Approaches to Support Parallel Programming on Workstation Clusters: A Survey

Bernd Freisleben and Thilo Kielmann

Department of Electrical Engineering and Computer Science (FB 12), University of Siegen
Hölderlinstr. 3, D–57068 Siegen, Germany
E-Mail: {freisleb,kielmann}@informatik.uni-siegen.de

Abstract

The goal of this report is to survey state of the art and existing approaches for parallel programming on workstation clusters with special emphasis on object–oriented programming. First, workstation clusters as parallel computing platforms are characterized and fundamental concepts for parallel programming are discussed. Then, an overview of existing tools, systems, languages, and environments is given. The report concludes by identifying features of software systems suitable for parallel object–oriented programming on top of workstation clusters.

1 Introduction

The rapid growth of interconnected high performance workstations has produced a new computing paradigm called clustered workstation computing. Its emergence has been fostered not only by the recent advances in computer hardware and networking technology, but also by the observation that the majority of the workstations included in today's networks may lie idle for as much as 95% of the time [95]. It is therefore not surprising that workstation clusters with high speed interconnections are among the candidate computer architectures for achieving the teracomputing goal required to solve grand challenge applications [15].

This report represents a survey of emerging software technologies and philosophies related to workstation clusters. Many stable software products are already available to assist with improving/exploiting the utilization of networked computer resources, both in environments where workstation clusters are simply used to replace traditional mainframe solutions and also where their parallel computing potential is desired to be exploited. In the absence of a physically shared memory, all packages are based on either one of the three primary communication paradigms for information exchange across networks: *message passing*, *remote procedure calls* and *distributed shared memory* [86, 104].

Several basic parallel programming paradigms can be considered for developing an application. The choice of paradigm is determined by the available computing resources and by the type of parallelism inherent in the problem. The commonly employed paradigms are *master-slave*, *data pipelining*, *SPMD*, and *task-parallel* computing [86].

The two general approaches to express these paradigms at the language level are (a) to extend existing sequential languages with parallel constructs to handle communications and synchronization, and (b) to define new parallel programming languages based on functional, object-oriented, or logical models. Different kinds of tools have been developed to support these approaches to facilitate parallel programming on workstation clusters, which can be grouped into the following major categories:

- *Message-passing systems* provide the capability for process communication in the form of message-passing library calls which are inserted into a sequential (usually C or Fortran) program to achieve the information exchange required for parallel computation.

- *Distributed shared memory systems* implement a shared-memory programming model by runtime and kernel-level emulation of physically shared memory in a distributed enviroment.
- *Parallel runtime systems* provide a parallel language compiler with an interface to the low-level facilities required to support interaction between concurrently executing program components.
- *Parallel numerical libraries* provide subroutines for scientific use, increasing the potential for widespread acceptance of workstation clusters in the natural sciences community.
- *Performance monitoring and debugging systems* aid the software developer with the required activities usually found during software development.
- *Parallel programming languages* address the characteristics of distributed and parallel applications and hide the operating system and hardware idiosyncrasies from the programmer. Many of the parallel languages are implemented as language extensions to existing languages, while others represent completely new developments.
- *Parallel programming environments* provide an integrated set of tools that support the programmer in the various stages of program development.

At present, the majority of parallel programming tools in each of these categories do in general not provide suitable abstractions and software engineering methods for structured application design, in contrast to sequential programming where object-oriented techniques are by now well established for designing and implementing large application systems. Therefore, in this report particular attention is devoted to describing parallel object-oriented approaches which still are not adequately pursued but steadily emerging. In this respect, the paper focusses on parallel class libraries for object-oriented languages, parallel object-oriented programming languages, and parallel object-oriented programming environments for managing the increased complexity of large-scale parallel applications.

The report is organized as follows. Section 2 presents an overview of several issues related to workstation clusters. Section 3 discusses fundamental parallel programming concepts relevant to cluster computing. In section 4 existing tools developed in the different categories are described. Section 5 discusses the object-oriented approaches for parallel programming. Section 6 concludes the report and outlines areas for future developments.

2 Workstation Clusters

In this section several issues related to workstation clusters are discussed: the motivations for using them, their goals and expectations, the required infrastructure for operating them, and how they are typically used.

2.1 Motivations

The utilization of workstation clusters can yield many potential benefits. Some of the motivations for harnessing the computing power of workstation clusters are [104]:

- the need to provide a significant number of cost effective CPU cycles to serve scientific applications with moderate computational requirements,
- the ever shrinking budgets which are being provided to meet the continuously growing demand for scientific computing resources,

- the desire to exploit the price/performance benefits of RISC processors,
- the need to provide single processor interactive FLOPS during prime hours and also provide abundant batch FLOPS during nonprime hours,
- the need to provide inexpensive configurations to develop and explore parallel applications,
- the desire to utilize abundant workstation cycles which have been estimated to be idle 95% of the time without additional financial investment,
- the desire to rely on commodity, consumer-based technology (e.g. workstations and networks) to minimize economic risk and provide an incremental growth path,
- the availability of large memories which are common on workstations,
- the desire to maintain the ease-to-use facilities which already exist in workstation environments,
- the realization that clusters offer the best features of sequential and parallel processing: speed of sequential processing and unlimited growth of computing power in parallel processing,
- the desire to exploit the benefits of heterogeneous computing environments and thereby produce the most efficient computing solutions.

Many of the items listed above require significant changes in the way computing resources are operated. Additionally, several of these items may still require significant efforts to reach mainstream acceptance.

2.2 Goals/Expectations

The deployment of workstation clusters has lead to the realization that several goals/expectations are inherently linked to their success. Among the goals/expectations of workstation clusters are:

- the provision of several types of transparancy (architecture/processor type transparency, network/communication transparency, parallelism transparency, fault transparency, task location transparency),
- the support for traditional high level languages to facilitate an easy-to-use, well-known and portable programming environment,
- the capability of delivering increased performance from parallelization and scalability,
- the property of not only accomodating heterogeneity but exploiting it.

Many of the goals/expectations listed above are years away. However, they provide a basis for insight into the environments which will form the backbone of scientific computing in the future.

2.3 Infrastructure Requirements

Workstation clusters provide unique infrastructure challenges. They may be geographically dispersed, have multiple owners, include multi-vendor hardware, and have various software differences (vendor enhanced operating systems for example). Some of the infrastructure requirements of workstation clusters are [104]:

- high bandwidth networks to support communication requirements (e. g. $100 - 800$ Mbits/sec per host),

- low-latency communication mechanisms (e.g. 100 – 500 microseconds between hosts),
- good scaling characteristics (e.g. 10 – 1000 hosts),
- support for high-bandwidth multicast communications,
- capability to automatically recover from network and node failures,
- standard low-level primitives (for communication, synchronization, and scheduling across architectures),
- heterogeneous high-level communication mechanisms that hide architecture, protocol, and system differences,
- real-time performance monitors,
- reliable production batch job schedulers,
- distributed and parallel application development tools,
- support for traditional high level languages for heterogeneous computing,
- applications which are capable of exploiting workstation clusters,
- new system administration tools to address system management issues for distributed and parallel computing resources,
- development of standards which protect software investments.

2.4 Types of Usage

Workstations clusters are typically used in two different ways: as a computational resource for batch or interactive processing replacing the traditional mainframe solutions, and as a geographically dispersed distributed system providing parallel computing power. These two approaches are described in the following.

2.4.1 Dedicated Workstation Clusters

The price/performance characteristics offered by workstations have enticed a significant number of institutions to consider alternatives to the traditional monolithic mainframe solution for interactive and batch jobs. "Dedicated" workstation clusters have been installed at several institutions as substitutes or replacements for traditional computing solutions. Clusters of workstations offer a cost-effective method of delivering both interactive and batch CPU cycles. Dedicated workstation clusters may be characterized as typically installed in 19 inch racks in a central computer room; they are also homogeneous configurations (within the cluster unit), managed by a single group which administers the cluster like a central mainframe, frequently interconnected by networks with better performance than ethernet (e.g. FDDI, IBM's SOCC, ATM etc.), and they are typically accessed only via a gateway or frontend system. The gateway system either attaches the user to an available single system or manages the queues which feed batch jobs to the cluster.

One of the obvious limitations of clusters is created by the relatively slow network interconnection hardware. The interface employed will depend on bandwidth requirements, latency requirements, distance limitations, and budget constraints. Ethernet is the most commonly implemented network and transmits information at 10 Mbits/sec, although first generation ethernet

cards operating at 100 Mbits/sec are close to being commercially available. Many dedicated clusters are interconnected by more expensive technologies to overcome the limitations induced by the speed of ethernet. The most common alternates to ethernet are FDDI, IMB's Serial Optical Channel Converter (SOCC) and ATM networks.

Dedicated clusters are commonly employed to increase overall batch computing capabilities. They achieve increased throughput by relying on parallelization at the single job level (e.g. individual jobs are submitted to a single workstation in the cluster). Most dedicated clusters have a "server" workstation which manages the job queue, provides extended disk space for I/O, and isolates users from the cluster (users login to the server to perfom interactive tasks, submit jobs, etc). Several software products have been developed to provide the job queuing functionality for clusters. One unique requirement for a cluster queuing system results from the use of workstation clusters as parallel processing environments. Only some of the systems support this capability. A second requirement is imposed by the need to automatically change the number of systems reserved for interactive and batch jobs based on work loads (typically, more systems are allocated for interactive use during the 8am to 5pm period).

2.4.2 Enterprise Workstation Clusters

One of the greatest attractions of workstation clusters is the ability to utilize idle resources distributed across an enterprise network. [95] reports that several studies have estimated that approximately 95% of workstation CPU cycles are unused. Enterprise clusters or decentralized clusters have the following characteristics: they are geographically distributed with multiple owners and consist of heterogeneous (multivendor) systems. They also have limited control of configuration, system, and cluster participation due to the distributed ownership. Most often, they are connected via ethernet and attempt to utilize "idle" cycles on existing systems.

The key requirement of enterprise clusters is software which includes the implementation of an "availability policy." An enterprise cluster is configured with workstations which are individually owned. Owners must have a motivation to participate in the cluster. This frequently means that the individual owners believe that they will receive more resources than they contribute. However, individual owners do not want to have their system saturated while they are trying to work. An availability policy allows the individual owner to define how their systems will participate in the resource pool and commonly includes that the system must be inactive for N minutes prior to receiving enterprise jobs, a keystroke by the owner suspends all enterprise jobs until the system remains idle for the specific period, the system must be able to define configuration parameters (e.g. hardware, software, OS release, etc.), and it must be able to dynamically remove or include their systems in the pool.

Two types of software systems exist for operating enterprise clusters. These are dynamic resource sharing and automatic resource sharing. Systems which support dynamic resource sharing have the capability to breakpoint a job (possibly when an owner becomes active) and actually move the job to another system in the pool. Dynamic resource sharing systems typically require the users to link their programs with a runtime library which supports this mechanism. Automatic resource sharing systems will dispatch jobs within the pool but once the job is initiated on a particular machine it is simply suspended when the owner becomes active. Typically, automatic resource sharing systems do not require any modifications to the user's software.

2.4.3 Batch/Interactive Job Distribution Systems

There are several batch/interactive job distribution software systems for exploiting the computational resources offered by dedicated and enterprise workstation clusters. Examples of such systems include *BALANS*, *CODINE*, *Condor*, *DQS*, *DNQS*, *LSF*, *NC Toolset*, *NQS*, *PBS*,

QBATCH, *QuaHog*, *Task Broker*, *UniJES*, and *UTOPIA*. Brief descriptions of these packages are given in [104].

3 Parallel Programming Concepts

Workstation clusters provide a unique flexibility to prototype and experiment with parallel programming. Previously, parallel programming required investments in computers designed and marketed for this specific purpose. Now any researcher with access to multiple workstations can develop and experiment with parallel programs; in [12] the capability to solve grand challenge problems using a workstation cluster has been demonstrated. In this section we present the issues associated with parallel programming on workstation clusters.

3.1 Information Exchange

Although different types of software systems have already been developed to take advantage of workstation clusters [2, 104], the exchange of information across networks is currently based on three primary communication paradigms: *message passing*, *remote procedure calls* and *distributed shared memory* [86].

3.1.1 Message Passing

Message passing provides a capability to explicitly communicate information among simultaneously executing components of an application. These components may be executing on the same system or on different systems. The messages are directed to a known set of the components, and these components may choose to expect to receive them. The transfer of a message may not have to result in the transfer of the execution thread, which gives direct rise to an ability to support parallel programming paradigms.

Sockets [31] are the simplest mechanism available for message passing and they provide the basic mechanism underlying many networked systems. They are a lower level interface to perform the operation, but may provide significant improvements in the efficiency of the message passing by minimizing software overhead at the expense of additional effort on the part of the programmer. They are available on a large number of systems but have significant limitations:

- very shallow queuing means that messages may be lost, particularly if there are hot spots in the communication patterns
- the lack of debugging tools can make development difficult
- the endpoints for the communications must be explicitly known by the program
- responsibility for all data format conversions rests with the programmer
- responsibility for creating parallelism rests with the programmer

In spite of all of this, there are significant applications that use sockets to obtain the highest possible performance from a networked system.

There are a number of different message passing systems available to choose from [104] but none of them currently have the support of a majority of either the computational community or the vendors. The packages differ in their functionality but they all rely on the basic capability to send and receive messages in an architecture independent manner and they elaborate on this functionality by providing variations on the sends (blocking, non-blocking, multicast, synchronous) and the receives (blocking, non-blocking, polling). More sophisticated constructs such as barriers, reduction operations, and process groups provide added value to a system, but the most common systems operate in very similar ways and are used for very similar things.

3.1.2 Remote Procedure Calls

Remote procedure calls allow the distribution of an application over a network and are therefore primarily intended for being used as a programming tool for client-server applications in distributed computing environments [91, 96]. Since with remote procedure calls a single thread of execution is passing across the network to another system, there is no inherent parallelism in their use; a threads package or particular system calls must be used in conjunction to provide parallelism.

Writing an RPC application requires the creation of an interface definition. The interface definition fully specifies the interaction between the client and the server by defining which procedures will be remote, the data types of all variables passed and whether the variables are input values or output values.

3.1.3 Distributed Shared Memory

Distributed shared memory [82, 102] is an abstraction for supporting the concept of real shared memory in a network environment without physically shared memory. In contrast to message passing and remote procedure calls, in a distributed shared memory system a process which stores a value does not need to know the existence or location of other processes which may fetch it. But distributed shared memory systems are difficult to implement because the inherent communication delays are significant and the shared memory paradigm requires all cooperating processes to have identical images of the shared address space.

3.2 Parallel Programming Paradigms

There are a number of basic parallel programming paradigms that can be considered when developing an application. The choice of paradigm is determined by the available computing resources and by the type of parallelism inherent in the problem. The computing resources define the granularity that may efficiently be supported on the system and may contribute to the need for explicitly performing load balancing within the application. The type of parallelism reflects the structure of either the application or the data and both types may exist in different parts of the same application.

Parallelism arising from the structure of the application is referred to as *functional parallelism*. In this situation, different parts of the system may be performing different functions concurrently, requiring different executables on the different systems. Functional parallelism is very commonly seen when using heterogeneous systems to take advantage of unique resources. Typically, this is exemplified in the application by a differentiation among the parts of the system, often with one or more components providing either services for or directions to the remaining components. The communications patterns are often unpredictable, with most of the communications occuring at the time of initiation.

Parallelism may also be found in the structure of the problem or data. This type of parallelism, called *problem-oriented* or *data parallelism*, can allow a number of systems to concurrently perform essentially identical work on parts of the problem or the data set. There may not be any differentiation between the components of the application running on different systems and they are all executing the same program. Any differences in the execution path arise from the nature of the problem or dataset. The communication patterns may be highly structured, often with a significant amount of communication occuring between the different components due to data exchange.

Several parallel programming paradigms have been developed for exploiting the different types of parallelism. These are presented in the following.

3.2.1 Master-Worker

The *master-worker* paradigm is the inverse of the *client-server* paradigm used in distributed systems, in that requests for the performance of work flow in the opposite direction, from a single process to a group of processes, which gives rise to the parallelism in the paradigm. A single process normally exerts control over the processing by determining the data layout and task distribution for the remaining processes.

Master-worker applications may either have static load balancing or dynamic load balancing. A statically load balanced application has an *a priori* division of the work determined by the master, which may result in a very low overhead for this process. This may also allow the master to participate in the computation after each of the workers has been allocated a fraction of the work. The allocation of work may occur a single time, with the master successively allocating pre-determined fractions.

A dynamically load balanced master-worker paradigm may be better suited to applications where:

- the number of tasks greatly exceeds the number of processors,
- the number of tasks may be unknown at the start,
- the number of tasks may vary from run to run,
- the time necessary to complete the tasks can vary.

The key feature of dynamic load balancing is its capability of adapting to changing conditions, either in the application or in the environment. This can be particularly important in applications such as database searches or pattern matching applications where the extent of the search may not be predictable or in environments in which the load or the availability for some of the systems may vary unpredictably. This is often the situation with heterogeneous systems and systems supporting other uses.

With a single master, this paradigm can show high degrees of scalability, although the scaling may not be linear. A set of masters, each controlling a different process group of workers, can further enhance the scalability, but bottlenecks may occur at the master(s) because of a message backlog and it may be aggravated by any additional processing the master may have to perform. The master-worker paradigm, which is based on centralized control of the computation, can simplify the creation of robust applications that are capable of surviving the loss of the workers or even the master.

3.2.2 Data Pipelining

Data pipelining is primarily a set of filters and is often used in data reduction or image processing systems. Generally, there is no direct interaction between the successive processes other than to pass along results. Typically, processing advances through different stages with differing numbers of processes. It is based on the consolidation and generation of processes through the use of parallel libraries that may occur at the different stages in the pipeline to take advantage of different computing resources at various stages or in response to varying loads. It may also be useful to start new processes as the need arises, through the arrival of new data or new processing requests, leading to multiple processes within a stage, each part of a different pipeline. The scalability of this paradigm is directly dependent on the ability to balance the load both within and across stages. Checkpointing can occur as part of the transition between stages and robustness can be enhanced by providing multiple independent paths across the stages.

3.2.3 Single Program Multiple Data

In the SPMD (Single Program Multiple Data) programming paradigm each of the participating processes is executing the same program, although the individual processes may be following a different execution path through the code.

The communication patterns for SPMD programs are usually highly structured and often extremely predictable, which can make the task of data layout more tractable for a homogeneous system. The data may initially be self-generated by each process or it may be sent from a process that has access to the data to the remaining processes as part of the initialization stage. All but the simplest SPMD programs will perform data exchange as the computation proceeds, usually with synchronization required among the processes at this stage.

SPMD calculations can be highly scalable if the structure of the data and communications permit. If the data layout is correct in terms of balancing the load among the processes, then all of them will arrive at the synchronization point simultaneously. This can be considerably complicated if the systems have different (and varying) loads, different capabilities, or the execution paths differ dramatically. This can result in lowered efficiency as faster processes wait for the slower ones to complete. Very often, the loss of a single process is enough to cause the calculation to reach a state of deadlock in which none of the processes can advance beyond a synchronization or data exchange step. The synchronization step can also provide a natural point at which to produce checkpoint files that contain the state of the calculation for restarting in case of a later failure.

3.2.4 Task Parallelism

Task-parallel computations extend the SPMD programming paradigm by allowing unrelated activities to take place concurrently. The need for task parallelism arises in time-dependent problems such as discrete-event simulation, in irregular problems such as sparse matrix problems, and in multidisciplinary simulations coupling multiple, possibly data-parallel, computations. Task-parallel programs may dynamically create multiple, potentially unrelated, threads of control. Communication and synchronization are between threads, rather than processors, and can occur asynchronously among any subset of threads and at any point in time.

4 Tools for Parallel Programming on Workstation Clusters

Two general approaches are used to implement parallelization over a workstation cluster. The first one is to extend existing sequential languages with parallel constructs to handle communications and synchronization, and the second one is to define new programming languages based on functional, object-oriented, or logical paradigms. For both general approaches, several tools have been developed to support parallel programming on workstation clusters. The main representatives of such software tools are described in this section.

4.1 Message-Passing Systems

Message-passing systems provide the capability for process communication in the form of message-passing library calls which are inserted into a sequential program to achieve the information exchange required for parallel computation. Nearly all message-passing systems described in this section deliver language bindings to C and Fortran.

4.1.1 Express

The Express/Cubix package [41] is a complete parallel programming support environment incorporating communication- and process-control functions and libraries, parallel network definition tools, and postmortem performance analysis utilities. Point-to-point, global broadcast, and shared-memory semaphore constructs are all available for communications, as well as functions allowing multiple programs running on a single node, and asynchronous "message-handler" capabilites, whereby the receipt of a particular message causes the execution of a specified function to deal with it. The network definition tools provide a graphic interface allowing the user the ability to cluster machines by architecture, control connectivity, and specify proportional computing capabilities.

4.1.2 MPI

MPI (Message Passing Interface) [76] is an effort by a group of vendors, computational scientists and computer scientists to consolidate the experiences gleaned from the use of message passing systems into a single standardized message passing system that will have the direct support of the computer vendors. Most of the message passing systems were developed by research groups to make it possible to do parallel heterogeneous computations and, consequently, they are usually not fully optimized for all of the hardware platforms that they execute on. In addition, the various message passing systems are incompatible, both in terms of source code and operation. Providing a single, widely accepted standard would greatly improve code portability.

MPI is intended to address only message passing and to build on the most useful concepts and constructs in the current message passing systems. These are primarily covered in the areas of point-to-point message passing, collective communications, process groups and topologies. It does not deal with I/O, process management or virtual shared memory. Commercial implementations of MPI are not yet available but a number of research groups are providing usable implementations of subsets of the MPI routines based on current message passing technologies. Examples of this include:

- **CHIMP/MPI**:
  CHIMP/MPI (Common High-Level Interface to Message Passing) [28] has been developed at Edinburgh Parallel Computing Centre for use in in-house parallel application work. It supports most of the MPI interface through an MPI compatibility library which sits on top of CHIMP.

- **LAM/MPI**:
  LAM/MPI (Local Area Multicomputer) [24] has been developed at the Ohio Supercomputer Center and provides a full implementation of the MPI standard for Fortran and C programs running on clusters of UNIX workstations. It also supports PVM compatibility, parallel I/O, and debugging and monitoring tools.

- **MPI Model Implementation**:
  The MPI Model Implementation [76, 53] was developed at Argonne National Laboratory to provide a full implementation of the MPI features, including both the Fortran and C bindings.

- **UNIFY**:
  UNIFY [88] is a subset of MPI that has been built on top of PVM. UNIFY is a dual-API (Application Program Interface) system. Users may write an application containing only MPI calls, or a mixture of MPI and PVM calls. The resulting executable will run in the PVM environment. This subset has been built to show the relative ease of implementation of MPI, and also to ease the migration of code from PVM to MPI.

4.1.3 NXLib

NXLib [101] is a package which provides Intel Paragon native message passing calls for use on a workstation cluster. It includes synchronous, asynchronous and interrupt driven Paragon communications and allows Paragon programs to be developed on a workstation cluster or programs previously written for the Paragon to be executed on a workstation cluster.

4.1.4 PVM

PVM (Parallel Virtual Machine) [103] was developed at Oak Ridge National Laboratory and is a message-passing software package which allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource. The individual computers may be shared- or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations, that may be interconnected by a variety of networks, such as ethernet, FDDI, or ATM. PVM support software executes on each machine in a user-configurable pool, and presents a unified, general, and powerful computational environment of concurrent applications. User programs written in C or Fortran are provided access to PVM through the use of calls to PVM library routines for functions such as process initiation, message transmission and reception, and synchronization via barriers or rendezvous. Users may optionally control the execution location of specific application components. The PVM system transparently handles message routing, data conversion for incompatible architectures, and other tasks that are necessary for operation in a heterogeneous, network environment.

There are several message-passing systems which run on top of PVM:

- **Chameleon**:
  Chameleon [52] was developed by Argonne National Laboratory and is a message-passing communications library for distributed memory parallel computers. Chameleon can be used on top of native (vendor) communications libraries, P4, PICL, or PVM and therefore supports many architectures, including networked workstations. It is designed to be portable, has a very low overhead and helps to standardize operations such as parallel program startup and group operations which differ from one system to another.

- **FortNet**:
  FortNet [1] is a message passing harness developed for use in applications. It is integrated with a number of other parallel tools including a graphical execution monitoring package and a parallel programming development tool.

- **PARMACS**:
  PARMACS (PARallel MACros) [60, 61] has been developed by the GMD Birlinghoven as a portable message passing system for Fortran and C programs running on a variety of machines.

- **PICL**:
  PICL (Portable Instrumented Communications Library) [47, 48] was developed at Oak Ridge National Laboratory. It is a subroutine library that implements a generic message-passing interface on a variety of multiprocessors. Programs written using PICL routines instead of the native commands for interprocessor communication are portable in the sense that they can be run on any machine on which the library has been implemented. Correct execution is also a function of the parameter values passed to the routines, but standard error trapping is used to inform the user when a parameter value is not legal on a particular machine. Programs written using PICL routines will also produce timestamped trace data on interprocessor communication, processor busy/idle times and simple user-defined events if a few additional statements are added to the source code.

- **ZIPCODE**:
  ZIPCODE [99] is a portable message passing system which includes a number of higher level features which are necessary for building libraries and large scale application software. These include process groups, communication contexts, collective operations, and virtual topologies.

4.1.5 P4

P4 (Portable Programs for Parallel Processors) was developed by Argonne National Laboratory [71] and is a library of macros and subroutines for programming a variety of parallel systems (including networked workstations). P4 supports both C and Fortran and includes monitors for shared memory models, message passing for the distributed memory model, and support for combining the two models. The goals of P4 are to provide a portable programming system, dependent only on a small number of computational models (shared memory, distributed memory, cluster), an efficient implementation on each architecture supported, shared-memory programming at the monitor level (not locks), message-passing at an efficient yet portable level, support for the cluster model of parallel programming, and a foundation for high-level programming systems and libraries.

4.1.6 SPPL

The Stuttgart Parallel Programming Library (SPPL) [68] allows to use a network of workstations as a parallel computer. SPPL offers functions to start processes on the workstations. Intermediate results can be exchanged through SPPL functions to send respectively to receive messages. Computers of different vendors use different internal representations for the same data. These representations are automatically transformed by SPPL as necessary using a description of the message. SPPL collects all data that should be sent in a message buffer and sends this buffer as a message. The receiver unpacks the data automatically.

4.1.7 TCGMSG

TCGMSG (Theoretical Chemistry Group Message Passing System) [2] is a toolkit for writing portable parallel programs using a message passing model; it includes both point-to-point communication and collective communication routines. Supported are a variety of common UNIX workstations, mini-super and supercomputers and heterogenous networks of the same, along with true parallel computers such as the Touchstone Delta, the Intel iPSC and the Alliant FX/2800 MPP system. Applications port among all of these environments without modification to the parallel constructs.

4.2 Distributed Shared Memory Systems

Many users find it easier to write parallel programs using a shared-memory programming model, while scalable parallel machines require that the physical memory be distributed. As a result, many modern parallel machines are built with distributed shared memory. Development is continuing on distributed memory message-passing machines, yet even for these there are efforts to implement a shared-memory programming model through runtime and kernel-level emulation of shared memory.

4.2.1 ADSMITH

ADSMITH [2] is a C package which provides a logical shared memory environment on a distributed memory machine. It sits on top of PVM.

4.2.2 ALMS

ALMS (Asynchronous Linked Memory System) [85] was developed at Brookhaven National Laboratory and is a toolkit for constructing parallel applications from independently coded modules. ALMS is a variation of the typical shared memory system because it only requires that all cooperating processes have an identical view of the sequence of successive values stored at a particular address during the computation. This implies that all processes may not see the same member of the sequence at the same time. ALMS is most suitable for asynchronous applications such as graphics/computation, least square minimization of a function over parameter space, and searches for minimum length paths on graphs.

4.2.3 DSM

DSM (Distributed Shared Memory) [104] was developed at Florida State University and is a communication system based on the distributed shared memory paradigm which allows parallel programs to "share" vectors of integers and reals in a simple and intuitive fashion. DSM uses UDP (universal datagram protocol) to provide broadcasting functionality.

4.3 DSM Compiler

Existing work in parallelizing compilers falls into two principal categories: compiling for uniform shared memory machines and compiling for distributed memory message passing machines. Little work has addressed compiler techniques for distributed shared memory machines. The DSM Compiler project [30] is currently conducted at the University of Rochester. Its aim is to develop a parallelizing compiler for distributed shared memory systems.

4.3.1 Linda

A particular type of distributed shared memory system based on the notion of *generative communication* is the Linda system [49, 23]. Linda was developed by Yale University, but there are several commercial implementations of Linda with Scientific Computing Associates being the most aggressive marketer of Linda products. Programming with this type of system is intended to be very simple and only requires the use of a small number of calls to put, examine or retrieve data in the distributed shared memory, referred to as a *tuple space*. There are no explicit calls to transfer data to different processors or other ways to pass information between specific machines. The operation *out* puts data into the tuple space and the data is then available through calls to *in* and *rd*, which read the data with *in* removing it from the space. New processes that are capable of accessing the tuple space are spawned with the *eval* operation, supporting multiple threads of execution. Linda relies on computations to reduce the need for communications and is intended to perform at a level comparable to explicit message passing systems.

Network Linda [7] has recently been released for use on top of PVM and P4 and supports Linda applications over a set of networked workstations. Piranha-Linda [74] is a research project which structures Linda modules as a "cloud of tasks" which computational "Piranhas" attack. The more Piranhas attack, the faster the job is completed.

4.3.2 Locust

Locust [29] is a distributed shared virtual memory system under development at the Experimental Systems Laboratory of the Computer Science Department at the State University of New York at Stony Brook. Locust has been designed for a network of workstations environment and a prototype is under development for a set of PC's connected by an Ethernet. The key idea

of Locust is the exploitation of data dependency information collected at compile time to hide message-latency and to reduce network traffic at run time.

Pupa [29] is the underlying communication subsystem of Locust. Pupa has been designed specially to support parallel computation in a LAN environment. It coexists with TCP/IP, and its implementation on a network of PC's connected by a 10 Mbits/sec Ethernet and running BSD has been claimed to be upto 6 times faster than the native TCP/IP implementation. A port to PCI based Pentium machines running BSD and connected by a 100 Mbits/sec Ethernet is planned, where the performance difference is expected to be more remarkable.

4.3.3 Mermera

Mermera [59, 58, 57] was developed at Boston University and is distributed shared memory system which gives the programmer the choice of coherent and non-coherent behavior in the same program. Mermera provides a wide variety of cleanly related memory operations satisfying different ordering constraints thereby providing an opportunity to trade-off programming simplicity for performance.

4.3.4 Mether

Mether [79, 78] was developed by the University of Delaware and provides a shared memory implementation over a group of Sun workstations connected via ethernet. Pages of memory in Mether migrate around the network in a demand-paged fashion and are accessed by the programmer in a fashion indistinguishable from other memory. Mether has been implemented as two separate components; a kernel driver which maintains a set of pages and their associated state and a user-level server driver which facilitates data transmission. The user-level server runs as an event-driven loop where events are UDP messages and Mether page-wanted/page-freed events.

4.3.5 Midway

Midway [18] is a software-based distributed shared memory system. Distributed shared memory provides the programmer with the illusion of a single virtual address space, which is shared among a network of processors that do not share physical memory. As local memory is updated, the modifications are propagated to the other processors, so that all maintain a consistent view.

Midway optimizes the propagation of updates by using a form of weak consistency called entry consistency. In Midway, there is an explicit binding of locks to the data that is logically guarded by each lock. As the application acquires a lock for its own synchronization, Midway piggybacks the memory updates on the lock acquisition message. Thus Midway sends no extra messages. Furthermore, the updates are sent only to the acquiring processor and only for the data explicitly guarded by the acquired lock. This serves to batch together updates and minimize the total amount of data transmitted.

Midway has other unique features besides the use of entry consistency. Midway detects updates to shared memory via compiler and runtime support, as opposed to more expensive virtual memory page protection. To provide high performance communication, Midway has its own application oriented protocols which reduce message counts, and it utilizes Mach's low-overhead network interfaces to reduce message latency. Midway currently runs on DECstations using Mach 3.0 connected via an ATM network or Ethernet.

4.3.6 Mirage

Mirage [40, 39] is a distributed shared memory system providing the illusion of a coarse grained multiprocessor to the programmer and algorithm designer. Mirage is designed to execute tra-

ditional System V Posix-compliant shared memory programs. A key feature in Mirage is the simplicity with which powerful networking can be employed using solutions that have been designed for uniprocessors and can be adapted to a more powerful network platform. Mirage implements DSM using a paged segmentation scheme where segments are partitioned into pages. The system runs on a cluster of IBM PS/2 class machines and is being ported to DEC Alpha AXP 150 personal computers.

4.3.7 SAM

SAM [94] was developed at Stanford University and is a run-time system that supports a shared name space in software on distributed-memory multiprocessors. There are a variety of scientific applications which operate on complicated data structures and access data in irregular and often data-dependent ways. SAM provides a global name space that facilitates programming these types of applications and caches data as necessary for efficient execution. All shared data in SAM is communicated in terms of user-defined data types, rather than fixed-size units such as pages. SAM provides simple primitives for accessing data from which more complex types of access can be built. SAM primitives can directly model the fundamental data relationships in parallel programs: producer-consumer, mutual exclusion, and chaotic relationships. SAM has been implemented on the Intel iPSC/860 and Paragon, the Thinking Machines CM-5, the IBM SP1, and on heterogeneous networks of workstations using PVM, and all SAM applications run portably on all these platforms.

4.3.8 Architectural Support for DSM Systems

Several projects have been initiated to provide low-level or architectural support for distributed shared memory computing on massively parallel systems and networks of workstations. Some of these projects are:

- the Wisconsin Wind Tunnel (WWT) project [64]
- the Berkeley NOW project [3]
- the Princeton SHRIMP project [20]
- the Minnesota DICE project [83]

4.4 Parallel Runtime Systems

A runtime system provides a parallel language compiler with an interface to the low-level facilities required to support interaction between concurrently executing program components. The runtime system defines the compiler's view of a parallel computer: how computational resources are allocated and controlled and how parallel components of a program interact, communicate and synchronize with one another.

Most existing runtime systems support the single-program, multiple-data (SPMD) programming model used to implement data-parallel languages. Since a compiler often has little global information about a task-parallel computation, efficient and portable runtime systems for task-parallel languages are harder to build.

In this section some of the approaches taken to develop runtime system for different parallel languages are presented.

4.4.1 Chant

Chant [54] is a threads package capable of supporting both point-to-point primitives and remote service requests (e.g., remote procedure call), using standard lightweight thread and communication libraries. Chant extends the POSIX pthreads interface to support two new global thread objects: a chanter, which is a so called talking thread, and a rope, which is a collection of chanter threads used for data parallel computations. All communication operations follow MPI syntax and semantics. Chant is currently being used to support the Opus language and runtime system, which extends High Performance Fortran (HPF) to support task parallel constructs.

4.4.2 Nexus

Nexus [45], currently developed at the Argonne National Laboratory, is a runtime system integrating lightweight threads and communication. Nexus supports multiple threads of control, dynamic processor acquisition, dynamic address space creation, a global memory model via interprocessor references, and asynchronous events. In addition, it supports heterogeneity at multiple levels, allowing a single computation to utilize different programming languages, executables, processors, and network protocols. Communication mechanisms that were considered in designing Nexus include message passing, shared memory, distributed shared memory, and message-driven computation. Nexus is intended primarily as a compiler target for languages supporting task-parallel and mixed data- and task-parallel execution; it is currently targetted by compilers for the parallel languages CC++ and Fortran M. It is operational on networks of Unix workstations communicating over TCP/IP networks, the IBM SP1, and the Intel Paragon using NX; it is being ported to other platforms and communication protocols.

4.4.3 PORTS

PORTS is a consortium of research universities, national laboratories, and computer vendors interested in developing a common runtime system to be used as a compiler target for various task and data parallel languages. Specific goals of the group are identifying opportunities for code sharing among projects, focusing on task and data parallel languages, and designing and implementing a common runtime system which supports task and data parallel languages and interoperability.

4.5 Parallel Numerical Libraries

The development of parallel numerical libraries projects will create a more widespread acceptance of workstation clusters in the natural sciences community, in order to promote the parallel implementation of scientific applications which can efficiently utilize distributed networked computing resources. This section overviews the major ongoing efforts directed at providing parallel scientific libraries, concentrating on two of the larger projects which make use of several low-level routines as building blocks.

4.5.1 The Multicomputer Toolbox

The Multicomputer Toolbox [98] is a project taking place at the Mississippi State University to provide scalable parallel libraries on a wide range of machines. The set of tools include a concurrent differential equation solver, subspace iterative methods, and sparse and dense LU solvers. These higher level libraries sit on top of lower-level packages, such as BLAS [36] and the ZIPCODE communication harness. Algorithms in the toolbox are formulated to be data distribution independent, allowing the higher level libraries to be flexible about the way the data

upon which they operate is distributed. This removes the need for applications to redistribute data prior to calling library routines.

4.5.2 ScaLAPACK

ScaLAPACK [35] is an ongoing project aimed at creating a parallel distributed memory version of the LAPACK software [36]. Part of the ScaLAPACK effort is involved with lower-level packages which provide building blocks for the higher-level numerical routines, but which are also useful packages in their own right, such as BLAS, BLACS and PB-BLAS [36]. Routines which are currently provided in ScaLAPACK include LU decomposition solvers, QR factorization solvers, Cholesky factorization solvers, and matrix reduction solvers.

4.6 Performance Monitoring and Debugging Systems

A critical component of developing distributed applications and particularly parallel applications is the availability of performance monitoring and debugging tools. Commercial vendors of products like Express and Linda provide excellent tools for aiding the software developer with these activities. However, there are also some specialized utilities which serve this requirement, as described in the following.

4.6.1 AIMS

AIMS [107] consists of a suite of software tools for measurement and analysis of performance. AIMS can be used to illustrate algorithm behavior, help analyze program execution and highlight problem areas that can then be modified to improve program execution. It includes a source-code instrumentor that supports Fortran77 and C message-passing programs written under PVM, a library of timestamping and trace-collection routines that run on Intel's iPSC/860 and Paragon, Thinking Machines' CM5, as well as networks of workstations (including Convex Cluster, SparcStations, and SGIs connected by a LAN), a trace post-processor that compensates for data collection overhead, and a trace-animation facility that supports simultaneous visualization of computation and communication patterns as well as analysis of data movements.

4.6.2 ParaGraph

ParaGraph [56] is a graphical display system developed at Oak Ridge Laboratory for visualizing the behavior and performance of parallel programs on message-passing parallel computers. It takes as input execution trace data provided by PICL or PVM, which optionally produce an execution trace during an actual run of a parallel program on a message-passing machine, and the resulting trace data can then be replayed pictorially with ParaGraph to display a dynamic, graphical depiction of the behavior of the parallel program. ParaGraph provides several distinct visual perspectives from which to view processor utilization, communication traffic, and other performance data in an attempt to gain insights that might be missed by any single view.

4.6.3 PVaniM

PVaniM [2] is an X-based tool which provides animated visualizations of PVM parallel programs. It includes a tracing package and a graphical viewer; it requires a C++ compiler and a machine running X-Windows.

4.6.4 TAPE/PVM

TAPE/PVM [2] is a tool to generate event traces for PVM applications, which can then be used for post-mortem performance analysis. Particular emphasis has been placed on the accuracy of the trace, and also on the minimization of the impact of the tool itself on the parallel application.

4.6.5 UPSHOT

UPSHOT [62] is an X-Based tool which graphically displays information about the execution of a parallel program, using data collected in a log file. It provides capabilities for analyzing the performance of parallel applications.

4.6.6 Xab

Xab [13], developed at Carnegie Mellon University, monitors a running PVM program, reporting on the status of message-passing, task-management, and information-gathering calls. This information can also be stored in a trace file, which can be converted to a format appropriate for ParaGraph. Xab consists of an instrumented version of the PVM library, a runtime "monitor" that catches PVM calls and writes out event records for each call, a graphical interface to view these event records, and a filter to convert the event records to PICL format.

4.6.7 XMPI

XMPI [11] is an X/Motif based graphical user interface for running and debugging MPI programs. It is implemented on top of LAM, an MPI cluster computing environment, but the interface is generally independent of LAM operating concepts. It allows the user to take a snapshot of the parallel MPI program any time during its execution. The tool provides information about the overall execution of the application, and the execution and communication status of each process.

4.6.8 XPVM

XPVM [11] provides a graphical interface to the PVM console commands and information, along with several animated views to monitor the execution of PVM programs. These views provide information about the interactions among tasks in a parallel PVM program, to assist in debugging and performance tuning. To analyze a program using XPVM, a user needs only compile his program using the PVM library, which has been instrumented to capture tracing information at run-time. Then, any task spawned from XPVM will return trace event information, for analysis in real time, or for post-mortem playback from saved trace files.

4.7 Parallel Programming Languages

Another approach to parallel computing using clusters is to implement programming languages which address the characteristics of distributed applications. Parallel programming languages offer several advantages over other approaches, such as hiding the operating system and hardware idiosyncracies to the programmer, portability, and optimization of operations. Some of the disadvantages of using special purpose programming languages for parallel applications are the cost of development, the possibly limited adoption of the language, and the requirement to learn a new language.

Many of the parallel languages are implemented as language extensions to existing languages, while others represent a completely new development. In this section, several parallel programming languages that can be used for cluster computing are described. Object-oriented

approaches are excluded from the discussion, since they are summarized in a dedicated section later.

4.7.1 DINO

DINO (Distributed Numerically Oriented Language) [92] was developed by the University of Colorado and is a C augmented language for writing parallel programs for distributed memory multiprocessors, including workstation clusters. DINO is directed at data parallel algorithms. The most significant characteristics of DINO are the ability to declare a virtual parallel computer that is best suited to parallel computation, the ability to map distributed data structures onto this virtual machine, and the ability to define procedures that will run on each processor of the virtual machine concurrently. Process management and interprocessor communications are handled automatically by the compiler.

4.7.2 Fortran M

Fortran M [42] was developed at Argonne National Laboratory and is a language for modular parallel programming. It provides extensions to Fortran 77 to support message passing and includes the following features for providing modularity (programs are constructed using explicitly-declared communication channels to plug together program modules called processes; a process can encapsulate common data, subprocesses, and internal communication), safety (operations on channels are restricted so as to guarantee deterministic execution), architecture independence (the mapping of processes to processors can be specified with respect to a virtual computer with size and shape different from that of the target computer) and efficiency (Fortran M can be compiled efficiently for uniprocessors, shared-memory computers, distributed-memory computers, and networks of workstations).

4.7.3 HPC

HPC (High Performance C) [105] provides data-parallel extensions to C. It is based on the High Performance Fortran (HPF) standard, but goes beyond HPF in a number of areas with features such as dynamically distributed data structures and dynamically allocatable irregular arrays.

4.7.4 HPF

HPF (High Performance Fortran) [63, 70] was developed by the High Performance Fortran Forum, a coalition of academic and industrial groups. The basic idea behind HPF is to provide a set of extensions to Fortran 90, allowing data-parallel programs to be written which achieve respectable performance on parallel machines. Several packages implementing some or all of the HPF standard have been developed. Examples which are available for PVM or MPI include ADAPTOR, HPF Mapper, PGI-HPF, PSI-Compiler, and xHPF [2, 66].

4.7.5 Jade

Jade [69, 90] is a parallel programming language (an extension to C) for distributed memory machines using SAM [94]. It exploits coarse-grain concurrency in sequential, imperative programs. Jade provides the convenience of a shared memory model by allowing any task to access shared objects transparently. Jade programmers augment their sequential programs with constructs that decompose the computation into tasks and declare how tasks access shared objects. The Jade implementation dynamically interprets this information to execute the program in parallel while preserving the sequential semantics – if there is a data dependence between tasks, tasks run in the same order as in the sequential execution. The structure of parallelism produced

by the Jade program is a directed acyclic graph of tasks, where the edges between tasks are the data dependence constraints. Because the constructs for declaring data accesses are executed dynamically, this task graph can be dynamic and can therefore express data-dependent concurrency available only at run-time.

4.7.6 Maisie

Maisie [104] is a C based parallel programming language. The primary enhancements to C are constructs to define, create and destroy processes, and send/receive messages. It works on PVM and UNIX sockets.

4.7.7 NESL

NESL [19] is a strongly-typed, functional, nested data-parallel language developed by the SCandAL project. It is intended to be used as a portable interface for programming a variety of parallel and vector supercomputers, and as a basis for teaching parallel algorithms. Parallelism is supplied through a simple set of data-parallel constructs based on sequences (ordered sets), including a mechanism for applying any function over the elements of a sequence in parallel and a set of parallel functions that manipulate sequences. The implementation is based on an intermediate language called VCODE and a low-level vector library called CVL. The NESL front end runs on a user's workstation, while allowing the remote execution of programs on parallel or vector supercomputers. The interactive environment includes a library of graphics routines, profiling and tracing facilities, and on-line documentation.

4.7.8 PARCS

PARCS (Parallel Constraint Logic Programming Language) [67] is a declarative parallel constraint logic programming language implemented for high performance execution on modern distributed memory parallel computers. PARCS exploits coarse grain parallelism that is derived from usual OR-parallelism and nondeterminism of instantiation of domain variables. Implementation techniques of PARCS for efficient execution on massively parallel processors include load balancing, a branching method and a compilation strategy based on extensive static analysis.

4.7.9 PCN

PCN (Program Composition Notation) [43] is a parallel programming system designed to improve the productivity of scientists and engineers using parallel computers. It provides a simple language for specifying concurrent algorithms, interfaces to Fortran and C, a portable toolkit that allows applications to be developed on a workstation or small parallel computer and run unchanged on supercomputers, and integrated debugging and performance analysis tools. PCN was developed at Argonne National Laboratory and the California Institute of Technology. It has been used to develop a wide variety of applications, in areas such as climate modeling, fluid dynamics, computational biology, chemistry, and circuit simulation.

4.7.10 Split-C

Split-C [32] is a parallel extension of the C programming language that supports efficient access to a global address space on current distributed memory multiprocessors. It retains the "small language" character of C and supports the engineering and optimization of programs by providing a simple, predictable cost model. This is in contrast to languages that rely on extensive program transformation at compile time to obtain performance on parallel machines. Split-C programs do what the programmer specifies; the compiler takes care of addressing and

communication, as well as code generation. Thus, the ability to exploit parallelism or locality is not limited by the compiler's recognition capability, nor is there need to second guess the compiler transformations while optimizing the program. The language provides a small set of global access primitives and simple parallel storage layout declarations. These seem to capture most of the useful elements of shared memory, message passing, and data parallel programming in a common, familiar context.

4.7.11 SR

SR (Synchronizing Resources) [4, 5] was developed at the University of Arizona and is a language for writing distributed and parallel programs. The language supports a wide variety of (reliable) message passing constructs, including shared variables (for processes on the same node), asynchronous message passing, rendezvous, remote procedure call, multicast.

4.8 Parallel Programming Environments

Parallel programming environments provide an integrated set of tools that support the programmer in the various stages of program development. Some of them are presented in the following. Again, object-oriented approaches are described later.

4.8.1 Enterprise

Enterprise [2] is a programming environment for designing, coding, debugging, testing, monitoring, profiling and executing programs in a distributed hardware environment. Enterprise code looks like familiar sequential C code since the parallelism is expressed graphically and is independent of the code. The system automatically inserts the code necessary to correctly handle communication and synchronization, allowing the rapid construction of distributed programs.

4.8.2 HeNCE

HeNCE (Heterogeneous Network Computing Environment) [10] is an X-window based software environment developed at Oak Ridge National Laboratory and designed to assist scientists in developing parallel programs that run on a network of computers. HeNCE provides the programmer with a high level abstraction for specifying parallelism. HeNCE is based on a parallel programming paradigm where an application program can be described by a graph. HeNCE graphs are variants of directed acyclic graphs, or DAGS. Nodes of the graph represent subroutines and the arcs represent data dependencies. Individual nodes are executed under PVM. HeNCE is composed of integrated graphical tools for creating, compiling, executing, and analyzing HeNCE programs. HeNCE relies on the PVM system for process initialization and communication. The HeNCE programmer, however, will never explicitly write PVM code. During or after execution, HeNCE displays an event-ordered animation of application execution, enabling the visualization of relative computational speeds, processor utilization, and load imbalances.

4.8.3 PACT

The Portable Application Code Toolkit (PACT) [22], developed at Lawrence Livermore National Laboratory, is a comprehensive, integrated, and portable software development environment created for applications having unique requirements not met with available software. By defining a single, higher level, standard programming interface, it shields application developers from the plethora of different hardware architectures and operating systems and their non-standard features. PACT is a set of libraries and utilities that can be integrated into existing software projects written in Fortran, C, C++, or Scheme. The set includes a low level, environment

balancing library, a math library, a process control library, a portable binary database management library, an interpreter for the Scheme dialect of the LISP language, a graphics library, a simulation code development system, a 1d data presentation, analysis, and manipulation tool, and Scheme with extensions.

4.8.4 TOPSYS

TOPSYS (Tools for Parallel Systems) [14, 16] is an integrated environment for developing applications for heterogeneous and distributed computing facilities. TOPSYS includes tools for parallel programming, monitoring, debugging, performance analysis, animation, and dynamic load balancing.

4.9 Other Issues

The focus of this paper was directed at surveying software products which could be employed to exploit the potential of workstation clusters. However, there are numerous ancillary topics which warrant discussion. This chapter includes brief overviews of several topics which are directly related to the main thrust of this report.

4.9.1 Collaboration Systems

Collaboration systems were identified in the original Metacenter proposal to the NSF [83] as one of the key technologies required to address grand challenge problems. The National Center for Supercomputer Applications (NCSA) responded to this requirement by establishing the DICE (Distributed Interactive Collaboration Environment) project to develop software tools which promote collaboration via networked computers. The first product (in Betarelease) released by the DICE project is a software system named Collage (COLLaborative Analysis and Graphics Environment). Collage has been characterized as a distributed WYSIWIS (what you see is what I see) environment. Collage allows multiple participants, connected via networked computers, to perform data analysis and interpretation as a collaborative endeavour. Collage supports real-time collaborative work across systems via a X-based environment, image display and analysis, color palette editing, spreadsheet display of floating point numbers, both text display and editing, a "whiteboard" for drawing, animation, and a HDF browser. All active participants may share common windows and communicate via dialog windows. The goal is to allow geographically distributed research collaborations using evolving high-speed networks and workstations.

4.9.2 Operating Systems

The operating systems used in mainstream computing environments are commonly referred to as "network operating systems". These operating systems do not provide networking functionality as an integral component of their design. Rather, networking primitives have been added to extend the environment so that a single computer can communicate with others. This method is not transparent due to the numerous differences between heterogeneous configurations. Distributed operating systems were designed to hide the machine characteristics and make collections of machines appear to be a single multi-user time-sharing system rather than a pool of autonomous systems. Distributed operating systems share a common kernel of each machine and handle resource management automatically. There are several research efforts presently directed at developing distributed operating systems: Amoeba [80], Charlotte [38], Chorus [93], Clouds [34].

Unfortunately, most of these operating systems do not provide the necessary support for efficient parallel program execution in an environment shared with sequential applications. In

order to realize this goal, the operating system must support gang-scheduling of parallel programs, identify idle resources in the network (CPU, disk capacity/bandwidth, memory capacity, network bandwidth), allow for process migration to support dynamic load balancing, and provide support for fast inter-process communication for both the operating system and user-level applications.

GLUnix [3] is an approach to provide functionality for parallel programming in networked workstations. It is built as a layer on top of existing operating systems. This strategy makes the system quickly portable, tracks vendor software upgrades, and reduces development time. This work seeks to design and implement a collection of low-latency, parallel communication paradigms. This includes extending current popular paradigms including sockets, remote procedure calls, and general messaging primitives on top of Active Messages. These communication layers will provide the missing link between the large existing base of parallel programs and fast but primitive communication layers.

SUNMOS/PUMA (Sandia/UNM Operating System) [72, 106] is a joint project between Sandia National Laboratories and the Computer Science Department at the University of New Mexico. The goal of the SUNMOS project is to develop a highly portable, yet efficient, operating system for massively parallel distributed memory systems. Where a decision between performance and added functionality must be made, SUNMOS usually favors performance. Most of SUNMOS is concerned with reliable and fast message passing. SUNMOS is a single-tasking kernel and does not provide demand paging. Once a SUNMOS application is loaded and running, it can manage practically all of the available memory on a node and use the full resources provided by the hardware. SUNMOS is intended to take over the compute nodes of a system. Applications are started and controlled from a host node process which runs on a SUN frontend for the nCUBE 2, and on a service node on the Intel Paragon.

5 Parallel Object-Oriented Approaches

The approaches described so far for parallel programming on workstation clusters do in general not provide suitable abstractions and software engineering methods for structured application design, in contrast to sequential programming where object-oriented techniques are by now well established for designing and implementing large application systems. The central notions exploited by object–oriented programming are objects, classes, and inheritance as means to structure applications and libraries of reusable software components. In the traditional object model, objects as the building blocks are defined as abstract data types which encapsulate their internal state through well-defined interfaces [77] and thus simply represent passive data containers.

In this section different approaches to enable parallel object-oriented programming on workstation clusters are presented.

5.1 Parallel Class Libraries

One possibility to utilize the potential of parallelism is to provide parallel runtime support to an existing sequential object-oriented language in the form of specialized class libraries. This approach is most promising in terms of allowing parallel programs to be written while keeping the associated overhead in reasonable bounds.

5.1.1 BEEBLEBROX

BEEBLEBROX [87] is a C++ class library for running divide-and-conquer type problems on a parallel machine. The parallel divide-and-conquer algorithms are encapsulated in the class

library, with the package user only needing to set up a class which specifies data flow between parent nodes and child nodes in the divide-and-conquer tree. It requires the LAM message passing harness to run.

5.1.2 Dome

The goal of the Dome (distributed object migration environment) project [6] conducted at Carnegie Mellon University is to facilitate the development of high performance computing applications running in parallel on networks of computers. Dome is intended to be used to develop grand challenge applications such as molecular dynamics, nuclear physics, distributed simulation, gene sequencing, and speech recognition. Dome makes it possible to take advantage of multicomputers (workstation networks), MPPs (massively parallel processors), advanced operating systems, and gigabit networks being developed under the HPCC program.

Dome provides this capability through libraries of distributed objects which can be used to program heterogeneous networks of computers as a single resource, obtaining performance that cannot be achieved on the individual machines. Dome addresses the problems of load balancing in a heterogeneous multiuser environment, ease of programming, and fault tolerance. Dome objects distribute themselves automatically over multiple computers. As a program runs, Dome will attempt to keep the workload balanced across the various machines. This is essential since the machines and networks have multiple users. Dome also supports checkpointing mechanisms for failure resilience in a heterogeneous environment, and adaptive collective communications for increased speed. It is a set of C++ libraries that run on various UNIX platforms using PVM and MPI.

5.1.3 DoPVM

DoPVM [46] is a shared object library and toolkit written in C++. It provides facilities for constructing objects which are shared across a distributed computing platform, and mechanisms for partitioning, scheduling and synchronization. It runs on top of PVM.

5.1.4 EPEE

EPEE [55] is an Eiffel class library containg data– and task-parallel programming models. It runs on TCP/IP on a network of UNIX workstations.

5.1.5 PETSc

PETSc (Portable Extensible Toolkit for Scientific computation) [75], developed at Argonne National Laboratory, is a library for portable, parallel (and serial) scientific computation that employs the MPI standard for message-passing communication. Most of the code is written in a data-structure-neutral manner to enable easy reuse and flexibility.

PETSc includes several components, all of which can be used in parallel from C, C++, and Fortran, such as a large suite of data structures and code for the manipulation of parallel sparse matrices, a collection of preconditioners, data-structure-neutral implementations of many iterative and non-linear numerical methods, and code for manipulating grids and discretizations in parallel. The PETSc components function similarly to C++ classes in terms of implementation and use. Each component manipulates a particular family of objects (for instance, vectors) through an abstract interface (simply a set of calling sequences) and one or more implementations using particular data structures.

5.1.6 POET

POET (Parallel Object-Oriented Environment and Toolkit) [73] employs an object-oriented frame-based approach to design and implement an object class library and the associated infrastructure of POET object code that can span various applications, in particular in physics and chemistry, on parallel platforms. The philosophy of the POET approach is to identify representations that solve entire classes of scientific problems. Thus, once the representation is defined for a particular problem area the mapping into the computing algorithm is handled automatically by the POET architecture. The approach does not attempt to dictate the language in which the numerical algorithm is written but embodies a set of algorithms. Each algorithm is implemented in an object contained within the system framework. PVM provides the baseline communication utilities for the system and can be replaced by any low level message passing utility by rewriting a communication object in the framework.

5.1.7 Presto

The PRESTO user-level threads library [17] provides an environment for writing object-oriented parallel programs in C++ for shared-memory multiprocessors running UNIX. The library provides basic classes useful for writing parallel programs, among them are thread manipulation routines for concurrency and synchronization primitives.

PRESTO's thread manipulation calls, such as thread creation, deletion, etc., are fairly standard, and are available in thread packages available on various machines. Programs are therefore easily portable to other architectures by replacing the PRESTO threads calls to those of another library.

5.1.8 ScaLAPACK++

ScaLAPACK++ [35] is an object-oriented extension for high performance linear algebra computations written jointly by Oak Ridge National Laboratory and the University of Tennessee. This version includes support for solving linear systems using LU, Cholesky, and QR matrix factorizations. ScaLAPACK++ supports various matrix classes for vectors, non-symmetric matrices, SPD matrices, symmetric matrices, banded, triangular, and tridiagonal matrices. Emphasis is given to routines for solving linear systems consisting of non-symmetric matrices, symmetric positive definite systems, and solving linear least-square systems. Support for eigenvalue problems and singular value decompositions are not included in the present release. Future versions of ScaLAPACK++ will support this as well as distributed matrix classes for parallel computer architectures.

5.2 Parallel Object-Oriented Programming Languages

In order to be able to explicitely express parallelism at the language level, several proposals have been made to introduce new parallel language constructs to object-oriented programming languages or to develop completely new parallel object-oriented languages from scratch. With this approach, programmers have full control over the parallelism offered by workstation clusters, but the approach is essentially equivalent to developing a new language and consequently leads to significant overheads for producing a new compiler. Selected parallel object-oriented programming languages are presented in the following.

5.2.1 CC++

CC++ (Compositional C++) [25, 97] from Caltech is a parallel programming language based on the C++ programming language. The CC++ system consists of the CC++ compiler and

a runtime environment. The runtime environment is called Nexus and provides support for implementing a wide range of task parallel programming languages. CC++ has three constructs for explicitly introducing parallel operations. Parallel computation in CC++ is expressed using parallel blocks, parallel loops, or unstructured parallel execution. For synchronization, CC++ provides synchronization variables and atomic functions. CC++ has an explicit locality model in which the programmer can distingish between objects that are inexpensive to access and objects that can be expensive to access.

CC++ has a library of common parallel programming paradigms and other utility routines that are of general use. The CC++ library contains routines for naming sets of nodes and using hostfiles, classes to simplify the porting of existing multi-threaded codes, atomic counters, using arrays of built-in types as arguments to functions called through global pointers, using the arguments to main when creating a new processor object, creating collections of processor objects spread across a set of nodes, and defining point-to-point communication channels. The library also contains an implementation of an SPMD programming environment that lets simple message passing code written using Intel's NX or IBM's MPL message passing library to execute in CC++ without change.

5.2.2 HPC++

The California Institute of Technology and Indiana University are collaborating to design and implement a parallel version of C++ in which programs that combine both task and data parallelism can be written. The motivation for this collaboration is the belief that programs that contain both task and data parallelism will become increasing important. The goal of the project is to produce a High Performance C++ (HPC++) [44], an object-oriented parallel programming language that will meet the needs of the next generation of parallel programs. Task parallelism in HPC++ is based on the CC++ programming language, and the data parallel part of HPC++ is derived from the pC++ language.

5.2.3 Oasis

Oasis (Object and Agent Specification and Implementation System) [26] was developed at the University of Michigan and is a distributed programming language which supports parallel computing. Computational processes are modeled as a distributed collection of autonomous cooperative agents. Oasis supports networked heterogeneous computers.

5.2.4 OOF

OOF (Object Oriented Fortran) [89, 100] was developed at Mississippi State University and is based on a minimum set of extensions to Fortran. OOF provides Fortran extensions to allow for object and operator declaration (including constructor and destructor functions) and invocation of operators. OOF does not implement common object oriented characteristics such as inheritance and strong typing. Parallelism in OOF is implicit since operators for different object instances are all independent and therefore may be operated on concurrently.

5.2.5 Orca

Orca [9, 8], developed at the University of Amsterdam, is a language for parallel programming on distributed systems, based on the shared data-object model. This model is a simple and portable form of object-based distributed shared memory. The language has been implemented (still in experimental form) on different platforms.

5.2.6 Parallel C++

The aim of the Parallel C++ project [37] funded by the European Commission ESPRIT Programme in a Working Group called EUROPA is to try to identify an emerging standard for a parallel version of C++, able to support real end user applications of high performance computing. The EUROPA Working Group is a collection of parallel C++ research groups in Europe, and computer hardware manufacturers. It is divided into 3 Special Interest Groups - Architecture SIG, Applications SIG, and Implementations SIG. Between these three groups, the aim is to identify emerging de facto standards for Parallel C++ compilers and ultimately produce a standard definition and library architecture for parallel C++ compilers.

5.2.7 pC++/Sage++

pC++ [21] is a portable parallel C++ for high performance computers. pC++ is a language extension to C++ that permits data-parallel style operations using "collections of objects" from some base "element" class. Member functions from this element class can be applied to the entire collection in parallel. This allows programmers to compose distributed data structures with parallel execution semantics. These distributed structures can be aligned and distributed over the memory hierarchy of the parallel machine much like HPF. pC++ also include a mechanism for encapsulating SPMD style computation in a thread-based computing model.

pC++ code is portable: just like Cfront translates C++ code into standard ANSI C that can be passed to the native C compiler, the pC++ preprocessor (written using Sage++) translates pC++ into C++, which is then compiled on the target architecture. Currently, pC++ has runtime systems for a variety of parallel machines, including networked workstation using PVM.

pC++ comes with a prototype version of the TAU tools (Tuning and Analysis Utilities), a visual programming and performance analysis environment for pC++. It is based on Sage++, which is an object-oriented compiler preprocessor toolkit. Using Sage++ to transform source code is a three step process: parsing source code written in FORTRAN, pC++, or C and saving it as a machine-independent parse tree, restructuring the parse tree to identify function-call sites, add variables, optimize loops, add tracing functions, or globally replace variables, and unparsing to recreate the source code and reflecting the changes that were made to the parsed form. After unparsing, the source code can be compiled with the native compiler.

5.2.8 pSather

pSather (Parallel Sather) [81] is a parallel version of the language Sather, developed and in use at ICSI Berkeley. Sather has parameterized classes, object-oriented dispatch, statically-checked strong (contravariant) typing, separate implementation and type inheritance, multiple inheritance, garbage collection, iteration abstraction, higher-order routines and iters, exception handling, assertions, preconditions, postconditions, and class invariants. Sather programs can be compiled into portable C code and can efficiently link with C object files. pSather inherits these properties. It addresses non-uniform-memory-access multiprocessor architectures but presents a shared memory model to the programmer. It extends serial Sather with threads, synchronization and data distribution. Unlike actor languages, multiple threads can execute in one object. A distinguished class GATE combines various dependent low-level synchronization mechanisms efficiently: locks, futures, and conditions. The pSather compiler is integrated into the serial Sather compiler and runs on various UNIX platforms.

5.2.9 UC++

UC++ [84] aims to provide minimal extensions to C++ to support explicit coarse-grain parallelism on a variety of distributed UNIX workstations. UC++ uses an allocation strategy to

express parallelism. This is done by the keyword `activenew` which is used in the same way as the standard C++ keyword `new` except that it creates an active object. Active objects are the units of parallelism in UC++.

The second keyword introduced by UC++ is `on`. This is used to place an active object on a specific processor memory pair in the system, rather than using the run time library's own allocation strategy. Parallel execution is initiated by means of the asyncronous function call operator which will send a function call to a remote active object and not wait for a return value. Functions can also be called synchronously as in standard C++. Further work is underway with UC++ to investigate fine grained parallelism and data parallel classes.

5.3 Parallel Object-Oriented Programming Environments

Parallel object-oriented programming environments provide an integrated set of tools for managing the increased complexity of large-scale parallel applications. Projects conducted in that direction are described in this section.

5.3.1 Castle

Castle [33] is currently under development at the UC Berkeley. The aim is to provide a parallel programming environment containing a number of facilities at different layers of abstraction: (1) an abstract computer incorporating active messages, (2) a physical multiprocessor containing shared memory and communication facilities, (3) a virtual multiprocessor containing distributed object and math libraries, (4) high-level languages including parallel C++ and HPF-like languages, and (5) applications, currently mainly taken from physics. Debugging, performance and optimizing tools can access all levels.

5.3.2 CHARM

CHARM [65] is a machine independent parallel programming system currently developed at the University of Illinois. Programs written using this system will run unchanged on MIMD machines with or without a shared memory. CHARM provides high-level mechanisms and strategies to facilitate the task of developing even highly complex parallel applications. Charm programs are written in C with a few syntactic extensions. It is possible to interface to other languages such as FORTRAN using the foreign language interface that C provides.

Charm++ is the C++-based parallel object oriented language having all features of Charm, which supports multiple inheritance, late bindings, and polymorphism. Charm features include effficent portability, latency tolerance, and dynamic load balancing. The system provides support for both regular and irregular computations.

Programs consist of potentially medium-grained processes (called chares), and a special type of replicated process. These processes interact with each other via messages and any of the other six information-sharing abstractions provided, each of which may be implemented differently and efficiently on different machines. The "replicated processes" can also be used for implementing novel information sharing abstractions, distributed data structures, and intermodule interfaces. The system can be considered a concurrent object-oriented system with a clear separation between sequential and parallel objects.

The modularity-related features make the system very attractive for building library modules that are highly reusable because they can be used in a variety of data-distributions. CHARM supports this with a "module" construct and associated mechanisms. These mechanisms allow for compositionality of modules without sacrificing the latency-tolerance. With them, two modules may exchange data in a distributed fashion.

For regular computations, the system is useful because it provides portability, static load balancing, and latency tolerance via message driven execution, and facilitates construction and flexible reuse of libraries. The system is unique for the extensive support it provides for highly irregular computations. This includes management of many medium-grained processes, support for prioritization, dynamic load balancing strategies, handling of dynamic data-structures such as lists and graphs, etc. The specific information sharing modes are especially useful for such computations.

Associated tools are: DagTool (allows specification of dependences between messages and sub-computations within a single process, provides a pictorial view), and Projections (a performance visualization and feedback tool); further tools are planned.

5.3.3 Concert

The goal of the Concert project [27] is to develop portable, efficient implementations of concurrent object-oriented languages on workstation clusters and other parallel machines. The Concert system consists of a high performance compiler and runtime (language implementations) as well as a complete set of tools (emulator, debugger, and performance tuning). The languages supported by Concert are Concurrent Aggregates and Illinois Concert C++ (ICC++).

The Concert programming model provides a globally shared namespace, object-based concurrency control and encapsulation, an integrated model of task and data parallelism, and a dynamic concurrency model. This model supports the easy use of distributed data structures, and expression of irregular parallelism. Because the runtime primitives have been highly tuned, exploitation of a relatively fine-grained concurrency (10–100 instructions) can be executed efficiently. Programmers can guide this exploitation using pragmas for locality and concurrency control.

5.3.4 Mentat

Mentat [51, 50] was developed by the University of Virginia as an object-oriented parallel processing system designed to directly address the difficulty of developing architecture-independent parallel programs. The fundamental objectives of Mentat are to provide easy-to-use parallelism, achieve high performance via parallel execution, and facilitate the execution of applications across a wide range of platforms. The Mentat approach exploits the object-oriented paradigm to provide high-level abstractions that mask the complex aspects of parallel programming, including communication, synchronization, and scheduling, from the programmer. Instead of managing these details, the programmer concentrates on the application. The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution.

Mentat combines a medium-grain, data-driven computation model with the object-oriented programming paradigm and provides automatic detection and management of data dependencies. The data-driven computation model supports high degrees of parallelism and a simple decentralized control, while the use of the object-oriented paradigm permits the hiding of much of the parallel environment from the programmer. Because Mentat uses a data-driven computation model, it is particularly well-suited for message passing, non-shared memory architectures.

There are two primary aspects of Mentat: the Mentat Programming Language (MPL) and the Mentat run-time system. The MPL is an object-oriented programming language based on C++ that masks the difficulty of the parallel environment from the programmer. The granule of computation is the Mentat class member function. The programmer is responsible for identifying those object classes whose member functions are of sufficient computational complexity to allow efficient parallel execution. Instances of Mentat classes are used exactly like C++ classes, freeing the programmer to concentrate on the algorithm, not on managing the

environment. The data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention. By splitting the responsibility between the compiler and the programmer, the strengths of each are exploited and the weaknesses of each are avoided. The underlying assumption is that the programmer can make better decisions regarding granularity and partitioning, while the compiler can better manage synchronization. This simplifies the task of writing parallel programs, making parallel architectures more accessible to non-computer scientist researchers.

The run-time system supports parallel object-oriented computing on top of a data-driven, message-passing model. It supports more than just method invocation by remote procedure call (RPC). Instead the run-time system supports a graph-based, data-driven computation model in which the invoker of an object member function need not wait for the result of the computation, or for that matter, ever receive a copy of the result. The run-time system constructs program graphs and manages communication and synchronization. Furthermore, the run-time system is portable across a wide variety of MIMD architectures and runs on top of the existing host operating system. The underlying operating system must provide process support and some form of inter-process communication. Mentat runs a variety of systems and has been successfully applied to a wide-variety of real-world problems.

6 Conclusions

In the near future, workstation clusters will play an important role in finding suitable platforms for parallel processing, mainly because of their attractive price/performance ratio and the increasing availability of high–performance networking infrastructures. The object–oriented programming paradigm is by now well established as the state–of–the–art software engineering methodology for sequential programming, and recent developments are emerging to establish object–orientation also in the area of parallel programming. The main concern of this survey was to outline the importance of workstation clusters and object-oriented technologies for future parallel processing. The report concludes by extracting features of software systems suitable for object–oriented parallel programming on top of workstation clusters.

The primary decision in finding the right approach for appropriate parallel tools is between parallel programming languages, class libraries, and programming environments. Clearly, new languages designed for specific purposes of parallelism bear the greatest potential for the provision of suitable and powerful programming abstractions. However, until now such languages have not found widespread use. This is primarily due to two reasons. First, new languages must be supported by appropriate compilers and runtime libraries. Their development requires significant efforts which to some extend prohibits the availability of new languages on diverse computing platforms. The second and most significant reason is the lacking user acceptance. New programming languages force programmers to learn new concepts which is neither popular amongst programmers nor encouraged by company management in order to protect investments in already existing programming infrastructures.

Similar objectives hold against integrated environments for parallel programming. Although they try to support programmers by collections of related tools like editors, browsers, debuggers, etc., they are even harder to implement for different platforms and force programmers to leave their used–to tools. This makes the acceptance problem even harder.

What remains is the approach which supports parallel programming by runtime libraries, or in the object–oriented context: by class libraries. With such libraries, existing monetary and intellectual investments in programming languages, compilers, and other tools can still be used. Hence, library approaches are widely accepted amongst programmers and company management. The great success of the PVM [46] and MPI [53, 76] libraries in traditional (non–object oriented)

programming supports this observation.

After having identified the library–based approach as the most suitable one, a set of abstractions to be provided by such a library must be determined. Since the goal is to provide abstractions for general–purpose parallel programming, it is not sufficient to simply identify structures useful for given problem domains, as most of the libraries surveyed in the previous section do. Instead, it is necessary to provide general abstractions on top of which solutions for specific application domains can be implemented. As shown above, both obvious programming models for distributed–memory programming, namely message passing and remote procedure call, are not well suited for efficient programming and execution of parallel applications. Instead, parallel programming on the basis of logical shared memory model provides abstractions very close to sequential programming and can make use of well–known synchronization concepts. Hence, a distributed shared memory model should be an integral part of a class library for parallel programming of distributed memory architectures like workstation clusters.

In addition to a shared memory model, a class library should provide means to deal with the concurrently executing activities themselves, namely with threads of control. The integration of threads, a shared memory model, and synchronization constructs for the threads being defined over operations on the shared memory provide a generally useful and expressive model for the implementation of parallel applications in different problem domains. The realization of control threads inside a class library furthermore enables the realization of mechanisms for load distribution and load balancing, two issues especially important for parallel processing on workstation clusters in which each processor's workload is dominated by processes from outside the parallel computation, originated from other users.

To conclude, we have identified the class library approach as the most suitable one for parallel object-oriented programming on workstation clusters. Unfortunately, there is no currently existing class library which implements the set of programming abstractions which we identified as suitable for distributed memory architectures. Therefore, it is subject to future projects to develop such a class library.

References

[1] R. J. Allan. Fortnet V4.0: The Parallel Programming Software. Technical Report, Daresbury Laboratory, 1992.

[2] R.J. Allan and P. Lockey. Survey of Parallel Software Packages of Potential Interest in Scientific Applications. Technical Report, Daresbury Laboratory, Daresbury, Warrington, UK, 1995.

[3] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW (Networks of Workstations). Technical Report, University of Berkeley, 1994.

[4] G. R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.

[5] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings, 1993.

[6] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel Programming in a Heterogeneous Multi-User Environment. Technical Report, Carnegie Mellon University, 1995.

[7] M. Arango, D. Berndt, N. Carriero, and D. Gilmore. Adventures with Network Linda. *Supercomputer Review*, 10(3):42–46, 1990.

[8] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with distributed programming in Orca. In *IEEE CS International Conference on Computer Languages*, pages 79–89, March 1990.

[9] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[10] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and R. Wade. HeNCE: A user's guide (Draft). Oak Ridge National Laboratory, November 1991.

[11] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing 1991*, pages 435–444, Albuquerque, 1991.

[12] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. S. Sunderam. Solving computational grand challenges using a network of heterogeneous supercomputers. In D. Sorensen, editor, *Proceedings of Fifth SIAM Conference on Parallel Processing*, Philadelphia, PA, 1991. SIAM.

[13] Adam Louis Beguelin. Xab: a tool for monitoring PVM programs. School of Computer Science, Carnegie Mellon University, June 5, 1992.

[14] H. Beier, T. Bemmerl, A. Bode, et. al. TOPSYS - tools for parallel systems. Technical Report SFB-Bericht 342/9/90 A, Technische Universität München, Munich, Germany, January 1990.

[15] G. Bell. Ultracomputers: A Teraflop Before its Time. *Communications of the ACM*, 35(8):27–47, 1992.

[16] Thomas Bemmerl and Bernhard Ries. Programming tools for distributed multiprocessor computing environments. In *Proceedings of the Heterogeneous Network-Based Concurrent Computing Workshop*, Tallahassee, FL, October 1991. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.91.

[17] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience*, 18(8):713–732, 1988.

[18] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[19] G. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[20] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 142–153, April 1994.

[21] F. Bodin, P. Beckman, D. B. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language. In *Proc. Supercomputing '91*, pages 273–282, 1991.

[22] Dennis Braddy and Stewart A. Brown. PACT User's Guide. Technical Report UCRL-MA-112087, Lawrence Livermore National Laboratory, 1995.

[23] Nicholas Carriero, David Gelernter, Timothy G. Mattson, and Andrew H. Sherman. The Linda alternative to message–passing systems. *Parallel Computing*, 20(4):633–655, 1994.

[24] Ohio Supercomputer Center. *LAM for C Programmers.* Ohio Supercomputer Center, 1994.

[25] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, 1993.

[26] F.-C. Cheong. *OASIS: An Agent–Oriented Programming Language for Heterogeneous Distributed Environments.* PhD dissertation, University of Michigan, School of Computer Science and Engineering, 1992.

[27] A. A. Chien and V. Karamcheti. Concert – Efficient Runtime Support for Concurrent Object–Oriented Programming Languages on Stock Hardware. In *Proceedings of SUPERCOMPUTING*, 1993.

[28] Chimp. CHIMP version 1.0 interface. Edinburgh Parallel Computing Center, The University of Edinburgh, UK, May 1992.

[29] T.-C. Chiueh and M. Verma. A Compiler-Directed Distributed Shared Memory System. In *Proc. of the 9th ACM Int'l Conf. on Supercomputing*, July 1995.

[30] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation.* ACM, 1995.

[31] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP*, volume Volume III: Client–Server Programming and Applications. Prentice Hall, 1993.

[32] D. E. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split–C. In *Proceedings of SUPERCOMPUTING*, 1993.

[33] D. E. Culler and K. Yelick. CASTLE: Practical Software Support for Parallel Computing. Technical Report, University of Berkeley, 1995.

[34] Partha Dasgupta, Richard LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24(11):34–44, November 1991.

[35] J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK++: An object-oriented linear algebra library for scalable systems. In *Proc. Scalable Parallel Libraries Conf.*, pages 216–223. IEEE Computer Society, 1993.

[36] J. Dongarra and D. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 1995. to appear.

[37] ESPRIT Working Group EUROPA. Parallel C++. http://www.lpac.ac.uk/europa, 1995.

[38] R. Finkel and et al. M. Scott. Experience with Charlotte: simplicity and function in a distributed operating system. *IEEE Transactions on Software Engineering*, 15:676–686, June 1989.

[39] B. D. Fleisch, R. L. Hyde, and N. C. Juul. MIRAGE+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers. *Software—Practice and Experience*, 24(8), August 1994.

[40] B. D. Fleisch and G. J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proc. of the 12th ACM Symp. on Operating Systems Principles (SOSP'89)*, pages 211–223, December 1989.

[41] J. Flower, A. Kolawa, and S. Bharadwaj. The Express way to distributed processing. *Supercomputing Review*, pages 54–55, May 1991.

[42] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *J. Parallel and Distributed Computing*, 25(1), 1995.

[43] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):51–66, 1992.

[44] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[45] Ian Foster, Carl Kesselman, Robert Olson, and Steve Tuecke. Nexus: An Interoperability Toolkit for Parallel and Distributed Computer Systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1994.

[46] G. Geist, A. Beguelin, J. Dongarra, W. Cheng, R. Manchek, and V. Sunderam. *Parallel Virtual Machine: A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, 1995.

[47] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communications library. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1990.

[48] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A user's guide to PICL: a portable instumented communications library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, Tennessee, January 1992.

[49] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[50] A. S. Grimshaw, W. T. Strayer, and P. Narayan. Dynamic Object-Oriented Parallel Processing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 5:33–47, 1993.

[51] Andrew S. Grimshaw. Easy–to–Use Object–Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(5):39 – 51, 1993.

[52] Bill Gropp and Barry Smith. Chameleon parallel programming tools user's manual. Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[53] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1995.

[54] M. Haines and W. Bohm. Task Management, Virtual Shared Memory, and Multithreading in a Distributed Memory Implementation of Sisal. In *Proc. of Parallel Architectures and Languages Europe (PARLE'93)*, pages 12–23, June 1993.

[55] F. Hamelin, J. M. Jézéquel, and T. Priol. A Multi–Paradigm Object–Oriented Parallel Environment. In *Proceedings of IPPS'94*, Cancun, Mexico, 1994.

[56] M. Heath. Recent developments and case studies in performance visualization using ParaGraph. In *Performance Measurement and Visualization of Parallel Systems*, pages 175–200. Elsevier Science Publishers, 1993.

[57] A. Heddaya, K. Park, and H. Sinha. Using Warp to Control Network Contention in Mermera. In *Proc. of the 27th Hawaii Int'l Conf. on System Sciences (HICSS-27)*, January 1994.

[58] A. Heddaya and H. Sinha. An Implementation of Mermera: A Shared Memory System that Mixes Coherence with Non-coherence. Technical Report BU-CS-93-006, Computer Science Department, Boston University, June 1993.

[59] A. Heddaya and H. Sinha. An Overview of Mermera: A System and Formalism for Non-coherent Distributed Parallel Memory. In *Proc. of the 26th Hawaii Int'l Conf. on System Sciences (HICSS-26)*, pages 164–173, January 1993.

[60] R. Hempel. The ANL/GMD macros (PARMACS) in Fortran for portable parallel programming using the message passing programming model – users' guide and reference manual. Technical Report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, 1991.

[61] R. Hempel, H.-C. Hoppe, and A. Supalov. PARMACS 6.0 library interface specification. Technical Report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, 1992.

[62] V. Herrarte and E. Lusk. Studying parallel program behavior with `upshot`. Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1991.

[63] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, Tex., 1993.

[64] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Trans. on Computer Systems*, 11(4):300–318, November 1993.

[65] L. V. Kale. A Tutorial Introduction to Charm. Parallel Programming Laboratory Report 92–6, University of Illinois, 1992.

[66] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[67] K. Konno, M. Nagatsuka, N. Kobayashi, S. Matsuoka, and A. Yonezawa. PARCS: An MPP-Oriented CLP Language. In *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO'94)*, pages 254–263, Linz, Austria, 1994. World Scientific.

[68] Stuttgart Parallel Programming Laboratory. http://www.informatik.uni-stuttgart.de/ipvr/as/grids/sppl/sppl-e.html, 1995.

[69] Monica S. Lam. Jade: a coarse-grain parallel programming language. In *Proceedings of the Heterogeneous Network-Based Concurrent Computing Workshop*, Tallahassee, FL, October 1991. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.91.

[70] D. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, 1993.

[71] Rusty Lusk and Ralph Butler. Portable parallel programming with p4. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.92.

[72] Arthur B. Maccabe, Kevin S. McCurley, Rolf Riesen, and Stephen R. Wheat. SUNMOS for the Intel Paragon: A Brief User's Guide. In *Proceedings of the Intel Supercomputer Users' Group*, pages 245–251, 1994.

[73] J. F. Macfarlane and R. Armstrong. POET: A Parallel Object–Oriented Environment and Toolkit for Enabling High–Performance Scientific Computing. Technical Report, Sandia National Laboratory, Livermore, Ca. 94551, 1993.

[74] T.G. Mattson, R. Bjornson, and D. Kaminsky. The C-Linda Language for Networks of Workstations. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, Florida, USA, 1992.

[75] L. C. McInnes and B. Smith. PETSc 2.0: A Case Study of Using MPI to Develop Numerical Software Libraries. In *Proc. of the 1995 MPI Developers Conference*, University of Notre Dame, 1995.

[76] Message Passing Interface Forum. MPI: A message passing interface. In *Proc. Supercomputing '93*, pages 878–883. IEEE Computer Society, 1993.

[77] Bertrand Meyer. *Object–oriented Software Construction*. Prentice Hall, New York, 1988.

[78] R. G. Minnich. Mether-NFS: A Modified NFS Which Supports Virtual Shared Memory. In *Proc. of the Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS-IV)*, pages 89–107, September 1993.

[79] Ronald G. Minnich and David J. Farber. The Mether system: distributed shared memory for SunOS 4.0. *Usenix*, Summer 1989.

[80] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. *IEEE Computer Magazine*, May 1990.

[81] Stephan Murer, Jerome A. Feldman, and Chu-Cheow Lim. pSather: Layered Extensions to an Object–Oriented Language for Efficient Parallel Computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, Ca., 1993.

[82] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, 1991.

[83] NSF. The National Science Foundation Metacenter. A report prepared for the program advisory committee to the national science foundation division of advanced scientific computing, National Science Foundation, 1992.

[84] T. O'Brian, G. R. Roberts, R. Winder, and A. McEwan. UC++ v1.4 Language Definition and Semantics. Technical Report, LPAC/UCL, 1995.

[85] Ronald Peierls and Graham Campbell. ALMS - programming tools for coupling application codes in a network environment. In *Proceedings of the Heterogeneous Network-Based Concurrent Computing Workshop*, Tallahassee, FL, October 1991. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.91.

[86] R.L. Pennington. Distributed and Heterogenous Computing. Technical Report, Pittsburgh Supercomputer Center, Pittsburgh, Pennsylvania, USA, 1995.

[87] A. J. Piper. Generalized Parallel Programming with Divide–and–Conquer: The Beeblebrox System. Technical Report CUED/F–INFENG/TR 132, Cambridge University, Engineering Department, 1993.

[88] P.L. Vaughan and A. Skjellum and D.S. Reese and F. Cheng. Migrating from PVM to MPI: The UNIFY System. Technical Report, Mississippi State University, 1994.

[89] Donna S. Reese and Ed Luke. Object oriented Fortran for development of portable parallel programs. In *Proceedings of the 3rd IEEE Symposium on Parallel Distributed Processing*, pages 608–615, Dallas, TX, December 1991.

[90] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Heterogeous parallel programming in Jade. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.92.

[91] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE.* O'Reilly & Associates, 1992.

[92] Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, Computer Science Department, University of Colorado at Boulder, April 1990.

[93] M. Rozier, V. Abrossimov, M. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. Technical Report CS/TR–90–25, Chorus Systemes, 1990.

[94] D. J. Scales and M. S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proc. of the Symp. on Operating Systems Design and Implementation (OSDI)*, pages 101–114, November 1994.

[95] Michael Schrage. Piranha processing - utilizing your down time. *HPCwire (Electronic Newsletter)*, August 1992.

[96] J. Shirley. *Guide to Writing DCE Applications.* O'Reilly & Associates, 1992.

[97] P. Sivilotti and P. Carlin. A tutorial for CC++. Technical Report CS-TR-94-02, Caltech, 1994.

[98] A. Skjellum. The Multicomputer Toolbox: Current and future directions. In *Proc. Scalable Parallel Libraries Conf.*, pages 94–103. IEEE Computer Society, 1993.

[99] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communications library atop the Reactive Kernel. In *Proceedings of the 5th Distributed Memory Computer Conference*, pages 767–776. IEEE Press, 1990.

[100] Glen Smith. Object-oriented Fortran tutorial. Engineering Research Center, Mississippi State University, 1992.

[101] G. Stellner, S. Lamberts, and T. Ludwig. *NXLib Users Guide Version1_1_2.* Technische Universität München, 1994.

[102] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, 1990.

[103] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[104] L.H. Turcotte. A Survey of Software Environments for Exploiting Networked Computing Resources. Technical Report, Engineering Research Center for Computational Field Simulation, Mississippi, MS, 1993.

[105] V. van Dongen, C. Bonello, and C. Freehill. Data Parallelism with High Performance C. In *Supercomputing Symposium 94, Canada's 8th Annual High Performance Computing Conference*, 1994.

[106] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: An Operating System for Massively Parallel Systems. *Scientific Programming*, 3:275–288, 1994.

[107] J. Yan, P. Hontalas, S. Listgarten, et al. The Automated Instrumentation and Monitoring System (AIMS) reference manual. NASA Technical Memorandum 108795, NASA Ames Research Center, Moffett Field, Calif., 1993.