DEPLOYING PARALLEL NUMERICAL LIBRARY ROUTINES TO CLUSTER COMPUTING IN A SELF ADAPTING FASHION

KENNETH J. ROCHE, JACK J. DONGARRA
Department of Computer Science, The University of Tennessee,
203 Claxton Complex,
Knoxville, Tennessee 37996-3450
roche,dongarra@cs.utk.edu

This paper discusses middleware under development which couples cluster system information with the specifics of a user problem to launch cluster based applications on the *best* set of resources available. The user is responsible for stating his numerical problem and is assumed to be working in a serial environment. He calls the middleware to execute his application. The middleware assesses the possibility of solving the problem faster on some subset of available resources based on information describing the state of the system. If so, the user's data is redistributed over the subset of processors, the problem is executed in parallel, and the solution is returned to the user. If it is no faster, or slower, the user's problem is solved with the best appropriate serial routine. The feasibility of this approach is empirically investigated on a typical target system and results reported which validate the method.

1 Overview

On multi-user computing systems in which the resources are dedicated but the allocation of resources is controlled by a scheduler, one expects a job, once allocated, to finish execution in a predictable amount of time. In fact, the user on such a system usually submits a job through a batch script which requires an upper bound on the predicted runtime of the task being queued. The user is billed for the entire time requested and thus has a responsibility to himself and other users to understand the behavior of his code. Scheduling schemes on such systems attempt to order the computations in a fair manner so that user jobs show progress towards completion in a timely fashion while the overall system throughput is maximized. One problem with this approach is that such a scheduling scheme is easier to talk about than to implement.(ref. [1]) It is a fact of life that the *multi-processor scheduling* problem is NP-complete in the strong sense.[a] Thus, developers must *look for* algorithms which are

[a] Theoretically, a decision problem Π is NP-complete in the *strong* sense if $(\Pi \in NP) \wedge (\exists \Pi_p$ which is NP-complete). For decision problem Π and polynomial p, defined over the integers, Π_p is the restriction of Π to instances $I \ni Max[i] \leq p(Length[I])$. If Π is solvable by a pseudo-polynomial time algorithm, then Π_p is solvable in polynomial time. Consider the

efficient. This is a difficult and time consuming task (which can't really be avoided on production scale or commodity clusters such as those at national laboratories and supercomputing centers). The large variance in the average duration and demand of user jobs, for instance, further complicates the task. It's complicated. Even though there's no provably optimal way of addressing this problem, people do it all the time because it has to be done. We don't make further considerations of such systems in this paper.

In shared, multi-user computing environments, such as clusters of workstations in a local area network, the notion of *determinism* in computations can be lost due to resource contention. Thus, successive runs of the same linear algebraic kernel with the same problem parameters, for instance, may result in grossly different wall clock times for completion due to the variability of the work load on the CPUs from competing tasks. If users compute responsibly and in coordination with one another such systems can be and are successful. (Administrators intervene otherwise to mediate serious contention problems.) This approach is often more efficient, for instance, in the developmental stage of parallel application codes due to the constant testing and debugging of software, or in groups where the average user job tends not to saturate the system resources and runs to completion on a relatively short time scale (*e.g.* minutes or even hours). One way for the user to possibly make better use of the available resources in such an environment is to employ the information describing the state of the computational system at runtime to *select* an appropriate subset of resources for the specific kernel at hand. It is acknowledged that making low risk predictions in such a system when multiple users are sharing the resources cannot be done with certainty. There is nothing to preclude the event that the demand on the resources may change dramatically in the time which transpires between deciding on a set of resources and getting one's problem set up and ready to go on the resources. Nonetheless, it *seems* negligent not to try to use available system related data at runtime. In the very least a user can identify saturated resources in the system and avoid allocating them for his/her run. In the event that the overall system behavior fluctuates about some time sensitive *normal* level of activity, then a statistical analysis of the collected data can be used as the basis for a predictive model of the system's behavior at some future time. Software, such as NWS, the

---

multi-processor scheduling problem Π_{MS}: given a finite set J of jobs, a length $l(j) \in Z^+ \forall j \in J$, a number, $m \in Z^+$, of processors, and deadline $D \in Z^+$, is there a partition $J = J_1 \cup J_2 \cup \cdots \cup J_m$ of J into m disjoint sets such that $max[\sum_{j \in J_i} l(j) : 1 \leq i \leq m] \leq D$? This problem is NP-complete in the strong sense and thus cannot be solved by a pseudo-polynomial time algorithm unless $P = NP$. (For *proof* see reference [2], for related information see references [3, 4, 5, 6, 7, 8].)

Network Weather Service, operates sensors in a distributed computing environment and periodically (in time) collects measured data from them.(ref. [9]) NWS includes sensors for end-to-end TCP/IP performance (bandwidth and latency), available CPU percentage, and available non-paged memory. The collected data is kept and analyzed as a time series which attempts to forecast the future behavior of the system through low order $ARMA$, *autoregressive moving averages*, methods.

This paper discusses software being developed which couples system information with information specifically related to the numerical kernel of interest. The model being used is that a user is assumed to contact the middleware through a library function call during a serial run. The middleware assesses the possibility of solving the problem faster on some subset of available resources based on information describing the state of the system. If so, the user's data is redistributed over the subset of processors, the problem is executed in parallel, and the solution is returned to the user. If it is no faster, or slower, the user's problem is solved with the best appropriate serial routine.

It is conjectured that if the underlying application software is scalable then there will be a problem size which marks a *turning point*, N_{tp}, for which the time saved because of the parallel run (as opposed to the best serial runs for the same problem) will be greater than the time lost moving the user's data around. At this value, such software is deemed useful in the sense that it provides an answer to the user's numerical question faster than had the user done as well as an expert working in the same serial environment. That is, it benefits even the expert user working on a single node of the shared system to use the proposed software for problem sizes in which $N_{user} > N_{tp}$.

As a case study we consider the problem of solving a system of dense, linear equations on a shared cluster of workstations using the $ScaLAPACK$ software.(ref. [10]) A discussion of some specific implementations tested is made and results of selected experiments are presented. It is observed that even with naive data handling the conjecture is validated in a finite ensemble of test cases. Thus there is motivation for future studies in this area. It is also observed that the expert user in the parallel environment can always complete the dense, algebraic task at hand faster than the proposed software. (There are no clear winners for small problem sizes since each approach solves the problem serially with the best available version of the library routine.) This is no surprise since even in the most ideal cases, the proposed software has to *touch* the user's data at least enough to impart the relevant data structure expected by the logical process grid. The parallel expert, on the other hand, is assumed to be able to generate relevant data structures in-core, in-parallel at the time of the distributed run. This user also knows how to initialize the

numerical library routine, and make compilation time optimizations. He/she is probably not the typical scientist who has likely already labored just to reduce their problem to linear algebra. There are, in fact, many details to account for by any user before the parallel kernel runs correctly. Reporting the results of this expert user provides a basis for comparison to the other projected users and scenarios.

2 Numerical libraries in shared, homogeneous, distributed environments

2.1 The computing environment

In the development of the current investigation heterogeneous grid computing systems have not been the central focus. (See references [11,12,13,14,15,16,17].) However, it is noteworthy that one of the goals in resource selection when considering a pool of heterogeneous (and potentially geographically distributed) candidate resources is to achieve as much *homogeneity* in the allocated resources possible. Here's at least one complication of scheduling in a shared distributed system which is general to both heterogeneous and homogeneous systems: *the scheduler of resources for a task in question has to try and allocate resources which not only look homogeneous at the instant of inquiry, but remain as homogeneous as possible for the duration of time that the task is in (parallel) execution.* In short, even if we could solve the general *multi-processor scheduling* problem at some specific instant in time, we can't count on this partitioning to assist us in forecasting the state of the system resources at some time in the future. This is due to the fluctuating properties of system resources which one can observe in a shared environment. This brief subsection intends to describe the notion of homogeneity in the context of the current study. Some sample results of timing various operations in one of the systems tested demonstrates the notion as it is observed empirically.

In complex mechanical systems the notion of homogeneity usually implies that the system behaves in a predictable manner when performing a specific task only in the absence of external influences. If this definition applies to computational systems (see Figure 1 for a sample computing environment), then it cannot be that a shared set of resources alone, such as a cluster of workstations in a local area network, is homogeneous. Usually such a system is only meaningful when responding to a user's requests. Users' requests are developed externally and then serviced by the system at runtime. Since there is no way to know when a user intends to make requests in such an open

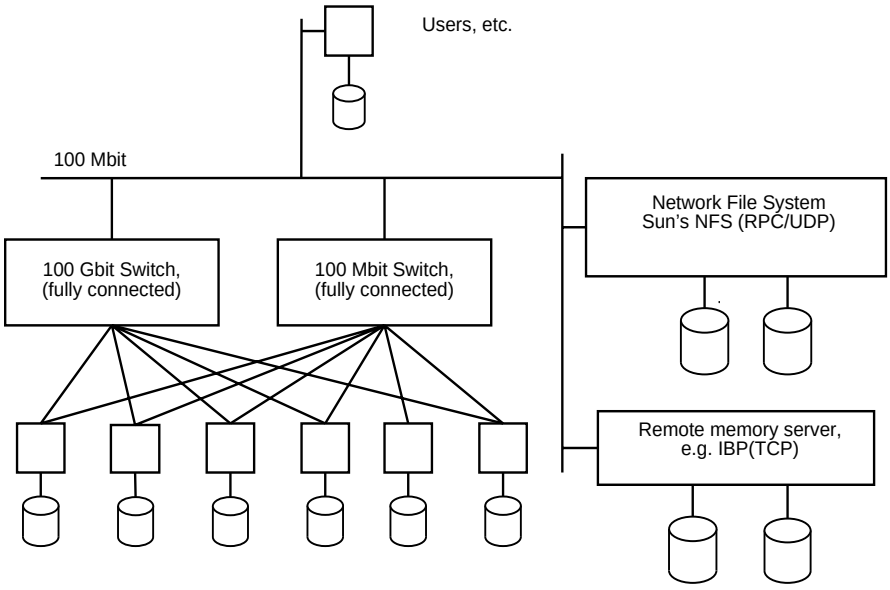

Figure 1. The figure is a diagram of part of the local area net work in which many of the current investigations were made. It is noteworthy for the purposes of interpreting some of the results presented in this paper that the memory depot and network file server are separate machines in reality sitting on a shared network. The clusters on which we have developed the current study are removed from the shared network through one of two switches through which the cluster of workstations is said to be fully connected. It is a factor for the types of studies we have made that there is only one, $100Mbit$ line for all of the network flow to and from the network disk or the memory server.

system, any specific task, such as solving a set of linear equations, is likely to exhibit different total wall times for completion on subsequent runs. So what does one mean by a shared homogeneous, distributed computing environment? Naively, it is assumed that the hardware specifications and available software are duplicated between compute nodes on such a system. This is not enough however (and may not be necessary). The notion of homogeneity has meaning only in terms of some specific system state engaged in some specified activity as observed in an ensemble of test cases. Let us elaborate on this thought a little.

Physical system parameters, when observed at equidistant time intervals and kept in collections of ordered data, comprise a time series.(see ref-

erences[18,19,20,21]) Because of the inherent fluctuations in the system parameters, CPU loads on a shared cluster for instance, the time series of the state of the shared system is a non-deterministic function. (For instance, the activity level of system resources often reflects the underlying nature of the humans which use the system. At lunch time, dinner time, or bed time one often observes a decrease in overall system activity. In the morning, one often observes some adiabatic growth of activity in the system as users arrive and begin working. This growth continues until some *normal* level of system activity is achieved. For some duration of time, one may expect the system resource activity levels to fluctuate around this normal. But the point is that the activity norm is *time of day* dependent more often than not.) Non-deterministic time series can only be described by statistical laws or models. To study such systems formally one assumes that a time series can only be described at a given instant in time, t, by a (discrete) random variable, X_t, and its associated probability distribution, f_{X_t}. Thus, an observed time series of the system parameters can be regarded as one realization of an infinite ensemble of functions which might have been generated by a stochastic process -in this case multi-users sharing a common set of resources as a function of time. Stochastic processes are strictly stationary when the joint probability distribution of any set of observations is invariant in time and are particularly useful for modeling processes whose parameters tend to remain in equilibrium about a *stationary* mean. Any stationary stochastic process can be described by estimating the statistical mean μ ($\overline{x} = n^{-1}\sum_{t=1}^{n} x_t$), its variance σ^2 ($s_x^2 = n^{-1}\sum_{t=1}^{n}(x_t - \overline{x})^2$), the autocovariance function ($c_{xx}(k) = n^{-1}\sum_{t=1}^{n-k}(x_t - \overline{x})(x_{t-k} - \overline{x})$ *e.g.* the extent to which two random variables are linearly independent), and the sample autocorrelation function which is a kind of correlation coefficient ($r_{xx}(k) = c_{xx}(k)(c_{xx}(0))^{-1}, k = 0, ..., n-1$). A discrete random process for which all the observed random variables are independent is the simplest form of a stationary stochastic process. For this process, the autocovariance is zero (for all lags not zero) and thus such a process is referred to as *purely random*, or *white noise*.

Usually, the observable system parameters such as CPU loads and available memory are not independent. Thus, the notion of homogeneity is manifest only in observing a specific task on the system, such as data I/O or multiplying matrices in-core, repeatedly under normal system activity on each of the computing nodes said to comprise the system. An expectation value (μ) for the specified task will be formed for each unit only in this time tested man-

ner.[b] One can subsequently compare the results from each of the units and determine a level of similarity between them. Ideally, the time to complete any serial task would be the same within some standard deviation (approximated by the square root of the variance) regardless of the compute node executing it. Further, the error bars should ideally tend to zero as the number of observations tends to infinity. This is not achievable in practice, clearly. (For non-stationary processes, one *filters* the available data sets thus transforming the problem into a stationary form.)

Figures 2, 3, 4, and 5 illustrate the results of portions of an empirical study on one of the systems used to develop the current investigation. In each of the figures, successive runs of a task are time stamped and recorded to file. The results presented were analyzed statistically and only the mean and root of the variance are reported. The runs were conducted over the course of weeks and, as much as possible, during the same hours of the day (10am until 5pm,EST). No attempt was made to push users off the shared system accept during development of some test cases.

In Figure 2, an assessment of the CPU of each node when executing (serial runs) a numerical kernel rich in floating point operations as a function of the numerical problem size is made. In this case the time to solution and performance are reported and there is a clear correlation in the expected behavior of each node.

Figure 3 is composed of three plots. The plots look at the *read* and *write* I/O times per node as a function of bytes on each node's local disk, on the local network users' disk (in our case operating under Sun's NFS - utilizing RPCs, UDP/IP), and on a memory server (running IBP[c] (ref. [23]), TCP/IP) on the same local network but with a different IP address from any node on the cluster or the NFS server itself. In plot one, the local disk accesses, one can only imagine that sporadic user activity is responsible for the larger variances in some of the reported results. To within the error bars

[b] If observational data is to be of use in developing advanced software, a standard metric has to be agreed upon and needs to be reliable across different platforms. $PAPI$ (ref. [22]), which accesses hardware counters as well as the system clock, provides such a tool and has been used throughout this investigation when recording observations.

[c] IBP, the Internet Backplane Protocol, is software for managing and using memory on a distribution of disks across a network. Its design is really intended for large scale, logistical networking. However, because it was designed with a client/server model in mind, is also useful for our purposes (as will be discussed further in this report.) The client accesses remote resources through function calls in C, for instance. The multi-threaded servers have their own IP addresses. The client has to know this address in advance as well as to which port the server listens. Usually one sets up his own IBP depot and can choose the port. The IBP group also manages some public domain (*free*) depots across the country which one can use.

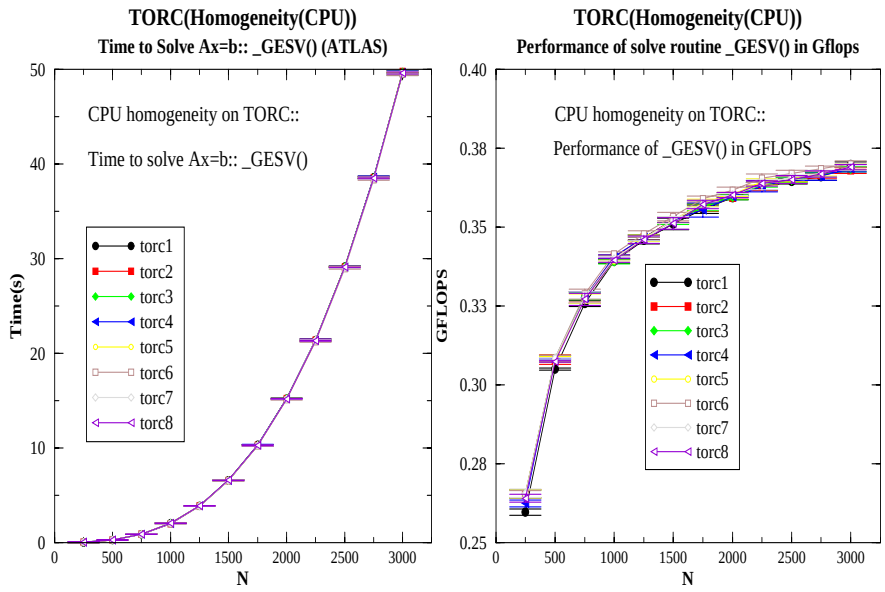

Figure 2. Performance and time to completion numbers for the serial, dense, linear solve routine *gesv*() from *ATLAS* are reported as an example. CPU homogeneity in the shared cluster is a very important criterion for developing numerical software intended for distributed environments.

the wall times reported are within seconds of one another and thus invoke some sense of homogeneity. In the software designed to date, we don't make explicit use local disk I/O. In plot two, the accesses to the local network disk as controlled by *NFS*, we again see fluctuations, in particular during the UNIX system *read*s. This data is of particular interest to us, as will become clear in the sections to follow. It is recalled that to access (*NFS* controlled) the network disk that data is moved over a single shared local communication line. Further, multiple users tax *NFS* due to the design of the shared local file system. One expects larger variances here. Nonetheless, the results again instill some confidence within the expected error. Further, we have to deal with reality. The systems being tested *a*re supposedly identical. However, in open systems we must work with the expectation that this notion is a fallacy -we can only make sensible predictions within some confidence limits which are set by the actual behavior of the system. It is fun, however, to guess at why the *write*s appear to have much tighter error bars than the *read*s.

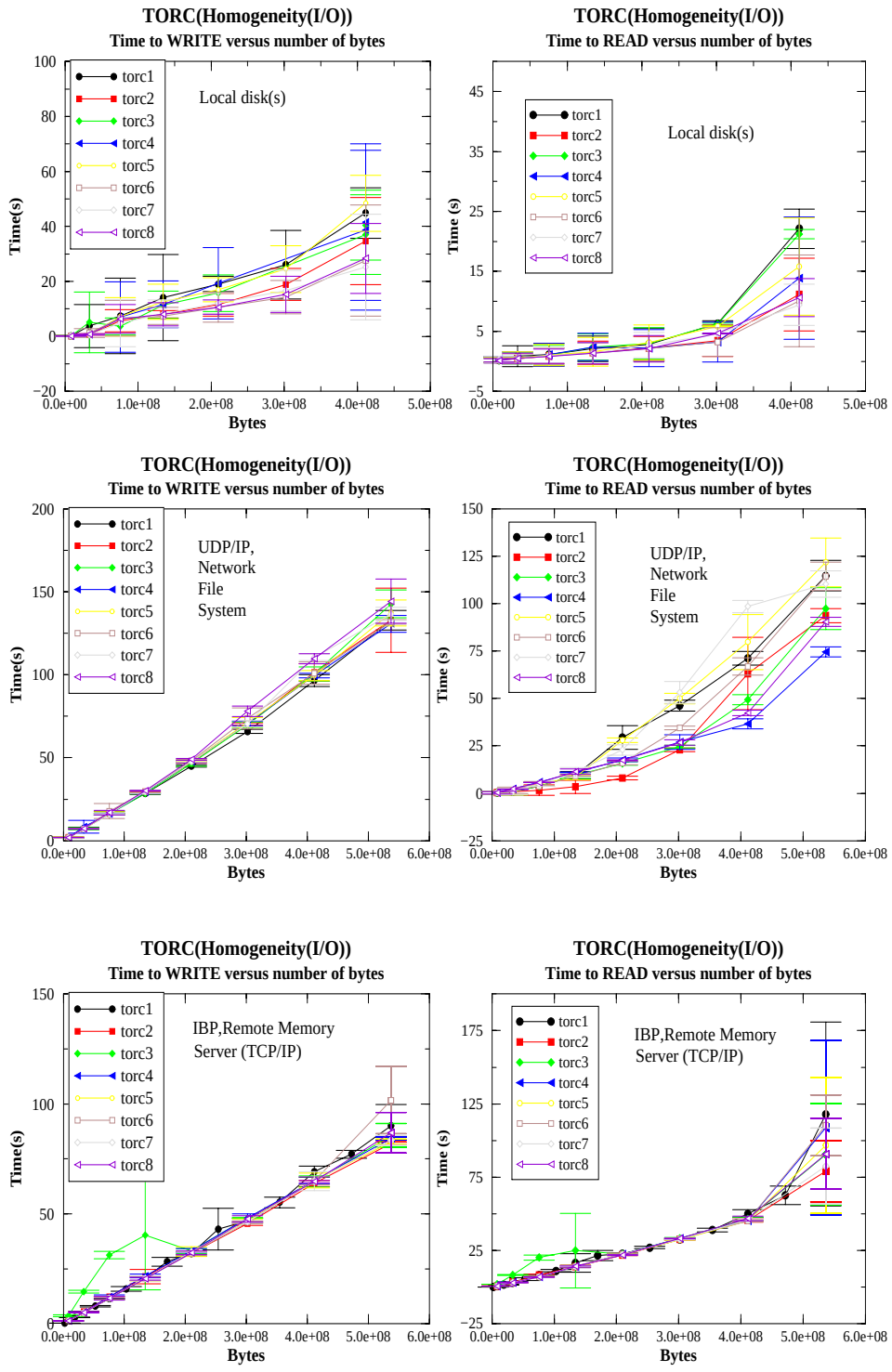

Figure 3. Empirical study of what is meant by I/O homogeneity on the shared cluster.

One can't be certain, of course, but maybe NFS is buffering part of the data to be written. Since NFS utilizes RPCs which use the UDP/IP protocol, the application does not require a response before sending the next message. Thus to the user, it appears as though all is well. This is a reliability issue and not addressed here. For the *read*s, the user doesn't report a time until he actually has his data. This may add to the fluctuations, but it is more likely due to the fact that *write*s require that the CPU (or I/O device) send both address and data and require no return of data, whereas usually the CPU must be waiting between sending the address and receiving the data on a *read*. Often, the CPU will not wait on *write*s. In plot three, the results are quite remarkable in some sense. For one thing, the accesses to the memory server, IBP on a LINUX workstation in the local area, do have to utilize the shared communication line as with the NFS case. However, this server is on a different machine in the local area from that which serves as the NFS server. Thus, one suspects that there is much less overall activity on this server as opposed to the the local network disk itself.

In Figure 3, there are two important points to keep in mind when considering the plots. One, for all three data sets, care must be taken to collect such numbers since buffering by various levels of software skews the numbers if successive runs aren't replicated scenarios. For instance, in the NFS runs if one collects such data with naive nested loop approaches, one can observe (not shown) that on the first run the time to report is *always* larger than subsequent runs. This is because of the buffering of data by either NFS, or possibly locally. If this is not accounted for, very spurious averages are formed which do **not** reflect the likely reality of servicing a user's request in numerical libraries such as those we are trying to build. The point is, unless the user makes successive requests with the same data set in mind, moving his/her data will be the unbuffered case -which is considerably larger in time. This is accounted for in the presentation here. Next, the reader is advised not to make serious comparisons of plots two and three. The machines servicing these requests employ different hardware and operating systems as well as network protocols. It is noteworthy that the NFS will likely exhibit larger variances since it is subject to a larger average activity in the local area network. One should keep in mind that the requests are all generated from some arbitrary node in our target computing system -this is why we care about these observations.

Figure 4 reports available physical memory in kilobytes per node. The plot is important because it demonstrates the reality that homogeneity can't be taken for granted -despite common resources at the onset. In particular, on the cluster under observation, there are certain nodes which are more

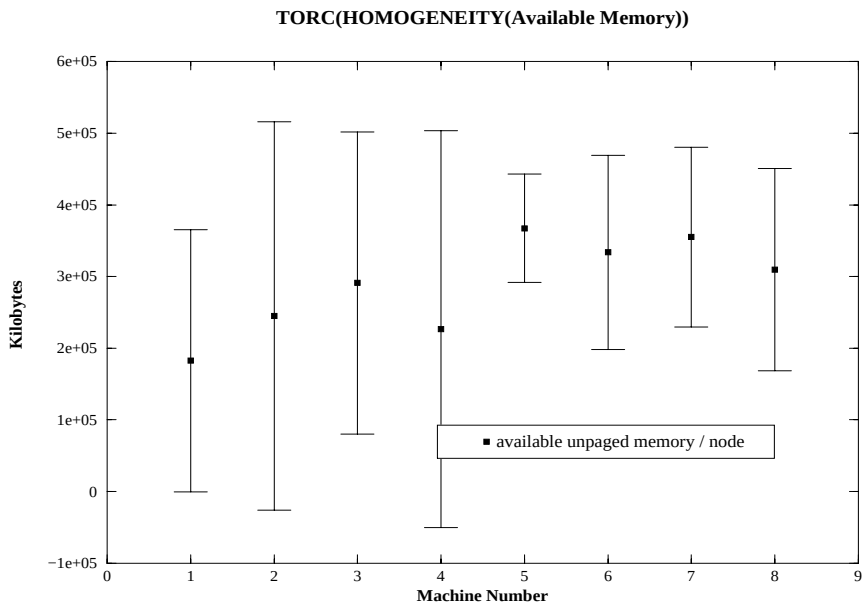

Figure 4. The available unpaged physical memory is reported in KBytes per node. Available memory is a critical criterion in resource selection since a user's job must *fit* in physical memory lest it suffer serious time penalties due to page swapping during execution. The variances for these observations are large since during heavy activity levels on a node, physical memory tends to be nearly fully in use -at other times, nearly completely idle save the OS.

commonly used than others. Who knows why this is, it just is.

Until now, we have ignored discussing some important observables such as time for *broadcast*s, *send*s, and *receive*s as a function of bytes when a collection of machines from the available resources is executing in parallel. In some sense, it is up to the library developer to *know his target system*. In other words, the parallel application may impose the system properties of interest. In our test case, we study the solution of systems of linear equations. We will discuss this more later, but the parallel kernel is known to be rich in matrix multiplies. We've seen CPU homogeneity already per node and so expect the local performance to be a good fit for such a kernel. In addition, during the factorization of A, the kernel is also rich in *broadcast*s. For this reason, the behavior of *broadcast*s from a root node (the $BLACS$ (ref. [24]) communication library implementation) as observed in our shared system is

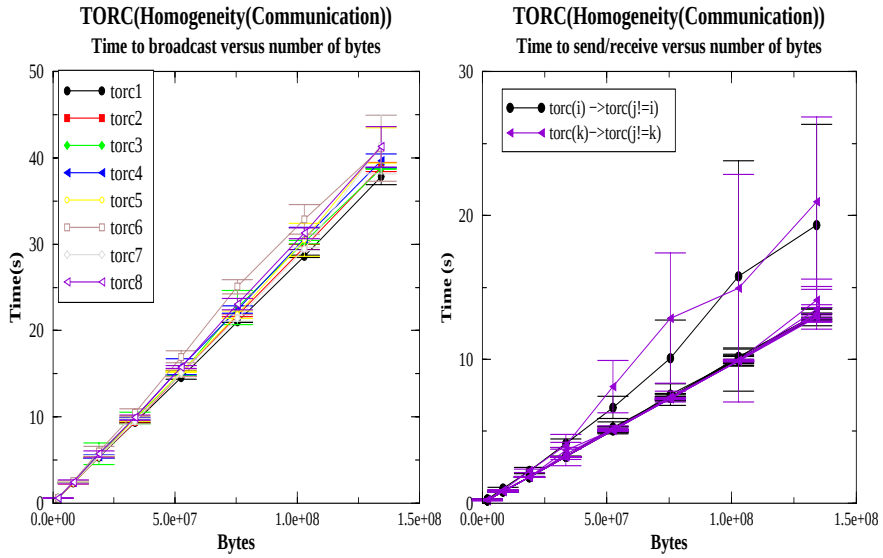

Figure 5. The plot on the left reports the time for *root* to *broadcast* N bytes of data during a parallel run in which 8 processors are selected from the cluster under investigation. Each computing node acts as *root*. Although the behavior is clearly homogeneous *looking*, there is a spurious node in the cluster as regards parallel communication of data. The fact is exposed through an investigation of the *send/receive* times (*point to point* communication) as a function of bytes. Plot two reports results from two different nodes acting as the lead node in different runs. The lead node contacts each other node in turn and *send*s N bytes of data to be *receive*d.

also reported. Each node is considered as *root* and results as a function of bytes *broadcast* timed. We wait for confirmation that each node has received the data before stopping the clock. The slowest node should always dominate -in this sense the results are meaningless for understanding the behavior of a single node's parallel communication with another particular node. The small errors are expected since the machines are fully connected through a common switch. Just in case there is bad communication between two nodes, *e.g.* to address an issue which the *broadcast* cannot, one can investigate the behavior of the *point to point* communications. Results are presented to this end. Two identical runs are considered (there are simply too many permutations of possible collections of allocated resources to report all such numbers here) in which the lead node is different. In the runs, *root* conducts *BLACS send*s

and *receive*s as a function of message size in bytes (always double precision data in our sample implementation) individually to each other process in the allocated resources.

The results in this subsection are intended help to classify our use of the term *homogeneous* as it applies to individual compute nodes from a pool of candidate resources. The important point is to understand why this is a desirable property for the development of user friendly numerical software. Consider block factorizations from linear algebra which depend upon lower level block operations that have been tuned to be *optimal* (ref. [25]) on the system for which they intend to be used. Multiplying matrices is one such operation and it is common knowledge that tuned versions of this kernel exist on most any platform. When factorizations, such as $A \rightarrow P^{-1}LU$, are implemented in parallel it remains important for matrix multiplication to be homogeneous between nodes in the parallel run to retain load balance and performance (and in fact scalability in the parallel case). Otherwise, there may be a single node which imparts large delays on the remaining nodes *or* sits idle waiting for the remaining nodes during execution of the application. Although the numerical answer should be the *same* regardless, time is the factor which the library routine attempts to minimize. (Naturally, the same could be said of the time to communicate a set number of bytes between processors during a solve. We hope that each node in a set of allocated resources is capable of sending and receiving data at the same rate. If this is not the case, the resources can be stuck waiting. Etc.) Thus, in some sense, to within the errors of each task sampled, there is some expectation as to how the system resources will behave. There is a theoretical *isotropy* in the compute nodes when homogeneity can be expected. That is, we expect that any particular computing node may be of equal use to us in the selection process subject to interpreting its current state.

2.2 The user

Our target user has a numerical application to execute and intends to do so on a single node of a shared, distributed computing environment which exhibits a strong degree of homogeneity (as described in the previous section). It is assumed that the user invokes one of our library routines to this end. We appeal to the user who may benefit from some user friendly software which intervenes on his behalf to determine if the problem can be solved more quickly through a redistribution of the user's data onto a team of computing nodes, solved in parallel, and the solution mapped back. A standard structure of our

target user's code is:

```
User_Code(){
 Define_Data_Handle(data_handle(out));
 Generate_Data(data(out),data_handle(in));
 Invoke_Numerical_Application_Routine(data_handle(in/out),
  routine_name(in),routine_parameters(in));
 Use_Solution();
 Clean_Up();
 Exit();
}
```

The user's *data* will be double precision matrix and vector elements, for instance, in most linear algebra kernels which concern us now. The storage may assume some sparse or dense (row or column major) data structure on input. At this point, we are inclined to assume that the user has correctly formatted his data for a serial execution of a specific routine from some specific numerical library. He has correctly identified the input *routine_parameters* for the *routine_name*. The *data_handle* is what the middleware uses as a key to handling the user's data. For now, suffice it to note that the user can invoke the application routine with his *data* in-core, in a file on the local network disk (*NFS* case), or in a file on a memory depot (*IBP*). The *data_handle*s are a pointer in physical memory, a file name (*e.g.* a path to the data in the file system) on the local network disk, or an *IBP capability* respectively. (A *capability* is the interface to a specific memory location in an *IBP* memory depot. It is the key to *allocate*ing, *store*ing, *load*ing, and *manage*ing a logically contiguous block of bytes on the memory server.)

One motivation for adding a remote memory depot is to allow for a user to generate problems of a magnitude which would otherwise, due to system constraints, not be feasible on a single node. This can be accomplished through buffered *store*s to a depot which has abundant available memory. We provide an interface for the user to achieve this set-up as well. Further, a user can choose to generate a data set in the memory depot and *share* the *capability* with others if so desired. This allows multiple users to collaborate on the generation of a data set and to share in the solution to a potentially common problem with no difficulties.

2.3 The middleware model

The middleware provides several services which are discussed in turn briefly. The entry point into the middleware is from the user's serial call *Invoke_Numerical_Application_Routine*() . The middleware, once invoked,

first assembles a collection of system specific parameters about the available resources which reflect the current *state* of the shared system. As previously mentioned NWS, or the like, can be used to achieve this step. Information about the number of candidate computing nodes, the free available memory per node, the CPU load (a number between 0 and 1 for each processor of each node) per node, and the bandwidth and latency for communication between each node in the cluster is gathered. Let us collect this information into the $system\_parameters$. A resource selection process is started which relies upon the evaluation of a time function which depends explicitly upon the computational demands of $routine\_name$ as a function of the $routine\_parameters$, and the $system\_parameters$ returned from the previous step. Let us call the team of resources which the middleware intends to allocate the $selected\_resources$. Next, if the $data\_handle$ specifies that the user's data has been generated in-core, then the middleware *write*s the user's data to the local network disk. (As will be discussed in the comments on data movement later in this paper, the developer has to decide whether to map the user's data at the time of this *write* or to simply *write* the data as is and allow the mapping to be made in the application routine prior to executing the parallel application. Both cases are considered in our experiments and results are presented later.) Otherwise, and after the *write* for the in-core case, the middleware assembles the command line and the machine file necessary for launching the parallel application. The application routine is *fork*ed and *wait*ed upon by the middleware. On return, if the user expects an answer in-core, the middleware assembles the solution (from disk) and passes it to the user. Otherwise, the middleware returns the modified $data\_handle$ which now contains updated information regarding where to find the expected solution. The process can be described as pseudo-code as follows:

```
Invoke_Numerical_Application_Routine(data_handle(in/out),
  routine_name(in),routine_parameters(in)){
 Get_State_of_System(system_parameters(out));
 Do_Resource_Selection(selected_resources(out),system_parameters(in),
  routine_name(in),routine_parameters(in));
 If(process==1){
  Special_Case_Serial_Run(data_handle(in/out),
  selected_resources(in),routine_name(in),routine_parameters(in));
  Return_Control_to_User();
 }
 Create_Machinefile_and_Dress_Command_Line(command_line(out),
  selected_resources(in),routine_name(in),routine_parameters(in));
 If(data_handle=="incore")
  Write_Data_to_Disk(data_handle(in));
 Fork_Application_Routine_and_Wait(command_line(in));
 If(data_handle=="incore")
  Read_Data_from_Disk(data_handle(in/out));
 Return_Control_to_User();
}
```

It is useful to look at *Do_Resource_Selection*(). A general structure for this stage of the middleware reads:

```
  Do_Resource_Selection(selected_resources(out),system_parameters(in),
routine_name(in),routine_parameters(in)){
 Find_Numerical_Library(library_name(out),routine_name(in));
 Find_Time_Function_for_Application_Routine(time_function(out),
  library_name(in),routine_name(in));
 Minimize_Time_Function(selected_resources(out),time_function(in),
  system_parameters(in),routine_parameters(in));
}
```

In its current form, the minimization of the time function disregards a careful analysis of the amount of time required to handle the user's data before (and after) executing the parallel application. (We simply assume that the total time to get the data in place is the cost of a single disk access per node plus, for instance in the case of dense linear kernels, the summation (over compute nodes) of the time for each node to move $O(N^2 * sizeof(double)/total\_processors)$ bytes over the shared network lines with a knowledge of the bandwidth and latency of the network at the time of the mapping.) This is agreeably naive and we are hard at work developing viable models of the handling/mapping of the user's data over the network. It is complicated by many factors however. The fact that TCP/IP, for instance,

imparts its own congestion control (*e.g.* sliding windows) when the network becomes congested complicates matters.(See references [26, 27].) Further, we don't know whether the system handling the disk accesses employs DMA or, if not, how many accesses to disk may be necessary to service a request, the time per disk access, etc. If we did know this, we could write a function to approximate the procedure. For instance, suppose we wanted to *load* N bytes from a network disk into local physical memory. Suppose we know the bandwidth and latency between the source and destination machines, the number of bytes *load*able per disk access as well as the time overhead per disk access. In this case, the total number of disk accesses is (N bytes) / (X bytes loaded per disk access), or N/X disk accesses. The time just to get the data to the network will then be (N/X disk accesses)$\times$ (Y seconds per disk access), or NY/X seconds, for instance. Next, to move the data over the network, we could assume the ideal and thus, the time to move the data over the network would be simply the latency (converted to seconds) plus N bytes divided by the network bandwidth (converted into bytes per second). Unfortunately, these simplifications ignore the true complexities of the underlying resources and protocols which are used in practice. Such models fail to yield realistic predictions in a shared system. Again, it's complicated and a work in progress. Naively, and for the sake of commentary, one assumes that the typical form for a time function will be of the sort, $T_{solve\_user's\_problem} \simeq T_{handle\_user's\_data} + T_{execute\_parallel\_application}$. Here, $T_{execute\_parallel\_application}$ and $T_{handle\_user's\_data}$ are functions of the *routine_parameters* and *system_parameters*. In these terms, our original conjecture for a study may be restated as: *if* $T_{serial\_expert} - T_{execute\_parallel\_application} > T_{handle\_user's\_data}$*, then the user benefits from invoking the middleware. Otherwise, the user's problem is simply solved serially without any benefits from having invoked the middleware.* Again, if the parallel application routine is scalable, we expect to find the problem size which marks the turning point which validates the method. We return to the issue when considering the test case.

2.4 The application layer

The application level of our current effort is built upon pre-existing numerical packages such as *PETSc* (ref. [28]) or *ScaLAPACK*. We have relied upon the scalability of such libraries to more than account for the time required for the middleware to handle the user's data. It is the developer's burden to understand the application routines from such libraries at an intricate level so that accurate time functions can be written to be

used by the middleware during the resource selection process. At this point, the typical time function for $T_{execute\_parallel\_application}$ has the form $T_{execute\_parallel\_application} \simeq T_{communicate\_data} + T_{floating\_point\_operations}$. In the example discussed in the next section, details of such a function are placed in context of an actual parallel kernel.

The application routine has to get the user's data in-core before the parallel execution. There have been several approaches tested to this end. Basically, the data may be pre-mapped by the middleware in which case the parallel application starts with a parallel *read* of the data from the local network filesystem. The user's data may reside on the local network disk or a memory server in an unmapped format. In this case, either each node assembles its own data through a series of random accesses to the file housing the data, or a single, *lead*, node is responsible for bringing the data incore and distributing it in a mapped manner -or in bulk letting each node claim its own data- through a series of *send/receive*s or *broadcast*s respectively. Let us agree for simplicity to lump all these scenarios into the routine *Get_User's_Data_Iown_Incore*(). Each processor in the allocated team of resources will use this. Then,

```
Parallel_Application_Routine(command_line(in)){
Parse_Command_Line(environment_information(out),data_handle(out),
 routine_parameters(out),command_line(in));
Initialize_Parallel_Environment(environment_information(in));
Get_User's_Data_Iown_Incore(data(out),data_handle(in));
Execute_Parallel_Application_Routine(data(in/out),routine_parameters(in));
Collect_Answer_to_Root(data(in/out));
If(Iam_Root)Write_Answer_to_Disk(data(in),data_handle(in));
Free_Parallel_Environment(environment_information(in));
Exit();
}
```

3 Sample software implementation and results

3.1 Some comments on the kernel, pdgesv()

In this section, results from a study of the kernel *pdgesv*() (from the numerical library *ScaLAPACK*) are presented and briefly discussed. In each case, it is assumed that the matrix elements form a well determined system.(ref. [29]) Routine *pdgesv*() computes solutions to the system of equations $Ax = b$ where $A \in \Re^{m,n}$, $x \in \Re^n$, and $b \in \Re^m$. In particular, a dense, block based factorization of the matrix A is made reducing it to the form $P^{-1}LU$ where P^{-1} is a pivot array, L is a lower triangular matrix (unit diagonal), and U is upper triangular. (Thus we expect the blocks $L_{1,2}$, and $U_{2,1}$ to be zero.)

After said reduction, two relatively trivial systems of equations are left to solve instead of the original set. Thus, one has $PA = LU$ or $A = P^{-1}LU$, solves $Ly = b$ for y, and finally solves $Ux = y$ for x. There are a couple of important points to make here. The algorithm for the factorization is applied recursively. Thus, the routine factors (Gaussian elimination) the $A_{1,1}$, $A_{2,1}$ blocks of A recursively employing partial pivoting over rows in a single column of the process grid (the other nodes remain idle). At the stop case, each process in the current, active, process column *broadcast*s the pivot information to all the remaining columns of processes. Each process can then apply the row interchanges (to all the columns that were not involved in the iteration) to reflect the changes from the previous step. After each such factorization, the $L_{1,1}$, $L_{2,1}$, and $U_{1,1}$ blocks are known. The evaluation of each block row of the matrix U requires the solution of a lower triangular system of equations over the elements in a single row of the process grid. Thus, $L_{1,1}$ is *broadcast* along the current row of processes converting panel $A_{1,2}$ to $U_{1,2}$ -*e.g.* $U_{1,2} \leftarrow L_{1,1}^{-1}A_{1,2}$. The last step in any iteration of the factorization is the Schur update of the trailing sub-matrix. The column block $L_{2,1}$ is *broadcast* over rows across all columns of the process grid. $U_{1,2}$ is *broadcast* over columns along all the rows of the process grid. Then the Schur update modifies each process' local portion of the block $A_{2,2}$. ($\tilde{A}_{2,2} \leftarrow A_{2,2} - L_{2,1}U_{1,2}$) This operation is a parallel matrix multiplication, *pdgemm*() (ref. [30]), and it dominates the execution time. After the update, the process is begun again recursively on block $\tilde{A}_{2,2}$. It is important to note that the factorization proceeds from left-to-right, top-to-bottom. Thus, the amount of work per node becomes uneven as the factorization progresses. It is noteworthy that the expert user recognizes this point as a need for optimizing both the block size and the grid aspect ratio (the logical dimensions of the rectangular process grid). The block size is generally chosen to coincide with the problem size which achieves 90% of the performance expected by *dgemm*() (*ATLAS*) on each node. Again, we have a need for homogeneity between nodes. The grid aspect ratio is no big deal really but it does change the time of execution for solving the problem. An example is provided in Figure 6. It is observed quite generally that for a $p \times q$ logical rectangular process grid (p,q denotes the number of logical process rows,columns respectively) $p/q < 1$ yields better performance for the specific kernel at hand than the cases where $p/q \geq 1$.

When allocating resources from a pool of computing resources for the kernel *pdgesv*(), the times of particular interest are $t_{factorization}$, $t_{broadcast}$, and t_{update}. On a user's request to solve a system of linear equations, the middleware ascertains the *state* of the shared system and invokes the selection process which attempts to minimize a time function. This function purports

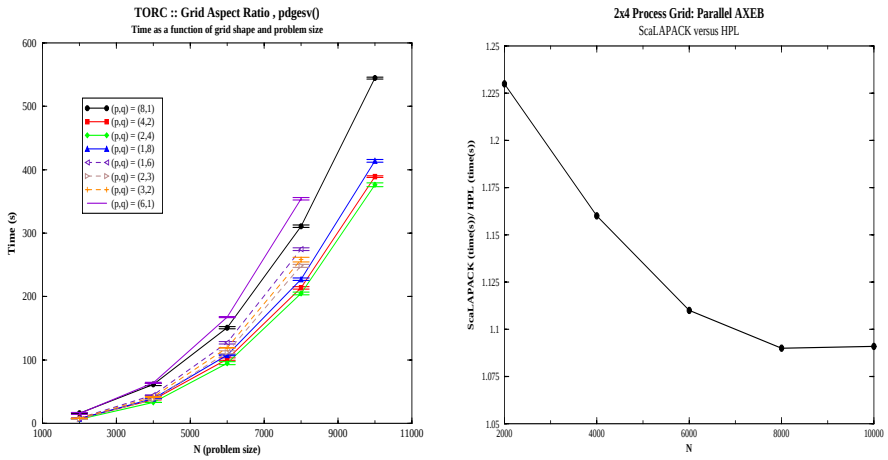

Figure 6. The developer is responsible for understanding how to get the most out of the kernel being implemented in the proposed library. The plot on the left demonstrates a time range of over $100 seconds$ for the 8 processor run due to either a very poor choice of logical grid layouts, or the best possible. The plot on the right compares (a ratio of execution times is formed) the best run of the $ScaLAPACK$ routine $pdgesv()$ versus the High Performance Linpack benchmark over the same number of nodes on the system. HPL is highly tuned and is really hard to beat in practice, even for experts. The results demonstrate the reliability of $ScaLAPACK$ to perform well provided the correct system parameters are identified. Tuning the major kernels in $ScaLAPACK$ has been a major concern on the side of the application routine developers.

to model the kernel as a function of the problem parameters as well as the state of the system. The basic communication model assumes a linear relation between any two nodes i, j such that the time to send a message of size X bytes is $t_{comm(i,j)}(bandwidth_{i,j}, latency_{i,j}) \simeq latency_{i,j} + bandwidth_{i,j}^{-1} \cdot X$. For a $p \times q$ process grid, the factorization of a $m \times n$ panel, again, (see the documentation regarding HPL (ref. [31])) occurs within a single process column (p processors). Since the algorithm recurses within a panel, the assumption is made that any of the p processors within the panel perform at level 3 BLAS (ref. [32]) rates. Here is where we employ the notion of homogeneity in our target system as regards both communication and computational expectations. In the kernel of interest we are always bound by the slowest machine in the *broadcast* and *update* phases. Thus we select p processors which reflect homogeneity in the sense defined in this paper. If one only considers the time the nearest neighboring process columns will spend in the *broadcast* phase (after the fac-

torization), then the time to factor and *broadcast* a block panel is estimated for a collection of homogeneous resources by $t_{factor}(m,n)+t_{broadcast}(m,n) \simeq f_{dgemm}(\frac{n^2m}{p} - \frac{n^3}{3}) + latency(1 + nlg(p)) + (bandwidth)^{-1}(2n^2lg(p) + \frac{mn}{p})$. The Schur update of the remaining $n \times n$ matrix is approximated for homogeneous systems by $t_{update} \simeq 3q^{-1}(bandwidth)^{-1}n \cdot nb + latency(p + lg(p) - 1) + f_{dgemm} \cdot n \cdot nb(\frac{nb}{q} + 2\frac{n}{pq})$. Here nb is the block size, m, n are the panel dimensions (not global) for the factorization and f_{dgemm} approximates the performance of matrix multiply on an arbitrary node of our homogeneous system. Naturally, the system constraints have to be considered despite the assumption of homogeneity. The total time to perform this factorization in a homogeneous system may be approximated as $\sim (start, i = 0) \sum_{i+=nb}^{n}(t_{factor}(n-i,nb) + t_{broadcast}(n-i,nb) + t_{update}(n-i-nb,nb))$. Quite frankly, one really requires a separate paper simply to work through the details of how we arrive at this time function and how it is used in the selection process. Clearly, we wish to minimize this function. Scheduling schemes are truly difficult however. People try simulated annealing, genetic algorithms, apply low order time series analyses, etc. The problem is of interest to the community in general. We will address the issue at this time. Future papers will describe our efforts in this area in detail. For now, suffice it to say, we crudley minimize this function based on the available system resources through an adhoc means and observe how well the allocated resources complete the task relative to the time we predict it should take based on analyzing the time function. The plots in Figure 7 reveal the faithfulness of our model in its current form.

3.2 Data movement scenarios

Moving and mapping $m \cdot n$ double precision matrix elements is informally discussed. The major consideration is on getting the data, matrices $A \in \Re^{m,n} \mapsto (m*n*sizeof(double))bytes$ and vectors $b \in \Re^{m} \mapsto (m*sizeof(double))bytes$, from the user and in place for the parallel solve routine.[d] It is assumed that the data is generated in a *natural* ordering, which is to say, in the language C for instance, that $*(A+i+j*n)$ references the value of the matrix element in row i and column j of matrix A when $0 \leq i < m$ and $0 \leq j < n$. This mapping is of general importance because the scalability of numerical calculations in a distributed computing environment is dependent upon spending more time computing than moving a user's data around. In particular, if numerical libraries such as that being investigated here are to be successful, it is necessary

[d] In the test cases $m = n$ as we consider square matrices in the parallel solve routine.

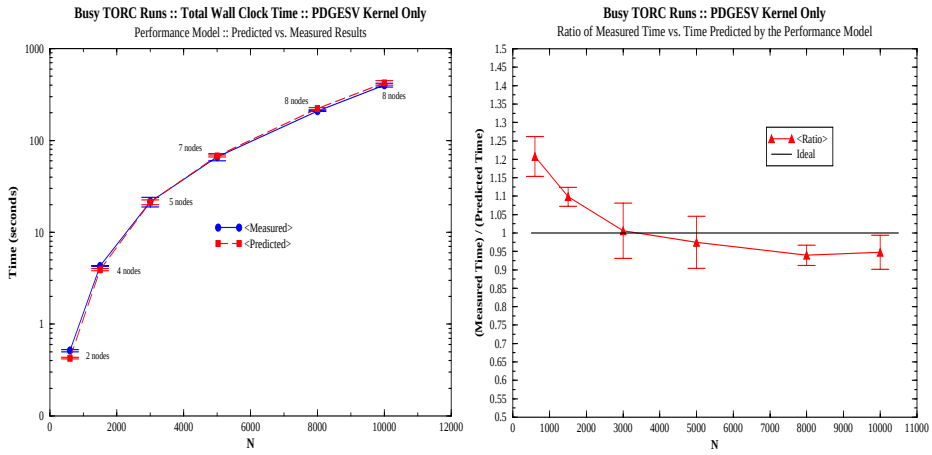

Figure 7. In plot one, the predicted wall times for execution given several different instances of the state of the system is plotted against the resulting execution times actually measured. The second plot is a dimensionless ratio of the two data sets of plot one. The ideal case is the value 1.

that the user's data can be forward mapped into a form expected at the application level (the parallel routines) and the solution backward mapped into natural form for the user in a timely fashion. For dense linear algebra kernels being studied in parallel computing environments it is known that a $2d$ block-cyclic mapping of the naturally structured data (*e.g.* matrix A and vector b in $Ax = b$) provides excellent load balance during parallel runs. (See references [33, 34, 35] for instance.) The mapping is a function of the problem size (mn, or n for square matrices where $[A \in \Re^{n,n} \mapsto (n^2 * sizeof(double))bytes] \rightarrow$ $problem\ size \equiv n$), the block sizes (nb_row, nb_column, or nb), the number of process columns in the logical rectangular process grid ($npcols$), the number of process rows in the logical rectangular process grid ($nprows$), and the matrix elements, A. In fact, the mapping is quite easy to implement naively but it is found that real improvements on such a mapping are more difficult to achieve than expected. In the remainder of this section we outline the general scenarios that we have considered. In each instance, the corresponding *read* of the data, from the perspective of the parallel application routine, is also discussed.

In scenario one, the user first generates the data in-core. There are $n^2 + n$

elements and each requires $sizeof(double)$ bytes of memory. This user then passes a pointer to the middleware which in turn (after deciding on the process grid) *write*s ($nprows \times npcols$) $2d$ block cyclically pre-mapped work files to the network disk (NFS) as well as $nprows$ work files for the vector elements. In total, of course, only $(n^2 + n) * sizeof(double)$ bytes are written to disk. However, it is useful to compare the time it takes to simply *write* natural A to a single file on the network disk versus the time to *write* the elements of A to multiple files in an order imposed by the $2d$ block cyclic mapping. (see Figure) In this scenario, the processors involved in the parallel application routine have only to *read* their matrix elements from disk before executing the parallel application. It is noted that no extra elements are read per processor as the mapping has already taken place in the middleware and is exact.

The second general scenario begins with the user's data having been written to a file on either the local network disk or on the memory depot. After the middleware determines a set of processes for the application routine, the *data_handle* is passed to the application routine from the middleware. The natural data is brought in-core by a chosen processor (root) from the logical process grid. The data may be distributed by root in a manner which imposes the mapping during *point to point* communication with the other nodes(2a). Alternatively, root may proceed to *broadcast* the data as is and let the mapping be done by each process locally(2b). (We label these scenarios here so the reader can make sense of the plots.)

In the final scenario we consider the case where the natural data is brought in-core in parallel by each processor in the logical process grid. The mapping may be imposed exactly during the load phase through random access to the file(3a). Alternatively, bulk data can be brought over the network by each process and the mapping carried out in-core(3b). It is noted that one may contend with network congestion in this scenario. In reality, the developer has to experiment with these approaches before truly understanding the best approach for a given system.

4 Results and conclusions

Figure 8 is interesting to think about. For one thing, we clearly see N_{tp} exists for each of the scenarios reported on the graph. Notice that the scale of the graph is linear-log. Thus, small separations on the graph suggest large separations in time. Not all the scenarios are reported but the point is clearly made -the user will benefit from interfacing with the proposed software. Plot two of Figure 8 demonstrates at once the strength $ScaLAPACK$ for the dense algebraic kernel, and the reason we pursued this idea in the first place

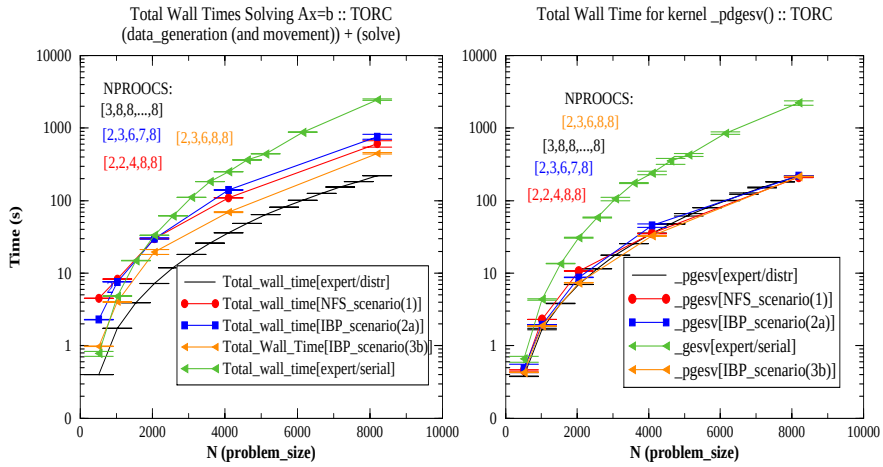

Figure 8. The total time is reported for multiple approaches to solving the linear problem $Ax = b$. The serial and parallel expert cases proceed without the intervention of additional software. All other runs reported invoke the middleware just as desribed in the current work.

-scalability. For each run on the graph the number of processors allocated for the parallel runs is provided. The expert user in the serial environment can no longer conduct his normal activities in a timely manner. His task is over 1000seconds behind the proposed software when we stop counting.

Figure 9 is also interesting because it displays the overheads involved in simply porting the user's data into a form amenable to the parallel application routine. The times reported for the top two plots are simply the time it takes for the expert user (serial, or parallel) to generate his data before calling the library routine. Next, if we look at the local network disk data (NFS), handling there are multiple plots. The original time it took for the middleware to generate the block cyclic data files for each process in the logical process grid was due to a careless mapping. This is an easy mistake for user's -even developers- to make. Simply compare the time it takes to *write* some set number of bytes to the network disk with these maps. The only difference in the poorly implemented case (which also affects the numbers in Figure 9) and the faster mappings is in buffering. Similarly, in the load of the workfiles in scenario (2a) attempts to first map the user's natural data and then communicate it. It is simply faster to let multiple requests engage the memory server. Of course, this is only true until such traffic generates

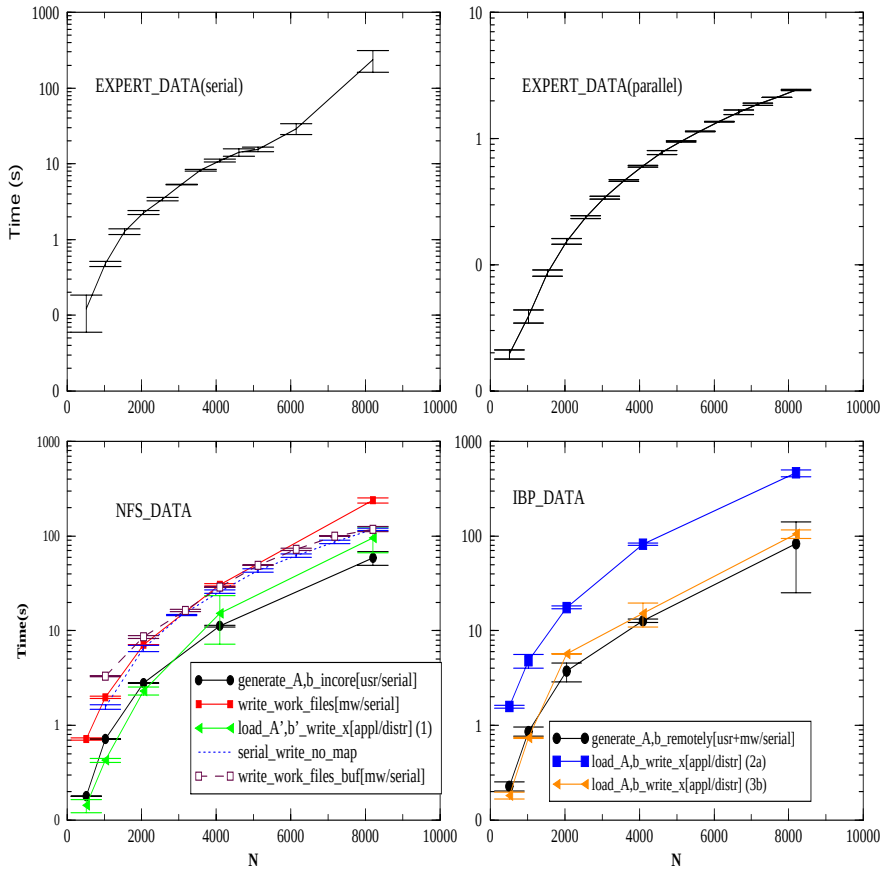

Figure 9.

congestion on the network. We don't see that here but easily could as larger problems and methods are investigated.

There is much left that could be discussed about such next generation software which we have not had time to even mention. For instance, the issue of how to build fault tolerance into such a set of library routines, or assembling contracts which attempt to monitor the progress of a user's task and act if necessary. The general problem of *re*distributing the user's data is also important.

In closing, we have attempted to define the target system through identifying a set of criterion relevant to the numerical library routines we are investigating. Homogeneity is an important criterion and as has been discussed, cannot be taken for granted. We have defined our target user and provided an interface for this user into the middleware which has been designed to utilize existing, scalable library routines. We have reserved a detailed conversation for scheduling in shared, homogeneous system for another time. This subject is of great importance however and cannot be avoided in practice. It is possible, if not likely, that techniques from time sensitive statistical investigations will play a major role in coming to terms with large scale systems -especially as they only become larger. Motivation to further pursue such approaches to numerical libraries on shared, homogeneous systems was provided. We are proposing a more thorough study at this point which attempts to ferret out many of the outstanding mysteries.

References

1. See documentation on IBM's *LoadLeveler* batch system. For instance *http :: //usgibm.nersc.gov/docs/LoadL/index.html* provides documentation of a specific implementation on *NERSC*'s IBM SP/RS6000, currently known as *seaborg.nersc.gov*.
2. Garey, M. and Johnson, D.,*Computers and Intractability, a Guide to the Theory of NP-Completeness*,Bell Telephone Laboratories,1979
3. Hopcroft, J. E. and Ullman, J. D.,*Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979
4. Ullman, J. D.,*"NP-complete scheduling problems,"*,Journal of Computer and Systems Sciences **10**:3, pp. 384-393
5. Knuth, D. E.,*Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, Addison-Wesley, 1968
6. Knuth, D. E.,*Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*, Addison-Wesley, 1969
7. Knuth, D. E.,*Sorting and Searching*, volume 3 of *The Art of Computer Programming*, Addison-Wesley, 1973
8. Cormen, T. H., Leiserson, C. E, Rivest, R. L., Stein, C.,*Introduction to Algorithms, 2nd edition*, MIT Press, 2001
9. *NWS, The Network Weather Service*, http://nws.cs.ucsb.edu/
10. Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R. C.,*ScaLAPACK Users' Guide*, SIAM, 1997
11. Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D.,

Johnsson, L., Kennedy, K., Kesselman, C., Mellor-Crummey, J., Reed, D., Torczon, L., Wolski, R.,*"The GrADS Project: Software Support for High-Level Grid Application Development,"* 2001, Rice University, Houston, Texas; see also *http://www.hipersoft.rice.edu/grads*

12. Foster, I. and Kesselman, C.,*"GLOBUS: A metacomputing infrastructure toolkit"*,International Journal of High Performance Computing Applications, **vol. 11**, pp. 115-128
13. Petitet, A., Blackford, S., Dongarra, J., Ellis, B., Fagg, G., Roche, K., Vadhiyar, S.,*"Numerical Libraries and the Grid,"*International Journal of High Performance Computing Applications, **vol. 15**, pp. 359-374
14. Foster, I. and Kesselman, C.,*The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Francisco, 1999
15. Petitet, A., Blackford, S., Dongarra, J., Ellis, B., Fagg, G., Roche, K., Vadhiyar, S.,*Numerical Libraries and the Grid: The GrADS Experiments with ScaLAPACK*, Computer Science Department Technical Report UT-CS-01-460, University of Tennessee, Knoxville, Tennessee 37996-3450
16. Boulet, P., Dongarra, J., Rastello, F., Robert, Y., Vivien, F.,*"Algorithmic issues on heterogenous computing platforms,"* Parallel Processing Letters, **9**:*2*, pp. 197-213, 1999
17. Kalinov, A. and Lastovetsky, A.,*"Heterogeneous Distribution of Computations While Solving Linear Algebra Problems on Networks of Heterogenous Computers,"* Journal of Parallel and Distributed Computing, **61**, *4*, pp. 520-535, 2001
18. Oksendal, B.,*Stochastic Differential Equations, 3rd edition*, Springer-Verlag (Berlin), 1992
19. Bremaud, P.,*Point Processes and Queues, Martingale Dynamics*, Springer-Verlag (New York), p.181 , 1981
20. Lipster, R. and Shiryayev, A.,*Statistics of Random Processes: General Theory*, Springer-Verlag (New York), 1977
21. Lipster, R. and Shiryayev, A.,*Statistics of Random Processes: Applications*, Springer-Verlag (New York), 1978
22. *PAPI*, Performance API, *http://icl.cs.utk.edu/projects/papi*
23. *IBP*, Internet Backplane Protocol, *http://loci.cs.utk.edu/ibp*
24. Whaley, R. C.,*Basic linear algebra communication subprograms: Analysis and implementation across multiple parallel architectures*, Computer Science Department Technical Report UT-CS-94-234, University of Tennessee, Knoxville, Tennessee 37996-3450
25. Whaley, R. C., Petitet, A., Dongarra, J. J.,*"Automated empirical optimizations of software and the ATLAS project,"* Parallel Computing, **27**, *1,2*, pp.3-35, 2001

26. Peterson, L. L. and Davie, B. S., *Computer Networks: A Systems Approach*, Morgan Kaufmann (San Francisco), 1996
27. Stevens, W. R.,*Unix Network Programming*, Prentice-Hall, 1990
28. Balay, S., Gropp, W., Curfman McInnes, L., Smith, B.,*PETSc Users Manual, rev. 2.1.0*, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439
29. Golub, G. H. and Van Loan, C. F.,*M*atrix Computations, 3rd edition, John's Hopkins University Press, 1996
30. Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D., Whaley, R. C.,*A Proposal for a Set of Parallel Basic Linear Algebra Subprograms*, Computer Science Department Technical Report UT-CS-95-292, University of Tennessee, Knoxville, Tennessee 37996-3450
31. *HPL*, High Performance Linpack benchmark,*http://www.netlib.org/benchmark/hpl*
32. Dongarra, J., Du Croz, J., Duff, I. S., Hammarling, S.,*"A Set of Level 3 Basic Linear Algebra Subprograms,"* ACM Trans. Math. Soft., **14**, pp. 1-17, 1988
33. Lichtenstein and Johnson,*"Block-cyclic dense linear algebra," SIAM* J. Sci. Stat. Compt., **14**, pp.1259-1288, 1993
34. Petitet, A. *A*lgorithmic Redistribution Methods for Block Cyclic Decompositions, PhD thesis, Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996-3450
35. Beaumont, O., Legrand, A., Rastello, F., Robert, Y.,*"Dense linear algebra kernels on heterogeneous platforms: Redistribution issues,"*, Parallel Computing, **28**, (issue 2), pp.155-185, 2002
36. Kernighan, B. W. and Ritchie, D. M.,*The C Programming Language, 2nd edition*,Bell Telephone Laboratories, Inc., 1988
37. Snir, M. , Otto, S. W., Huss-Lederman S., Walker, D. W., and Dongarra, J. J.,*MPI: The Complete Reference*,MIT Press,1996