Portland State University
PDXScholar

Computer Science Faculty Publications and Presentations

Computer Science

5-1994

A User-Level Process Package for Concurrent Computing

Ravi Konuru
Oregon Graduate Institute of Science & Technology

Steve Otto
Oregon Graduate Institute of Science & Technology

Jonathan Walpole
Oregon Graduate Institute of Science & Technology

Robert Prouty
Oregon Graduate Institute of Science & Technology

Jeremy Casas
Oregon Graduate Institute of Science & Technology

Let us know how access to this document benefits you.

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

Part of the Computer and Systems Architecture Commons, and the OS and Networks Commons

Citation Details

"A User-Level Process Package for PVM," Ravi Konuru, Jeremy Casas, Steve Otto, Robert Prouty, Jonathan Walpole, In Proceedings of the Scalable High Performance Computing Conference, pp. 48-55, Knoxville, Tennessee, May 23-25, 1994.

This Conference Proceeding 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 User-Level Process Package for Concurrent Computing

Ravi Konuru, Steve Otto, Jonathan Walpole, Robert Prouty, Jeremy Casas
{*konuru, otto, walpole, prouty, casas*}*@cse.ogi.edu*

TR-93-016

Department of Computer Science & Engineering
Oregon Graduate Institute of Science & Technology
20000 NW Walker Road, P. O. Box 91000, Portland OR 97291-1000

Abstract

A lightweight user-level process(ULP) package for parallel computing is described. Each ULP has its own register context, stack, data and heap space and communication with other ULPs is performed using locally synchronous, location transparent, message passing primitives. The aim of the package is to provide support for lightweight over-decomposition, optimized local communication and transparent dynamic migration. The package supports a subset of the Parallel Virtual Machine(PVM) interface[Sun90].

1 Introduction

A user-level library implementing a new abstraction called ULP is currently being developed. Each ULP has a unique ULP identifier, a register context, a stack, data, and a heap space much like a UNIX process. ULPs of the same application execute within a single protection domain. Message passing is the only means of communication among ULPs and is achieved through locally synchronous[1], location transparent primitives. The aim of the package is to provide support for:

- *Static virtualization of processors*: Programs can be coded assuming a static number of ULPs irrespective of the availability of the physical processors. In addition, the number of processors available to an application can vary dynamically during application execution. A key goal is to maintain simplicity of message based application programming using static parallelism.
- *Over-decomposition*: An application can be mapped onto more number of ULPs than the maximum number of physical processors available.
- *Overlap of communication with computation*: An over-decomposed application can achieve overlap of communication with computation when more than one ULP is mapped to a physical processor(multi-threading). When a ULP blocks on a remote communication, other ULPs of the application on the same processor can be scheduled for computation. This scheduling, in effect, achieves the function of overlap. Since ULPs are user-level entities, their scheduling

[1] A return from a message send primitive implies that the buffer used in the send can can be reused. A return from a message receive primitive implies that the the requested message has been received.

and context switch requires no kernel intervention. Further, multi-threading through over-decomposition avoids the use of asynchronous message based programming and therefore greatly decreases the complexity of applications[FM92].

- *Transparent migration*: The ULP layer provides support for transparent migration of ULPs that perform all their non-computation related functions through the ULP interface. A higher level policy module, co-existing with the ULP package, can invoke the ULP package to perform ULP migration.

Note that the ULP abstraction defines a distributed memory programming model unlike a thread abstraction that is based on shared memory. A ULP defines a distinct register context, stack, data and heap space. On the other hand, a thread defines only a register context and stack. Having a ULP abstraction that clearly delineates the data manipulated by a ULP makes it an easier model to implement transparent migration.

This paper describes the design of a proto-type ULP package on a network of HP series 9000/720 workstations. The rest of this paper is organized as follows; The ULP programming model is described in section 2. The design of the ULP package is presented in section 3. The mapping of PVM interface[Sun90] to ULP operations and the scope for further optimizations is presented in section 4. The engineering choices made in the implementation of the ULP package raises several design and implementation issues and some of these are discussed in section 5. Finally, preliminary conclusions are presented in section 6.

2 ULP programming model

The programming model supported is SPMD(Single Program Multiple Data) with static physical parallelism, i.e., an application is coded as a single program and is instantiated into a statically specified number of ULPs[2]. An example SPMD program in pseudo-code is given in figure 1. Each instance of this SPMD program corresponds to an ULP. Each ULP has a unique identity and the send and receive primitives supported are locally synchronous.

Theoretically, the ULP programming model can support both SPMD and MIMD applications that exhibit dynamic physical parallelism. Further, the constraint that a ULP communicate only using locally synchronous message passing primitives can be relaxed to allow other semantics such as rendezvous or asynchronous message passing. The simpler SPMD programming model was adopted for the proto-type implementation as the model covers a large class of scientific programs and is more amenable to implementation.

3 Design of the ULP package

The ULP package is designed as a library that is linked to the application program. A static view of an SPMD executable using the ULP package is shown in figure 2. The interface used by the program(PVM in this case) is mapped to the machine independent ULP interface by a machine independent layer. The implementation of the ULP interface however is machine dependent.

The number of virtual processors(ULPs) required by the SPMD application is specified by user as command line option in addition to the actual application arguments. The library uses this

[2]The number of ULPs to be created is specified as an extra command line option to the SPMD program being executed in addition to the program specific arguments

Figure 1: SPMD program: A stylized example

```
/* Global Variables of the program */

          id_t          id;
          data_t        data[CHUNKSIZE];
          extern        int numproc; /* Number of ULPs created */
          result_t      result_value;
          result_t      total_result;

/* Main Program */
          Begin Main Program
             id = get_my_identity();
             if ( even(id) ) begin
                 result_value = process_even_data(id);
                 globalsum(result_value, &total_result);
             end
             else begin
                 result_value = process_odd_data(id);
                 globalsum(result_value, &total_result);
             end
          End Main Program
```

number in the creation and the management of ULPs. The overall design is discussed under the following subsections:

- Initialization & address space layout
- ULP Interface
- ULP descriptor structure
- Scheduling policy
- Context switching mechanism
- Migration mechanism

3.1 Initialization & address space layout

The application program is compiled and linked to the ulp package. The application is then executed by typing

application name -nu *number of ULPS* [application arguments]

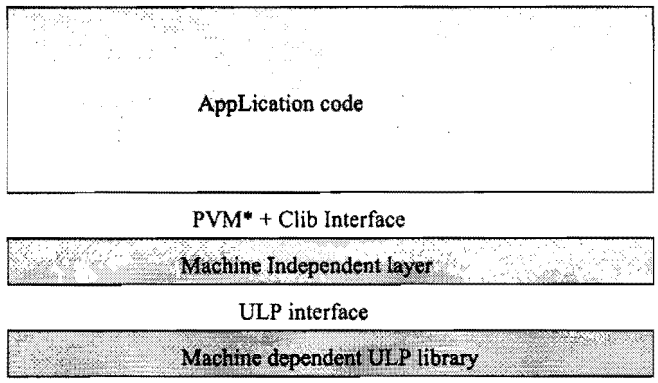

Figure 2: Static view of an SPMD executable using a ULP package

The ULP library initialization code is first executed before any code of the application. The initialization code performs partial ULP package data structure initialization and replicates the application program on a certain number of physical nodes based on the available resources. Each replica is a UNIX process consisting of the ULP library code and data structures in addition to the application specific code and data. The ULP library in each of these replicas has information on the configuration of the virtual machine and the host from which the application was started is recognized as a *home node*. The initialization code in each of these processes, then calculates the number of ULPs to be allocated on the corresponding host node, creates the ULPs and places them on the scheduling queue. Each ULP occupies a unique, per application, network wide virtual memory region as shown in figure 3. The figure shows an application divided into 4 ULPs on 2 processor nodes, 2 ULPs per processor. Note that each ULP occupies a unique virtual memory region. This region comprises the data segment, stack segment and the heap space of the ULP. Although the figure shows the virtual memory region to be a contiguous segment, this is not a necessary condition. The emphasis is on uniqueness rather than contiguity. The code region however is shared among all the ULPs within a UNIX process. This sharing is possible as the application is SPMD. The address space layout of ULPs was chosen to ease the process of ULP migration(see section 3.6).

Once the ULPs are created, the initial threads of control in each of the processes are then transformed into the first executing ULPs of the application. This completes the initialization process. Except for this initialization step which is pre-determined, the ULP library code is invoked during the application execution only under the following conditions.

- A ULP invokes a primitive that is either implemented by the machine independent layer or directly by the machine dependent ULP layer.
- A ULP invokes routines such as for memory allocation/deallocation that are implemented in the C library and have inherent assumptions of executing within the context of a UNIX process rather than in the context of a ULP. The memory management routines use sbrk(2) system call which manipulates the per UNIX process heap space. Since each ULP has its own heap space, use of these routines yields incorrect results. The memory management routines are the only few that are supported and are re-implemented within the ulp library. Implementing the entire UNIX interface is outside the scope of this research and is not supported in the proto-type(see section 5).

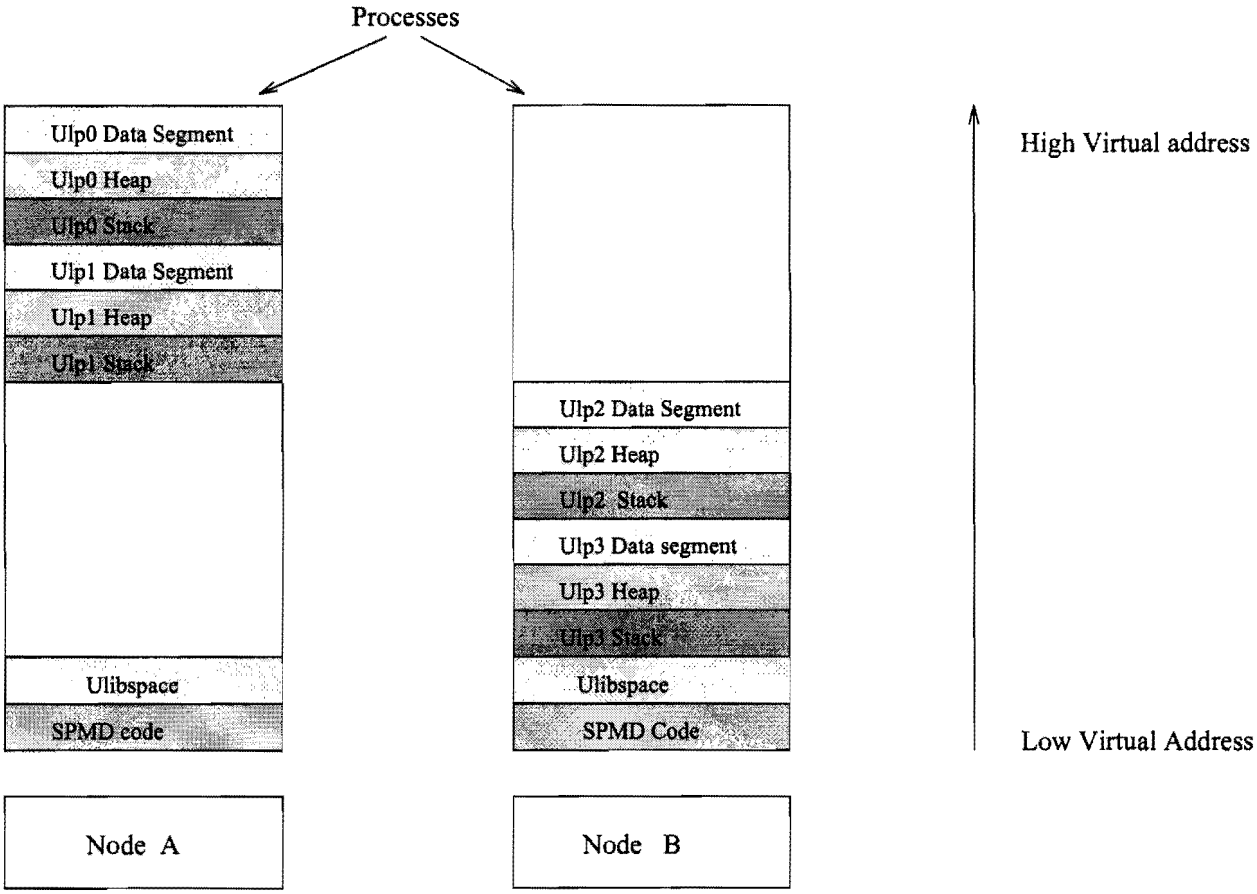

Figure 3: Address space partitioning of user level processes

- An asynchronous event is delivered to an ULP package by the operating system. Examples of such events are completion of I/O, timer expiration, urgent message on socket, workstation preemption, etc. The handling may result in the invocation of the scheduler.

In addition to the space used by the ULPs, there is a certain amount of virtual address space that is used by the ULP library itself. This space(`ulibspace`) contains various ULP tables, memory management data structures and scheduling queues. The same region of virtual address space is used by all the instances of the ULP library(one per physical processor). Note that this space is not directly migratable.

When a ULP is created certain variables in its data segment are initialized to point to data structures within the `ulibspace`. These variables can be thought of as type *constant* since once initialized, these variables are not modified by the ULP library nor the application. Examples of such variables are pointers to ULP tables, ULP run queue, ULP blocked queue, etc. These variables allow the ULP library to easily access its data structures irrespective of the ULP currently scheduled.

This design imposes the following requirements on the target platform: It must support a) a compiler that can generate position independent code and b) the ability to dynamically load and link object modules into an executing program. Requirement (*b*) is not necessary for SPMD

applications.

3.2 ULP interface

The ULP library exports a small and simple interface to ULPs. The functions in the ULP interface can be divided into four categories:process control, memory management, inter ULP communication, and file operations.

3.2.1 Functions for process control

There are three functions in this category:

- **ulpGetid()**: return the ulp identity of self.
- **ulpExit()**: terminate self.
- **ulpYield(ulpId)**: This call blocks the calling ULP and selects a new ULP to execute based on the value of the formal parameter `ulpId`. If the value of `ulpId` is ULP_ANY then the new ULP is selected based on the underlying scheduling policy. If `ulpId` is the identity of a local ULP then the local ULP is scheduled for execution. If `ulpId` is not a valid identity an error is returned to the calling ULP.

3.2.2 Functions for communication management

There are two functions in this category:

- **char* ulpMalloc(size)**: If there is sufficient unused per-ULP heap space, `size` bytes are allocated and a pointer is returned to the calling ULP. Otherwise a NULL value is returned.
- **ulpFree(addr)**: The space allocated to the ULP starting from `addr` is returned to the heap. An invalid value of `addr` is ignored.

3.2.3 Functions for communication management

There are four functions in this category:

- **ulpCbuf(size)**: A message buffer of size `size` bytes is allocated from `ulibspace` and a buffer identity is returned to the calling ULP. If there is no space an error is returned.
- **ulpKbuf(bufId)**: The buffer corresponding to this bufId is deallocated from the ULP name space. Usage of bufId after this call results in an error.
- **ulpSend(ulpId, bufid)**: The message in the buffer is queued for transmission to the ULP with identity `ulpId`. The buffer can be reused on returning from the call.
- **ulpRecv(ulpId)**: The calling ULP is blocked until a message from `ulpId` arrives. The message is accepted into a ULP library buffer and the blocked ULP is unblocked and put back on the scheduling queue. The identity of the message buffer is returned to the calling ULP.

3.2.4 Functions for file operations

These functions behave exactly like the standard unix file system calls except that in addition, they preserve location independence. The following functions are supported:

- **ulpOpen(char *filename, int oflag [, mode_t mode])**
- **ulpClose(int fd)**
- **ulpLseek(int fd, offset_t offset, int whence)**
- **ulpRead(int fd, char* buf, unsigned nbytes)**
- **ulpWrite(int fd, char* buf, unsigned nbytes)**

3.3 ULP descriptor structure

The ULP descriptor is shown as a C `struct` in figure 4. The fields `id`, `locn`, `state` are self-explanatory. `Host` points the node on which the ULP was last known to be present. `Sstart`, `dstart`, `tstart`, `treturn` point to the beginning of the per-ulp stack, per-ulp data segment, code entry and exit addresses respectively. Since we are implementing an SPMD model, the code entry and exit addresses of ULPs are identical. All the other fields mentioned above contain distinct values.

`Rc` points to the machine dependent register context. `Heap` is manipulated by the routines ulpMalloc() and ulpFree(). `Ftab` is a per-ulp file table and is manipulated through ulp file manipulation routines. `Btab` is a per-ulp table and is manipulated by the communication management routines. The pointer fields are used to link the ulp descriptor into various queues.

3.4 Scheduling

Scheduling among ULPS on the same processor is non-preemptive and occurs when

- a ULP makes a blocking call (for example, a blocking receive). If the data is already available the blocking call returns immediately. Otherwise the ULP is put on a blocked queue. If the source ULP of a blocking receive is located on the same processor, the the source ULP is next scheduled (hand-off scheduling). If the source ULP is remote, the first ULP in the run queue is scheduled for execution(FIFO).
- a ULP terminates. If there are runnable ULPs, the first ULP on the run queue is scheduled for execution. Otherwise the ULP library waits until a local ULP becomes unblocked, or the number of active ULPs in the application is greater than zero, or until an ULP is migrated to its processor.

When the number of active ULPs in an application becomes zero (all ULPs have terminated), the ULP library exits terminating the entire application.

3.5 Context switching Mechanism

Context switching between ULPs does not require kernel intervention. Further, since context switches occurs in a non-preemptive scheduling framework, relatively less state (callee-save registers) needs to be saved and restored. Since a ULP context switch involves one more register(the base

Figure 4: ULP descriptor

```
typedef struct ulp{
  UlpId          id;
  Locn           locn;                /* Local or remote */
  State          state;               /* Execution state */
  Node*          host;
  Regs           *rc;                 /* Register context */
  unsigned       sstart;              /* Stack begin */
  unsigned       dstart;              /* Data segment begin */
  unsigned       tstart;              /* Code Entry point */
  unsigned       treturn;             /* Code Exit point */
  Heap           heap;                /* heap management structure */
  int            argc;
  char**         argv;
  FileDesc       ftab[MAXULPFDS];     /* ulp file table */
  BufDesc        btab[MAXULPBDS];     /* buffer identity table */
  struct ulp*    self;                /* pointer to this structure */
  struct ulp*    next;                /* used when linked on queues */
  struct ulp*    prev;                /* used when linked on queues */
  } Ulp;
```

register that points to the data segment of the ULP) than a user-level thread(ULT) implementation, the context switch performance of ULPs and ULTs is almost identical.

3.6 Migration mechanism

In implementing transparent migration of ULPs, there are issues related to both policy and mechanism that need to be specified or isolated behind certain interfaces. The policy related issues of migration are:

- What are the constraints on ULP migration ?
- When should a ULP be migrated from a workstation ?
- Where should the ULP migrate to ?

A process executing on a UNIX like platform has access to the entire system call interface exported by the operating system. In our proto-type implementation, we restrict transparent migration to those processes that use ULP interface for all communication and file operations. All other process related functions (for example, signals, resource usage timers, pid, ppid) are not supported.

All policy issues related to detecting migration points and destination nodes for migration are handled by a global job scheduler in conjunction with a load monitor that runs as a separate process in the system (see fig 5).

Whenever migration is necessary[Op93], the global job scheduler asynchronously informs the ULP library that migration has to be performed and the destination of migration. It is the responsibility of the ULP library to implement the mechanisms for achieving migration.

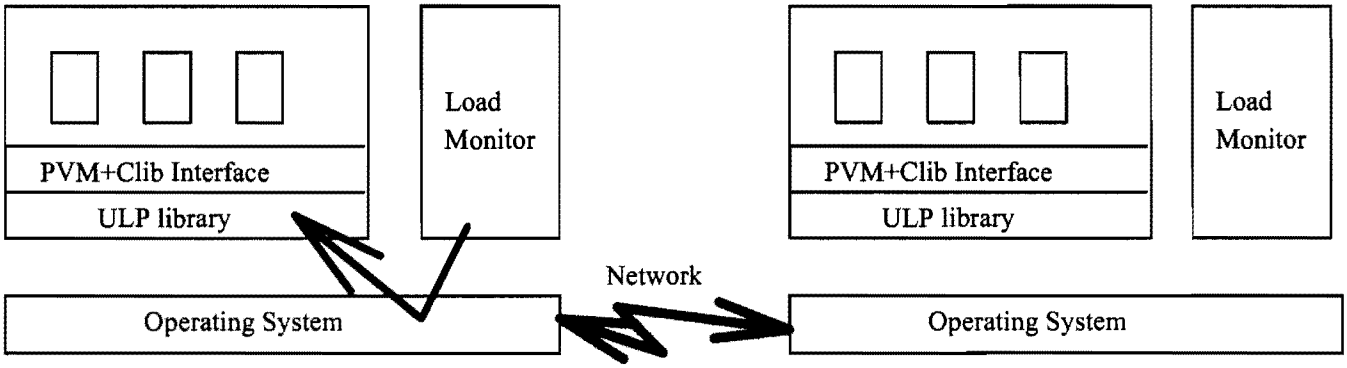

Figure 5: Software architecture of an ULP based system

Our algorithm for migration is based on ideas similar to those of Mike Powell et al[PM83] and Jonathan Smith et al[SI89]. A level of indirection is maintained by the ULP library between the file operations executed by the application and the file operations implemented by the operating system, i.e., there is an indirection file descriptor table maintained by the ULP library. An outline of the migration procedure is given below:

1. if the ULP library is not in critical section, start ULP migration. Otherwise update migration flag and information. All ULPs that attempt to send to this migrating ULP are blocked for the duration of migration.

2. Initiate communication with the destination node. This might involve creating an operating system level process in the case when that particular node is not being used by the application. The ULP library in the destination node initializes itself and receives the migrating ULP's descriptor, file indirection table, register context, text and data segments.

3. The ULP library on the source node drains the kernel buffers for any messages that were intended for this ULP into ulibspace. These buffers are then sent to the destination node's ULP library. The forwarding address for the migrating ULP is set to that of the destination node.

4. The destination ULP library re-establishes connections for remote communications with other processes transparently to the ULPs. It uses the register context in resuming the ULP. The information in the file indirection table is used to either communicate with the source workstation for file i/o or perform direct communication in presence of a location independent file system.

5. If the number of ULPs at a node is zero, the ULP library blocks until all ULPs of an application terminate or it needs to participate in an ULP migration. If all ULPs terminate, the ULP library terminates which in turn leads to the termination of the parallel application.

4 Implementing the PVM interface

To use existing applications, A subset of the PVM interface has been chosen for implementation on top of the ULP library. The functions supported are:

- pvm_mytid()
- pvm_exit()
- pvm_mkbuf()
- pvm_initsend()
- pvm_freebuf()
- pvm_send()
- pvm_recv()
- pvm_pk*()
- pvm_unpk*()
- pvm_set?buf()
- pvm_get?buf()

All communication is handled through buffers allocated in the PVM library. Applications refer to these buffers by *buffer identifier* only and not by a memory address.

Sending a message involves three steps. First a send buffer must be initialized by a call to pvm_initsend() or pvm_mkbuf(). Second a message should be 'packed' into this buffer by various packing routines provided. Then the send can be performed by a pvm_send.

Receiving a message involves specifying the source task and message tag in the pvm_recv() call. A wild card for the source can also be specified. The task is blocked until the message is received. On completion of the call the identity of the buffer containing the received message is returned.

4.1 Basic implementation

The ULP library implements ULP creation, termination, scheduling, context switch and local communication at user level. Thus all the above operations do not need kernel intervention and this in turn implies that ULPs do not incur the costs of entering the kernel and cross-protection domain communication.

Scheduling strategy is non-preemptive. This implies that state of ULP is saved only at well defined points (where the ULP executes blocking calls). This implies that the procedure calling conventions of the architecture and operating system can be used to reduce the amount of state. In other words, only the callee savee registers need to be saved.

In the special case of SPMD applications, the code segment is the same on all participating nodes. In this case, migration of a ULP from one participating node to another can be reduced only to movement of the data segment and the register context of the ULP at the source node to the destination node. This avoids the need and the operating system costs to create a new process on the destination node with the required execution context.

4.2 Optimized Implementation

There are several optimizations that can be applied to ULP scheduling and message passing within the ULP library. These stem from the characteristics of the PVM message passing interface. The PVM interface manipulates message buffers in terms of buffer identifiers only. The actual buffer addresses are not used. Each buffer has an associated encoding that can be specified by the application. The encoding can be `PvmDataDefault`, `PvmDataRaw`, `PvmDataInPlace`. The first two encodings determine how data is formatted before packing into the message buffer. The third encoding specifies that the data should be copied directly out of user memory rather than making an explicit copy of the data in the message buffer. Any message packing routines invoked result in recording the pointers to the source data in the message buffer. Any modifications of the source data after 'packing' and before a pvm_send() will be reflected in the sent message. Given these semantics, we now discuss some potential optimizations that can be performed in the ULP library.

If the encoding is `PvmDataDefault` or `PvmDataRaw`, a message send to one or more local ULPs can be achieved without actual copy of bytes to the destination ULP[s] by sharing the buffer read-only among all the ULP[s] involved in the communication. However, buffer management should be done carefully so that the message passing semantics are maintained. For example, if the sending process needs to send a new message using the same buffer id, a new buffer should be allocated transparently and by maintaining a level of indirection between the buffer identities seen by the process and the actual buffer identities managed by the ULP library.

The encoding `PvmDataInPlace` poses a different kind of problem. Since the actual data is in user space and not in the message buffer, the process can change this data unnoticed by the library. In this case, one approach is to perform an explicit copy of the message at the invocation of pvm_send() or pvm_mcast(). However, there is still some scope for optimization. First, a new buffer can be allocated and followed by explicit user level copy. If the message is being multi-cast and more than one ULP happens to be local, then this buffer can be shared between the multiple local ULPS.

Another approach to handling 'PvmDataInPlace' that can potentially avoid a copy all together is to switch to the local receiving ULP (L2) on the send operation from a local ULP(L1). Since L1 does not execute, it cannot modify its data. L2 can then copy the message directly into its own variables. However, there are problems with this approach. It is not clear when L2 is done with reading the received message. It is possible that the L2 reads only a subset of the message, or it does not do an explicit pvm_freebuf() to convey to the library that it has completed processing the received message. In these extreme cases, the L1 may have to be blocked until the receiving process terminates. If L2 in turn performs a pvm_send() to L1 before doing a pvm_receive() (this is a valid sequence according to the synchronous message passing semantics), we then have a deadlock situation. We can recover from this dead lock by performing an explicit copy of the message into a newly allocated message buffer, update the buffer identity indirection table and resuming the ULP L1. Since this deadlock situation can occur when multiple ULPs form ring communication with sends followed by receives, there should be enough information recorded in the ULP library to efficiently detect and recover from this deadlocks.

5 Unresolved Issues / Discussion

The idea of user-level processes address the issues of light weight over-decomposition and transparent migration for message based communicating processes. However there are several other issues that have been glossed over while describing the design of the ULP library and are discussed below:

5.1 Supporting a truly user-level process

Supporting a truly user-level process implies that all functionality and interface supported by an operating system level process should be exported to the programmer. This implies that interfaces such as signals, process identifiers, scheduling assumptions (pre-emptive scheduling), resource usage timers, etc, need to be supported. Further, an operating system level process abstraction defines a protection domain. A ULP on the other hand shares its protection domain with other ULPs of the application. However, supporting all this functionality at user-level would add overheads for those applications that do not need this full generality. Ideally the ULP abstraction should be implemented in such a way so that extending the functionality of a ULP is a straight forward activity.

5.2 Heterogeneity

ULP migration is performed between workstation architectures that are binary compatible. Heterogeneity is possible but restricted in the ULP environment. An application can be executed such that some its workstations are of say, type A and others are of say, type B. Then the ULP library maintains two virtual address space partitionings, one for type A, and one for type B and allow the migration of ULPs among the same type. This design raises questions about boundary conditions: what happens when there is only one machine of a type and it has to be evacuated ?

There can be several solutions to this problem. These include a) Terminate the entire parallel application b) The ULPs on the workstation can be suspended until the workstation is available for reuse. This would potential halt the entire parallel application's progress. The job scheduler can reallocate these workstations to another application, c) Suspend the ULPs on the workstation and send a signal to the user program allowing the users to specify their own application specific recovery. Among these three options, option 'c' seems to be the best but least transparent. Availability of application specific information to the ULP library through some higher level entity such as a compiler would remove the drawback of loosing transparency.

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 user can access these shared libraries. However on migration, these variables are usually not valid and can lead to unpredictable results. Currently we restrict the scope of migration to programs that perform static linking.

5.3 Portability

A practical aspect that needs to be considered in using any package is its portability to new architectures. Ideally, the porting process should involve only a compilation on the target workstation with the appropriate switches. However, it is not as easy with the ULP package. The tasks that need to be done can be divided into those that are specific to the portability of the ULP package and those that are necessary for the correct operation of the ULP package.

The compilers on target workstation should be able to generate position independent code with respect to some user accessible general register so that the object code can be loaded into any region of a process virtual address space. Further, the operating system should provide for an interface to dynamically load and link code and data modules into an existing virtual address space. Once these tools and interface are available, routines then need to written for the the machine dependent parts of ULP creation, termination, and context switch.

In addition, the target workstation should be configured with a load monitor daemon that communicates with a global job scheduler. This provides the hook for the ULP library to perform transparent migration.

5.4 Protection, Debugging and Programming language

ULPs of an application execute within the same protection domain unlike operating system level processes of an application. This brings up the issues of programming language and debugging. Since this a user-level package, a normal operating system process debugger does not recognize these ULPs reducing the ease in debugging of an application.

If the application is programmed in a language like C that allows language level pointer manipulation, a buggy program could corrupt the contents of the ULP library or the context of another local ULP leading to unpredictable bugs that are difficult to isolate. However, if an application is programmed in a language like FORTRAN that that does not allow pointer manipulation, then it is much easier to isolate the bugs caused by one ULP from another. This argument about language and protection can be extended to include the whole programming system, i.e., given a language that guarantees correctness and safety, it should be possible to have the operating system and all applications to execute within a single protection domain. This goal of having a single protection domain programming system however is outside the scope of this research.

6 Conclusions

A user level process package is presented that aims to perform light weight over decomposition and transparent migration for message passing parallel applications on shared multi-computers. The application is expected to be programmed in terms virtual processors based on maximum application level parallelism. The ULP package in conjunction with a load monitor and global scheduler performs migration and reconfiguration when necessary.

The design of the ULP package has been outlined and important issues that arise due this design have been discussed. The package is currently being implemented on a network of HP 9000/700 series workstations.

References

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

[AE85] R. Agrawal and A. K. Ezzat. Processor sharing in NEST: A network of computer workstations. In *The 1st International Conference on Computer Workstations*, pages 198–208. IEEE, 1985.

[CAL+89] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. Technical Report 89-04-01, Dept of Computer Science and Engineering, University of Washington, 1989. Also in 12th ACM SOSP.

[FM92] E. W. Felten and D. McNamee. Improving the performance of message passing applications by multithreading. Technical Report 92-09-07, Dept of Computer Science and Engineering, University of Washington, 1992.

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

[MSLM91] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, Pacific Grove, CA, October 1991. ACM.

[Op93] Ogi-pvm. A distributed load manager for shared workstation networks. Technical report, Oregon Graduate Institute of Sci. & Tech., 1993. In Preparation.

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

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

[SCSK93] M. Swanson, T. Critchlow, L. Stoller, and R. Kessler. The design of the schizophrenic workstation system. In *Usenix Symposium Proceedings*, pages 291–306, Santa Fe, NM, April 1993. USENIX.

[SI89] J. M. Smith and J. Ioannidis. Implementing remote fork() with checkpoint/restart. *IEEE Technical Committee on Operating Systems Newsletter*, 3(1):15–19, Winter 1989.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.