A Migratable User-Level Process Package for PVM

Ravindranath Bala Konuru
M.Tech, Osmania University, India 1985

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

July 1995

The dissertation "A Migratable User-Level Process Package for PVM" by Ravindranath Bala Konuru has been examined and approved by the following Examination Committee:

Steve W. Otto
Assistant Professor
Thesis Research Advisor

Jonathan Walpole
Associate Professor
Thesis Research Advisor

David Maier
Professor

Adam Beguelin
Assistant Professor
Carnegie Mellon University

Dedication

To my parents, Raghu, Raji, auntie, uncle and Harini

Acknowledgements

I am deeply indebted to my advisors Steve Otto and Jonathan Walpole, whose wisdom, understanding, and encouragement were instrumental in completing my thesis. I greatly enjoyed working with them and it was a wonderful learning experience.

I would like to thank the other members of my thesis committee, David Maier and Adam Beguelin. Their perceptive comments have helped me immensely in improving both the style and the technical content of the thesis.

It was a pleasure to work with Dan, Jeremy and Rob, members of our MIST group and I thank them for the many stimulating technical and philosophical discussions.

Jon Inouye has been my friend, project-mate and colleague for the past few years and shared many happy and miserable moments. I will miss his cheerful and helpful presence when I leave.

I have been fortunate to find friends such as Bennet, Jenny, Judy, Kay, Nanda, Richard, Roger and Scott. We have had some wonderful times together and they helped me in maintaining a balanced perspective on life.

I would like to thank our secretaries Jo Ann and Kelly for always helping us out with a smile and our system staff Bruce, Marion, Mark, and Nike for the smooth running of the computer science department.

Finally, I thank my parents, Raghu, Raji, auntie, uncle and Harini for their love, support, patience and understanding.

Contents

List of Tables

List of Figures

Abstract

A Migratable User-Level Process Package for PVM

Ravindranath Bala Konuru
Oregon Graduate Institute of Science & Technology

Supervising Professors: Steve Otto and Jonathan Walpole

This dissertation studies an approach to supporting efficient processor virtualization and dynamic load balancing for message-based, parallel programs.

We propose the User-Level Process (ULP) abstraction that can be used to implement efficient local communication and transparent migration. The viability of ULPs is demonstrated through UPVM, a prototype implementation of the PVM message passing interface using ULPs. Typically, PVM programs written in Single Program Multiple Data (SPMD) style need only be re-compiled to use this package. The design of the package is presented and the performance analyzed with respect to both micro-benchmarks and some complete PVM applications.

Finally, we discuss aspects of the ULP package that affect its portability and its support for heterogeneity, application transparency, and application debugging.

Chapter 1

Introduction

This thesis is concerned with supporting simple and high performance message-based parallel computing on Distributed Memory Multiprocessors (DMMPs). A DMMP is a set of processors, each with its own memory and connected via an inter-connection network (see Figure 1.1). Each processor considers its own memory to be *local*, and that belonging to a different processor *remote*. Access to remote memory requires the intervention of the processor local to that memory. Because of this restriction, DMMP architectures are also called NORMA (No Remote Memory Access) architectures. Each processor has an operating system (OS) that manages the processes on that processor and services system calls. By far the most widely used approach to writing programs for DMMPs is message-based programming. Typically, it involves coding the application as a group of virtual processors (VPs), each VP sequentially executing its own code and using message passing calls such as **send()** and **receive()** for communication and synchronization.

An example of the DMMP architecture is the Intel Paragon, IBM SP2, or any network of workstations (NOWs). With the advent of high-performance processors and high-speed networking, NOWs have become an attractive and economical alternative to super-computers for parallel computing [Tur93, KN93, BBB$^+$93].

An emerging characteristic of modern DMMPs is that they are being used as multi-user, multi-tasking systems. In this multi-user environment, the number of parallel applications executing can vary unpredictably and the DMMP nodes themselves may "belong" to different users (particularly in the case of NOWs). Such an environment

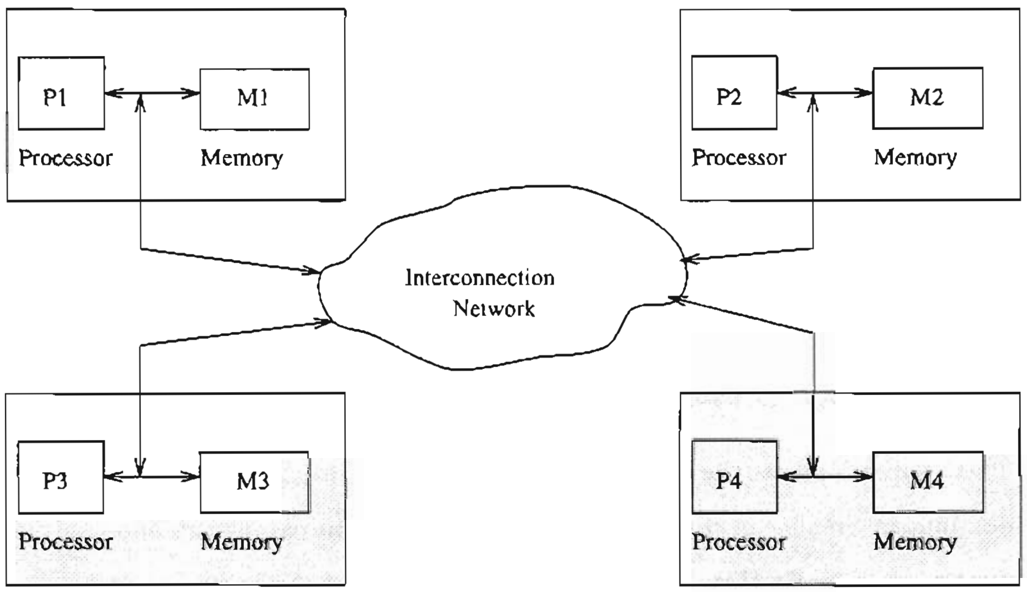

Figure 1.1: Distributed memory multiprocessor architecture

raises several difficult problems that limit application performance and overall system resource utilization.

First, the end-to-end message communication latency on a DMMP is typically large when compared to that of a local memory access, and can have a significant effect on application performance. Consider the following commonly accepted application cost model, in which the program is assumed to alternate between computation and communication phases and communication takes a time linear in message size, plus a start-up cost [FJL+88]. Using this model, the program run time is $T = t_{compute} + t_{communicate}$ and $t_{communicate} = n_{comm}(t_{start} + l_{msg}t_{byte})$, where t_{start} is the message startup-cost, t_{byte} is the time per byte transfer over the communication medium, l_{msg} is the length of the message, and n_{comm} is the number of communications. Note that since the processor is used to execute message startup code, the actual time (t_{cpu}) spent on the processor is the sum of $t_{compute}$ and $(n_{comm}t_{start})$. Thus one can rewrite the equation for program run time as $T = t_{cpu} + n_{comm}l_{msg}t_{byte}$.

In order to achieve 90% of the peak processor performance, a programmer has to divide the application into large partitions for achieving a computation to communication ratio that is sufficiently high to guarantee $t_{compute} \geq 9t_{communicate}$. Large partitions result in better processor locality and reduce inter-processor communication. However, having such large partitions tends to reduce application parallelism and increase overall application run-time.

If message communication can be overlapped with the application's computation though, the application run-time is given by the equation

$$T = max(t_{cpu}, n_{comm} l_{msg} t_{byte}).$$

This approach allows the partition size to be reduced until the second term $(n_{comm} l_{msg} t_{byte})$ dominates. Smaller partitions imply higher application parallelism and can therefore reduce overall application run-time.

Second, the distribution of load among allocated processors significantly affects application's performance. Assume that an application's computation is equally divided among these processors. If some of the processors have more load than others, application performance is then limited by the computation speed of the processor with the heaviest load. Schemes to achieve load balancing can be static or dynamic. A static scheme is one in which the application load is distributed across the allocated processors based on their load at application startup. However, such schemes are suitable only in those cases where the application load can be predicted accurately and is executed on a fixed number of processors. On DMMPs where processors loads are unpredictable, dynamic load balancing schemes have been shown to be superior to static schemes in terms of application performance [ISB86].

Third, the use of NOWs introduces the need to manage workstations belonging to different users. An application belonging to one user should only use the "idle" or unutilized cycles of another user's workstation. If the owner requires more cycles of her workstation, the parallel computation must use fewer cycles, or treat the workstation as lost and evict the computation from that workstation. In other words, the parallel computation should be *unobtrusive* [LLM88]. Workstations can also be lost from a parallel computation when there are shutdowns for maintenance or OS upgrades. The need to deal with evictions such as these, and to utilize newly available workstations means that parallel applications on NOWs must be able to adapt to changes in in the number of processors allocated during execution and continue to run efficiently.

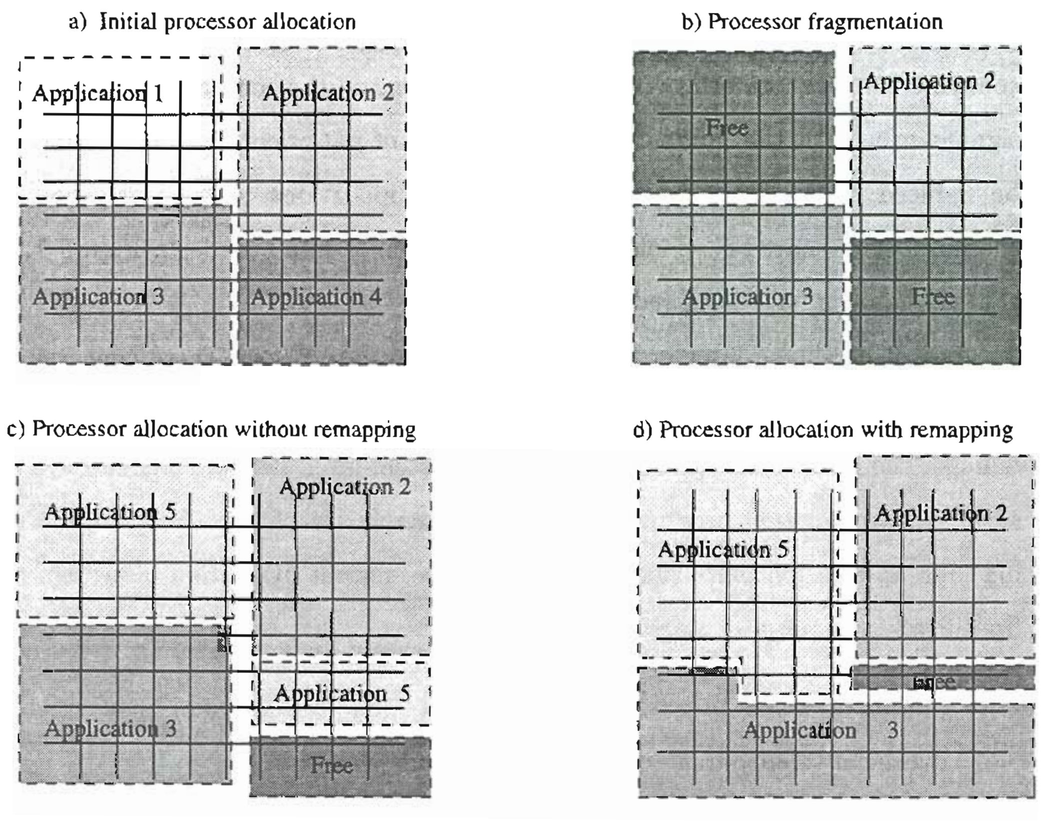

Figure 1.2: Processor fragmentation and use of dynamic reallocation

Finally, preventing fragmentation of DMMP resources is an important issue and can have a significant impact on application performance. Large DMMPs typically have multi-hop topologies [NM93]. That is, the communication between two processors can involve intermediate processors and the physical communication channels are shared. Such DMMPs, when used over long periods of time by applications with varying levels of parallelism, can experience *processor fragmentation*. For example, Figure 1.2 shows a mesh-based DMMP. After applications 2 and 3 terminate, the available processors in the mesh are not localized instead, they are fragmented across the DMMP. If these processors are allocated to an application, the communication between application's VPs contends for the shared channels with the communication of other applications. See Figure 1.2.(c). If these applications are long-lived, both application and overall system performance degrades. This degradation can be especially significant for workstation networks where it is typical for a single organization to divide its workstation network into sub-nets connected through gateways or bridges. In such cases, the communication cost increases significantly with the number of hops (number of gateways crossed). Contention can be reduced by dynamically reconfiguring the applications on to processors such that the number of shared channels among applications is minimized. In Figure 1.2.(d), the shared channels have been reduced to zero.

As will be shown later in this chapter, attempts to resolve these concerns by applications programmers result in programs that are complicated to write and difficult to debug. The goal of this thesis is to develop system-level abstractions and software that shield the application programmer from the complexity of these issues while maintaining high application performance and achieving efficient utilization of system resources. Specifically, we propose a software layer that:

- Supports virtual processors (VPs) that are simple-to-use and have sufficiently low overhead to encourage over-decomposition[1] [Val90].
- Supports dynamic and application-transparent VP migration.
- Is programming-language independent.
- Supports legacy, process-based, message passing with little or no rewriting.

In the following sections, we argue that these features reduce application programming complexity and offer the potential to achieve high performance.

[1]The use of more VPs than there are physical processors

1.1 The case for over-decomposition

Message passing primitives used by applications can be divided into two broad categories: blocking and non-blocking. For illustrative purposes, a template application written using blocking primitives is shown in Figure 1.3.(a) and a possible version of the same program using non-blocking primitives is shown in Figure 1.3.(b). The semantics of these primitives are adopted from the Intel NX library [Int91]

```
 Begin main_program
       read_my_portion_of_data();
       send(destnId, databuf);
 /* The buffer 'databuf' can be reused at this point */
      perform_computation_on_my_portion();
      receive(destnId, databuf);
/* databuf contains the message from the process *
 * having id = destnId */
      compute_based_on_received_data();
      store_results();
      exit();
End main_program
```

a) Use of blocking primitives

```
 Begin main_program
     read_my_portion_of_data();
      post_send(destnId, databuf,  snd_hdl);
    /* The contents of databuf are undefined and cannot *
     * be used until the function snd_hdl() is executed   */
    do_computation_not_using_databuf();
    /* Need databuf contents to go further, block on the *
     * post_request Id returned by post_send             */
    block(rqstId);
    /* Now databuf can be reused for other purposes */
    rqstId = post_receive(destnId, databuf, rcv_hdl);
    /* databuf will contain the message from destnId      *
     *  only when rcv_hdl() is executed                   */
    do_computation_in_parallel_with_recv();
    Mask_signals();
         execute_critical_section();
    Enable_signals();
    /* Need the message from destnId now. So block until *
     * the receive has actually completed */
    While (rcv_not_done(rqstId))  busy_wait();
    complete_rest_of_computation();
    store_results();
    exit()
 End Main_program
```

b)Possible use of non-blocking primitives

Figure 1.3: Use of blocking versus non-blocking primitives

Consider Figure 1.3.(a). The program reads the input data and sends the contents of `databuf` to the VP identified by `destnId`. The blocking semantics of `send()` specify that `databuf` can be modified when the program returns from the call and that the message sent will always correspond to the contents in `databuf` at the time of `send()`. The program then performs its computation until it needs data from `destnId`. It then invokes the blocking receive primitive **receive**(). This call causes the calling process to block until the message from `destnId` is received by the underlying operating system and placed into `databuf`. After returning from the receive call, the program performs some computation on `databuf`, stores the results and exits.

Blocking semantics, in combination with a one-to-one mapping of VPs to processors, can result in poor processor utilization, thereby degrading application performance. This under-utilization of processors can occur due to one or both of the following reasons:

- If an application's VPs are not executing in phase, then executing a blocking **receive**() causes a VP to block until a) the corresponding **send**() is executed by the destination VP and b) the message is delivered to the source VP. Even in the absence of hardware support for communication, the portion of time spent blocking for event a) to occur is better utilized for computation rather than blindly waiting for a communication that has not yet occurred.

- If hardware support for communication does exist, then blocking semantics can result in unnecessary serialization of certain operations and result in a VP being blocked longer than really needed. For example, because of the communication hardware, it is possible for a VP to execute on the processor while a message is being copied into its message buffers by the communication hardware. However, because of the blocking semantics, the VP has to be blocked until the communication hardware finishes copying the message into the VP's message buffers.

Thus, to achieve optimal processor utilization, it is essential that the message communication latency is hidden by overlapping message communication with the application's computation.

Programming with non-blocking message passing primitives is one way to achieve this overlap. Typically, non-blocking communication involves posting a request for communication, and using a variety of mechanisms to check for completion of the request.

The important characteristic is that the VP does not block when it posts a request. Non-blocking communication is useful with or without the presence of communication hardware.

In the absence of communication hardware, a VP can at least eliminate the time spent in a blocking **receive()** waiting for the message to arrive. One approach is to poll the OS for the arrival of the message while performing computation. The VP blocks only when all further computation is dependent on the receipt of the message. This reduction in blocking time improves processor utilization and consequently performance.

In the presence of communication hardware, non-blocking **send()** and **receive()** can be used such that hardware supported communication can occur in parallel with computation. This parallelism improves processor utilization and thereby performance.

Figure 1.3.b shows the non-blocking `post_send()`, that requests the underlying message system to perform the send operation. The VP is then free to continue its computation in parallel with the send operation until it reaches a point in the control flow where it needs to reuse `databuf` for some other operation. At this point, the program executes the statement `block(rqstId)`, where it waits for the send to complete. The `post_receive()` statement informs the underlying message system that `rcv_hdl()` should be invoked after the message requested is available in `databuf`. The program is then free to overlap its computation with the receipt of data into `databuf`. This approach is in contrast to a blocking receive, where a program has to remain blocked for a message to arrive and then be copied into *databuf*. Finally, when `databuf` needs to be accessed, the program checks for completion of the receive and performs the rest of the computation.

Such a program using non-blocking primitives overlaps its communication with computation, reducing the time spent waiting for completion of message communication, and consequently improving performance. However, notice the relative increase in the size and complexity of the program. Because of non-blocking message semantics, variables and data buffers can change concurrently with program execution as a result of previously posted, non-blocking requests. To manage this complexity, programmers may have to protect sections of their code that require sequential access (as shown by

`Mask_signals()` and `Enable_signals()` in the example). In short, non-blocking communication can improve performance but results in a significant increase in application programming complexity [FM92b]

An alternative and a much simpler programming approach for achieving overlap of communication with computation is to write a program using blocking primitives and instead of creating one VP per allocated processor, create multiple VPs per processor. Such a technique is called *over-decomposition* [Val90]. When one VP blocks, the underlying VP system simply schedules another VP. This scheduling essentially achieves an overlap of one VP's communication with another VP's computation. Using over-decomposition, application programmers can gain the software engineering benefits of coding with blocking message primitives.

1.2 The case for dynamic and transparent VP migration

System-level support for dynamic and transparent VP migration is a step towards reducing application programming complexity that stems from explicit management of load distribution within the application.

The programming complexity due to load-distribution depends on the parameters considered and the adaptiveness of the load-distribution scheme. The simplest schemes are static load-balancing schemes that assume that the application is allocated a fixed set of processors that are dedicated to the application or those for which the load remains relatively unchanged during application execution.

However, in a multi-user DMMP environment, this approach is not suitable both for the reasons of functionality and performance. Static schemes do not monitor and react to execution time events such as changes in processor load and processor owners requiring more cycles from their respective processors. This lack of adaptiveness in static schemes may degrade application performance, but more importantly, it is unacceptable for multi-user DMMPS like NOWs where unobtrusiveness is essential.

Adaptive load-balancing schemes perform dynamic load distribution based on information collected at execution time, such as processor, network, memory usage, and on dynamic changes in processor availability. The programming complexity of these schemes comes from an inherent need for inter-processor communication for collecting system information, reaching consensus, signalling and load re-distribution [Zho87].

One way in which adaptive applications can be written is to do some sort of polling for system status periodically during their execution and respond to any changes in system status. If the polling is frequent, it improves the application's responsiveness to changes in system status. However, frequent polling degrades applications performance. On the other hand, infrequent polling reduces the cost of polling but results in slower reaction times to changes in system status. Unfortunately, slower reaction times are unacceptable for NOWs, where unobtrusiveness is essential.

An alternative method of programming the application is to define and establish asynchronous event handlers for such eviction events. Such an approach is undesirable as it re-introduces asynchronous programming complexity within the application.

We therefore suggest that transparent VP migration be provided by VP systems. Such support has the following advantages. First, load-distribution can be achieved by migrating VPs instead of programmers explicitly writing application-specific code that redistributes the application's computation among the allocated processors. In other words, programmers are freed from having to think about load distribution mechanisms.

Second, because the load-distribution scheme is now independent of the application's data structures and based on VP migration, application programmers are freed from having to think about load-balancing schemes.

Finally, in conjunction with over-decomposition, a VP system that supports transparent VP migration has the ability to perform better load balancing than one without over-decomposition. This superiority stems from the fact that a larger number of smaller-size VPs can be distributed more evenly than a smaller number of large-size VPs. The underlying VP system can also use transparent migration to perform global resource management for improving overall system performance.

In summary, transparent VP migration frees application programmers from the burden of monitoring physical resources and dynamic changes in processor availability. Programmers can think and code solely in terms of the parallelism within their application. In addition to reducing the complexity of application programming, the underlying VP system can use transparent migration for better load-balancing and global resource management.

1.3 The case for supporting process-based, legacy applications

There already exists a large body of process-based message-passing applications written in languages such as FORTRAN and C and extensively used on DMMPs. In addition, many programming language specific support libraries and tools are available that make parallel programming an easier task. Requiring the applications to be re-coded to take advantage of the features of a new VP system can be impractical in many cases. The application may be too large and complicated or the original application writers may no longer be available. To support this legacy of existing tools and programs, our approach must be programming-language independent while supporting process-based applications.

1.4 Thesis statement and contribution

Our thesis is that message passing systems that support light-weight and transparently migratable VPs can be used to ease the programming complexity of unobtrusively achieving high application performance in multi-user DMMP environments.

We examine current approaches to building VP systems and show how they fail to meet the goals of this research. We define the ULP abstraction, a new kind of VP that is compatible with the process abstraction. This compatibility allows the wide body of existing process-based applications and application programmers to use ULPs with little effort. The most important distinction between a *process* and a ULP is that, unlike a process, a ULP does not define a protection domain. Multiple ULPs of the same application execute within a shared protection domain and the hypothesis is that this characteristic leads to an efficient, light-weight VP implementation.

In order to validate our hypothesis, we have built UPVM, a ULP-based prototype package that supports PVM, a widely used message-passing interface [GS92]. The performance of the ULP-based prototype is evaluated against both micro and application-level benchmarks and a side-by-side comparison with the standard process-based PVM library. The comparison shows that ULPs are viable and that light-weight VP support can be provided in a language-independent, application independent and application-transparent manner.

While the dissertation addresses the mechanisms necessary for implementing the ULP abstraction and migration, it does not deal with the higher-level issues such as policies for allocation of ULPs to processors, finding idle processors, deciding when and where to migrate, etc. Neither do we address processor and network failures. These related issues are addressed elsewhere [GS93, LP93, CK79, HJ86, FZ86, BR94, Bir93].

1.5 Dissertation organization

The remainder of the dissertation is organized as follows. In Chapter 2, we expand on the basic ideas of light-weight VP systems and identify the set of requirements that they need to satisfy in order to be used for message-based parallel computing. We then examine current approaches to building VP systems with respect to these requirements and show how they are unsatisfactory. Finally we propose and introduce the ULP abstraction.

Chapter 3 discusses the general issues and guidelines for designing a ULP system. In Chapter 4 the overall design approach is applied to the design of UPVM, a ULP-system supporting the PVM message passing interface. The implementation of UPVM on a HP workstation network is discussed in Chapter 5. Performance results and a comparison with PVM are presented in Chapter 6. Chapter 7 presents related work. Finally, we conclude and outline the scope for future work in Chapter 8.

Chapter 2

Supporting light-weight, transparently migratable VPs

In the preceding chapter, we laid out the foundation for choosing a set of VP system features for supporting application programmers in multi-user DMMP environments. In this chapter, we first examine common VP abstractions and general approaches to implementing them. We then explain the requirements and goals of our research and argue that these abstractions and VP systems are inappropriate. Finally, we introduce a new VP abstraction, called the ULP, and discuss how it can be used to meet our research goals.

2.1 Common VPs and implementation approaches

A variety of virtual processors have been defined in the literature that model different processor architectures, and depending on the platform on which these VPs are implemented, they offer different trade-offs in functionality versus performance.

With respect to the processor architecture exported, VPs can be categorized into the following general models :

- UMA: A VP abstracts a processor in a shared memory multi-processor machine with uniform memory access times. The Sequent Symmetry [LT88] and the Encore Multimax [Cor87] are examples of shared memory multi-processors.

- NUMA: A VP abstracts a processor in a shared memory multi-processor with a non-uniform memory access times. That is, the concept of local and remote memory is supported such that local memory access time is less than remote memory access time. The BBN Butterfly [Inc87] is an example of a NUMA multi-processor.
- NORMA: A VP abstracts a processor in a multi-processor with no remote memory accesses. Communication among VPs can occur only via message passing. The Intel Paragon, nCube, and NOWs are examples of NORMA multi-processors.
- Hybrid: A VP abstracts a combination of one or more of the architectures above. A common hybrid architecture is a NORMA multi-processor with each processor exporting a UMA model. Such an architecture is exhibited by a network of shared memory multi-processors.

With respect to this classification, a multi-user DMMP can be labeled as a multi-user NORMA architecture in which it is possible for processor loads and the number of processors available to change during application execution. One of the main goals of this research is to hide the complexity of managing this varying number of processors from applications. This support for hiding implies that a VP system on a multi-user NORMA platform must be able to migrate VPs transparently based on processor availability.

In the rest of this section, we define and discuss common VP abstractions. For each abstraction, we identify the processor architecture it models, outline typical implementation approaches, and discuss the inherent costs and programming complexity of supporting a dynamic, transparently migratable VP system using that abstraction.

2.1.1 Process

The process abstraction, as implemented by operating systems such as NX [Int91], DEMOS/MP [PM83], and early versions of UNIX [RT78], abstracts a NORMA processor model.

The typical implementation of processes within an OS is as follows. The NORMA model is realized by mapping the memory of the process as a protected virtual address space and thus a process cannot access outside this space. The process address space contains its code, data, and stack segments. The stack is used for procedure-based execution of the code segment. In order to simulate the execution of a process, the OS sets the current processor address context to that of the process and executes the process code in that context.

In the presence of multiple users, the integrity of the OS has to be preserved so that the process abstraction is uncompromised. For this purpose, three important issues have to be taken care of. First, the OS code and data must be protected from inadvertent or malicious accesses resulting from process execution. Second, there must be way of regaining control of the processor by the OS so that it can schedule another process. Third, there should be a controlled method by which processes can invoke the OS to perform operations such as process creation, termination and IPC.

The protection issue is resolved by placing the OS code and data in a separate protection domain and with a higher privilege. Process code is always executed at lower privilege.

The issue of regaining control is achieved by scheduling a hardware timer interrupt before executing the process code. Thus, even if the process code contains an infinite loop, the timer interrupt transfers control to the OS interrupt-handling code that can then schedule and dispatch a new process for execution.

The OS invocation issue is resolved by providing processes with access to special procedure calls called system calls. System calls typically cause a hardware trap to occur that causes a change in execution privilege to that of the OS. Code within the OS that is associated with the hardware trap is then executed. In this way, the OS code can now execute in the context of the process. Typically a kernel stack is also allocated by the OS to each process and it is this stack that is used by OS code to execute system calls on behalf of the process. Since a process stack is used for executing the process code, a switching of stacks occurs during the execution of a system call.

In order to preserve the abstraction of a dedicated processor on a real shared processor, the OS implements context switching for processes. This operation potentially involves updating the processor memory management hardware (TLB, mapping registers, cache) in addition to saving and restoring the register state. When to perform context switching and which process to schedule is decided by a scheduling strategy that is implemented within the OS.

Given this OS implementation of the process abstraction, several observations can be made regarding the complexity and cost of process-related operations. First, all process-related operations need to invoke the OS via system calls. Thus on system-call invocation, OS processes incur the overhead of switching privilege and protection domains, switching stacks, and potential copying between protected address spaces. Because processes are protected from one another, IPC between processes even on the same hardware (local IPC) requires OS intervention. Typically, IPC requires copying the message first into OS space and then copying it into the destination process. Although, certain virtual memory mapping techniques reduce the need for actual message copying, they are not always applicable [YJR+87].

Second, since the process abstraction and scheduling is implemented within the OS, it is typically not possible to customize the process abstraction or the scheduling policy to the individual needs of applications. Thus an application may incur overhead because of certain features of the process implementation, even though it does not need them.

Third, because of the various cost factors discussed above, context switching also becomes expensive enough that application programmers using processes on a NORMA multi-processor resort to using non-blocking communication primitives for overlapping communication with computation instead of using the much simpler over-decomposition technique.

Finally, given this process abstraction, transparent migration of processes among processors is relatively simple for both shared memory and NORMA multi-processors. On a shared-memory multi-processor, saving the register context of the process on its kernel stack and handing over the process data structure to the destination processor is sufficient to achieve process migration.

On a NORMA multi-processor however, since each processor implements its own virtual address space, the execution context of a process must be captured and explicitly sent over the network to the destination processor. Because the process execution state is clearly captured by the protected address space and the register context, it is easy to see that process state transfer can be achieved in a relatively simple manner. Once the process state is transferred from the source processor to the destination processor, it is as if the process had never executed on the source processor. That is, there are no *residual dependencies* [DO91] resulting from the migration.

Modern operating systems, to support a notion of shared memory among processes, have extended the process abstraction to abstract a hybrid NORMA multi-processor with some amount of shared memory [CHKS86, SUN88, LMKQ89]. In the sequel, this process abstraction that exports this hybrid model is referred to as the *hybrid* process and the simpler process abstraction that exports a NORMA model as a *pure* process.

To support the hybrid process abstraction, modern operating systems provide system calls, in addition to those for pure processes, via which processes can typically create shared memory of certain size and map it into their address spaces [LMKQ89]. Accesses to the shared memory does not require OS intervention. Thus, processes can either perform IPC using system calls or communicate directly through the shared memory. Synchronization primitives are also provided by the OS so that processes can coordinate their shared memory accesses.

Many of the statements made about pure process implementations are still valid for hybrid process implementations. Operations such as process creation, termination, non-shared memory communication, context switching and scheduling are implemented in the OS and require the process to invoke system calls. Thus this approach also incurs the overhead of crossing protection domains for VP operations. Further, implementation of scheduling policy and the process abstraction in the OS precludes customization by applications.

The cost of shared memory accesses depends on whether the OS implements the hybrid process abstraction on UMA multi-processor or NORMA multi-processor hardware. On a UMA multi-processor, once the shared memory is mapped, accesses to it do not require OS intervention and thus communication via shared memory will always be faster than communicating with the cooperating process via system calls. Also, transparent migration of hybrid processes within a UMA multi-processor is similar to that of pure processes, since shared memory is globally accessible to all processors of the UMA multi-processor.

On a NORMA multi-processor, however, supporting shared memory across processors requires software support, because there is no physically shared memory between the processors. The OS can implement a software support layer known as distributed shared memory (DSM) across processors [NL91]. The OS dynamically manipulates page protections and reacts to protection faults to implement the shared memory abstraction. In other words, the DSM layer transforms the NORMA multi-processor essentially into a shared memory multi-processor. Thus transparent re-mapping for hybrid processes is much more complicated than for the pure processes and the performance of the shared memory abstraction depends on the efficiency of the DSM layer and memory access patterns of the processes that use shared memory.

2.1.2 Threads

Threads are simply the abstractions of processors in a UMA multi-processor and thus exports a UMA model of computation. In this model, programs are written as a collection of VPs, all executing and communicating in the context of the same shared memory.

Thread abstractions have been implemented at both OS and user level [ABB+86, RAA+88, Doe87, FM92b]. Typical implementation of a thread in an OS is as follows. Each thread is defined by its processor register state, a stack that is used to execute application code, and the address space in which it executes. Several threads can share the same address space. In order to simulate thread execution, OS code first checks if the current address space is that of the thread. If it is, the OS loads the processor register set and the program counter with the corresponding register values from the thread's context. Such a load operation results in the thread's execution in the appropriate address space. Thus context switching among threads that belong to the same address space does not need to update the memory management hardware and can typically achieve at least an order of magnitude improvement in performance over that of processes [ALBL91]. However, it does require OS intervention to save and restore registers and execute the scheduler code.

Also, just as in the process implementation, the combination of a protected OS, hardware interrupts and system calls are used to preserve the integrity and the functionality of the OS. For all thread operations such as thread creation, suspension, resumption, and termination, the OS has to be invoked through system calls. In addition, thread scheduling and context switching is implemented within the OS. Thus the cost of crossing protection domains (for system calls and context switching) and the lack of customizability exist for OS implementation.

However, unlike a pure process, threads communicate entirely through shared memory without requiring OS intervention. This use of shared memory makes local communication costs the same as that of memory accesses.

The complexity of supporting transparent migration depends on the underlying hardware. On UMA or NUMA hardware, transparent migration is similar to that of processes: save the register state of the source processor on the thread's kernel stack, set the memory management hardware of the destination processor to the thread's address space and load the processor registers from the thread's kernel stack.

On the other hand, NORMA hardware requires more support from the OS. Like the support required for transparent re-mapping of hybrid processes on NORMA hardware, the OS must provide DSM so as to export a shared memory multi-processor abstraction to thread-based applications. Just as in the case of hybrid processes, the performance of the thread application on NORMA multi-processors is directly related to the efficiency of the DSM implementation and the access patterns of threads.

In order to reduce OS intervention, threads have been implemented at the user level, outside the OS, and have shown an order-of-magnitude improvement in performance over their OS counterparts [MSLM91]. Thus all thread operations can be performed without OS intervention. The scheduling policy is customizable because it is part of the user-level thread library. User-level thread libraries have been built that schedule user-level threads on one or more OS VPs [FM92b, BLL88, PKB+91].

Since user-level threads execute on VPs provided by the OS, thread migration deals with the problem of moving the execution of a user-level thread from one OS VP to another. Given an OS that exports a UMA or NUMA model to applications, migrating user-level threads is similar to that of migrating an OS thread from one processor to another on a shared-memory multi-processor.

The disadvantage of user-level thread libraries is that they can be corrupted by errant thread execution. However, since the libraries are outside the OS, the damage is restricted to the address space in which the errant thread executed.

Another disadvantage of user-level approaches is the potential integration problems with the OS. Typically, the OS is unaware of the multiple user-level threads executing above its VPs. It is therefore possible for an entire set of user-level threads to be blocked because of page fault or I/O calls of one user-level thread [ABLL92, MSLM91]. We will revisit integration problems in Chapter 7.

Modern operating systems have extended the thread abstraction to model NORMA multi-processors with UMA nodes. In this hybrid model, parallel computation is viewed as groups of threads communicating through shared memory within the group and using explicit messages for inter-group communication. In the sequel, threads that export this hybrid model will be referred to as *hybrid* threads and those that export a pure UMA model as *pure* threads.

Hybrid threads have been implemented by the OS and by user-level libraries above the OS. The OS implementation is essentially the same as that of pure threads except that, in addition, message communication between threads is implemented and system calls are provided to perform IPC. Thus all thread operations except intra-thread-group communication incurs the overhead of crossing protection and privilege domains. Further, the thread abstraction and scheduling is not customizable because of the implementation in the OS.

Transparent migration of hybrid threads can be discussed under two cases: a) migrating threads within a group and b) migrating threads between groups. Within a thread group that follows the UMA model, transparent migration is similar to that performed for pure threads. However, transparently migrating between different groups is not possible. Since each thread group belongs to a different UMA processor, thread groups have distinct address spaces. Since threads in this model are tied to an address space, migrating between address spaces does not make sense.

Hybrid threads are also supported by user-level libraries to reduce OS-entry costs. The same performance, complexity and OS integration arguments as for pure user-level threads apply here.

2.1.3 Language-based VPs

In addition to the approaches above for implementing VPs that are language independent, languages or run-time systems have been proposed and implemented that either define new kinds of VPs or integrate familiar abstractions such as process or thread into the language or the run-time system. In this section, some representative approaches are discussed.

High Performance FORTRAN (HPF) [For93] is a data-parallel language that provides a master-slave, UMA model of computation. In this model, a HPF program essentially executes as a sequential program except during the execution of a parallel construct. At this point, multiple VPs are allocated to the application and execute the computation specified within the parallel construct in parallel. The same instruction stream is used by all the VPs. The program data is globally accessible to all VPs and thus these VPs are essentially threads. The number of threads required and the data alignment and distribution needed can be specified explicitly by applications. The compiler, for reasons of performance and functionality, may use different techniques to map the computation model to target hardware.

Data Parallel C (DPC) [HAJLQA91], in contrast, exports a master-slave, NUMA, SIMD model of computation. In this model, a DPC program consists of sequential and parallel parts, the sequential part executing on the master VP and the code in the parallel portion is executed by all the slave VPs. Each slave VP has its own memory, and access time for local memory is smaller than that for accessing the memory of another VP (either slave or master). Thus the DPC VPs are essentially threads on a NUMA multi-processor, with one thread having different functionality than all the other threads. The number of VPs and the data decomposition required by the program is specified at compile time. The language has been implemented quite efficiently on both UMA and NORMA multi-processors. Further, the SIMD model has been used to implement transparent re-mapping across NORMA multi-processors with heterogeneous processors.

Amber [CAL+89], COOL [LAJ91], and Emerald [RTL+91, Jul89] are distributed object-based systems and export a NORMA model of computation because of the properties of data encapsulation and message-based invocation exhibited by objects. The VPs defined by these systems are essentially threads that invoke methods on objects in a shared object space. All the implementations are user-level and all VPs of an application that are executing on one UMA processor are located in a common protection domain. Thus, compared to communication between VPs that are in separate protection domains on the same UMA processor, communication between VPs in these object based systems exhibits at least an order of magnitude improvement in performance. Also, in all the three systems, the unit of migration is an object or a collection of objects. Any threads currently executing within the migrating objects are also migrated to the destination as a byproduct of object migration.

2.2 Our Goals and Requirements for VP systems

In the context of the discussion above, the goal of our research is to reduce parallel programming complexity by providing applications with a VP system that exports a dedicated, NORMA multi-processor model of computation and efficiently maps this model on to a multi-user, NORMA multi-processor with dynamically varying number of available processors. Further, the mapping must be realized in a language-independent manner. In this section, we first present the functionality requirements imposed on VP systems by these research goals. Second, the performance criteria for evaluating VP systems are discussed. Finally, we match the requirements and performance criteria with VP systems discussed so far and argue that these systems are inappropriate.

2.2.1 Functionality Requirements

First, because one of our goals is to support existing parallel applications that obey a NORMA computation model, the VP system must support processes and message-based IPC between these processes.

Second, in order to provide a dedicated multi-processor of user-definable size, the VP system must support multi-programming of processes on the target platform. In other words, it must support over-decomposition.

Third, because the processor availability on the target platform can vary dynamically, the VP system, in order to supported a dedicated multi-processor model, must support application-transparent migration of processes. In other words, the VP system must be able to migrate processes away from a processor when it is declared unavailable and utilize newly available processors.

Finally, this functionality must be provided in a language-independent manner so that the VP system can be also used by existing parallel applications written in conventional programming languages such as FORTRAN, C and C^{++} with little effort.

2.2.2 Performance Goals

We divide the performance goals that a VP system should satisfy into static and dynamic cases. The static case is when the migration mechanism in unused. Thus the static case focuses on the overhead incurred by over-decomposition. The dynamic case is when the migration mechanism is exercised.

To encourage programmers to over-decompose their applications, the overhead incurred for over-decomposition should ideally be zero. Our goal is to provide a VP system that has a performance and overheads at least comparable to that of thread abstractions that are implemented outside the OS. Naturally, such a goal implies that the VP system should perform better than an OS-based implementation of the process abstraction. To satisfy this goal, at least the following sub-goals should be satisfied. First, the code in a VP should run at the same rate, when selected, as in a regular OS process. In particular, very little additional overhead should be incurred during normal course of operations. Also, an ensemble of VPs should have similar blocking characteristics to OS-processes. That is, the blocking of one VP should not interfere with the execution of another.

A VP migration mechanism can be used for load-balancing, global resource management, or for unobtrusive computing on multi-user NORMA multi-processors with individually owned processors (i.e., workstation networks). Of these goals, unobtrusive computing makes the most severe demands on the migration mechanism. First, the VP system should be responsive to processor eviction events. Second, the VP migration mechanism should be fast so that VPs are off-loaded from a processor that is declared unavailable.

When there is a conflict between performance and being unobtrusive, the VP system should favour unobtrusiveness. Thus, the primary performance metric of the VP system in the dynamic case is a measure of its unobtrusiveness and overall migration cost.

The unobtrusiveness cost can be divided into its three distinct components:

1. The time elapsed from the instant a migration event is sent to the VP system to the time it detects the migration event. This cost should be in the order of milliseconds.

2. The time elapsed between detection of the event and the instant the VP system actually begins migrating a VP. This cost should be in the order of milli-seconds.

3. The time spent in "off-loading" a VP. The VP state is queued for transfer and not necessarily arrived at the destination. This cost typically depends on the the speed of memory copy on the target processor, so our goal for the off-loading time to be close to the memory copying speed of the processor.

The overall migration cost is the time elapsed from the instant a migration event was received at the source node to the instant a VP is received and put on a scheduling queue at the destination node. Thus migration cost is always greater than or equal to unobtrusiveness cost. We require that the migration cost to be be proportional to the size of the VP state and close to the bandwidth limitations of the underlying network.

2.2.3 Matching VP systems with the requirements and goals

In view of our requirements, thread based systems cannot be used for this research as they do not export a NORMA model of computation. The object-system approaches offer a NORMA model and map VPs to NORMA platforms efficiently. However, to take advantage of their capability, applications have to be written in the language supported by the system. This approach violates the language-independence requirement.

The process abstraction appears to be the right abstraction to use and VP systems that implement the process abstraction (i.e., operating systems and message passing libraries such as PVM, TCGMSG [GBD$^+$93, Har91] that offer a restricted interface to OS processes) satisfy the research requirements. However, these VP systems do not evaluate well with respect to the performance goals. Because of the high over-head of system calls, context switch, and local communication, OS-supported processes are undesirable. With respect to these attributes, thread abstractions mapped to user level offer the best performance but unfortunately do not export a NORMA computation model.

2.3 Our approach

This dissertation presents an approach that combines the low-overhead and multi-threading benefits of user-level implementations of threads with the migration capability and programming model of processes. To this end, a new VP abstraction, the User-Level Process (ULP), is defined. Like a thread, a ULP defines a register context and a stack. However, ULPs differ from threads in that they also define a private data and heap space. ULPs differ from processes in that their data and heap space is not protected from other ULPs of the same application. That is, ULPs do not define a protection domain.

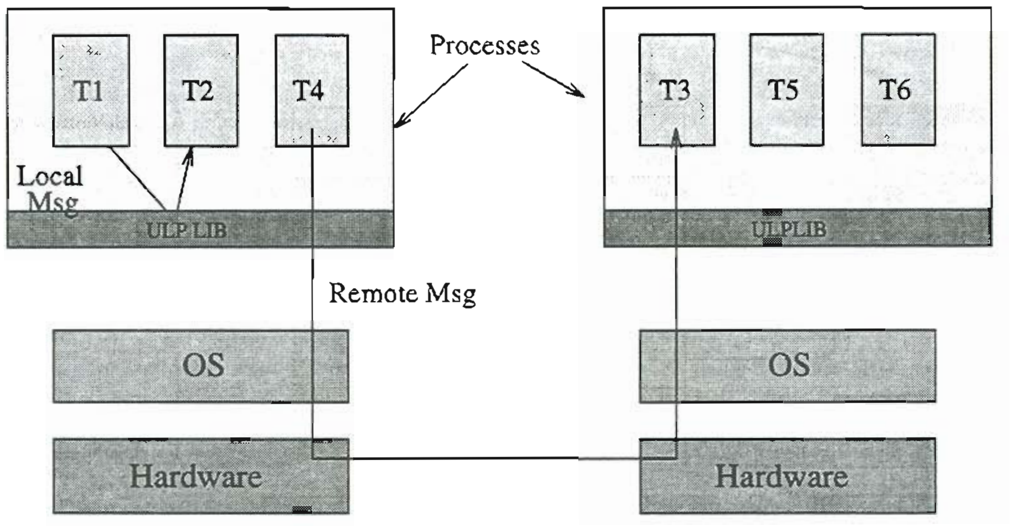

Figure 2.1: ULP System

From the application programmer's perspective, ULPs look like OS processes. Consequently, existing message-based, parallel applications that use processes as their VPs can use ULPs with little modification. From the ULP library's perspective, there are potentially many ULPs per OS process. (See Figure 2.1.) All ULPs within a single OS process are scheduled by the ULP library code that also resides in that process. This means that operations such as ULP context switching and scheduling do not require OS intervention. From the perspective of the OS, there is only one process per application on any given processor. In this way, the parallel programmer's notion of "processor" is virtualized, while maintaining the efficiency of one OS process per physical processor.

The ULP library implements memory management, scheduling and context switching for the ULPs and the VP interface that applications must use in order to use the library. An essential part of the VP interface is the message-passing interface that is used by applications for IPC. ULPs must communicate with each other only via this message passing interface. Thus, because of the explicit nature of this communication, a ULP's address space and context is clearly isolated from that of other ULPs. Because of this isolation, a ULP's migratable state can be easily identified and migrated without the danger of inadvertently causing memory inconsistencies.

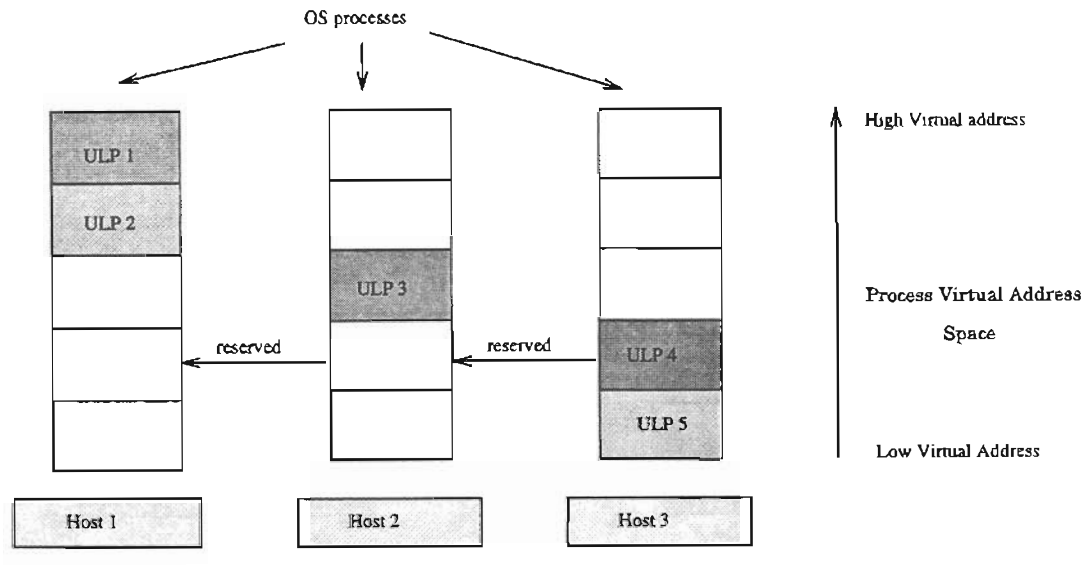

Figure 2.2: Per-Application, network-wide, virtual address space partitioning

A potential problem with migration concerns pointers in the application program. That is, if a ULP is relocated to a different place in the address space of a process, pointers might have to be modified. To eliminate the need for this, the mapping of a ULP to a set of virtual addresses is made unique across all the processes of the application. For example, consider an application that is decomposed into 5 ULPs across 3 processes, one process per host. (See Figure 2.2.) If ULP4 is allocated a virtual address region V1 on Host 3, then V1 is also reserved for ULP4 on all the other hosts which may be targets for migration of ULP4, even though it is not currently present on them. Thus, when ULP4 is migrated from Host 3 to Host 1, it is moved into its reserved slot in the application process on Host 1. Thus no pointers need to be modified. A similar approach is used in the Amber system [CAL+89].

In the rest of this section, we expand on the brief introduction to the ULP approach and discuss several criteria that must be considered in designing and developing a ULP-based system. For each of these criteria, we supply the rationale and point out the research constraints that led to its inclusion.

2.3.1 ULP Programming model

For process-based applications to use a ULP-based system with little to no effort, the programming model exported by a ULP-based system should be that of a process-based programming model. In other words, writing applications that make use of ULPs should be similar to that of writing message-passing applications that make use of OS processes. For this purpose, we require a ULP system to satisfy the following properties.

First, each ULP should provide for variables that are global to all code in the ULP address space, local or automatic variables that exist for the duration of procedure calls, and locally scoped static variables. It should be possible for ULPs to have the same name for a global variable without causing an addressing conflict. That is, each ULP must have a distinct mapping of the global variable name to a memory location.

Second, ULPs should be able to allocate and deallocate memory dynamically and this memory should be private to the ULP.

Finally, when a ULP executes a blocking ULP system call which cannot be satisfied immediately, the ULP system is expected to block the current process and schedule the next runnable ULP, if any. The policy that dictates which ULP is considered "next" is not part of the ULP programming model. The only requirement is that ULPs of the parallel application eventually execute. That is, ULP scheduling must not result in starvation.

From the definition of a ULP, it is easy to see that first two properties are exhibited by any system that faithfully implements the ULP abstraction. Like processes, ULPs define an address space and a single thread of control. Just as with OS processes, ULPs do not impose restrictions on languages used by programmers to write their applications.

However, application programmers need to be aware of certain critical differences between the ULP definition and processes as implemented by current operating systems. First, unlike processes, ULPs do not define a distinct protection domain. It is possible for two or more ULPs of an application to execute within a single protection domain.

The rationale here is that VPs of the same parallel application are cooperating with each other and do not need a protection domain between them unless for bug isolation. The same rationale is used by thread-based programming models, and we see no reason why it cannot be extended to ULPS that belong to the same parallel application. Since the cost of switching protection domains during VP system calls, context switch and local IPC is avoided, there is better potential for performance than in a process-based approach. Programmers using ULPs must be aware that the incorrect execution of one ULP can corrupt the data of another ULP. Note that ULPs of different applications are still protected from one another.

Second, ULPs are targeted for parallel computing and support a location-independent, special-purpose interface when compared to an interface supported by a general purpose, OS-based process implementation. Programs must use only the interface implemented by a ULP library for doing their computation and communication. The rationale for this restriction is that OS processes are general purpose and carry a lot of state that is not necessary for most parallel applications. Further, OS processes allow certain operations that are location-dependent that complicates process migration [BLL91, DO91]. Thus, special-purpose interfaces can result in reducing the size of a ULP's state and simplify the task of ULP migration.

2.3.2 Programming interface

The programming interface to a ULP system can be an existing interface such as TCGMSG, P4, PVM, or NX so that existing applications can be supported, or it can be newly defined which implies that applications need to be written to take advantage of the interface. For example, for this dissertation, we chose to implement a ULP system that supports the PVM programming interface.

The only constraint imposed on a programming interface is that the interface be location-transparent. Message passing must be the only means of communication between ULPs. In other words, the applications must have been designed as a set of VPs that communicate via location-transparent message passing only and cannot share memory. This constraint has the advantage that a) a ULP state is clearly delineated from that of other ULPs b) the explicit ULP communication through message passing allows for clear identification of interaction between ULPs and c) the communication end-points remain valid across ULP migration. These characteristics lead to a simple scheme for ULP migration. See Chapter 3 for more details.

It is possible that an existing interface may have some location-dependent attributes. In this case, the ULP system supports only that portion of the interface that is location-independent. Some changes to application code may be necessary so that the code does not use any of the location-dependent aspects of the programming interface. For example, PVM interface allows applications to specify a target host for task creation. Since such a specification conflicts with global resource management and transparent migration, PVM applications are expected to be modified so that they do not specify a location for task creation.

2.3.3 Supporting transparent ULP migration

Migrating a ULP involves capturing the ULP state on the source node, transferring the ULP state from the source to the destination, and continuing the ULP execution at the destination node. The state of a ULP is defined by its individual context and its message context relative to other ULPs of the application. The individual context is its text, data, heap, and stack segments and its register state. Because of the machine-dependent nature of this context, ULP migration is constrained to occur among homogeneous pools of processors only. That is, once a ULP is created on a processor, it is constrained to migrate among processors of identical architecture. Heterogeneous, language-independent, transparent VP migration is still an open issue (see section 7.3) and is not addressed by this research.

Note that when a ULP migrates, in addition to resuming it at the right program counter value, all messages destined for this ULP should be received and in a way that does not violate the ordering semantics of the message passing interface. No messages should be lost as a result of migration.

2.4 Summary

The goal of this research is to realize a language-independent, VP system that provides applications with a dedicated NORMA multi-processor model of computation while executing on a multi-user NORMA multi-processor with dynamically varying number of available processors.

Current implementation of processes, threads and language-based VP approaches are not suitable either with respect to the functionality requirements or with respect to the performance goals we expect of VP systems. A new VP abstraction called ULP is therefore introduced that combines the potential performance benefits of user-level threads with the programming model of processes.

Chapter 3

ULP System Design

This chapter discusses the general issues in designing a ULP-based system and suggests an overall design approach for satisfying the functional requirements and performance goals mentioned in Chapter 2. In presenting the approach, we discuss the design alternatives considered, the implications for implementation, and rationale for our choices.

The rest of this chapter is divided into three sections based on issues that deal with a) supporting and managing ULPs within an OS process (Section 3.1), b) mapping VP interfaces on top of these ULPs (Section 3.2), and c) migrating ULPs in an application-transparent manner (Section 3.3).

3.1 Supporting and managing ULPs within OS processes

A ULP system must be able to support multiple ULPs within an OS process with low overhead. This low overhead in turn implies that a ULP system must be able to access and manipulate the address space of the process efficiently. The most efficient mode of such access is when the ULP system executes within the same protection domain as the application process. The ULP system is therefore designed to be a library since it is a simple, efficient and portable method of achieving this goal. In this section, we explain and discuss the various issues of implementing the ULP abstraction in a library outside the OS.

3.1.1 Library initialization

Library initialization deals with determining which portions of process virtual address space to use to support ULPs, creating the ULP library's internal data structures, setting up handlers for migration, and converting the initial thread of control of the OS process into the first executing ULP of the application. If the application is allocated more than one processor and the number of ULPs to create initially is known at application startup, then the initialization is also responsible for creating an OS process on each of the allocated processors and creating ULPs within these processes. Since many of these tasks need to be performed before executing application code, program executables need to be created such that at program startup, control is transferred to the ULP library.

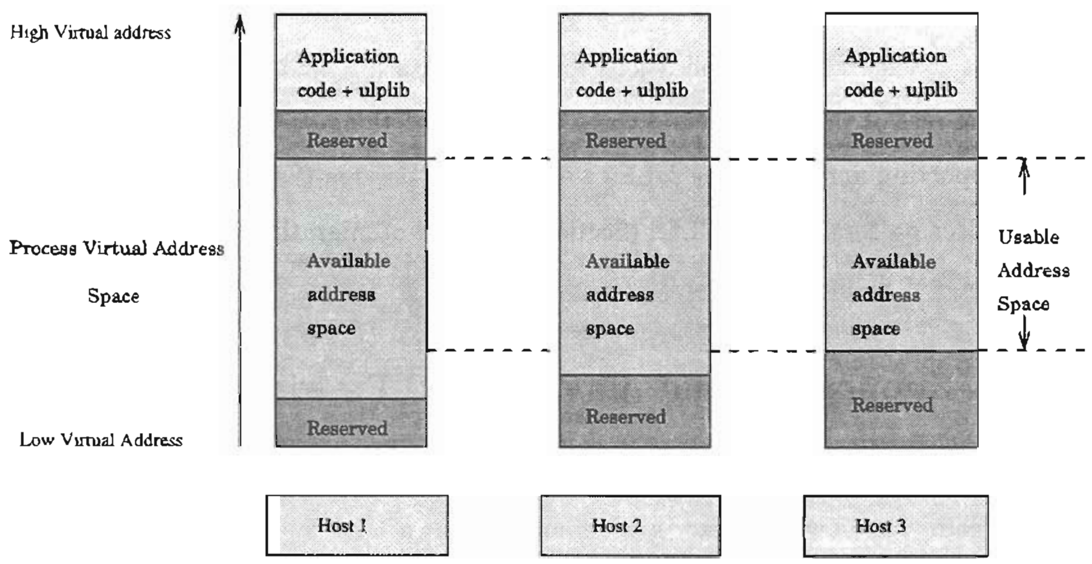

Figure 3.1: Determining usable address space

The available address space within an OS process is the set of virtual addresses that can be used by the application (that is, those which are not reserved by the operating system). An example scenario is shown in Figure 3.1. The ULP library should use this space not only for allocating ULPs but also for its own internal use. Note that for a ULP to be migratable from a process on one processor to a process on another processor, the virtual addresses used by the migrating ULP must be available for application use in the destination process. In other words, ULPs should be allocated only to virtual address ranges that are available across all processes. We refer to this subset as the *usable virtual address space* (see Figure 3.1). For a given pool of homogeneous processors, the usable address space is fixed irrespective of their availability and is assumed to be communicated to the ULP library from an external source. (See Chapter 4.) This usable address space becomes the *global memory pool* from which address space is allocated to ULPs. Note that the access to this pool must be globally consistent. Thus, allocating ULPs from this global memory pool ensures that ULPs can be migrated freely to any processor in the homogeneous processor pool. Note that since ULP migration is possible only among homogeneous processors, ULPs of different processor pools can have overlapping ranges of virtual addresses.

Also note that the focus here is on providing mechanisms in the ULP library and leaving issues regarding policy to be realized by higher-level policy modules. It is assumed that the policy information is made available to the ULP library by some means. (See Chapter 4.) Thus decisions such as determining the number of ULPs to create initially, the number of processors to allocate and the mapping of ULPs to processors must be communicated to the ULP library.

In order to support and manage ULPs, the ULP library maintains several data structures. The main data structures are the ULP descriptor table, ULP run and wait queues, the library heap from which dynamic memory request of the library are serviced, and a process descriptor that contains address space layout of the host process. Once all these ULP library data structures are initialized and ULPs have been created, the bulk of initialization is complete. At this point, the ULP library must prepare itself to receive requests to migrate and accept migrating ULPs. Since these requests need to be handled as quickly as possible, the ULP library must establish *asynchronous* handlers for these requests. In other words, the handlers are invoked immediately when the requests arrive even when the process is not executing in the ULP library. The exact details of how these handlers are registered and invoked depends on the underlying operating system and the communication interface used by ULP libraryṠee details of our implementation in Chapter 5. The final task of initialization is transferring control to the ULP scheduler to dispatch the first ULPs of the application for execution.

In summary, the ULP library initialization involves the following steps:

1. One OS process is created for each processor allocated to the application, then each instance of ULP library goes through the remaining steps given below.
2. Determine the usable virtual address space within the application processes for creating ULPs and ULP library data structures. As mentioned earlier, this information is assumed to be made available to the ULP library by some means.
3. Initialize the ULP library data structures and create ULPs.
4. Register asynchronous handlers for migration events.
5. Transferring control to the ULP scheduler.

3.1.2 ULP creation

Since a ULP is similar to a process, some of the issues in creating a process apply to ULPs. To create a process, an OS generally has to load the code and data segments of the corresponding program, allocate a stack, set up the space for the heap, and initialize the process register and stack state. A ULP library also needs to go through similar steps to create a ULP, however, there are some significant differences.

OS loads the code and data segments of a process into portions of physical address space. In contrast, a ULP library loads the code and data segments of a ULP into portions of virtual address space within a OS process.

In the case of a process, the OS has the freedom to use any region of the physical address space for loading the process code and data segments. All the OS has to do is to set up the memory translation hardware such that virtual addresses generated by an executing process is translated to the right physical addresses.

In contrast, the ULP library may not always have the freedom of using any region of the process virtual address for loading the ULP's code and data segment. If the code

```
0xF000:    ldi 100, r1
0xF004:    subi 1, r1
0xF008:    beq 0xFFF0
..            ...
..            ...
..            ...
..            ...
0xFFF0:       ...
```

Figure 3.2: machine code with fixed virtual addresses

segment contains fixed virtual addresses, then loading the segment blindly into the process virtual addresses space can result in erroneous execution. This problem is illustrated by the code segment in Figure 3.2. The addresses 0xF000, 0xF004, etc., are virtual addresses. Note that the branch instruction has an absolute virtual address (0xFFF0) as its destination. Irrespective of where the code segment is loaded into a process address space, it will always generate the virtual address 0xFFF0 as the destination of the branch instruction. Thus, this code segment will execute correctly only if it is loaded beginning from the virtual address 0xF000 in the process virtual address space.

A similar situation arises with absolute virtual addresses in the code segment that refer to the data segment. Loading a data segment into a different virtual address region other than where the code segment expects can result in erroneous execution.

This problem with dynamic loading is not new and has been addressed in different ways [Jul88, Sab90]. One approach is to modify at load time the absolute virtual addresses in the code based on the virtual address at which the code is loaded. For the example in Figure 3.2, if the code is loaded at location 0x1000 instead of 0xF000, the address 0xFFF0 in the code can be modified to 0x1FF0. This approach corrects the references in the code segment however, notice that we have actually modified code. On some processor architectures that have a separate instruction and a data cache, modifications to instruction stream are treated as data modifications and result in the instructions being placed in the data cache. Therefore, it might be necessary to flush this 'data' out of the data cache before the code segment can be executed.

Although this approach is conceptually simple, the implementation needs to handle all possible types of branch instructions for the target processor and is hence non-portable. Another problem with this approach is that it is not conducive to code sharing among ULPs that use the same program code but operate on different data. To use the same code segment with multiple data segments, the absolute virtual address references to the data segment in the code segment would have to be changed on every context switch. Thus, sharing in this approach is expensive and impractical. Instead of sharing, multiple copies of the same program code can be maintained in different text segments that differ only in the data addresses used.

Another approach is to compile the program in such a way that all data references are relocatable. That is, some means of indirection is used to access data. Typically, the indirection is through a dedicated register. Thus if multiple ULPs share the same program code and access different data, all the ULPs can share a single text segment and on a ULP context switch (discussed later in this chapter), the indirection register needs to be changed so that it points to the beginning of the data segment of the new ULP. Thus this approach is useful for implementing an SPMD style application where all the ULPs share the same code segment but operate on different data segments. However, this approach has the limitation that code segments are not relocatable and thus cannot be loaded freely into the process address space. Such a restriction creates problems for supporting true program parallelism, that is ULPs having different text and data segments.

Finally, true program parallelism can be supported by make both code and data position-independent by using compilers to generate *position-independent code* (PIC). This approach is used to implement shared libraries on modern operating systems [Sab90]. PIC allows the code and data segments to be loaded anywhere in the virtual address space and the indirection approach described above can be extended to handle the switching of code segments.

Any one of the approaches can be used by a ULP system depending on the application domain. We do not suggest any particular approach.

So far we have talked about the issues of assuring that a loaded ULP accesses its own data and code correctly. However, there are more issues to be resolved. Since there is only one instance of the ULP library per process, all unresolved code and data references in the ULP text segment and exported by ULP library must be resolved. Examples of such references are the functions in the message passing interface implemented by the ULP library. This *dynamic linking* can be done at load-time or when the unresolved function is invoked by a ULP during its execution.

Note that the approaches discussed above do not address the problem of one ULP generating illegal address references due to buggy code and accessing or corrupting another ULP's address space. As stated in the introduction to the ULP approach, we consider all ULPs to be part of a single application and inter-ULP protection is not a security issue. However, inter-ULP protection is useful in fault isolation and debugging. Approaches to achieving protection are discussed under related work (Chapter 7).

A brief note on memory allocation. Since each ULP requires its own heap, the ULP library needs to provide its own version of memory management routines (for example, such as **malloc()**, **free()**, and **realloc()**). These routines should be written such that they allocate from the global memory pool and keep track of the memory allocated on a per-ULP basis. Since the memory is allocated from the global pool, addresses allocated to one ULP will not overlap with the addresses allocated to another ULP at any instant of time.

3.1.3 Scheduling and context switching

Scheduling cost is pure overhead and should be kept to the minimum so that most of the CPU cycles are used in executing application code. In order to have low overhead, the scheduler should be invoked only when the currently executing ULP cannot execute further. The rationale for such a non-preemptive policy is that all the ULPs belong to the same parallel application and thus time spent in handling timer interrupts is better spent in executing application code. Software timer-interrupt handling outside the OS is extremely expensive when compared to hardware timer-interrupt handling within the OS. Also, since ULPs of a parallel application communicate with each other, the communication can potentially reduce the skew that may occur because of the non-preemptive scheduling. Further, using a simple non-preemptive scheduling policy rather a sophisticated preemptive policy has been shown to be sufficient for most parallel computations [VR88]. For these reasons, the ULP library is expected to implement non-preemptive scheduling of ULPs except when it receives a ULP migration event. In the event the currently executing ULP is to be migrated, then the ULP library, to maintain unobtrusiveness, can preempt the ULP and migrate it to the destination processor.

The presence of a non-preemptive scheduler implies that the ULP scheduling code should be invoked within the ULP library only under one of the following cases:

1. the executing ULP terminates
2. the executing ULP invokes a blocking message passing primitive or a blocking I/O primitive that cannot be satisfied immediately.
3. there is no currently executing ULPs and one or more ULPs are runnable.
4. the ULP execution causes a page fault.

It is clear that the ULP library cannot afford to let the OS process block because of of one ULP's execution, otherwise other runnable ULPs within the OS process cannot be scheduled during this time. Case 1 above is trivial to handle since ULP termination is not a blocking operation. On ULP termination, the control passes back into the ULP library which then has to schedule the next runnable ULP or wait for Case 3 to occur. Note that Case 3 can also occur if the currently executing ULP has been migrated.

For handling Case 2 and Case 4, the ULP library requires different extents of support from the underlying OS. The OS must provide one or more ways of transforming the blocking primitive executed by a ULP into a non-blocking request and one or more ways of checking the status of completion of the request. The ULP library can then provide ULP programmers with primitives with blocking semantics and map these primitives into non-blocking requests at the OS level. If the request cannot be satisfied immediately, the non-blocking primitives return with an error. The ULP library can then insert the executing ULP on a blocked queue and schedule the next runnable ULP. Eventually, when it is known that the request is complete (via the OS-provided mechanisms), the ULP library can make the blocked ULP runnable and insert it on the run queue. Thus, the ULPs are provided with semantics of blocking primitives while the ULP library itself uses non-blocking and associated primitives to multiplex among several ULPs. In this way, the ULP library is similar to an OS that context switches to other VPs when one VP has to blocked for I/O and uses hardware interrupts to recognize that I/O can be performed and eventually reschedules the blocked VP for execution.

Case 4 however requires much more sophisticated form of support from the OS. Note that in Case 2, all primitives invoked by a ULP are trapped by the ULP library and so it therefore has no problem in performing operations discussed above. In contrast, a page fault is typically transparent to OS processes and this transparency in turn means that the ULP library has no knowledge of the occurrence of the page fault. Thus, even though there may be runnable ULPs within the process, the OS has no knowledge of these user-level entities and blocks the process that caused the page fault, swaps in the required page into physical memory, updates the memory management mappings, and reschedules the process for execution. This integration problem between OS and abstractions implemented at user-level has been observed and solutions for better integration have been proposed [ABLL91, MSLM91]. These solutions essentially provide a way to communicate events in the OS such as a page fault to the user-level library. Availability of the support described in these solutions in the OS will allow a ULP library to switch to another ULP on a page-fault event. In the absence of such support, there is nothing that can be done within the ULP library.

The ULP context switching mechanism itself can be separated into two distinct components: saving the state of the current ULP and loading the state of the new ULP. To improve performance, the ULP system can specialize the context-switch code based on the different execution states of a ULP. At least the following cases should be handled:

- If the new ULP has never run before, there are very few registers to 'restore' and hence the number of registers loaded for the new ULP can be minimized.
- The ULP context switch code can take advantage of non-preemptive scheduling and the fact that most operating systems on modern RISC architectures define procedure calling conventions such that only a subset of the register set need be saved and restored. That is, saving the state of a ULP involves saving a subset of the processor registers and restoring the same set for the new ULP.

However if the ULP has been preempted for migration, no specialization is possible. The entire register state of a ULP needs to be stored and transferred to the destination processor. In this case, at the destination, the context switch code should recognize that it is a preempted and migrated ULP and load its entire register state. Luckily, this case only ever follows a migration that is at least an order of magnitude more expensive than context switch and thus the switching cost is relatively negligible.

3.2 Mapping VP interfaces on to ULPs

VP interfaces used for message-based parallel programming consist of a set of functions to perform message passing. In addition, the VP interface may also define functions that deal with creation and control of VPs, a set of functions to perform file and terminal I/O, etc. The VP creation and control functions must be mapped to the core ULP functionality discussed earlier in this chapter. In this section, we discuss a) the mapping of communication end-points of message passing interfaces on to ULPs, b) the interaction of message-passing semantics with ULP scheduling, c) supporting file and device I/O for ULPs and d) the effect of VP interfaces on the implementation complexity of ULP systems.

3.2.1 Mapping the communication end-points of the VP interface

Typically, message-passing interfaces use VP identifiers (VPIDs) as the end-points of communication. For example, a message *send* primitive specifies the destination VPID as one of the arguments to the VP system. However, it is also possible for message passing interfaces to use an abstraction called a *Port* for communication [ABB+86, RAA+88]. A port is essentially a queue of messages and offers a level of indirection between a VP and the ports it acts upon. A single VP may have many ports on which it can receive messages. In a port-based message-passing interface, a port identifier (PORTID) is used as the destination of a message.

Whatever the type of the destination identifier (DID), the ULP library must support the message passing interface at the ULP-level and map the DID to a <destination processor#, OS process identifier> pair followed by enough information to identify the DID uniquely within the destination OS process.

3.2.2 Message-passing semantics and ULP scheduling

The conditions under which a ULP must be blocked depends on the semantics defined by the message passing interface used by the application. For illustrative purposes, consider the MPI message passing interface, which defines a wide variety of blocking and non-blocking communication primitives [MPIf93a, MPIf93b]. Note that there are other message passing models such as active messages [vECGS92] that are neither equivalent nor subsumed by the MPI model. The intent of this section is give a flavour for the interaction between message-passing semantics and scheduling.

MPI defines blocking and non-blocking message semantics for both *send* and *receive* operations. Each of these semantics has multiple variations. We first present the semantics for the blocking operations followed by those for non-blocking operations. For instructional purposes, we use the term *user buffer* to denote a buffer in a VP's address space that is the source or destination of a send or receive operation. The term *system buffer* denotes a buffer within the underlying system to which a VP has no direct access.

Preserving the semantics for blocking send and receive :

Blocking send: A blocking send can be executed in one of four different modes: **buffered**, **synchronous**, **standard** or **ready**. In the buffered mode, the blocking send returns after the message is copied from the user buffer into a system buffer. The send operation is decoupled from the receive operation. Thus, a ULP executing this type of send must not be preempted by the scheduler.

In the synchronous mode, the send operation blocks until a matching receive operation is posted. This mode allows communication to be implemented with a minimum number of message copies and with no intermediate buffering within the ULP library. Note that the scheduler must block the sending ULP if no matching receive has been posted.

In the standard mode, a send operation *may* result in message buffering or might result in blocking the sending ULP until a matching receive has been posted. In other words, a standard send operation can be implemented as a buffered or synchronous send by the ULP library. Since the synchronous mode of the send primitive is cheaper, the ULP library should implement the synchronous-mode send for the the standard-mode send.

In the ready mode, the send operation can be initiated only if a matching receive is already posted. The MPI standard does not define the outcome when a matching receive is not already posted, other than specify that it is erroneous. The ULP system must implement a sensible error-handling scheme for such an event.

Blocking receive: There is only one semantics for a blocking receive: the receive operation returns after the message has been stored into the user buffer. If the message is already present in the system buffers, the message is copied into the specified user buffer and the call returns. Otherwise, the ULP executing the blocking receive primitive must be blocked until the requested message arrives and has been transferred into a user buffer in the ULP's data space.

Preserving semantics for non-blocking send and receive :

Non-blocking communication allows a VP to continue execution in the presence of a potential blocking operation. In general terms, a VP posts a *communication_start* request the allows communication to happen potentially in parallel with the VP's computation. The VP can check for the completion of the request through a *communication_complete* primitive.

This non-blocking communication implies that the ULP library needs to invoke the ULP scheduler only on the execution of a communication-complete primitive, and block the executing ULP only under certain conditions. Some of these conditions are discussed below in the context of non-blocking send and receive primitives.

Non-blocking send In MPI, a nonblocking "send-start" primitive can use the same four modes as the blocking send primitive: buffered, synchronous, standard and ready and the behaviour of these primitives is explained below.

All the variants of non-blocking send, with the exception of the ready-mode send, should always complete successfully irrespective of the execution state of other VPs or whether the user buffer has been copied into a system buffer.

Therefore in the case of buffered, synchronous and standard modes of send, an explicit "send-complete" call must be executed by the ULP to assure that the corresponding non-blocking send has completed and the user buffer is free for reuse.

If the communication mode is synchronous, buffered or standard, execution of a send-complete call should result in blocking the executing ULP if a matching receive is not posted or lack of adequate buffer space in the VP system space.

In contrast, the ready-mode send does not require buffering or blocking. The send operation can be initiated only if a matching receive is already posted. If it is, the send operation is performed and call should return successfully to the ULP. Otherwise, the call should simply return with an error status.

Non-blocking receive The non-blocking receive-start call registers a message receive request with the underlying ULP system. A ULP should not be blocked for this call. This call can return before a message is fully received into the user receive buffer. A receive_complete call confirms that the data has been received into the receive buffer. If the requested message is not yet received or available in user space, the executing ULP must be blocked by the ULP library.

Notice that scheduling framework discussed above specifies when a ULP should be blocked or not. However, it does not specify the policy for scheduling another runnable ULP. A particular design can choose a policy of its choice.

The only exception to the purely non-preemptive scheduling scheme is due to the ULP library's requirement to support unobtrusive computation. If the global scheduler sends a migration event to the ULP library while a ULP is executing, the ULP cannot be allowed to continue execution until it blocks. In this case, the ULP library needs the information about which ULPs to migrate and their destinations. It is assumed that this information is made available asynchronously to the ULP library by an external global scheduler entity. The ULP library simply implements the mechanism for transparent migration. If the list of ULPs to be migrated contains the preempted ULP, then the ULP library, after off-loading the specified ULPs, should schedule the next runnable ULP on its run queue.

3.2.3 Handling I/O

VP interfaces do not typically define an interface of their own. Instead, they use the interface supplied by the underlying OS. I/O can be classified as file I/O and terminal I/O. Operating systems provide operations that include open, read, write and close. Both blocking and non-blocking versions of these operations are typically supported. Since these operations are implemented as system calls, an application incurs high overhead when making a large number of reads and writes that involve small numbers of bytes.

To alleviate this problem, modern operating systems typically provide memory-mapped I/O. That is, once a file or device has been opened for reading or writing, the capability returned by the OS can be used to map the opened resource into user address space. Thus instead of read and write system calls, the application can read and write bytes into memory without OS intervention.

Non-blocking I/O is complicated to program and introduces complexity into applications and this introduction is not desirable with respect to the goals of this research. Memory-mapped I/O is relatively simple to use however, maintaining the semantics of memory-mapped I/O in the presence of migration and on conventional NOWs is a difficult task [DO91]. If a ULP maps a device into its address space on one processor and is migrated to another processor, to maintain the semantics of memory-mapped I/O, the ULP library has to provide something similar to DSM. As discussed in Chapter 2, such a DSM could result in performance degradation. We therefore suggest that the ULP system should not support memory mapped I/O and of the basic file and device I/O interface (open, read, write, close, etc), only those operations that exports blocking semantics should be supported. Thus programmers are expected to code their ULPs using blocking message primitives and I/O primitives. When one ULP executes a blocking primitive that cannot be satisfied immediately, the ULP is blocked by the ULP system and the next runnable ULP is scheduled. Further, I/O executes correctly irrespective of the location of a ULP or its migration. In other words, I/O is both location and migration transparent.

3.2.4 Providing a single system image

The single system image (SSI) issue is typically regarded as the property of a software system to create the illusion in the minds of the users that the entire network of processors is effectively a single system, rather than a collection of distinct machines [Tan95]. Based on this definition, the following statements can be made regarding VP systems that support SSI:

- Any VP can communicate with any other VP in the system irrespective of the destination VP's location.
- Irrespective of the processor location of a VP, calls made to the VP system exhibit the same behaviour.
- If a VP's location is changed transparently to the application, this migration will not affect the behaviour of the parallel application except perhaps in performance.
- All resources, such as files and devices, that are accessible to a VP on one processor are accessible to all VPs of the application on all processors via the same resource identifiers. In other words, all resources in the system including VPs are uniquely addressable.
- If additional facilities such as signals, timers, memory-mapped I/O are supported, then the interface defined to use these facilities exhibits the same behaviour irrespective of VP location and migration.

Thus the complexity of developing such a VP system that provides a SSI is dependent on the number of different abstractions defined by its VP interface and the SSI properties of the target platform on which the VP system is being implemented. If the target platform has almost non-existent SSI properties (such as NOW running UNIX), then the VP system has to support SSI for the VP interface entirely at user-level. If the VP interface defines a number of abstractions and facilities, then completely supporting a SSI is an extremely difficult task. This complexity is the reason why process migration systems that have been implemented at user-level, instead of supporting the entire OS interface, define a special-purpose VP interface for which they provide SSI [LLM88, NR94, WZAL93]. However, if the target platform supports SSI (such as a NOW running the Sprite operating system [DO91]), then implementing SSI on top of the OS-provided abstractions becomes a much simpler task.

We therefore suggest that ULP libraries should adopt the approach taken by other user-level systems. That is, offer a SSI with respect to a VP interface that is customized to the application domain. Such an approach simplifies the VP system at the same time improving its portability.

3.3 ULP Migration

In order to be useful for unobtrusive computing, ULP migration should satisfy two criteria. First, migration should be responsive, that is, the latency between the instant a migration event is sent to the ULP library to the instant the ULP library invokes the migration mechanism should be as small as possible (in the order of milliseconds). Second, the ULP migration mechanism should be efficient, that is, the time taken to transfer ULP state of certain size should be close to the transmission time on the underlying network. For example, transferring 1 MB of ULP state on a 10 Mb/sec (1.25 MB/sec) ethernet network should be close to 0.8 seconds.

3.3.1 Responding to migration events

To be responsive, a ULP library should register asynchronous message handlers as discussed in Section 3.1.1. These handlers allow the ULP library to be interrupted and the ULP relocation (migration or receipt) messages to be handled asynchronously to the execution of application code.

Asynchronous handling of migration events can cause inconsistencies if shared and writable data structures exist between code executed during normal execution and that executed by the invocation of the migration-event handler. One approach to avoid inconsistencies is to control the execution of these *critical sections* of code that manipulate shared data structures via semaphores, locks, etc. A critical section is any code segment within the library that includes a transition into an inconsistent state. An example is the manipulation of ULP run queue. A code fragment is given in Figure 3.3. Suppose the function `unext()` is being executed within ULP library and the program counter is at the beginning of statement `s2`. Further suppose that the ULP library now contains a ULP of `uid = 2`. If ULP library is at interrupted before executing s2 and the migration

```
Ulp* unext(Ulpq *q)
{
    Ulp* u = uget( q, ULP_ANY);    /* s1 */
    return u;          /* s2 */
}
```

Figure 3.3: Critical section

message includes ULP 2, then the migration handler will not be able find ULP 2 on any queue. Although it might still be possible to recover from such errors, it leads to complicated code that is difficult to debug. Therefore, if the ULP library is in a critical section, migration should be deferred to the point when the library leaves the critical section.

Note that application program code does not contain any critical sections because it obeys the NORMA model of computation. Assuming that an application spends has a reasonably high computation to communication ratio, a ULP library should, in most cases, handle migration messages immediately.

3.3.2 Choosing a migration protocol

After receiving an asynchronous migration event, a ULP library then has to migrate the ULP[s] within its process to the destination host[s]. Different mechanisms for migration have been proposed and implemented. Some of the mechanisms are implemented within the OS and, with very few restrictions, transparently migrate OS processes [PM83, AF89, The86, Dou90, Zay87]. Implementing migration within the OS has the advantage of direct access to the entire process state. Further, techniques such lazy paging can be implemented with small cost and can significantly improve initial process migration time [Dou90, Zay87]. In the case of a ULP library, the mechanism is implemented at the user level for portability and availability.

Because ULP applications obey a NORMA programming model, ULP migration can be achieved by migrating the ULP's text, data, stack, heap, register context and guarantee that no messages for the migrating ULP are lost, while preserving message ordering semantics. Within this scope, a migration mechanism can be designed as one with or without *residual dependencies* [Dou90]. A migration mechanism exhibits residual dependencies if a source host on which the ULP executed previously continues to serve the ULP even after the ULP has migrated to a different host. For example, message forwarding by the source host after a ULP has migrated to another host is a residual dependency that is due to the migration protocol. Residual dependencies can also occur due to a VP interface that offers a per-processor view of resources. File naming in typical UNIX environments is an excellent example. There are times when one user has different home directories on different workstations. If these workstations are harnessed for parallel processing, then upon ULP migration, requests to open files still need to be sent to the source node to interpret the pathname specified. Residual dependencies therefore may effect the source host's performance and consequently fail to be unobtrusive for the host workstation owner. It is left to the ULP system designers to choose and implement a migration protocol that is most suitable for their computing environment.

A ULP migration protocol can be divided into four major stages.

1. Receiving asynchronous migration events from global scheduler: The global scheduler (GS) (see Section 4.3) is expected to send a migration message directly to the process containing the ULP to be migrated. If the ULP library is in a critical section as described above, migration should be deferred to the point when the library leaves the critical section.

2. Reaching a consistent state: The ULP library should reach a consistent state with regards to messages such that no messages are lost and future messages destined for the migrating ULP are received in a proper manner at the destination. The details of how this state is reached are dependent on the message-passing semantics and a particular migration-protocol design. In Chapter 4, we present how this step is performed for UPVM.

3. ULP state transfer: The ULP state, consisting of text, data, stack, heap, register context, its current execution state, and messages not yet received by the ULP, is

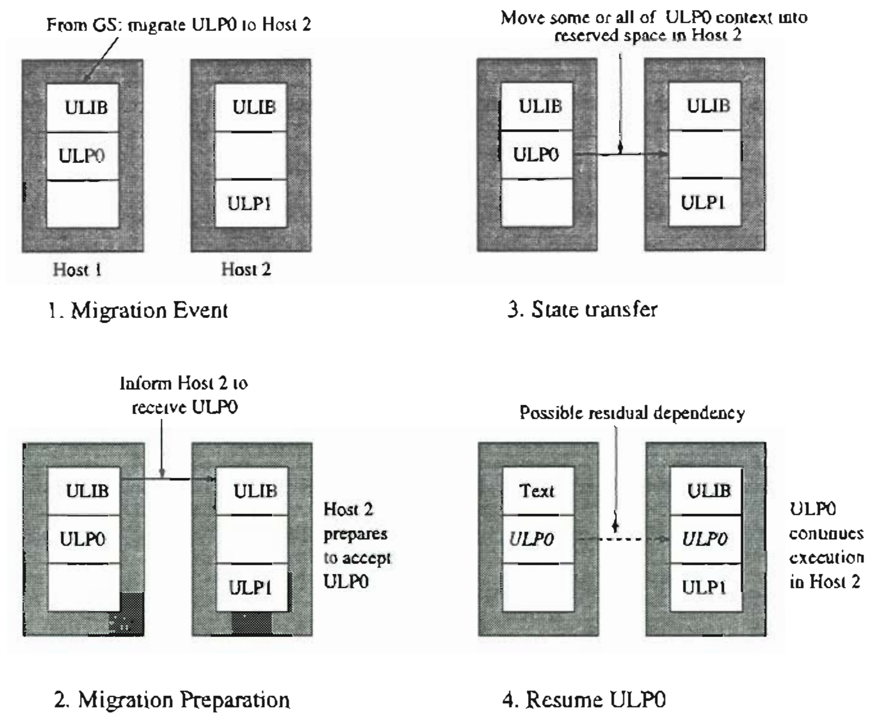
From GS: migrate ULP0 to Host 2
ULIB
ULP0
ULIB
ULP1
Host 1
Host 2
1. Migration Event
Move some or all of ULP0 context into reserved space in Host 2
ULIB
ULP0
ULIB
ULP1
3. State transfer
Inform Host 2 to receive ULP0
ULIB
ULP0
ULIB
ULP1
Host 2 prepares to accept ULP0
2. Migration Preparation
Possible residual dependency
Text
ULP0
ULIB
ULP0
ULP1
ULP0 continues execution in Host 2
4. Resume ULP0

Figure 3.4: Stages in migrating ULP0 from Host 1 to Host 2

sent to the target process. The ULP library in the target process receives the ULP state and places the ULP in its allotted set of virtual address regions.

4. Resuming ULP execution: The ULP is placed in the appropriate scheduler queue (run queue or blocked queue) so that it will eventually execute. Depending on the migration protocol, this stage might involve sending special control messages to other instances of the ULP library or involve the source node of the migrated ULP because of residual dependencies.

3.4 Summary

In this chapter, we discussed the major issues involved in designing a ULP system. The first issue is that of creating and managing ULPs within OS processes. We suggest that ULP system should be designed as a library that is linked to the application program.

When the application is executed, the ULP library code is executed before any of the application code. This initialization code uses the information supplied by the user and external entities such as a global scheduler in determining the number of ULPs to be created initially, the number of processors allocated initially, the mapping of ULPs to processors and the *usable address space* within the OS process within which the ULPs need to be allocated. A variety of approaches can be used to create ULPs and ULP system designers are free to choose an approach depending on the application domain. Finally, as suggested by other researchers, we suggest that simple non-preemptive ULP scheduling be employed because of a) the high cost of implementing preemptive ULP scheduling, b) the fact that all ULPs belong to the same application, and c) inter-ULP communication tends to reduce the skew in the execution of ULPs.

Second, we discussed the issues involved in mapping a VP interface on to ULPs and the effect of target platforms on the complexity of implementing a ULP library. Essentially, the communication end-points of the VP interface should be mapped on to the communication end-points of the underlying OS with some additional protocol. The interplay between message passing semantics and scheduling should be respected and is illustrated by the example on MPI.

As regards to I/O, VP interfaces do not typically define an interface of their own. Instead, they use the interface supplied by the underlying OS. We suggest that the ULP system support that subset of the file and terminal I/O interface that exports blocking semantics. Given a VP interface, the complexity of implementing a single system image (SSI) is directly related to the number of different abstractions the VP interface defines and inversely related to the SSI properties of the target platform. So we suggest that the VP interface be customized to the application domain so that the complexity of the ULP library is reduced at the same time improving the library's portability.

Finally, we discussed the design of the migration mechanism with respect to unobtrusiveness and migration protocols. We suggest that a combination of asynchronous handlers and protected critical sections is the solution for quick detection of migration events. Migration protocols may cause residual dependencies and it is up to the the ULP system designers to choose a protocol that suits their application domain.

With these design issues, possible approaches, and general suggestions in mind, a ULP library for the PVM message passing interface has been designed and is the subject of the next chapter.

Chapter 4

UPVM Design

In this chapter, we apply the general approach to ULP system design described earlier to the design of UPVM, a ULP-based package that supports the Parallel Virtual Machine (PVM) message-passing interface [GBD$^+$93].

The rest of this chapter is organized as follows. Section 4.1 introduces PVM. We then specify, in Section 4.2, the restrictions we impose on PVM applications in UPVM. Section 4.3 presents the design of UPVM, assuming that all PVM applications are SPMD and the number of ULPs needed by the application is known at application startup. Finally, in Section 4.4, we remove these assumptions and discuss how the static, SPMD UPVM design can be extended to handle dynamic, program parallelism.

4.1 PVM

The PVM message-passing interface is designed to permit a network of heterogeneous UNIX computers to be used as a single, large parallel computer. The PVM interface is implemented by the PVM system, which consists of a daemon process (pvmd) that runs on each workstation, and a run-time library (pvmlib) that contains the PVM interface routines. (See Figure 4.1.) The pvmd is responsible for VP creation and control. VPs in PVM are Unix processes (called tasks) linked with the pvmlib. Each task has a unique task identifier (tid) that defines the end points of task-to-task message communication and this is the only means of communication among VPs. Thus, PVM exports a NORMA computation model using OS process as its VPs. In this section, we present a brief overview of the PVM interface as is relevant for UPVM design.

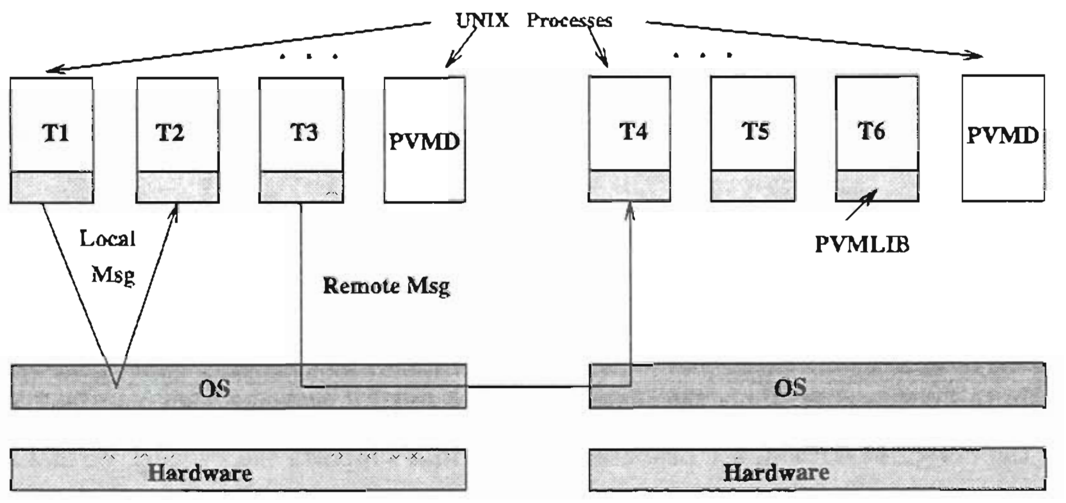

Figure 4.1: PVM system

The PVM interface can be divided into those that deal with task management and those that deal with message communication. The main process management function **pvm_spawn**() creates a PVM task. It takes as arguments the name of an executable program, the arguments to be passed to the program, and where to spawn the program. Thus it is possible in PVM not only to dynamically create tasks but also to specify the location or processor architecture on which a task should execute. This function returns a task identifier (`tid`), which as explained earlier, defines an end point of message communication.

The PVM interface supports the concept of message buffers that are expected to be available within the PVM library. Further, the library is expected to define a default send buffer and a default receive buffer. These default buffers act as implicit arguments to some of the functions defined in the PVM interface. Tasks can refer to buffers by their buffer identifiers (BIDs) only and not by their addresses.

Sending a message has four steps in PVM: 1) allocate and initialize a PVM buffer, 2) make the PVM buffer the default send buffer, 3) Marshall the data values to be sent into the default send buffer (this operation is referred to as packing in PVM terminology), and 4) send the message in the default send buffer to one or more tasks.

A PVM buffer can be allocated and initialized by calling **pvm_mkbuf()** and the buffer can be made the default send buffer by calling **pvm_setsbuf()**. *Packing* routines convert the machine dependent representation of data types into their machine independent representation and insert them into the default send buffer. PVM provides one packing routine for each scalar data type in C and FORTRAN. Finally, the completed (or packed) buffer is sent to another task by calling **pvm_send()** or to multiple tasks by calling **pvm_mcast()**.

A message can be received by calling the blocking receive primitive **pvm_recv()**. The BID of the buffer containing the received message is returned to the application. Further, this buffer is made the default receive buffer. The application then "unpacks" the message by calling suitable unpacking routines that act upon the default receive buffer and perform the converse of the packing operations: they convert the message contents in the machine independent representation to the machine representation of the receiving host. A receive primitive can be invoked to accept the first message received, or a message from a specified `tid`, or a message with a specified `tag`, or a combination of `tid` and `tag`.

Further, applications are provided the guarantee that messages sent by one task to another are received in the same order that they are sent. PVM applications typically take advantage of this message ordering semantics and therefore any alternative software that implements the PVM interface must maintain the same semantics.

4.2 Restrictions on PVM applications

Recall that the goal of our research is to map a dedicated NORMA multiprocessor model on to a NORMA multiprocessor with dynamically varying number of available processors. Such a goal can be achieved only if the exported VP interface is location-transparent. Otherwise, the very purpose of supporting transparent migration is defeated.

The PVM interface, as it is currently defined (PVM 3.2), contains functions and argument options that make processor locations of VPs visible to application code. The call **pvm_spawn**(), as mentioned earlier, allows the application to specify a host for the execution of one or more of its tasks.

The other function that breaks location transparency is **pvm_config**(), which returns the task-to-processor mapping for all the tasks belonging to the application. In standard PVM, the application can assume that these mappings are static and remain valid through the lifetime of its execution. However in UPVM, these mappings may not be static due to transparent migration. Although the PVM interface specification does not make any statements about the validity of the information, there is a possibility that applications may make use of the location information. For example, based upon stale configuration information, it is possible for an application to assume that one of its tasks still executes on a processor and request for a task-spawn on the same processor even though the processor is no longer available and the application's task has been migrated to another processor. In order to avoid such problems, we explicitly forbid the use of the information obtained from **pvm_config**() in performing location-dependent actions.

Also, in order to facilitate an incremental presentation of UPVM's design, we impose two additional restrictions on PVM applications for the moment. First, applications are assumed to have be written in an SPMD (Single Program Multiple Data) style. That is, the application is coded as a single program and is instantiated into multiple VPs. Figure 4.2 shows an SPMD program in C as an example. The master process uses **pvm_spawn** to create clones of itself and distributes data to the clones. Each clone does the same computation on the received data partition and sends the results to the master. The master also does the same computation as the clones. It then receives the results from the clones and does a final processing step and exits. Thus, all the VPs of the application use the same code even though they execute different parts of it.

Second, the applications are assumed to exhibit static parallelism, that is, the number of VPs required by the parallel application must be specified at application startup. In Section 4.4, we discuss extending the UPVM design to support both program and dynamic application parallelism. That is, applications can consist of VPs that use distinct

```
#include    "pvm3.h"
int         mytid, parent_tid, clone_tid[NPROC];
int         data[NPROC * CHUNKSIZE];
int         nprocs = NPROC - 1;

main(int argc, char* argv[])
{
    int         i;
    int         sum=0;
    int         psum=0;
    mytid       = pvm_mytid();
    parent_tid  = pvm_parent();
    if ( parent_tid == PvmNoParent) {
        pvm_spawn(argv[0], &argv[1], 0, "", nprocs, &clone_tid[1]);
        send_data_to_clones();
        do_computation(data, mytid);
        receive_results_from_clones();
        process_results();
    }
    else {
        receive_from_parent();
        do_computation(data, mytid);
        send_results_to_parent();
    }
    pvm_exit();
}
```

Figure 4.2: Example of an SPMD program

code segments and data segments and these VPs can be dynamically spawned.

4.3 Design of UPVM

The overall design of UPVM is summarized in the Figure 4.3. Each PVM task that was mapped to an OS process in vanilla PVM is now mapped to a ULP in UPVM. All ULPs of an application on a processor execute in the context of a single OS process. The ULP library in conjunction with the PVMD, supports the PVM interface and implements memory management, context switching, scheduling and a transparent migration mechanism for the ULPs. The PVMDs are used for task creation and control. The

GS in the figure represents the local representative of a global scheduler to which the UPVM library interfaces. The GS manages the processor pool. It services requests for allocation of new processors, monitors parameters such as processor load, network load, and user activity, and may order applications to migrate their ULPs, either to preserve unobtrusiveness or for load balancing, To perform ULP migration, the mapping of ULPs to processors is also maintained with the GS. Since GS is the processor pool manager, it also maintains information about each processor in the processor pool. Thus, it can determine the usable virtual address space among a set of processors in the processor pool. This information is communicated to the UPVM library during the library initialization before ULP creation.

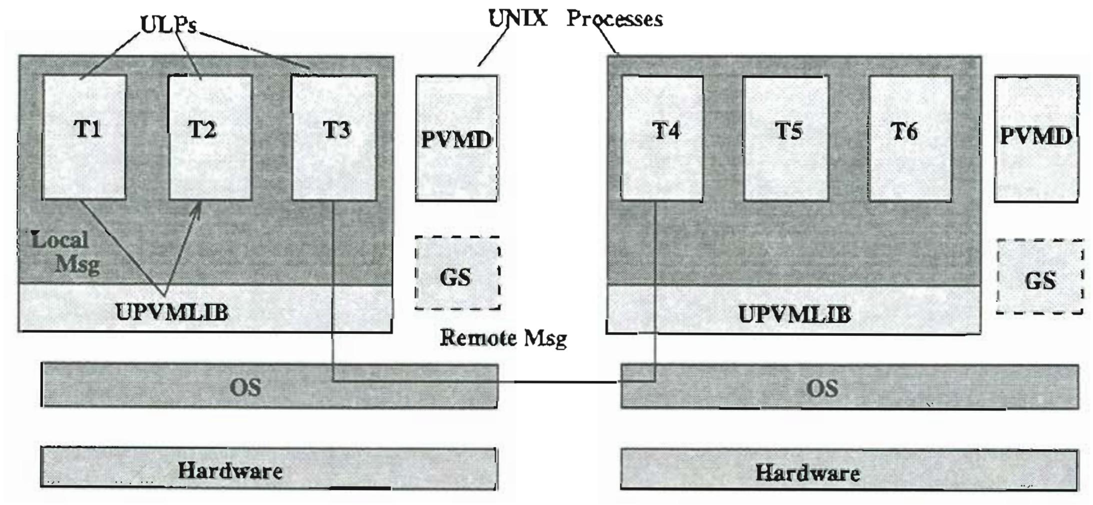

Figure 4.3: UPVM design

To implement the PVM interface, the ULP library does not invoke the OS unless it is an operation that cannot be performed within the context of the ULP library. An example of such an operation is a **pvm_send**() to a ULP that is located within a different process on a different processor.

In the case of inter-ULP communication among the ULPs (local IPC), the ULP library exploits the fact that these ULPs are within the same UNIX process and optimizes the communication. Recall that PVM interface defines the concept of buffers within the library.

In vanilla PVM, a **pvm_send()** is implemented above the OS IPC and consequently for local IPC, it incurs the overhead of invoking the OS, the copying of the buffer into the OS and from the OS into the buffer in the destination process. In contrast, in UPVM, a local IPC is handled by handing over the library buffer to the destination ULP, thus eliminating all extra copying. For this hand-off to work correctly in presence of multiple ULPS performing IPC, the UPVM library maintains a level of indirection between the buffer identifiers visible to the ULPs and the real buffer identifiers within the UPVM library and uses a reference counting scheme for garbage collection. (See Section 4.3.6.)

When a ULP invokes a blocking primitive such as **pvm_recv()** or blocking I/O call, the ULP library maps the blocking primitive to a non-blocking version within the library so that the entire OS process does not block because of one ULP's execution. If the blocking primitive cannot be satisfied immediately, the ULP is put on a blocked queue, and the scheduler is invoked to dispatch the next runnable ULP for execution.

To handle asynchronous events such as completion of I/O, availability of a new message, and a signal to migrate one or more ULPs, the ULP library registers event handlers with the OS. Depending on OS support, another possible asynchronous event is a page fault. Although page faults are caused by code execution and thus defined as synchronous events from the perspective of OS, with respect to the ULP library page faults are asynchronous because of their unpredictability. Based on the event that occurred, the OS will invoke the appropriate handler that had been registered. The handlers may read a message from the OS buffers, change a ULP state from blocked to runnable, migrate a ULP or invoke the ULP scheduling code.

Also, for the purposes of transparent ULP migration, ULP library establishes and maintains disjoint ULP address spaces across all processes of the application. That is, a ULP address space is reserved across all processes of the application even though at any instant, it can exist in one of those processes only. Simply put, moving ULP from one reserved space to another results in migration. This disjoint address space approach resolves the problem of handling pointers in the ULP address space but as discussed in in Chapter 2, does not deal with heterogeneity.

The rest of this section gives a detailed description of the UPVM design. Specifically, we present the various data structures, details of UPVM library initialization, ULP scheduling and context switching, details of how the different functions of the PVM interface are supported, the optimization of local IPC and finally the details of ULP migration.

4.3.1 PVM independent objects and protocols

This section describes the main data structures that are used in ULP management and are not particular to UPVM. These are the ULP descriptor (`ulpd`), the ULP library descriptor (`libd`), the heap descriptor (`Heap`), and the OS process descriptor (`procdesc`).

ULP descriptor

A `ulpd` as the name suggests, is used for maintaining information about a ULP. There is one `ulpd` descriptor per ULP executing within an application and it has the structure shown in Figure 4.4.

A ULP is identified by its `Uid` which is unique within a parallel application. `Uvpid` is the identity of the OS process (PID) within which the ULP is currently resident. When a ULP migrates, `uvpid` changes to the PID of the destination OS process. The `stkbeg` and `udatbeg` identify the beginning of the stack and data segments of this ULP within the process address space. Dynamic memory is provided through the `uheap` object. The structure of the `uheap` object is implementation-dependent. The only requirement is that at any instant of time, a set of memory addresses either belongs to the free pool or is allocated to only one ULP within the parallel application.

`Uentry` contains the entry point to the code executed by this ULP. For SPMD applications, `uentry` of all ULPS will be identical and will point to the `main` of the application program. `Uexit` is used to initialize the return frame of the ULP stack. If a ULP returns from `main()`, then this initialization causes the code at the label specified by `uexit` to be executed. The idea is that this code terminates the ULP cleanly. `Uargc` and `uargv`

```
struct ulpd {
    Ulpid           uid;            /* ULP id */
    VPid            uvpid;          /* OS VP in which the ULP executes */
    Stack           ustack;         /* ULP's stack */
    Data            udatbeg;        /* ULP's data segment */
    Heap            uheap;          /* ULP's heap */
    Vaddr           uentry;         /* ULP's code entry pt, e.g. main() */
    Vaddr           uexit;          /* code label to jump on ULP exit */
    int             uargc;          /* ULP's argc ... */
    char**          uargv;          /* and argv: command line arguments */
    UlpState        ustate          /* ULP's current execution state */
    Event           uev;            /* Reason why the ULP is blocked */
    MachineContext  uregs;          /* Processor register context */
    int             uislocal;       /* true, false, unknown */
    int             uismigrating;   /* true, false */
    int             umiglock;       /* for protecting critical sections */
    Btab            ubtab;          /* for PVM buffers */
    Ulpmq           umsgq;          /* Queue of PVM messages */
    struct ulpd*   urlink;
    struct ulpd*   ullink;
};
typedef struct ulpd    Ulp;
```

Figure 4.4: ULP descriptor

are used to imitate command-line argument passing provided to processes by most operating systems. Ustate specifies the current state of the ULP. A ULP can be in a running, ready-to-run, blocked, migrated, or terminated state. When a ULP is blocked, uev identifies the event for which the ULP is blocked. On a context switch, the machine state of the processor is saved in uregs.

Uislocal identifies whether the ULP is local or remote and is used in implementing inter-ULP communication. (See the algorithms for supporting **pvm-send()** and **pvm-recv()**.) If the ulp is remote then uvpid identifies the host process in which the ULP resides. Thus, if the application consists of N processes, there are N ulpds for each ULP. At any instant of time, only one ulpd of a ULP is marked local and all other ulpds of the ULP are marked remote.

The variable Uismigrating is set during the time the ULP is being migrated. Umiglock

Function prototype	Description
`Ulpid ucreate (Ulpid)`	Create ULP with given Ulpid
`void uterminate (Ulpid)`	Mark ULP as terminated
`UlpState* ugetstate (Ulpid)`	Return ULP's execution state
`void usetstate (Ulpid, UlpState)`	Set ULP state to UlpState
`void uload (Ulpid)`	Load machine context of ULP
`void usetvpid (Ulpid, VPid)`	Set host VPid of given ULP
`int uislocal (Ulpid)`	Check if ULP is local
`void usetlocal (Ulpid)`	Mark the ULP local
`void usetremote (Ulpid)`	Mark the ULP non-local
`char *umalloc (Bytes)`	Like malloc (3C) but acts on per-ULP heap
`char *urealloc (char*, Bytes)`	Like realloc (3C) but acts on per-ULP heap
`void ufree (char*)`	Like free (3C) but acts on per-ULP heap

Table 4.1: Functions exported by a ULP object

is set when migration cannot be done and reset when migration is permitted.

The data structures `ubtab` and `umsgq` within the ULP descriptor directly relate to supporting the PVM interface and are discussed in Section 4.3.3. For now, just note that `ubtab` maintains information regarding PVM buffers accessible to the ULP and that the ULP message queue `umsgq` contains messages destined for the ULP but not yet requested by the ULP.

The main functions exported by a ULP descriptor are given in table 4.1 along with a brief description. Their use will be discussed when discussing UPVM initialization and supporting the PVM interface.

Library descriptor

The library descriptor is the central object of UPVM and its structure is shown in Figure 4.5. Every object of UPVM can be accessed through this descriptor. There is one such data structure for each instance of the UPVM library. In other words, if an application comprises N OS processes excluding the daemons, there are N `Libd` data structures, one per OS process.

`Ltab` and `llen` fields in `Libd` are the ULP table and its size respectively. The ULP table is essentially an array of ULP descriptors. At any instant of time, a ULP can be active and local in only one ULP table.

```
struct    libd {
    Ulp*            ltab;          /* ULP table */
    int             llen;          /* number of ULPs */
    Ulpq            lrunq;         /* ULP run queue */
    Ulpq            lwtq;          /* ULP wait queue */
    Ulp*            lcurulp;       /* currently running ULP */
    int             lmyvpid;       /* the VPID of host OS process */
    Heap            lheap;         /* for ulibspace management      */
    Procdesc        lproc;         /* attributes of the unix process */
    int             lnvps;         /* No. of application processes */
    VPtab           lvpids;        /* ....and their vpids */
    int             lmiglock;      /* migration is OK if lmiglock == 0 */
    int             lmigsigs;      /* No. of migration events pending */
}
typedef struct libd        Libd;
```

Figure 4.5: Library descriptor

Lrunq contains ULPs that are ready to run and **lwtq** contains ULPs that are blocked within this process waiting for some event to occur. **Lcurulp** denotes the currently executing ULP within the process. On a context switch (Section 4.3.5), **lcurulp** is updated to point to the new ULP.

Lnvps denotes the number of OS processes comprising this application and the PIDs of these processes is maintained in the **lvpids**.

Lheap identifies the heap exclusively for the use of the ULP library. ULP executes dynamic memory allocation requests made in the context of ULP library result in memory being allocated through this structure.

Lproc is used to store attributes that describe the process in which ULPs are created. Mainly it contains information regarding the address space layout of the process and the memory regions that can be used for ULP allocation. This structure is described later in this section.

lmiglock. **lmigsigs**, are used in providing an interruptible yet re-entrant communication interface to an external ULP scheduler.

Function prototype	Description
void hinit(Heap* hp, Vaddr hbeg, int hlen)	Initialize named heap
void* hmalloc(Heap*, Bytes)	allocate memory from heap
void* hrealloc(Heap*, char*, Bytes)	reallocate memory from heap
void hfree(Heap*, void*)	free memory to the heap
void hdisplay(Heap*, unsigned int)	Display heap map and usage
unsigned hleft(Heap*)	number of byes bytes unused in the heap
Vaddr gethmin(Heap*)	the lowest address used in the heap
Vaddr gethmax(Heap*)	the highest address used in the heap

Table 4.2: Interface to a heap object

The heap

A heap provides the interface shown in Table 4.2. A heap is first initialized with the block of memory it is supposed to manage. **hmalloc()**, **hfree()** and **hrealloc()** are similar in behaviour to **malloc()**, **free()** and **realloc()** respectively, except that they operate on the heap object instead of on an implicit process-wide heap. **Gethmin()** and **gethmax()** return the lowest and highest memory address respectively that is currently allocated from this heap. These functions are used in ULP migration to avoid unnecessary data transfers. (See Section 4.3.7.)

The process descriptor

A process descriptor contains information regarding OS process address space availability and direction of stack growth. The intersection of the available address space of all the application processes determines the usable address space from which ULPs can be allocated. Performing this intersection allows ULPs to freely migrate from one OS process to another. The direction of stack growth determines how a ULP's stack frame is initialized prior to the ULP's execution. The structure of this descriptor as well as its interface is implementation dependent and are discussed in Chapter 5.

Library-library protocol

Since UPVM is designed as a user-level library, it needs to invoke host operating system for performing communication among ULPs at different sites and to perform I/O.

Table 4.3: Library-to-library protocol

Message Type	Field-1	Field-2	Field-3
ULP_CONFIGQ	-	-	-
ULP_CONFIG	No. of ULPS	-	-
ULP_STATUSQ	ULP id	-	-
ULP_STATUS	ULP id	-	-
ULP_SEND	Destn ULP	Source ULP	-
ULP_MSGFWD	Destn ULP	Source ULP	-
ULP_SIGNAL	Destn Ulpid	Source ULP	Signal #
ULP_EXIT	ULP id	-	-
ULP_LIBALIVEQ	Source VP id	-	-
ULP_LIBALIVE	Source VP id	-	-

Since the OS does not know of ULPs, a protocol must be built on top of OS communication primitives to communicate between the ULP libraries. Since it is not known until a **pvm_send()** is invoked where a particular PVM buffer has to be sent, a fixed 4-word protocol descriptor is always made part of each message. The first word identifies the type of message and the rest of the fields qualify the message type. The different message types possible are shown in Table 4.3. A '-' indicates that the field is unused for that message type.

For local IPC, the packed buffer is simply handed over to the destination ULP. For remote IPC, the OS primitives are invoked to send the packed buffer to the remote process. Notice that there are two kinds of messages between any two instances of the ULP library. The end-points of some of the messages are the libraries themselves. No part such a message reaches a ULP. ULP_LIBALIVE is one such example. ULP_SEND and ULP_MSGFWD are examples of those messages that have ULPs as their end-points. In both cases, the ULP library must parse an incoming message to identify the type of the message. We will revisit the protocols when discussing the details of supporting the PVM interface in Section 4.3.6.

4.3.2 UPVM Initialization and address space layout

The steps involved in initializing and laying out a UPVM application address space is simpler in UPVM than those presented for the general design framework. The simplicity results from supporting only those applications that are written in SPMD style and for which the number of VPs is known at application startup.

Before any application code gets executed, control is first transferred to the initialization code in the UPVM library. The initialization code performs the following steps:

1. Determine how many processors to allocate initially to the application. A process is created on each of the processors.

   Each instance of the library then goes through the following steps.

2. Determine how many ULPs to create. This number is made available to the library as a command-line argument.

3. Determine the locations and extent of OS process address space available in the process.

4. Among the ranges of virtual addresses available, select those ranges that are available within the OS processes of the application. These ranges are used in ULP allocation and this selection ensures that a ULP created within one process can be moved to any other process. Create a memory pool containing these usable ranges. This memory pool is kept globally consistent. In other words, at any instant of time, a range of addresses can be allocated to only one process of the application.

5. From the global memory pool, allocate space for the library use `ulibspace`. This space is located at the same place for each OS process and is used for ULP management (ULP table, scheduling queues, etc) and for implementing the message passing interface.

6. Create ULPs: Creation involves two steps: determining where to create the ULP and the actual creation of the ULP. The *where* information is again a policy matter.

The library assumes the presence of a ULP allocation module that makes these decisions. Since the application is SPMD and the text is shared among all ULPs, the actual ULP creation involves allocating 'sufficient' memory for the data and stack segments of the ULP from the global memory pool and setting up the ULP's initial execution context (setting up the stack frame, initializing the data segment, and initializing the ULP descriptor in the ULP table).

7. Register handlers for asynchronous migration events: These handlers are invoked when a migration event is raised by an external module such as a global scheduler.

8. Schedule the first runnable ULP from the run queue.

Initialization is completed at this point.

4.3.3 Objects specific to supporting PVM

In order to optimize local IPC while exporting the concept of PVM buffers to applications, UPVM defines two main objects: `Btab` and `Gbdesc`. Both these objects provide information related to PVM buffers. However, each offers a different view of the information. The `Gbdesc` object maintains a process-wide view of PVM buffers and provides the interface shown in Table 4.4. The buffer IDs used to access the `Gbdesc` object are *global* buffer IDs or `gbids`. Given a `gbid`, there can only be one PVM buffer within the entire process with that id.

On the other hand, `bptab` is used to provide a per-ULP, local view of PVM buffers by providing *local* buffer ads or `lbids` to ULPs. There is one `Btab` object per ULP named `ubtab`.

`Bptab` contains information about the mapping of each allocated `lbid` to a `gbid`. In addition, associated with each `libd` is the ULP's view of the encoded and decoded state of the buffer.

More than one `lbid` can be mapped to the same `gbid`. The reference count associated with a `gbid` denotes the number of `lbids` that are mapped to that `gbid`. The access to

Function prototype	Description
Bufid gnew()	Allocate new buffer and new buffer id
void gfree(Bufid global_bid)	Free buffer and buffer id
void ginitref(Bufid global_bid)	Initialize ref. count of the buffer
int gincref(Bufid global_bid)	Increment ref. count of the buffer
int gdecref(Bufid global_bid)	Decrement ref. count of buffer

Table 4.4: Interface to Gbdesc object

Function prototype	Description
Bufid btbid_new(Btab*)	Allocate a new buffer id
void btbid_free(Btab*, Bufid ubid)	Free buffer id
void btsetgbid(Btab*, int ubid, int gbid)	set ubid-gbid mapping
int btgetgbid(Btab*, int ubid)	get gbid bound to `ubid`
int btgetsgbid(Btab*)	get gbid bound to default send buffer
int btgetrgbid(Btab*)	get gbid bound to default recv buffer
int btgetsbid(Btab*)	get local, default, send buffer id in
int btsetsbid(Btab*, int ubid)	set local, default, send buffer id
int btgetrbid(Btab*)	get local, default, receive buffer id
int btsetrbid(Btab*, int ubid)	set local, default, recv. buffer id

Table 4.5: Operations on a Btab object

`bptab` is through the interface given in Table 4.5. How these objects are initialized and interact with each other is discussed in Section 5.

4.3.4 Scheduling

The state transition diagram of a ULP in UPVM is given figure 4.6. A ULP is created in the state `UnInit`. When the ULP is scheduled to run, its state changes to `Running`. A ULP's `Running` state changes only under the following conditions:

- a ULP invokes a blocking communication primitive such as `pvm_recv` or a blocking I/O call that cannot be completed immediately, or a page-fault event has been sent to the ULP library indicating that the currently executing ULP has caused a page fault and cannot execute further. In this case the ULP state is changed to `Blocked` and the ULP is put on the blocked queue (`lwtq`). The ULP remains in that state until the requested operation is completed or, in the case of a page fault, until the OS informs the ULP library that the faulted page has been swapped into memory. Then the ULP state is changed to `Runnable` and the ULP is moved to the run queue `lrunq`.

- a ULP terminates. The ULP state is changed to `Exited`.

- UPVM receives an asynchronous migration message to migrate a set of ULPs to another processor. If the current ULP needs to be migrated, its status is changed to `Running+Migrating` during its migration. At the destination processor its state is changed to `Runnable` and it is put on the run queue. Note that a ULP at the time of migration can be in the run queue or the blocked queue. Defining migration state as a qualifier to the actual execution state of ULPs helps in preserving the ULP state across migration.

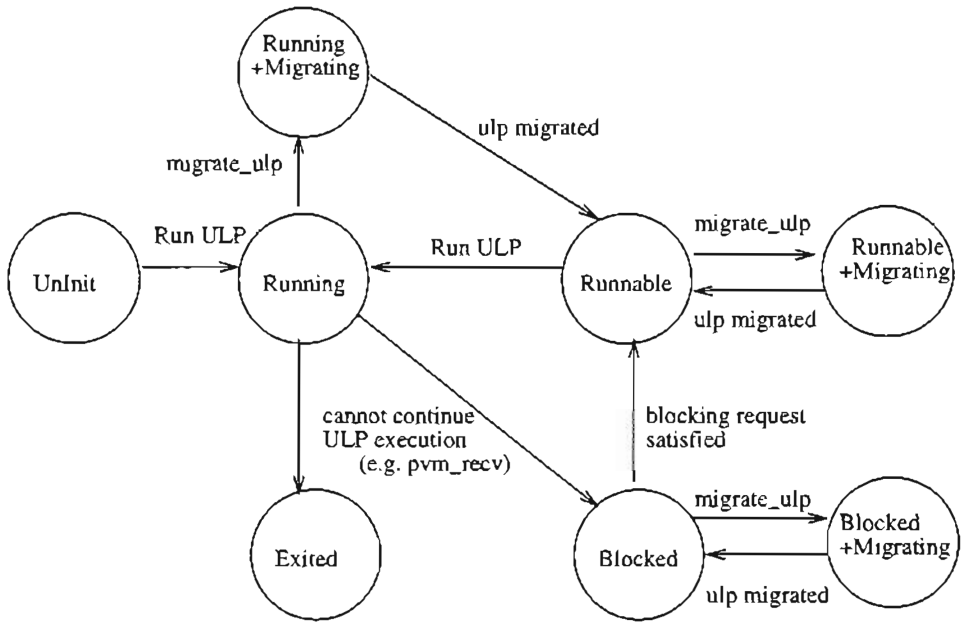

Figure 4.6: ULP state transition diagram

Note that in all the three cases above, the currently executing ULP is no longer able to continue execution. Thus a new runnable ULP, if available, must be scheduled. UPVM performs one of the following operations:

1. If the currently executing ULP invoked a **pvm_recv()** call that cannot be completed immediately and the source ULP specified in the blocking receive is runnable and located on the same processor, then the source ULP is next scheduled for execution (hand-off scheduling [Bla90]). Otherwise go to Step 2.
2. If there are runnable ULPs on the run queue, select the one at head of the queue and dispatch the ULP for execution.
3. If all ULPs of the application have terminated, then perform the application exit protocol to terminate the parallel application.
4. Otherwise, block within UPVM waiting for receipt of new messages or events.

4.3.5 Context switch

The context switch operation is divided into two parts: saving the state of the currently executing ULP and loading the register context associated with the new ULP.

In the absence of migration, context switching occurs only when a ULP terminates or executes a blocking call. In the former case, ULP state is not saved. In the latter case, the register state is saved by the function `int usave(void)`. This function is expected to take advantage of the procedure calling conventions of the host platform to optimize the save time. (See Chapter 5 for context-switch optimization on the HP workstations.)

However, to migrate a ULP that was preempted, the entire register context of the ULP must be saved. The exact method used to save is implementation dependent. We assume that the migration mechanism saves the complete register context of the pre-empted ULP.

Loading the context of a ULP is achieved through the function `void uload (Ulpid id)`. `Uload` is customized to the state of the runnable ULP. If a ULP is going to execute for the first time, then the function **asm_ustart()** is called, which loads only a minimal set of registers (stack pointer, pc, and argument registers). If the ULP is marked runnable, then the function **asm_uload()** is called that makes use of the procedure calling conventions of the target architecture to perform the load. If the ULP is marked

Name	Description
void asm_ustart (Ulp *u)	load registers for first ever execution of a ULP.
void asm_uload (Ulp *u)	load only the registers necessary to preserve procedure calling conventions
void asm_ufullLoad (Ulp *u)	load the entire context of the ULP

Table 4.6: Functions to load ULP state

"migrated while running" then the function **asm_ufullLoad()** is called to load the entire register set. The three load functions are summarized in Table 4.6.

4.3.6 Realizing the PVM interface

In this section we show how different functions of the PVM interface are realized in terms of the data structures and functions described in the previous sections. We will use an example execution scenario of UPVM objects, shown in Figure 4.7, for illustrative purposes. The figure shows the per-process Libd object, which contains all the different UPVM objects accessible in this process. In this example, the Libd object contains 3 ULP objects (ULP 0, ULP 2, ULP 5), the Gbdesc object, a ULP run queue, ULP wait queue, and a heap object for use by the Libd object. Note that each ULP object in turn contains a Btab object, a ULP message queue object, and a heap object. The Gbdesc object contains three process-wide message buffers P1, P2 and P3. The number corresponding to Rc underneath the buffers indicates current reference count for the buffers. An Rc of 2 indicates that the two local bids are mapped to the same process-wide message buffer. In the example, ULP 0's lbid of 1 and ULP 1's lbid of 1 are both mapped to the same process-wide buffer P1.

Enrolling into the PVM environment: pvm_mytid

This function enrolls the ULP into PVM on the first call and returns the uid of the ULP on every call. In UPVM, the ULPs are already created during initialization. This function simply returns the allocated uid of the executing ULP. For example, if ULP 0 executed **pvm_mytid()**, then a zero is returned.

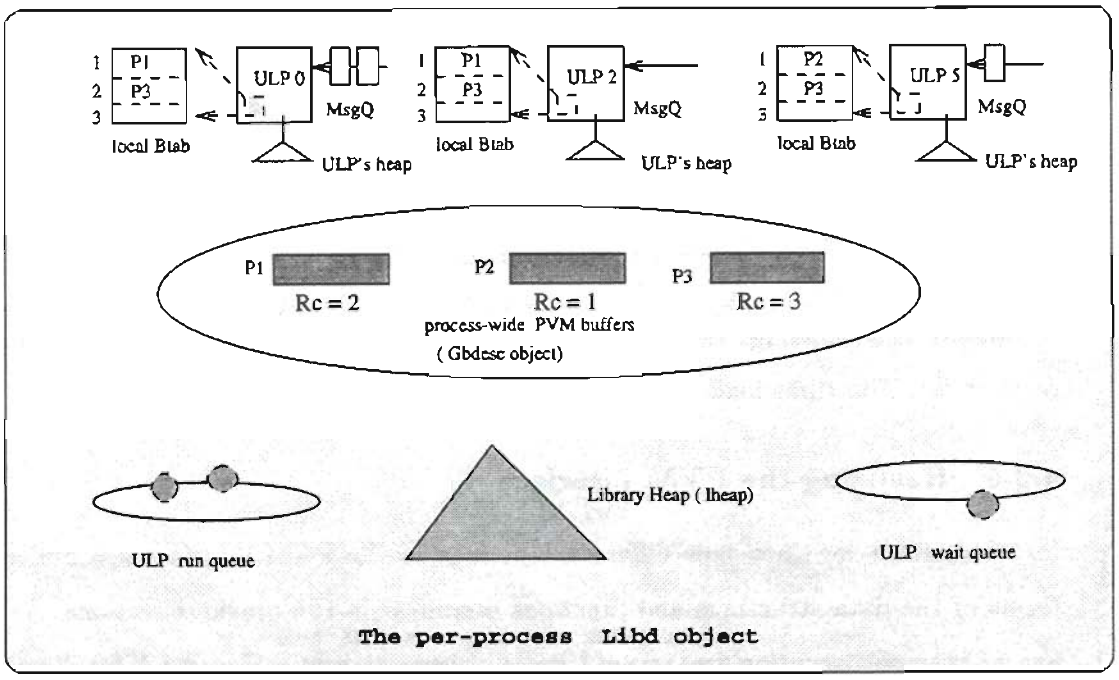

Figure 4.7: An example object execution scenario in UPVM

Allocating message buffers: pvm_mkbuf

The **Pvm_mkbuf**() creates a new message buffer and returns the `bufid` of the buffer. To support this functionality, the function **gnew (encoding)** is first invoked on the `Gbdesc` object to create a new message buffer. Upon buffer creation, **Gnew**() returns a buffer identifier (**gbid**) that is unique within the process. A per-ULP buffer identifier (`lbid`) is then created by invoking **btbid_new**() on the `Btab` of the ULP that executed **pvm_mkbuf**(). A mapping is then established between `lbid` and **gbid** by invoking **btsetgbid**() on the ULP's Btab. Finally, the global buffer's reference count is initialized to the number of mappings to it that currently exist.

For example, suppose that ULP 5 invoked **pvm_mkbuf**() (Figure 4.7). Then the invocation of **gnew**() creates a new buffer with an id, say P4. A unused `lbid` in ULP 3 is then allocated by calling **btbid_new**(). Looking at ULP 3's btab, let us suppose that the `lbid` returned is 3. Then invoking **btsetgbid**() sets the third row of the ULP

Table 4.7: Packing functions in the PVM interface

Name
int pvm_pkbyte (char *xp, int nitem, int stride)
int pvm_pkcplx (char *xp, int nitem, int stride)
int pvm_pkdcplx (char *xp, int nitem, int stride)
int pvm_pkdouble (char *xp, int nitem, int stride)
int pvm_pkfloat (char *xp, int nitem, int stride)
int pvm_pklong (char *xp, int nitem, int stride)
int pvm_pkshort (char *xp, int nitem, int stride)
int pvm_pkstr (char *xp)

3's btab to P4. Finally the reference count of P4 is set to one, the number of mappings that currently exist to the buffer, by calling **ginitref()**.

As described later in this section, maintaining a per-ULP view of "real" message buffers simplifies some of the problems associated with optimizing local IPC among ULPs.

Packing routines (pvm_pk*)

The packing routines pack data into the default send buffer in a machine-independent format. A different packing routine is available for each available data type. For the C language, the routines provided by the PVM interface are shown in Table 4.7.

The packing functions are supported as follows. First, they find the process-wide buffer identifier `gbid` that corresponds to the `lbid` of the default send buffer of the ULP. The `gbid` is then used in packing `nitem` number of the given type into the corresponding buffer. The `stride` is used to choose the next item from the given array `xp`. That is, a stride of one is equal to choosing first `nitem` items from xp, a stride of two corresponds to choosing every alternate item from xp, and so on. If the size of the data is larger than the size of the buffer, the routines allocate memory from the library heap before doing the packing. Thus buffers are contiguous conceptually but are realized as a linked list of contiguous segments.

Table 4.8: Unpacking functions in the PVM interface

Name
int pvm_upkbyte (char *xp, int nitem, int stride)
int pvm_upkcplx (char *xp, int nitem, int stride)
int pvm_upkdcplx (char *xp, int nitem, int stride)
int pvm_upkdouble (char *xp, int nitem, int stride)
int pvm_upkfloat (char *xp, int nitem, int stride)
int pvm_upklong (char *xp, int nitem, int stride)
int pvm_upkshort (char *xp, int nitem, int stride)
int pvm_upkstr (char *xp)

Unpacking routines: (pvm_upk*)

The unpacking routines perform the reverse operation of the packing routines. (See table 4.8.) They unpack data from the default receive buffer and place it starting from address (xp) with a stride of `stride`.

Because of the way local communication is designed, it is possible in UPVM for multiple ULPs to refer to the same message buffer. This potential buffer sharing is the reason why we chose to maintain the current unpacking state on a per-ULP basis within the `btab` object. The unpacking state refers to the number of data items that have already been unpacked from a message buffer by a ULP. Unpacking routines look at this state to figure which data items to unpack next, in a manner similar to that of a **read()** operation on a file that uses the file pointer to figure out where to perform the next read. Thus on a unpacking routine invocation, the ULP's view of the buffer's unpacking status is first accessed. Based on this state, the unpacking is performed, and the new unpacking state for this buffer is updated.

Sending messages: pvm_send

Pvm_send (uid, msgtag) sends the message in the default send buffer to the ULP with the specified **uid**. **Msgtag** is used to label the content of the message. The length of the message sent is implicitly equal to the size of the default send buffer. On return from the call, the default send buffer can be re-used to pack and send more messages. The send buffer retains its contents across successive calls to **pvm_send()** as long as the buffer is not explicitly freed by the ULP.

When a ULP invokes **pvm_send()**, the uid is first examined to check if the destination ULP is within the same address space. If so, a message-descriptor is created and put on the destination ULP's message queue. This message descriptor identifies the source-ULP, the gbid of the process-wide message buffer, the message type, and the length of the message. The reference count of gbid is incremented to record that one more ULP can potentially access gbid. If the destination ULP already executed a **pvm_recv()** for this message and is blocked in the lwtq, that ULP is unblocked and put on the run queue. Note that no actual copying of the message is performed.

In contrast, if the ULP is remote, the ULP communication has to be wrapped as communication between processes since the OS has no knowledge of ULPs. The wrapping affect is achieved by attaching a ULP library-to-library protocol header with values corresponding to the ULP_SEND entry in Table 4.3 to the ULP's message, and invoking an OS-provided communication primitive to send the message to the remote destination. Note that the destination argument to this OS's send primitive will be the PID within which the destination ULP executes.

Releasing a message buffer: pvm_freebuf

pvm_freebuf (int bid) frees the message buffer associated with the specified buffer identifier. However in UPVM, the bid passed to **pvm_freebuf()** is not a process-wide identifier but a local buffer identifier. To support this function, the gbid associated with this bid is first obtained (**btgetgbid (ULPś Btab, bid)**. The reference count associated with gbid is then decremented. If the count is zero, the memory associated with the buffer is returned to libheap. Otherwise, the buffer is retained in the Gbdesc object. Finally, the local buffer identifier, bid, is returned to the invoking ULP's Btab.

Receiving messages: pvm_recv

Pvm_recv (uid, msgtag) blocks the ULP until a message with the label msgtag has arrived from the uid. **Pvm_recv()** then places the message in a new default receive buffer and returns the buffer id of this new buffer. Either one of these parameters can be wild cards.

In UPVM, it is possible for messages to a ULP to arrive before a **pvm_recv()** is invoked by that ULP. Thus, when a **pvm_recv()** is invoked by a ULP, the ULP's message queue `Ulpmq` is first checked to determine if a message that matches the arguments has already arrived. If the match succeeds, a new local buffer identifier (`lbid`) is allocated from the ULP's `ubtab`, the `gbid` specified in the message descriptor is mapped to the `lbid`, and the `lbid` is made the ULP's new active receive buffer.

If there is no match, the ULP's wait descriptor is updated and the ULP is put on the wait queue. Control is then handed over to the UPVM scheduler.

Exiting from the PVM environment: pvm_exit

Pvm_exit() removes the invoking process from the PVM environment. After returning from the call, the ULP can continue to execute just like any other serial process. However, it cannot use the PVM interface until it re-enrolls itself into the UPVM environment by calling **pvm_mytid()**.

This function is supported by qualifying the ULP state as `Upass`, that is, alive but not participating. This state is orthogonal to the states shown in Figure 4.6 and not integrated into the diagram due to space considerations. For example, a ULP can be in a state `Running` and `Upass` simultaneously or it can be `Blocked` (on an I/O) and `Upass` simultaneously. When a ULP invokes the PVM interface when it is qualified by the state `Upass`, the calls simply return an error.

Creating a new ULP: pvm_spawn

For static, SPMD parallelism, this function is essentially a null operation. In section 4.4, we will discuss the design of this function to support dynamic ULP creation in the context of program parallelism.

4.3.7 ULP migration

Because residual dependencies can affect the performance of a host processor and consequently fail to be unobtrusive to the host's owner, the ULP migration mechanism in UPVM is designed to have no residual dependencies. The migration protocol is divided into four major stages. (See Figure 4.8.)

1. Migration event. The global scheduler (GS) sends a migration message directly to the process containing the ULP to be migrated. The process is interrupted and control is transferred to the migration handler **mighdl()** within the ULP library. The migration handler checks if the library is within a critical section. If the ULP library is in a critical section, migration is deferred to the point when the library leaves the critical section. Otherwise, the library reads the migration message, determines which ULPs to migrate, and prepares for ULP migration.

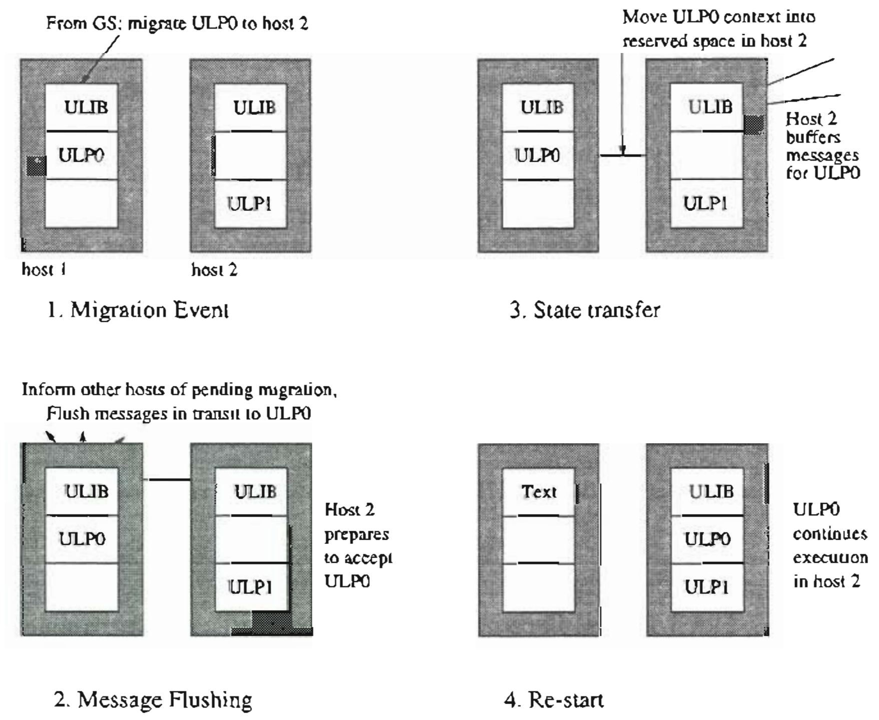
From GS: migrate ULP0 to host 2
ULIB
ULP0
ULIB
ULP1
host 1
host 2
1. Migration Event
Move ULP0 context into reserved space in host 2
ULIB
ULP0
ULIB
ULP1
Host 2 buffers messages for ULP0
3. State transfer
Inform other hosts of pending migration, Flush messages in transit to ULP0
ULIB
ULP0
ULIB
ULP1
Host 2 prepares to accept ULP0
2. Message Flushing
Text
ULIB
ULP0
ULP1
ULP0 continues execution in host 2
4. Re-start

Figure 4.8: Stages in migrating ULP0 from host1 to host2

2. Message flushing. This step ensures that ULPs send all their future messages destined for the migrating ULP to the new destination processor and no in-transit messages are dropped during migration. To achieve this step, a "ULP-migrating" message is broadcast from the source host to all the other hosts allocated to the application. The ULP library on the source waits on a barrier equal to N, where N is one fewer than the number of hosts allocated to the application. Receipt of one "ULP-migrating" acknowledgement message from each of the N hosts allows the blocked ULP library to fall through the barrier and goes to the next step in migration. This barrier scheme relies on the assumption that the receipt of the "ack" message from a host implies that all previous messages from the host have been received. If inter-OS communication primitives do not provide this ordering semantics, then the UPVM library would have to building a message ordering layer on top of the OS primitives and use this layer for message communication.

   Given that the ordering semantics are available, the messages received up until the fall through the barrier are packaged up with the ULP state. All future messages to the migrating ULP are now directly to the new destination.

3. ULP state transfer. The ULP state, consisting of text, data, stack, heap, register context and messages as yet un-received by the ULP, is sent to the target UPVM library. To optimize the heap transfer, the functions **gethmin()** and **gethmax()** are used to obtain the smallest and the largest heap address being used. Only the data within these bounds is transferred. The target UPVM library receives the ULP state and places the ULP in its allotted set of virtual address regions. Further, the messages in the ULP state are added to the front of the message queue at the destination processor so as to preserve PVM message ordering semantics.

4. Restart. The ULP is placed in the appropriate scheduler queue (run queue or blocked queue) so that it will eventually execute. Since the hosts allocated to the application already know the new location of the ULP, no further communication related to ULP migration is needed.

4.4 Supporting program and dynamic parallelism

Program parallelism allows parallel application to be made up of ULPs that differ in their code segments. Dynamic parallelism allows an application to control the degree of application parallelism at run-time. Thus providing the support for dynamically spawning ULPs in UPVM and removing the SPMD constraint is sufficient to accommodate the two kinds of parallelism.

To provide this support, it is clear that the static, SPMD version of UPVM design needs to be extended. However, as can be seen from closer examination, the issues of managing, scheduling and realizing the PVM interface remain unchanged. Mainly, only two areas of UPVM design need to be revisited: supporting a dynamic **pvm_spawn()** and ULP migration.

Supporting a dynamic **pvm_spawn()** in turn maps to dynamic ULP creation. Therefore many of the issues, approaches and trade-offs discussed in Chapter 3 apply here. In terms of the ULP interface, **ucreate()** needs to take in additional arguments: an executable file name and a list of program arguments. The **ucreate()** function goes through one of the following steps:

- If the file has already been loaded in the context of another ULP, allocate memory from the global memory pool and load the data segment. Then perform the same initialization steps as those described for ULP initialization in the SPMD, UPVM design.

- If the file has not been loaded before, then determine the extent of the code and data segments and allocate sufficient memory from the global memory pool. Load the code segment and dynamically link the code with the functions exported by the UPVM library interface. Then load the data segment and initialize the ULP.

ULP migration in the presence of dynamic spawning of ULPs cannot assume that the code segment is present in the destination process because of some other ULP. The ULP migration protocol needs to be extended to check for the presence of code and transfer the code segment only if the code segment is absent at the destination.

Further, the ULP restart may become more complicated if the code segment is migrated. If the UPVM library exists at the same addresses on all processors, then ULP restart reduces to that of the SPMD, UPVM design. Otherwise, the migrated ULP's code segment needs to be re-linked at the destination with the UPVM library. In our approach, we chose load the UPVM library at the same addresses on all processors so as to reduce the restart cost.

4.5 Summary

This chapter described the design of UPVM, a ULP system that supports the PVM message passing interface. The interface has certain location-dependent aspects that are not supported by UPVM because they break location transparency which is essential for supporting transparent migration. The design of UPVM was presented incrementally. Temporarily assuming that applications were SPMD and exhibited static parallelism, the design discussed the objects needed irrespective of the VP interface supported, the PVM-specific structures needed to efficiently support local communication, and ULP scheduling. Further, we showed how the PVM interface can be realized using the data structures and the scheduling infrastructure. ULP migration was then presented assuming an SPMD model. Because the code was the same for all ULPs, the code segment need not be transferred during ULP migration for SPMD applications.

We then extended the SPMD, UPVM design to handle program and dynamic parallelism. Extensions are mainly in ULP creation and migration. ULP creation must now be able to dynamically load and link to ULP code segments. In migrating a ULP, the ULP code segment may need to be transferred if the code segment is not available at the destination. We avoid the potentially re-linking at the restart stage of ULP migration by loading the UPVM library at exactly the same addresses within all OS processes. In the next chapter, we use the UPVM design presented here as the basis for the implementation of our UPVM prototype.

Chapter 5

UPVM Implementation on HP workstations

This chapter describes a prototype implementation of UPVM on a network of HP 9000 series 700 workstations and is divided into two major sections. The first section presents relevant details of the HP workstation and software used to develop the UPVM prototype. The second section describes the main details of the UPVM implementation.

5.1 The implementation platform

The implementation platform was network of HP 9000 series 700 workstations running the HP-UX version 9.03 operating system. In order to implement UPVM, the following major issues had to be resolved:

- How can multiple ULPs be created within the context of a HP-UX process on the HP workstations?
- How can distribution and remote communication be achieved?

For ULP creation, certain characteristics of the HP workstation need to be well understood. These characteristics are discussed in Section 5.1.1.

For achieving distribution and remote communication, two options were considered. One was to use UNIX sockets and remote execution features such as **rsh** and **rexec()**. The other option was to use the standard PVM library released from Oak Ridge National

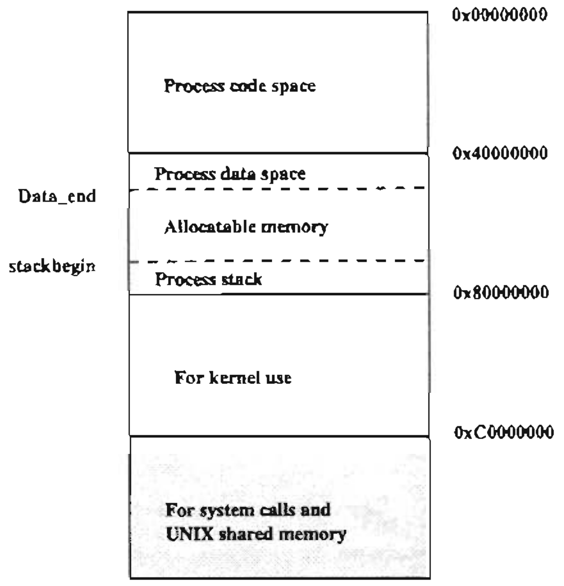

Figure 5.1: Process virtual address space layout in HP-UX 9.0

Laboratory (ORNL). The PVM library was chosen because it simplified UPVM development. Relevant aspects of the PVM library implementation are discussed in Section 5.1.2.

5.1.1 HP 9000/700 series workstations

The HP 9000 series 700 workstations are based on the PA-RISC 1.1 processor [Hew90] and run the HP-UX 9.0.3 operating system.

To determine where ULPs can be created, the usable virtual address space of the process has to be determined. (Recall discussion in Section 3.1.1.) The virtual-address space layout of a HP-UX process is shown in Figure 5.1.

The PA-RISC processor architecture defines a global 64-bit address space that can be accessed using short (32-bit) and long (64-bit) pointers. UNIX processes normally use short pointer addressing. In this mode, the highest-order two bits of the 32-bit address denote the *space register* that supplies up to 32 bits that form the higher word of the 64-bit address. In other words, there are up to 2^{32} 4-GB address spaces possible.

Thus as shown in Figure 5.1, the 32-bit address space of a UNIX process can be divided into 4 quadrants. The text segment is allocated in the first quadrant. The user data segment and stack are allocated in the second quadrant. The kernel stack of the process is allocated in the third quadrant. System call entry points and shared memory segments are allocated in the fourth quadrant.

On the HP workstations, the process stack grows towards higher addresses. Thus the stack can grow from `stackbegin` up to and not including 0x80000000. The value of `stackbegin` on HP-UX 9.0.3 is 0x7b033000.

The HP-UX linker defines two variables `__data_start` and `end` that have special significance. The variable `__data_start` contains value that denotes the beginning of the process data segment. The address of the variable `end` identifies the end of the data segment. In the figure, `Data_end` refers to the address of the `end` variable. Thus the memory addresses between `Data_end` and `stackbegin` are essentially available to UPVM library for ULP and memory allocation.

The HP-UX compilers generate code that references data relative to a base register called the `data pointer` (dp) register. As described later, this indirection is used in the UPVM implementation to support the notion of separate data regions for multiple ULPs executing in a single process address space.

Although the processor defines 32 general purpose registers and 32 floating point registers, not all of them need to be saved on procedure calls. The procedure calling conventions divide the register sets into caller-save and callee-save registers. Knowledge of these registers is used in optimizing the ULP context-switch.

5.1.2 The PVM system

The PVM system from ORNL is built as a layer above the operating system and on a network of UNIX workstations uses sockets and remote execution facilities of UNIX to implement the PVM interface.

As mentioned in Chapter 4, the PVM system consists of a daemon process (pvmd) that runs on each workstation, and a run-time library (pvmlib) that contains the PVM interface routines (Figure 5.2). The pvmd is responsible for VP creation and control. VPs in PVM are Unix processes (called tasks) linked with the pvmlib. Each task has a unique task identifier (tid) that defines the end points of task-to-task communication.

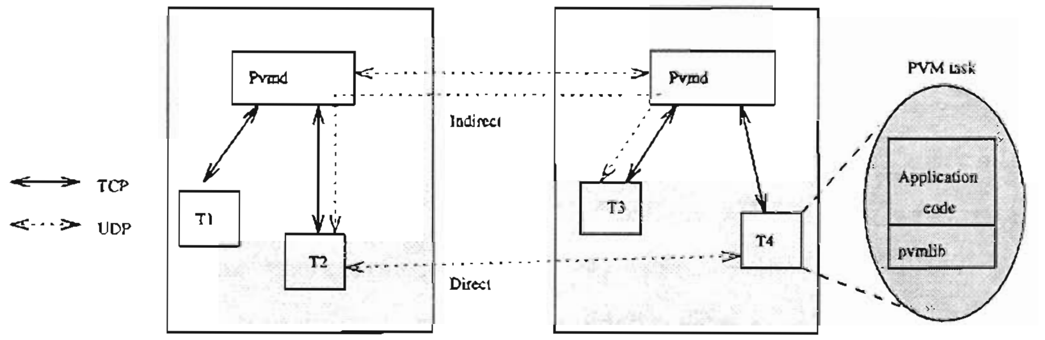

Figure 5.2: PVM implementation structure

To support the PVM interface, the code in pvmlib implements buffer management and communicates with the pvmd and other PVM tasks. Since the UPVM library has to support the same interface, most of the PVM system has been reused in the context of UPVM. In other words, a UPVM system has the structure shown in Figure 5.3. While the pvmds are used unchanged, applications now have to link to *upvmlib* instead of pvmlib.

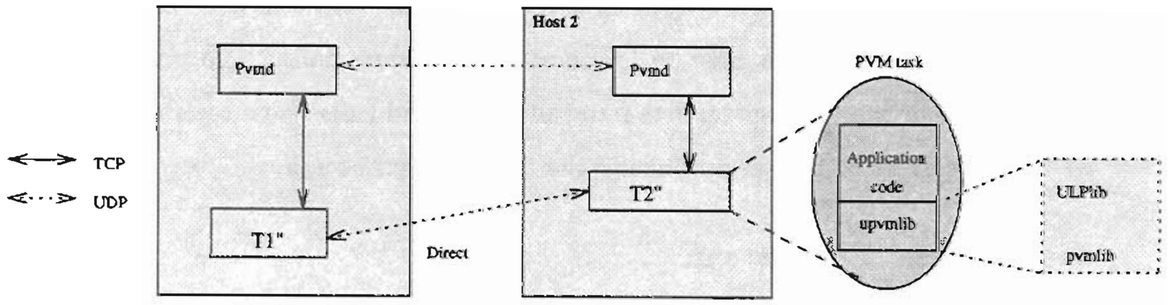

Figure 5.3: UPVM implementation structure

In developing upvmlib itself, most of the pvmlib code was directly reused. However, because of certain assumptions in the pvmlib code, some portions of pvmlib were modified so that pvmlib worked in the context of UPVM. We discuss these assumptions and our modifications in Section 5.2.3.

5.2 UPVM implementation

For the development of the prototype, two assumptions were made regarding PVM applications that greatly simplified the implementation while still allowing us to verify our ideas about ULPs.

First, applications are assumed to have be written in an SPMD (Single Program Multiple Data) style. That is, the application is coded as a single program and is instantiated into multiple VPs. This assumption simplified the design of context switching among ULPs and ULP migration.

Second, the applications are assumed to exhibit static parallelism, that is, the number of VPs required by the parallel application must be specified at application startup. This assumption simplified the problem of ULP creation.

This section is organized as follows. Application compilation is discussed in Section 5.2.1. Details of UPVM initialization and creating the global virtual address space are given in Section 5.2.2. The modifications to pvmlib for use in the context of multiple ULPS are discussed in Section 5.2.3. Context switching details are presented in Section 5.2.4. Finally, ULP migration is described in Section 5.2.5.

5.2.1 Compiling applications

For PVM applications that are SPMD and exhibit static parallelism, no changes are required to the source code to compile the applications with UPVM. By static parallelism, we mean that the number of virtual processors used by the application is either a fixed constant or a variable number that is known at application startup.

To initialize UPVM code before application execution, upvmlib defines its own main() and then calls the application's main(). To prevent linking errors, the application's main() is renamed to Main().

Since ULPs have distinct heaps, the standard memory management routines provided by the OS that assume a single, process-wide heap cannot be used directly. We need to trap all calls to memory management routines and execute the required operation on the right heap. A simple source-level renaming of all memory management routines is not enough since the standard C library contains routines that call `malloc()` internally. So upvmlib supplies its own version of `malloc()` and other memory management routines that are linked to the application program.

However, for applications that are not SPMD and exhibit variable parallelism, source modifications cannot be avoided. PVM applications that are non-SPMD have to be converted first into their SPMD counterparts. In most cases the process is straightforward and includes writing a single wrapper main program, changing the `main()` of each of the independent programs into function names, and compiling all the program files into a single executable.

Also, PVM applications that can spawn a variable number of tasks must be modified such that the number of tasks spawned can be determined and specified as a ULP system argument at application startup.

5.2.2 Initialization and address space layout

A UPVM application is executed by typing

application name [application arguments] [ULP system arguments]

The ULP system arguments provide the initialization code with the number of processors (`nvps`) to use, the number of ULPs (`nulps`) to create on these processors, the size of the virtual address space required (`vspace`), and the size of the space within this `vspace` for message handling and storage. Currently, the mapping of ULPs to processes is a compile-time option for upvmlib. Thus, upvmlib can be compiled so that it maps the given number of ULPs in an interleaved or blocked manner on to the OS processes created.

At application startup, control is first transferred to the `main()` in the UPVM initialization code via `/lib/crt0.o`. The initialization code spawns `nvps` tasks using pvmlib

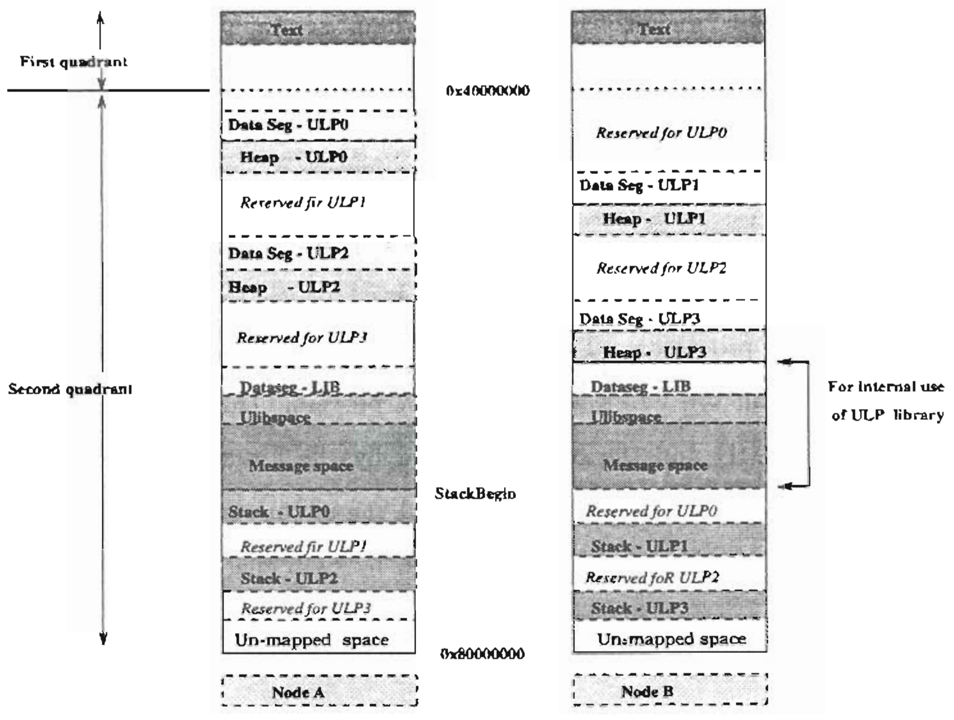

Figure 5.4: ULP layout

code. Each task has the same text segment as the original process (SPMD) and all of them go through the identical steps. Note that as far as the pvmds are concerned, the tasks spawned are PVM tasks.

The UPVM initialization code in each of these PVM tasks has information on the configuration of the virtual machine and the host from which the application was started is recognized as a *home node*. The initialization code in each of these PVM tasks then allocates the specified virtual address space `vspace` using `sbrk()`, determines which ULPs to allocate in that process, and creates and initializes the `libd` descriptor. This initialization is followed by the creation of ULPs (`ucreate`) and the ULPs are placed on the ULP run queue. Each ULP occupies a unique, per application, network-wide virtual memory region as shown in Figure 5.4. The figure shows an application divided into 4 ULPs on 2 processors , 2 ULPs per processor. Note that each ULP occupies a unique set of virtual memory regions. These regions comprises the data segment, stack segment and the heap space of the ULP.

In addition to the space used by the ULPs, there is a certain amount of virtual address space that is used by upvmlib itself. This space(`ulibspace`) (Figure 5.4) contains various ULP tables, memory management data structures and ULP scheduling queues. The same region of virtual address space is used by all the instances of the upvmlib (one per physical processor). This space is not migratable. Note that the code region is shared among all the ULPs within a UNIX process. This sharing is possible as the application is SPMD and the code generated by the hp-ux compilers makes all data references through a designated register called the `datapointer` (gr27). Simply stated, sharing of the same code can be accomplished by switching the datapointer register to point to the data segment of the ULP being scheduled.

When a ULP is created certain variables in its data segment are initialized to point to data structures within the `ulibspace`. These variables can be thought of as type *constant* since, once initialized, they are modified neither by upvmlib nor the application. Examples of such variables are pointers to ULP tables, ULP run queue, ULP blocked queue, etc. These variables allow the ULP library to access its data structures irrespective of the ULP currently scheduled.

The space labeled `msgspace` is used by upvmlib for receiving and sending PVM messages. Since there are multiple ULPs per PVM task, additional information is added to each PVM message so that the message can be de-multiplexed into the current ULP receiving queue in the receiving PVM task.

Once the ULPs are created, the initialization code in each of the PVM tasks sets up signal handlers for handling migrate events that can be sent from the global scheduler (GS). The signal is symbolically SIGMIG.

Finally, control is transferred to the ULP scheduler. Each ULP scheduler then picks the first available ULP from its run queue and dispatches them for execution. It is at this point the application code gets executed.

Except for this initialization step, which is pre-determined, the upvmlib code is invoked during the application execution only under the following conditions.

- A PVM function is invoked.
- A memory management routine is invoked.
- A SIGMIG asynchronously interrupts the host PVM task in which upvmlib has registered migration handlers.

The implementation does not handle page faults because HP-UX does not inform upvmlib about such events. The Support for I/O is under development. (See future work in Chapter 8.)

5.2.3 Invoking pvmlib routines within upvmlib

For code reuse and ease of implementation, upvmlib reuses most of the pvmlib code distributed by ORNL. As shown in Figure 5.3, the upvmlib code can be conceptually thought of as two layers of code: a ulplib layer implemented on top of the vanilla pvmlib code. Implementation-wise however, it was not so simple.

Pvmlib has been designed and implemented to execute within the context of a single UNIX process. The code contains assumptions that the library a) executes within the context of a single data segment and b) only one thread of control is executing within the library at any instant of time.

In the upvmlib environment, the pvmlib code can be invoked from multiple ULPs. Since each ULP has a different data segment, executing the pvmlib code in the ULP's context would cause inconsistencies in the pvmlib's data structures. For example, since there are multiple instances of a global variable (one per ULP context), accessing the global variables from pvmlib from different contexts can yield different values.

Thus for pvmlib code to work correctly in the ULP environment, a separate data segment is allocated for the execution of upvmlib. Specifically, on the invocation of a PVM function, the dp register is changed to point to this data segment, the pvmlib routine is executed, and the original value of dp is restored after the completion of the call. Having such a data segment results in all of the pvmlib data structures being consistent and up-to-date. In the figure 5.4, `ulibspace` identifies this data segment. In terms of instruction counts at user level, executing upvmlib in its own data segment requires three instructions more than that of executing a simple procedure call to upvmlib, which does not require a switching of data segments.

Another minor integration issue deals with the PVM default send and receive buffer identifiers as implemented by pvmlib. These buffer identifiers are defined as global variables in pvmlib and are used by the PVM packing, unpacking, send and receive routines. Since each ULP has its own default send and receive buffers, these global variables are changed at every ULP context switch to the values of the default send and receive buffer ids of the new ULP. The costs of saving these buffer ids is included in the performance analysis of context-switching in the next chapter.

5.2.4 Context switch

Recall from Chapter 4 that context switching is implemented by the functions **usave()** and **uload()** which in turn call the specialized machine-dependent functions **asm_ustart()** and **asm_uload()**, **asm_ufullLoad()**, and **asm_usave()** depending on a ULP's execution state. Such specialization offers the best potential for performance. This sections presents the implementation of these machine dependent functions below.

The function **asm_ustart()** simply initializes the stack-pointer register, data-pointer register, the argument registers, and the return-pointer register based on the information in the ULP's descriptor and branches to the address contained in `uentry`. For C programs, uentry will be `main`. This function is called the very first time a ULP begins to execute.

Asm_uload() restores only the callee-save registers (gr3-gr18, fr0-fr3, fr12-fr21). Since in the common case, ULP context switching occurs voluntarily, the invocation of the context-switching code is treated as a normal procedure call by the compiler and obeys the procedure calling conventions described in the HP Precision Architecture 1.1

manual [Hew90]. Hence the need to restore the callee-save registers. **Asm_usave()** does the converse.

Finally, **asm_ufullLoad()** is used in the case when a ULP has migrated while it was running on another processor. It loads all the 31 general registers and 32 floating point registers.

5.2.5 Migration mechanism

A ULP executing on the UNIX platform has access to the entire system-call interface. In our prototype implementation, we support transparent migration for those processes that use only the PVM interface. UNIX specific functions such as signals, resource usage timers, getpid, etc, are not supported for PVM applications.

All policy issues related to detecting migration points and destination nodes for migration are assumed to be handled by a global scheduler (GS) entity. The migration protocol essentially follows UPVM's design and is divided into four major stages:

1. Migration event. The migration of a ULP is triggered by a migration signal from the GS since it is a simple way of interrupting a UNIX process. Unfortunately one cannot convey additional information apart from the signal itself. Therefore the GS also sends a migration (ULP_MIG) message to the PVM task in which the ULP is located. The ULP_MIG message contains the UID of the ULP and the destination to which the ULP is to be migrated. ULP migration is handled totally within the upvmlib and pvmds are unaware of this migration. Upon receipt of a SIGMIG signal the process is interrupted and control is transferred to the migration handler **mighdl()** within the ULP library. The migration handler checks if the library is within a critical section. If the ULP library is in a critical section, migration is deferred to the point when the library leaves the critical section. Otherwise, the library reads in the migration message and prepares for ULP migration.

2. Message flushing. Recall that in the UPVM's design, an assumption was made that the underlying communication interface used by the PVM library provided message

ordering on a per-node basis. By using pvmlib as the underlying communication interface, the ordering semantics were achieved for free since pvmlib provides such ordering semantics.

3. ULP state transfer. The entire ULP state, consisting of data, stack, heap, register context and messages not yet received by the ULP, is sent to the target process by using packing and send routines defined by pvmlib. The upvmlib in the target process receives the ULP state using receive and unpacking routines provide by pvmlib and places the ULP in its allotted set of virtual address regions.

4. Restart. It is exactly the same as described in UPVM design.

5.3 Summary

In this chapter, we presented the implementation details of UPVM on a network of HP 9000 series 700 workstations running HP-UX 9.0.3 operating system. The implementation runs correctly under the restriction that a global scheduler sends a ULP migration request to a UPVM library only after the previous request is completed. We are currently working on removing this bottleneck.

Several simplifications were made in the prototype development to focus on the validating the idea of ULPs rather than building a full-fledged system. First, only SPMD PVM applications were supported. Second, only static parallelism was supported. Third, the number of processors and the required virtual address space were made available as command-line arguments which the user typed in, rather than upvmlib communicating with the GS in order to determine the number of available processors. Finally, the global scheduling environment was imitated by a small GS program, which when run, sent the specified process a SIGMIG signal and a migration message specifying which ULPs to migrate and their destinations. Thus, we were able to test the unobtrusiveness and migration cost of the ULP system without requiring the development of an infrastructure for scheduling mechanisms and policy. The scheduling policy and mechanisms are being investigated in a related research project by our group [CCG+95].

UPVM is implemented such that PVM tasks that have migrated all their ULPs away do not consume CPU. The tasks remain blocked waiting for a message from the GS.

In the next chapter, we evaluate the performance of the UPVM implementation with respect to both micro-benchmarks and PVM applications.

Chapter 6

Performance Analysis

The performance of the UPVM package is analyzed in two ways. We first present the results of micro-benchmarks for context switch, local communication, and remote communication. The goal is to ascertain the costs of the primitive operations provided by UPVM. We then analyze two applications: a ring communication application that is communication bound, and a two-dimensional grid-based Laplace solver that is computation bound. Finally, the migration performance of UPVM is analyzed by benchmarking a neural-network classifier application.

6.1 Benchmarking environment

Because of limited resources, all experiments were conducted on two HP series 9000/720 workstations that were otherwise idle, connected over a 10Mb/sec Ethernet. Each of the workstations has a PA-RISC 1.1 processor, 64 MB main memory, and is running the HP-UX 9.03 operating system.

6.2 Micro-benchmarks

6.2.1 Context switch

The context switch benchmark measures the average time taken for one VP (an OS process or ULP) to yield to another of the same kind over 10,000 yields. For comparison purposes, the cost of executing a null procedure call on the HP-UX workstation is 0.65

micro-seconds. Table 6.1 gives the context switch cost of ULPs and OS processes, both in absolute time and as a ratio to null procedure call cost. Figure 6.1 shows a bar chart of the context-switching costs.

Type	Cost (micro-seconds)	Ratio
ULP switch	4.74	7.30
UNIX switch	195.00	300.46

Table 6.1: Context switch costs (absolute and relative)

Isolating the process context switch cost in a portable manner is extremely difficult, since there is no equivalent of a yield-to-another-process system call on UNIX. Our solution to this problem was to use Ousterhout's context switch benchmark [Ous90]. In this case, we calculate half the time taken by two UNIX processes to alternately read and write one byte from a pair of pipes. This implies that the UNIX process switch cost given in table 6.1 includes the cost of reading and writing one byte from a pipe in addition to the true process switch costs. However, even if we consider only half of the observed process switch costs, the ULP switch is still more than an order of magnitude faster.

The ULP package performance can be attributed to two factors. First, since the ULPs are within the same OS process, performing system calls is not necessary to yield to another ULP. Second, the ULP package employs hand-off scheduling, which eliminates the latency in scheduling the destination ULP.

6.2.2 Local communication

The local communication benchmark measures the round-trip message communication cost between two VPs, averaged over a large number of trips ($>$ 1000). The cost of packing and unpacking the message is included in this cost. The benchmark is compiled and linked with the PVM library and then with UPVM, yielding two different executables. In the case of PVM, the local communication cost measured is between two UNIX processes on the same node. In the case of UPVM, the cost measured is between two

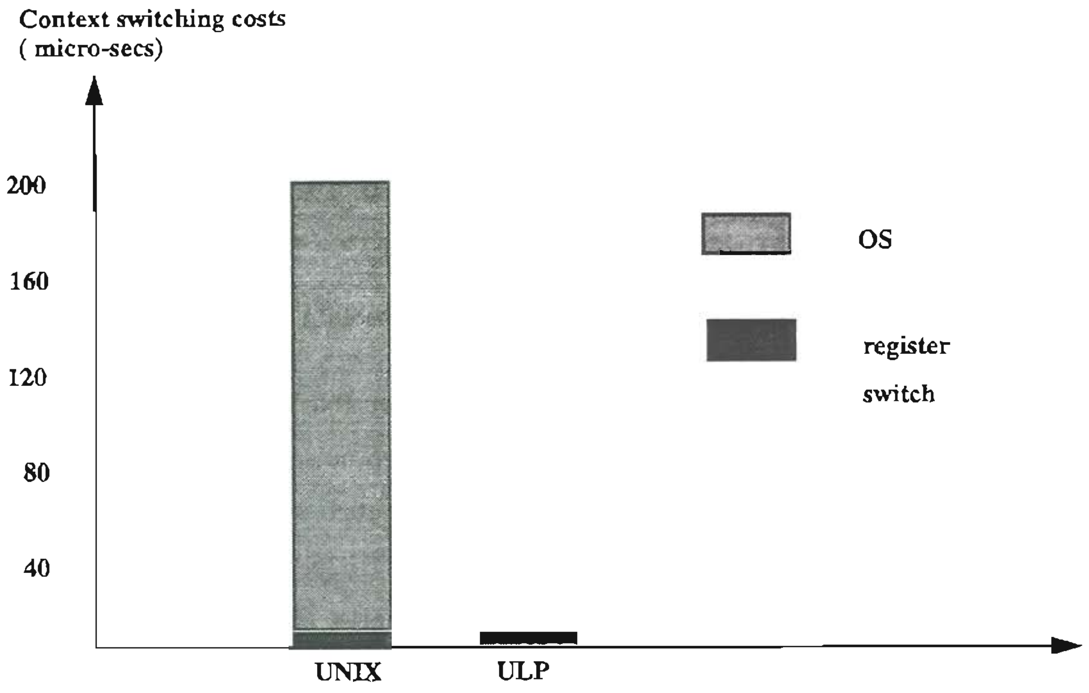

Figure 6.1: UNIX process versus ULP context switching

ULPs that are executing within the same UNIX process. The numbers shown in Table 6.2 and plotted in Figure 6.2 are half the round-trip cost. We assume that this cost closely approximates the one-way communication cost.

Message size(bytes)	PVM(ms)	UPVM(ms)
0	1.40	0.12
1	1.42	0.12
512	1.61	0.14
1000	1.85	0.14
10000	6.55	0.39
100000	47.36	5.55

Table 6.2: Local communication costs

The local communication cost of UPVM is around an order of magnitude better than that of PVM. This improvement can be attributed to two factors: the low ULP context switch costs, and optimized message passing that takes advantage of the shared address space (as described in Chapter 3). In other words, local ULP communication avoids

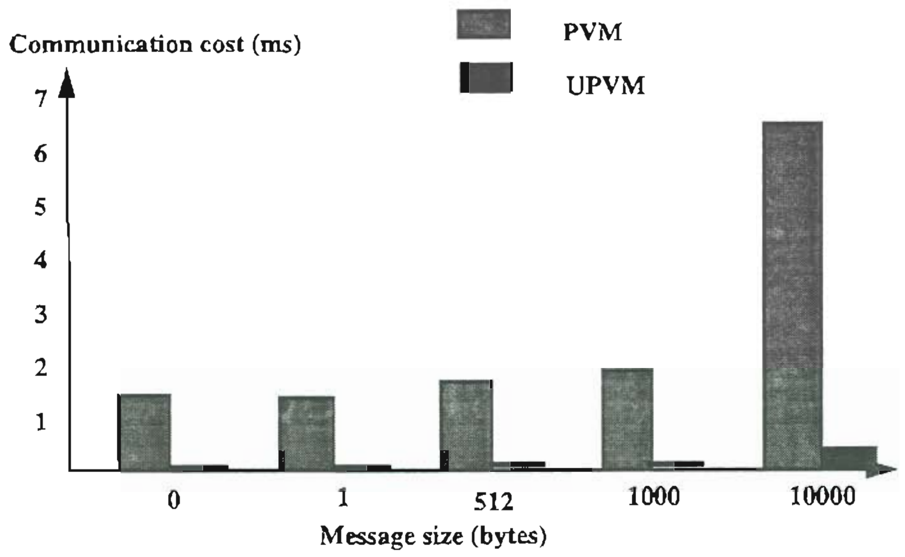

Figure 6.2: Local communications costs

the cost of system call invocation, process context switch, message-buffer copy from the source process into the OS, and message-buffer copy from the OS into the destination process.

6.2.3 Remote communication

The remote communication benchmark is the same program that was used for benchmarking local communication. In this case, the VPs are allocated on different nodes. Again, the costs reported are averaged over 1000 communications. Since UPVM uses PVM for remote communication and treats PVM as a black box as much as possible, we expected a marginal increase in the cost of the remote communication when comparing UPVM to the vanilla PVM.

As seen from Table 6.3 and Figure 6.3, remote communication costs in UPVM are about 3.5 %, 3% and 1% higher than that of PVM for 1K, 10K and 100K message sizes respectively. The overhead is due to a combination of per-ULP buffer table operations, the reference-counting mechanism, a locality check, and some run-time debugging code.

Message size(bytes)	PVM(ms)	UPVM(ms)
0	2.65	2.80
1	2.63	2.80
512	3.35	3.50
1000	4.01	4.15
10000	17.06	17.60
100000	144.70	146.36

Table 6.3: Remote communication costs

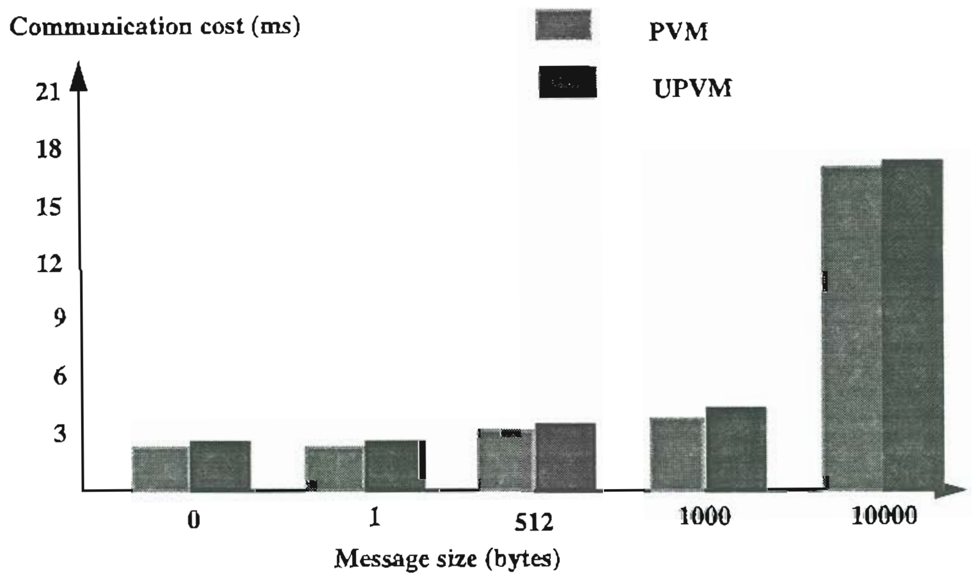

Figure 6.3: Remote communication costs

6.3 Application Benchmarks

In this section, the performance of two PVM applications is analyzed. We chose these applications to examine the two extremes of communication: the ring application performs almost no computation and is always performing communication, and the two-dimensional parallel grid solver with high computation and very little communication.

6.3.1 Ring

The ring program creates a specified number of VPs that then perform ring communication using small (one-integer data item) messages. The time measured is the average time taken by a message to go once around the ring.

The first experiment measures the ring program performance when all VPs are allocated on a single node. The results are shown in Table 6.4 and plotted in Figure 6.4. Since all VP communication is local, the order of magnitude improvement in UPVM performance over PVM is in line with the local communication and context-switch results.

# VPs	PVM(ms)	UPVM(ms)
2	2.55	0.26
4	5.64	0.56
6	8.42	0.82
8	11.16	1.15
10	14.35	1.33
14	21.50	1.90
20	32.86	2.87
24	42.85	3.50

Table 6.4: Ring on one node

The second experiment examines the performance effects of two VP-to-processor allocation strategies, *interleaved* and *block-decomposed.* In the interleaved (**Intlv**) scheme, the application VPs are distributed over two processors such that every inter-VP communication is remote. In other words, VPs that are "neighbours" in the ring are allocated to different processors. Thus, this allocation is a worst-case scenario in UPVM since there is no possibility for optimizing local communication. In contrast, the block-decomposed (**Blk**) allocation scheme takes advantage of the ring communication pattern. The ring of VPs is cut in the middle and the two parts are allocated to the different processors. Thus, irrespective of the degree of VP decomposition, only two remote communications are needed in sending a message once around the ring, and all other communications will be local to the processors.

As expected, the performances of ring on PVM and UPVM are comparable for the interleaved scheme. (See Table 6.5.) However, UPVM performs significantly better than PVM for the block-decomposed scheme whenever there are more number of VPs than there are processors. (See Table 6.5 and Figure 6.5 that plots the block-decomposed scheme.) Specifically, UPVM performs at least twice as well as PVM for 8 or more VPs.

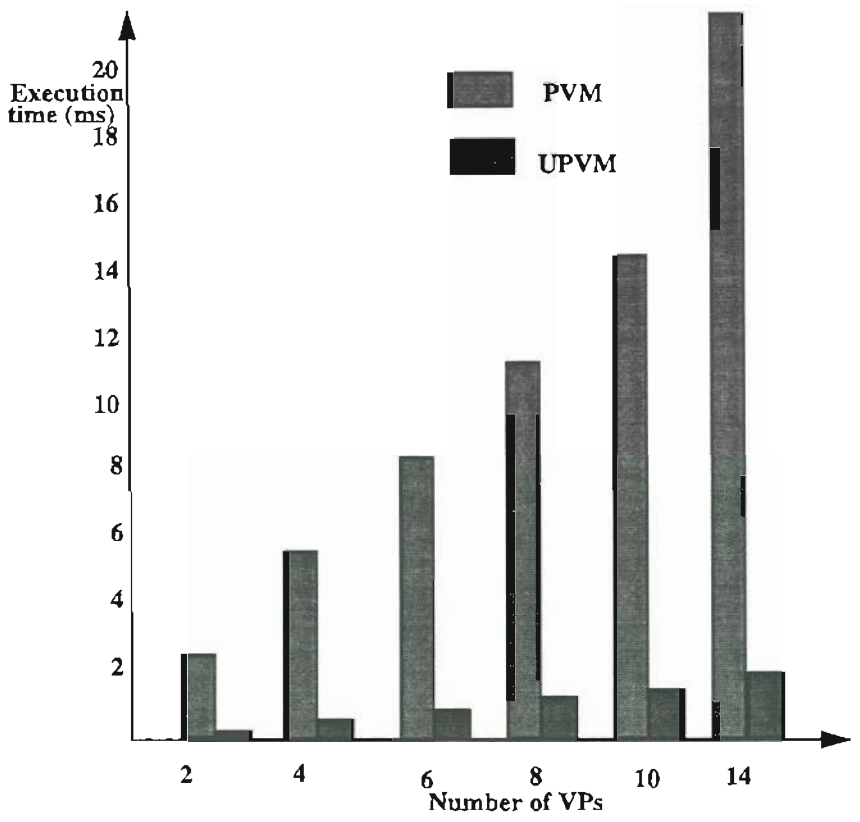

Figure 6.4: Ring on a single node

This improvement is due to UPVM's gains from its local communication optimizations as the number of VPs in each block increase. Thus, a suitable VP allocation scheme is a critical factor for UPVM in achieving high performance.

6.3.2 Laplace grid solver

The Laplace 2-dimensional grid solver (LGS) uses the Gauss-Jacobi method for solving a 128x128 grid. The grid is distributed to the application VPs along the column dimension using block decomposition. For example, if the application is decomposed into two VPs, each VP gets a 128x64 grid. Each VP "sweeps" over its portion of the grid 10 times doing an averaging operation at each point of its grid and then performs a pair-wise exchange with its neighbouring VP to update its border-element strip. After 5000 sweeps, the

# VPs	PVM(ms)		UPVM(ms)	
	Intlv	Blk	Intlv	Blk
2	4.82	4.82	5.01	5.01
4	9.76	7.86	10.27	5.19
6	14.64	10.82	15.08	6.14
8	21.75	14.01	20.34	6.40
10	26.28	17.06	25.59	7.21
14	36.86	23.48	35.67	7.69
20	52.88	33.66	51.30	8.95
24	64.86	41.86	60.88	9.73

Table 6.5: Ring on two nodes

application terminates. For the 128x128 grid, the border-element strip is 512 bytes (128 floating point numbers) long. Since there are 500 border-strip communications, the total number of messages during this application execution is equal to $(N-1)\cdot 2\cdot 500$, where N is the number of VPs.

Table 6.6 and Figure 6.6 show the results for LGS executing on one processor. For comparison, the performance of the sequential LGS is 2.79 Mflops. The main thing to note is that the performance is comparable for a small number of VPs since the application has a large computation-to-communication ratio. However, as the number of VPs increases, so does the number of local messages as calculated from the formula above. This accounts for the performance improvement of UPVM over PVM for larger numbers of VPs. For example, at 11 VPs, PVM performance has degraded by about 16%, while UPVM has degraded by only about 8%.

Table 6.7 and Figure 6.7 show the results of the application running on two processors. The VPs are block-allocated, that is, VPs operating on neighbouring portions of the grid are allocated to the same processor. Thus, remote communication is reduced to one pair-wise exchange of border strips, once per 10 sweeps.

As expected, PVM performs better in the two-VP case, since all communication is remote. However, we see that UPVM performs better than PVM for all other cases. PVM has a performance degradation of about 21% and 23% for 5 and 11 VPs respectively. For UPVM, the degradation is about 17.1% for 5 VPs and 17.9% for 11 VPs.

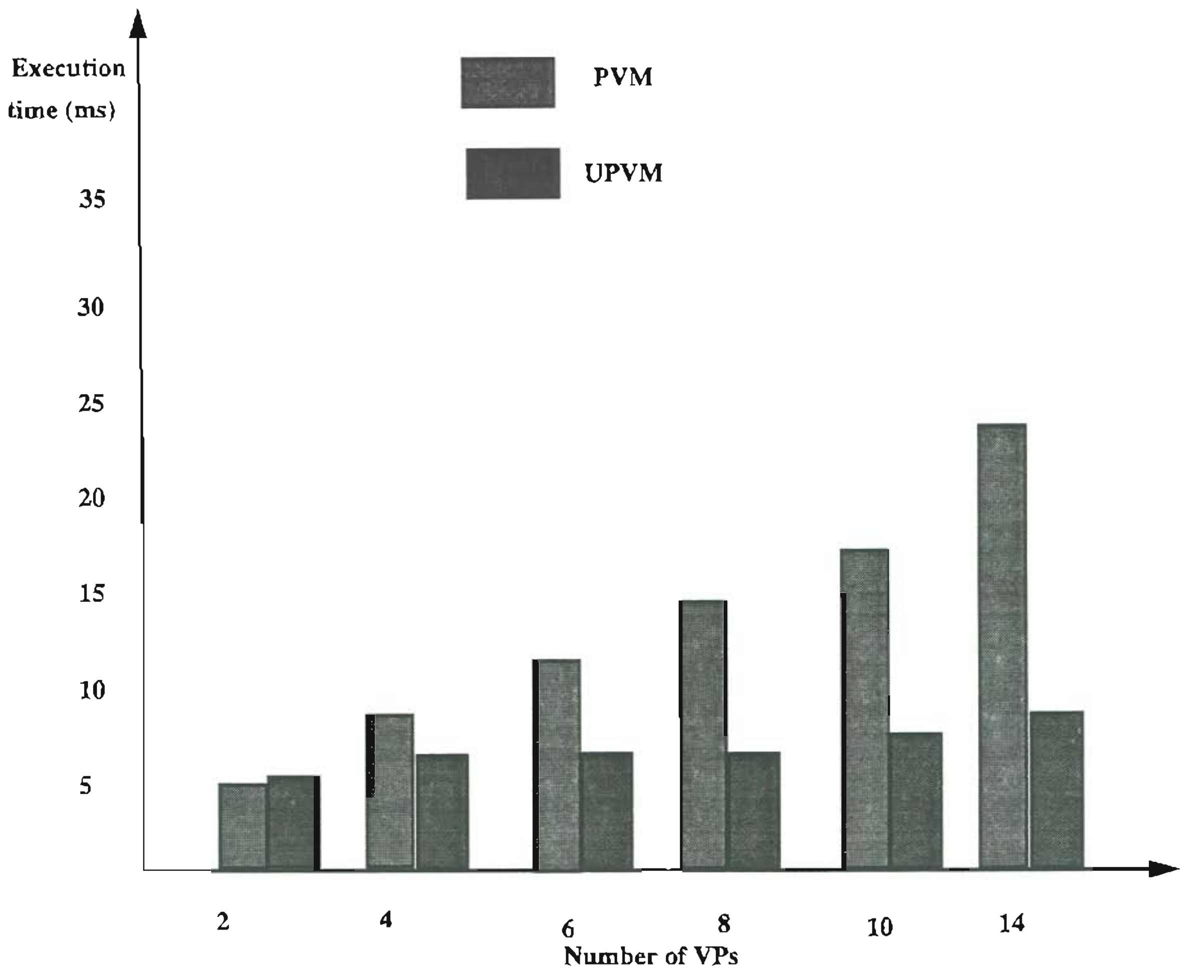

Figure 6.5: Ring on two nodes with blocked allocation of VPs

Note the different performance trends of odd and even number of VPs in Table 6.7 and Figure 6.7. The performance of the even-numbered VPs is decreasing while that of odd-numbered VPs is increasing as we go down the table. The reason for this behaviour is the load imbalance between the two processors. The three-VP case has the worst performance in both systems because it has the most imbalance in load. As the number of VPs increase, the amount of imbalance decreases in the odd case, thus improving the performance.

For the case of even number of VPs however, the application is always load balanced.

# VPs	PVM (Mflops)	UPVM (Mflops)
2	2.75	2.79
3	2.68	2.69
4	2.63	2.68
5	2.57	2.67
6	2.54	2.66
7	2.50	2.65
8	2.45	2.59
9	2.42	2.58
10	2.38	2.58
11	2.34	2.56

Table 6.6: LGS on one node

# VPs	PVM (Mflops)	UPVM (Mflops)
2	5.19	5.02
3	3.84	3.93
4	4.84	4.96
5	4.10	4.30
6	4.75	4.94
7	4.15	4.32
8	4.44	4.63
9	4.11	4.41
10	4.41	4.63
11	3.99	4.26

Table 6.7: LGS on two nodes

Thus performance degrades with the increasing overhead of supporting additional VPs.

In summary, UPVM supports over-decomposition much better than PVM.

6.4 Migration performance

Since the main goal of UPVM is to achieve unobtrusive and efficient parallel computation, we use three basic measures in characterizing its performance. These are:

1. Inherent method overhead. How much overhead does an application incur when using UPVM as compared to using a straightforward implementation (i.e., standard PVM) when no migration takes place? That is, what is the overhead of UPVM in the quiet case?

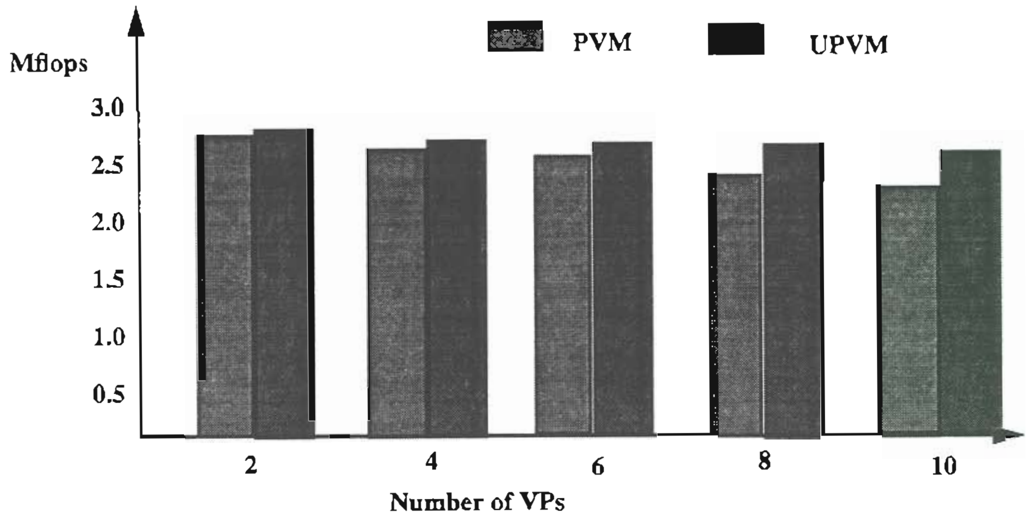

Figure 6.6: Laplace grid solver on single node

2. Obtrusiveness. What is the time taken from the instant a migration event was received to the instant the application is "off" the processor? That is, what is the impact on workstations owners when they want their workstation back and computation has to be moved away from their workstations?

3. Migration cost. What is the time taken from the instant the migration event is received to the instant the migrated unit of work is integrated back into the parallel job? That is, what is the impact on the parallel computation?

Method overhead

The overhead incurred by an application during normal execution can be attributed to the three factors: 1) the cost of avoiding potential re-entrancy problems in the library, 2) the mapping of application tids into actual tids for message communication and 3) the mapping of of the **pvm_recv**() on to non-blocking functions within UPVM such that it does not block the entire ULP library while being able to react to migration events. (See chapter 4.) In addition, UPVM adds extra information for remote messages that result in marginally slower remote communication than PVM. Thus, the overheads incurred by

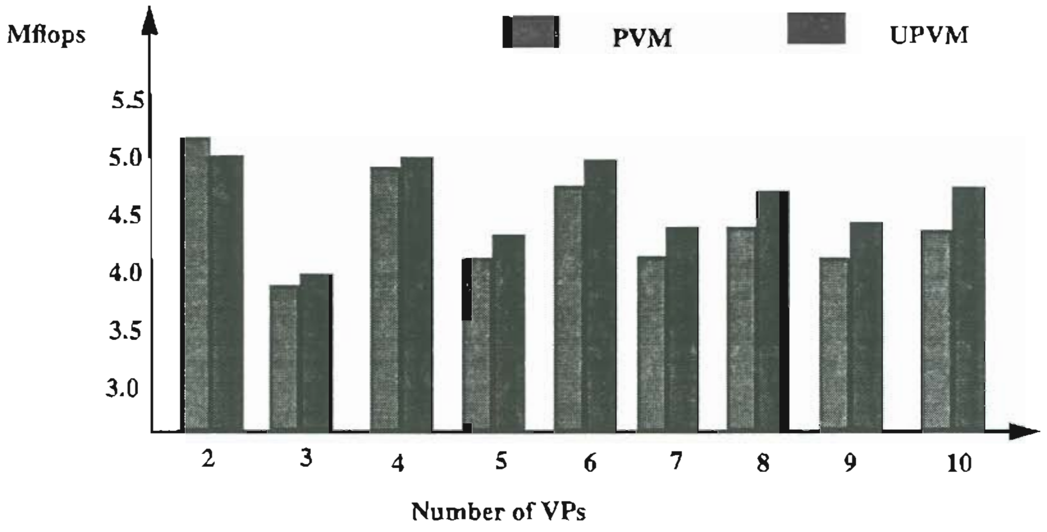

Figure 6.7: Laplace grid solver on two nodes

UPVM can be examined by comparing its performance to UPVM when an application is divided into as many VPs as there are processors. Since the ring program has almost no computation, the overhead of UPVM is not masked by any computation. The execution times of the ring program using PVM and UPVM have been shown earlier and are reproduced for convenience in Table 6.8. UPVM overhead increases by 0.19 ms, which is 3.9% increase over using PVM. This increase is an artifact of the current implementation and we expect that a UPVM implementation directly on top of OS will reduce overheads to comparable to that of vanilla PVM.

PVM (ms)	UPVM (ms)
4.82	5.01

Table 6.8: UPVM overhead over PVM

Obtrusiveness

For measuring obtrusiveness and migration costs in UPVM, we benchmarked a neural-network classifying application called "Opt". Opt based on conjugate-gradient optimization [BC89] and is generally employed as a speech classifier utilizing large (500KB to 400MB) training sets as input. Opt works by applying an initial neural net to a series of floating point vectors called exemplars (representing digitized speech sound) so that a gradient is found. This gradient is then used to modify the neural net, training it. This process is repeated until error values pass a threshold or a predetermined number of iterations has been performed.

For our tests, we used a parallel version of Opt (PVM_opt). PVM_opt has one master VP and 2 slave VPs, one on each machine with data equally distributed among the slaves. The master VP computes a new gradient from partial gradients computed by the slaves, applies this gradient to the neural net, and broadcasts the new net to the slaves. The slave VPs apply the new net to the exemplars to get a new partial gradient for the next cycle.

Since the package supports only SPMD applications, an SPMD version of the PVM_opt was created. The SPMD Opt program retains the same structure as PVM_opt in that one of the VPs exclusively functions as the master and the rest of the VPs execute as slaves.

The obtrusiveness cost measured in this experiment is the time it takes from when a migration event is received to when all the state of the migrating ULP is off-loaded from the source host. Migration was caused by a simple GS program that sent a migration signal and a migration message to the specified PVM task. The migration message specified the slave ULP as the migration victim.

Table 6.9 and Figure 6.8 show the obtrusiveness costs for various data sizes. For comparison, two more items are shown. First, the cost of using TCP/IP for the various data sizes is shown in order to establish the lower bound for minimum transfer times possible on the underlying network. Second, the ratio of obtrusiveness cost to TCP/IP cost is given to show how well UPVM does against this lower bound. The obtrusiveness cost increases almost linearly with the data size, from 1.10 seconds for 0.3 MB and 4.24 seconds for 2.9 MB. The migration cost, as expected, is slightly higher but closely parallels the obtrusiveness cost.

Also, as the data size increases, the time to transfer data becomes a more prominent factor in the overall obtrusiveness cost. Thus, the ratio of obtrusiveness cost to TCP/IP cost decreases, since obtrusiveness cost is the sum total of time spent in the ULP migration protocol and the time spent in transferring data over the network. This behavior is shown by the fifth column in Table 6.9.

Input Data Size	Obtrus. cost(sec)	Migr. cost(sec)	TCP cost (sec)	Ratio (obtr/TCP)
0.3 MB	1.10	1.18	0.27	4.07
0.5 MB	1.32	1.42	0.47	2.81
1.0 MB	1.91	1.98	0.92	2.08
1.6 MB	2.55	2.67	1.40	1.82
2.1 MB	3.19	3.28	1.88	1.70
2.9 MB	4.24	4.47	2.51	1.69

Table 6.9: UPVM: Obtrusiveness and Migration costs

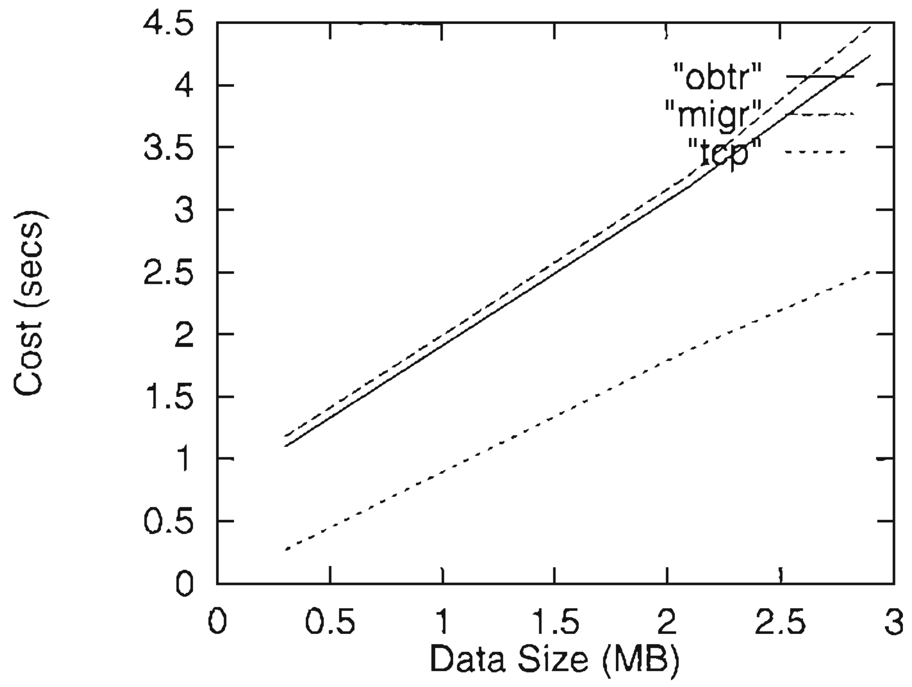

Figure 6.8: UPVM migration performance for Opt

However, notice a disturbing trend in the TCP/IP and the ULP obtrusiveness costs. As the data sizes increase, the curves diverge. Such behaviour is apparently incorrect, given that data transfer times dominate at larger data sizes. This divergence is an artifact of UPVM's implementation. Currently, ULP data and state is transferred over the network by first packing the data and state into buffers similar to that done by PVM applications and then transmitting it using send routines similar to those provided by the standard PVM library. Packing routines essentially result in copying the entire data and ULP state twice, and as data sizes increase, the effect of memory accesses and cache misses increase the cost of ULP state transfer relatively more than simple TCP/IP that does not perform this double copy. Hence, the curves diverge.

Migration cost

As in the case of obtrusiveness, the migration costs in UPVM increase almost linearly with the data size, with 1.18 seconds for 0.3 MB and 4.47 seconds for 2.9 MB. (See Table 6.9 and Figure 6.8.)

Again, as with the obtrusiveness costs, the cost of using PVM packing, unpacking, and send calls in UPVM increase with the training set size. Thus for larger data sizes, the cost of migrating a ULP also diverges slightly from that of obtrusiveness costs. Our next version of UPVM will use a direct TCP connection and should eliminate the double copying costs, thus reducing the cost of both obtrusiveness and migration.

Note that during ULP migration, the source and destination OS processes are executing within the ULP library to perform the ULP migration. This execution implies that other ULPs within the source and destination OS processes cannot execute during this time. ULPs within other processes of the parallel application are free to continue and execute and communicate with each other during the ULP migration. There is a small cost incurred by each process that deals with responding and sending a ULP-migration-ack message to the source process. However, there is a possibility for the entire parallel application to block if ULPs in all the processes are waiting on one or more ULPs in the source or the destination OS process to execute and send a message.

6.5 Summary

The UPVM prototype has demonstrated an order of magnitude performance improvement over PVM for the communications on the same node. Over-decomposed applications, for which the amount of remote communication can be controlled, also perform better with proper allocation of ULPs to processors. This has been shown by both the ring and Laplace benchmarks.

However, UPVM is still constrained by its remote communication performance. Applications that use broadcasts among VPs cannot be over-decomposed without increasing the number of remote communications. Considering the current implementation, these broadcasts will result in large overheads. In our future work, we plan to optimize remote communication along with several other portions of the UPVM prototype.

ULP migration cost is almost linear in the size of the ULP state and even with our un-optimized data transfer scheme, achieves about 60% utilization of the maximum possible bandwidth of the underlying network for data sizes greater than 2.1 MB. Finally, we believe that ULP migration can be further optimized by eliminating the double-copy problem described in the previous section.

The experiments in this chapter do not expose the scalability of the migration mechanism and are part of our future plan. However, the scalability of the migration mechanism based on the remote communication benchmark and the migration results. Given an idle network, the lower bound on the ULP migration cost when N processors are involved can be given by the relation: $T_{mc} = T_{2p} + 2 * T_{rc20} * (N - 2)$, where T_{2p} is the ULP migration cost in the two processor case, and T_{rc20} is the cost of one-way remote communication of a twenty-byte message. Twenty-bytes is the size of the migration and acknowledgement messages in the current implementation.

Another feature of over-decomposition that has not been exposed is the overlap of computation with communication. All experiments show that applications using PVM, with one task per node, have better performance than using over-decomposition using UPVM. The reason for better PVM performance is a combination of the high message startup overhead to achieve communication and the idleness of network. Because of the high startup cost there is less potential for overlapping computation with computation. Further, because the network was idle, the one VP-per-processor PVM programs did not spend as much time blocking for a message as they would on a heavily loaded network. In such cases, we expect an over-decomposed application using UPVM to perform better than a one-task-per-processor decomposition using PVM.

Chapter 7

Discussion and Related Work

The idea of user-level processes is one approach to the problem of providing light-weight over-decomposition and transparent migration for message based parallel applications. However, there are several issues that need to be considered when implementing, porting, programming, or determining the applicability of UPVM. This chapter discusses some of the main issues.

7.1 OS support for performance

The problems of supporting programming abstractions at user-level are well explored in the literature [MSLM91, ABLL92]. Operating systems manage processes or threads, and do not know about abstractions implemented at user-level. This "mismatch" can result in performance degradation of applications. For example, because the OS does not know about ULPs, a page fault incurred by one ULP blocks the entire OS process, even if other ULPs are ready to run within that process. The same situation occurs for blocking I/O operations.

However, these problems have been addressed in the context of user-level, thread-based systems using scheduler activations [ABLL92], first-class user-level threads [MSLM91] and new types of signals in the Solaris operating system [PKB$^+$91].

The scheduler-activation approach is designed for efficient implementations of user-level abstractions on shared-memory multiprocessors. The approach requires changes to the OS such that it communicates all OS-level events such as page-faults, processor preemption, and blocking due to I/O. The user-level library registers the event-handler code that should be used by the OS in its up-calls. Each up-call results in the event-handling code being called with a different stack and, with the proper design of the handler, it is possible to handle multiple events simultaneously. The OS however always has complete control over the system resource management. Thus a user-level library, as described in UPVM's design, can make use of these events in scheduling of user-level abstractions.

The first-class user-level thread approach is another alternative towards integration with the OS. In this approach, the OS and the user-level library communicate through shared memory for efficiency and the OS uses a signal-like mechanism to inform the user-level library of system events. To avoid preemption during critical sections, the user-level library sets a flag within the shared memory that consequently delays kernel preemption from the processor. Although this approach violates the notion that the OS is the manager of resources, it makes sense on a NUMA multi-processor because pre-emptive migration of user-level abstractions is expensive.

Solaris provides additional signals that inform application programs of some OS events. However, there is much room for better integration. We believe that this attempt for better integration from a commercial operating system is a trend indicating that future commercial operating systems will provide more support for integrating user-level abstractions.

7.2 Supporting a general-purpose ULP

ULPs have been designed specifically to support message-based scientific computing. Consequently, general-purpose operations permitted by their OS counterparts are not supported. For example, true preemptive scheduling and interfaces for forking, sockets, signals, and resource-usage timers are not supported.

Although this functionality could be supported by ULPs, it would add significant overhead for those applications that do not need this full generality and add more complexity to ULP migration. At the limit, supporting all functionality would require a re-implementation of the OS at user-level. One consequence would be poor portability and an inability to support true VP virtualization because of incompatibility among the various OS interface and a lack of a single system image.

For these reasons, we suggest that a specialized application interface be used for scientific computing that is much narrower than a general purpose OS interface. Applications operating within this specialized environment can obtain the benefits of location independence, transparent migration and dynamic load balancing that are essential for shared workstation networks. There is ongoing work here at OGI, Carnegie Mellon University, Oak Ridge National Laboratory, and University of Tennessee at Knoxville to define such an interface, called the Concurrent Processing Environment (CPE) interface, for PVM-based parallel applications [BDG+93].

7.3 Migration

ULP migration is designed to work between workstation architectures that are binary compatible. Heterogeneity is possible, but restricted, in the ULP environment. An application can be executed such that some of its workstations are of say, architecture A and others are of architecture B. The ULP system then maintains two virtual address spaces, one for architecture A, and one for architecture B and allows the migration of ULPs among the same architecture. Maintaining separate address space for different processor pools allows for ULPs that are created on one architecture to have overlapping addresses with the ULPs created on a different architecture.

Migrating processes across heterogeneous architectures in a language independent and application-transparent manner is extremely difficult. Processors of different architectures can vary in the instruction set, number of registers, type of registers, the size of addresses, etc. To migrate a process to a workstation of different architecture, the process address space as well as its stack and register context needs to transformed to an equivalent execution context on the target architecture. Different types and sizes of register sets between the processors imply there is no simple mapping of the register context. Another problem is dealing with pointers not only in the address space but also on the stack. Pointers are not always traceable in a language-independent manner and mishandling these pointers can change the behaviour of the migrated process, thus violating the fundamental rule of transparent migration.

To support heterogeneous migration, several approaches have been proposed in literature. Some approaches require programming in a particular language [HAJLQA91, TH92], others, such as the DOME environment [BSS94], require the application to explicitly identify the data that should be preserved across migration. However these methods are not applicable in the context of our research due to need to provide transparent migration of VPs while not constraining the user to any one programming language or environment.

Even within a binary-compatible pool of processors, shared libraries present yet another problem for migration. These libraries are shared read-only by multiple executing processes on a workstation. When a process starts executing, a dynamic linkage table within the user process is initialized by the dynamic loader so that process can access these shared libraries. If the operating systems on different processors map the shared libraries at different regions because of difference in the size of physical memory available, this difference in mapping can cause migration problems. Upon migration, on a call to a shared library routine, the dynamic linkage table is examined. Finding the location initialized, the call is transferred to the address found in the location. Since the shared libraries are mapped differently, this transfer can result in incorrect execution. Although conceptually simple to correct, the location, structure, and the manipulation of the the dynamic linkage table is operating-system specific and adds complexity to migration. Because of these problems, we restrict the scope of migration currently to statically linked programs.

7.4 Portability

Three portability issues have been considered while designing the ULP package. One issue is porting the ULP package to different architectures. The second issue is that of supporting SPMD versus task parallelism. Finally, we considered supporting a message-passing interface other than PVM.

For porting to a new architecture, the instruction set, register architecture, accessibility of these registers to user-level code, and the procedure-calling conventions of the OS need to be understood. These conventions determine the general and floating point registers that must be saved and restored in a ULP context switch. Since ULPs are laid out in distinct regions of a process virtual address space, the virtual memory layout, as defined by the OS, must also be taken into account.

To support SPMD applications only, it is sufficient to have a compiler on the target workstation capable of generating instructions that access data relative to a user accessible general register (such as DP). Since text is shared among all ULPs in an SPMD application, a ULP context switch simply becomes the act of saving and restoring this DP register, in addition to the general register context.

On the other hand, extending support to task parallelism requires more effort. The compiler on the target workstation must be able to generate position-independent code so that the object code can be loaded into any virtual address region within a process. Furthermore, the operating system must provide an interface to dynamically load and link code and data modules into an existing virtual address space.

The concept of ULPs is clearly applicable to process based applications using message-passing interfaces other than PVM. The ULP creation, control, context switch, scheduling, memory allocation, file access, and a portion of the migration mechanism are all independent of the message-passing interface. Thus, for supporting a ULP package for another message-passing interface, only the inter-ULP communication and a portion of the migration mechanism needs to be rewritten.

7.5 Protection and Debugging

One potential source of difficulty is that the ULP system does not provide protection between the local VPs of an application. This lack of protection means that the execution of multiple ULPs within the same process can cause unexpected side-effects.

A more practical problem is that operating system utilities such as debuggers and profilers that work on processes do not recognize ULPs. Thus, debugging an application using ULPs is difficult. Similarly, profilers have problems understanding the control flow within a multi-threaded process.

Since UPVM provides the same interface as PVM, a simple approach (from the UPVM developer's perspective) is to debug and profile PVM applications as normal UNIX processes. Once the application is debugged, it can then be compiled with UPVM. In fact, this was the approach we adopted in running PVM programs on UPVM.

Another approach is to use the **mprotect()**(2) system call in altering the protections of ULP address spaces on every context switch. This approach is impractical since the cost of altering protections at user-level would be at least an order of magnitude more than a process context switch, defeating the very purpose of creating a user-level abstraction.

A much more attractive approach is Software Fault Isolation (SFI) [WLAG93], where code is modified while ULP loading to achieve a *sand-boxing* effect. Because of this sand-boxing, memory accesses during code execution do not go outside the bounds specified at the modification time. Further, the overall behaviour of the code remains unchanged. Execution times of the modified code are only slightly more expensive compared with that of the original code. Integrating this approach with UPVM is certainly an item in future work.

Chapter 8

Conclusions and Future Work

Efficient utilization of multi-user DMMPs, such as workstation networks, require message-based parallel applications to overlap their communication with computation, perform dynamic load balancing based on the variations in processor load, and at the same remain unobtrusive to workstation owners. Without proper system-level support, application programmers have to deal with these issues, which results in complicated application code that is difficult to debug.

To insulate application programmers from many of these programming complexities, this thesis focused on system-level solutions that are application and language independent. Because of the wide body of process-based legacy applications, we placed a further constraint that the system-level solutions must be able to support these applications with few or no changes to application code.

We made a case that a light-weight, transparently migratable VP system, in combination with over-decomposition of parallel applications, can maintain unobtrusiveness while achieving high performance through dynamic load balancing and overlap of communication and computation.

The general goals for a VP system were then transformed into a list of functionality requirements and performance criteria with which we examined current VP systems. We showed how existing approaches are inappropriate and we defined a new light-weight, migratable VP abstraction called the user-level-process (ULP).

We discussed the general issues in designing ULP systems and applied the resulting framework to the design of UPVM, a ULP-based package to support the PVM message-passing interface.

To demonstrate the viability of ULPs and the UPVM design, a prototype UPVM package was implemented on a network of HP 9000 series 700 workstations. The performance of the prototype was analyzed with respect to both micro and application-level benchmarks and a side-by-side comparison was made with the standard process-based PVM library. The comparison shows that the context switch and local IPC costs in UPVM are at least an order of magnitude better than the PVM library. UPVM performs better than PVM whenever there are more VPs than there are processors. ULP migration has negligible run-time overhead during the absence of any migration. For a data size of about 3 MB, ULP migration takes 4.41 seconds on a pool of two processors, which is around 60% of the bandwidth possible on a ethernet network using TCP/IP. The migration results are especially encouraging since the optimizations to the migration mechanism are yet to be implemented.

Comparing the functionality and performance of UPVM with the list of functionality requirements and performance goals, the following statements can be made about UPVM:

- The UPVM prototype provides support for message-based IPC and transparent migration for PVM applications that are based on the process model. However the applications can currently be only SPMD and have to exhibit static parallelism. As discussed in Chapter 4, this is only a restriction in the prototype and it should be possible to extend UPVM to support more general parallelism. Thus we conclude that the functionality requirements have been satisfied.

- UPVM performs better than the process-based PVM library whenever there are more VPs than there are processors. Thus we conclude that the minimum goals for UPVM performance have been met.

- We argue that UPVM has context switch performance that is comparable to a user-level thread implementation because UPVM needs to switch only one more register in addition to that of a user-level thread switch.

- Local IPC costs in UPVM will always be at least as expensive as a local IPC among user-level threads because threads can share memory directly while ULPs cannot.
- The overhead caused by the presence of migration in UPVM is negligible, as shown by the performance of the Laplace grid solver. Hence UPVM meets the performance goal for low-overhead.
- The time to respond to a migration event for the benchmark shown was in the order of milliseconds, which satisfies the responsiveness goal. However, responsiveness may not always be as good in the UPVM prototype. If UPVM is in a critical section, then the ULP library must defer ULP migration until it leaves from the critical section. In the prototype, message send and receive operations are implemented within critical sections. Since messages can be arbitrarily long, the time spent in the critical sections is effectively unbounded. However, since applications are expected to spend more time in computation than in communication, we expect this case to occur infrequently in the UPVM prototype. We conclude that with further modifications to the UPVM library, the time spent in critical sections can be made smaller and the response goals can be achieved.
- Obtrusiveness cost and migration cost are almost linear in the size of ULP state. Further, the un-optimized migration cost in the order of a few seconds for 3 MB of data. In this case, we conclude that we have achieved our performance goal for ULP migration.
- As shown by the over-decomposed Laplace and ring benchmarks, the mapping of ULPs to processors is a critical factor for application performance.
- Applications using PVM, with one task per node, have better performance than using over-decomposition using UPVM. The reason for better PVM performance is a combination of the high message startup overhead to achieve communication and

the idle network. Because of the high startup cost there is less potential for overlapping computation with computation. Further, because the network was idle, the one VP-per-processor PVM programs did not spend as much time blocking for a message as they would on a heavily loaded network. On such heavily-loaded networks, we expect an over-decomposed application using UPVM to perform better than a one-task-per-processor decomposition using PVM.

Also, the source code for UPVM has been made available to the NEXUS research group at the California Institute of Technology.

8.1 Future work

The UPVM prototype can be extended in many ways:

Limiting the use of swap space: As currently implemented, UPVM allocates the sum-total of the ULP address spaces on all workstations irrespective of the presence of ULPs on those workstations. Because of this allocation strategy, the swap partitions of the workstations are under-utilized and, more importantly, interfere with workstation use. Resolving this limitation is the first priority.

Supporting I/O: The UPVM prototype exhibits the problems of typical user-level VP implementations with regards to I/O. A blocking I/O call made by one ULP blocks the entire process. However, within the current framework, UPVM can be extended in a straightforward manner to map blocking I/O primitives in terms of non-blocking I/O primitives.

Integration with global scheduler (GS): Currently, UPVM's interface with the global scheduler is primitive and a global scheduling environment is still missing towards realizing a practical, unobtrusive computing environment.

UPVM's interface with the GS should be extended and UPVM itself modified such that determining the number of processors to initially use, the virtual address space available, etc, are requested from the GS instead of the user typing them as command-line arguments.

The GS environment should consist of efficient mechanisms to monitor parameters such as processor load, network load and user activity and backed by a scheduling policy that makes decisions based on the values of these parameters. Work is on-going to developing this infrastructure in a related research project here at OGI [CCG+95].

Supporting program and dynamic parallelism: Supporting such a functionality requires the prototype to be modified to support dynamic ULP creation from different program executables. Further, the prototype needs to be integrated with the GS as discussed above so that it can take advantage of new processors and perform unobtrusively. Also, there is a lot of scope for experimenting with different ways of program compilation and the resultant effects on dynamic loading, ULP context switching, and portability of the UPVM prototype.

Supporting inter-ULP protection: In this regard, the software fault isolation approach is very attractive. Specifically, it does not require OS intervention and the overhead is certainly more acceptable than the **mprotect()** approach discussed in the previous chapter. We plan to look into ways of integrating the sand-boxing ideas into UPVM.

Optimizing migration: The migration mechanism that is currently using pvmlib routines for packing and sending ULP state can be replaced by a mechanism that uses TCP/IP sockets directly. As explained in Chapter 6, such an approach should eliminate the double-copying problem that exists currently and bring the migration performance closer to that of TCP/IP.

Multi-processor support: The UPVM prototype is implemented for uniprocessors and can be extended to provide support for shared-memory multiprocessors. Related research [BLL88, FM92a, CAL+89] can be used to perform this extension.

Porting to other architectures: We plan on porting the UPVM prototype to the SPARC architecture. Because of the SPARC's register windows, not all execution state is directly available at user-level, and this lack of availability makes context switching tricky. However, user-level threads have been implemented on the SPARC [Kep91] and we plan to use this research as our starting point.

Bibliography

[ABB$^+$86] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, Georgia, 1986.

[ABLL91] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, Pacific Grove, CA, October 1991.

[ABLL92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[AF89] Y. Artsy and R. Finkel. Designing a process migration facility: The charlotte experience. *IEEE Computer*, 22(9):47–56, September 1989.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, Santa Clara, CA (USA), April 1991.

[BBB$^+$93] L. Branscomb, T. Belytschko, P. Bridenbaugh, T. Chay, J. Dozier, G. Grest, E. Hayes, B. Honig, N. Lande, Jr. W. Lester, G. McRae, J. Sethian, B. Smith, and M. Vernon. From desktop to teraflop: Exploiting the U.S. lead in high performance computing. Technical report, NSFF Blue Ribbon Panel on High Performance Computing, 1993.

[BC89] Etienne Barnard and Ronald Cole. A neural-net training program based on conjugate-gradient optimization. Technical Report CSE-89-014, Dept. of

Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 1989.

[BDG+93] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Steve Otto, and Jon Walpole. PVM: Experiences, current status and future direction. In *Supercomputing'93 Proceedings*, pages 765–6, Portland, OR, November 1993.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[Bla90] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[BLL88] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[BLL91] A. Bricker, M. Litzkow, and M. Livny. Condor technical summary. Technical report, University of Wisconsin at Madison, October 1991.

[BR94] K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with the ISIS toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[BSS94] Adam Beguelin, Erik Seligman, and Michael Starkey. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, Carnegie Mellon University, May 1994.

[CAL+89] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 147–158, Litchfield Park, AZ, December 1989.

[CCG+95] J. Casas, D. Clark, P. Galbiati, R. Konuru, R. Prouty, S. Otto, and J. Walpole. Mist: Pvm with transparent migration and checkpointing. In *Third Annual PVM Users' Group Meeting*, Pittsburgh, PA, May 1995.

[CHKS86] Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag. The HP-UX operating system on HP Precision Architecture computers. *Hewlett-Packard Journal*, 37(12):4–22, December 1986.

[CK79] Y. C. Chow and W. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, C-28(5):354–361, May 1979.

[Cor87] Encore Computer Corporation. *UMAX 4.2 Programmer's reference Manual.* Marlborough, MA, 1987.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience*, 21(8):757–785, August 1991.

[Doe87] Thomas W. Doeppner. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Department of Computer Science Brown University, Providence, RI 02912, June 1987.

[Dou90] F. Douglis. *Transparent Process Migration in the Sprite Operating System.* PhD thesis, University of California, Berkeley, CA 94720, September 1990. Available as Technical Report UCB/CSD 90/598.

[FJL$^+$88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, NJ, 1988.

[FM92a] E. W. Felten and D. McNamee. Improving the performance of message passing applications by multithreading. Technical Report 92-09-07, Dept of Computer Science and Engineering, University of Washington, 1992.

[FM92b] E.W. Felten and D. McNamee. Improving the performance of message-passing applications by by multithreading. In *Proceeding of the Scalable High Performance Computing Conference*, pages 84–9, Los Alamitos, CA, April 1992.

[For93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[FZ86] D. Ferrari and S. Zhou. A load index for dynamic load balancing. In *Proceedings of the Fall Joint Computer Conference*, pages 684–690, Dallas, TX, November 1986.

[GBD+93] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[GS92] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[GS93] T. P. Green and J. Snyder. DQS, a distributed queueing system. Technical report, Florida State University, March 1993.

[HAJLQA91] Philip J. Hatcher, Robert R. Jones Anthony J. Lapadula, Michael J. Quinn, and Ray J. Anderson. A production-quality C* compiler for hypercube multicomputers. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 73–82, Williamsburg, VA, April 1991.

[Har91] R. J. Harrison. Portable tools and applications for parallel computers. *Int. J. Quantum Chem.*, 40:847–863, 1991.

[Hew90] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, first edition, November 1990.

[HJ86] A. Hac and T. J. Johnson. A study of dynamic load balancing in a distributed system. *Computer Communication Review*, 16(3):348–56, August 1986.

[Inc87] BBN Advanced Computers Inc. *Inside the Butterfly plus*, October 1987.

[Int91] Intel Corporation. *iPSC/2 and iPSC/860 Programmer's Reference Manual*, April 1991.

[ISB86] M. A. Iqbal, J. H. Saltz, and S. H. Bokhari. A comparative analysis of static and dynamic load balancing strategies. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 1040–1047, University Park, PA, August 1986.

[Jul88] Eric Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, University of Washington, Seattle, USA, 1988.

[Jul89] E. Jul. Migration of light-weight processes in Emerald. *IEEE Technical Committee on Operating Systems Newsletter*, 3(1):20–23, Winter 1989.

[Kep91] David Keppel. Register windows and user-space threads on the SPARC. Technical Report TR 91-08-01, University of Washington, August 1991.

[KN93] Joseph A. Kaplan and Michael L. Nelson. A comparison of queueing, cluster and distributed computing systems. Technical Report 109025, NASA Langley Research Center, Hampton, Virginia 23681-0001, October 1993.

[LAJ91] Rodger Lea, Paulo Amaral, and Christian Jacquemot. COOL-2: An object oriented support platform built above the Chorus micro-kernel. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pages 68–72, Palo Alto, CA, October 1991.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, CA, June 1988.

[LMKQ89] Samuel L. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[LP93] M. Livny and J. Pruyne. Scheduling PVM on workstation clusters using condor. Technical report, University of Wisconsin at Madison, May 1993.

[LT88] Tom Lovett and Shreekant Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, University Park, PA, August 1988.

[MPIf93a] The Message Passing Interface forum. Document for a standard message-passing interface. Technical Report CS-93-214, Dept. of Computer Science, University of Tennessee, 1993.

[MPIf93b] The Message-Passing Interface forum. MPI: A message-passing interface. In *Supercomputing '93 Proceedings*, pages 878–883, Portland, OR, November 1993.

[MSLM91] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on*

Operating Systems Principles, pages 95–109, Pacific Grove, CA, October 1991.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[NM93] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76, February 1993.

[NR94] B. Clifford Neuman and Santosh Rao. The Prospero Resource Manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, 6(4):339–355, June 1994.

[Ous90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, Anaheim, CA, June 1990.

[PKB+91] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter 1991 USENIX conference*, pages 1–14, Dallas, TX, January 1991.

[PM83] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 110–119, Bretton Woods, NH, October 1983.

[RAA+88] Marc Rozier, Vadim Abrossimov, Francois Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrman, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. Chorus distributed operating systems. *Computing Systems Journal*, 1(4):305–370, December 1988.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software—Practice and Experience*, 21(1):91–118, January 1991.

[Sab90] Marc Sabatella. Issues in shard library design. In *Proceedings of the Summer 1990 USENIX Conference*, pages 11–23, Anaheim, CA, June 1990.

[SUN88] SUN Microsystems Inc. *SUN OS Reference Manual*, May 1988.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[TH92] M. Thiemer and B. Hayes. Hetergenous process migration by recompilation. Technical Report CSL-92-3, Xerox Palo Alto Research Center, CA, 1992.

[The86] M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.* PhD thesis, Stanford University, 1986.

[Tur93] Louis H. Turcotte. A survey of software environments for exploiting networked computing resources. Technical Report MSU-EIRS-ERC-93-2, Mississippi State Engineering Research Center for Computational Field Simulation, P.O. Box 6176 Mississippi State, MS 39762, June 1993.

[Val90] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–11, 1990.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[VR88] M. Vandevoorde and E. Roberts. Workcrews: An abstraction for controlling parallelism. *Int. J. Parallel Programming*, 17(4):347–366, August 1988.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.

[WZAL93] J. Wang, S. Zhou, K. Ahmed, and W. Long. Lsbatch: A distributed load sharing batch system. Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto, Toronto, Canada, April 1993.

[YJR+87] Michael Young, Avadis Tevanian Jr., Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, and Robert Baron. The duality of memory and communication in the implementation of a multi-processor operating system. In *Procedings of the 11th ACM Symposium on Operating System Principles*, pages 63–75, Austin, TX, November 1987.

[Zay87] Edward R. Zayas. Attacking the process migration bottleneck. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 13–24, Austin, TX, November 1987.

[Zho87] S. Zhou. *Performance Studies of Dynamic Load Balancing in Distributed Systems*. PhD thesis, University of California, Berkeley, CA, October 1987. Technical Report No. UCB/CSD 87/376.

Biographical Note

Ravi Konuru was born in Hyderabad, India on May 11, 1963. He did his Bachelor's degree in Electronics and Communication Engineering from Osmania University India in 1983 and Master's degree in Computer Science and Engineering in 1985. He joined the PhD Program in Computer Science and Engineering at the Oregon Graduate Institute of Science & Technology in the 1989 and completed his PhD in 1995. In June 1995 he moved to New York to assume a research position in IBM's T. J. Watson Research Center. He is a member of ACM, SIGARCH and USENIX.