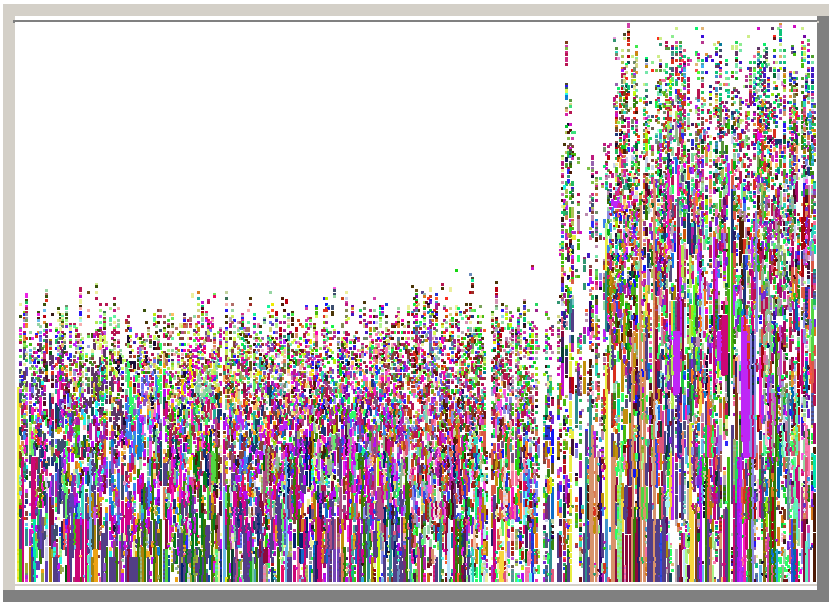
Scaling Up Parallel Scientific Applicaitons on the IBM SP

David Skinner, NERSC HPCF

Table of Contents



Abstract

This document provides a technical description of the IBM SP seaborg.nersc.gov with an emphasis on how the system performs as a production environment for large scale scientific applications. The overall goal is to provide developers with the information they need to write and run such applications. While some of the information presented here may be applicable in a larger context, the focus is on experiences and scaling techniques specific to seaborg.nersc.gov.

In the first few sections we seek to determine how well the theoretical capabilities of this machine may be realized in practice. Most of the measured performance numbers come from small code microkernels or test codes rather than from real user codes. In a companion document several real applications are explored in detail.

The microkernel approach described above has value to the user since the most widely quoted performance numbers often reflect the theoretical (peak) performance values rather than those realized in practice. Likewise anecdotes from full blown applications often involve mixed algorithms, hidden constraints or other specifics which make the results hard to generalize. Microkernels and test codes provide a middle ground which is a reasonable best case scenario for real user codes and provide the user with a small kernel of code can serve as a template for the writing or modification of more substantial codes.

In what follows we will present several such examples alongside performance and scaling information. The next section introduces the role of concurrency in HPC. The following sections treat parallel scaling from the machine and application development perspectives.

Suggestions for improving this draft document are welcome. Please contact the author if you have comments, corrections, or questions.

Introduction

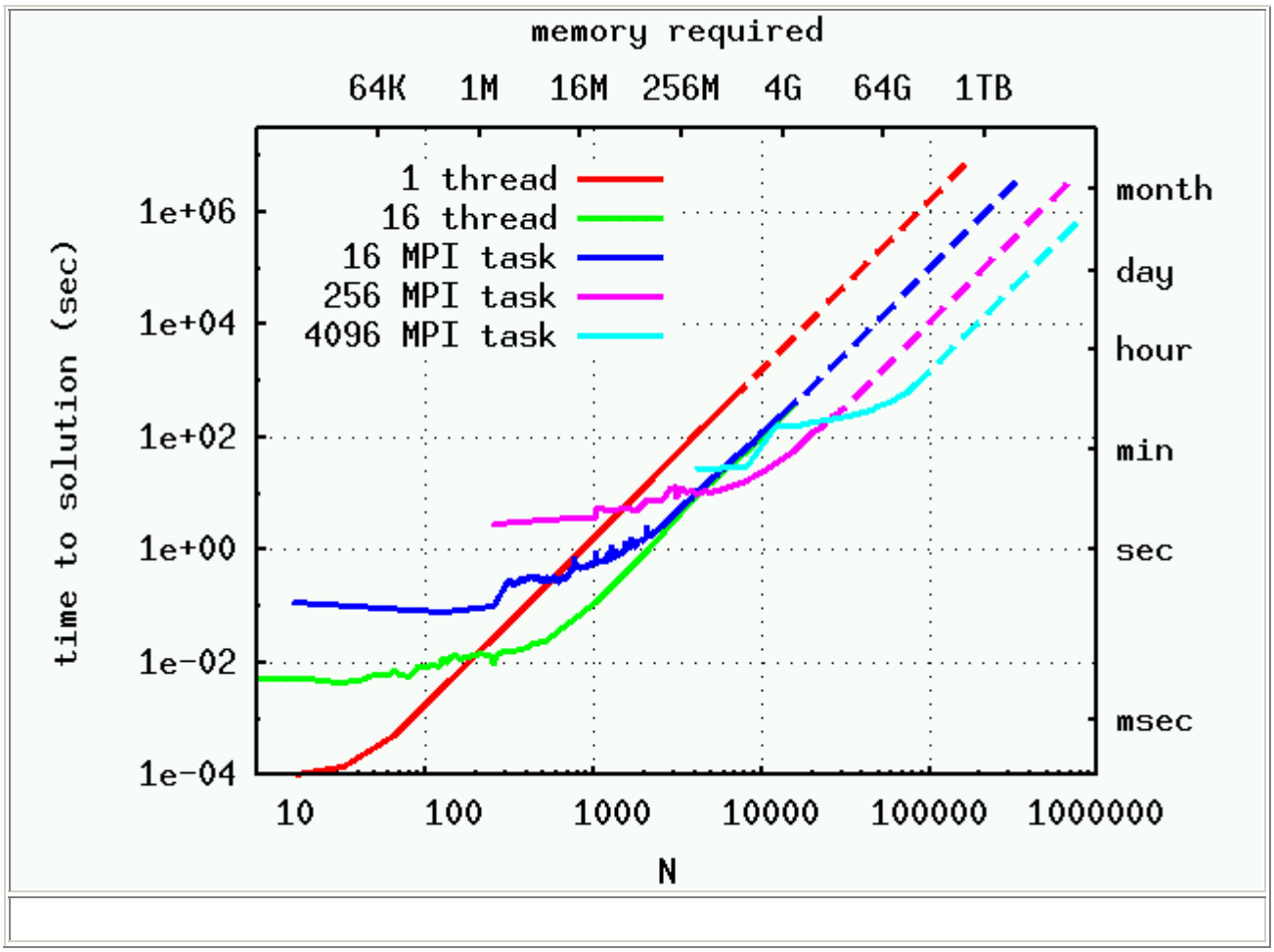
Capability Computing

Running large scale scientific computations which can not be reasonably implemented on clusters of independent or weakly coupled machines is commonly referred to as *capability computing*. Managing a large number of tasks working independent problems is referred to as *Capacity computing*. As seaborg is a DOE resource for non-classified capability computing, this document will focus on the implementation of high concurrency solutions to large scale scientific problems. Thus, for the most part, it is in the context of capability, rather that capacity, that we mean parallel scaling.

Finding the level of parallelism best suited to a given problem can be challenging. In general it involves careful consideration of the nature and size of the problem to be solved, the properties of the compute hardware and interconnect, the algorithm and the time scale on which a solution is required.

Consider for example dense matrix multiplication, `C(n,n) = A(n,n)*B(n,n)`, encountered in LAPACK, ESSL, SCALAPACK and PESSL as a DGEMM routine. The level of parallelism which provides the fastest solution depends on the scale of the problem. For small problems the setup and overhead from parallelism dominates any benefit from the capabilities that parallel computing brings. Beyond this point is the regime of capability computing.

Scaling of NxN matrix-matrix multiply



Capability computing means more than providing a faster solution. For problems of sufficient scale, capability computing is a requirement for any solution at all. E.g., in most cases there is no practical way to extend single 32 bit CPU solutions beyond the 2G address space which constrains N < 10e4 for such an approach. Likewise for extending shared memory solutions beyond the memory limits of a single SMP node. This makes certain classes of problems unapproachable on commodity hardware and ultimately necessitates certain capability hardware such as high bandwidth low latency interconnects and parallel filesystems.

Specific properties of the compute hardware are also demonstrated above. If the nodes had 8 CPUs, instead of 16, the cross over point between a single node and multiple node solution would be at a different problem size. On seaborg's 16 way SMP nodes the threaded and MPI based matrix multiply show asymptotically identical performance. Both are using shared memory and avoid switch communication.

Dense matrix multiplication is extremely simple from an algorithmic perspective and not too much should be inferred from the above scaling data as it regards other algorithms or computations. As a rough sketch, however, it does represent how scaling of problem size and machine capabilities impact optimal solution strategy. Other algorithms will have different scaling properties, but the overall trends and transition to capability computing similarly occur.

It's worth mentioning that not all tasks scientific or otherwise need massive parallelism. Post-processing of data, debug work, and data workup are important parts of scientific computing and often achievable on a single node or CPU. As such not every job run on a machine such as seaborg will be a capability job.

For the class of scientific problems which do require the capabilities offered by large scale parallel computation seaborg provides development and production environments for implementing and solving scientific problems of scale.

This document and the consultancy resources available within the NERSC User Services Group can provide answers to researchers about scaling and optimal implementation of scientific codes on NERSC hardware.

Constraints to Scaling

As a way of setting boundary conditions, it is useful to lay out what architectural constraints exist on the IBM SP to running parallel at large scale. These limitations are intrinsic to the machine and are mentioned (along with some notes on how they are mitigated) only briefly before moving on to application level issues.

4096 way MPI :

> Currently the MPI implementation on seaborg supports up to 4096 tasks or 256 fully packed nodes. Higher concurrency is achievable only through using mixtures of threads and MPI tasks (e.g., OpenMP). When approaching this upper bound on MPI tasks issues involving performance and memory arise.

Process Scheduling:

> As with most types of cluster computing there is no fine grained synchronization between nodes. This means synchronizing

becomes more difficult at higher concurrency. NERSC does its best to deal with this by enforcing resource uniformity, e.g., not sharing nodes between user jobs and automatically detecting and dealing with errant processes on every node. Synchronization is increasingly important as jobs scale up and is a common bottleneck when scaling up parallel codes.

Job Scheduling:

Scheduling jobs requires waiting for a number of resources proportional to the concurrency to become available. The lack of checkpoint/restart or gang scheduling capabilities with performance sufficient to deal with large scale parallel jobs means that the scheduling of small long running jobs will be at crossed purposes to scheduling large scale parallel jobs. NERSC elevates the priority of larger concurrency jobs to enhance their throughput and has regular NUG discussions about queue structure.

Scaling up a parallel application is largely about avoiding constraints and bottlenecks. Aside from the unavoidable contrasts above, many parts of code itself may come under algorithmic strain as concurrency is increased. Knowing the constraints of the chosen algorithms and their alternatives is of great benefit in avoiding bottlenecks.

Methods of dealing with these constraints and bottlenecks are provided in the next two sections.

SMP Scaling

Nighthawk II node

Seaborg compute resources consist of 416 compute node each with 16 CPUs per node. Each node is capable for performing at most 24 GFLOP/second. All the nodes are IBM 375 Mhz NightHawk II nodes with the following overall CPU and memory specification.

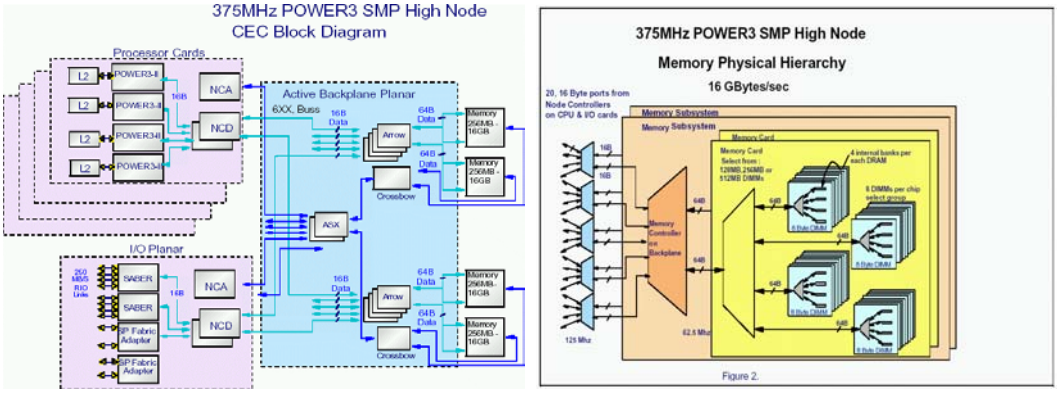
Processor	
Processor class	POWER_630
Clock frequency	374.7 MHz
Floating Point Units	2 (*,+,FMA)
Peak GFLOP/s	1.5
Real Registers	40
Virtual Registers	64
L1 Inst Cache Size	32 KB
L1 Data Cache Size	64 KB
L1 Data Cache Line Size	128 B
L1 Cache Associativity	4 way by line
L1 latency / bandwidth	5 nsec / 3.2 GB/sec
L2 Cache Size	8192 KB
L2 Cache Associativity	4
L2 latency / bandwidth	45 nsec / 6.4 GB/sec

Memory	
Memory topology	crossbar
Memory format	SDRAM DIMMs
Memory banking	4 banks / DRAM
Peak Memory BW	16 GB/sec
Memory bus speed	187 Mhz (2:1)
Page Size	4 KB
TLB size	128x2 Pages
TLB miss penalty	25-125 cycles
L2->L1 Prefetch Registers	10
L2->L1 Prefetch Streams	4
L1 -> registers	2 Word/cycle Load
L1 <-> registers	1 Word/cycle Load/Store
L2 -> L1	1.3 Word/cycle
Memory -> L1/L2	1 Word/cycle

Each CPU has its own separate caches so there is minimal resource sharing or cache conflict between CPUs on an SMP power III node. This separation provides an important simplification to the developer of parallel codes. Since each CPU's filling and invalidation of cache impacts only code running on that CPU there is less contention at this lowest level than on machines where low level resources are highly shared. Application programmers need not partition memory or cache access patterns along processor card or multi-chip module (MCM) boundaries as cache memory affinity is not an issue.

Conversely, for main memory there is no notion of local memory. All CPUs within a node access main memory over a uniform crossbar switch. While the possibility of contention over this switch is real (and

will be treated below), there is no need for the application programmer to keep track of which parts of main memory are local to the CPU.



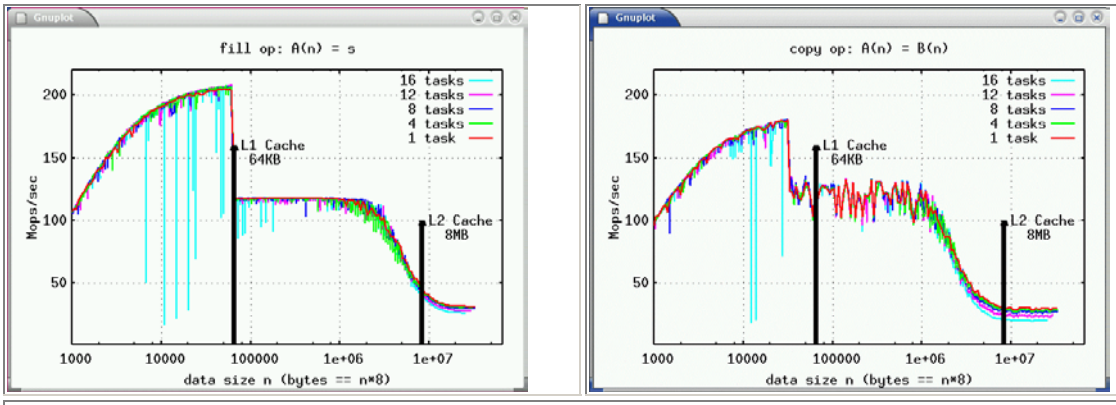
from: [RS/6000 SP 375MHz POWER3 SMP High Node Overview]

Memory Contention

As main memory is shared on an SMP, contention may occur. The peak main memory bandwidth is 15.6 GB/s based on the crossbar memory subsystem detailed above.

The full bandwidth is not available to a single task. In order to saturate the main memory bandwidth multiple tasks are required. A more detailed understanding of memory contention on the nighthawk II node can be arrived at by considering how the performance of N memory intensive tasks

Scaling of SMP memory contention.

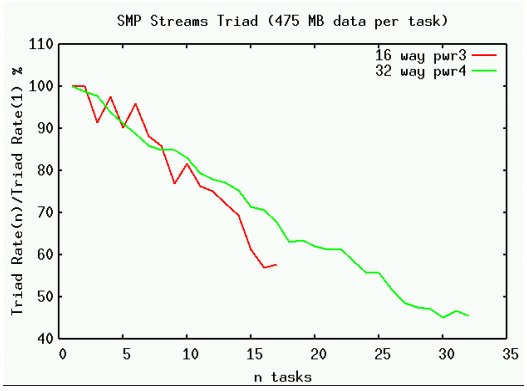


Source: [xtream] memory profiling tool (concurrency through MPI).
[More results.]

Two aspects of how memory contention scales with concurrency in parallel applications are demonstrated above:

- When the data structures being acted on are sufficiently small that cache reuse is possible, increasing concurrency has minimal impact. While the tasks in the parallel application are not contending with one another in this regime, they are contending with the OS, cron jobs, and other system interruptions. As the concurrency of the code running on the node increases it becomes increasingly likely that any such system interruption will create a resource contention with the code. These interruptions tend to be very brief and their frequency increases with SMP concurrency. On seaborg, NERSC, does what it can to keep these interruptions to a minimum. Intra-node concurrencies above 16 above obviously not recommended.
- As the memory space required by the application extends into main memory a clear trend toward resource contention is observed. Here the contention for memory bandwidth exhibits a sustained drop in performance of each task. The magnitude of this contention depends on concurrency and on the operation being performed on the data. As multiple CPUs draw on the SDRAM main memory they contend on the memory controller.

As the SMP is loaded with more processes the main memory bandwidth available to each task individually will decrease. This is summarized below for the `daxpy` like Triad microkernel (a(i) = b(i) + s*c(i)). As the number of tasks on the node increases the main memory bandwidth per task, shown here as a percentage, decreases.



That tasks in an cache based SMP compete for main memory access is certainly no surprise and in practice the situation is not as bad as it may seem. The example above is roughly a worst case scenario for memory contention, where n memory bandwidth bound processes

contend for access to main memory. Applications which are not strictly memory bound, show more varied memory accesses, or greater memory reuse should in practice show less contention.

Developers of scientific applications should realize the above issue of memory contention may also to varying degrees impact their applications. For applications or algorithms which are particularly starved for memory bandwidth there may be benefit from decreasing the number of tasks run per node. This could yield a faster time to solution trading off of course a smaller maximum FLOP/s and possibly a decreased percentage of peak performance. Taking this approach should be done cautiously as in many cases it would lead to less efficient utilization of the nodes in a batch job.

MPI message sent inside the SMP through shared memory are detailed here.

MPI Scaling

Abstract

In this section we will present an brief overview of how MPI is implemented on the IBM SP followed by details on how the implementation performs at high concurrency alongside information on how to mitigate several common problems encountered when scaling up codes.

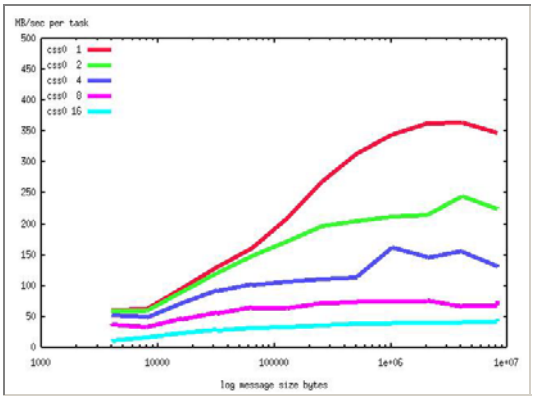
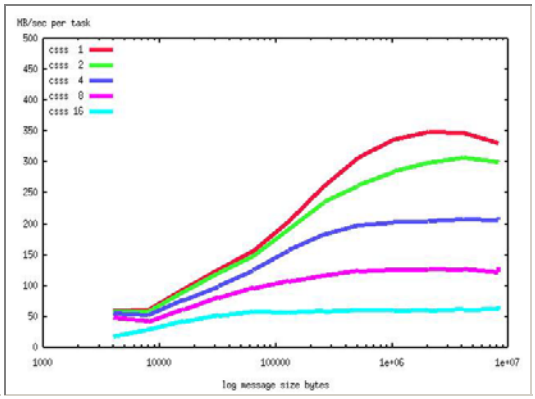
Colony Switch

The network fabric which joins the compute nodes on seaborg is the SP Colony switch II . Each NightHawk II node has two switch adapters (`css0` and `css1`) which connect directly to the memory bus of the NightHawk II node. These adapters interface to a three level switch.

In order to demonstrate scaling bottlenecks in point to point switch bandwidth, an experiment similar to what was done above for memory bandwidth can be done. Bandwidth across the switch as a function of message size is measured between two different nodes in the cluster. Each measurement involves two processes, one on each node which act as both sender and receiver (MPI roundtrip bandwidth is reported). The number of such pairs running concurrently is increased from 1 to

16 in order to quantitatively demonstrate the scaling properties of contention and switch bandwidth.

Two such experiments were done; the first on a single adapter `css0` and the second on the multi-link adapter `csss`.

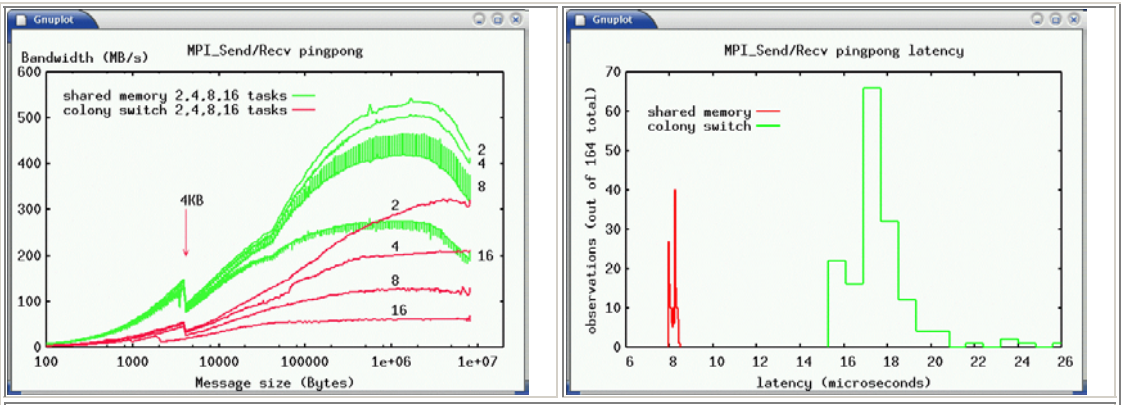
css0	**csss**
	

While for a single pair of tasks the two adapters perform roughly the same, the `csss` adapter shows better performance (less contention) when the number of concurrently communicating tasks increases. Currently the `csss` or multi-link adapter does not support striping of individual messages across the physical adapters. For this reason a single stream sees no additional bandwidth by using the multi-link adapter, but multiple streams will see a better aggregate bandwidth.

The `csss` adapter is used by default on seaborg and most application developers should use it since it is most often the best route to the switch. At earlier software levels there were some constraints on the use of `csss` which have been removed.

Not all MPI communication goes across the switch. For communications between tasks that reside on the same node it is much more efficient to route MPI message through shared memory buffers. The environment variable "MP_SHARED_MEMORY=YES" is set by default to allow this. An experiment similar to that done above for `css0` and `csss` is presented below where intra and inter-node bandwidths for pairs of MPI tasks are compared.

Intra and Inter Node Messaging



source: [pingpongn.c], latency samples collected over 33 hour period

Shared memory messages in the range of 1MB in length (10^5 doubles) show optimal bandwidth.

Likewise it can be seen that the latency of messages is much lower on a node than through the switch. This is to be expected, but it is useful to keep in mind that rapid exchanges of small messages may benefit from being kept on node or possibly aggregated before being sent off node.

That the distribution of latencies is much tighter for shared memory than switch routed MPI makes sense as well. The number of compute resources, and in particular shared compute resources, that are involved in an MPI message sent over the switch is much greater than shared memory messaging on a dedicated node. For instances GPFS, which we'll discuss in the next section, uses the same switch for its data. As a result deviations from optimal latency are expected based on contention for this resource.

IBM's MPI implementation

The software stack for MPI on the SP is, as of this writing (06/03), as follows:

API	Library	Description
MPI	libmpi.a	The MPI API itself, implementing MPI1 and MPI2 (except dynamic processes)
MPCI	libmpci.a	A point to point API in which MPI library is written.
PIPES	libmpci.a	Low level message buffer system.
HAL	libhal	Hardware abstraction layer (IP or US)

The above library are available in 32 and 64 bit, thread-safe and non-thread-safe as well as internet protocol (IP) and user space (US) versions. Threadsafe compilation (via the "_r" compilers) is required for the 64 bit MPI library. US and IP differ in the paths that messages traverse between MPI processes.

	seaborg node 1		**seaborg node 2**
IP	user code -> kernel -> adapter ->	switch fabric	-> adapter -> kernel -> user code
US	user code -> adapter ->	switch fabric	-> adapter -> user code

User space communication allows your program to talk directly to the switch adapter avoiding kernel interruption. The user space MPI library is the default as its performance (latency and bandwidth) far exceeds that of the IP version. In the preceding and following examples user space MPI is understood unless specifically noted otherwise.

LAPI is also available on the SP, but will not be treated in this work where we focus on the scaling of MPI codes.

MPI based programs are run via the Parallel Environment (PE) most often using `poe`. Programs compiled with the PE compilers, e.g., `mpxlf`, `mpcc`, `mpxlf90` have their main entry point replaced with a new entry point which handles node allocation and process setup on all the nodes involved in the parallel job. A large number of environment variables ("MP_*") and command line options affect the way that PE starts the job. A few relevant to scaling which NERSC sets by default for all users are listed below:

MP_SHARED_MEMORY=YES	Intranode messages through memory: faster, less switch traffic
MP_EUIDEVICE=csss	Use both switch adapters: more switch bandwidth
MP_EUILIB=us	User space MPI: faster MPI messaging
MP_RETRANSMIT_INTERVAL=50000	How often to retry dropped packets: empirically best setting

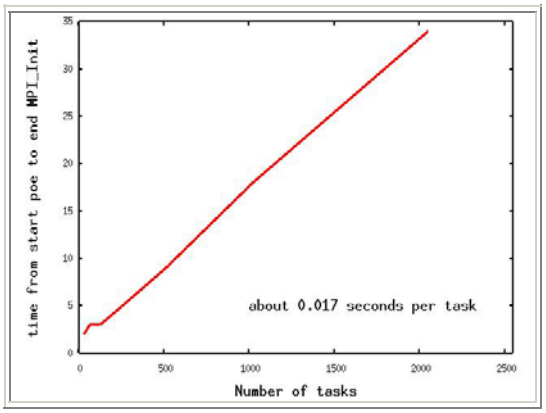
Other environment variable setting that impact application scaling performance are touched on in the following sections .

The users program code is then called after all PE startup work is complete. The time required to complete this startup is examined below.

Job Startup

Often the first step toward doing production work at a higher level of parallelism involves trying short running test codes with more tasks. One of the first things that a seaborg user may notice when examining scaling this way is a increased sluggishness of program startup and termination.

Therefore in order to properly gauge the timings from such scaling tests it is important to take into account the scaling job startup time. Below results are shown for the time taken to initiate `poe` and complete `MPI_Init()` for a varying number of tasks.



Since this is a one-time cost, its overall impact on user codes is minimal except for very short running jobs. A parallel application based on several short running parallel job steps would be impacted in proportion to its MPI concurrency.

It is however useful to know the time scale for parallel job startup. Especially for applications running at large concurrency it may be important to wait a minute or so before concluding that the job is in trouble. This small time window, while waiting for the first line of output from the program, is often highly scrutinized by users when running at a higher concurrency for the first time. Before deciding the job has failed or is hung make sure to wait for this startup to complete.

For longer running production jobs this issue is less important.

MPI Memory Usage

Aside from the fixed amount of memory required to build the PIPEs,

PIPE memory per task = 2*MP_PIPE_SIZE*(# of MPI tasks total - 1)

various MPI functions require internal temporary buffers. The size of these buffers is not documented, but is seen to vary with message buffer size, MPI_TYPE, and between the various MPI functions themselves. E.g., MPI collective and reduction operations tend to require greater internal buffer space than point to point MPI functions.

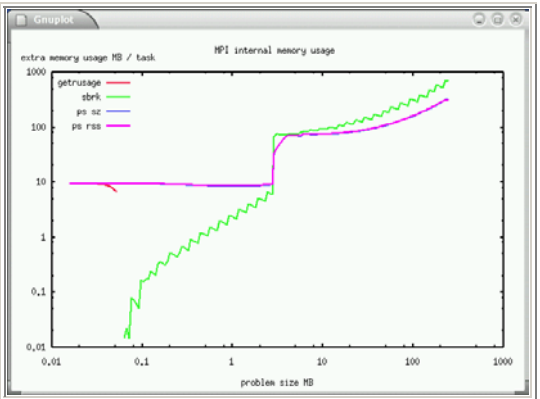
Characterizing the amount of memory required for a given message size to each MPI function call while potentially useful would probably provide too much information to be easily digested. The need for temporary storage by certain routines can be inferred from the algorithmic specifics of an MPI call.

From the perspective of someone writing MPI applications, the important question here is "How much memory can I use in my code and how much do I have to set aside for MPI?"

In a somewhat reduced approach to answering this question a microkernel code mpimem.c was written which exercises some of the most often used MPI functions (MPI_Barrier, MPI_Bcast, MPI_Alltoall, MPI_Allreduce, MPI_Reduce) and measures by way of system calls (e.g. `ps`, `sbrk`, `getrusage`) the memory used by the process as a function of problem/message size. Memory used by the process which is not allocated by the user code is considered part of the memory used internally by MPI.

The following plot shows the measurement described above for a 1024 way job:

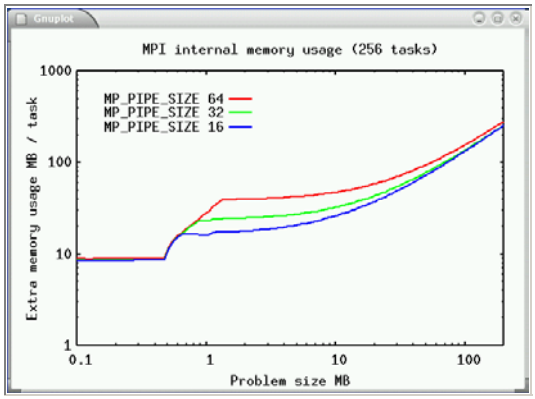
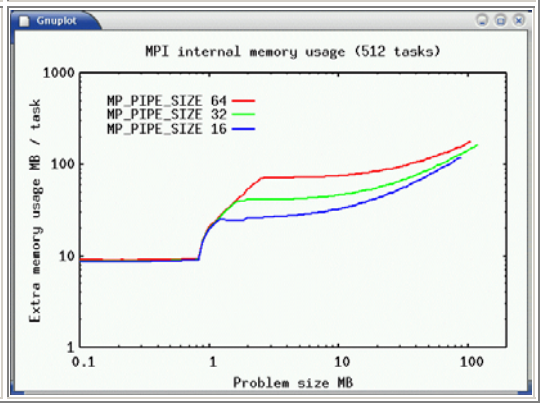
MPI Memory Usage

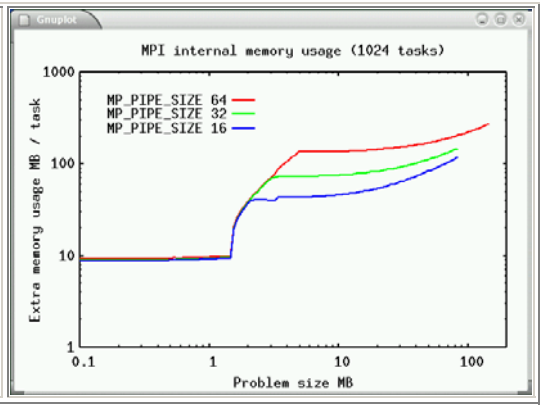


source: [mpimem.c]

For large concurrency jobs, internal memory usage by MPI can become a significant issue leading unexpected to job failures.

IBM's MPI implementation allows throttling down the size of the PIPEs via the PE environment variable MP_PIPE_SIZE. The default setting of 64 KB per PIPE can be decreased to either 32 KB or 16 KB. The memory savings by decreasing MP_PIPE_SIZE is shown in the following graphs:

# tasks	MP_PIPE_SIZE
256	
512	

1024	

Since there are potential performance penalties when using smaller MPI buffers it is recommended that the use of MP_PIPE_SIZE=32 or 16 should be viewed as a workaround for codes running out of memory rather than a standard programming practice for large concurrency jobs.

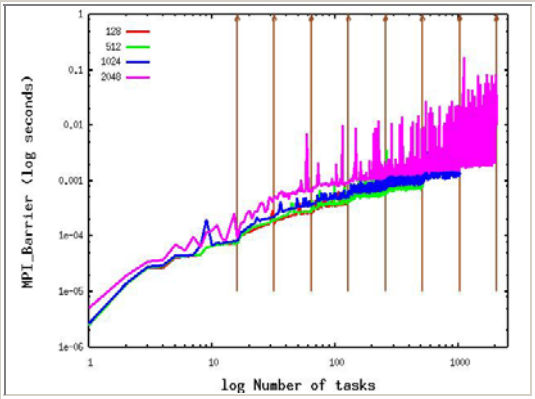
Synchronization

One of the major impediments to scaling up the concurrency of scientific codes is latency in synchronization. This is a well understood aspect of coordinating parallel operation of several resources when each may or may not be ready for work at a given time. As the number of resources increases the frequency and/or duration of interruptions must decrease in order to maintain the same efficiency.

Unfortunately for clusters of independently scheduled PEs the frequency and duration of interruptions are roughly fixed, being intensive properties of the cluster's building blocks. The cumulative time required to schedule and complete collective work across the cluster is extensive and scales with the concurrency of the parallel job. It is largely for this reason that we focus on mitigating the impacts of synchronization at high concurrency.

The graph below shows the time required to complete a synchronizing MPI_Barrier call for a varying number of tasks. Four jobs were run on 8, 32, 64, and 128 nodes respectively. Within each job the timings of MPI_Barrier were measured for MPI communicators of varying size.

MPI_Barrier Scaling



source code comm.c

The impact of concurrency on synchronization times is dramatic, spanning four orders of magnitude. The IBM SP is a cluster of independently scheduled instances of AIX, for this reason it grows increasingly unlikely that at a given moment all the nodes involved will be ready to complete a synchronizing operation.

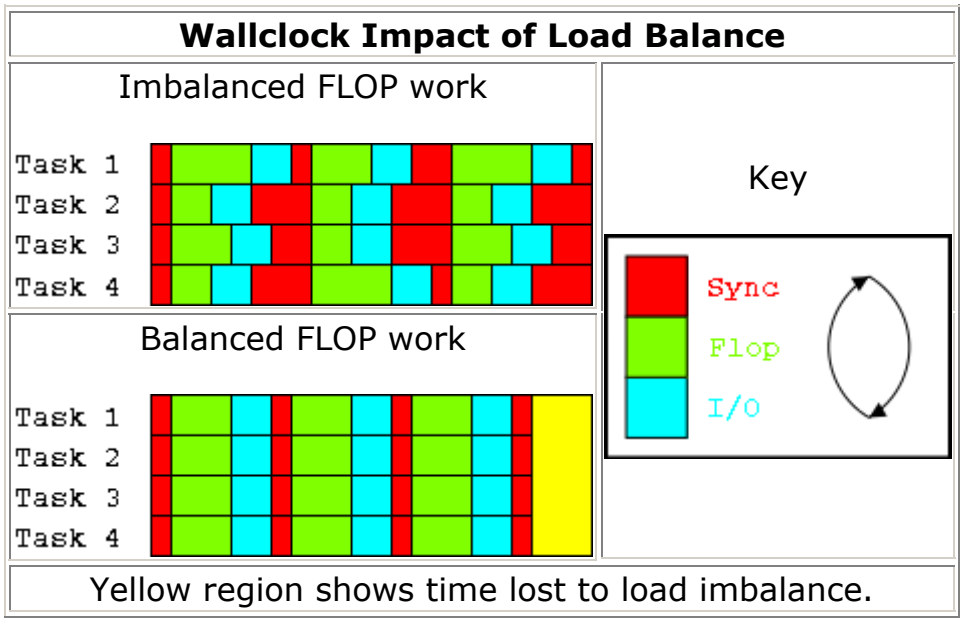
Advice on mitigating the impact of high concurrency global synchronization.

- Identify from a high level algorithmic point of view the places in your code which certainly require full synchronization. Try to aggregate these together in so far as possible in order to accomplish as much of the synchronized work in chunks.
- Where possible use less synchrozing MPI calls.
- Use subcommunicators to restrict synchronization into atomic regions.
- Before inserting an MPI_Barrier first determine if the code is already reliably synchronized at that point in the execution stream or if synchronization is really required by the computation.
- NERSC does what it can to provide quiet, dedicated nodes. If you notice otherwise or have ideas on how to improve this at NERSC, please to contact NERSC staff.

Load Balancing

Aside from the intrinsic delays (due to kernel, OS background activity, etc.) in scheduling a large number of tasks, additional delays can be introduced through unequal partition of work which will further increase the delay when a synchronizing section of the code is reached.

To illustrate the impact of load imbalance on parallel execution consider a hypothetical code running on 4 tasks. The code proceeds through iterations composed of synchronizing, doing work and doing I/O. The time to solution is negatively impacted if the time spent doing work shows wide variance. For such an application the impact of load imbalance scales dramatically with concurrency, as it introduces more serial time into an Amdahl like model for parallel efficiency.



In practice the MPI portion of the code could be in in either or both of the I/O or Sync parts of the code. While load imbalance can occur in these parts of a code (particularly if the I/O involves disk activity), the most important path to enforcing load balance is in evenly partitioning the problem space across the PEs.

Equally distributing problem domains across a processor topology can be made easier through libraries such as Metis which can produce optimal partitions. Scalapack and other parallel libraries have functionality to decompose problems across PEs. It is generally a good idea to check for load imbalance by, e.g., looking at the distribution of HPM statistics when preparing production runs for a new problem or at a new oncurrency.

One may also choose to adopt a partitioning scheme which may be suboptimal in some sense if it makes load balance easier. E.g., a hyperslab decomposition might be chosen over a block cyclic one if it leads to more manageable and predictable load balance. The tradeoff

here is between time lost to the load balance versus time lost to the algorithm itself.

MPI Collectives

It's difficult to provide a taxonomy which describes the level of synchronization required by MPI library calls. This is due to the multitude of ways they can be called and the fact that the implementation may choose certain synchronization rules at run time (e.g., eager vs. rendevous protocols). However, generally speaking it is possible to categorize certain commonly used MPI calls on their overall level of synchronization based on when tasks are allowed to leave an MPI function call.

By MPI collectives we mean MPI routines which all tasks in a communicator call and either

- no task leaves before all tasks (MPI_AllReduce, MPI_Alltoall)

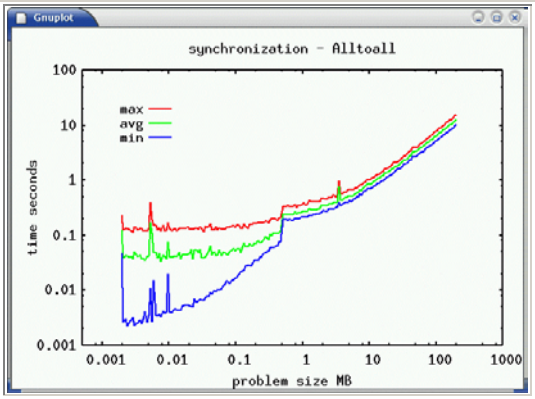
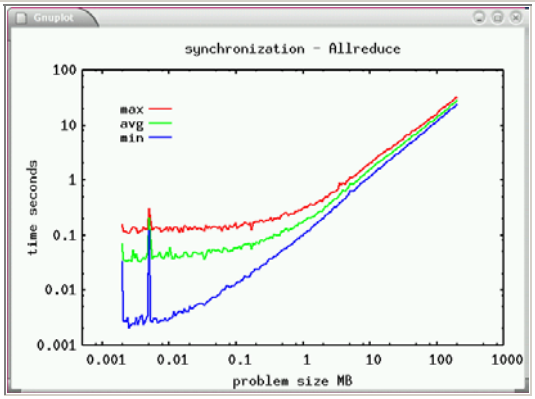
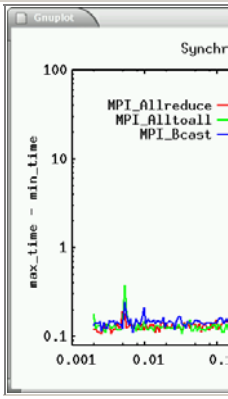
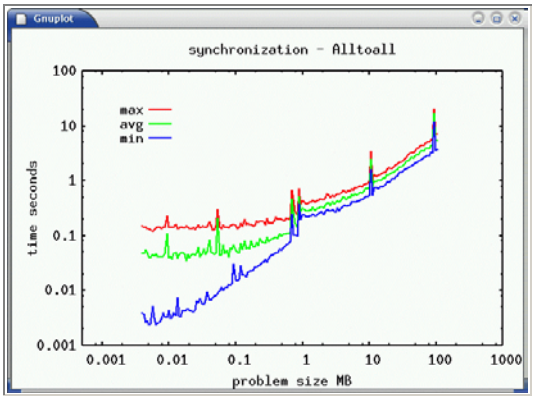
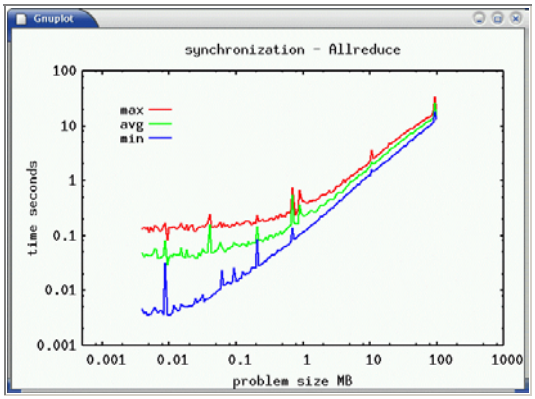
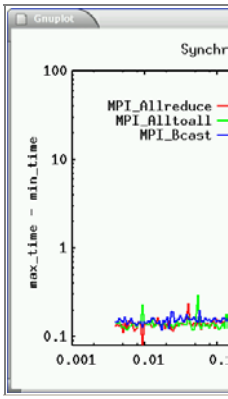
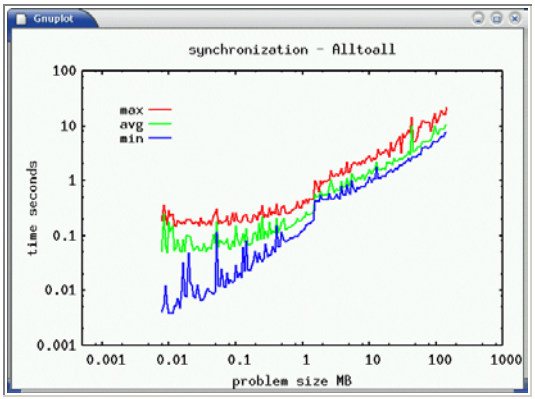
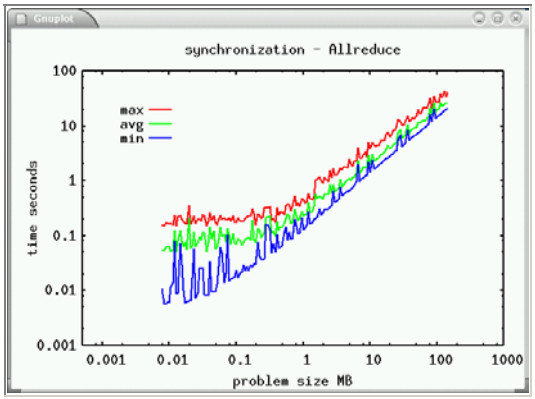
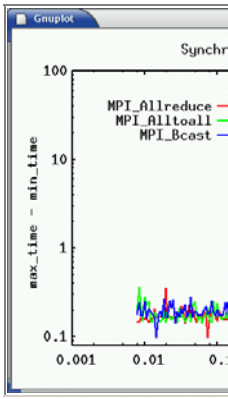
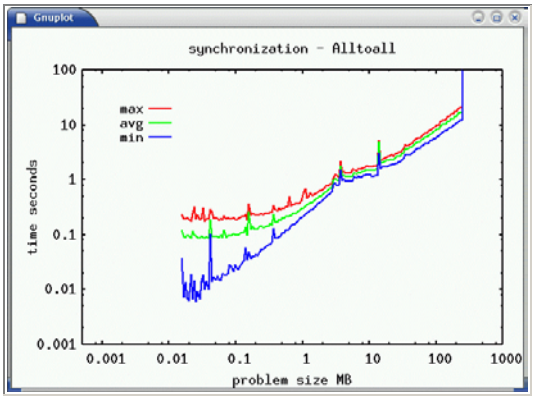
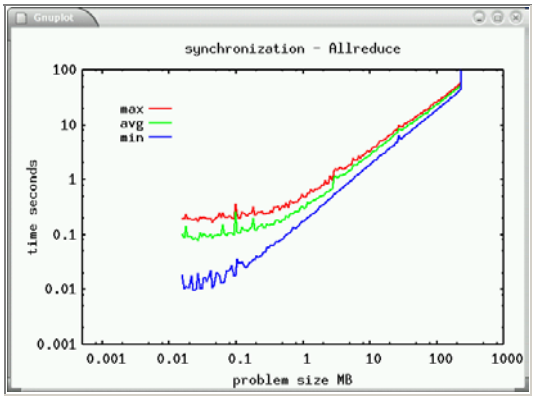
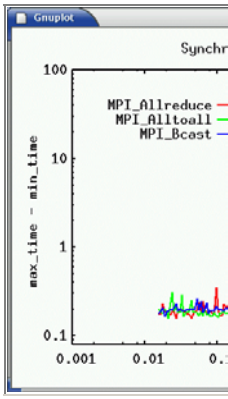
or

- tasks may leave before all tasks enter (MPI_Bcast, MPI_Reduce)

MPI calls which are know to suffer from scaling performance problems are certain MPI collective calls (e.g., MPI_Alltoall and MPI_Allreduce operations). The reasons for this are more complicated than in a simple MPI_Barrier, since not only synchronization, but switch bandwidth and the algorithmic specifics of the MPI calls come into play. Aside from the some degree of synchronization collective MPI calls involve some amount of work in relation to the size of messages being communicated or reduced.

Below the time to complete an MPI_Alltoall and MPI_ALlreduce for varying message sizes is shown:

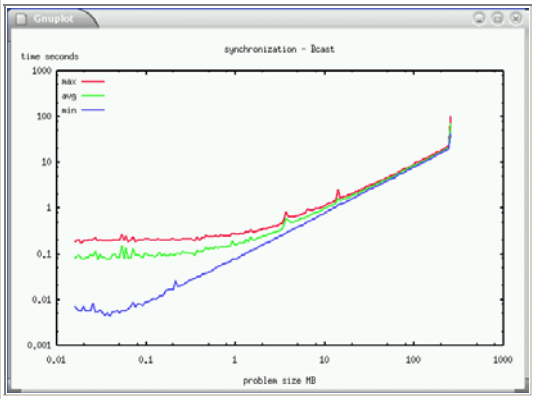
# tasks	MPI_Alltoall	MPI_Allreduce	Sync

256			
512			
1024			
2048			

MPI_Bcast which is not fully synchronizing does not show as large or as varied timings. Though a broadcast and a all-reduce are quite

different, there are cases where not all tasks need the reduced data or do not need at the same time and set of broadcasts can replace a more highly synchronizing collective.

MPI_Bcast



source: [mpimem.c]

One aspect of the scaling of MPI collectives which is not depicted above is the variability in timings. Highly synchronizing calls of any sort not only show greater average delays at higher concurrency, but also show greater relative variability in the length of delays.

Some of the performance shortcomings of the MPI implementation on the SP arise from (or are exacerbated by) certain generalities in the MPI standard. In the interest of standards compliance certain general cases must be taken into account which lead to worse performance than one might expect from a rough estimate. In general, use the most contiguous representation possible when creating MPI_Types.

If the user code/application in question does not make use of MPI_Types or other higher level MPI constructs, e.g., the message data are all vectors of atomic types (double, int, real*8, etc.) it has been demonstrated that better performing (non standards compliant) variants of MPI collectives may written.

IBM's ACTC provides such an MPI variant, Turbo MP. Likewise users have presented their own hand coded MPI collective routines at various conferences . It is important to realize that this version of the MPI library is neither standards compliant nor a supported IBM software product. In evaluating such approaches the user must evaluate the tradeoffs between standards compliance, portability, and performance. As of this writing, this library requires both MPI and LAPI

so in general codes will be limited to 8 tasks per node in order to accomodate 2 switch windows per task.

Example : Reducing MPI_Allreduce

Crucial to the scaling efficiency of most codes is reducing the frequency and duration of synchronizing MPI collectives. Global reductions are often unavoidable from an algorithmic perspective, but the impact on the code performance may often be mitigated by consolidating the global reduction to the smallest set of exchanges.

More Synchronizing

```
      sump=0.0
      do i=1,np
       sum=zero
       do j=1,nz
        jj=j+rank*nz

sum=sum+phi(i,j,ip)*psi(jj,n,il2)
       continue
       sumr=real(sum)
       sumi=aimag(sum)
       call
mpi_allreduce(sumr,sumrt,1,mpi_real8,
     1 mpi_sum,mpi_comm_world,ierr)
       sumr=sumrt
       call
mpi_allreduce(sumi,sumit,1,mpi_real8,
     1 mpi_sum,mpi_comm_world,ierr)
       sumi=sumit

sump=sump+hzz*(sumr**2+sumi**2)
      continue
```

Less Synchronizing

```
       do i=1,np
        sum = zero
        sumpit(i) = zero
        sumpi(i) = zero
        do j=1,nz
         jj=j+rank*nz

sum=sum+phi(i,j,ip)*chi(jj,n,il2)
        continue
        sumpi(i) = sum
       continue



       call
MPI_Reduce(sumpi,sumpit,np,mpi_complex16,
     1
MPI_SUM,0,mpi_comm_word,ierr)
       if(rank .eq. 0) then
        sump=0.0
        do i=1,np
         sump = sump +
     1  (real(sumpit(i))**2 +
aimag(sumpit(i))**2)*hzz
        enddo
       endif
       call
MPI_Bcast(sump,1,mpi_complex16,
     1  0,mpi_comm_world,ierr)
```

MPI Point to Point

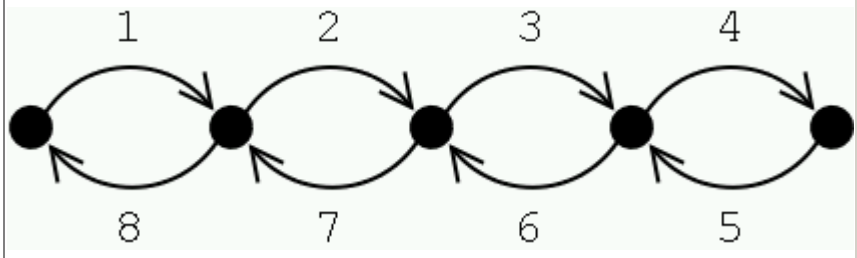
Comparing timings and usage of globally called collectives is much simpler than surveying the space of possible pairwise communications. The pattern of messages in a code using point to point messages will depend closely on the problem being solved and the best we can do here is provide some general information and a few specific examples.

Here we focus on nearest neighbor exchanges which are common to solving PDE's on regular grids. An examination based on dimensionality of the grid (and therefore number of neighbors) along with message size is presented. The information below along with information about the switch should be suffcient to answer most questions about point to point communication on seaborg. If you have further questions feel free to contact NERSC Consultants.

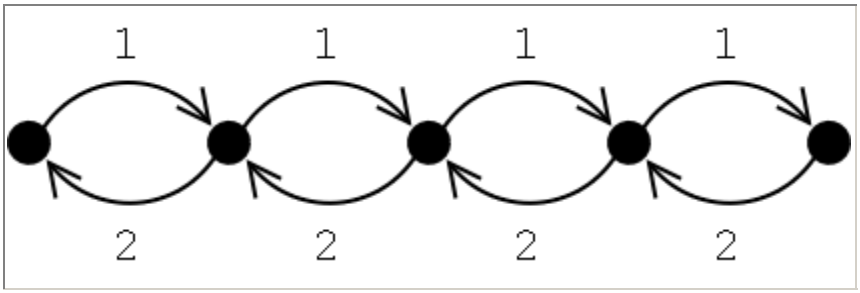
Halo Communication I - Synchronous

A common communication pattern in scientific codes is halo or N-nearest neighbor communication. While the ordering and number of steps will depend on the calculation at hand, there are certain worst case patterns that applicaiton programmers will most always want to avoid.

The following cartoons and fragments from a real code contain two nearest neighbor exchanges that show very different levels of synchronization. The second approach outperforms the second by 40% on 128 tasks.




```
if(rank.ne.size-1) then
 call mpi send(mesh(1,ny),nx,mpi r
       rank+1,1,mpi_comm_world,ier
 call mpi_recv(back,nx,mpi_real8,
       mpi any source,2,mpi comm w
end if
if(rank.ne.0) then
 call mpi_recv(front,nx,mpi_real8,
       mpi any source,1,mpi comm w
 call mpi send(mesh(1,1),nx,mpi re
       rank-1,2,mpi_comm_world,ier
end if
```




```
rankf = rank+1
rankb = rank-1
if(rank.eq.nproc-1) rankf = MPI PR
if(rank.eq.0) rankb = MPI PROC NUL
call mpi sendrecv(mesh(1,ny),nx,mp
       rankf,1,front,nx,mpi_real8,
       rankb,1,mpi comm world,ista
call mpi sendrecv(mesh(1,1),nx,mpi
       rankb,2,back,nx,mpi_real8,
       rankf,2,mpi comm world,ista
```

Choose messaging strategies that wait as long as required (but not longer) to initiate the communucation needed for the computation. The above example includes more delays (though blocking MPI) than are required for each node to exchange boundary data with it's neighbors.

Halo Communication II - Asynchronous

Another common approach to domain decomposed exchanges is to post MPI_Irecvs, MPI_Isends, and then wait for the communication to complete. This asynchronous approach imposes a minimal amount of blocking and exposes opportunity for overlap of communication and computation. In practice on seaborg, the benefit from the former is greater than for the latter.

Below we will compare five strategies that differ in their degree of snychronization. The meshes are set up such that each node has a neighboring rank, neigh[dim][{0,1}], where dim is 0,1,2 (for one, to and three dimensioanl meshes) and {0,1} indiciates the directiona long each dimension.

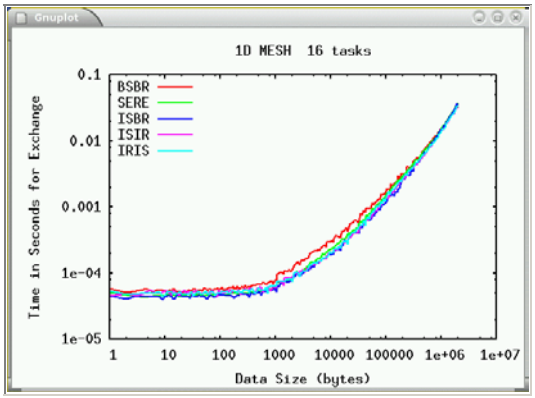
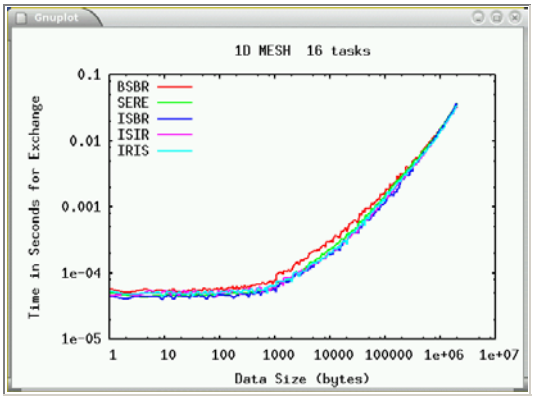
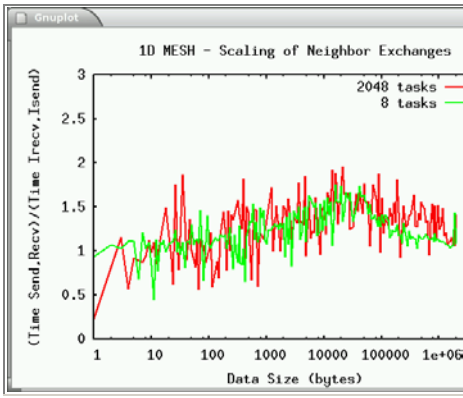
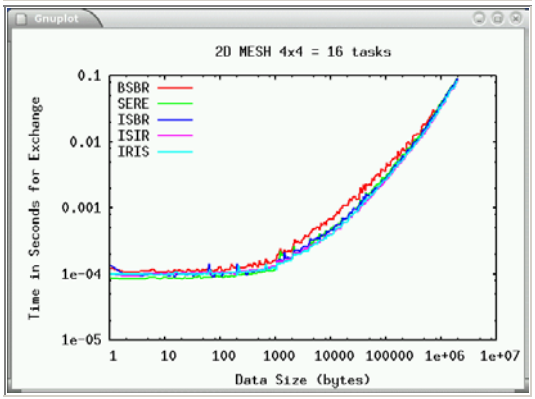
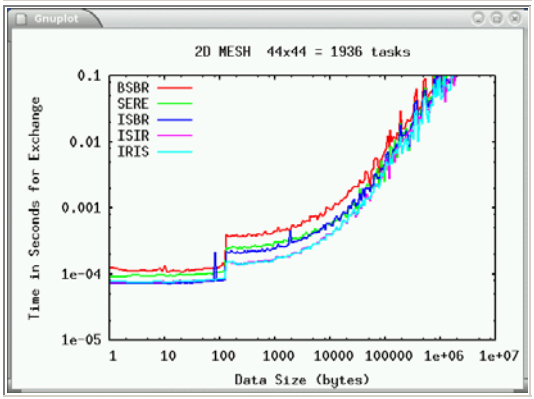
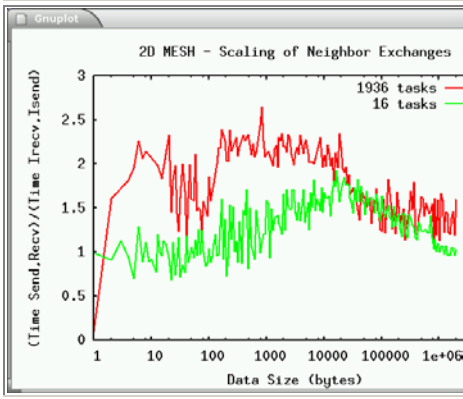
The message exchange stratgies and a simple mnemonic tag for each are:

tag	MPI Calls	code
BSBR	MPI_Send, MPI_Recv	`if(rank%2) { MPI_Send(obuf+0*bytes, bytes, MPI_BYTE, neigh[0][0], 0, comm); MPI_Recv(ibuf+0*bytes, bytes, MPI_BYTE, neigh[0][1], 0, comm, s+0); MPI_Send(obuf+1*bytes, bytes, MPI_BYTE, neigh[0][1], 0, comm); MPI_Recv(ibuf+1*bytes, bytes, MPI_BYTE, neigh[0][0], 0, comm, s+0); } else { MPI_Recv(ibuf+0*bytes, bytes, MPI_BYTE, neigh[0][1], 0, comm, s+0); MPI_Send(obuf+0*bytes, bytes, MPI_BYTE, neigh[0][0], 0, comm); MPI_Recv(ibuf+1*bytes, bytes, MPI_BYTE, neigh[0][0], 0, comm, s+0); MPI_Send(obuf+1*bytes, bytes, MPI_BYTE, neigh[0][1], 0, comm); }`
SERE	MPI_Sendrecv	`MPI_Sendrecv(obuf+0*bytes,bytes,MPI_BYTE,neigh[0][0],0, ibuf+0*bytes,bytes,MPI_BYTE,neigh[0][1],0,comm,s+0); MPI_Sendrecv(obuf+1*bytes,bytes,MPI_BYTE,neigh[0][1],0, ibuf+1*bytes,bytes,MPI_BYTE,neigh[0][0],0,comm,s+1);`
ISBR	MPI_Isend, MPI_Recv	`MPI_Isend(obuf+0*bytes, bytes, MPI_BYTE, neigh[0][0], 0, comm,r+0); MPI_Isend(obuf+1*bytes, bytes, MPI_BYTE, neigh[0][1], 0, comm,r+1); MPI Recv(ibuf+0*bytes, bytes, MPI BYTE, neigh[0][1], 0, comm, s+0); MPI Recv(ibuf+1*bytes, bytes, MPI BYTE, neigh[0][0],`

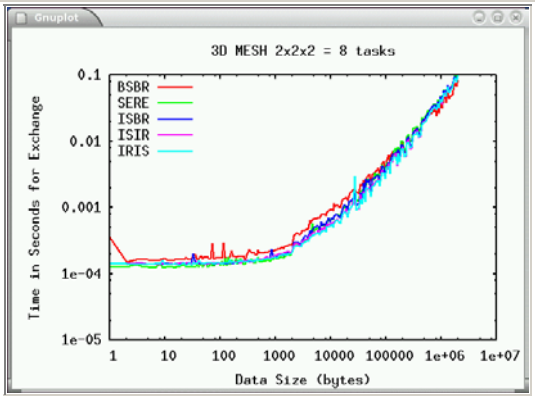
		0, comm, s+1);
ISIR	MPI_Isend, MPI_Irecv	MPI_Isend(obuf+0*bytes, bytes, MPI_BYTE, neigh[0][0], 0, comm, r+0); MPI_Isend(obuf+1*bytes, bytes, MPI_BYTE, neigh[0][1], 0, comm, r+1); MPI_Irecv(ibuf+0*bytes, bytes, MPI_BYTE, neigh[0][1], 0, comm, r+2); MPI_Irecv(ibuf+1*bytes, bytes, MPI_BYTE, neigh[0][0], 0, comm, r+3); MPI_Waitall(4,r,s);
IRIS	MPI_Isend, MPI_Irecv	MPI_Irecv(ibuf+0*bytes, bytes, MPI_BYTE, neigh[0][1], 0, comm, r+0); MPI_Irecv(ibuf+1*bytes, bytes, MPI_BYTE, neigh[0][0], 0, comm, r+1); MPI_Isend(obuf+0*bytes, bytes, MPI_BYTE, neigh[0][0], 0, comm, r+2); MPI_Isend(obuf+1*bytes, bytes, MPI_BYTE, neigh[0][1], 0, comm, r+3); MPI_Waitall(4,r,s);

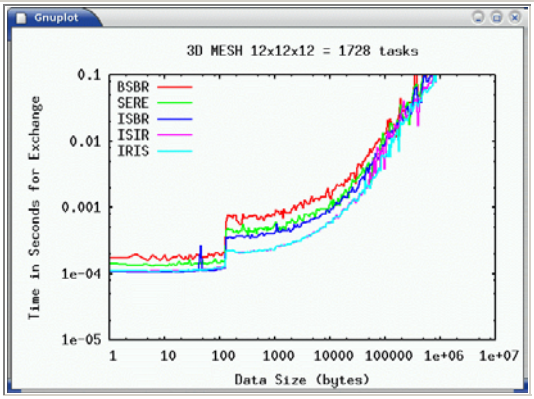
The ibuf and obuf are non overlapping receive and send buffers. The boundary conditions are periodic.

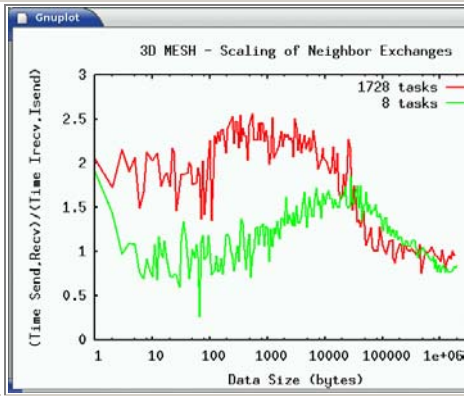
The time to complete an exchange as a funciton of message size is shown below.

ology	Low Concurrency	High Concurrency	Asynchronous Speedup
Ring			
Mesh			

Mesh







This experiment shows the benefit from asynchonous point to point becomes significant at large scale concurrencies. Even more so when the topolgy of the problem include many point to point pairs. For seaborg the timings generally follow the trend:

Isend/Irecv < Isend/Recv < Sendrecv < Send/Recv

Avoiding Synchronization

Avoiding synchronization in so far as possible can be of great benefit to the scaling performance of a parallel code. The way in which a problem is coded has direct impact on the amount and degree of synchronization that happens between tasks. Some amount of synchronization, either partial or global, is intrinsic to most algorithms, other uneccessary synchronization is often the result of a mismatch between the order in which events are specified (in the code) to occur and the order in which they must occcur algorithmically. The structure of MPI makes is easy to unintentionally introduce extraneous points of synchronization which if removed may benefit the scaling performance of the code.

Parallel I/O Scaling

Abstract

Here we will address performance and scaling concerns when moving data from memory to disk.

An important first step in examining a slow running or poorly scaling IO code section is to consider the amount of metadata movement versus user data movement. Similarly to the guiding principle for MPI above, fewer large IO transactions are generally preferable. Frequently writing small temporary scratch files from each task may not noticeably impede a code running 128 way and yet may have significant impact at 1024 way.

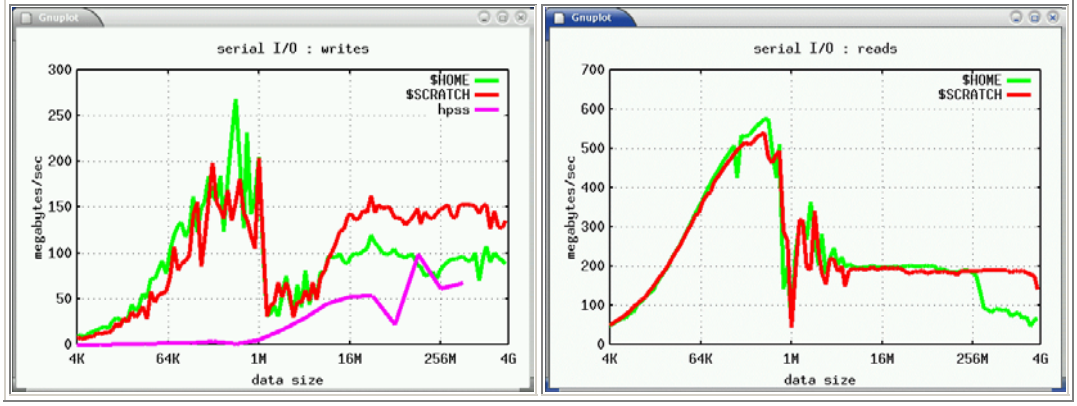
Concurrency, data size and data topology are typically the determining factors in deciding how to accomplish I/O from a parallel code. In what follows the first two will be treated as free variables. Spaning the space of all data topologies is not possible here so the test below are restricted to the following rough sketches; largeblock contiguous, interleaved blocks, and scattered.

File Systems

NERSC's SP relies on GPFS for most user file I/O. Both home ($HOME) and scratch ($SCRATCH) filesystems on mounted globally on all nodes via GPFS.

Users have two main choices for parallel filesystems. All $HOME directories and each of the users $SCRATCH directory are in /usr/common/homefs and /usr/common/scratchfs. The resources mentioned in the hardware section (spindles, adapters, etc GETINFO) are allocated between these filesystems in an asymmetric way. While both filesystem are robust and global across the machine, $SCRATCH has greater bandwidth to disk.

Seaborg File I/O Resources



A small local disk filesystem (/tmp) exists on each node, but this space is tiny and to be used only for AIX and system temporary files. Fortran

and other software which use the TMPDIR variable will write their scratch files to $SCRATCH, a large fast parallel file system in GPFS. In this paper, all considerations of how user file I/O scales are based on GPFS rather than local disk.

The overall parallel strategy of GPFS is to load balance I/O requests across the machine by distributing ownership of data blocks widely across a large number of servers. These participants in the GPFS filesystem are connected via the colony switch which provides considerable bandwidth for data transfers. Modern versions of GPFS (> 1.3) support memory mapped files and most other POSIX I/O functions.

Likewise it can be important to keep in mind that due to the distributed nature of GPFS nodes other that the GPFS server nodes, e.g. batch nodes acting as GPFS clients, can also fulfill requests for disk data. Each client node has a "buddy buffer" of up to 256 MB from which it may serve blocks requested by other clients if such a request permitted based on file system locks.

The GPFS filesystems on seaborg are built from 44 TB of SSA disks, served from 16 GPFS server nodes. A large amount of memory, 32 GB on each server node is available as a cache buffer.

GPFS Basics

Scalability results from the distribution of file data across a large number of nodes. The distribution is at the block level. Blocks are 512Kbytes. Switch data movement is through KLAPI.

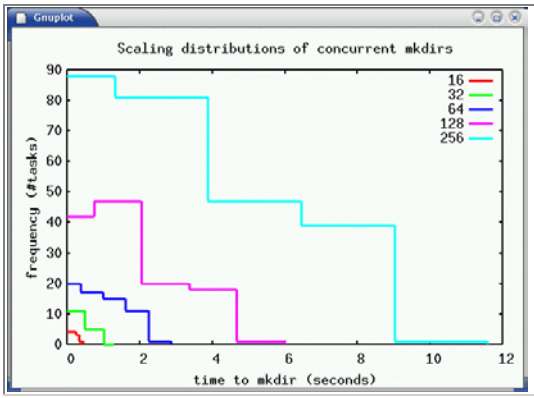
Each node participating as a GPFS client can obtain a write lock for a range of blocks within a file and all or some portion of that data may reside in the client ndoes memory (rather than on disk). If a read request is made for that data the client node may provide the data directly or the transaction may be completed

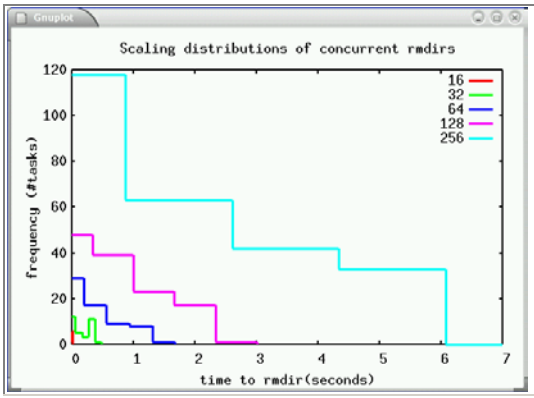
GPFS supports byte range locking, which means that several tasks may read or write disjoint areas of a file without competing for exclusive locks to the file. No special coding, other than making sure the accesses do not overlap is required.

In order to implement parallel I/O well on seaborg it is useful to understand what parallel performance the GPFS filesystems are capable of, but also what types of file activitiy to avoid.

Despite it's distributed nature, GPFS must still use exclusive locks to maintain data coherence and the management of these locks is handled in a more centralized way than the way that the data itself is distributed. While not delving into a detailed description of GPFS metadata, mind that there is a metadata workload associated with each

Scaling of Directory Operations





Calling libc(2) functions from each MPI task (not shell/system calls).

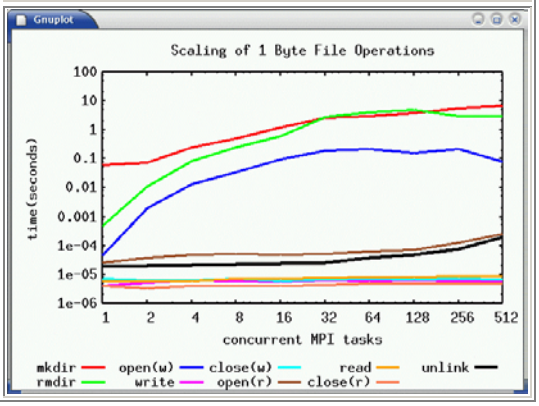
From the above it should be obvious that directory creation or removal from each task of a parallel job is a severe impediment to scaling. Certain I/O operations do not scale well with concurrency as they involve large metadata workloads (inode creation or destruction ~ concurrency) or tax GPFS's token management. Knowing which operations these are is useful when choosing the building blocks of a parallel I/O strategy.

As a rough sketch, this is demonstrated by using a neglibly small data size and testing the impact on concrrency alone on file operations.

pseudocode

```
mkdir(fname,S_IRWXU);
rmdir(fname);
fd = open(fname,O_TRUNC,S_IRWXU);
write(fd,byte,1);
close(fd);
fd = open(fname,O_RDONLY,S_IRWXU);
read(fd,byte,1);
close(fd);
unlink(fname);
```

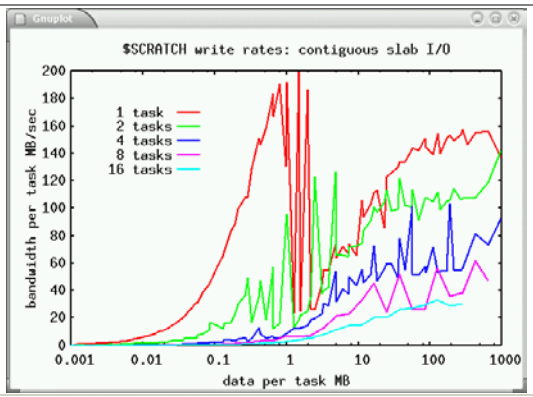
timings



code

The performance considerations arising from the above analysis have to do with near zero (1B) data size per task. While it is useful to understand the impact of concurrency itself, in all real applications tradeoffs between concurency and data size will dominate the decisions made about I/O strategies.

GPFS Performance Contiguous Writes



Parallel I/O Goals

Before turning to the identification of optimal strategies for parallel I/O it's worth clarifying that the important goals are. Typically the primary goal of a parallel I/O strategy is to increase the read/write bandwidths from memory to disk. Other considerations include minimizing on disk storage size, conserving inodes, overlapping I/O with computation, and preserving a particular organization of data within a file.

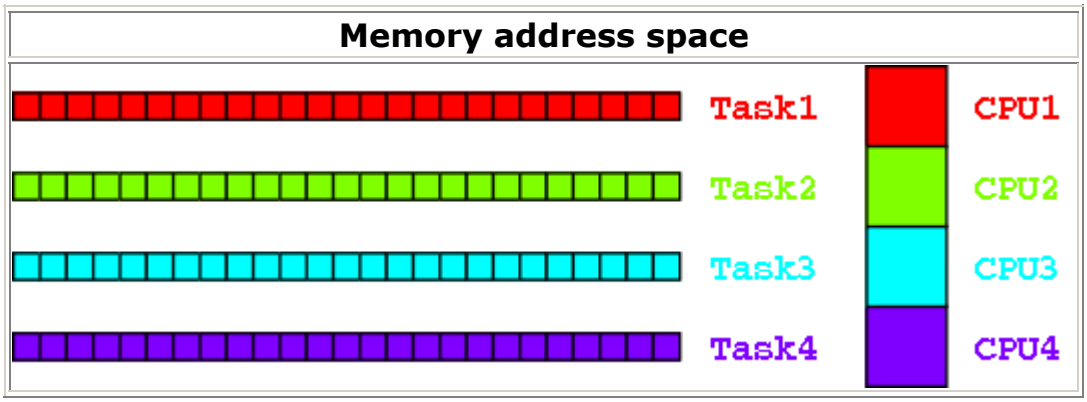
- Maximize Performance
- Conserve disk resources (blocks and/or inodes)
- Maintain file organization or data structure

Realistically, many applications will benefit from a balance of these goals. Many applications and existing codes may have built in constraints or rely heavily on certain I/O strategies which impede performance under GPFS on the SP. By using the quantitative comparisons provided in the next section, researchers may determine at what point the burdens of modifying their code are overtaken by sufficient increases in performance.

Parallel I/O access patterns come in many types, e.g.,

- Structured / Unstructured
- Synchronized / Asynchronous
- Transactional and database access patterns.

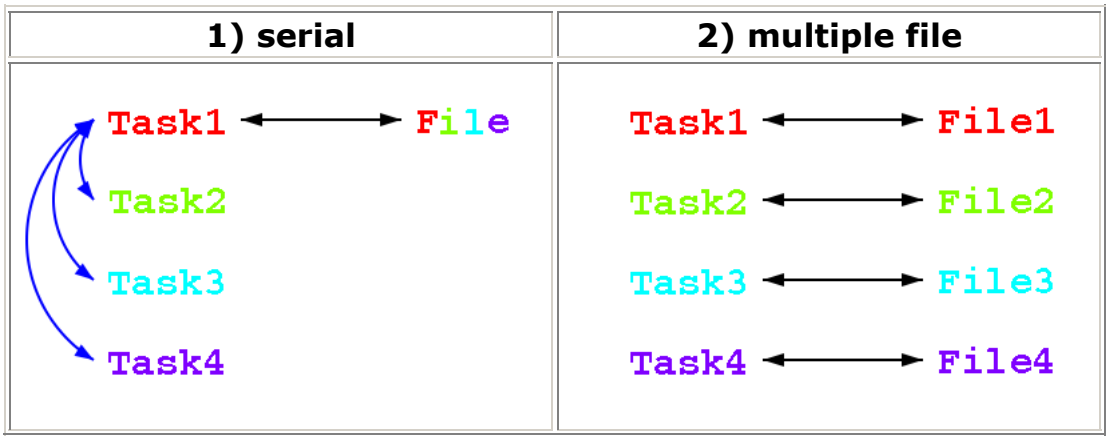
Applications also vary greatly in their built in assumptions about I/O patterns. In the comparisons that follow we will first focus on the simple block I/O pattern in which n tasks each move a block of double precision numbers to and from disk.

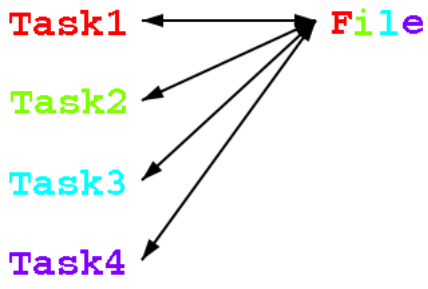
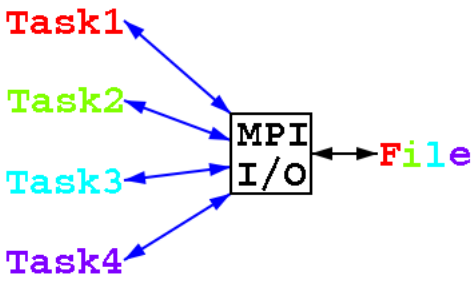


The assumption that the the data is contiguous in the memory address space may not be valid for all applications, but the access times for discontiguous memory organization will typically be orders of magnitude smaller than the I/O times involved. For this reason the primary concern is the structure of the data in the file offset space.

Parallel I/O Strategies

There are many ways to organize the movement of data between memory and disk. Below are diagrams showing four strategies which will be compared in the following sections. Movement of data through MPI is shown in blue and disk I/O is shown in black.



3) POSIX I/O	4) MPI I/O
	

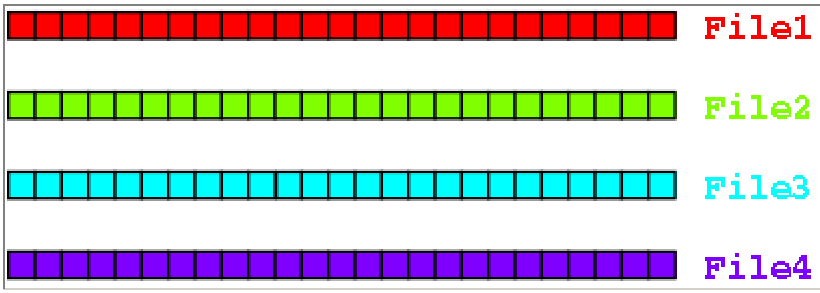
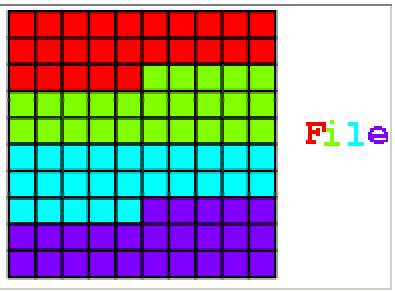
Which strategy is optimal in terms of performance depends largely on the organization of the data on disk and on the total concurrency of the parallel code. Treating a large number of I/O patterns is not feasible so in this writing we will stick to the three identified previously, treating them in turn.

Contiguous File Structure

Hyperslab decomposition of grids is a common type of partitioning which leads to logically contiguous and typically large regions of data in memory and disk.

E.g. task i owns the data in a multidimensional tensor from some lower_index(i) to some upper_index(i) of the slowest running index.

Contiguous memory spaces	
Fortran	`grid(..., lower_index(i))` through `grid(..., upper index(i))`
C	`grid[lower_index(i)]` through `grid[upper_index(i)]`

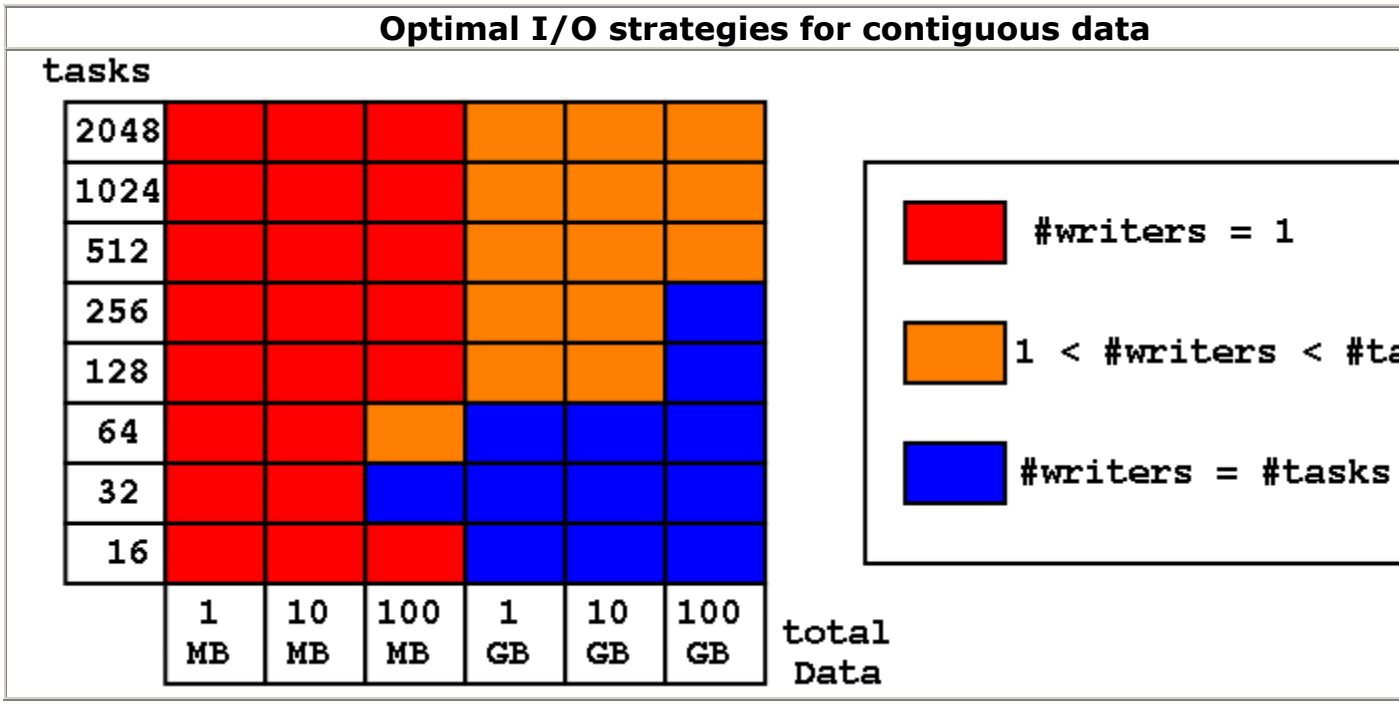
Contiguous file spaces	
multiple files	**POSIX I/O**
	

In this case each task does its `nbytes` of I/O to a separate file or a unique region of a common file.

Multiple Files	POSIX I/O

`t0 = MPI_Wtime();`
 	
`MPI_Barrier(MPI_COMM_WORLD);`
`fp=fopen(fname_rank,"w");`
`fwrite(data,nbyte,1,fp);`
`fclose(fp);`
 	
`MPI_Barrier(MPI_COMM_WORLD);`
`t1 = MPI_Wtime();`
	`t0 = MPI_Wtime();`
	`MPI_Barrier(MPI_COMM_WORLD);`
	`fd=open(fname_global,O_CREAT|O_RDWR,`
	`S_IRUSR|S_IWUSR);`
	`lseek(fd,(off_t)(rank*nbyte)-1,SEEK_SET);`
	`write(fd,data,1);`
	`close(fd);`
	`MPI_Barrier(MPI_COMM_WORLD);`
	`t1 = MPI_Wtime();`

The performance of these strategies are depicted below.

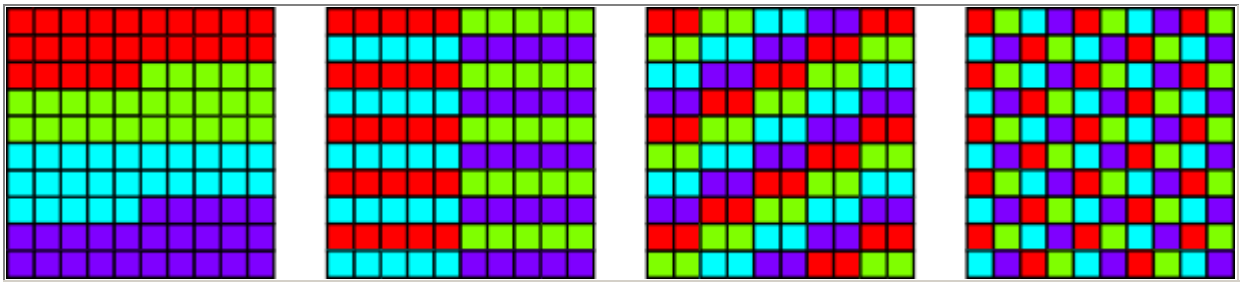


In summary for contiguous I/O patterns the performance have little to do with the topology of data on disk but rather controlling the number of tasks writing concurrently.

Noncontiguous Parallel I/O

This section examines the case of multiple contiguous sections of data. Here both the total size and number of sections are relevant to how to best perform I/O. That is both the concurrency and problem size determine the best I/O algorithm. Block cyclic decompositions common in distributed linear algebra lead to this sort of blocked file structure.

Noncontiguous file structure



decreasing block size ------>

For suffciently small block size the metadata and file locking tasks required from GPFS are expected to impede performance. Each of the I/O operations, taken individually is more complicated for the filesystem than if coordianted at a global level by MPI-I/O. For this reason MPI-I/O should be able to provide the benefit when the block size is small enough.

POSIX-I/O

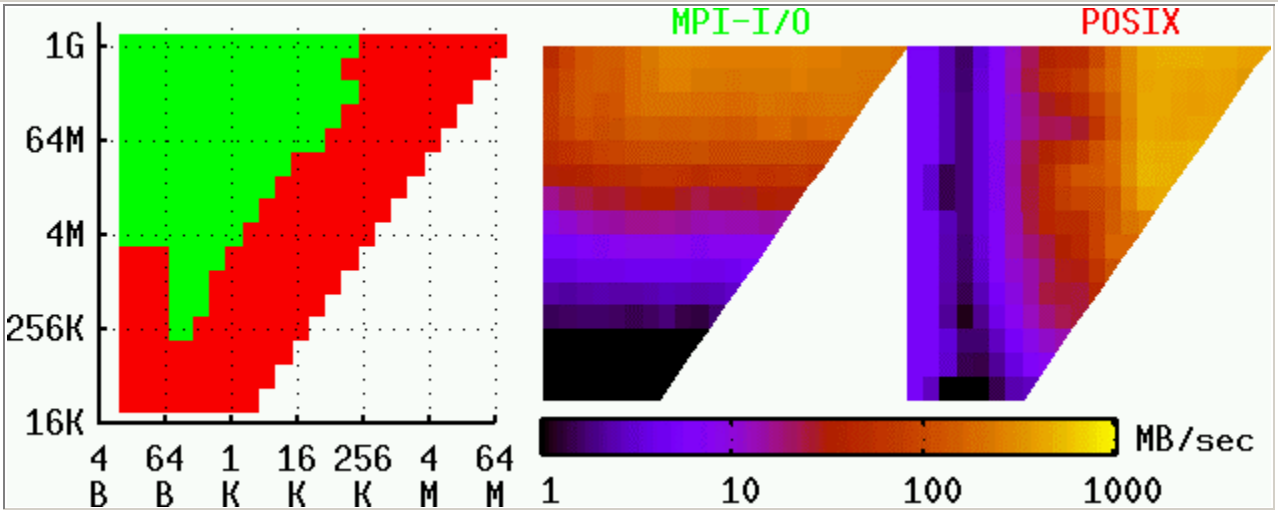
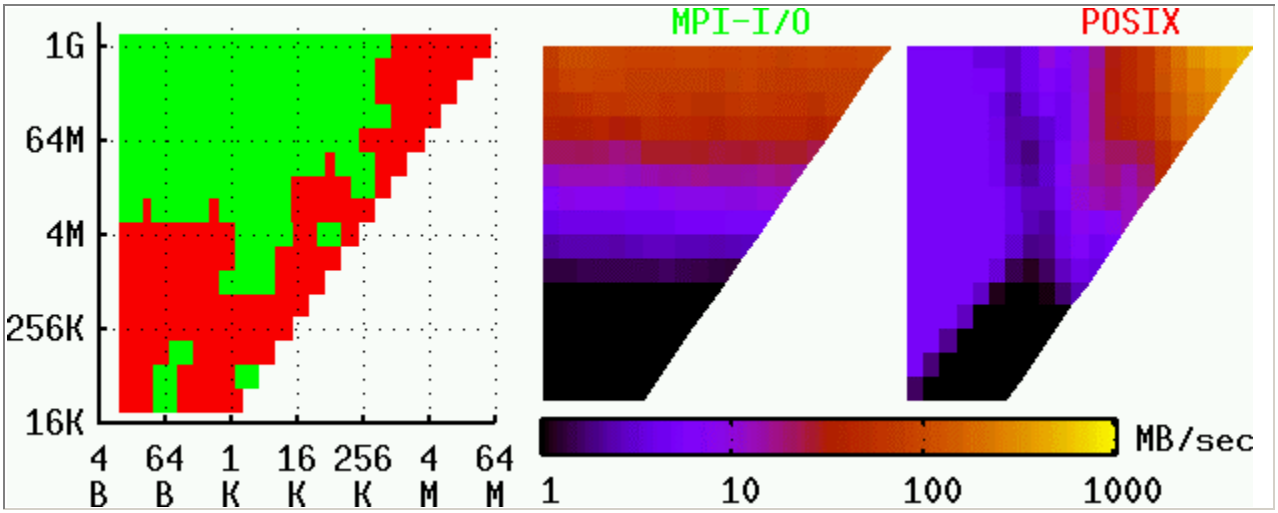
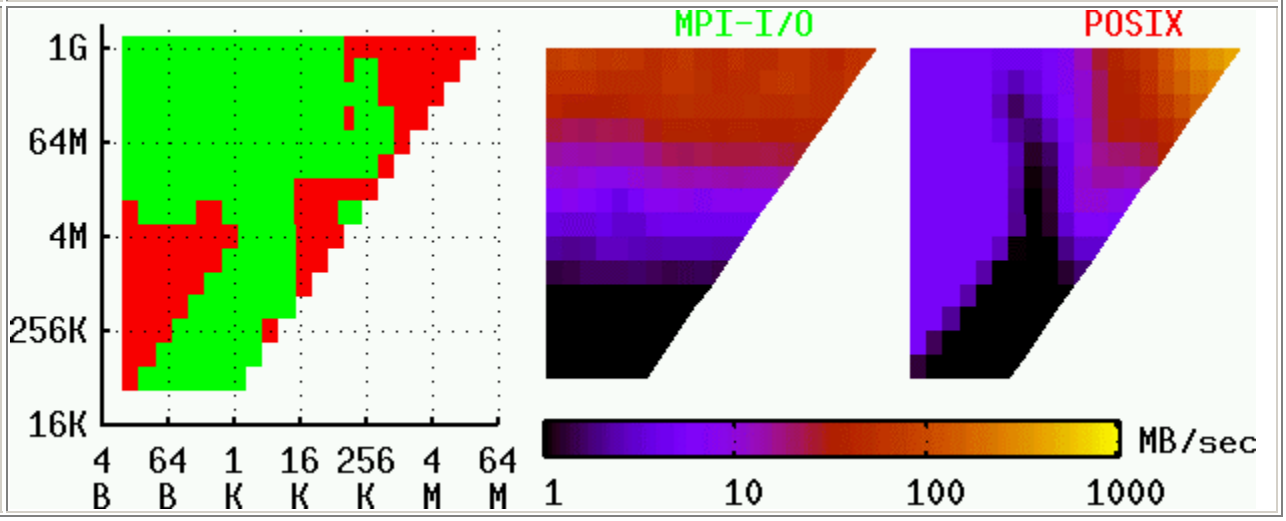
```
   t0 = MPI_Wtime();
   fp=fopen(fname,"w");
   MPI_Barrier(MPI_COMM_WORLD);
   for(i=0;i<n/bn;i++) {
    fseek(fp,
     (off_t)((i*size +
rank)*bn*sizeof(DATA_T)),
     SEEK_SET);

fwrite(data+i*bn,bn*sizeof(DATA_T),1,fp);
   }
   fclose(fp);
   MPI_Barrier(MPI_COMM_WORLD);
   t1= MPI_Wtime();
```

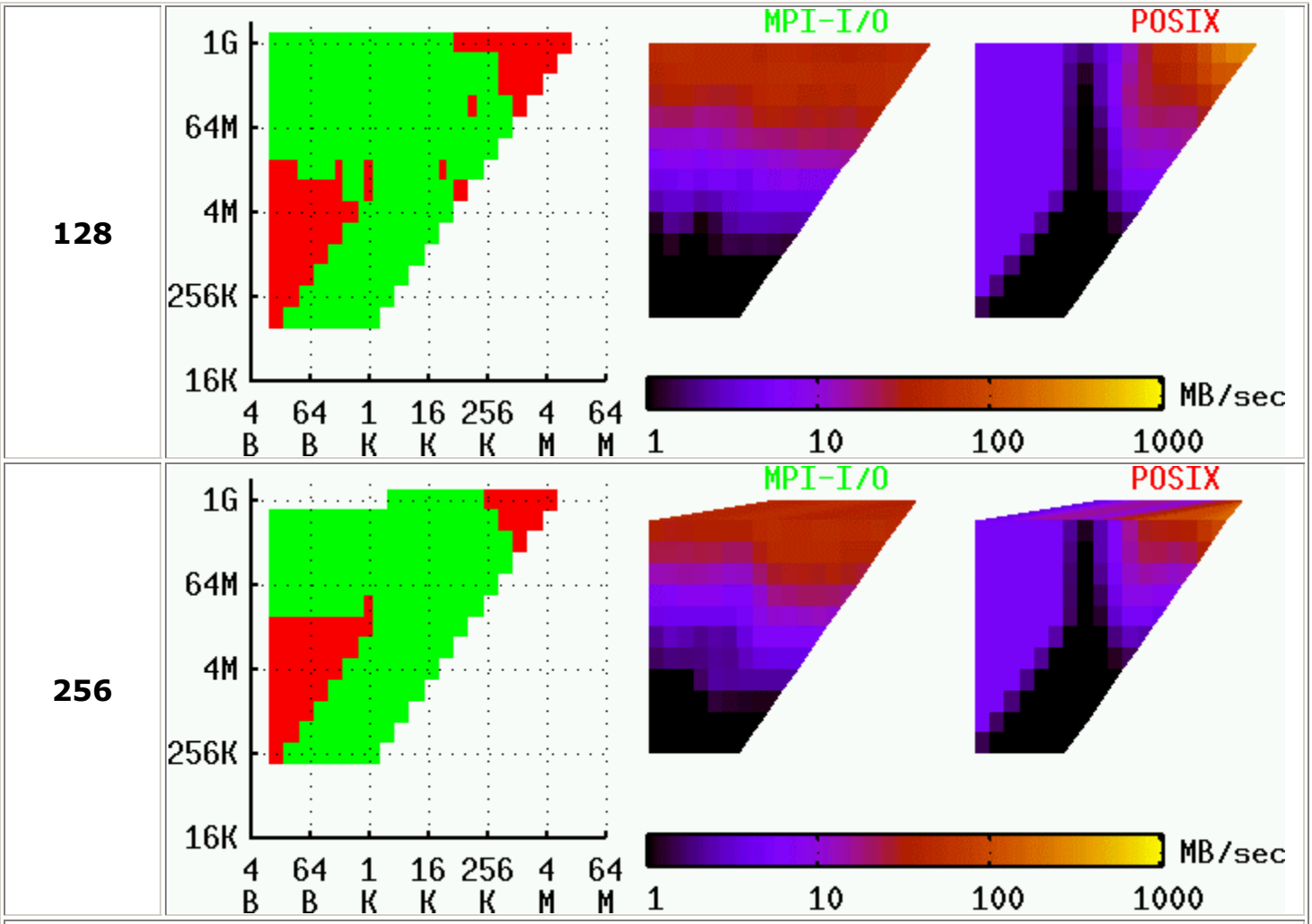
MPI-I/O

```
   t0 = MPI_Wtime();
   MPI_Type_vector(n/bn, bn,
size*bn, MPI_DOUBLE, &vectype);
   MPI_Type_commit(&vectype);
   MPI_Type_size(vectype,&bvect);
bvect/=sizeof(int);
   MPI_File_open(MPI_COMM_WORLD,
fname,
    MPI MODE CREATE | MPI MODE RDWR,
MPI_INFO_NULL, &fh);
   MPI_File_set_view(fh,
rank*bn*sizeof(double), MPI_BYTE,
         vectype, "native",
MPI_INFO_NULL);
/*
MPI File preallocate(fh,nbyte*size);
*/
   MPI_File_write_all(fh, data,
bvect, MPI_INT, &s);
   MPI_File_sync(fh);
   MPI_File_close(&fh);
   MPI_Type_free(&vectype);
   MPI_Barrier(MPI_COMM_WORLD);
   t1 = MPI_Wtime();
```

Testing the above strategies on seaborg's $SCRATCH filesystem shows the following results.

MPI_Size	Parallel Write Strategies and Rates

16	 1G 64M 4M 256K 16K 4 B 64 B 1 K 16 K 256 K 4 M 64 M MPI-I/O POSIX MB/sec 1 10 100 1000	
32	 1G 64M 4M 256K 16K 4 B 64 B 1 K 16 K 256 K 4 M 64 M MPI-I/O POSIX MB/sec 1 10 100 1000	
64	 1G 64M 4M 256K 16K 4 B 64 B 1 K 16 K 256 K 4 M 64 M MPI-I/O POSIX MB/sec 1 10 100 1000	



In each graph: x axis is the block size, y axis is total data size. First graph shows winning I/O strategy.

Aspects of parallel I/O are demonstrated above:

- I/O performance increases as the block size increases. When possible use large/contiguous transfers to disk. An IBM specific file hint exists which when used helps recover the performance difference between the two strategies for very large block sizes. It's notable that the relation between block size and performance is not however monotonic.
- The important role played by MPI-I/O in aggregating small I/O requests together is seen in the the write performance of MPI-I/O extends further into regions of smaller block size than POSIX I/O.
- Even in regions of reasonable I/O performance, the average performance for a fixed problem size decreases with

concurrency. This is a commonly encountered isoefficeincy issue when scaling up parallel applications. If the increase in parallelism is not matched by an increase in the total data involved then the I/O requests will necessarily be smaller and as a result less efficient.

Conclusions

Implementing parallel scientific codes that scale well can be challenging. This document has tried to address some of the impediments to concurrency faced by researchers deploying HPC codes on NERSC hardware. The goal of such an endeavor is not scaling itself but rather the advancement of scientific research through computation. The choices in the level of parallelism that most effectively drives that advancement can be hopefully be expanded by such an examination.

If you have questions about parallel code scaling on seaborg not answered in this document, feel free to contact the author.

References

Useful Documents

- Scientific Applications in RS/6000 SP Environments
- RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide
- RS/6000 SP: Practical MPI Programming
- Understanding and Using the SP Switch
- NERSC IBM SP Memory Page
- NERSC MPI-I/O Page
- RS/6000 SP 375MHz POWER3 SMP High Node Overview , August 29, 2000
- On seaborg see, /usr/lpp/LoadL/README S/LoadL.README and /usr/lpp/LoadL//README S/poe.README

Microkernels and Test Codes

- triad.f

- mpimem.c
- p2p.c
- comm.c
