Oregon Health & Science University
OHSU Digital Commons

CSETech

January 1994

A user-Level process package for PVM

Ravi Konuru

Jeremy Casas

Steve Otto

Robert Prouty

Jonathan Walpole

Follow this and additional works at: http://digitalcommons.ohsu.edu/csetech

Recommended Citation
Konuru, Ravi; Casas, Jeremy; Otto, Steve; Prouty, Robert; and Walpole, Jonathan, "A user-Level process package for PVM" (1994). *CSETech*. 329.
http://digitalcommons.ohsu.edu/csetech/329

This Article is brought to you for free and open access by OHSU Digital Commons. It has been accepted for inclusion in CSETech by an authorized administrator of OHSU Digital Commons. For more information, please contact champieu@ohsu.edu.

A User-Level Process Package for PVM*

Ravi Konuru, Jeremy Casas, Steve Otto, Robert Prouty, Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
{*konuru, casas, prouty, otto, walpole*}*@cse.ogi.edu*

Abstract

This paper describes an approach to supporting efficient processor virtualization and dynamic load balancing for message-based, parallel programs. Specifically, a user-level process package (UPVM) for SPMD-style PVM applications is presented. UPVM supports light-weight virtual processors that are transparently and independently migratable. It also implements a source-code compatible PVM interface, which means that existing PVM programs only need to be recompiled and re-linked. The performance of UPVM is discussed and compared with that of standard PVM.

1 Introduction

Processor virtualization is an attractive goal because it frees application programmers from the burden of managing physical processor location and availability. Virtual processors (VPs) allow programmers to think and code solely in terms of the parallelism within their application. Processor virtualization also improves system resource utilization because it allows systems software to transparently adapt to changes in processor availability, preemption, and load imbalance. Support for dynamic reallocation is useful in large multicomputers and essential in shared workstation environments.

Parallel processing packages, such as PVM [11] and P4 [4], use operating system (OS) processes as their VPs. 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.

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. Consequently, applications are forced to either suspend until the correct number of processors becomes available, or "double-up" on the remaining processors. In the latter case, application performance suffers due to operating system overhead and load-imbalance.

Alternatively, dynamic changes in processor availability can be managed within the application. In this approach, application programmers are responsible for 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), 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 is 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 OS processes leads to excessive overhead.

Attempts to address the high cost of OS processes have introduced a new OS abstraction called the *thread* [12, 9, 6, 19]. 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

*To appear in SHPCC'94 (contributed paper).

the thread abstraction above the OS at user level [7, 8]. 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.

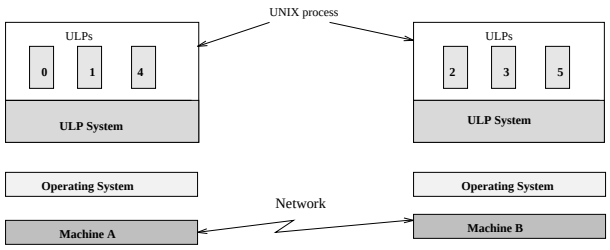

Figure 1: ULP System

From the application programmer's perspective, ULPs look like OS processes. Consequently, existing message-based, parallel applications that use processes as their VPs can use ULPs with little modification. From the ULP library's perspective, there are potentially many ULPs per OS process (see 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.[1] 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 UPVM, a prototype ULP system for PVM applications. Existing SPMD-style PVM applications typically require only recompilation 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 remainder of the paper is organized as follows. Section 2 presents related work. Section 3 outlines the implementation of UPVM. Performance results and a comparison with PVM[2] are presented in section 4. Section 5 discusses some of the main issues raised by this research and section 6 gives conclusions.

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 Chorus [12], Mach [9], V [6], and Solaris [19], 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 [10, 16, 1]. 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 [17].

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. 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

[1] The ULP system enforces this constraint.

[2] PVM library from ORNL, version 3.1.4.

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 [13] 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.[3] 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 [5] and COOL [15] 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.

[3] except for certain small critical sections within the ULP system.

3 UPVM implementation

In this section, we outline the implementation of the UPVM prototype on HP series 9000/720 workstations, running the HP-UX 9.01 operating system. Porting to other systems is discussed in Sec. 5.4. A more detailed description is provided in [14]. Some familiarity with the PVM interface is assumed.

When the application is invoked, control is immediately transferred into the UPVM library. At this point, the number of idle processors is determined, a process is created on each allocated processor, and the ULPs are created within these processes. The number of ULPs created is specified by the application programmer at startup. The number of VPs is a runtime parameter, but cannot be changed during the run — there is no facility for new ULP creation during the run. The amount of physical parallelism can change during a run, as VPs migrate to newly idle processors.

Each ULP has an opaque, location-independent identifier(UID) that is unique within an application. This identifier replaces the TID exported by PVM. All communication is performed using these UIDs. Thus applications that treat TIDs as opaque objects have no problem in using UPVM.

Each ULP is assigned distinct data, heap and stack regions in the address space of the process. In other words, each ULP has its own view of global variables and current execution state. **Malloc** and **free** functions invoked by a ULP result in operations on the per-ULP heap space. Furthermore, 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 10 ULPs across 5 processes. If ULP 1 is allocated a virtual address region V1 in process A, then V1 is reserved for ULP 1 in all other processes, even though ULP 1 is not present in those processes. This mapping allows a ULP to migrate from one process to another without requiring pointer manipulation.

HP-UX compilers generate all data references relative to a user-accessible register called the *data pointer* (DP). Thus for an SPMD application, for which all ULPs share the same code, a ULP context switch simply involves storing the general register state, stack pointer and DP, and restoring these for the new ULP. Since UPVM implements non-preemptive scheduling, only those registers that need to be saved on a procedure call (callee-save registers) are saved.

The non-preemptive scheduler manages a ULP *run* queue and a *blocked* queue. A ULP blocks when it executes a **pvm_recv** for which the required message has not arrived. The ULP chosen to run next depends on the location of the source ULP specified in the **pvm_recv**. If the source ULP is local, that

is, within the same process, and is runnable, UPVM schedules the destination ULP regardless of its position in the run queue. This technique is called hand-off scheduling[3].

The PVM buffer management interface is implemented by giving each ULP its own local view of buffer identifiers, the current send buffer and receive buffer. These local buffer identifiers (LBIDs) are mapped into global buffer identifiers (GBIDs) by UPVM. The global buffers are created within the library and are indirectly accessed by all ULPs within the same process using the PVM interface. It is possible for two or more ULPs to have LBIDs pointing to the same GBID. This indirection is used by UPVM to optimize local communication. When the destination of a message is a local ULP, the 'communication' is implemented by mapping the GBID of the send buffer into the LBID table of the destination ULP. Thus, copying of buffers from one ULP to another is eliminated. The LBID indirection is also useful when a ULP executes a **pvm_setsbuf(LBID)** call. If the global buffer mapped to the LBID is shared with other ULPs, then a new global buffer is allocated transparently to the ULP. The LBID-GBID consistency is managed through reference counting mechanisms.

Since UPVM implements the PVM interface, a lot of functionality provided by the standard PVM library needs to be duplicated within UPVM. To facilitate a quick implementation, the UPVM prototype is built on top of the PVM library. In other words, UPVM implements the ULP abstraction, heap space management, ULP scheduling, and local ULP communication and uses PVM for actual buffer creation, packing, unpacking, and remote communication. In fact, the GBIDs are the buffer identifiers returned by the PVM library. For example, a **mk_buf** call in UPVM is implemented by calling **mk_buf** in PVM, allocating a LBID in the ULP, and establishing a LBID-to-GBID mapping. Thus, the call has a slightly higher overhead in UPVM than in PVM. A similar approach is adopted to implement the packing and unpacking interface.

The decision to use PVM as the underlying platform for UPVM resulted in a certain complication. The PVM library makes the assumption that it is called in the context of a single process (i.e, a single data space), which is no longer true in the case of ULPs. Since each ULP has its own global variables, PVM code will access values from different locations depending on the ULP context in which it is executed. We solved this problem by allocating a specific data segment in which all PVM code is executed. In other words, for every PVM call, the DP register is changed to point to PVM's data segment, PVM code is executed, and the original DP is restored at the end of the call. Also, certain extra UPVM data is sent on each of the **pvm_send** calls that is used for protocol exchange between the different UPVM processes.

Finally, dynamic memory management routines such as **malloc** and **free**, provided by OS libraries, cannot be used since the routines need to use different heaps depending on the execution context. To get around this problem, we implemented our own memory management routines.

4 Performance Analysis

The performance analysis of the UPVM package is divided into two sections. We first present the results of certain 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 program that implements ring communication and one that implements a Laplace solver for two-dimensional grids.

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 9.01 operating system.

4.1 Context switch

The context switch benchmark measures the time taken for one VP (an OS process or ULP) to yield to another of the same kind. For comparison purposes, the cost of executing a null procedure call on the HP-UX workstation is 0.65 micro-seconds. Figure 2 gives the context switch cost of ULPs and OS processes, both in absolute time and as a ratio to null procedure call cost.

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

Figure 2: Context switch costs (absolute and relative)

Isolating the process context switch cost in a portable manner is extremely difficult, since there is no equivalent of a yield-to-another-process system call on UNIX. Our solution to this problem was to use Ousterhout's context switch benchmark [18]. 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 figure 2 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 Local communication

The local communication benchmark measures the round-trip message communication cost between two VPs. The benchmark is compiled with 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 in figure 3 are half

the round-trip cost. We assume that this closely approximates the one-way communication cost.

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

Figure 3: Local communication costs

The local communication cost of UPVM is around an order of magnitude better than that of PVM. This improvement can be attributed to two factors: the low ULP context switch costs, and optimized message passing that takes advantage of the shared address space(as described in section 3). In other words, local ULP communication avoids the cost of system call invocation, process context switch, message buffer copy from source process into kernel, and message buffer copy from kernel into the destination process.

4.3 Remote communication

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. This increase is visible in figure 4, which shows that UPVM costs are about 3.5 %, 3% and 1% higher for 1K, 10K and 100K message sizes respectively. The overhead is due to a combination of per-ULP buffer table operations, the reference counting mechanism, a locality check, and some run-time debugging code.

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

Figure 4: Remote communication costs

4.4 Ring communication

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 figure 5. 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 *blocked*. In the interleaved (**Intlv**) scheme, the application VPs are distributed over two processors such that every inter-VP communication is remote. In other words, VPs that are 'neighbours' in the ring are allocated to different processors. Thus, this is worst-case scenario in UPVM since there is no possibility for optimizing local communication. In contrast, the blocked (**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. Figure 6 summarizes the results for the two schemes.

# 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

Figure 5: Ring on one node

As expected, performances of ring on PVM and UPVM are comparable for the interleaved scheme. UPVM performs significantly better than PVM for the blocked scheme. Specifically, it performs at least twice as better as PVM for 8 or more VPs. This is because, as the number of VPs increase, UPVM gains more from its local communication optimizations. Thus, a suitable VP allocation scheme is a critical factor for UPVM in achieving high performance.

# 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

Figure 6: Ring on two nodes

4.5 Laplace grid solver

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

the 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.

Figure 7 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 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 above formula. This accounts for the performance improvement of UPVM over PVM for larger number of VPs. For example, at 11 VPs, PVM performance has degraded by about 16%, while UPVM has degraded by about 8%.

# 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

Figure 7: LGS on one node

Figure 8 shows the results of the application running on two processors. The VPs are block-allocated, that is, VPs operating on neighbouring portions of the grid are allocated to the same processor. Thus, remote communication is reduced to one pair-wise exchange of border strips, once per 10 sweeps.

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

# 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

Figure 8: LGS on two nodes

Note the different performance trends of odd and even number of VPs in figure 8. 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 is that of 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.

5 Discussion

Based on the experimental results, what can be said about the feasibility of over-decomposing PVM applications using UPVM?

The UPVM prototype has demonstrated an order of magnitude performance improvement over PVM for the communications on the same node. Applications, for which the amount of remote communication can be controlled, also perform better with proper allocation of ULPs to processors. The ULP allocation scheme is critical in achieving optimal performance. 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, this will result in large overheads. In our future work, we plan to optimize remote communication along with several other portions of the UPVM prototype.

The idea of user-level processes is one approach to the problem of providing light-weight OD 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. Some of the main issues are discussed below.

5.1 OS support for performance

The problems of supporting programming abstractions at user-level are well explored in the literature [16, 1]. Operating systems manage processes or OS 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.

Another significant problem is the decrease in the CPU time allocated to the application in a time-sharing environment. The OS allocates one CPU time quantum to a process even if there are multiple(say N) user-level abstractions executing within the process. Thus, instead of getting N time quantums per unit time, the process gets 1 time quantum that is shared among all the user-level abstractions. Because of such problems, care had to be taken in benchmarking the UPVM package.

However, most of these problems have been addressed in the context of user-level, thread-based systems using scheduler activations [1] and first-class user-level threads [16]. The mechanisms implemented therein provide the required OS support for user-level abstractions. Similarly, the Solaris operating system [19] implements new types of signals that can be used to achieve better integration of user-level libraries with the OS. We believe that future commercial operating systems will provide more support for integrating user-level abstractions.

5.2 Supporting a general purpose ULP

ULPs have been specifically designed 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, resource usage timers, etc, 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.

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 with the ORNL PVM group to define such an interface, called the Concurrent Processing Environment (CPE) interface, for PVM-based parallel applications[2].

5.3 Migration

Transparent ULP migration is one of the major goals of this work and is under current development. 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. This allows for ULPs that are created on one architecture to have overlapping addresses with the ULPs created on a different architecture.

Shared libraries present yet another problem for migration even within a binary compatible environment. These libraries are shared read-only by multiple executing processes on a workstation. When a process starts executing, certain variables within the user process are initialized by the dynamic loader so that process can access these shared libraries. These variables must be handled properly on migration.

Currently we restrict the scope of migration 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 procedure calling conventions of the OS need to be understood. These conventions determine the general and floating point registers that must be saved and restored in a ULP context switch. Since ULPs are laid out in distinct regions of a process virtual address space, the virtual memory layout, as defined by the OS, must also be taken into account.

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

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

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

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 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, one can 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.

6 Conclusions and Future work

The UPVM package provides processor virtualization for PVM. Currently, existing SPMD applications can be run using UPVM, usually with no modifications to the application source. Micro-benchmarks on the

initial UPVM implementation show an order of magnitude improvement over PVM in the local case. In particular, the over-decomposition results are encouraging. When virtual processors (ULPs) are allocated properly, performance of the application benchmarks on UPVM performs and scales better than on PVM. Our efforts are now directed toward the completion of the ULP migration mechanism and porting to other architectures.

Acknowledgements

We thank Jon Inouye and Khaled Al Saqabi for discussions.

References

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

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

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

[4] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National laboratory, 1992.

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

[6] David R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[7] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February 1988.

[8] Thomas W. Doeppner. Threads: A System for the Support of Concurrent Programming. Technical Report CS-87-11, Department of Computer Science Brown University, Providence, RI 02912, June 1987.

[9] Mike Accetta et al. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, Georgia, 1986.

[10] E.W. Felten and D. McNamee. Improving application performance by multithreading. In *Proceeding of the Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[11] G. A. Geist and V. S. Sunderam. Network-Base Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[12] Michel Gien. Micro-kernel design. *UNIX REVIEW*, 8(11):58–63, November 1990.

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

[14] Ravi Konuru, Jeremy Casas, Robert Prouty, Steve Otto, and Jonathan Walpole. A user-level process package for concurrent computing. Technical Report TR-93-016, Dept of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 1993.

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

[16] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, Pacific Grove, CA, October 1991.

[17] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

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

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