MPI, SHMEM, and UPC Performance on the Cray X1
— A Case Study using APEX-Map

Hongzhang Shan and Erich Strohmaier
{hshan, estrohmaier@lbl.gov }

Computational Research Division
Lawrence Berkeley National Laboratory

Abstract

APEX-Map is a synthetic performance probe for characterizing access behavior to global data structures. It is designed around concepts for temporal locality and spatial locality and can be used to analyze the performance characteristics of a computing platform across a whole range of localities. It can also be used to compare performance across different architectures or different programming paradigms. In this paper, we present the results of APEX-Map on the Cray X1 and use them to analyze this specific computing platform. We have implemented APEX-Map using MPI, SHMEM, and UPC and compare these three programming paradigms on the Cray X1. We are also going to discuss some performance problem we have found regarding the current MPI library implementation.

1. Introduction

The delivered performance of current parallel computers is closely related with how fast global data can be fed into the computing units inside the CPUs. The effective capability to move data depends on both the characteristics of the applications and of the underlying hardware. Unfortunately, until now, we lack a standard to measure this capability and compare it across different platforms. APEX-Map is designed to fulfill this purpose.

APEX-Map assumes that an application's data access streams can be characterized by a few parameters. Currently, we selected three parameters for this characterization, the accessed data size (M), the temporal locality (K), and the spatial locality (L). By integrating these three parameters in a simple kernel, APEX-Map tries to mimic the application's memory access behavior. It can then be used as proxy for the performance behavior of the underlying codes. By varying the values of these parameters independently between their extreme values a performance map (or performance signature) can be generated for each platform. These performance maps can be used not only to understand each platform's characteristics but also to compare performance across different parallel programming models. In this paper, we focus on comparing the performance of three parallel programming models, MPI, SHMEM, and UPC, on the Cray X1.
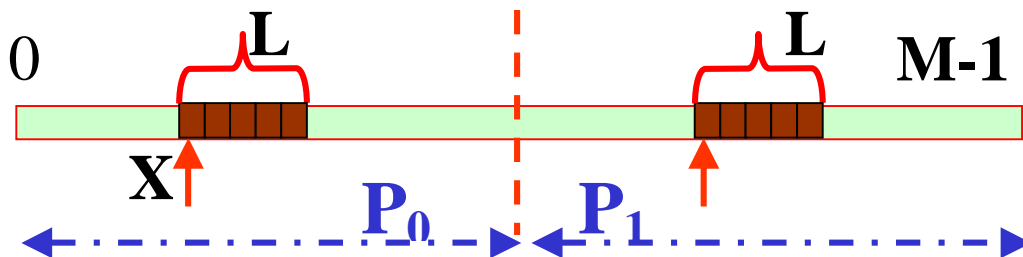

Fig. 1: APEX-Map data distribution and data access

The implementation of Apex-Map will be briefly described in Section 2. In Section 3, we will examine the performances of the three parallel programming models used and we will discuss the performance problems of current MPI implementation. Finally, we summarize our results in Section 4.

2. APEX-Map Implementation

In APEX-Map, the global data are evenly distributed across all the processes as shown in Fig. 1 for a two-processor case and data access is in block mode. In implementations using message passing, the message size is equal to the block size. The starting address, X, for each block is computed based on a non-uniform random process following a power distribution function controlled by the parameters M, K, and L and these addresses are stored in an auxiliary array. For each process, the computed indices will be adjusted according to its rank to reflect its local view of the global data.

Table 1: The Outline of APEX-Map

Repeat N Times
Generate Index Array
CLOCK(start)
For each Index *i* in the Array
If (data not in local memory)
Get Remote Data
End If
Compute
CLOCK(end)
RunningTime += end – start;
End Repeat

Table 2: The Implementation Differences between MPI, SHMEM, and UPC

MPI	SHMEM	UPC
Repeat N Times	**Repeat N Times**	**Repeat N Times**
Generate Index Array	Generate Index Array	Generate Index Array
CLOCK (start)	CLOCK (start)	CLOCK (start)
For each Index *i* in the Array	For each Index *i* in the Array	For each Index *i* in the Array
If (not local data)	If (not local data)	If (not local data)
Generate Remote Request	SHMEM_DOUBLE_GET()	**//method 1:**
Else	End If	UPC_MEMGET()
Compute	Compute	**//method 2:**
End If	CLOCK (end)	p = global_data[rid]
Serve Incoming Requests	RunningTime += end - start	for (i = 0; i < L; i++)
Process Replies	**End Repeat**	sum+=p[offset+i]
CLOCK (end)		End If
RunningTime += end-start		Compute
End Repeat		CLOCK (end)
CLOCK (start)		RunningTime += end - start
Wait For Finish		**End Repeat**
CLOCK (end)		
RunningTime += end-start		

The outline of APEX-Map is shown in Table 1. For each data address stored in the index array, it is checked if the data is in local memory, in which case the computation starts immediately, or if the data resides in remote memory in which case an inter-process communication will be activated to fetch the remote data. In principle we do allow several outstanding requests for remote data and out of order execution on the arriving data, however, in practice it might be difficult or impossible to take advantage of this in many programming paradigms. The compute module is essential for APEX-Map since it measures

the rate of the global data being fed into the computing units not only into local memories. More implementation details can be found at [1].

How to fetch the remote data differs substantially among different parallel programming models. Table 2 displays the main differences between MPI, SHMEM, and UPC. The SHMEM implementation is straightforward and based on the sequential model. The data can be directly fetched into a process's local memory by shmem_double_get() or other similar functions due to its one-sided communication model. However we cannot take advantage of potential out of order executions as the SHMEM interface does not provide an asynchronous get function. For UPC, there are two obvious ways to carry out the remote access. Like SHMEM, UPC can also call a function UPC_MEMGET to bring all the requested remote data into a local buffer. Another way is to take advantage of the shared memory model to load the data directly using regular load operations one data item at a time. The pointer p is used to point to the remote data.

The implementation of APEX-Map in MPI is much more complicated. Due to the random nature of the communication, non-blocking functions must be used to achieve acceptable performance. A process not only has to request data from other processes but also has to serve incoming requests. At the same time, it has to process returning replies and manage the necessary receive buffers. After a process has completed its own computations, it still has to wait for all other processes to finish in case some further requests for data arrive. Therefore, MPI incurs much more implementation overhead than the other two programming models. However we can implement out of order execution on arriving data quite easily.

3. Performance Data

Apex-Map outputs the average cycles per data access for one process and the aggregate bandwidth in MB/s for a given set of parameters. By running a set of parameters, such as K = 0.001 to 1.0 and L = 1 to 65536 words, Apex-Map can generate two dimensional performance surfaces which allow to visualize the performance effects of temporal locality and spatial locality. Fig. 2 shows the performance map for MPI with 256 processes. The global data array size used is 256*512MB = 128GB.
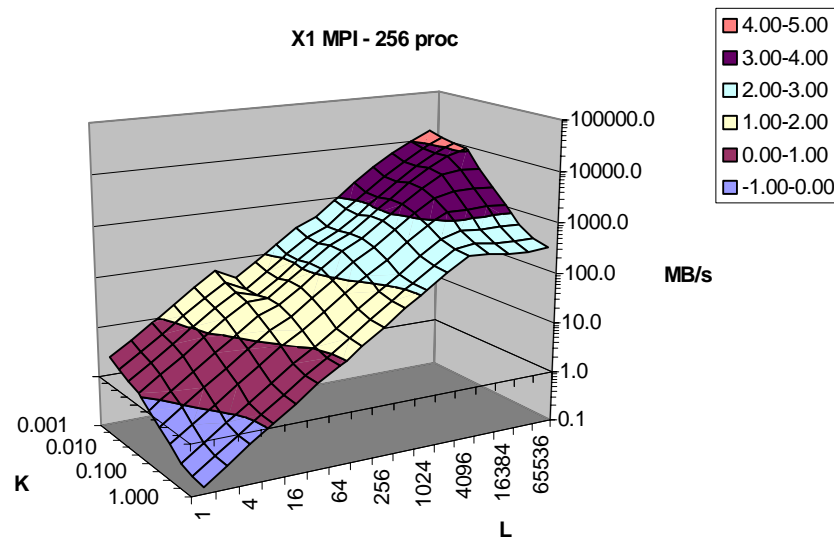

Fig. 2: The aggregate bandwidth (MB/s) for MPI with 256 processes

In many cases, the performance effects of temporal locality and the spatial locality can cover for each other. Let's examine the contour line of 100MB/s bandwidth in Fig. 2. With high temporal locality K = 0.001, the spatial locality needs to be only around 200 to reach this performance. If the temporal locality is reduced to K = 1.0, we can obtain the same bandwidth by increasing the spatial locality to around 700. However, the spatial locality has much more significant effect than the temporal locality on the Cray X1. In most of the cases, the total aggregate bandwidth scales almost linearly with the increase of L. The

performance drop when L equals 64 is related to the MPI implementation. A MPI call MPI_Iprobe is called frequently in APEX-Map to check for incoming messages. The incurred overhead by this function suddenly increases after message size becomes larger than 32 words due to protocol changes.
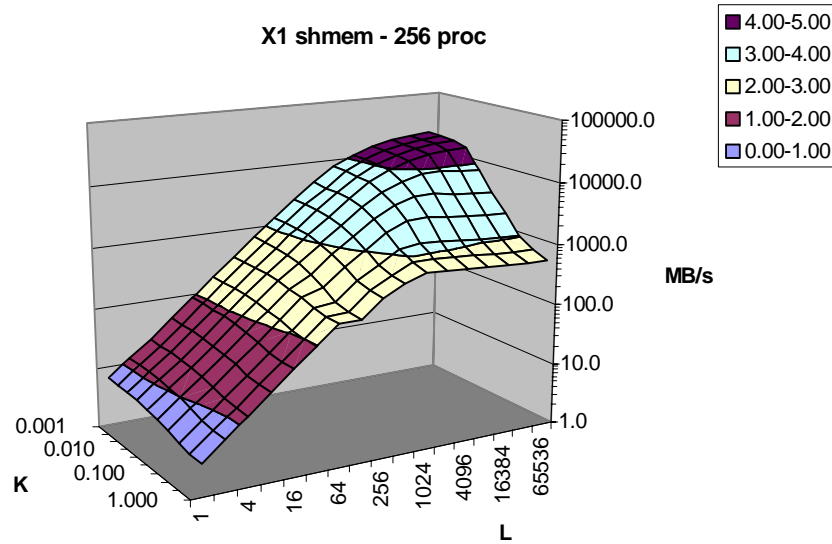

Fig. 3: The aggregate bandwidth (MB/s) for SHMEM with 256 processes

The performance of SHMEM is illustrated in Fig. 3. The MPI related performance drop for L = 64 does not appear here. Compared with the MPI performance, SHMEM delivers much higher bandwidth and the effect of temporal locality becomes even smaller.
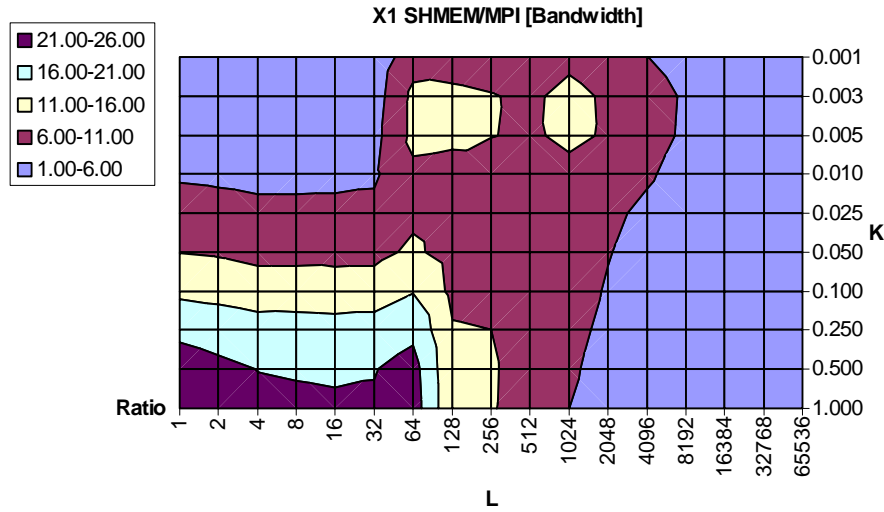

Fig. 4: The SHMEM/MPI Bandwidth ratio for 256 Processes

Fig. 4 shows the bandwidth ratio between SHMEM and MPI. For the area with low temporal locality and low spatial locality (lower-left corner), SHMEM performs substantially better than MPI. In the best case, SHMEM delivered 24 times higher bandwidth than MPI despite the fact that the MPI implementation takes advantage of out of order executions. With the increase of temporal locality or spatial locality, the advantage of SHMEM becomes relatively smaller. On the right area when message size becomes larger than 4096 words, SHMEM only provides 1 or 2 times higher bandwidth.

Table 3: The Bandwidth Ratio of UPC (method 1) vs. SHMEM

	0.0010	0.0025	0.0050	0.0100	0.0250	0.0500	0.1000	0.2500	0.5000	1.0000
1	1.03	1.05	1.01	0.97	0.90	0.81	0.70	0.60	0.54	0.53
2	1.01	1.02	0.99	0.95	0.89	0.77	0.68	0.58	0.53	0.51
4	1.05	1.04	1.02	0.98	0.88	0.80	0.70	0.59	0.54	0.52
8	1.03	1.05	1.01	0.98	0.88	0.80	0.68	0.59	0.54	0.52
16	1.04	1.04	1.02	0.97	0.88	0.79	0.70	0.59	0.55	0.52
32	1.05	1.05	1.03	0.97	0.87	0.80	0.71	0.59	0.55	0.52
64	1.06	1.03	1.03	1.01	0.92	0.83	0.68	0.60	0.55	0.53
128	1.06	1.06	1.06	1.09	1.02	1.04	1.06	1.06	1.03	1.01
256	1.06	1.08	1.08	1.07	1.09	1.07	1.06	1.05	1.02	0.99
512	1.05	1.09	1.11	1.13	1.07	1.07	1.10	1.08	1.04	1.03
1024	1.04	1.12	1.08	1.08	1.03	1.07	1.10	1.06	1.01	0.94
2048	1.01	1.04	1.03	1.02	1.06	1.11	1.08	1.02	1.00	0.98
4096	1.04	1.04	1.03	1.03	1.10	1.05	1.04	1.09	1.11	1.03
8192	1.05	1.05	1.04	1.04	1.07	1.06	1.18	1.27	1.22	1.14
16384	1.06	1.06	1.03	1.05	1.01	1.28	1.39	1.40	1.34	1.07
32768	1.03	1.03	1.03	1.03	1.18	1.38	1.54	1.51	1.35	1.21
65536	1.00	1.03	1.00	1.00	1.17	1.40	1.60	1.62	1.40	1.33

As we described earlier, we have two UPC implementations. The first method is to call a function UPC_MEMGET that performs similarly with SHMEM_DOUBLE_GET. The performance ratio between this implementation and SHMEM is shown in Table 3. For the left half of the table (high temporal locality), the performance of UPC and SHMEM are close to each other. For the right half of the table (lower temporal locality), UPC performs worse for short messages and better for longer messages.

Table 4: The Bandwidth Ratio of UPC (Method 2 vs. Method 1)

	0.0010	0.0025	0.0050	0.0100	0.0250	0.0500	0.1000	0.2500	0.5000	1.0000
1	5.17	5.48	6.00	5.89	6.37	7.04	9.24	12.16	13.48	14.68
2	6.95	6.39	6.47	6.19	6.84	8.30	9.62	12.47	14.66	15.34
4	7.48	7.24	7.03	6.75	7.13	8.75	10.34	12.39	15.00	15.91
8	7.30	6.87	6.95	7.05	7.49	8.48	11.04	13.62	15.95	16.33
16	7.06	6.58	6.50	6.61	7.06	8.82	10.13	12.90	15.62	16.13
32	7.24	7.14	6.82	7.02	7.26	8.89	10.47	13.27	15.11	15.49
64	6.61	6.52	6.03	6.87	6.13	8.78	10.56	11.93	11.54	9.76
128	6.28	5.41	5.79	6.13	6.97	7.30	9.08	7.68	5.87	4.85
256	4.78	5.19	4.92	5.39	5.45	5.74	5.48	3.67	2.52	2.41
512	3.71	4.22	4.13	4.44	4.59	3.88	2.87	1.62	1.29	1.16
1024	2.96	3.26	3.11	3.21	3.10	1.97	1.28	0.89	0.79	0.84
2048	2.16	2.27	1.33	2.44	1.77	1.09	0.80	0.72	0.72	0.76
4096	1.65	1.73	1.76	1.68	1.05	0.77	0.65	0.62	0.64	0.71
8192	1.35	1.39	1.45	1.36	0.83	0.65	0.58	0.54	0.57	0.65
16384	1.17	1.20	1.27	1.14	0.69	0.54	0.49	0.48	0.52	0.69
32768	1.10	1.14	1.13	1.08	0.59	0.50	0.44	0.45	0.51	0.60
65536	1.07	0.56	1.10	1.00	0.59	0.48	0.41	0.40	0.48	0.55

The second method is to take advantage of the shared memory model provided by UPC to use regular load operation. Compared with method1, using load operation instead of the block transfer function UPC_MEMGET is far more efficient for short messages. This can be clearly seen from Table 4 that shows the performance ratio between regular load operation (method 2) and block transfer operation (method 1). For longer messages, the block transfer is better, especially for lower temporal locality area where remote communication dominates. Therefore the choice of implementation to achieve best performance in UPC on the Cray X1 depends on the level of spatial and temporal locality during execution. The best UPC implementation always performs better than SHMEM.

3.1 Problem with current MPI Implementation

During our experiments, surprisingly we found that the performance of our MPI APEX-Map implementation is highly affected by the behavior of the process with rank 0. The above-presented MPI results are obtained by using one extra process, i.e., process 0 is always idle, and the work for process i has been shifted to process $i+1$. If we do not use one extra process, the behavior of process 0 will significantly degrade the performance. Fig. 5 shows the time breakdown for 256 processes with and without one extra

process. As we described earlier, in our MPI implementation, there is a "wait for finish" stage after a process has finished its own computation because it may still have to serve others' requests. The "Working" stage is the time each process needs to finish its own computations. The "Waiting" stage is the time to wait for all other processes to finish. For the case without an extra process, the working stage for the process with rank 0 is significantly longer than for other processes which therefore have to spend more than 40% time purely waiting for process 0. If we use one more process so that we can shift the process i's work to process $i+1$ then process 0 has nothing to do in the working stage and all its running time is waiting for others to finish. This is exactly what we see in Fig. 5. The reason which leads to this problem is under investigation.
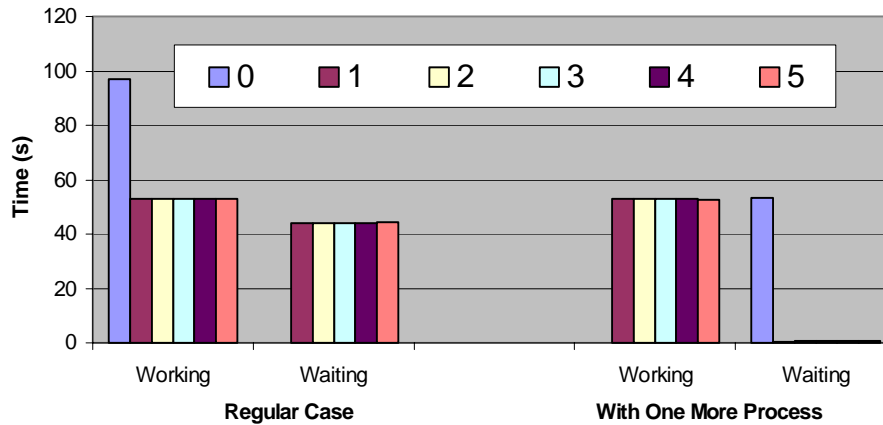

Fig. 5: The time breakdown with and without one more process

(only the first 5 processes are displayed for simplicity)

4.   Summary

In this paper, we use Apex-Map, a parameterized synthetic performance probe, to analyze the performance of the Cray X1 under different parallel programming models. Apex-Map measures the capability of a machine to feed data into the computing units for address streams with different temporal locality and spatial locality. We find that UPC can deliver the best performance compared with SHMEM and MPI. While the performance of SHMEM is close to UPC, the MPI performance suffers substantially due to its two-sided communication model, which incurs a lot of implementation overhead. We also find that in the current MPI implementation, the process with rank 0 does not behave properly in our code.

5.   References

[1] Apex-Map: http://ftg.lbl.gov
[2] Berkeley UPC – Unified Parallel C: http://upc.nersc.gov