX-Map parameters can be found in [9].

2.2 Sequential Implementation

Apex-Map uses indexed access to simulate random access streams as illustrated in Fig. 1. The random starting addresses (X) are aligned by length L and generated by a power distribution function whose shape is controlled by the temporal locality parameter α, which can take values between 0 and 1. Once the starting address has been accessed, the following L-1 continuous addresses will also be accessed in a stride 1 loop.

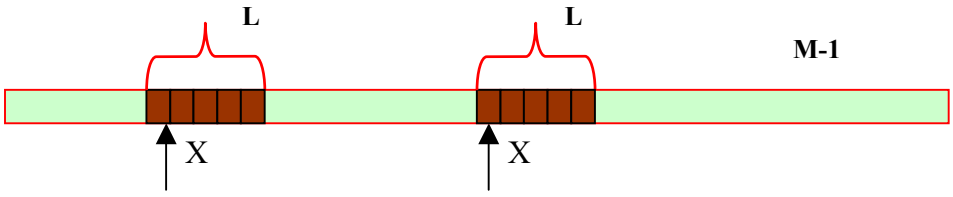

Fig. 1: The Data Access Model of Apex-Map

The starting addresses cannot be dynamically generated during execution since the time needed to generate them is too long compared with the memory access times. Therefore the addresses are computed in advance and stored in an index buffer. The left side of Table 1 shows the core part of the sequential Apex-Map code: A compute module is essential since Apex-Map measures the capability of the system to feed global

Table 1: The Outline of the Apex-Map Implementations

Sequential	Parallel	Compute Module
Repeat N times *InitIndexArray_seq(I);* *CLOCK (start);* *for(j = 0; j < I; j++) {* *pos = ind[j];* *compute(data[pos : pos+L-1]);* *}* *CLOCK (end);* *RunningTime += end – start;* *End Repeat*	*Repeat N times* *InitIndexArray_par(I);* *CLOCK (start);* *for(j = 0; j < I; j++) {* *pos= ind[j];* *if (remotel data)* *GetRemoteData* *endif* *compute(data[pos : pos+L-1]);* *}* *CLOCK (end);* *RunningTime += end – start;* *End Repeat*	*compute(data[pos : pos+L-1);* *for(k = 0; k < L; k++) {* *sum += data[pos+k];* *}*

data into the CPU not only the cache or memory so that the reported performance reflects the effect of whole memory architecture. The details of the computation chosen for the compute module should however not influence performance. The current operation in this module shown in the third column of Table 1 is the global sum of all accessed array elements. The effect of the temporal locality parameter α on cache hit and miss rates is illustrated in Fig. 2 for a ratio of cache size to used memory of 1:256.

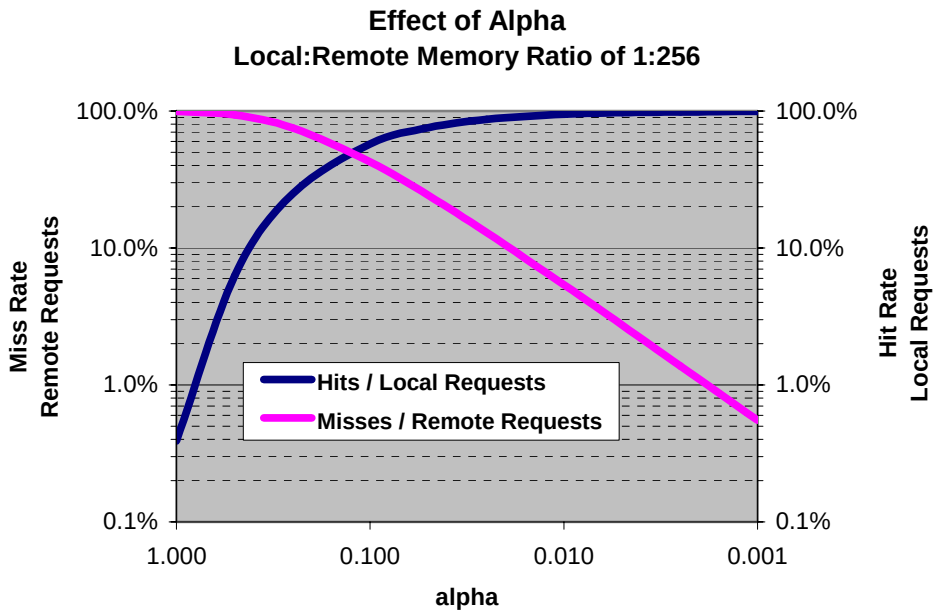

Fig. 2: Effect of alpha on hit/miss rates and ratio of local to remote data requests for a local:remote memory ratio of 1:256

The performance measurements reported by APEX-Map are the average access time per memory access in cycles and the corresponding bandwidth in MB/s. In the sequential version Apex-Map also reports the standard deviation for the access times. The parameters I (buffer size) and N (repetitions) are auxiliary parameters and should be set to appropriate values to achieve reliable results. On some platforms due to inefficiencies of the compilers, the inner loop needs to be manually unrolled several times to achieve best performance. Therefore, in our sequential experiments, the inner loop has been unrolled 1, 2, 4, or 8 times and only the best results are reported.

2.3 Parallel Implementation

The parallel implementation uses the same concept as the sequential version. Its outline is shown in Table 1. In the parallel implementation, the global data is evenly distributed across all the processes in a block distribution. Each process computes its own global address-stream using the same power distribution function. However, these global addresses will be adjusted based on the rank of the process by shifting the computed address with the beginning address of the processes global array portion. This ensures that each process accesses its own memory with the highest probability. 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. The frequency with which remote data access occurs is mainly determined by the temporal locality parameter α. For example, for 256 processes and α =1 the data accesses follow a uniform random distribution and the percentage of remote access is 255/256 (=99.6%) (Fig 2). With the increase of temporal locality, the percentage reduces to 0.55% when α = 0.001.

The major implementation issue for the parallel version of Apex-Map is the question how to fetch remote data. Conceptually we do allow out of order execution, which presents a major possibility for optimizing the execution. However, how remote data can be fetched and if an effective out of order execution model can be implemented depends heavily on the parallel programming model used. Currently, Apex-Map is implemented with three parallel programming models: MPI, SHMEM, and UPC [10]. MPI2 would also allow one-sided message passing similar to the functionality of SHMEM, but we have not yet had the opportunity to implement APEX-Map in it. The major differences in our implementations for these three parallel programming paradigms are shown in Table 2.

Table 2: The MPI, SHMEM, and UPC implementation

MPI	SHMEM	UPC
Repeat N times *InitIndexArray_par(I);* *CLOCK (start);* *for(j = 0; j < I; j++) {* *pos= ind[j];* *if (remotel data)* *Generate_Remote_Request()* *else* *Compute(data[pos: pos+L-1])* *endif* *Serve Incoming Requests()* *Process Replies()* *}* *CLOCK (end);* *RunningTime += end – start;* *End Repeat* *CLOCK (start)* *WaitForFinish()* *CLOCK (end)* *RunningTime += end – start*	*Repeat N times* *InitIndexArray_par(I);* *CLOCK (start);* *for(j = 0; j < I; j++) {* *pos= ind[j];* *if (remotel data)* *SHMEM_DOUBLE_GET(p)* *Compute(p[0: L-1])* *else* *Compute(data[pos: pos+L-1]);* *endif* *}* *CLOCK (end);* *RunningTime += end – start;* *End Repeat*	*Repeat N times* *InitIndexArray_par(I);* *CLOCK (start);* *for(j = 0; j < I; j++) {* *pos= ind[j];* *if (remotel data)* *// Method 1* *UPC_MEMGET(p)* *Compute(p[0 : L-1])* *// Method 2* *Compute(data[pos:pos+L-1])* *else* *Compute(data[pos:pos+L-1])* *endif* *}* *CLOCK (end);* *RunningTime += end – start;* *End Repeat*

Due to the one-sided communication, fetching the remote data in SHMEM is straightforward and is implemented by calling the SHMEM_DOUBLE_GET function, which results in messages of size L. For UPC, there are two options. Like in SHMEM, in UPC, a process can also call a function UPC_MEMGET to bring all the remote data into its local memory at once. Another way is to take advantage of the shared memory model by allocating the global data array in shared mode and accessing it directly.

The MPI implementation is much more complicated mainly due to its two-sided communication model. Since the message addresses in Apex-Map are generated based on a non-uniform random access, non-blocking, asynchronous MPI functions are used to avoid blocking and deadlock. Not only does a process have to send requests for data out, but also it has to prepare to receive the replies and process them. Meanwhile, a process also has to check for incoming requests to serve data to the other processes. Therefore,

even if a process has finished its own work, it still needs to wait for all others to finish in case some other processes may request data from it. All these operations can incur substantial overheads.

There are also different implementations imaginable. 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 however 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 in the end 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.

3. Performance Analysis for different Processors

In this section, we are going to analyze the performance of some widely used processors. Different from other benchmarks, which usually provide only several performance points, Apex-Map can generate continuous performance surfaces (performance maps) 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. The typical values we use to generate these maps are α = [0.001 to 1.0], L = [1 to 65536] words, and M = 64 MWords (512 MB). In cases of insufficient memory, half of the available memory size is used. α = 1.0 means the global data access follows uniform random access while α=0.001 indicates the accessed data inherits very high temporal locality. Therefore increasing the value of α will reduce the temporal locality. On the contrary, increasing the value of L will increase the spatial locality. We have obtained results for IBM PowerPC 440 (700 MHz), Power3 (375 MHz), Power4 (1300 MHz), Power5 (1.9 GHz), Intel Itanium2 (1.5 GHz), Xeon (2.2 GHz), AMD Opteron (2.2 GHz), Cray X1 (800 MHz), and NEC SX6a (565 MHz).

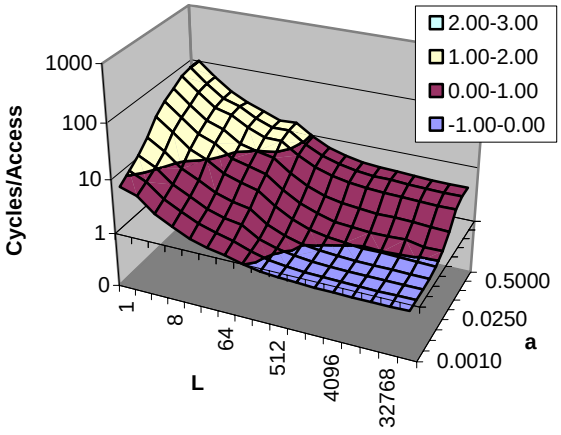

Fig. 3: The performance surface for Itanium II

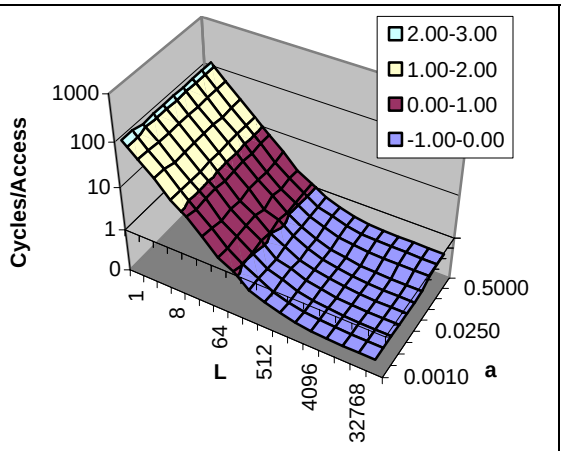

Fig. 4: The performance surface for Cray X1

3.1 Superscalar processor vs. Vector processor

As an example, Fig. 3 and Fig. 4 display the performance surfaces for a superscalar processor, the Itanium II, and a vector processor, the Cray X1. The Z-axis shows average cycles per data access in log-scale. The performance surfaces of these two kinds of processors differ significantly from each other. For the Itanium, both the temporal locality and the spatial locality affect the performance substantially. The worst performance occurs when both temporal locality and spatial locality reach the lowest point we have tested (α = 1, L = 1), for which around 100 cycles are needed per data access. Increasing either the temporal locality or spatial locality reduces the average number of cycles per data access needed. The Itanium II needs only 0.77 cycles for α = 0.001 and L = 2048. Further increasing the spatial locality does not improve per-

formance much. Further we can find that temporal locality and spatial locality can be substituted for each other to some degree. The line of equal access speed of Z=10 illustrates this as it is almost diagonal. A reduction of temporal locality could be compensated by an increase in spatial locality and vice versa.

For the Cray X1, the spatial locality affects the performance much more prominently. The X1 can easily tolerate a decrease in temporal locality but is very sensitive to the loss of spatial locality (and the corresponding reduction in vector-length). The differences between these two kinds of performance surfaces clearly reflect their different design concepts. Superscalar processors depend on the elaborate cache and memory hierarchies to take advantage of both temporal and spatial locality while vector processors typically have sophisticated memory access sub-systems which makes them insensitive to the lack of temporal locality and programmers therefore only have to focus on expressing and exploiting spatial locality.

3.2 Uniform Random Access (α = 1.0)

For examining the performance of Apex-Map on a variety of processors we have to focus on more specific parameters values, since it is difficult to visualize all the data. We show results for two values of α, α = 1.0 (uniform random access) and α = 0.01 (high temporal locality). Fig. 5 shows the obtained bandwidth when the data access follows uniform random access pattern (α =1). The two vector processors, Cray X1 and SX6a, deliver far superior bandwidth to the other processors once spatial locality is above 16. For L = 65536, the SX6a achieves a 9 times higher bandwidth than Power5 though its CPU clock speed is more than 3 times slower. When the spatial locality is relatively lower (L <= 16), the performance of the two vectors platforms falls behind of the Itanium2 and Opteron, close to the Power5 and Xeon, but still better than the older Power4, Power3, and PowerPC. The bandwidth obtained on the Power3 is the lowest, at least 5 times worse than the others for smaller values of L. With the increase of spatial locality, its performance improves and it catches up with the PowerPC at L = 1024, before falling off again.

The performances of the superscalar processors are shown in Fig. 6 with a linear axis to allow a more detailed view of how their performances compare with each other. The Opteron performs best before L reaches 2048. Then, its performance drops like a staircase and finally descend to the Power4 level. The Itanium2 performs very close to Opteron for small L. However, with the increase of spatial locality, the gap between them becomes larger. In the end, its performance exceeds the Opteron due to the performance plunge of the Opteron. For lower spatial locality, the performance of the Power5 is behind the Opteron and Itanium but close to the Xeon and Power4. It excels when the value of L is been increased to 128 and achieves highest bandwidth after L becomes larger than 4096. The performance of the Xeon and Power4 are close to each other. They also share a similar pattern, going up with the increase of L and then falling down after 1024. The PowerPC performs a little better than the Power3.

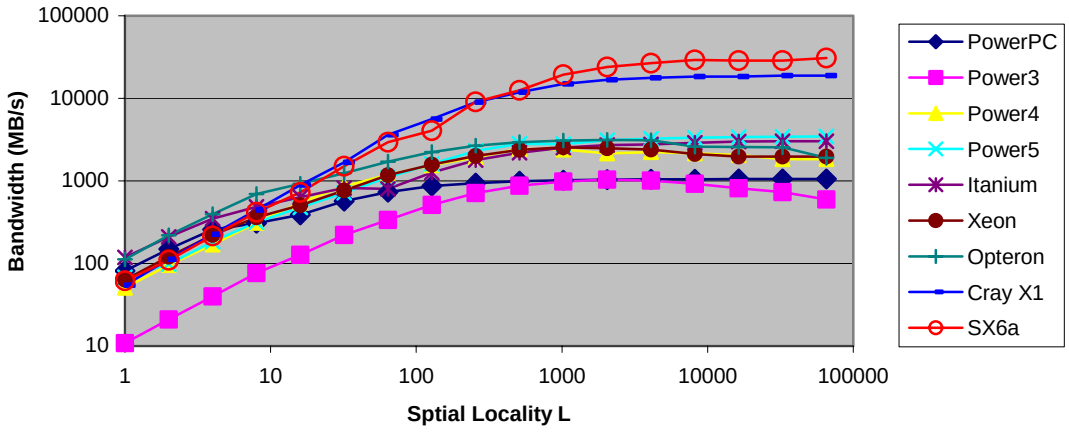

Fig. 5: The achieved bandwidth (MB/s) for all processors when α = 1.0.

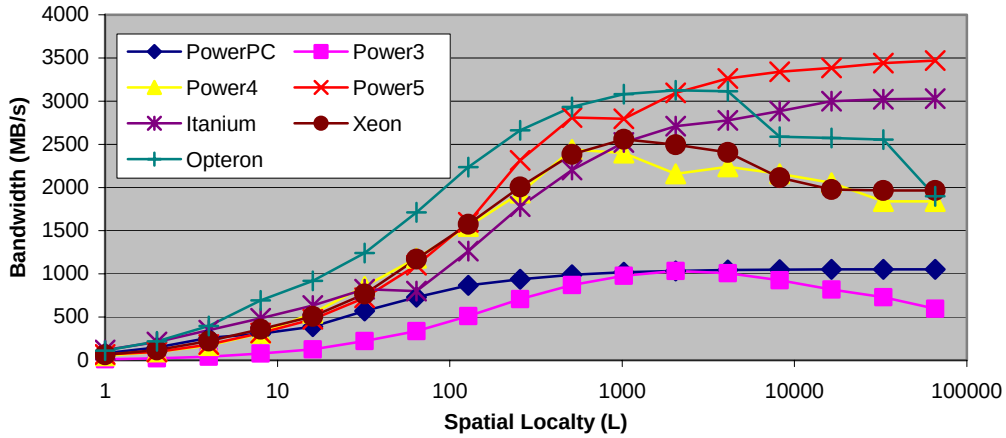

Fig. 6: The achieved bandwidth for superscalar processors when α = 1.0.

3.3 Higher Temporal Locality Case (α = 0.01)

The delivered bandwidths for α = 0.01 are displayed in Fig. 7. Compared with α = 1.0, the data access has much higher temporal locality. Once again, the two vectors processors achieve asymptotically higher performance than any superscalar processors. This time the performance of the vector processors are similar to each other. For lower spatial locality (and vector length), all the superscalar processors outperform the vector processors. This can be attributed to the complicated hierarchical design of memory systems of superscalar processors, which enables them to efficiently reuse the data due to its high temporal locality. With the gradual increase of the spatial locality, the vector processors performance improves very quickly and exceeds all superscalar processors.

The performances of the superscalar processors are plotted on a linear scale in Fig. 8. The Opteron follows the same pattern as in the α = 1.0 case. It performs best up to L = 512 but is exceeded by the Power5 thereafter. This time the Itanium2 gets the highest bandwidth for large values of L. The performance of the Power4 is still in the middle among these processors. The performance of the Xeon drops sharply when spatial locality increases beyond 4096. The older Power3 and the PowerPC still show the lowest performance, but this time, the Power3 performs better than the PowerPC possibly due to its larger cache sizes.

In a summary, vector processors behave very differently from superscalar processors. They are sensitive to the spatial locality and insensitive to temporal locality while superscalar processors are sensitive to both temporal locality and spatial locality, which often can substitute each other. Current vector processors can deliver far superior bandwidth to scalar processors for data streams with high spatial locality. On the other hand, for data stream with low spatial locality some superscalar processor show superior performance.

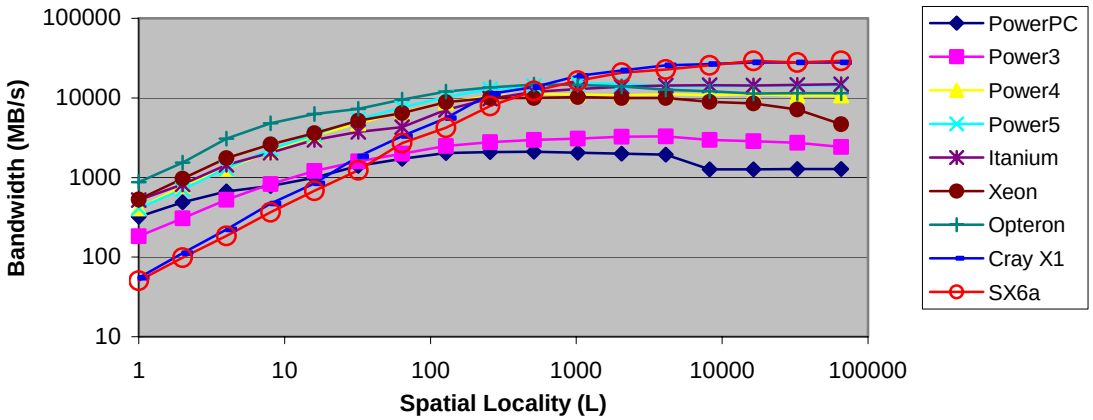

Fig. 7: The achieved bandwidth for all processors for α = 0.01.

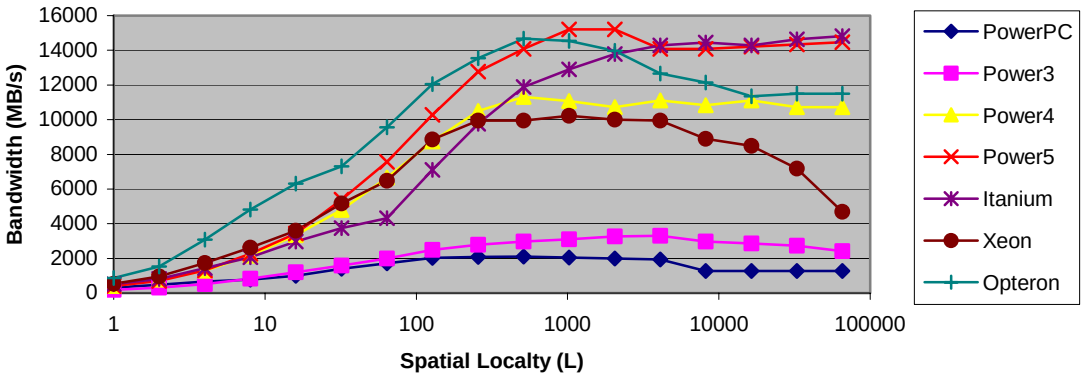

Fig. 8: The achieved bandwidth for superscalar processors for α = 0.01.

Among the superscalar processors, the Opteron often provides higher bandwidth than the Intel and IBM processors when the data stream does not have very high spatial locality. For high spatial localities the Power5 or the Itanium2 perform best. Which of these two is better really depends on the characteristics of the data access stream. The Power4 and Xeon usually follow behind closely, but in several cases when spatial locality is low, their performance may even outperform the Power5 or Itanium. The older Power3 and the low-power PowerPC provide the lowest bandwidth.

4. Performance Analysis for HPC Multiprocessor Platforms

In this section, we are going to analyze the performance of several high-performance computing platforms that are currently being deployed either as production platform or as cutting-edge test systems. Table 3 lists some features of these systems. The results are obtained using the MPI implementation as it is the only commonly available parallel programming paradigm. We use the same range of values for the temporal and spatial localities as in the sequential version, α = 0.001 to 1.0 and L =1 to 65536 Words. The global data size M becomes 64Mwords * P so that each process uses a local memory size of 64Mwords. P is the number of processes and by default we present results for 256 processes. The output of Apex-Map we used is the obtained bandwidth (MB/s) per process. Next we are going to compare several interesting cases.

Table 3: Some Features of the Systems Used

Name	CPU	CPUs/ Node	Network	Memory Bandwidth	Site
Seaborg	Power3 375 MHz	16	IBM Colony-II 1GB/s/node	16GB/s/node 1GB/s/processor	NERSC
Cheetah	Power4 1.3 GHz	32	IBM Federation 4GB/s/node	44GB/s/node 1.375GB/s/processor	Oak Ridge
BG/L	PowerPC440 700 MHz	2	3-D Torus, 2.1GB/node Global Tree, 2.1GB/node Global Interrupt	5.5 GB/s/processor	Argonne
Cray X1	Cray X1, 800 MHz	4	Cray 2-D Torus, 25GB/s/node	25.6GB/s/MSP	Oak Ridge
Jacquard	Opteron, 2.2 GHz	2	Infiniband, 1GB/s/node	6.4GB/s/processor	NERSC
Thunder	Itanium2, 1.4 GHz	4	Quadrics, 1 GB/s/node	6.4GB/s/processor	LLNL
Altix	Itanium2, 1.5 GHz	2	Fat-tree, NUMALink 6.4GB/link	6.4GB/s/processor	Oak Ridge

4.1 Vector Platform vs. Superscalar Platform

As a first example we analyze the performance surfaces for two of the systems, Cheetah in Fig. 9 and Phoenix in Fig. 10. Cheetah is a superscalar platform based on the Power4 while Phoenix is the Cray X1 vector platform. Again, the parallel performance surfaces exhibit similar differences between vector and superscalar platforms as for the sequential results. For Cheetah, 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.

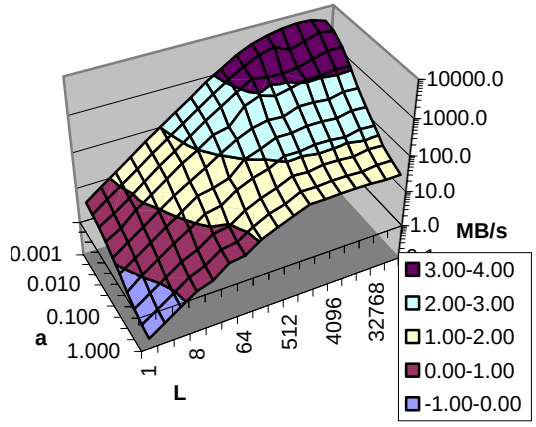

Fig. 9: The performance surface for Cheetah

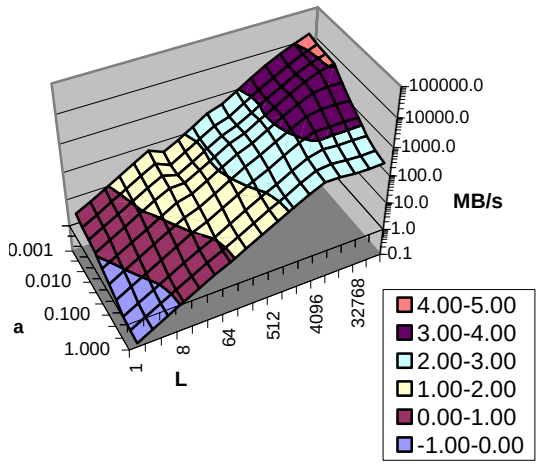

Fig. 10: The performance surface for Phoenix

4.2 Overall Comparison

Fig. 11 lists the achieved per processor bandwidth on all platforms when $\alpha = 1.0$. Seaborg delivers the worst bandwidth per processor across all the values of spatial locality, almost an order of magnitude lower. For high spatial locality, the vector platforms deliver far superior bandwidth than other platforms, benefiting from both the vector computing units and underlying faster interconnects. BlueGene/L performs a little better than Altix but very close to each other. Jacquard is an Opteron cluster connected by Infiniband switch. Each node has two processors. Its performance suddenly drops when L changes from 1024 to 2048. This is directly related with the buffer management in MPI implementation. Infiniband requires that memory must be pinned before communication is initiated. The MPI communication on Jacquard uses different protocols depending on the size of the message. The threshold is 1536 words. If message size smaller than this value, it copies data in and out of pre-allocated buffers that are registered by MPI at startup. For large messages, the data will be sent directly from the user-specified buffer on the sender to the user-specified buffer on the receiver. MPI has to dynamically register the user buffer. Dynamic registration is very expensive causing the performance plummet. But its performance is still comparable with BlueGene and Altix and much better than Cheetah.

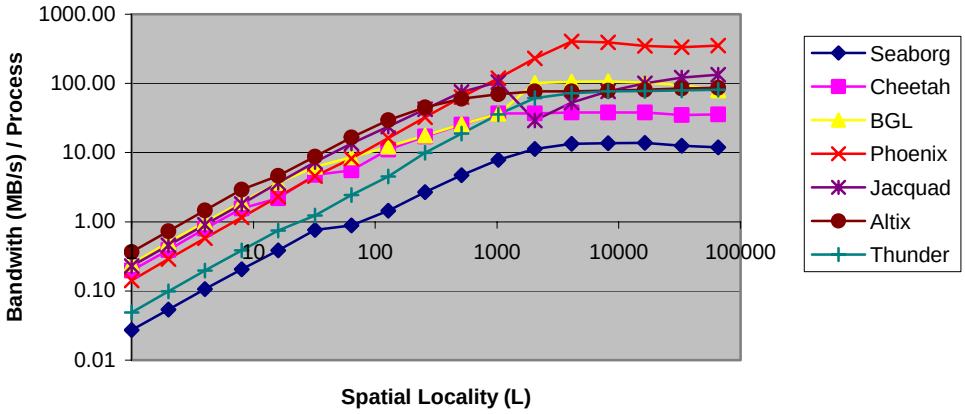

Fig. 11: Achieved bandwidth on all platforms for uniform data distribution using 256 processors.

For lower spatial locality, Altix performs best in most of the cases. Following Altix, Jacquard, Cheetah, and Bluegene are very close and Phoenix is a little behind. Overall, the bandwidths delivered on all platforms in this case are very low.

4.3 Scaling Study

Finally, let's take a look of the scalability of all platforms. We focus on the low temporal locality case (α = 1.0), the most difficult case. Scaling will become much easier for high temporal locality cases. Fig. 12 and Fig. 13 display the average bandwidth delivered per processor on all platform when L=4096 and L= 1 individually. Therefore, on a perfect scaling platform, the per processor bandwidths should be a horizontal line across the number of processors. The faster they go down, the worse the scalability.

The architectural effects of SMP nodes can be seen on Seaborg, Cheetah, Thunder, and Jacquard in Fig. 12. Inside a SMP (Cheetah up to 32, Seaborg up to 16, Thunder up to 4, and Jacquard up to 2) the scaling is not bad. After we use more than one SMP, the scaling degrades immediately but gradually recover with the increase of the number of SMPs. Especially for Jacquard, its asymptotic scaling behavior is much better than Seaborg and Cheetah. BlueGene scales very well and its performance drop on 64 processors is related to the architecture. Apex-Map runs in co-processor mode, i.e., one cpu inside a node will be dedicated to communication. Therefore we actually require 64 nodes in this case. With 64 nodes request, the scheduler will only assign a smaller partition with a mesh network not a torus. With 32 nodes partitions, you are remaining on a board, with 64 nodes partitions you are across boards. With 256 nodes partitions, we get full midplanes. If we require a larger partition and run Apex-Map again using 64 nodes only, the

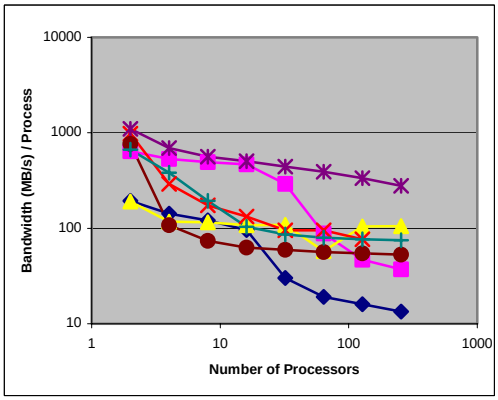

Fig. 12: The total aggregate bandwidth for L=4096 and α = 1.0

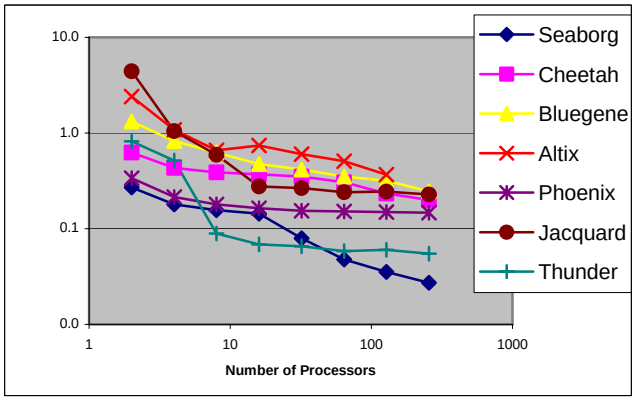

Fig. 13: The total aggregate bandwidth for L=1 and α = 1.0

performance drop will disappear. In general the results of Apex-Map can reflect many specific characteristics of the underlying platform.

The scaling performance for L = 1 is displayed in Fig. 13. In this case, the latency will become more important than bandwidth due to the short message size. This effect is reflected from the changes of the position order among these systems. In terms of scaling, BlueGene does not scale as well as in Fig. 12. But Cheetah improves a lot, especially when compared with Seaborg. Though the scalability is good for Thunder after the SMP threshold, its absolute performance still stays low.

5. Effect of Different Parallel Programming Paradigms

All previous parallel results were obtained with the MPI implementation of Apex-Map. However, due to its two-sided communication model, MPI is not so efficient to exploit the full potential of the underlying hardware. In this section, we are going to analyze the effect of different parallel programming models. We have implemented Apex-Map in two other promising paradigms, SHMEM and UPC. We are going to analyze the performance of these three programming models on Phoenix and Altix.

5.1 SHMEM vs. MPI

The performance ratios between SHMEM and MPI on Phoenix and Altix are shown in Fig. 14 and Fig. 15. On Cray X1, SHMEM always performs better than MPI. For areas with high spatial locality, where the long messages dominate the performance, SHMEM delivers 1 to 5 times higher bandwidth. This is also the case for areas with high temporal locality where the local memory access dominates the performance. However, as we decrease either the temporal locality or spatial locality, approaching the lower-left corner, the advantage of SHMEM becomes more and more prominent. In the best case, it can deliver 25 times better performance than MPI. The lower-left corner is the area with the lowest temporal locality and lowest spatial locality. Achieving good performance in this corner is notoriously difficult. The SHMEM programming model exhibits much more power to extract performance from the underlying hardware.

Surprisingly, on the Altix, SHMEM performs worse than MPI at the lower-right corner. We are currently investigating this anomaly. For all other areas, SHMEM is better. For the areas with large spatial locality, SHMEM performs around 1.5 – 2.5 times better. The place where SHMEM really excels is the area where the temporal locality is very high and spatial locality is relatively lower. When a data stream has high temporal locality, data access on this platform has become very efficient due to the hierarchical memory design. At this time, the MPI overhead induced by checking incoming requests, managing receive buffers, and calling non-blocking, asynchronous functions will become very expensive and significantly hurt the performance while SHMEM does not have such kind of overheads.

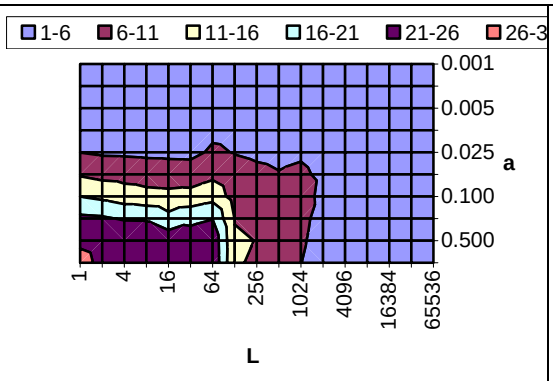

Fig 14: The bandwidth ratio between SHMEM and MPI on Cray X1

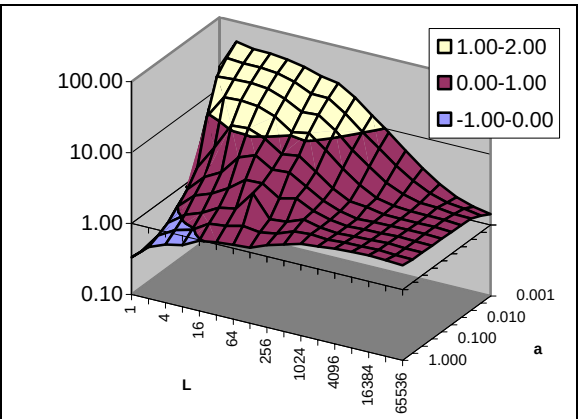

Fig. 15: The bandwidth ratio (log-scale) between SHMEM and MPI on Altix

5.2 UPC vs. SHMEM

UPC is also a single program multiple data programming model similar to MPI. But a major difference is that it provides a shared address space as well so that the "shared" data can be accessed by regular load and store operations. In our implementation of Apex-Map, the global data is allocated in the shared space and each process owns a chunk of the global data locally. The performance ratio between UPC and SHMEM is displayed in Fig. 16. UPC can perform 4 to 9 times better than SHMEM on the left side for low spatial locality area. This is mainly due to the underlying hardware that supports direct cache-line-size load operations from remote nodes. Compared with SHMEM, UPC does not need to pay any message overhead. However, with the increase of spatial locality, this advantage disappears gradually. The performance of UPC becomes even inferior to SHMEM at the right side where long messages dominate. In the worst case, UPC only delivers 70% of the SHMEM performance. The message overhead can well be recovered if large chunk of data can be moved together. UPC provides a similar function UPC_MEMGET to SHMEM function SHMEM_DOUBLE_GET. If we use UPC_MEMGET, the performance for long messages can improve substantially. The result indicates that it can even perform a little better than SHMEM. However, this implementation of Apex-Map is not efficient for short messages.

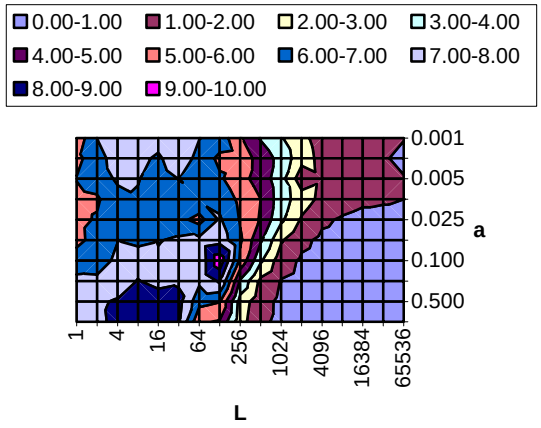

Fig. 16: The bandwidth ratio between UPC and SHMEM on Phoenix

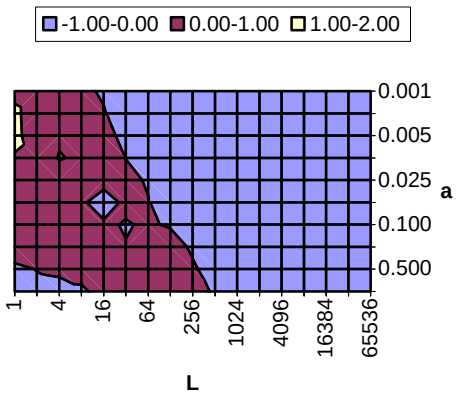

Fig. 17: The bandwidth ratio between UPC and SHMEM on Altix (log-scale)

The comparison ratio (in log-scale) between UPC and SHMEM on Altix is shown in Fig. 17. For that small area, upper-left in yellow color, the UPC performance could be astonishingly higher than SHMEM. This is perhaps related with the efficient cache design in the Itanium2 microprocessor to support data reuse. However, with the decrease of either the temporal locality or spatial locality, the performance drops sharply. In most of the cases, the performance of UPC falls behind SHMEM. In the worst case, UPC delivers 7 times lower bandwidth. Though using the block transfer function UPC_MEMGET helps when spatial locality is high, but this time the UPC performance never catches up with SHMEM. The different results on Phoenix and Altix indicate that the relative performance of UPC to SHMEM is really dependent on the UPC implementation and the underlying hardware support.

In summary, SHMEM usually performs better than MPI due to its efficient one-sided communication. On Phoenix, the biggest difference occurs when both the temporal locality and spatial locality reaches their lowest points while on Altix, it happens when spatial locality is lower but temporal locality is higher. Most of the times, UPC performs better than SHMEM when spatial locality is low due to efficient load/store operations. But when a data stream has high spatial locality, its performance is closely related with the implementation and underlying hardware.

6. Summary and Future Work

In this paper, we have introduced a novel benchmark, which focuses on global data access and measures how fast global data can be loaded into the CPU. It has three parameters, the global memory size M used, the temporal locality α, and the spatial locality L. By varying these parameters, we can analyze the performance characteristics of processors or parallel computing systems in this multidimensional space.

The results clearly reflect the different design principles between vector and superscalar platforms. The performance of superscalar platforms is sensitive to both temporal and spatial locality. These two kinds of localities can compensate each other to some degree. The performance of vector platforms is strongly sensitive to spatial locality and much less sensitive to temporal locality.

Apex-Map results also reflect many specific performance characteristics of the underlying platforms. They show the SMP scaling effect on all systems with SMP nodes and the partition effect for the 64-processor case on BlueGene. Apex-Map can also help to identify the MPI implementation strategies, such as the switch from eager to rendezvous mode on Phoenix and the memory register policy on Jacquard.

The Apex-Map results also indicate that the parallel programming languages significantly affect the delivered performance. To which degree depends on the specific platform and the characteristics of the accessed data stream. In most of the cases, SHMEM performs better than MPI. UPC could even perform significantly better than SHMEM on Phoenix, especially for low spatial locality data access.

We are also investigating methods to characterize parallel applications with Apex-Map parameters. In our earlier work, we have successfully characterized several sequential scientific kernels [8] this way. Such a characterization would allow us to use Apex-Map as a performance proxy for real scientific applications.

7. Acknowledgements

The authors would like to thank NERSC/LBNL, ORNL, Argonne National Laboratory, and Lawrence Livermore National Laboratory greatly to provide us the chance to access their computer systems.

8. References

[1] Top500 Super Computer Sites, http://www.top500.org
[2] Application Performance Characterization benchmarking, http://ftg.lbl.gov
[3] STREAM: http://www.cs.virginia.edu/stream/
[4] HPC Challenge Benchmark, http://icl.cs.utk.edu/hpcc/
[5] MAPS: http://www.sdsc.edu/PMaC/MAPs/maps.html
[6] NAS Parallel Benchmarks: http://www.nas.nasa.gov/Software/NPB/
[7] SPEC: http://www.spec.org/
[8] Erich Strohmaier and Hongzhang Shan, "*Apex-Map: A Synthetic Scalable Benchmark Probe to Explore Data Access Performance on Highly Parallel Systems*", EuroPar2005, Lisbon, Portugal, Aug. 2005.
[9] Erich Strohmaier and 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.
[10] Berkeley UPC – Unified Parallel C, http://upc.nersc.gov

DISCLAIMER