Programming Tools for Distributed Multiprocessor Computing Environments

Thomas Bemmerl and Bernhard Ries
Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Arcisstr. 21, D-8000 München 2, FRG
Tel.: +49-89-2105-8247 or -2382, Fax.: +49-89-2800-529
e-mail: {bemmerl, ries}@informatik.tu-muenchen.de

With the increasing availability of multiprocessor platforms based on different types of architectures (shared memory, distributed memory, network based), users will be increasingly faced with heterogeneous and distributed multiprocessor computing facilities. Programming environment concepts have to be found which enable the user to program and use these heterogeneous computing resources, consisting of different multiprocessor architectures, in a transparent manner. Topics which have to be addressed and tools which have to be developed are parallel programming models, monitoring systems, debuggers, performance analyzers, animation systems and dynamic load-balancing schemes. The paper describes the integrated tool environment TOPSYS (TOols for Parallel SYStems), available for the iPSC/2 and iPSC/860 hypercube, as well as its adaptation to a network of UNIX-based workstations.

1 Introduction and Motivation

1.1 Distributed Multiprocessor Computing Environments

An increasing number of multiprocessor platforms is being integrated into computing facilities and centers in both research and industry. Therefore typical computing environments can be characterized as consisting of machines belonging to three different types of computer architectures:

- Single processor workstations from different manufacturers and with different processors.
- Shared memory multiprocessor workstations and mini-supercomputers with up to several tens of processors.
- Distributed memory multiprocessor systems with MIMD or SIMD architecture and up to thousands of processors.

Multiple computing systems based on these three different kinds of architectures are typically connected via various forms of networks. All three types of systems are used in cooperation in order to provide the services requested by different users of such a distributed multiprocessor computing facility. Figure 1 illustrates, as a representative example, the distributed multiprocessor computing facility used as an experimentation environment for the research described in this paper.

Various types of operating systems, programming models, communication services and programming tools are provided on the different classes of machines. While versions of the UNIX operating system are widely available on single processor workstations and shared memory multiprocessors, the distributed memory multiprocessors provide only very restricted and limited operating system facilities until now. The same heterogeneous situation exists with respect to

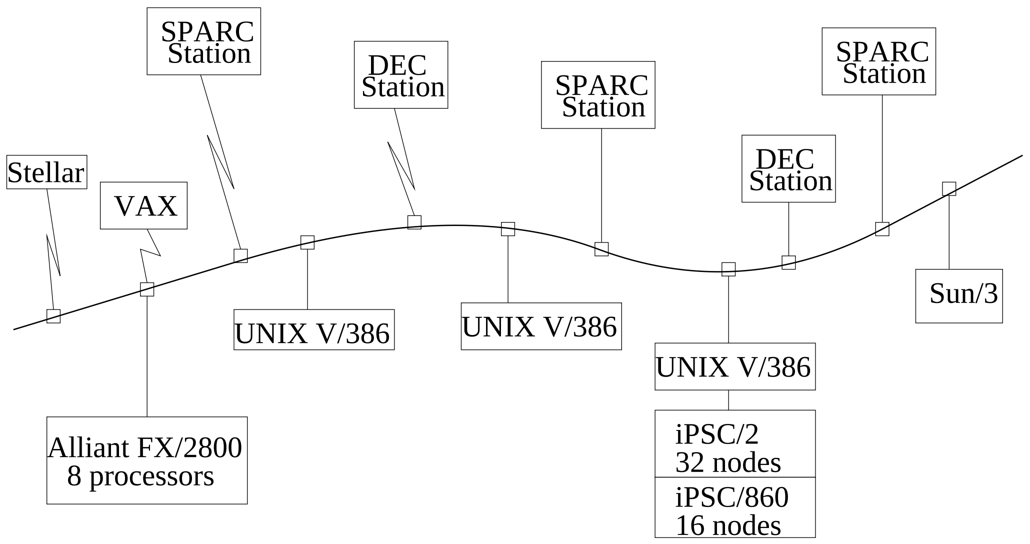


Figure 1: Distributed multiprocessor computing environment

programming models. On single processor workstations, sequential or multiprocess programs are used. On shared memory multiprocessor systems, programmers write their programs based on different types of parallel programming models using shared variables for communication purposes [12]. Finally, on distributed memory multiprocessors, different styles of message passing parallel programming models are provided. Message passing programming models are also favored for communication between workstations via the various types of networks [3, 8].

1.2 User Requirements and Problems

Users of a distributed multiprocessor computing environment have the general requirement to use the available services and resources as flexibly as possible. This means that they are interested in using the available computer architectures in a cooperating manner. The following requirements for a distributed multiprocessor computing environment can be deduced from this statement:

- It must be possible to run each application on every type of machine.
- Performance scalability of application programs across the available resources must be guaranteed, e.g. by running the application with a small problem size on a single workstation and using the whole computing facility for large problem sizes.
- The use and programming of all resources and types of architectures available in the computing facility must be transparent.

These general requirements which application programmers have when using distributed multiprocessor computing environments lead to three classes of technical problems in programming these types of computing facilities. The research done in the TOPSYS (**TO**ols for **P**arallel **SYS**tems) project addresses these problems:

- Unified parallel and distributed programming models for the various types of computing systems: The programming model has to support the programming of a single machine (single processor, shared memory multiprocessor, distributed multiprocessor) as well as programming a cooperation between any number and type of machines.
- Common transparent programming tools to deal with all the different multiprocessor architectures: It is particularly important to focus on tools for the late phases of parallel program

development (e.g. debuggers, performance analyzers, visualizers, monitors, etc.) because the design of these tools is more architecture dependent.

- Operating system and management software for control of distributed multiprocessor computing environments: Problems to be solved here are user management, accounting, distributed program loading, load-balancing, network management, etc.

1.3 Outline of the TOPSYS Project

The Munich University of Technology (TUM) has implemented MMK (**M**ultiprocessor **M**ultitasking **K**ernel), a multithreaded programming model, and TOPSYS, a set of programming tools for MMK. The target hardware environment for MMK has been so far the distributed memory multiprocessors iPSC/2 and iPSC/860. The user interface components of TOPSYS are implemented on UNIX based workstations running X-Windows.

The long-term objective of the TOPSYS project is to simplify not only the use and programming of distributed memory multiprocessors. The parallel programming model and integrated tool environment is being extended to support distributed computing environments also. Therefore, prototype software has to be developed in the future which implements:

- a unified programming environment for distributed multiprocessor applications and
- programming tools for that environment.

The key steps toward the extension of MMK and TOPSYS with programming models and tools to support distributed multiprocessor computing environments are:

1) To port MMK so that it also targets networks of workstations and shared memory multiprocessors. To port TOPSYS so that it targets applications built on the new version of MMK.
2) To enhance MMK to support distributed multiprocess applications as well as multithreaded applications within a single unified parallel programming model based on distributed and shared memory multiprocessors as well as networks of workstations.
3) To enhance TOPSYS to provide programming tools for the enhanced MMK parallel programming model.

This paper describes the port of MMK and TOPSYS to a network of Sun SPARCstations [10] (Step 1 of the research project outlined above). The intention of the document is to give an introduction to MMK and TOPSYS as well as to explain major design decisions and implementation details of the port.

1.4 Summary of the MMK and TOPSYS Environment

Within the TOPSYS project which has been going on for several years at the Department of Computer Science at the Munich University of Technology, a parallel programming model and various tools for programming parallel systems have been developed. The first version of TOPSYS was implemented for distributed memory multiprocessors of type iPSC/2 and iPSC/860. Approximately 25 installations of the TOPSYS environment are in use at the moment. The system currently consists of the following components:

- The MMK message passing multi-threaded programming model provided as a programming library,
- the distributed and parallel debugger DETOP,
- the performance analyzer PATOP,

- the visualization tool VISTOP for animation of thread interaction,
- an application transparent dynamic load-balancing component and
- the software package ALLOCCUBE for managing the space-sharing multiuser access to distributed memory multiprocessors in a network based environment.

A number of projects address the problem of providing programming models for distributed computing environments [1, 2, 7, 11, 13, 14]. However, most of these systems provide little or no support for debugging and performance tuning of distributed applications developed using these programming paradigms. In contrast, the TOPSYS environment comprises tools which support the programmer during most of the critical phases of the software development cycle.

The most important feature of the TOPSYS environment is its location transparency. This means that all the tools provide their services independent of the location of processes in the distributed multiprocessor. The following chapters will describe the architecture and the functionality of the MMK and the TOPSYS tool environment in more detail.

2 The MMK Programming Model

The MMK (Multiprocessor Multitasking Kernel) is a library of routines that provide lightweight concurrency on uniprocessor nodes and transparent communication and synchronization mechanisms across processor nodes. The implementation is based on the NX/2 operating system for the iPSC family of distributed memory multiprocessors. The following section describes the key features of the MMK programming model and the existing implementation [4, 6].

At the moment there are two contrary approaches for the design of programming languages for distributed memory machines. The first and conventional approach consists of extending existing sequential languages (C, Fortran, Pascal) with parallel constructs (e.g. communication and synchronization primitives). The second approach tries to define new parallel languages based on functional, object oriented or logical programming paradigms [3]. MMK is a compromise between the two approaches. The MMK programming model is based on conventional sequential languages but also provides dynamically creatable objects and global operations on these objects. The result is a dynamic, transparent process model which allows the programmer to leave the specific architecture out of consideration during program design. The key features and design concepts of the MMK are:

- The MMK provides a lightweight process model with transparent global operations. The programmer can define any number of processes. During program construction the programmer has to keep in mind neither the number of processes nor their intended location on the processor nodes.
- When using the MMK programming model, the programmer thinks about his parallel program in an object oriented manner. The MMK provides active objects (tasks), communication objects (mailboxes), and synchronization objects (semaphores) with a predefined set of operations for object manipulation.
- The MMK programming model is based on the message-passing paradigm. Tasks exchange messages and synchronize by explicitly calling the appropriate MMK routines.
- All objects and operations can be accessed locally and globally in exactly the same way. The programmer does not have to keep in mind whether the objects he manipulates are located on the local processor node or on a remote one.
- The MMK provides only a minimum set of objects with their corresponding operations (tasks, mailboxes, semaphores). This design concept of minimality reduces the complexity of implementing dynamic object migration, dynamic load-balancing strategies and monitor support.

- All objects of a parallel program based on MMK can be dynamically created and deleted. The number of objects is only limited by the amount of available memory. This dynamic characteristic of the MMK allows user implemented dynamic load balancing strategies and the implementation of non-numerical applications based on code (task) partitioning.

We will now discuss the MMK programming model in more detail. In particular, we describe the object types and the operations for object manipulation.

MMK supports three different object types: tasks, mailboxes and semaphores. The semantics of mailbox and semaphore objects are predefined (tunable through a number of parameters) whereas the semantics of tasks are defined by the programmer. We will call tasks active objects and refer to mailboxes and semaphores as passive objects. Objects can only be manipulated via operations defined on them including operations to dynamically create and delete objects.

2.1 Tasks

The active objects of a distributed MMK application are the tasks. A task is a thread of control whose semantics are defined by a program component (called task body) written by the application programmer. Each task belonging to an application program is identified by its object identifier and the name of its task body. In this way, replication of tasks is possible by having tasks with different identifiers but identical task bodies. MMK supports C, C++ and FORTRAN, and allows an arbitrary mixture of task bodies written in any of these languages. The structure of a MMK task body is shown in figure 2.

```
♯include "mmk.h" /* MMK_SYSTEM Include */

TASK(doit_code, pred, succ, output, sync)
{

        unsigned int **matrix;
        struct Initpar *initpar;
        int reply,length;
        ...

        /* receive initial message */
        initpar = (struct Initpar *)recmsg(pred,UNLIMITED,&length);
        ...

        /* synchronize with host program */
        reqsema(sync,1,UNLIMITED,&reply);
        ...

}
```

Figure 2: MMK Task Body

There may, of course, be more than one task on a processing node. All tasks on a given processor node execute within one process of the underlying operating system and thus share a common address space. Tasks which share a processing node are scheduled according to a round-robin scheduling strategy.

The information exchange between tasks is done using passive objects: mailboxes are used for communication and semaphores for synchronization. In order to access a passive object, a task must know the object identifier. This identifier can be statically passed to the task as a parameter or announced dynamically by sending the identifier to the task. In the following sections we will discuss these passive objects in detail.

2.2 Mailboxes

The communication between tasks is implemented using mailboxes. Mailboxes support a wide range of varying communication protocols. They can be attached to two tasks (providing one-to-one communication) or to three or more tasks (providing many-to-many communication). Every task can read from and write to mailboxes; there is no predefined communication direction.

Every mailbox has a user-defined size which specifies the number of messages a mailbox can hold. Messages can be of any length, from zero bytes to the limit of physical (or virtual) memory. When a message is sent to a mailbox, a memory buffer is allocated, the message is copied into the buffer, and the pointer is stored in a list.

When a task receives a message from a mailbox, a pointer to the memory buffer is returned and the pointer is deleted from the mailboxes' list. The user program can then manipulate the message and is responsible for freeing the memory buffer. Any given message can only be received by one task. Since the identifier of the sender is not contained in the message, it is not possible to receive messages selectively.

Mailboxes support a timeout mechanism for tasks waiting for completion of their send and receive calls. The programmer can use this to specify waiting times from zero to unlimited in order to implement asynchronous or synchronous communication with the mailbox. The possibilities resulting from combinations of buffer size and waiting time will be discussed later.

2.3 Semaphores

MMK provides counting semaphores with request and release operations for the purpose of task synchronization. In analogy to mailboxes, semaphores may also be attached to two or more tasks and provide the same timeout mechanism for tasks requesting semaphore units.

2.4 Object Connection and Mapping

After having written the task bodies, the user must specify the connectivity of the objects, i.e. the relation between passive and active objects, and the mapping of these objects onto the processing nodes. The user specifies the object connection and mapping in a special description file called the mapping file. This file has the following structure:

Every static object which is to be created at program start, has to be declared. In order to provide globally unique objects, each object entry in the mapping file must be assigned a unique name. The mapping of the objects onto the processor nodes is performed by supplying a node number for every object. This makes it comparatively easy to adapt an application to a varying number of processor nodes.

In addition to name and node number there are a number of other parameters which can be used to specify the exact semantics of objects (task stack size, size of mailbox etc.).

A task also needs to know the identifiers of the objects which it has to access (e.g. mailbox-identifiers). These identifiers can be passed to the task by adding their names to the task object definition in the mapping file, which causes the identifiers to be passed to the task as parameters when the task is created.

A typical MMK mapping file is shown in figure 3.

3 The TOPSYS Integrated Tool Environment

TOPSYS (TOols for Parallel SYStems) is an integrated tool environment which provides a general methodology and programming support for all the phases of the software development cycle for parallel applications [4, 5]. The tool environment currently consists of a graphical debugger, a

```
GLOBAL_BEGIN

SEMA( sync INIT 0 NODE 0)

MBOX( input INIT 50 NODE 0)
MBOX( output INIT 50 NODE 0)
MBOX( mb0 INIT 0 NODE 0)
MBOX( mb1 INIT 0 NODE 0)
MBOX( mb2 INIT 0 NODE 1)
MBOX( mb3 INIT 0 NODE 2)
MBOX( mb4 INIT 0 NODE 3)

TASK( upper INIT top , 32000, input, mb0, sync NODE 0)
TASK( task_A INIT doit_code, 32000, mb0, mb1, output, sync NODE 0)
TASK( task_B INIT doit_code, 32000, mb1, mb2, output, sync NODE 1)
TASK( task_C INIT doit_code, 32000, mb2, mb3, output, sync NODE 2)
TASK( task_D INIT doit_code, 32000, mb3, mb4, output, sync NODE 3)
TASK( lower INIT bottom , 32000, mb4, output, sync NODE 3)

HOSTTASK( output, input, sync)

GLOBAL_END
```

Figure 3: MMK Mapping File

performance analyzer and a visualization tool for parallel programs implemented with the MMK programming model. All the tools need information about the dynamic behavior of the parallel application. This information is gathered by a monitor system which supports both software and hardware monitoring techniques. The implementation of TOPSYS is based on a hierarchy of subsystems which leads to a modular and portable design. We will first describe this hierarchical tool model and then go on to explain the different tools of the TOPSYS architecture in more detail.

3.1 The TOPSYS Hierarchical Tool Model

The TOPSYS architecture is based on a hierarchical, modular design. This layered approach is called the TOPSYS hierarchical tool model and is illustrated in figure 4. The figure shows the interfaces between the subsystems of the development environment. The hierarchical approach is motivated by the idea that it should be possible to use any tool with any monitoring technique regardless of the implementation of the corresponding partner. Apart from the definition of some smaller interfaces, the integration idea has led to the definition of two important interfaces - the monitor interface (UTL, Universal Target Layer) and the tool interface (HLTL, High Level Language Tool Layer). The monitor interface is a command driven interface through which the upper layers of the tool model request runtime information from the different monitors on the processor nodes. The monitors are replicated and adapted or downloaded to the processor nodes of the parallel target machine. All monitor types of this distributed monitoring system provide the same functionality to the upper layers of the tool environment and can therefore be freely exchanged. The monitor interface is based on virtual addresses and internal object identifiers of the MMK kernel, i.e. all objects within the commands of the monitor interface are referred to by their virtual address or object identifier.

The UTL provides the same services as the tool interface at a lower abstraction level and

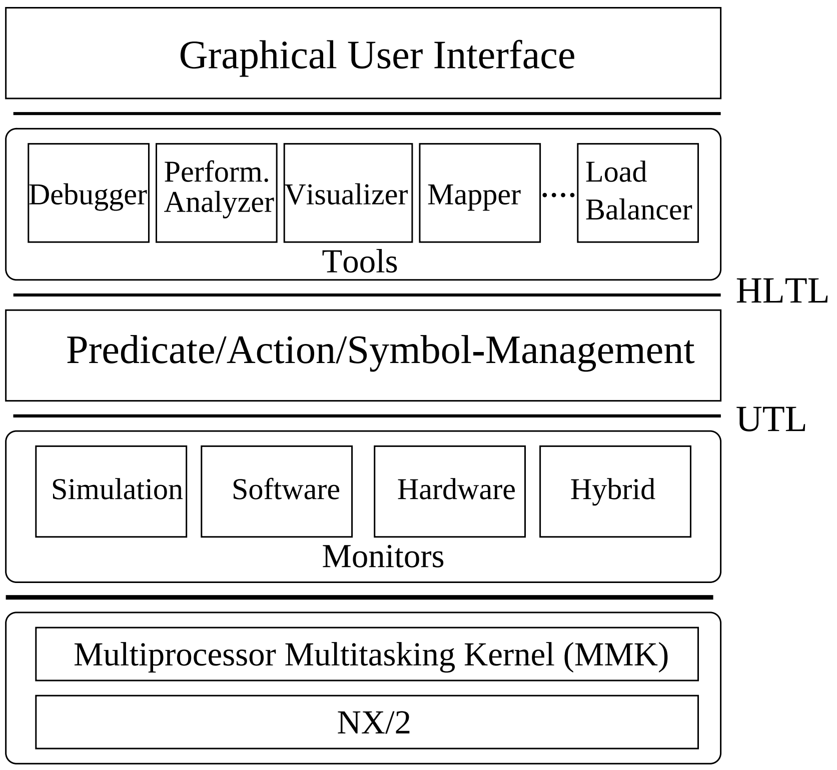


Figure 4: The TOPSYS Hierarchical Tool Model

restricted to one processor node (local). The UTL commands are therefore almost identical to the commands of the HLTL: However, while the UTL provides commands at the assembly language level, the tool interface (HLTL) works at the abstraction level of the source program. All objects of this interface are referred to by their symbolic names in the source program, e.g. variable names, task names, etc. This interface makes it possible to extend the functionality of the tools without modifying the lower layers. The tool interface is a procedural interface. The tools use it to request runtime information about the dynamic program execution at the source code level. This is done through a set of calls which allow the tools to control the parallel execution, to monitor execution, to inspect the state and to gather performance data. The HLTL calls can be grouped into the following categories:

- Calls used to start tasks and to execute tasks in single-step mode.
- Calls used to monitor program execution. These include commands to set, clear and list breakpoints or traces and to inspect the contents of trace buffers.
- Calls used to determine the state of the program, i.e. list existing MMK objects, list variables and show variable contents.
- Calls which modify the state of the program, e.g. modify variable contents.
- Calls used to define, start and remove performance measurements and to read performance data.
- Additional calls necessary to initialize the tool layer and to inspect the symbol table.

The functionality provided by the tool interface can be used by a variety of different tools (see Fig. 4). The current implementation of TOPSYS consists of a parallel debugging and test system, a performance analyzer and a visualization tool.

These tools demonstrate how the functionality of the monitor and tool interfaces can be used to provide detailed information on the dynamic behavior of a parallel program during runtime. It is relatively easy to create new tools with new functionalities based on the runtime information provided by the system calls of the tool interface. Another important point is, that it is possible to use all the tools in a cooperating manner. The programmer can very easily switch between the different tools by simply moving the mouse from one window to another.

The transformation between the abstraction level of the tool interface (source code) and the abstraction level of the monitor interface (machine code) is done within a central layer of the tool environment. This layer (Predicate/Action/Symbol-Management, see Fig. 4) is responsible for the management of specified events, actions and symbol table objects. The services provided by this layer are used by all tools.

Since most of the tools implemented within the TOPSYS project are interactive tools, special attention was given to the development of appropriate and easy to use user interfaces. All tools of the tool environment have a graphical and menu-driven user interface with the same look and feel, i.e. with the same philosophy of usage. A special graphics library developed on top of X-windows is used by all TOPSYS tools to implement this common graphical user interface. The tool environment is integrated into the network based host/target environment which is characteristic for distributed memory multiprocessors. A consequence of this is that the different layers of the tool model run on different machines. The distributed monitor system and the MMK kernel are implemented on each processor node of the parallel target system. The predicate/action/symbol management layer and the tools are implemented on the host which is logically connected to each processor node of the target system. The graphical interfaces can run on any X-window based workstation connected to the host computer via a local area network. This network based approach makes the TOPSYS functionality available to each programmer within the local area network. The fact that the different components are distributed across the host/target environment does, however, not restrict the portability of the TOPSYS architecture. In environments which do not have a special host system (e.g. multi-workstation nets) the upper layers of the tool environment can simply run on one of the processing nodes (i.e. on one workstation).

3.2 The Parallel Debugger DETOP

DETOP is a graphical high-level debugger for MMK applications. DETOP provides all the commands which can usually be found in sequential debuggers, e.g. starting and stopping tasks, determining task states, defining breakpoints and traces, displaying and modifying variable contents, listing source files etc. In addition, DETOP provides some unique features which are particularly helpful when debugging distributed applications:

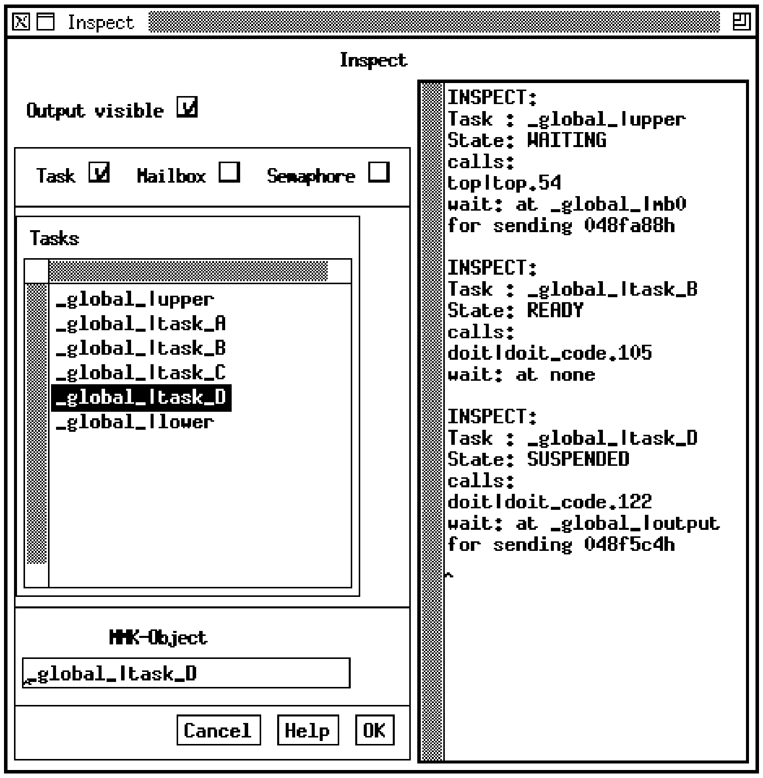
- In contrast to most other distributed debuggers, DETOP provides a global view of the distributed system. The user does not need to worry about processing nodes or internal object identifiers. All objects are accessed through symbolic names.

- DETOP not only monitors the execution of tasks but also communication between tasks and synchronization of tasks. It is, for example, possible to define traces which show how often a task sends messages to a mailbox.

- DETOP supports the distributed breakpoint concept described in the previous section. A breakpoint is executed when the associated breakpoint predicate is satisfied. This predicate consists of simple predicates which concern the state of objects that may reside on different processing nodes. Thus, a possible breakpoint definition would be: "Stop all tasks if task t_1 sends a message to mailbox m_1 and task t_2 receives a message from mailbox m_1 afterwards". Such a definition is possible regardless of the position of t_1, t_2 and m_1 on the processing nodes.

- DETOP supports different monitoring techniques (software monitoring, hardware monitoring, hybrid monitoring, simulation).

The DETOP user interface is based on the X-Window standard. Commands are selected using menus, and standardized windows are used to specify parameters and to display messages and output. Figure 5 gives an example of how DETOP can be used to determine the state of a distributed application. In this example, the user has executed a series of calls to inspect task objects. The output can be seen in the rightmost window.



Output visible ☑

Task ☑ Mailbox ☐ Semaphore ☐

```
_global_lupper
_global_ltask_A
_global_ltask_B
_global_ltask_C
_global_ltask_D
_global_llower
```

MMK-Object: _global_ltask_D

```
INSPECT:
Task : _global_lupper
State: WAITING
calls:
top|top.54
wait: at _global_lmb0
for sending 048fa88h

INSPECT:
Task : _global_ltask_B
State: READY
calls:
doit|doit_code.105
wait: at none

INSPECT:
Task : _global_ltask_D
State: SUSPENDED
calls:
doit|doit_code.122
wait: at _global_loutput
for sending 048f5c4h
^
```

Figure 5: Determining Task States in DETOP

3.3 The PATOP Performance Analyzer

PATOP is an interactive performance analysis tool for distributed applications. This tool can be used to reveal performance bottlenecks which can be caused by overloaded communication links and by an inappropriate distribution of the processor workload. This is done by defining performance measurements and using the results to tune the program and to optimize the mapping of program objects onto processor nodes.

PATOP supports performance measurements at different abstraction levels. The first possibility is to examine the system's performance as a whole. The next level supports performance measurements for the different processor nodes which form the distributed system. The finest level of granularity makes it possible to measure the performance in terms of the objects of the parallel programming model (i.e. in terms of MMK objects). Measurements supported on the different levels are idle-times, delays which occur waiting for resources and resource utilization. PATOP runs in parallel to the execution of the distributed application and the results are displayed on-line.

The following measurements can be specified using PATOP:

- System Level
  - Idle time: the sum of the idle time percentage on all processor nodes
  - Ready Queue: mean length of the ready-queue on all processor nodes

- Node Level
  - idle time percentage on a specific processor node
  - mean length of the ready-queue on a specific processor node
  - data volume exchanged between nodes
  - mean length of the remote action queue (this is an indicator for the average time tasks spend waiting at passive objects)
- Object Level
  - percentage of CPU-time consumed by a task
  - time a specific task spends waiting at mailboxes and/or semaphores (arbitrary combinations of tasks and passive objects are allowed, e.g. task t_1 and mailbox m_1, task t_1 and all semaphores etc.)
  - data volume sent from tasks and received by tasks from mailboxes

The PATOP user interface is very similar to the DETOP interface described in the previous section. The results of performance measurements can be displayed using line graphs or bar graphs. Figure 6 illustrates the output produced by PATOP for an idle time measurement.

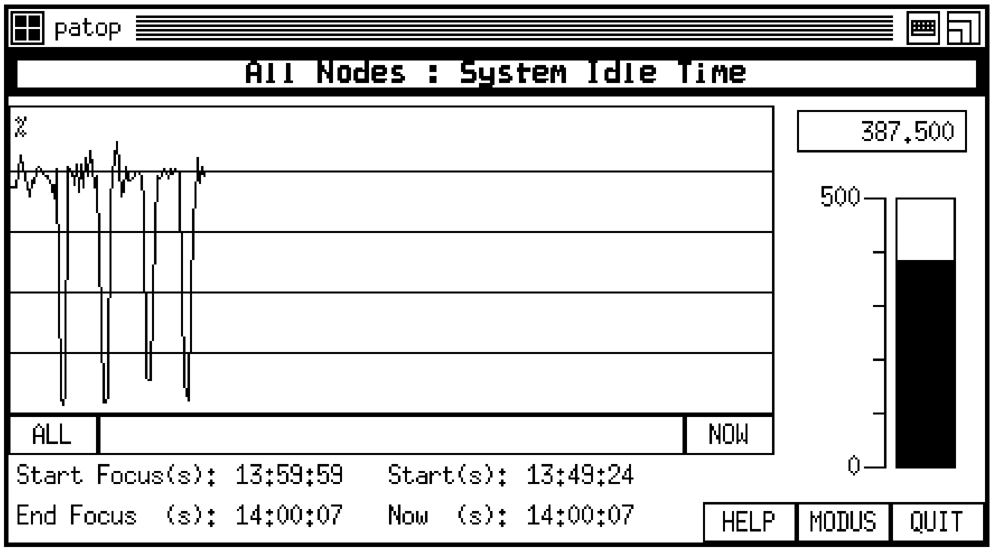
3.4 The VISTOP Visualizer

The visualization tool VISTOP allows the graphical animation of MMK programs and thus helps the programmer to visualize the topology and communication structure of a parallel program. VISTOP visualizes the structure of an application in terms of the basic MMK objects, i.e. tasks, mailboxes and semaphores. In order to visualize a distributed application, VISTOP collects run time information whenever a communication or synchronization takes place. The information consists of so called snapshots, which contain the status of every object under supervision. Snapshots are stored in an internal data structure, called the "animation queue". The events that have to be monitored are:

- a task sends a message to a mailbox,
- a task receives a message from a mailbox,
- a task requests units from a semaphore,
- a task releases units to a semaphore.

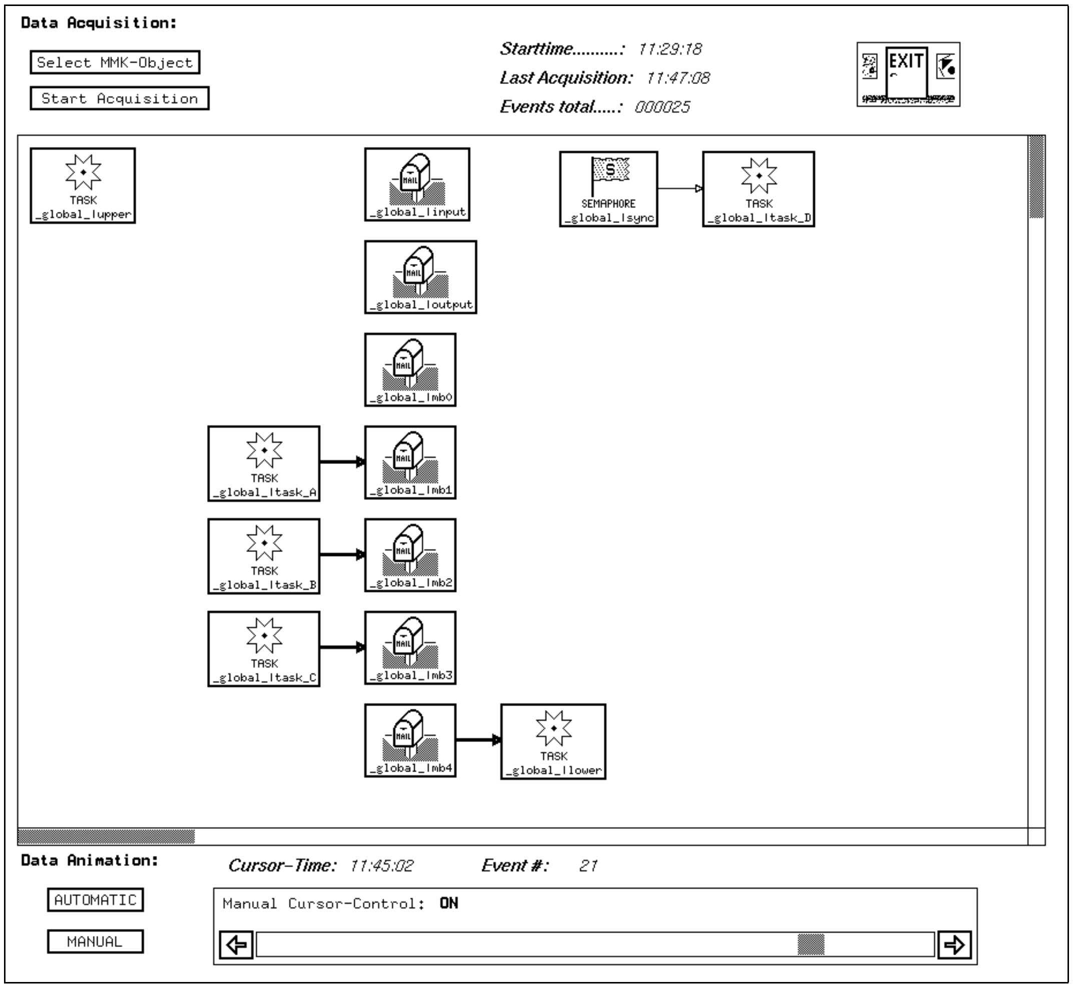
The collection of data from the monitoring system runs in parallel to the animation of the application. The buffering of snapshots in the animation queue thus allows an arbitrary replay of the program execution.

Figure 7 illustrates the visualization of a parallel program using VISTOP. Each MMK object is represented by an icon which shows the object type and the object name. If a task communicates with a mailbox or a semaphore, or if a task has to wait at a mailbox or a semaphore, the corresponding task icon is inserted into a queue of tasks waiting at the passive object. This is achieved by moving the icon from its location to the last position in the queue. Queues are visualized as chains of arrows between objects. Thin arrows characterize communication. Thick arrows indicate that a task has to wait. The direction of an arrow shows the direction of the requested data transfer. Thus, if a task is sending (or waiting to send) to a mailbox, the arrow is drawn from the task to the mailbox. If the task is receiving (or waiting for receipt) of a message, the arrow points to the task. Queues of sending tasks are located on the left side of the mailbox window, receiving tasks are placed at the right side of the mailbox window. The same idea is used to visualize request and release operations on semaphores. When a task leaves the queue, its window is moved back



Start Focus(s): 13:59:59   Start(s): 13:49:24
End Focus (s): 14:00:07   Now (s): 14:00:07

Figure 6: PATOP Measurement



Starttime..........: 11:29:18
Last Acquisition: 11:47:08
Events total.....: 000025

Cursor-Time: 11:45:02   *Event #:* 21

Manual Cursor-Control: **ON**

Figure 7: Program Animation in VISTOP

to its home position. The home position is either the place where the task was initially placed by VISTOP or the place the user assigned to it. The associated arrows are deleted.

All icons can be individually placed by the user. In addition, it is possible to deiconify the object icons. The icon is then replaced by a window which contains detailed information about the state of the selected object (task state, contents of mailbox buffers etc.). The icons used to represent MMK objects are shown in figure 7.

4 Adaptation of MMK and TOPSYS to Networks of Workstations

The following section describes the major design choices and decisions made during the port of MMK and TOPSYS to a network of UNIX based workstations. We will henceforth refer to this implementation as MMK/X (MMK/uniX) and TOPSYS/X.

4.1 Process Model

One of the most important design decisions was how to implement the MMK process model under UNIX. One possibility would have been to depart from the lightweight process model and implement MMK objects as heavyweight processes. The PVM system [13] is based on this approach. However, the heavyweight process model is not appropriate for massively parallel applications in which hundreds or thousands of threads may exist. We therefore chose to implement a true lightweight process model, even though this added complexity to the implementation, since we had to deal with all the problems typical for user-level threads packages (context switch, non-blocking I/O etc).

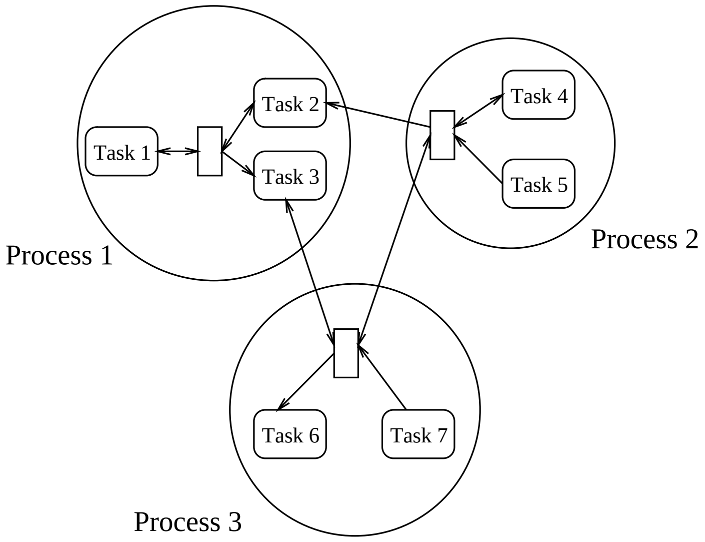


Figure 8: UNIX process as a virtual processor

In the MMK/X model, a UNIX process is seen as a virtual processor which may contain any number of MMK objects (see figure 8). The MMK scheduler is used to do the context switch within a virtual processor without kernel interaction. The application programmer specifies the mapping of MMK objects to processing nodes in two steps. The first step consists of grouping MMK objects into virtual processing nodes. This is done using the mapping notation shown in figure 3. A second file is used to specify the mapping of virtual processors onto real machines (see figure 9). It is, of course, possible to map more than one virtual processor onto a single machine (this can for example be used to provide true parallelism on a shared memory multiprocessor).

```
sunbode7       0
sunbode7       1
allbode1       2
131.159.0.145  3
```

Figure 9: Mapping virtual procesors onto processing nodes

4.2 Communication Primitives

The second critical issue consisted of how to implement the MMK communication and synchronization primitives. While communication within a virtual processor (i.e. a UNIX process) can simply be done by copying data between the tasks' address spaces, we need some form of global interprocess communication mechanism to implement communication and synchronization across the network. Possible alternatives include RPC-packages [8], the TLI interface or socket primitives. For compatibility reasons, we chose the Berkeley socket-interface to implement MMK/X communication. The socket primitives are accessed through a software layer that provides the same functionality as the NX/2 operating system on the iPSC. This layer is used to dynamically open and close socket connections between the virtual processors.

4.3 Daemons and Server Programs

A number of service programs provide support for starting and running a distributed MMK/X application. Every machine which is to be used as a processing node must run a daemon program. This daemon is contacted when the application is started and is used to spawn new virtual processors and control the application. In addition to the MMK communication primitives, MMK/X tasks can also access a library which provides remote I/O functions, such as reading and writing from a terminal. This library sends the task's requests to a server process which is started together with the distributed application. The application programmer thus has full control over the distributed application.

4.4 Support for Heterogeneity

One of the main goals of the TOPSYS project is to provide support for programming heterogeneous distributed computing environments. Special care was therefore taken to provide support for heterogeneity in MMK/X. A set of functions is provided to construct messages in a machine independent form. An archiver is used to merge executable files for different target architectures into a single program. This makes it easy to program applications which run on different machine types. MMK/X is currently implemented for the SPARC [9] and on the Alliant FX/2800 (an i860 based shared-memory multiprocessor).

4.5 Adaptation of the Programming Tools

In order to port the TOPSYS programming tools to a network of workstations running MMK/X, the upper layers of the hierarchy shown in figure 4 had to be adapted to the new target architecture.

Since the graphical user interface is implemented using X-Windows and UNIX, no major changes to this layer were necessary. The predicate/action/symbol management is inherently compiler and machine dependent and had to be adapted to the SPARC architecture and compilers.

The most complex part of the port consisted of adapting the software monitor to the SPARC platform and the process structure of a distributed MMK/X application. Monitor functions like breakpoints and data traces are very machine dependent and in some cases difficult to implement for a RISC processor such as the SPARC. Nevertheless, it was possible to retain the basic structure of the software monitor and the upper tool layers. The software monitor is implemented as a pair of MMK tasks which are responsible for program instrumentation and event detection. The monitor code is linked with the application program and runs in the context of the application (i.e. one software monitor on every virtual processor). The monitors communicate with each other and with the tools using socket primitives.

5 Conclusion and Future Work

The first phase of the project consisted of the port of MMK and TOPSYS onto a network of workstations in which single processor SPARCstations are interconnected via a TCP/IP network. The result of this work is an experimentation environment which implements one possibility to program distributed memory computing environments. A very important subsequent step to this phase has to be the use of the network-based MMK and TOPSYS by real users. MMK/X and TOPSYS/X should be intensively used and evaluated by application programmers. The feedback from these users will guide subsequent developments in this field. Some user experiences are already available from the use of MMK and TOPSYS within the distributed memory multiprocessors iPSC/2 and iPSC/860.

It is already clear that the most important question not addressed in the first phase of the project deals with the integration of shared memory, distributed memory and network-based multiprocessor technology. However, appropriate answers to this question will play a central role in using future distributed multiprocessor computing environments. Therefore the next phases of the project have to deal in particular with solutions to the question mentioned above.

The major answers and concepts, which have to be addressed will satisfy the user requirements 1) to 3) of section 1.2. The following alternatives and questions have to be discussed in the future:

- Integration of shared memory and distributed memory parallel programming models. Alternatives, which have to be discussed here are: pure message passing on a single processor and across a network; multi-threaded and multi-process models with shared variables in combination with remote procedure calls; shared address spaces and multi-threaded programming models across processor boundaries based on virtual shared memory, etc.). The choice of the most appropriate programming model should be guided by the principle of achieving a compromise between programming comfort and performance.

- The functionality of the TOPSYS tools has to be adapted adequately to the new programming model for distributed multiprocessor computing environments. In particular, concepts for supporting shared variables and address spaces have to be found, e.g. breakpoints on locks etc.

- Distributed monitoring techniques and location transparent name services for tools like parallel debuggers, performance analyzers and visualizers. Within this research topics, the very dynamic behavior of multithreaded programming concepts has to be dealt with. A key question is how threads, communication and synchronization objects which are scheduled dynamically on different processor nodes, can be monitored.

- Location transparent user and program management in distributed multiprocessor computing environments: Concepts have to be devised, which make it possible to use a network of multiprocessor systems with different architecture for solving one application. Problems are program loading, dynamic configuration, access rights on other machines, accounting and dynamic load-balancing.

Future work has to discuss the advantages and disadvantages of the various alternatives and derive a design concept for a unified programming model and programming tools for distributed multiprocessor computing environments.

References

[1] F. Armand, F. Herrman, J. Lipkis, and M. Rozier. Multi-threaded Processes in Chorus/MIX. In *EUUG Spring 1990 Conference Proceedings*, pages 1–13, Munich, Germany, Apr. 1990.

[2] O. Babaoglu, L. Alvisi, A. Amoroso, and R. Davoli. Paralex: An Environment for Parallel Programming in Distributed Systems. Technical Report UB-LCS-91-01, Department of Mathematics, University of Bologna, Piazza Porta S. Donato, 5, 40127, Bologna, Italy, Feb. 1991.

[3] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, Sept. 1989.

[4] H. Beier, T. Bemmerl, A. Bode, et al. TOPSYS - Tools for Parallel Systems. SFB-Bericht 342/9/90 A, Technische Universität München, Munich, Germany, Jan. 1990.

[5] T. Bemmerl, A. Bode, et al. TOPSYS - Tools for Parallel Systems (User's Overview and User's Manual). SFB-Bericht 342/25/90 A, Technische Universität München, Munich, Germany, Dec. 1990.

[6] T. Bemmerl, A. Bode, T. Ludwig, and S. Tritscher. MMK - Multiprocessor Multitasking Kernel (User's Guide and User's Reference Manual. SFB-Bericht 342/26/90 A, Technische Universität München, Munich, Germany, Dec. 1990.

[7] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. *ACM Operating Systems Review*, 23(5):147–158, Dec. 1989.

[8] J. R. Corbin. *The Art of Distributed Applications - Programming Techniques for Remote Procedure Calls*. Sun Technical Reference Library. Springer Verlag, New York, Berlin, Heidelberg, 1990.

[9] R. Garner, A. Agrawal, F. Briggs, et al. The Scalable Processor Architecture (SPARC). In *Proceedings of the 33rd IEEE Computer Society International Conference (Spring CompCon '88)*, pages 278–283, San Francisco, CA, Mar. 1988. IEEE.

[10] S. Kleiman and D. Williams. SunOS on SPARC. In *Proceedings of the 33rd IEEE Computer Society International Conference (Spring CompCon '88)*, pages 289–293, San Francisco, CA, Mar. 1988. IEEE.

[11] H. Moons and P. Verbaeten. Distributed Computing in Heterogeneous Environments. In *EUUG Spring '90 Conference Proceedings*, pages 15–25, Munich, Germany, Apr. 1990.

[12] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multithread Architecture. In *Proceedings of the USENIX Winter '91 Conference*, pages 1–14, Dallas, TX, 1991.

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

[14] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach Threads and the Unix Kernel: The Battle for Control. In *Summer 1987 USENIX Technical Conference*, pages 185–197, Phoenix, AZ, June 1987.