

Portland State University PDXScholar

Computer Science Faculty Publications and
Presentations

Computer Science

1-1997

A Migratable User-Level Process Package for PVM

Ravi Kunuru

Oregon Graduate Institute of Science & Technology


Steve Otto

Jonathan Walpole

Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: https://pdxscholar.library.pdx.edu/compsci_fac

 Part of the [OS and Networks Commons](#), and the [Systems Architecture Commons](#)

Citation Details

Kunuru, Ravi; Otto, Steve; and Walpole, Jonathan, "A Migratable User-Level Process Package for PVM" (1997). *Computer Science Faculty Publications and Presentations*. 46.

https://pdxscholar.library.pdx.edu/compsci_fac/46

This Post-Print is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

A Migratable User-Level Process Package for PVM

Ravi B. Konuru
ravik@watson.ibm.com

Steve W. Otto
otto@cse.ogi.edu

Jonathan Walpole
walpole@cse.ogi.edu

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Oregon Graduate Institute of Science & Technology
P. O. Box 91000
Portland, OR 97291-1000

Title:

A Migratable User-Level Process Package for PVM

Corresponding Author:

Ravi B. Konuru
Mail Stop H0-H21
IBM T. J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532
e-mail: ravik@watson.ibm.com
Tel: 914-784-7180
Fax: 914-784-7455

A Migratable User-Level Process Package for PVM

Shared, multi-user, workstation networks are characterized by unpredictable variability in system load. Further, the concept of workstation *ownership* is typically present. For efficient and unobtrusive computing in such environments, applications must not only overlap their computation with communication but also redistribute their computations adaptively based on changes in workstation availability and load. Managing these issues at application level leads to programs that are difficult to write and debug.

In this paper, we present a system that manages this dynamic multi-processor environment while exporting a simple message-based programming model of a dedicated, distributed memory multiprocessor to applications. Programmers are thus insulated from the many complexities of the dynamic environment at the same time are able to achieve the benefits of multi-threading, adaptive load distribution and unobtrusive computing. To support the dedicated multi-processor model efficiently, the system defines a new kind of virtual processor called User-Level Process (ULP) that can be used to implement efficient multi-threading and application-transparent migration. The viability of ULPs is demonstrated through UPVM, a prototype implementation of the PVM message passing interface using ULPs. Typically, existing 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.

1 Introduction

Message based parallel programs on workstation networks use the facilities provided by packages such as as PVM [17] and P4 [9] in order to create virtual processors(VPs) and communicate via the exported message passing interface . These packages in turn use operating system (OS) processes as their VPs and consequently, system calls provided by the OS are used to implement their message-passing and task-management interfaces. While this approach simplifies the development and portability of such systems, the need to invoke the OS for operations such as local communication and scheduling leads to significant overhead. For example, communication between two processes on the same node involves switching the register and virtual memory context as well as copying the message by the OS between the two processes. The cost of these operations is high compared to alternatives, such as direct copy or pointer manipulation, within the same address space.

Because of these overheads, a common approach is to maintain a one-to-one mapping of processes to processors, removing the need for local communication and scheduling. A side-effect of this approach, however, is that programmers resort to non-blocking message-passing primitives in an attempt to overlap communication with computation. The use of such primitives is generally undesirable because it increases programming complexity [15].

A one-to-one mapping is also undesirable in multi-user environments because it limits application parallelism to the number of currently available physical processors. This link between application parallelism and physical parallelism is particularly problematic in shared workstation environments where the number of physical processors available for parallel processing changes frequently. In this environment, new workstations become idle or allocated workstations are reclaimed by their owners [24]. Consequently, applications are forced to either suspend until the correct number of processors becomes available, or “double-up” on the

remaining processors. In both cases, application performance suffers. In the former case, the performance degrades because the entire application slows down because of the suspended virtual processor and in the latter case because of operating system overheads.

Alternatively, dynamic changes in processor availability can be managed within the application. In this approach, application programmers are responsible for monitoring system events and redistributing work dynamically. This option may have the greatest potential for high performance, but results in a significant increase in application programming complexity.

A simpler approach, called over-decomposition (OD)[32], is to create many more VPs than there are processors, and to delegate the responsibility of handling changes in processor parallelism and load balancing to the underlying VP system. Application-independent, dynamic load balancing can then be performed by the VP system through migration of these small-grain VPs. Also, OD allows the communication of one VP to be overlapped with the computation of another VP, hence removing the need for non-blocking message-passing primitives. If the overhead of VPs is low enough, this approach becomes attractive. However, OD at the granularity of heavy-weight OS processes leads to excessive overhead.

Attempts to address the high cost of OS processes have introduced a new OS abstraction called the *thread* [1, 30]. Like processes, threads have a register context and a stack. However, unlike processes, threads do not have their own private address space. Consequently, thread switches can be cheaper than process switches because they need not involve virtual memory context switches. Similarly, local communication is reduced to accessing memory locations in the same address space. Some packages implement the thread abstraction above the OS at user level [13, 14]. These user-level thread packages further reduce the cost of thread operations by avoiding the need to enter the OS for thread scheduling and management. The lower cost of local communication and context switching for both user and OS-level threads means that OD can be implemented efficiently. However, the fact that threads share memory means that it is difficult to delineate the state of one thread from another. Hence, it is difficult to migrate threads independently of each other. Furthermore, existing process-based applications require extensive modification to take advantage of threads.

The approach presented in this paper combines the low-overhead of user-level 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 private protection domain or address space. By convention, ULPs only communicate with each other via message passing. Hence, when a ULP must migrate, its state is clearly captured in its data space, heap, register context and stack. These can all be transferred to the target machine independently of other ULPs.

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 fig 1). All ULPs within a single OS process are scheduled by the ULP library code that also resides in that process. This means that ULP creation, context switching, and local communication 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.

This paper presents the design of UPVM, a ULP system for PVM applications. Using this design, a UPVM prototype has been implemented on HP series 9000/720 workstations. Existing SPMD-style PVM applications typically require only re-compilation and relinking to use UPVM. Performance results are presented at both the micro-benchmarking level and at the application level, and UPVM performance is compared with that of standard, UNIX-process-based PVM. The results indicate that ULPs are viable and light-weight and migratable virtual processor support can be provided in a language independent and application transparent manner.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 describes the design of UPVM. Performance results and a comparison with PVM[†] are presented in section 4. Section 5 discusses some main issues raised by this approach. Finally, we conclude in section 6.

2 Related work

There is a wide body of work that addresses finer virtual processor granularity than that of OS processes. These approaches can be broadly classified as OS-thread based, user-level-thread based, language-based, and object-based approaches.

Operating systems such as Mach [1] and Solaris [30], provide OS threads that can be used to reduce the cost of OD. Context switches between OS threads of the same process do not require the switching of the virtual memory context. Consequently, thread context switch is generally an order of magnitude or so faster than process context switch. An additional advantage is that local, inter-thread communication can be performed using shared memory. Further, the reduced thread context switch costs increase the scope for overlap of remote communication with computation.

To further reduce the cost of thread operations, user-level thread libraries have been proposed that obviate operating system intervention for thread creation, termination, context switch and scheduling [15, 26, 2]. Generally, user-level thread performance is an order of magnitude better than OS threads. Although both these thread-based approaches offer significant improvement over the use of OS processes, there are two main objections to thread based approaches.

First, threads export a shared-memory programming model that poses several obstacles in achieving an efficient implementation for distributed architectures. The root of many of these problems is the need to preserve the memory consistency imposed by the shared-memory programming model. Distributed shared memory (DSM) mechanisms exist that provide consistency at the granularity of the machine page size rather than the size of the actual data structure being shared [28].

Second, thread migration is complicated in the context of a shared-memory programming model because a thread's state can be implicitly changed by other threads through shared memory. Thus, accurate inter-thread, data-dependence information must be known about the application to achieve an optimal migration.

*The ULP system enforces this constraint.

[†]PVM library from ORNL, version 3.1.4.

In the absence of such information, application performance is limited to that achieved from a general-purpose DSM implementation. In contrast, ULPs do not have any implicit data-dependence amongst them as each ULP has independent register context, data and stack segments. All changes in a ULP state due to other ULPs occur through explicit messages between ULPs. Since a ULP state is so clearly delineated from other ULPs, ULP migration becomes a much simpler problem.

Further, thread-based approaches also have a practical disadvantage. Existing process based applications have to be extensively modified or rewritten to take advantage of threads. In contrast, our approach supports the familiar process programming model. This implies that existing programs employing the process model can directly benefit from ULPs.

The Data Parallel C (DPC) compiler and run-time environment [20] export a SIMD, shared address space model operated upon by a user-specified number of VPs. This number is usually much larger than the number of processors available. The multicomputer DPC compiler translates the SIMD DPC program source into SPMD C code. The SPMD C code is then compiled into an executable image using a C compiler and an OS process is created on each allocated processor. Multiple VPs are then emulated within each process.

Both DPC and the ULP system separate application-level parallelism from processor availability and make the efficient choice of one process per allocated processor. Another similarity between DPC and the ULP system is that dynamic load balancing is performed at the granularity of VPs. However, there are significant differences between the two approaches. The combination of the language and the SIMD programming model allow DPC to reduce VP emulation into simple indexing operations. All VPs share a single stack and have no special state to be saved or restored on a context switch. For example, switching to a new VP in the same process is simply a matter of using a new index for accessing the VP's data. Thus, heterogeneous migration is possible and OD costs are extremely small. However, the VP emulation model in DPC constrains VP migration to specific points in execution. Specifically, VP migration is possible only at the beginning or end of code segments that emulate a single VP.

In contrast, the ULP-based approach is more language independent and VP migration can be performed at any instant, except for certain small critical sections within the ULP system. Since an independent data, stack and register context is maintained for each ULP, preemption of a ULP is possible by simply saving its current register context and restoring it on resumption. In other words, the per-ULP context allows ULP emulation and migration to be independent of the execution state of other ULPs. However, these benefits are achieved at the cost of higher context switch overheads and the restriction of migration to homogeneous pools of processors.

Object-based systems such as Amber [11], Ariadne[27] and COOL [23] provide a programming environment that exports a thread-based object oriented programming model to the user. The objects share a single address space per application. The address space is distributed across the nodes in the network and the objects are free (with certain restrictions) to migrate from one node to another. Instead of providing these facilities in the context of an object creation and invocation model of programming, our approach aims to provide the same benefits in the context of a procedure oriented, process model.

In addition to these systems that provide finer-grained VPs, systems such as MPVM [10] support

a process-based programming model directly on OS processes and provide transparent migration at the granularity of these processes. When a node needs to be evacuated, an OS process on the node has to be migrated as a whole to the destination node. In contrast, UPVM implements the process abstraction *above* the OS in term of ULPs and provides support for both multi-threading and transparent migration. Since all ULPs of an application on a node share the same address space local communication can be implemented efficiently. Further, when a node needs to be evacuated, the ULPs within the source OS process can be migrated to different destination nodes thus providing a better potential for load-balancing.

3 UPVM Design

This section is organized as follows. Section 3.1 introduces PVM. Section 3.2 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 3.3, we remove these assumptions and discuss how the static, SPMD UPVM design can be extended to handle dynamic, program parallelism.

3.1 PVM

The PVM message-passing interface is designed to permit a network of heterogeneous UNIX computers to be used as a single 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 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 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.

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.

3.2 Design of UPVM

The overall design of UPVM is summarized in the Figure 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.

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 3.2.6.)

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.

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 4.) 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 [11]. See section 5 for implications of this approach.

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.

3.2.1 PVM independent objects and protocols

This section describes the main data structures and protocols that are used in ULP management and are not particular to UPVM. The primary data structures are the ULP descriptor, the ULP library descriptor, the heap descriptor, and the OS process descriptor.

ULP descriptor : The ULP descriptor (`ulpd`) maintains information about a ULP and there is one `ulpd` descriptor per ULP executing within an application and it contains the following types of information. It contains a *ULP identity* which is unique within a parallel application, the identity of the host OS process (PID) within which the ULP is currently resident. When a ULP migrates, this host process identity changes to the PID of the destination OS process. The descriptor also identifies the beginning of the stack and data

segments of this ULP within the process address space. Dynamic memory for a ULP is managed via a `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.

The *current state* of a ULP can be running, ready-to-run, blocked, migrated, or terminated. When a ULP is blocked, an event descriptor is maintained that identifies the event for which the ULP is blocked. On a context switch, the machine state of the processor is saved in the descriptor.

A ULP is marked *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 host process identity denotes where the ULP resides. Thus, if the application consists of N processes, there are N ULP descriptors for each ULP. At any instant of time, only one descriptor of a ULP is marked local and all other descriptors of the ULP are marked remote.

Two data structures `ubtab` and `umsgq` within the ULP descriptor directly relate to supporting the PVM interface and are discussed in Section 3.2.3. 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 I 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. 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 library descriptors, one per OS process. This descriptor primarily contains the ULP table, the local ULP run and wait queues, and a heap. The ULP table and the queues are used in scheduling while the heap is used dynamic memory requirements of the library itself and for implementing message passing. A *current ulp* structure denotes the currently executing ULP within the process. On a context switch (Section 3.2.5), `lcurulp` is updated to point to the new ULP. Also, a process descriptor is maintained that stores 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.

The heap : A heap provides the interface shown in Table II. A heap is first initialized with the block of memory it is supposed to manage. `hmalloc()`, `hfree()` and `hrealloc()` are similar in behavior 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 3.2.7.)

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.

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. 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 III. 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 of 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 3.2.6.

3.2.2 UPVM Initialization and address space layout

The simplicity of the following steps results from assuming that only applications 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:

- i. 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.
- ii. Determine how many ULPs to create. This number is made available to the library as a command-line argument.
- iii. Determine the locations and extent of OS process address space available in the process.
- iv. 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.
- v. 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.

- vi. 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).
- vii. 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.
- viii. Schedule the first runnable ULP from the run queue.

These steps complete the initialization process.

3.2.3 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 IV. 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`. `Btab` contains information about the mapping of each allocated `lbid` to a `gbid`. In addition, associated with each `lbid` 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 `btab` is through the interface given in Table V. How these objects are initialized and interact with each other is discussed in section 3.2.6.

3.2.4 Scheduling

The state transition diagram of a ULP in UPVM is given figure 5. 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. 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.

- 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.

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:

- i. 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 [8]). Otherwise, go to Step ii.
- ii. If there are runnable ULPs on the run queue, select the one at head of the queue and dispatch the ULP for execution.
- iii. If all ULPs of the application have terminated, then perform the application exit protocol to terminate the parallel application.
- iv. Otherwise, block within UPVM waiting for receipt of new messages or events.

3.2.5 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. 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 “migrated while running” then the function `asm_ufullLoad()` is called to load the entire register set. The three load functions are summarized in Table VI.

3.2.6 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 6, 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.

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 6). 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 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 VII.

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.

Unpacking routines (`pvm_upk*`) : The unpacking routines perform the reverse operation of the packing routines. (See table VIII.) 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 III 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** (ULP's **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_rcv) : **Pvm_rcv** (**uid**, **msgtag**) blocks the ULP until a message with the label **msgtag** has arrived from the **uid**. **Pvm_rcv()** 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_rcv()** is invoked by that ULP. Thus, when a **pvm_rcv()** 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.

Creating a new ULP (pvm_spawn) : For static, SPMD parallelism, this function is essentially a null operation. In section 3.3, we will discuss the design of this function to support dynamic ULP creation in the context of program parallelism.

3.2.7 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 7.)

- i. 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.
- ii. 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" acknowledgment 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.

- iii. 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.
- iv. 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.

3.3 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. 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 does the following: 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 to load the UPVM library at the same addresses on all processors so as to reduce the restart cost.

4 Performance Analysis

A UPVM prototype has been implemented on HP series 9000/720 workstations, the details of which are presented in [22]. To simplify the development and comparison with vanilla PVM, the prototype implements optimized communication, context switch and scheduling while using standard PVM library from ORNL for remote communication. The effect of this implementation decision is that the communication between remote VPs in UPVM is marginally slower than that between two remote VPs in vanilla PVM.

UPVM's performance 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.

4.1 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.

4.2 Micro-benchmarks

4.2.1 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 IX gives the context switch cost of ULPs and OS processes, both in absolute time and as a ratio to null procedure call cost.

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 [29]. 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 IX 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.

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

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 section 3). In other words, local ULP communication avoids 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.

4.2.3 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 XI, 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.

4.3 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.

4.3.1 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 XII. 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.

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 “neighbors” 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 XIII.) 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 XIII.) Specifically, UPVM performs at least twice as well as PVM for 8 or more VPs. 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.

4.3.2 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 neighboring VP to update its border-element strip. After 5000 sweeps, the 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 XIV shows 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 XV shows the results of the application running on two processors. The VPs are block-allocated, that is, VPs operating on neighboring 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.

Note the different performance trends of odd and even number of VPs in Table XV. 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 behavior 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. Thus performance degrades with the increasing overhead of supporting additional VPs.

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

4.4 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: (a) *Inherent method overhead* - the overhead an application incurs when using UPVM as compared to using a straightforward implementation (i.e., standard PVM) when no migration takes place, i.e., the overhead of UPVM in the quiet case. (b) *Obtrusiveness* - the time taken from the instant a migration event was received to the instant the application is “off” the processor, i.e., the impact on workstations owners when they want their workstation back and computation has to be moved away from their workstations. (c) *Migration cost* - 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, i.e., 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 section 3.) In addition, UPVM adds extra information for remote messages that result in marginally slower remote communication than PVM. Thus, the overheads incurred by 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 XVI. 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.

Obtrusiveness For measuring obtrusiveness and migration costs in UPVM, we benchmarked a neural-network classifying application called “Opt”. Opt based on conjugate-gradient optimization [3] 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 XVII shows 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 XVII.

However, notice a disturbing trend in the TCP/IP and the ULP obtrusiveness costs. As the data sizes increase, the curves diverge. Such behavior 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. This double copying is also the reason why the performance of MPVM [10] is better than UPVM for migrating a VP of equivalent size.

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 XVII.)

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.

4.5 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 section do not expose the scalability of the migration mechanism and are part of our future plan. However, the scalability of the migration mechanism can be discussed 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 acknowledgment messages in the current implementation. Given this migration cost, the affect of migration on the overall application performance is dictated by the pattern of migration in a real cycle-stealing system. This pattern in turn is dictated by the frequency of changes in node availability. Obviously, if this frequency is high enough, migration would result in decreased performance. On the other hand, infrequent migration which allows computation to remain load-balanced even in the face of machines becoming unavailable obviously results in better performance when compared to a non-migrating system that would be forced to stall when a machine was not available. In this paper, we have not attempted to explore such trade-offs because they

are environment specific.

Another feature that has not been exposed is the overlap of computation with communication. Over-decomposition allows a VP that would have blocked on a receive to be swapped out and another VP of the same application to be run its place. In a one-to-one mapping, the receiving VP would block leaving the machine idle. However, 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 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 of asynchrony due to network loads, we expect an over-decomposed application using UPVM to perform better than a one-task-per-processor decomposition using PVM and is an interesting experiment for the future.

5 Discussion

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 section discusses some of the main issues.

5.1 OS support for performance

The problems of supporting programming abstractions at user-level are well explored in the literature [26, 2]. 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 [2], first-class user-level threads [26] and new types of signals in the Solaris operating system [30].

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.

5.2 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 [4].

5.3 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 be 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 behavior 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 [20, 31], others, such as the DOME environment [5], 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.

5.4 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. The assembly language code required for this purpose is relatively small. For example, on the HP workstation the assembly code is about 300 lines compared to about 14000 lines of C code. Also, 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.

5.5 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) [33], 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 behavior 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.

6 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 paper 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. We showed how existing approaches are inappropriate and we defined a new light-weight, migratable VP abstraction called the user-level-process (ULP).

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. Finally, we discussed aspects of ULP systems that impact on their portability and support for heterogeneity, application transparency and debugging.

6.1 Future work

The UPVM prototype can be extended in many ways[22]. Here we mention some of them.

Integration with a Global Scheduler: While the paper 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 [18, 25, 12, 19, 16, 7].

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.

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 section. We plan to look into ways of integrating the

sand-boxing ideas into UPVM.

Multi-processor support: The UPVM prototype is implemented for uniprocessors and can be extended to provide support for shared-memory multiprocessors. Related research [6, 15, 11] 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 [21] and we plan to use this research as our starting point.

Acknowledgments

We thank the anonymous reviewers for their constructive comments on the paper.

References

- [1] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conference*, pp. 93–112, Atlanta, Georgia, 1986.
- [2] 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, Feb. 1992.
- [3] E. Barnard and R. 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.
- [4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto, and J. Walpole. PVM: Experiences, current status and future direction. In *Supercomputing '93 Proc.*, pp. 765–6, Portland, OR, Nov. 1993.
- [5] A. Beguelin, E. Seligman, and M. Starkey. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, Carnegie Mellon University, May 1994.
- [6] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, Aug. 1988.
- [7] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.
- [8] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.
- [9] R. Butler and E. Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National laboratory, 1992.
- [10] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2), spring 1995.
- [11] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proc. of the 12th ACM Symposium on Operating System Principles*, pp. 147–158, Litchfield Park, AZ, December 1989.

- [12] 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.
- [13] E. C. Cooper and R. P. Draves. C Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, Feb. 1988.
- [14] T. 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.
- [15] E. Felten and D. McNamee. Improving the performance of message-passing applications by multithreading. In *Proceeding of the Scalable High Performance Computing Conference*, pp. 84–9, Los Alamitos, CA, Apr. 1992.
- [16] D. Ferrari and S. Zhou. A load index for dynamic load balancing. In *Proc. of the Fall Joint Computer Conference*, pp. 684–690, Dallas, TX, November 1986.
- [17] 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.
- [18] T. P. Green and J. Snyder. DQS, a distributed queueing system. Technical report, Florida State University, Mar. 1993.
- [19] A. Hac and T. J. Johnson. A study of dynamic load balancing in a distributed system. *Computer Communication Review*, 16(3):348–56, Aug. 1986.
- [20] P. J. Hatcher, R. R. J. Anthony J. Lapadula, M. J. Quinn, and R. J. Anderson. A production-quality C* compiler for hypercube multicomputers. In *Proc. of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pp. 73–82, Williamsburg, VA, Apr. 1991.
- [21] D. Keppel. Register windows and user-space threads on the SPARC. Technical Report TR 91-08-01, University of Washington, August 1991.
- [22] R. Konuru. *A Migratable User-Level Process Package for PVM*. PhD thesis, Oregon Graduate Institute of Science & Technology, 1995.
- [23] R. Lea, P. Amaral, and C. Jacquemot. COOL-2: An object oriented support platform built above the Chorus micro-kernel. In *Proc. of the 1991 International Workshop on Object Orientation in Operating Systems*, pp. 68–72, Palo Alto, CA, Oct. 1991.
- [24] M. Litzkow, M. Livny, and M. Mutka. Condor — a hunter of idle workstations. In *Proc. of the 8th International Conference on Distributed Computing Systems*, pp. 104–111, San Jose, CA, June 1988.
- [25] M. Livny and J. Pruyne. Scheduling PVM on workstation clusters using condor. Technical report, University of Wisconsin at Madison, May 1993.
- [26] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pp. 95–109, Pacific Grove, CA, Oct. 1991.
- [27] E. Mascarenhas and V. Rego. Ariadne: Architecture of a portable threads system supporting thread migration. *Software—Practice and Experience*, 26(3):327–356, MAR 1996.
- [28] B. Nitzberg and V. Lo. Distributed Shared Memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, Aug. 1991.

- [29] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. of the Summer 1990 USENIX Conference*, pp. 247–256, Anaheim, CA, June 1990.
- [30] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *Proc. of the Winter 1991 USENIX conference*, pp. 1–14, Dallas, TX, Jan. 1991.
- [31] M. Thiemer and B. Hayes. Heterogenous process migration by recompilation. Technical Report CSL-92-3, Xerox Palo Alto Research Center, CA, 1992.
- [32] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–11, 1990.
- [33] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pp. 203–216, Asheville, NC, Dec. 1993.

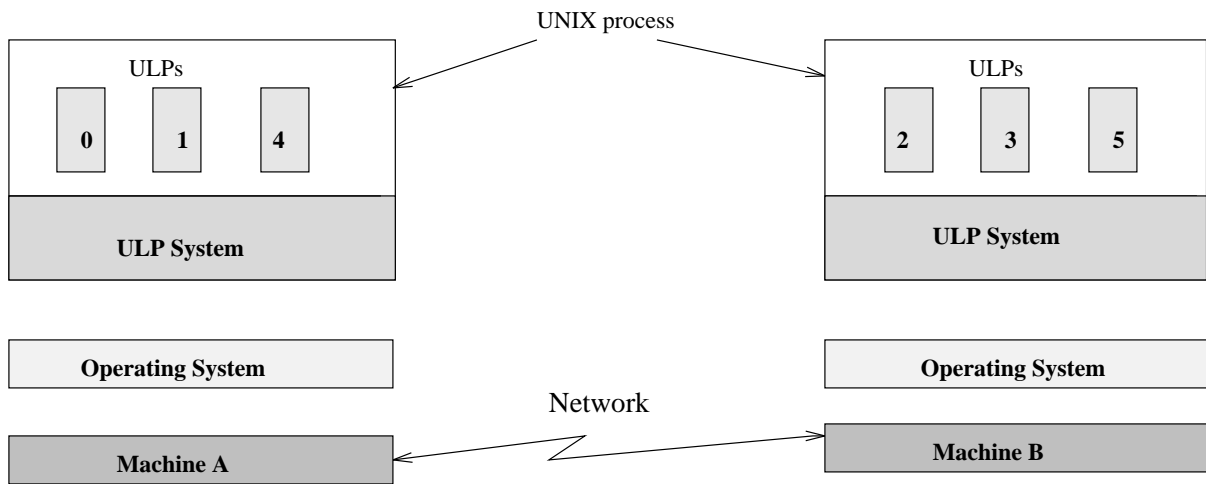


Figure 1: ULP System

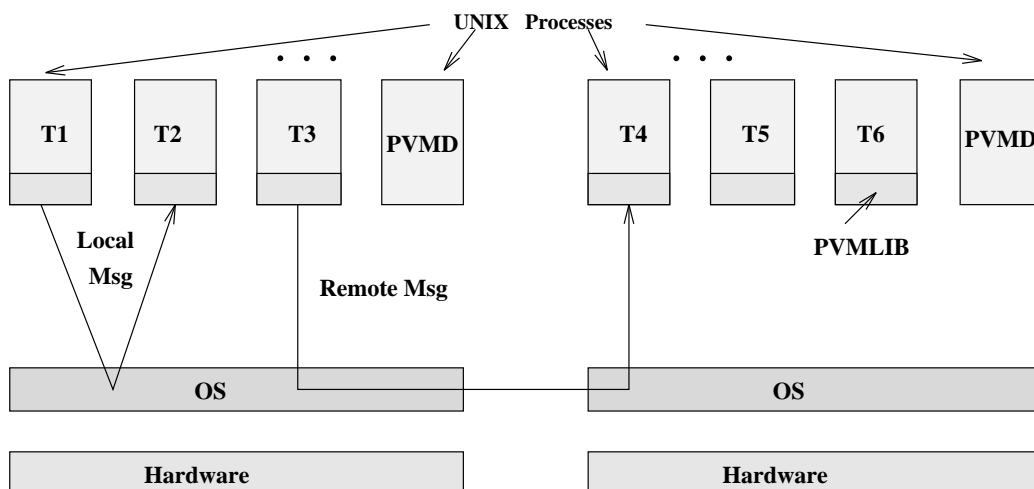


Figure 2: PVM system

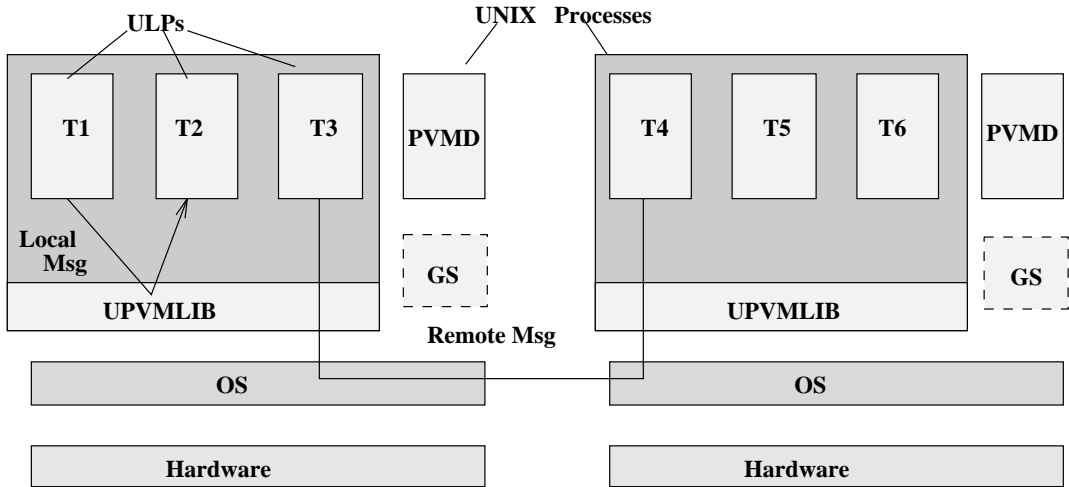


Figure 3: UPVM design

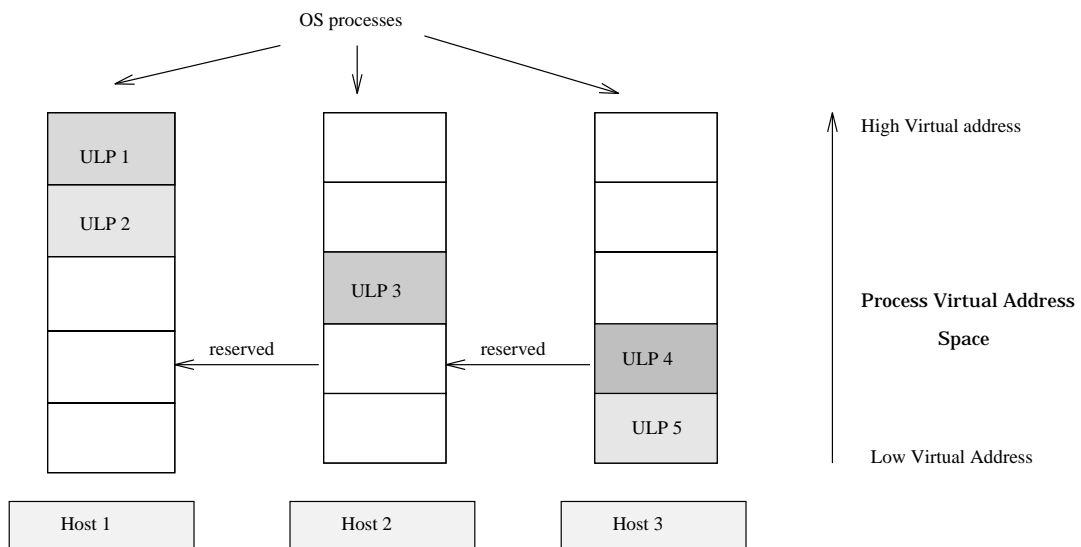


Figure 4: Per-Application, network-wide, virtual address space partitioning

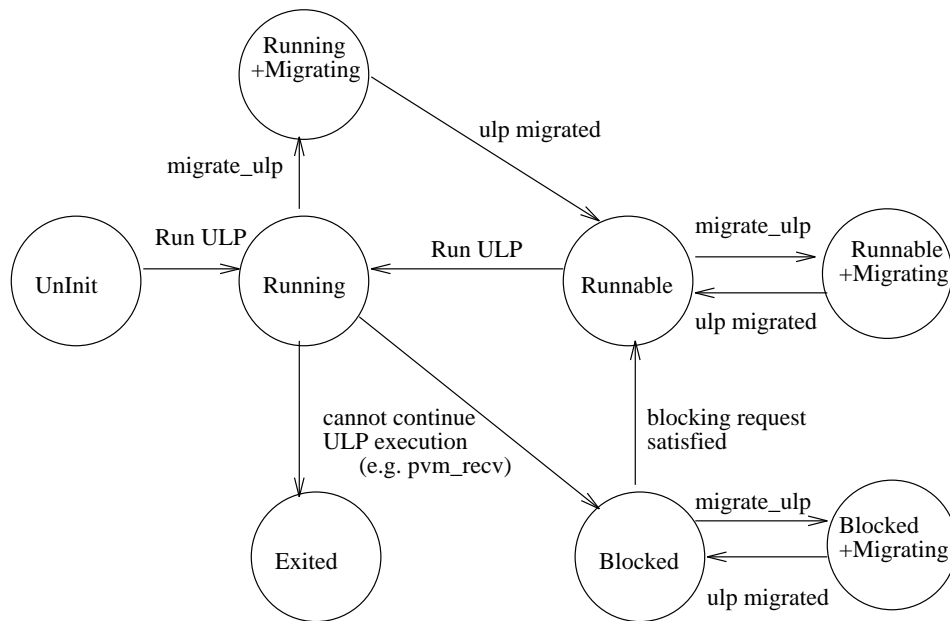


Figure 5: ULP state transition diagram

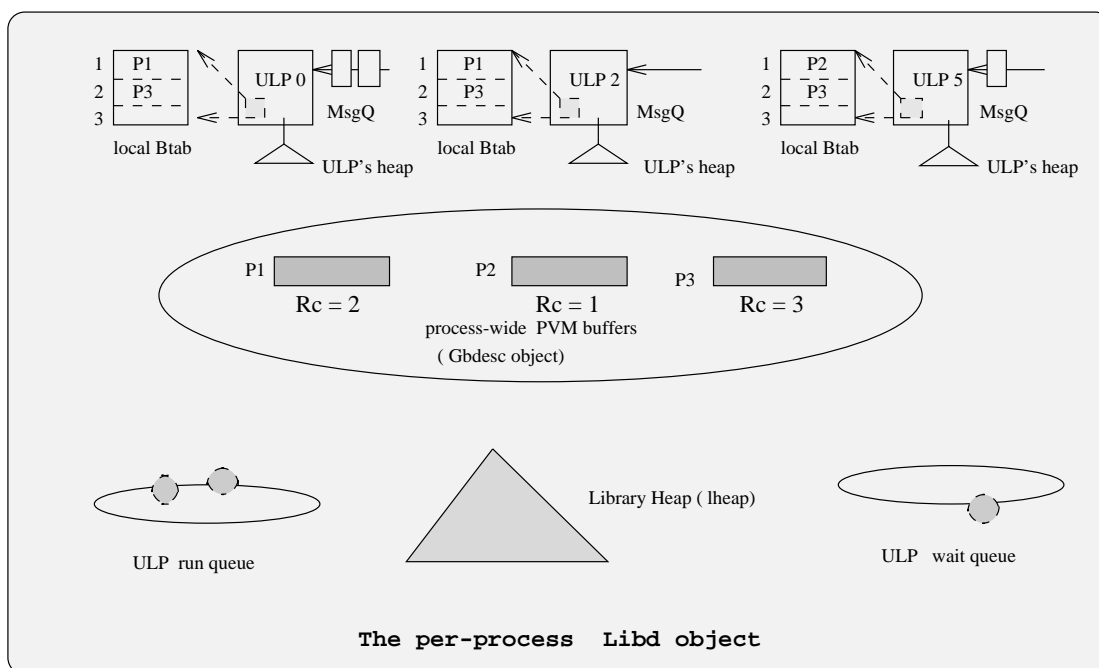


Figure 6: An example object execution scenario in UPVM

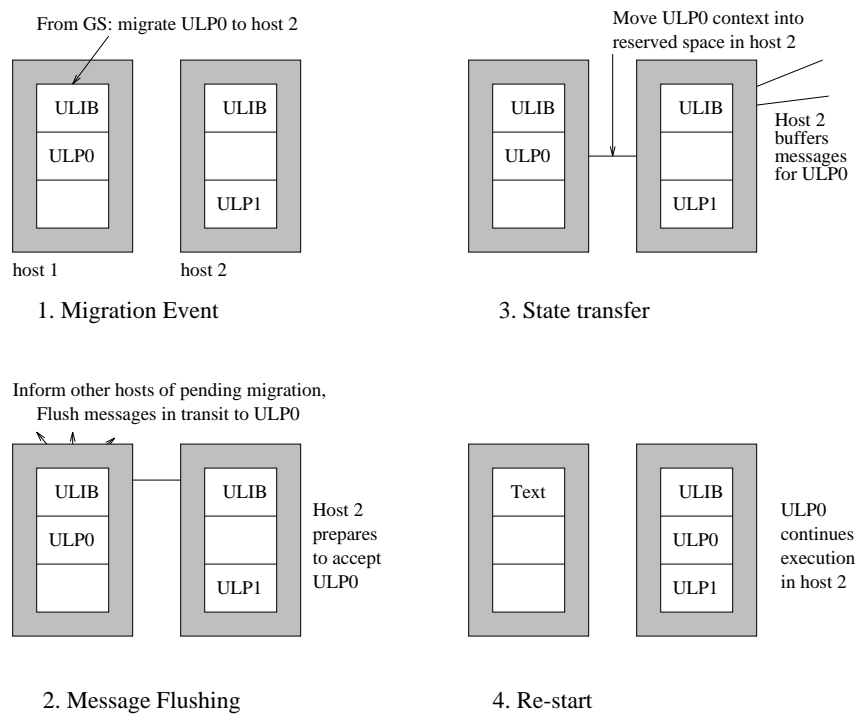


Figure 7: Stages in migrating ULP0 from host1 to host2

Table I: Functions exported by a ULP object

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 usestate (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 II: Interface to a heap object

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 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 III: 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	-	-

Table IV: Interface to Gbdesc object

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 V: Operations on a Btab 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, receive buffer id

Table VI: Functions to load ULP state

Name	Description
void asm_ustart (Ulp *u)	load registers for first ever execution of a ULP.
void asm_oload (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 VII: 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)

Table VIII: 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)

Table IX: Context switch costs (absolute and relative)

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

Table X: Local communication costs

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 XI: Remote communication costs

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 XII: Ring on one node

# 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 XIII: Ring on two nodes

# 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 XIV: LGS on one node

# 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 XV: LGS on two nodes

# 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 XVI: UPVM overhead over PVM

PVM (ms)	UPVM (ms)
4.82	5.01

Table XVII: UPVM: Obtrusiveness and Migration costs

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