

Efficient Coupling of Parallel Applications Using PAWS

Peter H. Beckman Patricia K. Fasel William F. Humphrey
Susan M. Mniszewski
Los Alamos National Laboratory
Los Alamos, NM 87501
{beckman , pkf , bfh , smm}@lanl . gov

Abstract

PAWS (Parallel Application WorkSpace) is a software infrastructure for use in connecting separate parallel applications within a component-like model. A central PAWS Controller coordinates the linking of serial or parallel applications across a network to allow them to share parallel data structures such as multidimensional arrays. Applications use the PAWS API to indicate which data structures are to be shared and at what points the data is ready to be sent or received. PAWS implements a general parallel data descriptor, and automatically carries out parallel layout remapping when necessary. Connections can be dynamically established and dropped, and can use multiple data transfer pathways between applications. PAWS uses the NEXUS communication library and is independent of the application's parallel communication mechanism.

1. Introduction

Development of large, high performance scientific simulation applications on serial or parallel supercomputers has generally followed a “monolithic” software engineering approach: develop a single executable or set of executables which include all functionality of the simulation, for which additions or changes to any portion of the code signal the need for a complete rebuild of the entire application. This model has evolved due to the very specialized nature of supercomputer architectures, and the need to develop applications for use in very specific problem domains. A growing need for applications of this nature is the ability to develop these complex simulation codes from more basic, well-understood simulation *components*, and to link these components and full applications together across disparate networks, machine architectures, and programming languages.

At Los Alamos National Laboratory and other institutions, very large computational problems are be-

ing addressed by applications that span a wide range of algorithms, programming languages, and resource requirements. An efficient, flexible mechanism to allow independently-developed simulation capabilities to exchange data on both serial and parallel architectures is highly desirable. A general component model which can work with large-scale scientific simulation codes is seen as a powerful development method for these types of applications.

Component development and usage has become a powerful and flexible tool in industry today, and component models for distributed application development such as CORBA [1], DCOM [14], or Java RMI [2] have become popular architectures for building serial applications. These models, however, unfortunately fall short of providing the capabilities required to allow them to work with parallel data structures.

PAWS (Parallel Application WorkSpace) is a project intended to help application scientists develop modular, component-like high performance programs, by providing a common mechanism for linking together parallel applications and for sharing parallel data structures between these codes. There are a number of challenges involved in this issue:

- Allow for *dynamic* coupling of applications, at any time during their execution;
- Avoid *serialization* bottlenecks when transferring data between parallel applications;
- Allow for parallel data structures to be shared between applications using different parallel layout strategies;
- Allow for data exchange between applications developed in different programming languages.

The strategy which PAWS takes to meet these challenges is to implement a standard mechanism for specifying the parallel nature of user-specified data structures through a PAWS API, and to provide a PAWS Controller application

to coordinate the connection of applications and their parallel data objects. In this model, applications become components which can be linked together with the help of the PAWS controller. The PAWS API and controller manage the details of exchanging data between parallel application with possibly different data decomposition methods, taking advantage of multiple network connection pathways when possible.

In addition to the standard component models, a number of other systems have been developed for sharing data structures between serial or parallel applications, such as Cumulvs [6], DAQV [8], and PARDIS [10], to name a few. PAWS differs from projects such as PARDIS in its *data-centric* view, in which the primary inter-application mechanism is via exchange of the current state of a parallel data structure, and an application API which does not focus on the use of an IDL. PAWS is not focused on the issues of computational steering, which are an important part of the Cumulvs and DAQV projects; instead, the “WorkSpace” portion of PAWS refers to the ability of PAWS, through applications employing the PAWS API and connecting to the PAWS controller, to create and manage a data-flow-like network (WorkSpace) of interacting parallel components.

2. Standard PAWS Environment

The PAWS API and controller provide, primarily, a means for two or more programs to share data stored in parallel data structures, in networks such as the prototypical connections shown in Figure 1. A component model allows one to develop basic, fundamental blocks which perform a single task, such as the separate simulation, visualization, or analysis portions of the networks in Figure 1.

Figure 2 illustrates the standard PAWS working environment, demonstrating two separate parallel applications linked together and sharing parallel data structures. The environment consists of two basic concepts:

- The PAWS *Controller*, the process that coordinates the work of linking applications and data structures together, manages resource allocation, and handles user authentication. This is similar in nature to an Object Request Broker (ORB).
- The PAWS *Applications*, which use the PAWS API to communicate with the controller, to make their data structures available to other applications, and to transfer data when necessary.

When a program using the PAWS API first starts, it contacts the PAWS controller and registers itself as an active application. During the course of its execution, the application registers those data structures which it would like to

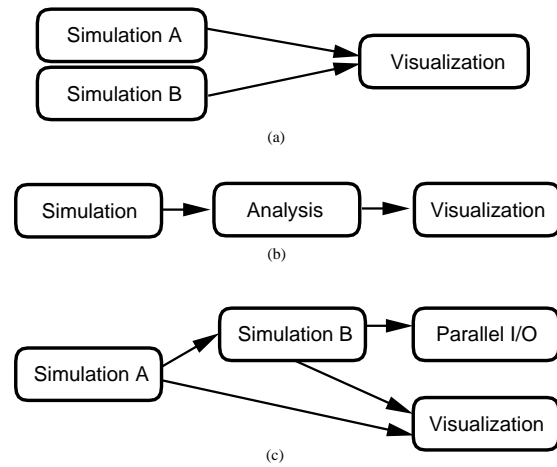


Figure 1. Prototypical networks of simulation components. (a) Visualization of two independent simulations. (b) Filter network. (c) Two linked simulations, connected to a parallel I/O module and a visualization component.

make available for other applications to share. The controller, via an interpreted script provided to the controller by a user or by a visual setup environment, connects data structures together by establishing data transfer pathways directly between the relevant applications. Data transfer occurs across multiple connections, taking advantage of the maximum available network bandwidth. Thus, the applications using the PAWS API represent the simulation software components in this environment, and the PAWS controller manages the infrastructure for connecting these components together.

Due to the scripted nature of the controller, parallel component networks can easily be established, and dynamically altered to replace or reconfigure the components into new relationships. A key feature of this system is the use of parallel data transfer network pathways directly between the applications sharing a particular parallel data structure, which addresses the challenge of connecting parallel components together in an efficient manner.

3. PAWS Controller

The PAWS controller is implemented as a standalone application which acts as a simple repository of information on the state of the PAWS working environment (WorkSpace). This information includes:

- The location and configuration of available compute *resources*.

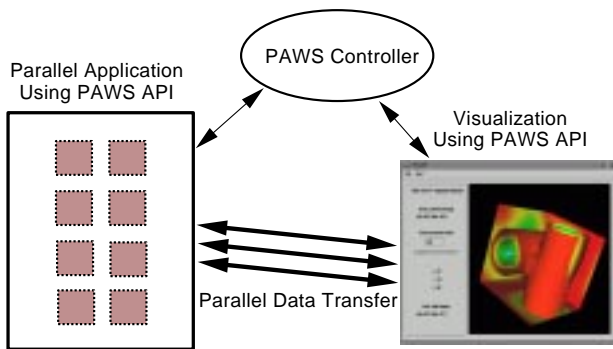


Figure 2. The standard PAWS Workspace environment, demonstrating two parallel applications sharing data and in contact with the PAWS controller.

- The list of active *applications* which use the PAWS API.
- The list of registered *data objects* in the active applications, and their characteristics.
- The list of established *connections* between data objects.

The controller can be queried for information by PAWS applications or other processes, and implements a defined mechanism to process requests for information through a simple Tcl [11] scripting interface. This scripting interface is also used to initialize the controller with information on the configuration of the available machines, to request the controller to launch new applications, and to create or destroy connections between data objects in registered applications. Access to controller information about available applications, objects, and resources may be available in the future through a Lightweight Directory Access Protocol (LDAP) [9] interface.

A controller must be available and running when a PAWS application starts, in order for that application to be able to register itself and its data objects with the controller. A controller must also be available when a connection between two data objects is established. No interaction with the controller is required, however, during actual data transfer between two applications. The information about the state of the controller and Workspace are kept in persistent storage, in order to improve fault tolerance – if the controller dies while active PAWS applications are still present, a new controller may be started and use the persistent state information to restore the original Workspace.

The controller is also responsible for providing resource allocation and user authentication facilities to the user and PAWS applications. Applications launched by the controller must be executed on the proper architecture and with

sufficient resources, thus access permission for these resources must be obtained by the PAWS controller. We plan to utilize the Globus [3] metacomputing environment for these purposes.

4. PAWS API

The PAWS user API is provided as the mechanism by which a new or existing parallel application may participate in the PAWS Workspace. The API is implemented as a C++ class library at present; C and Fortran interfaces are in development.

Applications first use the API to register themselves with a running PAWS controller, which is accomplished by creating an instance of the `PawsApplication` class. `PawsApplication` must be given a URL indicating where to contact the PAWS controller; the URL may be provided either directly by the user (if the application is started by hand), or by the PAWS controller (if the application is started by the controller). PAWS uses the Nexus communication library [4] for PAWS-related intra- and inter-application communication, and for communication between the application and the controller. `PawsApplication`, when instantiated, establishes contact with the controller and all processes of the parallel application, and provides the user interface for controller queries. The Nexus-based PAWS communication mechanism works independently of the particular intra-process communication method employed by the parallel application, such as MPI [7], PVM [5], or shared memory.

Each component application contains a number of parallel data objects, and the PAWS API is used to register with the controller the appropriate data objects which should be available for sharing with another application. For each such data object, a `PawsData` instance is created, which stores the connection state of the data object. Registering a data object does not result in a connection being established, it only lets the controller and other applications know the name, location, and characteristics of the data structure. Connections are established through requests to the controller, either through the controller script interface, or through a call to the PAWS user API within an application. Connections may be established or disconnected at any time. A data structure may be connected to any number of other structures: applications themselves are not aware of what other objects are connected to their own objects, as PAWS maintains the information about the data connection network within the controller and the `PawsData` objects. The connected objects need only be of the same total size; they can, in many circumstances, have different parallel layouts or be implemented in different languages.

When two data structures in separate applications have been connected together (either through a controller com-

mand, or a call to the PAWS API), they are maintaining a shared image of each other. In any connection, one application is designated as the sender, and the other as the receiver, which is determined when the connection is established. Both applications use the API to indicate when their data structures are in a ready and consistent state so that data transfer may occur, and updates of the image maintained by each application occur when both sides are ready for transfer. When both sides are ready, PAWS handles the work of determining where each portion of the parallel data object should be sent to or received from, and uses Nexus to perform the data transfer. There are, however, different synchronization possibilities, which affect the behavior of applications at data image update junctures:

- *Fully synchronous* connections require both applications to make sure the other side is ready for data transfer, and to make sure the transfers occur before continuing execution. For example, two simulation components using PAWS to connect the output of one model as the input of a second model would require a fully synchronous connection so that no time steps or frames of the shared data are lost.
- *Fully asynchronous* connections result in data transfer occurring only when both applications happen to be ready at the same time. A visualization program is a good example of this situation: a simulation component may be connected to a visualization component, but the simulation should not be slowed down if the visualization program is busy. An asynchronous connection allows the simulation program to skip data transfers except when the visualization program is prepared to accept the data at the time when the simulation program is ready to send.
- *Partially synchronous* connections act like fully synchronous connections, but only within a user-specified timeout period. If either application, when it reaches a point where it indicates it is ready to send or receive data, waits longer than the timeout period, the application will continue execution without performing the transfer.

The synchronization mode is selected when the connection is established, and can be changed at any time while the connection is active.

5. Parallel Application Interoperability

The most difficult challenge which PAWS must address in order to efficiently couple parallel applications is to provide a mechanism for redistributing data structures from one parallel decomposition pattern to another. This problem

can occur in a number of situations, such as when applications use differing number of processors, or use different parallel layout strategies.

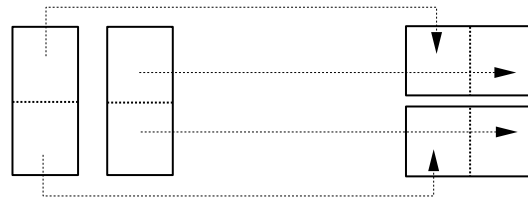


Figure 3. The communication schedule necessary to redistribute data from a 2D parallel array with a parallel-serial layout to one with a serial-parallel layout. PAWS provides a mechanism to calculate this communication schedule and to perform the data transfer.

Consider the simple case of sharing data in a two-dimensional array between two parallel components, the first using a parallel-serial distribution, the other serial-parallel, as shown in Figure 3. The communication pattern needed to redistribute data from the former to the latter layout is indicated in the figure. When PAWS establishes a connection between two data objects, the controller is informed of the parallel layout for the data in both of the applications, and computes the communication schedule for the data transfer. This schedule is given to both applications, which use the information to send and receive the necessary messages for the transfer. PAWS avoid the bottleneck of serializing all messages through a single communication channel by establishing Nexus communication pathways from each process in one parallel application to the other connected application processes that communicate with that node.

PAWS includes the flexibility to support new types of parallel data structures, through extension of PAWS base classes. When a parallel data structure is registered by an application as available for connection to another program, the application must supply information about the parallel layout, the location, and the storage type of the data. The PAWS API includes the following classes to represent this information:

- `PawsRepresentation` manages the information about how the data is distributed among processors. It contains the functionality to build a schedule of messages which must be sent or received during a parallel data transfer.
- `PawsDescriptor` contains a reference to the data itself, and provides the functionality to create, send, and receive the necessary messages during an update. The message schedule generated by

PawsRepresentation provides the information about what messages are required.

- PawsData objects, created by the user for each parallel data object registered with the controller, each contain a single PawsDescriptor instance, and one or more pairs of PawsRepresentation instances, one for each active connection in which the the data object is involved. The PawsRepresentation pair includes the local representation, and the representation within the connected application. When a connection is established, the controller sends the PawsRepresentation information to both sides, which can then compute the necessary communication schedule.

The PawsRepresentation class stores the parallel decomposition information in a standard PAWS format, consisting of a list of domains of the form $(first, last, stride)$ and their mappings to physical processors. For many data structures this is sufficient, and if a new data structure can represent its layout in this format, PAWS can calculate the necessary schedule to send data from one specific representation to another. Use of this standard representation format allows PAWS to provide interoperability between many parallel applications – as long as both applications in a connection provide a data structure with the same total size and a representation in standard PAWS format, PAWS can efficiently transfer the data between the two parallel layouts. If a new data structure cannot be represented in the standard PAWS format, however, a specialized *user* representation may be provided. In this case, the user must also provide the mechanism for generating the communication schedule to map this user representation to a different layout. Parallel data objects using specialized user representations can only be linked with other data objects using the same specialized representation, however.

The PAWS API includes a simple PawsArray object as a sample parallel data structure capable of working within the PAWS Workspace. Figure 4 summarizes the PawsArray class and its relationship with other PAWS support classes. PawsArray extends the PawsDescriptor object, while a PawsArrayRepresentation class extends PawsRepresentation. The new representation class stores the layout information for PawsArray in a standard PAWS layout format, which PAWS uses to generate a communication schedule. To enable a new parallel data type to work within the PAWS Workspace, the user need only extend the existing PAWS classes in the same way as was done for PawsArray. As an example, the POOMA [12, 13] parallel application framework has been updated to allow its parallel multidimensional array data structures to interoperate with other application

components using PAWS.

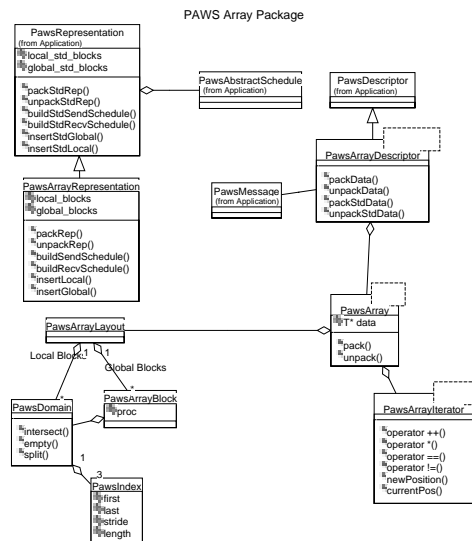


Figure 4. The PAWS class hierarchy used for the PawsArray parallel data structure.

6. Performance

To characterize the behavior of applications which use the PAWS API to share data, we consider the case of two simulation codes exchanging values for 3D arrays of size N^3 . Both applications contain two arrays A and B use them in a series of computations and data transfers. The first application, which represents a “compute” component, first sends array A to the second application, performs a complex computation involving A, then receives new values for array B from the second program. The second application, representing a simple “filter” component, simply receives new values for A, copies A to B, and transmits B back to the first application. This models the situation where two applications are linked with different performance characteristics, and share multiple data objects. The compute component in this example performs a 3D stencil operation, a common operation in partial differential equation solvers. Both benchmark codes are implemented using the POOMA Framework, and use a fully synchronous coordinate mode. Figure 5 illustrates the actions of application 1 (the compute component) and application 2 (the filter component).

The two applications were run using a number of different parallel node combinations, using a single 32-node SGI Origin 2000 parallel machine at Los Alamos National Laboratory. The operations summarized in Fig. 5 were performed 100 times; the total simulation time, the time to

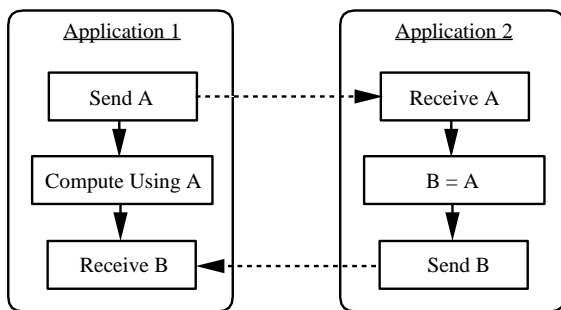


Figure 5. The benchmark used to characterize the performance of two parallel applications communicating using PAWS.

perform PAWS send operations, and the time to perform PAWS receive operations were measured for both applications. Runs were performed using 1, 2, 4, and 8 nodes for the applications, in all possible combinations, for 3D arrays of size N^3 with $N=64$ and $N=128$.

Figure 6 shows the average total simulation time for each node of the first (compute) application for different combinations of nodes. Figure 6a gives the results for $N=64$, while Fig. 6b gives the results for $N=128$. The axis on the right gives the number of nodes used for the second (filter) application, while the axis on the left gives the number of compute application nodes. The total times for the filter application are almost identical, since the codes are synchronized. The shapes of the graphs for $N=64$ and $N=128$ are quite similar; the $N=128$ times are in general a factor of eight larger than those for $N=64$, since there is 8 times more data for $N=128$ than for $N=64$. There are two important trends to the data:

1. For a fixed number of compute application nodes, as you increase the number of filter nodes the total simulation time decreases by relatively small amounts.
2. For a fixed number of filter application nodes, as you increase the number of compute nodes the the simulation time decreases by large amounts, in general almost a factor of two when the compute nodes are doubled.

The reduction in time due to increasing the number of filter nodes illustrates the performance gain from using multiple communication pathways, and indicates PAWS communication benefits from this capability. Reduction in simulation time as we increase the number of compute nodes, close to a factor of two in almost all cases, is to be expected since the majority of time is spent performing the 3D stencil operation and this parallelizes well. This continues to scale in combination with different numbers of filter nodes, with the exception of large numbers of compute nodes and small

numbers of filter nodes. Normally, the compute application requires more time than the filter, but in this case, the filter application becomes the bottleneck.

Figure 7 breaks the total times shown in Fig. 6a for $N=64$ into the times to perform the PAWS send (Fig. 7a) and receive (Fig. 7b) operations. These times are for the first application, which performs the computation; the times for the second filter application are similar, but with longer send and receive times in most cases resulting from the fact that the compute application is slower due to the 3D stencil computation. The times to perform PAWS send operations are in all cases less than 5 percent of the total simulation time. Receive operations, which include time spent waiting for all necessary messages from the other nodes, take noticeably longer, from 15 to 30 percent of the total simulation time.

The send time graph reveals that for a fixed number of filter nodes, increasing the number of compute nodes reduces the send time, since each compute node has less data to send in smaller numbers of messages. This trend matches the total simulation time trend. However, for a fixed number of compute nodes, increasing the filter node count slightly increases the total send time. More filter nodes require, in most configurations, more messages to be sent by each compute node. This extra overhead is slight compared to the total run time, however, and is more than offset by improvement in the time to receive. The only anomaly in this graph is the case of 8 compute nodes and 1 filter node, a result of the filter application becoming slower than the compute application.

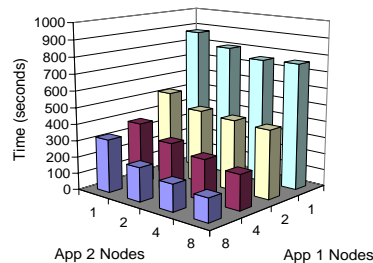
The most striking feature of the receive time graph is the decrease in the time for application 1 to receive the data back from application 2 as you increase the number of nodes in application 2. For a fixed number of nodes in application 1, the amount of data each node must receive is the same, but this is done in smaller increments with more messages which helps to hide communication latency. In contrast, the receive time for a fixed number of filter nodes as the number of compute nodes is increased is fairly constant, and is basically a function of the number of filter nodes. One would expect that larger numbers of compute nodes, which must each receive smaller amounts of data, should require less time to receive their values. That this is not seen is not a fundamental limitation of using multiple connection pathways, however, but a characteristic of the current PAWS implementation which is addressable through reduction in the overhead associated with maintaining a fully synchronous connection mode.

References

- [1] *The Common Object Request Broker: Architecture and Specification (Draft)*, 10 December 1991. Revision 1.1.

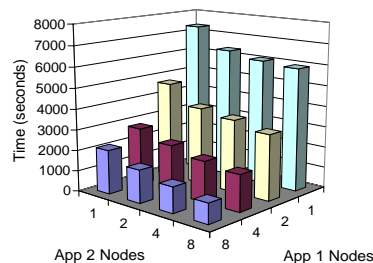
- [2] R. Englander. *Developing Java Beans*. O'Reilly & Associates, 1997.
- [3] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomputer Applications*, 1998. In press.
- [4] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *J. Parallel and Distributed Computing*, 37:70–82, 1996.
- [5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- [6] A. Geist, J. Kohl, and P. Papadopoulos. CUMULVS: Providing fault tolerance, visualization and steering of parallel applications. *Intl. J. Supercomputer Applications*, 11:224–235, 1997.
- [7] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.
- [8] S. Hackstadt and A. Malony. Distributed array query and visualization for high performance fortran. In *Proceedings of Euro-Par '96, Lyon, France, 1996*.
- [9] T. Howes and M. Smith. *LDAP: Programming Directory-Enabled Applications With Lightweight Directory Access Protocol*. Macmillan Technical Publishing, 1997.
- [10] K. Keahey and D. Gannon. PARDIS: A parallel approach to CORBA. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing, Portland, OR, August, 1997*.
- [11] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [12] J. V. W. Reynders. The POOMA framework: A templated class library for parallel scientific computing. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 14-17, 1997*.
- [13] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn. POOMA. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, chapter 14, pages 547–587. The MIT Press, 1996.
- [14] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997.

Total Time for N = 64



(a)

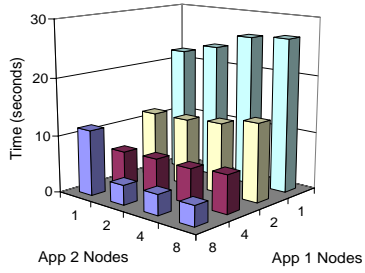
Total Time for N = 128



(b)

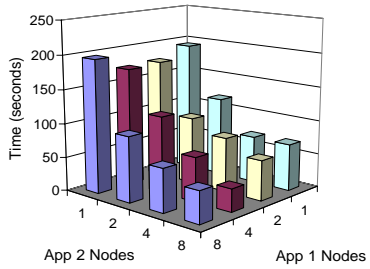
Figure 6. Summary of the total time required to perform 100 iteration loops for application 1 (the compute application), for two different problem sizes. N refers to the number of elements along each axis of the 3D data structure. The total time includes the time to send, compute, and receive.

Send Time for N = 64



(a)

Receive Time for N = 64



(b)

Figure 7. Send and receive times for the case of N=64, from the same runs as shown in Fig. 6