Chapter 4

Porting HPCC Applications onto the World-Wide Virtual Machine Platform

A major goal of this thesis was to make various commonly used programming models available on top of the WWVM. Due to the construction of its communication layer, the WWVM naturally supports PVM-based [GBD+ 93, DGMS 93] message-passing programming. WWVM also provides an interface layer that emulates MPI, Express, and TCGMSG message-passing calls in terms of PVM library calls. For many potential users, parallel programming on the WWVM could be tedious if message-passing programming were the only parallel programming model supported by the WWVM. The programmer had to deal with low-level synchronization and message-passing details in such a case. A representative library-based, shared-memory model (Global Arrays) and a language-based, shared-memory model (HPF) were therefore ported onto the WWVM platform. This chapter will discuss the programming models supported by the WWVM and their implementation in the context of the WWVM.

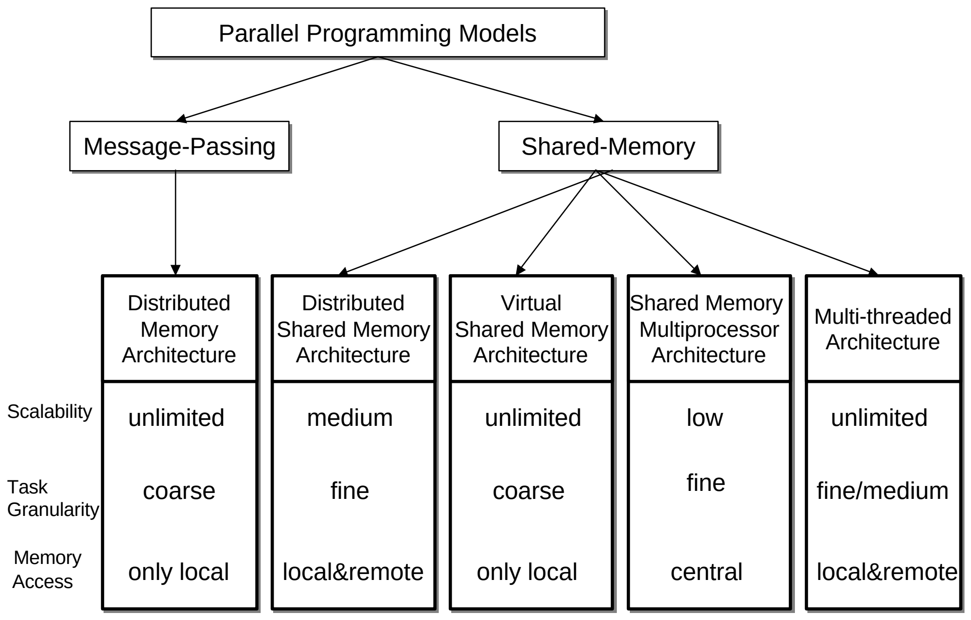
Parallel Programming Models
Message-Passing
Shared-Memory
Distributed Memory Architecture
Distributed Shared Memory Architecture
Virtual Shared Memory Architecture
Shared Memory Multiprocessor Architecture
Multi-threaded Architecture
Scalability
Task Granularity
Memory Access
unlimited
medium
unlimited
low
unlimited
coarse
fine
coarse
fine
fine/medium
only local
local&remote
only local
central
local&remote

Figure 4 1. Message-passing and shared-memory programming models implemented on top of various types of parallel architectures, and their prominent features.

4.1 Parallel Programming Models

Programmers view a computer in terms of a high-level abstraction called a *programming model*. The programming model defines the types of operations available to the programmer. For sequential computers, the von Neumann model serves as a common programming model but parallel computing lacks a single, universal programming model. Parallel programming models can be divided into two broad categories, the message-passing model and the shared-memory model, are further divided into subcategories, depending on their implementation. These memory models have a significant impact on the amount of effort required to produce correct and efficient parallel programs.

In the *message-passing model*, program variables are partitioned among the instances of some type of a *processor* abstraction. Each process can directly access to local variables on its own processor, while it must send messages to other processes in order to access their variables. The message-passing model is based on *send* and *receives* that are usually implemented as part of library packages with Fortran and C programming interfaces. The user is responsible for all aspects of parallelization and data placement. In the *shared-memory model*, processes can access all variables in the same way, irrespective of their location. This makes the shared-memory model easier to use than the message-passing model.

As shown in Figure 4-1, the message-passing programming model is the native model for distributed-memory parallel architectures. On the other hand, by a combination of architecture and operating system mechanisms, a shared-memory abstraction can be supported on a wide variety of parallel architectures. Further information about these mechanisms can be found in [Giloi 94]. However, only the user-level, software-based shared-memory models in the framework of the WWVM will be investigated here. The two common software approaches for implementing a user-level, shared-memory model are library-based and language-based.

4.1.1 Library-Based Shared-Memory Implementations

Library-based implementations provide a set of software library routines for automatically accessing non-local partitions of data indirectly through pointers obtained from these routines. The library package is linked into programs written in a conventional sequential language. Library-based implementations have the advantages of portability and simplicity, but they are limited to a subroutine-call interface and cannot take advantage of a compiler-based static analysis of program control and data flow. Global Arrays libraries that will later be elaborated on in Section 4.4 are an example for this category.

4.1.2 Language-Based Shared-Memory Implementations

Language-based implementations are accepted as a more viable long-term solution for shared-memory parallel programming than library-based shared-memory implementations, because the use of specialized compilers relieves the user of the burden of operating at the low level required by library-based implementations. Language-based shared-memory implementations are targeted toward promoting a wider use of parallel systems and can be presented in four groups [Zoma 96]:

1. *New parallel languages.* Specially designed parallel languages can help to express only application algorithms, not the implementation issues dictated by hardware and system software. The parallelism inherent in an application is made available to the compilers, and the potential parallelism can be utilized to take advantage of system resources. Sisal [McGr 93] is the best known of such languages in the high-performance computing community. It is a functional language that provides special constructs to express scientific parallel numerical algorithms in a form close to their mathematical formulation. The Sisal compiler automatically detects the parallelism inherent in the program and exploits it.
2. *Coordination languages*. These languages are targeted toward coordinating parallel activities. A parallel program is viewed as a collection of interacting distinct processes with coordination operations such as synchronization, information exchange, or process management. Linda [Gele 85, Gele 88, Gele 89, GCCC 85, GelP 90] is a representative example of this category. Evolved from a Yale University research project and that has become commercialized, it uses an abstraction called “tuple-space” via which cooperating processes communicate by writing or reading data items. Tuple space is a shared, associatively addressed, semi-persistent store that holds tuples. Tuple space effectively

replaces message passing and shared memory. A Linda program is initiated by putting tuples containing code in tuple space. These tuples are then extracted and executed.

3. *Parallelizing compilers and preprocessors.* The compiler of an existing sequential language is enhanced to detect inherent parallelism in a sequentially constructed program. The principal advantage of this approach is that sequential programs can be moved to the target parallel machine without requiring any changes in the existing codes, thus exploiting its parallel facilities relatively inexpensively and quickly. Early work on parallel Fortran dialects such as Parafrase [PGH+ 89] focused on restructuring loops to minimize data dependencies and spread the iterations of the loop across different processors.
4. *Extending an existing language with features that explicitly deal with parallellism.* This approach requires some minor changes to the existing software before being ported onto a parallel machine. Many extensions have been developed by different groups using the same language base, thus leading to the definition of nonstandard variants. This approach is reflected in several Fortran proposals recently proposed for distributed-memory systems programming, such as Vienna Fortran [CMZ 92] and Fortran D [FHK+ 91], and their standardized version, High Performance Fortran [HPFF 93]. HPF is an emerging shared-memory parallel language standard that makes the underlying distributed-memory architecture transparent to the programmer. Similar approaches have been taken in the development of pC++ [BFG+ 93, LiG 91, MMB+ 94], a parallel C++ dialect, and Jade [RinL 93, RSL 93], a parallel dialect of C; pC++ is suitable for expressing medium to coarse-grained parallelism and its *distributed collection model* represents a data structure as a collection of homogeneous elements and allows the programmer to specify how the data structure is to be partitioned among the processors. Communication is done through messages and synchronization implemented through the compiler. Jade is a parallel programming

language designed to exploit coarse-grained control parallelism in a shared-memory system. It provides convenience of shared-memory model by allowing any task to access shared objects transparently. The programmer, however, must supply information about how to decompose the computation into tasks and how tasks access the shared objects.

4.1.3 Evaluation of Message-Passing and Shared-Memory Models

Using parallelizing compilers is very attractive in terms of application development, but their performance is limited by the fact that they only parallelize loops. On the other hand, the application and development of new languages has been hindered by the fact that vendors will support a new language only if many users request it, and many users will try a new language only if there are mature vendor-supported compilers. As a result, evolutionary approaches such as extending a language with parallel statements or directives have become more practical. A few, such as HPF, have succeeded in being commercialized. Developers and users of extended languages report encouraging performance results (see [BCF+ 94, HKT 92, MMB+ 94]), but still more work is needed in order to deliver a scalable high performance for real-world applications.

Although the shared-memory model automatically provides global memory abstraction, because of the limitations in current tools many real-world application developers still prefer low-level tools such as message-passing libraries that have better performance figures as compared to shared-memory tools. Actually, the message-passing model is the most commonly used parallel programming model in today's high-performance computing arena, and the problem of having numerous nonstandard variants of message-passing libraries is about to disappear with the arrival of MPI. The MPI standard [MPIF 94] of 1994 defines both the syntax and semantics of core message-passing library routines and, with its portable, high-performance implementations, MPI will soon become the most commonly used (and possibly the only) message-passing library.

4.2 MPI–to–PVM Interface Layer

This section describes the building of MPI protocols over the underlying PVM communication layer of WWVM. The main features of MPI not supported by PVM are summarized and then porting strategy is described.

4.2.1 Message-Passing Interface (MPI)

Users' and developers' experience with PVM and other first generation libraries formed the foundation for standardizing message-passing interfaces. This experience revealed two main weaknesses in those early systems:

1. *Lack of safe message passing* that causes a message exchange in one part of a program to interfere with the message exchange in other parts of the program, although such interference is not intentional.
2. *Lack of flexible group operations,* although many applications require that processes be organized into groups and that global operations be performed only over the members of a particular group.

MPI, on the other hand, is a large step forward in support of standard library development. It was proposed as a standard message-passing interface by a committee of vendors, implementers, and users [MPIF 94, GLS 95]. Its design gathers the most attractive features of a number of existing message-passing systems such as the work at the IBM T. J. Watson Research Center [BBC+ 93, FBH+ 92], Intel's NX/2 [Pier 88], ParaSoft's Express [FKB 91, Kola 92, Para 90, Para 92], nCUBE's Vertex [nCUB 90], PARMACS [Hemp 91] **(**developed at ANL), and p4 [LO+ 87, ButL 92, LusB 92] which is a successor of PARMACS. Other important contributions have come from Zipcode [SkjL 90, SSL+ 92] **(**developed at Livermore), Chimp [CHI 91, CHI 92] **(**developed at UK), PVM [GBD+ 93], Chameleon [GS 93], and PICL [GHP+ 90].

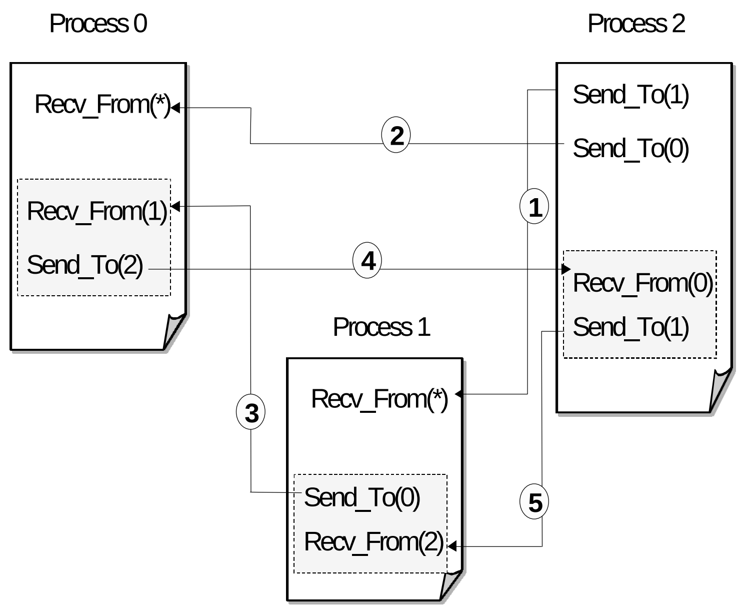

Figure 4-2. Illustration of the intended behavior of a message-passing code with embedded library calls.

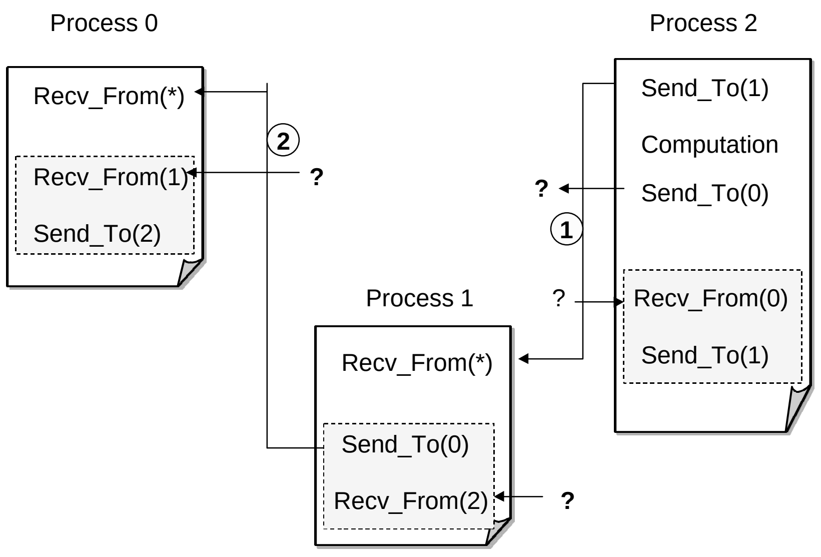

Figure 4 3. Illustration of a possible anomaly (i.e., deadlock situation) in a message-passing code when contexts are not used in message-passing calls.

MPI defines both the syntax and semantics (i.e., behavior) of a core set of library routines. Those routines will be useful to a wide range of users, can be efficiently implemented on a wide range of computers, and will establish a high performance on both massively parallel machines and clusters of workstations. MPI focuses on the standardization of process communication, synchronization, and group operations. It supports contexts and groups, and it adds some new functionality not found in PVM such as thread safety, user-defined data types, gather/scatters, overlapping groups, and virtual topologies. It offers portability and high performance for a wide variety of problems.

Several MPI implementations are built upon established message-passing libraries. Chameleon-based MPICH [DGLS 93, GLDS 96] is a product of the collaboration between the Argonne National Lab and Mississippi State University; LAM MPI [BDV 94] comes from the Ohio Supercomputing Center; the Chimp implementation of MPI was developed at Edinburgh Parallel Computing Centre; and Unify [Cheng 94] of Mississippi State University runs over the PVM. There are also other freely available versions of MPI for Windows NT workstations, MS-Win32 clusters, MS-Windows 3.1, and SPARCstation clusters. There also exist proprietary MPI versions for Cray T3D, IBM SP-2, Fujitsu AP1000, and SGI Power Challenge from their vendors.

In the MPI programming model, a computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes. In most MPI implementations a fixed set of processes is created at program initialization, and one process is created per processor. However, these processes may execute different programs, hence the MPI programming model is sometimes referred to as *multiple-program/multiple-data* (MPMD) – in which different executable codes with separate data sets are assigned to each process – to distinguish it from the *single-program/multiple-data* (SPMD) model. In the SPMD model, every processor executes the same program but takes different branches, depending on the unique identifier assigned to each process.

4.2.1.1 Safe Message Passing in MPI

Applications often need to call subprograms (such as numerical solvers) and perform global operations (such as global summation). These subprograms may be developed as libraries by different organizations. In these cases, messages used in the subprograms must not unintentionally interfere with the messages used in other parts of the program.

There are two ways in which a call to a library routine can lead to ambiguities in the message traffic of different libraries or in the rest of the application:

- A message may be received incorrectly in the library routine if the application process enters a library routine synchronously when a send has been initiated but the matching receive is posted after the library call either for overlapping communication and computation or because of some application-dependent requirements.
- A non-deterministic behavior may occur, as shown in Figures 4-2 and 4-3, when different processes enter a library routine *asynchronously*. Embedded library calls are shown in shared areas. The intended behavior is that processes 0 and 1 each receive a message from processor 2 and all three processes call a library routine. In the library routine, each processor receives a message from its right neighbor and sends a message to its left neighbor. To achieve high performance, it is a common practice to use a `receive()` that matches any incoming messages. If contexts are not used, this may fail and a wildcarded receive in process 0 may be satisfied by a send from process 1 instead of process 2 if process 2 lags a little before the second send operation. Obviously, this will lead to incorrect results, as illustrated in Figure 4-

3. Furthermore, this kind of intermittent, time-dependent error can be very difficult to reproduce and locate.

4.2.2 Implementation of the MPI-to-PVM Interface Layer

The MPI-to-PVM interface layer emulates MPI calls in terms of PVM calls. It is based on Mississippi State University’s public domain software, Unify[1] [Cheng 94, CVRS 94, VSRC 95]. Besides Unify, there are two other experimental MPI implementations on top of the PVM system, namely, PVMPI [FagD 96] from the University of Tennessee and EZP [Cole 96] from NASA. However, these became available much later than Unify, therefore Unify was used as the only available PVM-based MPI implementation when this study was conducted.

Unify supports a dual-API that permits the communication functions of PVM, MPI, or both message-passing libraries to co-exist in the same application program. This enables users to take current PVM applications written in C or Fortran and slowly migrate toward complete MPI applications without having to make a complete conceptual jump from one system to another.

Unify modifies PVM functions by adding the communication “contexts” needed for MPI protocols. Unify also addresses the difficulty of mapping identifiers between the PVM and MPI domains where each system uses a different scheme; PVM uses a 32-bit integer and MPI uses a handle to an opaque internal structure together with a rank inside that structure. Unify provides only two new additional calls: one from MPI to PVM task identifier (task ID), and vice versa. Although all the MPI intra-communicator point-to-point and collective operations and communicator management and environment management functions are supported, Unify does not support virtual topologies, profiling, attribute caching, and inter-communicators.

[1] Version 0.9.2 of September 1994.

	Point-to-point Primitives	Collective Operations	Groups	Context	Application Topologies	General Data Types
PVM	yes	yes	yes (dynamic)	no (yes in v3.4)	no	no
MPI	yes	yes	yes (static)	yes	yes	yes

Table 4 1. A general comparison of functionality provided by PVM and MPI.

Unify supports only the SPMD model where all processes get copies of the same code. Unify failed to exploit PVM's dynamic spawning capability and forced the user to spawn a fixed number of master–slave SPMD processes from the command line.[2] Thus no other PVM process (including the console process) could start a Unify application.

Since the Unify project was originally a master's project and has never reached full maturity, numerous bugs in Unify had to be eliminated in order to cope with the requirements of real-world applications written in MPI. Making this interface bulletproof was an important step during the implementation of the WWVM, because the proper functioning of the communication and runtime support libraries of the Syracuse F90D/HPF compiler, a large number of benchmark programs, and Global arrays libraries rely on this interface. The implementation of the WWVM MPI-to-PVM interface layer necessitated adding new routines to Unify in order to support Cartesian coordinates processor topologies. Ports to other systems that would be part of the WWVM were also accomplished.

[2] More specifically, the start-up sequence consisted of a process that checked to see whether it was a master by the existence of a parent process and then spawned N-1 copies of itself. If a parent existed, the process was assumed to be a slave and would block on a *receive()*, awaiting a task id list so that it could build its MPI_COMM_WORLD, MPI_COMM_SIZE, and MPI_COMM_RANK values [FagD 96].

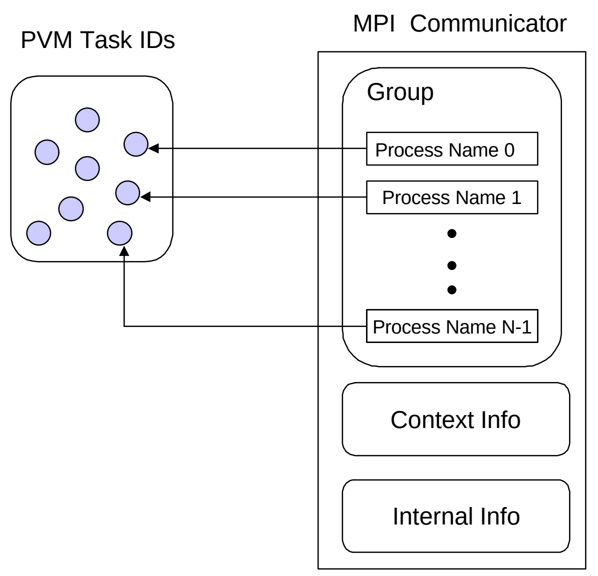

Figure 4 4. An MPI communicator consists of a group, context and some other internal information. Group Ids are matched to PVM task Ids.

The implementation of the interface layer functionality is not difficult as far as the following two points are taken into consideration:

1. PVM point-to-point primitives and collective operations have poor performance characteristics. In addition, PVM functions are unable to overlap computation and communication. PVM is more appropriate for problems where the message-passing performance is not very important. This low performance issue is not a problem in WWVM, since the targeted problems are coarse-grained and not communication intensive.

2. The PVM library has significantly less functionality than the MPI library. As shown in Table 4-1, PVM lacks the contexts, certain group operations, application topologies, and general data types found in MPI. The forthcoming sections will present a detailed comparison of MPI

and PVM routines and emphasize ways to present the required MPI functionality by using PVM routines.

4.2.2.1 Contexts, Process Groups, and Communicators

Communication Contexts

MPI uses a system-defined tag, or *context*, to divide a communication domain into non-interfering subdomains. Contexts are generally used by different layers in a library to eliminate possible interference between those layers, i.e., to ensure safety, when using libraries with embedded message-passing within an application. In PVM, for example, any task can send a message to any other task, whether the receiving task wishes to interact with the sender or not (as in the case of two separate applications). In MPI, on the other hand, the two applications have two separate message universes or contexts and only processes in the same context can communicate with each other. The use of contexts also promotes software modularity by allowing the construction of independent communication streams between processes.

There are two alternative methods for representing communication contexts in PVM:

- Communication contexts can be directly embedded within PVM tags by using the upper half-word to represent the communication context, while leaving the lower half-word to keep the message tags. However, this limits the range of MPI tags to 64K and does not permit wildcarded sends and receives.
- Communication contexts can be carried as separate words with communication primitives. Unify uses this approach by upgrading the receipt selectivity of certain primitives and extends

the PVM's message envelope[3] to include the sender's or receiver's rank, message tag, and communicator. Modified PVM primitives are transparent to the user and are called indirectly through macro extensions.

Process Groups

A *process group* is an ordered collection of processes (or tasks) in which each process is uniquely identified by its rank within the ordering. Process groups are useful in two situations: first, they can be used to specify the processes involved in collective operations; second, they are used to introduce MIMD task parallelism to the application by helping to assign different tasks to different groups. Process groups play a crucial role in the representation of certain data structures of complex applications, such as fluid-structures in interdisciplinary applications, or rows/columns of a matrix in linear algebra computations.

PVM processes can join and leave any number of groups at any time, making membership completely dynamic. Processes are allocated instance numbers in the order in which they join a group. The first join operation creates the group, which is destroyed when membership falls to zero. This dynamic characteristic brings inefficiency, since it requires a group server that supervises all collective operations in order to resolve race conditions.

MPI provides extensive support for groups, including overlapping groups (overlap possible in PVM but not as useful) and virtual topologies on groups. Processes in MPI are arranged in rank order from `0` to `N-1`, where `N` is the number of processes in a group. These process groups define the scope for all collective operations within that group. The process groups in Unify are implemented as a list of PVM task IDs.

[3] In addition to the data part, the messages carry information that can be used to distinguish them and selectively receive them. This information is called the *message envelope*, and consists of a number of fields such as source processor, destination processor, tag, and possibly a communicator in the case of MPI.

Although the MPI standard does not state how processes are started, it does state how and in which order processes become MPI processes. All MPI processes join the MPI system by calling `MPI_Init()` and leave by calling `MPI_Finalize()`. Once all the expected processes have joined the system, a common communicator, `MPI_COMM_WORLD`, is created that allows all processes in that "world" to communicate with each other. Communications between processes within the same communicator or group are referred to as *intracommunicator communications*.

Communicator Objects

The process group and context, together with other information about topologies and local attributes, constitute a *communicator* (Figure 4-4). Use of a communicator object specifies the scope of a communication operation and isolates the communication operations within that communicator from other communication that is taking place within the system.

When sending or receiving a message, the process and message identifiers must be specified. A process involved in a communication operation is identified by group and rank with that group (i.e., `Process ID=(group, rank)`), whereas messages are considered labeled by communication context and message tag within that context (i.e., `Message ID=(context, tag)`). The group and context are specified by means of a communicator object in the argument list of the send and receive routines. Within a given scope, the group and context components of a communicator must be indicated, whereas either or both of the rank and tag parameters may be wildcarded.

4.2.2.2 Application Topologies

Application topologies help to efficiently assign processes to physical processors and handle the translation between process identifier and location in the topology, and vice versa. An MPI group may be assigned a Cartesian or graph topology. General application topologies are specified by a *graph topology* with arcs connecting communicating processes. In many other applications the processes are arranged with a particular *Cartesian topology* instead, such as a two- or three-dimensional grid with periodic or non-periodic boundary conditions in any specific dimension.

Cartesian topologies provide support for shifting data along a specified dimension of a Cartesian grid. It is also possible to perform collective communication operations, such as multicast, within groups by partitioning a Cartesian grid into hyper-plane groups by removing a specified set of dimensions. Cartesian topology operations are extensively used in the implementation of runtime support libraries and communication libraries of the F90D/HPF compiler, as well as in many other parallel applications with regularly distributed data structures.

Cartesian topologies are initialized by a special routine, `MPI_CART_CREATE()`, that specifies the topology of a given group, and new processes are added into a topology using the `MPI_CART_INC()` function. Cartesian query functions for returning the associated topology for a group (`MPI_TOPO_TEST()`), determining the size and periodicity of a topology (`MPI_CARTDIM_GET()`), finding the rank for a given location (`MPI_CART_RANK()`), and for determining the location of a process in the topology (`MPI_CART_COORDS()`) were implemented. Cartesian functions for set operations like *union*, *intersection*, and *difference* were also added to the implementation. For shift operations, a special function called `MPI_CART_SHIFT()` returns the ranks of the processes that a process participating in the shift must send data to and receive data from. Once the source and destination processes are known for each process, the shift is performed by calling the routine `MPI_SENDRECV()` that allows each process to send data to one process while receiving data from another.

4.2.2.3 Point-to-Point Communication

Point-to-point send and receive modules are the core of any message-passing library. The differences between two message-passing libraries come from the semantics and implementation of functions. *Semantics* is the behavior of a library routine, while the *implementation* consists of the low-level coding details required for insuring correct semantics. Whether or not the send buffer can be modified immediately after a message is sent is related to semantics, but using buffered or unbuffered messages in send operations is an implementation decision.

Both MPI and PVM provide blocking and non-blocking point-to-point send and receive (Table 4-2). Non-blocking sends can be matched by the blocking receives, and vice versa.

In *blocking*[4] or *synchronous* communication, control is not returned to the calling process until the communication action has completed. The process has no requirement to check for completion of the communication and is assured of the integrity of the communication buffer when it receives control.

In *non-blocking*[5] or *asynchronous* communication, control returns to the calling process before the actual communication action has been completed. This allows the processes to perform other tasks while waiting for communication to complete (i.e., overlapping communication and computation.) The burden of insuring uncorrupted data belongs to the process making the call. This process also needs to check for completion of the send/receive before reusing the communication buffers.

[4] A routine is *blocking* if its completion (return of control to the calling routine) may depend on an external event (an event that is outside the control of the routine itself.) Example: a send is blocking if it does not return until there is a matching receive.
5 A routine is *non-blocking* if it is guaranteed to complete regardless of external events. Example: a send is non-blocking if it is guaranteed to return whether or not there is a matching receive.

	Routine	Blocking	Comment
PVM	**pvm_send**	no	may block under unusual circumstances, returns whether or not a matching receive
	pvm_recv	yes	
	pvm_nrecv	no	
	pvm_trecv	yes	returns if message does not arrive within a specified period of time (timeout)
MPI	**MPI_Send**	yes	implementation may block or not
	MPI_Isend	no	
	MPI_Ssend	yes	synchronizes with receiver
	MPI_Issend	no	synchronizes with receiver
	MPI_Bsend	no	employs a user-supplied buffer
	MPI_Ibsend	no	
	MPI_Rsend	no	a matching receive must be waiting
	MPI_Irsend	no	a matching receive must be waiting
	MPI_Recv	yes	
	MPI_Irecv	no	

Table 4 2. Semantics of PVM and MPI point-to-point primitives.

A routine is *synchronizing* if it causes two separate processes to become synchronized.

MPI offers a choice of several communication modes that specify the conditions under which the sending of a message may be initiated, or when it completes:

- In the *standard mode* a message may be sent, regardless of whether a corresponding receive has been initiated.

- The *buffered mode* communication operation can be started whether or not a matching receive has been posted. Unlike the standard send, this operation is local and its completion does not depend on the occurrence of a matching receive.
- The *ready mode* requires a message to be sent only if a corresponding receive has been initiated.
- The *synchronous mode* is similar to the standard mode except that a send operation will not complete until a corresponding receive has been initiated on the receiving process.

MPI uses all four modes for message sending, and only the standard mode for message receiving.

Implementation of Point-to-Point Operations

The point-to-point functions of MPI can be mapped one-to-one to the PVM's functions. The same thing is true for environmental management and collective functions. Table 4-3 shows the C bindings for both PVM and MPI; the full Fortran translations are similar. Group arguments for PVM correspond to the dynamic groups, while those for MPI correspond to the static groups.

PVM has both probe and nonblocking receive. PVM's nonblocking receive is not like MPI's `MPI_Irecv()`; rather, it combines the effects of `MPI_Iprobe()` and `MPI_Recv()`. Every send in MPI has a datatype argument that matches PVM's use of different datatypes for different types of data.

An evaluation of MPI's various send and receive primitives can be found in [Saph 94]. According to this study, MPI_Send(), MPI_Recv(),MPI_Isend(), and MPI_Irecv() can be efficiently implemented on any platform, but others have limited usage and poor performance characteristics. Therefore, WWVM MPI-to-PVM interface ignores the translation of those primitives.

MPI	PVM
MPI_Init(argc, argv)	mytid = pvm_mytid()
MPI_Finalize()	pvm_exit()
MPI_Comm_Rank(comm, rank)	rank = pvm_getinst(grp, mytid)
MPI_Comm_Size(comm, size)	size = pvm_gsize(grp)
MPI_Send(buf, buflen, datatype, dest, tag, comm)	pvm_psend(tid, tag, buf, buflen, datatype)
MPI_Recv(buf, buflen, datatype, dest, tag, comm)	pvm_precv(tid, tag, buf, buflen, datatype)

MPI_Iprobe(…)
if (flag) MPI_Recv(buf_type, count, datatype, src, tag, comm, status)
	bufid = pvm_nrecv(tid, tag, len)
	if(bufid>=0) pvm_upk(datatype)

Table 4 3. PVM mapping of point-to-point and environmental management primitives of MPI.

Communication Completion

Following a call to a non-blocking send or receive routine, the handle returned by the call can be used to check the completion status of the communication operation or to suspend further execution until the operation is complete. The functions `MPI_Wait()` and `MPI_Test()` are used to complete a non-blocking communication call. `MPI_Wait()` is a non-local and blocking function that returns when a message has been safely sent (not when it is delivered to the receiver), whereas `MPI_Test()` is local.

	barrier	Global Computation			Collective data movement		
		reduce	user reduce	scan	scatter	gather	broadcast
PVM	Yes	yes	no	no	no	no	yes
MPI	Yes	all-all all-one	yes	yes	one-all all-all	all-all one-all	one-all all-all

Table 4 4. Collective communication routines of PVM and MPI.

Send-and-Receive Operations

Send-and-receive operations combine in one call the sending of a message to one destination and the receiving of another message from another process. This operation is especially useful for executing a shift operation across a chain of processes. The send-and-receive operation also helps to prevent cyclic-dependencies that may lead to deadlock due to improper ordering of separate send and receive primitives. PVM lacks a send-receive primitive, therefore it needs to be implemented using separate send and receive primitives.

4.2.2.4 Collective Communication Operations

Collective communication is the communication that takes place among a group of processes. All processes in a group must call the associated collective communication routine with matching arguments. Collective communication operations include broadcast, synchronization (barrier), global operations (reductions), scatter/gather, and parallel prefix (scan). All collective operations defined by MPI and PVM are blocking and synchronizing. PVM provides a limited number of collective operations such as barrier and reduce (Table 4-4). Also provided is a broadcast operation that allows messages to be sent to all members of a group.

MPI	PVM
MPI_Bcast(buf, buflen, datatype, root, comm)	pvm_mcast(ntask, tids, tag)
MPI_Bcast(buf, count, datatype, root, comm)	pvm_bcast(grp, tag)
MPI_Barrier(comm)	pvm_barrier(grp, count)

Table 4 5. PVM mapping of MPI collective communication routines.

Missing collective operations were implemented using point-to-point operations and required an explicit look-up to process addresses. Simple operations (broadcast, global sum) should scale as *ln N* if implemented with point-to-point (if network scales). Complex operations (all-to-all) should scale as *N* (if network scales well).

As shown in Table 4-5, the MPI broadcast operation maps to the `pvm_bcast()` call in PVM. Note that the tag argument of PVM's broadcast has no corresponding entry in MPI. The tag is implicitly specified in MPI functions using a communicator, while it is explicit in PVM. Furthermore, while PVM uses a list of task IDs to specify the processes that will participate in a multicast or broadcast operation, MPI specifies them by means of a communicator object that is associated with a previously formed process group. Another difference between the two libraries is that PVM always uses the active message buffer in communication operations.

4.2.2.5 General and User-Defined Data Types

Both MPI and PVM can send typed data and allow type checking for C programs. PVM has support for most of the basic datatypes supported by the host language, whether Fortran or C. MPI predefines an MPI basic data type for all data types specified in Fortran and C.

MPI also has some other mechanisms to describe arbitrarily complicated user-defined data types similar to an array of data items or structures. In this way, communication of array sections and structures involves combinations of primitive data types. In order to emulate the user-defined data types of MPI, PVM's pack/unpack functions are used. Data is explicitly packed into a contiguous buffer before receiving it, and unpacked from a contiguous buffer after it is received.

4.3 High Performance Fortran (HPF) and PCRC Runtime Support Libraries

For large-scale scientific computing, where distributed-memory machines dominate the market, much attention recently has been devoted to the development of efficient shared-memory programming models. Most notably, many research groups have cooperated in the development of HPF [HPFF 93]. HPF is an attempt to stop the proliferation of Fortran language extensions. It extends Fortran 90's array operations on whole arrays and array sections with data parallel constructs, intrinsic functions, and data distribution and alignment directives developed in the context of Fortran D [FHK+ 91] and Vienna Fortran [CMZ 92] projects. The programmer specifies how and where data are located, which loops are to be executed in parallel and, indirectly, where the work will be performed. The HPF compiler does the data partitioning and generates the required communication calls.

Syracuse Fortran 90D/HPF compiler [BCF+ 94] and its associated runtime support (RTS) libraries [ABB+ 92] were ported onto the WWVM platform. This compiler is targeted toward implementing a chosen set of functionalities defined in the Subset HPF [HPFF 93] which is a minimal starting set of features from Fortran 90 and HPF to encourage early release of compilers with HPF features. Static data mapping features of HPF such as `PROCESSORS`, `TEMPLATE`, `ALIGN`, and `DISTRIBUTE`, the single statement `FORALL,` and some of the Fortran 90 intrinsics were implemented, whereas the `INHERIT` directive, `INDEPENDENT` directive, `FORALL` construct, and HPF intrinsics were not included in the prototype implementation.

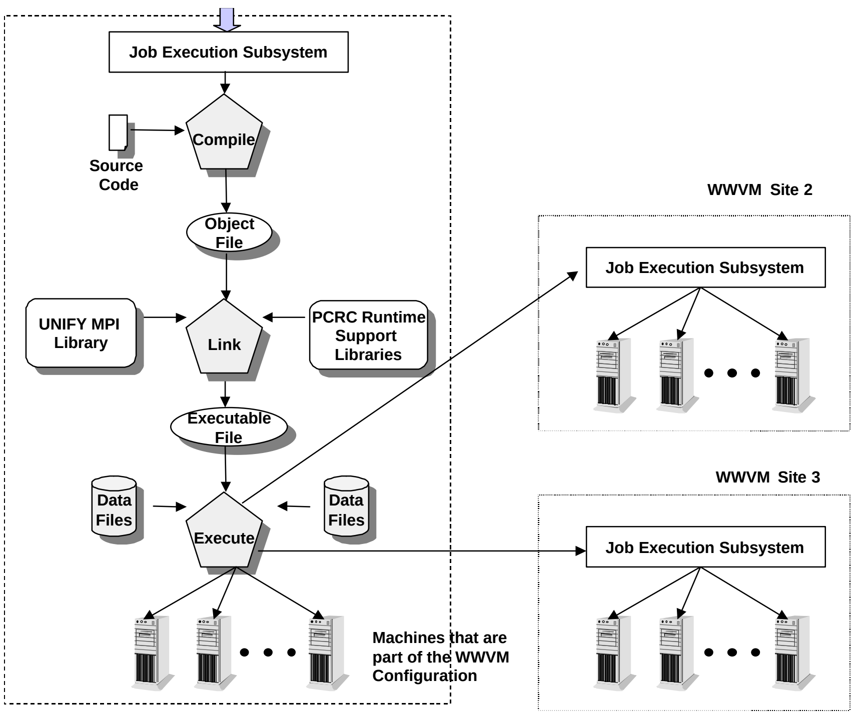

Figure 4 5. The life cycle of a HPF program from compilation to execution on the WWVM.

The Fortran 90D/HPF compiler translates HPF source code into Fortran 77 with calls to the message-passing and RTS libraries. It parses the HPF program, partitions work and data, and detects and generates required communication. The compiler bases its parallelization on the *owner-computes rule,* which causes the computation to be partitioned according to the distribution of the assigned portion of the computation and involves localization based on the left-hand-side (lhs) of an array assignment statement. For example, in a FORALL statement this involves localizing the bounds of the FORALL statement according to the array elements owned by the lhs and adjusting the loop bounds to index the slice of data owned by the current processor.

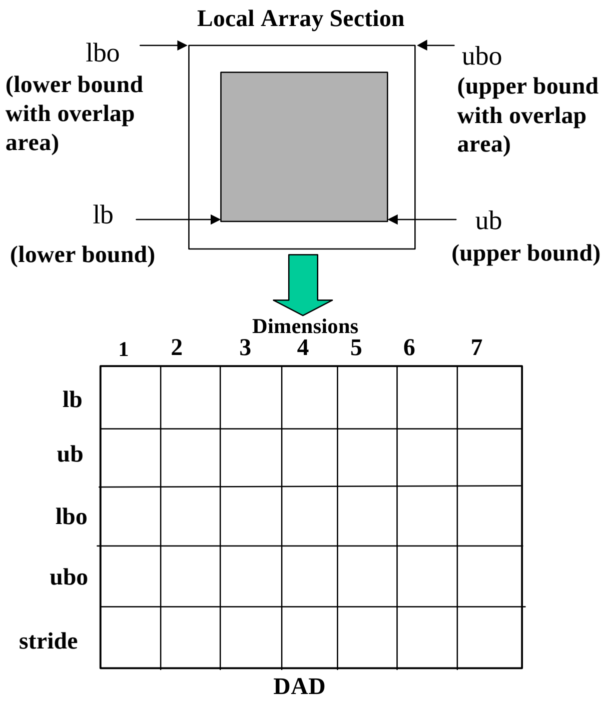

Figure 4 6. A local array section and its corresponding DAD in Fortran 90D/HPF RTS libraries. This is used to pass array specifications to the intrinsic routines.

The compiled SPMD node programs are linked with the RTS libraries to obtain executable codes on each node. The HPF tasks are then mapped onto the nodes of the WWVM with one or more processes per Web server, according to the number of machines coordinated by each server (Figure 4-5.)

The HPF program passes information about an array to the runtime support libraries by means of a distributed array descriptor (DAD). This DAD contains the lower and upper bounds of an array with and without overlap areas in each dimension, and the stride for each array dimension. A representative array section and the specification of its properties to RTS library routines by means of a DAD are displayed in Figure 4-6. Furthermore, the id and Cartesian coordinates of each processor, and the number of processors in each Cartesian dimension, are kept separately and used during the computation of local and/or global coordinates of distributed arrays.

NPAC's PCRC runtime support group has recently replaced the communication layer of the Fortran 90D/HPF RTS libraries with a portable one that uses the MPI for inter-node communication and renamed it as PCRC (Portable Compiler and Runtime Support Consortium) libraries [LDL 95]. This makes the executable codes portable on a wide variety of platforms. By

F90D	**Express**

MINVAL(array, dim, mask)
MAXVAL(array, dim, mask)
	excombine

ALL(mask, dim)
ANY(mask, dim)
COUNT(mask, dim)
	excombine

SUM(array, dim, mask)	excombine
CSHIFT(array, shift, dim) EOSHIFT(array, shift, boundary, dim)	exread & exwrite
TRANSPOSE(matrix) only (BLOCK, BLOCK)	exread & exwrite

MINLOC(array, mask)
MAXLOC(array, mask)
	excombine

DOT_PRODUCT(vectorA, vectorB)	excombine
MATMUL(matrixA, matrixB)	exread & exwrite

Table 4 6. Express functions used in the implementation of Fortran 90D/HPF compiler intrinsic routines.

F90D/HPF	**Express**
BROADCAST	exbroadcast
CONCATENATE	excombine & exconcat
COPY	exread & exwrite
SHIFT	exread & exwrite
SPREAD	exbroadcast

Table 4 7. Express functions used in the implementation of Fortran 90D/HPF compiler collective communication routines.

means of the MPI-to-PVM interface, they were also ported onto the WWVM platform.

4.3.1 Fortran 90D/HPF Intrinsic Functions

Many of the frequently required primitives that operate on one- or more dimensional arrays are provided as part of the HPF language and called *intrinsic functions*. The implemented intrinsic routines are the ones that are included in Fortran 90 and contain:

- All vector and matrix multiply functions: `DOT_PRODUCT`, `MATMUL`
- Array reduction functions: `ALL`, `ANY`, `COUNT`, `MAXVAL`, `MINVAL`, `PRODUCT`, and `SUM`
- One of the array construction functions: `SPREAD`
- All array manipulation functions: `CSHIFT`, `EOSHIFT`, `TRANSPOSE`
- All array location functions: `MAXLOC`, `MINLOC`.

The mapping between Fortran 90D/HPF intrinsics functions and Express functions is given in Table 4-6.

Express	MPI
exparam(&env)	MPI_Init(argc, argv)
rank = env.procnum	MPI_Comm_rank(MPI_COMM_WORLD, rank)
size = env.nprocs	MPI_Comm_size(MPI_COMM_WORLD, size)
exread(buf, buflen, src, tag)	MPI_Send(buf, buflen, MPI_BYTE, dest, tag, comm)
exwrite(buf, buflen, dest, tag)	MPI_Recv(buf, buflen, MPI_BYTE, src, tag, comm, status)
exsend(buf, buflen, src, tag, status)	MPI_Isend(buf, buflen, MPI_BYTE, dest, tag, comm, request)
exreceive(buf, buflen, src, tag, status)	MPI_Irecv(buf, buflen, MPI_BYTE, src, tag, comm, request)
(status < 0) ?	MPI_TEST(request, status)
exvread(buf, buflen, offset, count, src, tag)	MPI_Recv(…) MPI_UnPack(inbuf, inbuflen, datatype, outbuf, outbuflen, pos, comm)
exvwrite(buf, buflen, offset, count, src, tag)	MPI_Pack(inbuf, inbuflen, datatype, outbuf, outbuflen, pos, comm) MPI_Send(…)

Table 4 8. One-to-one mapping of Express and MPI point-to-point communication and environment management routines.

4.3.2 Collective Communication Functions

The *collective communication* routines perform common forms of data movement operations among processors such as broadcast, concatenate, copy, shift, and spread. The Fortran 90D/HPF compiler analyzes the given HPF program, automatically detects patterns that match the collective communication calls, and replaces them with calls to these routines. Express functions used in the implementation of the collective communication functions are shown in Table 4-7.

4.3.3 Implementing Express Routines in Terms of MPI Routines

The original Fortran 90D/HPF libraries employ the Express message-passing libraries for conducting inter-process communication on distributed-memory machines. Express is a commercial product that includes a portable message-passing library as well as other associated tools and utilities for debugging, graphics, and performance tuning. Its basic functions can be ported to MPI in a one-to-one fashion, as shown in Table 4-8.

Express has a simple set of collective operations that are limited to synchronization, broadcast, and combine. Many of the collective operations are performed by specifying different functions to the `combine()` function. All or some of the processes may participate in the collective operations. The list of processors that will participate in an operation is specified by an argument. They can be simply converted to MPI collective functions (Table 4-9). The MPI communicator argument specifies the processors participating in the collective communication operations, which should be explicitly specified using two arguments, *`nnodes`* and *`nodelist`* in Express.

The feature of Express that has been extensively used in F90D/HPF RTS libraries is the support for a virtual grid topology of processors. Express provides a set of conversion routines between the physical processor numbers and their logical positions on the processor grid. This

Express	MPI
exsync()	MPI_Barrier(comm)
exbroadcast(buf, comm, buflen, nnodes, nodelist, tag)	MPI_Bcast(buf, buflen, datatype, root, comm)
excombine(buf, func, buflen, count, nnodes, nodelist, tag)	MPI_Allreduce(sendbuf, recvbuf, count, datatype, func, comm)

Table 4 9.One-to-one mapping of Express and MPI collective communication routines.

allows the construction of a virtual grid by merely specifying the number of dimensions in the grid and the number of physical processors on each dimension. Express implicitly creates a virtual grid to keep the mapping of actual physical processor numbers to the coordinates of the grid, and allows the performance of grid-based communication in a convenient manner.

4.4 Global Arrays Programming Model

Global Arrays (GA) [NHL 94, NieF 96, NieH 96, NHL 95, NLR 95] is a library-based shared-memory programming model developed at the Pacific Northwest Laboratory Environmental and Molecular Science Laboratory. The GA programming model acknowledges that the remote data access is slower than the local data access due to the NUMA properties of the underlying distributed-memory parallel architectures. It provides the programmer with explicit data distribution and transfer mechanisms for specifying and exploiting data locality.

The design and implementation of the GA library were motivated by the distinctive characteristics of computational chemistry applications. These applications manipulate very large matrices and require MIMD-style task parallelism in addition to data parallelism where each task accesses only small matrix sections in an unpredictable manner (shown in Figure 4-7).

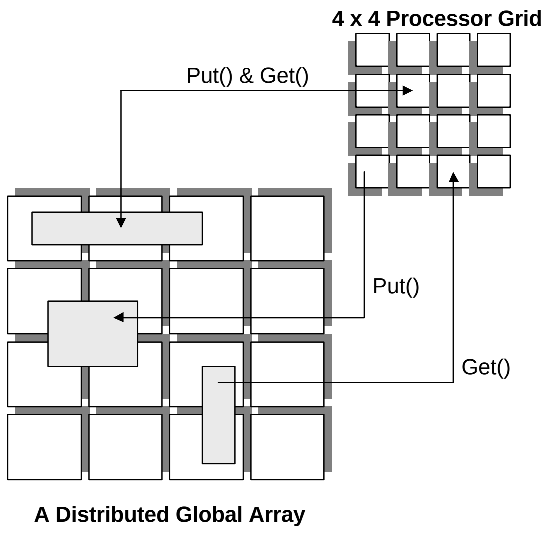

Figure 4 7. Asynchronous one-sided `get()` and `put()` accesses to arbitrary sections of a distributed GA matrix.

The task execution times vary greatly. The amount of computation required is generally on the order of O(N^3) to O(N^4), which makes the computation-to-data-movement ratio large.

The GA library was designed to complement rather than replace the message-passing programming model. The programmer is free to use both the shared-memory and the message-passing paradigms in the same program, and to take advantage of the existing message-passing library routines.

The GA library provides the user with the means to exploit data locality by specifying the distribution of data on distributed-memory platforms. Arrays can be distributed *block-wise* or in an *irregular* fashion on each dimension. All non-GA data items are replicated on each processor by default, which ensures MIMD parallelism. Each task can determine which portion of each distributed matrix is stored *locally* and can communicate with peers either by directly accessing

GA	`integer gl_A, lb1, ub1, lb2, ub2, locdim`
	`double  precision loc_A(1:locdim, *)`
	`call ga_create(MT_DBL, n, m, ‘A’, 10, 5, gl_A)`
	`call ga_zero(gl_A)`
	`call ga_put(gl_A, lb1, ub1, lb2, ub2, loc_A,locdim)`

HPF	`      integer lb1,ub1,lb2,ub2,locdim`
	`      double  precision A(n, m)`
	`      double precision loc_A(1:locdim, *)`
	`!HPF$ distribute A(block(10), block(5))`
	`      A = 0.0`
	`      A(lb1:ub1,lb2:ub2) = loc_A(1:ub1- lb1+1, 1:ub2-lb2+1)`

Table 4 10. Two code samples with similar functionalitywritten using GA library calls and HPF.

sections of distributed matrices or by using explicit asynchronous message-passing primitives without requiring the explicit cooperation of other tasks.

GA operations can be classified into two categories, primitive and composite, based on the complexity of the operation. *Primitive operations* are implementation-dependent and can be divided into two classes according to whether they should be executed synchronously or not. Cooperation of all processes is required to call the *synchronous primitive operations,* such as process synchronization, or array creation/destruction. On the other hand, any process may call *asynchronous primitive operations* independently. Non-atomic fetch and store (i.e., get and put), array gather and scatter, read-and-increment operation, inquiry operations, direct access to local array elements, and atomic accumulate operations on one- or two-dimensional arrays are some of the asynchronous primitive operations.

Composite operations are built on top of the primitive operations in an implementation-independent manner. They are optimized to reduce communication and data copying costs by directly accessing to local data. They are able to manipulate full arrays or selected array sections, and they include vector operations like *dot-product* or *scale* and matrix operations like matrix multiplication.

4.4.1 A Brief Comparison of HPF and GA Libraries

Most of GA's basic functionality (such as create, fetch, store, accumulate, gather, scatter, and the data parallel operations) can be expressed with array notation, data parallel statements, and data distribution directives of HPF. Yet GA provides random, independent access to distributed array regions from within an MIMD parallel subroutine call-tree and reduction into overlapping regions (patches, sections) of shared arrays that are not supported by HPF [NHL 94]. Furthermore, as opposed to HPF, arrays involved in matrix operations do not have to be conforming — it is enough if they have elements of the same type and number.

The GA and HPF code samples given in Table 4-10 are similar in functionality except for the following three differences: First, HPF uses a separate directive to distribute an array, while the GA `ga_create()` call both declares and specifies the distribution for the array. Second, GA uses library calls for performing array operations, while HPF uses Fortran statements. Third, HPF executes the assignment statements in a data-parallel fashion using the owner computes rule, whereas the corresponding GA put and get operations are executed in MIMD mode, i.e., each process references a different array section.

4.4.2 Porting Global Arrays onto the WWVM platform

This particular implementation of the GA library[6] [GA 2.0] uses the TCGMSG portable message-passing library on top of TCP/IP protocol. Except for a few, most functions of the GA were implemented in terms of TCGMSG calls. Therefore, it would suffice to implement

[6] Version 2.0.

TCGMSG	**MPI**
SND(tag, buf, buflen, dest, 1)	MPI_SSEND(buf, buflen, MPI_BYTE, dest, tag, comm, ierror)
RCV(tag, buf, buflen, lenmes, nodesel, src, 1)	MPI_RECV(buf, buflen, MPI_BYTE, src, tag, comm, status, ierror)
SND(tag, buf, buflen, dest, sync)	MPI_ISEND(buf, buflen, MPI_BYTE, src, tag, comm, request, ierror)
RCV(tag, buf, buflen, nodesel, src,0)	MPI_IRECV(buf, buflen, MPI_BYTE, src, tag, comm, request, ierror)
SND(tag | type, buf, buflen, dest, sync)	MPI_SEND(buf, buflen, MPI_BYTE, dest, tag, comm, ierror)
RCV(tag | type, buf, buflen, nodesel, src, sync)	MPI_RECV(buf, buflen, datatype, src, tag, comm, status, ierror)

Table 4 11. One-to-one mapping of TCGMSG point-to-point functions to MPI functions.

TCGMSG routines in MPI in order to support the GA programming model on top of the WWVM.

TCGMSG [Harr 91] (Theoretical Chemistry Group MeSsaGe-passing libraries) is a message-passing library widely used in the computational chemistry community.[7] Its programming model and interface is directly modeled after Argonne National Lab's PARMACS libraries [BBD+ 87]. TCGMSG primitives have slightly restricted semantics, but are highly efficient on both high-performance parallel computers and workstation clusters. According to one

[7] It was begun at Argonne National Laboratory and completed at Pacific Northwest Laboratory.

study presented by Douglas, et al. [DMS 93], TCGMSG is much more efficient than C-Linda, p4, and PVM.

Processes are connected with ordered, synchronous channels. On message-passing architectures, channels are set up using TCP sockets; shared-memory mechanisms are used on shared-memory machines. On a true message-passing machine, TCGMSG is just a thin layer over the system interface.

4.4.2.1 Point-to-Point Operations

TCGMSG supports both blocking and nonblocking send and receive operations. The last argument of send and receive primitives determines whether the primitive is blocking or not. The source and destination of sends and receives can be wildcarded, while tags must be explicitly specified. Mapping of TCGMSG point-to-point primitives to corresponding MPI primitives is given in Table 4-11.

TCGMSG	MPI
DGOP(tag, buf, buflen, size, func, datatype)	MPI_REDUCE(sendbuf, recvbuf, buflen, MPI_DOUBLE_PRECISION, func, root, comm, ierr)
IGOP(tag, –buf, buflen, size, func, datatype)	MPI_REDUCE(sendbuf, recvbuf, buflen, MPI_INTEGER, func, root, comm, ierr)
BRDCAST(tag, data, buflen, root)	MPI_BCAST(buf, buflen, datatype, root, comm, ierr)
SYNCH(tag)	MPI_BARRIER(comm)

Table 4 12. One-to-one mapping of TCGMSG collective operation functions to MPI functions.

TCGMSG	MPI
PBEGINF()	MPI_INIT(argc, argv)
PEND()	MPI_FINALIZE()
me = NODEID()	MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
np = NNODES()	MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
WAITCOM(src)	MPI_WAIT(request, status, ierror)
PROBE(tag, src)	MPI_IPROBE(src, tag, comm, flag, status)
PARERR(code, str)	MPI_ERROR_STRING(code, str, reslength, ierror)
TCGTIME()	MPI_WTIME()
MTIME()	MPI_WTIME()
EVON(), EVOFF()	MPI_PCONTROL(level)

Table 4-13. One-to-one mapping of remaining TCGMSG functions to MPI functions.

When two machines with different byte orderings or data representations need to communicate. TCGMSG routines take care of the required data translations using XDR. The data type is indicated by *or*ing the message tag field with MSGDBL, MSGINT, or MSGCHR, corresponding to double, integer, and character data.

4.4.2.2 Collective Communication Operations

TCGMSG includes collective computation for +, *, *max*, *min*, *absmax*, and *absmin* integer and double-precision operations. Except for *absmin* and *absmax*, MPI has counterparts of these operations (Table 4-12) and of others (Table 4-13).

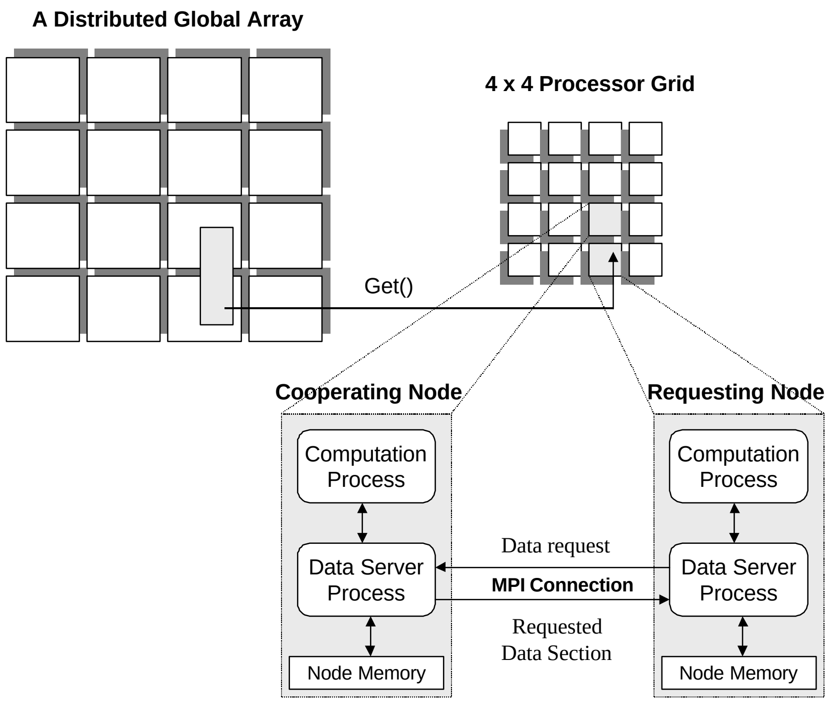

Figure 4 8. Order of operations during the interpretation of a remote get() operation on the WWVM architecture.

4.4.2.3 GA Data Server

The WWVM architecture is very similar to the cluster of workstations on a network. On each computation node of the WWVM a *data server* controls the local memory and responds to all requests coming from remote sites by sending requested data items, as shown Figure 4-8. Control of the local memory is achieved by using the *dynamic memory allocator* (DMA) [DMA], which is a portable library developed at Pacific Northwest Laboratory. It includes functions for heap and stack memory management, memory location debugging and verification support, memory usage statistics, and quantitative memory availability information. DMA is especially important for Fortran applications that do not support dynamic memory allocation as C does. Data server helps to emulate the one-sided communication and shared counter operation that are not supported by MPI.

4.4.2.4 One-Sided Communication

GA libraries provide several operations for one-sided access to distributed arrays, such as put, get, gather, scatter, atomic accumulate, and atomic read-and-increment. In the one-sided communication model, processes can access remote data asynchronously without the explicit cooperation of processes that own the data [NLR 95]. One-sided communication is crucial for programs with unpredictable remote data access patterns that cannot be detected by compiler analysis and that can be implemented by using active messages, hardware get and put, interrupt receive, shared memory, or threads on different computing platforms. It can be alternatively implemented using a *polling* mechanism, but this is not preferred because of the negative effects on the communication and computation performance.

MPI-2 is expected to support one-sided communication. A data server also helps in emulating the one-sided communication routines of the GA. A reference to a global array section is decomposed into local references on specific processors. Asynchronous one-sided operations like *get*, *put*, *store*, or *accumulate* cause the requesting process to send a single message containing the operation type, data size, and the data itself.

4.4.2.5 Emulating the TCGMSG NXTVAL() Routine

TCGMSG includes a function called `NXTVAL()` that simulates a built-in, shared-memory counter and returns the next value of this counter when it is called. This counter can be implemented using an interrupt handler on machines with interrupt-driven receives. On other machines a specific process numbered higher than any other application process is designated to provide this service.

The data server also helps in simulating a simple shared counter used for the dynamic distribution of loop iterations to processors. Although this approach is quite portable, it suffers from slower access to local data than the approach where the data resides directly in the application process. Furthermore, an additional layer is required on top of the message-passing libraries to hide the server processes from the application.

The implementation is as follows: Once all processes are registered, they synchronously call a counter initialization routine that separates the `MPI_COMM_WORLD` into two distinct groups. One of the groups keeps the single-server process, while the other group keeps the rest of the processes that will be used throughout the program. The server process receives waits for requests from ot6her processes and sends the current value of the counter by the `MPI_SEND()` operation. Alternatively, all processes may be responsible for the counter service. In this case, each process periodically probes the system to see if there is any pending request to the counter service and, if so, sends the new value of the counter to that process. The service requests are determined by using a a special tag, but wildcarded destination process, in the send operation.

4.5 Summary: The Big Picture

This chapter has provided the details of the implementation of a generic application programming layer on top of the WWVM that directly supports PVM but also provides high-level interfaces to other message-passing libraries such as MPI, Express, and TCGMSG. Other shared-memory programming models using HPF or Global Arrays libraries are supported on top of the message-passing communication layer.

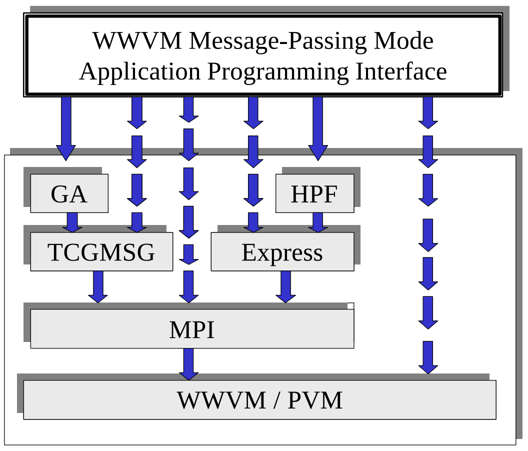

Figure 4 9. Shared-Memory models and message-passing libraries supported by WWVM.

Since there is a strong trend nowadays toward porting every type of HPCC application onto the MPI platform, making the MPI to PVM interface as flexible and dependable as possible has remained the main focus. The same interface could be used to port other applications that may become available in the future. Another concern has been to separate the actual message-passing implementation of the WWVM from its application-programming interface. If any MPI implementation with more focus on process management and dynamic process capabilities becomes available in the future, the message-passing communication layer of the WWVM can be replaced without affecting the application layer, by an MPI-based one by following the guidelines presented in this work.