MPI Process Swapping: Architecture and Experimental Verification

Otto Sievert* Henri Casanova*†

*Department of Computer Science and Engineering
University of California at San Diego

†San Diego Supercomputer Center
University of California at San Diego

otto@cs.ucsd.edu, casanova@sdsc.edu

Abstract—

Parallel computing is now popular and mainstream, but performance and ease-of-use remain elusive to many end-users. There exists a need for performance improvements that can be easily retrofitted to existing parallel applications. In this paper we present *MPI process swapping*, a simple performance enhancing add-on to the MPI programming paradigm. MPI process swapping improves performance by dynamically choosing the best available resources throughout application execution, using MPI process *over-allocation* and real-time performance measurement. Swapping provides fully automated performance monitoring and process management, and a rich set of primitives to control execution behavior manually or through an external tool. Swapping, as defined in this implementation, can be added to iterative MPI applications and requires as few as three lines of source code change. We verify our design for a particle dynamics application in a commercial production computing environment.

1. Introduction

While parallel computing has been an active area of research and development for several decades, it is still a daunting proposition for many programmers and end-users. In order to alleviate the burden for the developer, a number of programming models have been proposed. Perhaps the most popular such model is the message passing paradigm, which provides low-level abstractions for process communication and synchronization. The Message Passing Interface (MPI) standard [7] has been embraced by the parallel computing community and a large number of MPI applications have been developed in many fields of science and engineering. While MPI provides the necessary abstractions for writing parallel applications and harnessing multiple processors, the primary parallel computing challenges of application scalability and performance remain. These challenges can be addressed via intensive performance engineering and tuning by parallel computing experts for particular applications and platforms, but typical end-users (e.g. disciplinary scientists) often lack the time and expertise required. As a result, many end-users sacrifice performance in exchange for ease-of-use.

Parallel computing generally enjoys ease-of-use or high performance, but rarely both at the same time. Applications that provide easy access to parallel execution environments, by hiding or abstracting parallel computing details, often fail to yield high performance *because* such details are hidden. Intelligent infrastructure is one way to break this tension between ease-of-use and performance. One challenge here is to make such infrastructure seamlessly accessible to application

This material is based upon work supported by the National Science Foundation under Grant #9975020.

developers, and thus directly applicable to existing applications. A simple technique or tool that provides a sub-optimal (but still beneficial) improvement can be more appealing in practice than an optimal solution that requires substantial effort to implement.

We primarily focus on the impact of fluctuating resource availabilities on application performance. While this issue is not directly relevant for space-shared resources (e.g. batch-scheduled MPPs and clusters), it is critical for time-shared environments, such as Networks of Workstations (NOWs) and ensembles of desktop resources. This latter type of platform is cost-effective and commonplace, having steadily gained in popularity and performance in arenas such as enterprise computing. We target heterogeneous execution environments in which the available computing power of each processor varies throughout time due to external load (e.g. CPU load generated by other users and applications). In this paper we propose a framework that makes it simple to dynamically assign application computation to MPI processes based on performance fluctuations of the resources. An important point is that our work focuses on improving overall application performance (as opposed to addressing fundamental issues of fault-tolerance, for example). We demonstrate our approach for the broad class of iterative applications. For these applications, our processor swapping enhancement can be included with only a few simple modifications to the source code of existing MPI applications. As a result, potential performance benefits come at virtually no cost for the developer and end-user. We present experimental results obtained with our system in a production commercial environment.

The remainder of this paper is organized as follows. Section 2 discusses related work and, in particular, puts this work in perspective with efforts in the area of process migration and fault-tolerance. Section 3 introduces the concept of MPI process swapping and Section 4 describes the run-time swap architecture and the swapping source code architecture. In Section 5 experimental results are presented. Future directions of this work are described in Section 6. Finally, Section 7 concludes the paper.

2. Related Work

Our implementation of MPI process swapping is a sleight-of-hand played in MPI user space, rather than a true infrastructure feature. A checkpointing facility such as that provided by the recent MPICH-V [2] or by Co-Check [14] would improve the capabilities of this system. These checkpointing/migration mechanisms could be combined with the process swapping services and policies, improving the robustness and generality over the current process swapping solution. In particular, a checkpointing facility would allow a better process swapping implementation by (a) removing the restriction of working only with iterative applications; (b) further reducing the source code invasiveness (there would be no need to register iteration and other critical variables, or to explicitly make a function call to determine whether to swap or not); and (c) reducing or removing the need to over-allocate MPI processes at the beginning of execution.

If MPI process swapping were to be combined with the cycle-stealing facilities of Condor [11], a more powerful system would result. Condor uses migration primarily to allow the capture of spare CPU cycles on idle personal workstations; when a resource owner starts to use their machine, Condor evicts the process. By combining swapping policies with Condor's eviction mechanism, a process might also be evicted and migrated for application performance reasons. Such a combined system would not only provide high throughput, but individual application performance as well. One difficulty would be to allow network connections to survive process migration. An approach like the one in MPICH-V (discussed above) could be used.

MPI process swapping shares performance ideas and methodologies with traditional applica-

tion schedulers such as those found in the AppLeS [1] and GrADS [10] projects. These systems are also concerned with achieving high performance in the face of dynamic parallel execution environments. Additionally, they strive for ease-of-use, knowing that common users such as disciplinary scientists are often not parallel computing experts. The performance measurement and prediction techniques used in process swapping share much with these projects; all use application and environmental measurements to determine future execution characteristics that improve application performance (e.g. via the NWS [15] or MDS [6]).

3. MPI Process Swapping

MPI process swapping is a simple performance-enhancing add-on to standard MPI 1.1. It works with new and existing MPI programs, providing an intelligent infrastructure that increases performance in shared computing environments. Process swapping automatically determines the best processors to use when running an MPI application. During execution, the system periodically checks the performance of the machines in its pool, and swaps the application processes from slow processors to fast processors. Because MPI process swapping improves performance by selecting the best performing processors, it is useful in environments where the processor pool is shared. Typical examples of such environments are networks of workstations, workstation clusters, and computational grids. Such environments are common in enterprise computing and production environments such as those found in academic and commercial research and development facilities. Process swapping is not useful in space-shared parallel environments such as those typically found in MPP and other supercomputing environments, where a processor is dedicated to an application for the lifetime of that application, and the pool of processors is accessed via a batch scheduler.

MPI process swapping is neither a sophisticated nor an elegant methodology. It is not a universally optimal performance methodology for pristine parallel execution situations. It is a practical solution for practical situations. Other techniques, such as checkpoint/restart and dynamic load balancing, provide elegance and performance. However, techniques like these require substantial application support or restricted execution environments. By comparison, the very simple process swapping often provides a better price/performance ratio.

Dynamic load balancing is one of the best known methods to fully utilize heterogenous processing power [3, 4, 5]. Its performance overhead can be tuned by altering the granularity of the work, and is often very low. By its very nature it efficiently uses processors of widely disparate computing power by redistributing load (unlike MPI process swapping, which can swap processors but is constrained to use the initial data partitioning). In short, dynamic load balancing is a very good performance enhancing technique. Load balancing is not without its limitations, however. First, dynamic load balancing requires an application (algorithm) that is amenable, in the limit, to arbitrary data partitioning. Some algorithms demand fundamentally rigid data partitioning. The second limitation of dynamic load balancing is the development effort required to support it. Support for uneven, dynamic data partitioning adds complexity to an application, and complexity takes time to develop and effort to maintain. Lastly, the performance of an application that supports dynamic load balancing is limited by the achievable performance on the processors that are used. A well-balanced execution can still run slowly if all the processors used operate at a fraction of their peak performance.

Checkpointing is not limited to the processors on which execution is started, so it does not have to remain running on a set of slow processors. It also does not require a sophisticated data partitioning algorithm, and can thus be used with a wider variety of applications/algorithms. Unfor-

tunately, parallel heterogenous checkpoint/restart is a difficult task; it remains the subject of several active research projects.

By hijacking common MPI functions we can offer sustained execution efficiency in dynamic environments, with very small effort. Relative to these other techniques the potential performance improvement is similar, depending on the execution environment characteristics. Figure 1 illustrates the relative performance potential and implementation effort of process swapping versus dynamic load balancing and checkpoint/restart.

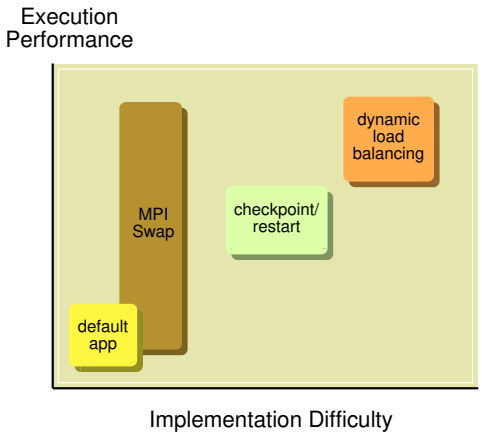

Fig. 1. Swapping brings potential performance benefits with relatively low effort.

MPI 1.1 does not support the adding and removing of processors to the global communicator, so MPI process swapping relies on overallocation of processes at the beginning of execution to get a pool of possible processors. Swapping chooses the best subset to actively participate in the application execution; the rest remain inactive. MPI 2 has support for adding and removing processors to a communicator [12]. However, MPI 2 it is not widely supported, and the communicator modification functionality is not transparent, requiring significant source code modification for existing MPI 1.1 applications.

Swapping does not increase or decrease the total number of active processes used to execute a program — parallel applications generally have a parabolic speedup curve and operate most efficiently on a particular number of processors. Swapping simply chooses the best processors to use.

Because it intelligently decides which processors actively participate in program execution, process swapping is better than simply replicating work. The simplest work replication option is to execute the application twice. In a dynamic environment, however, it is likely that at least one processor used by each replicated run will have decreased performance, causing both applications to execute slowly. In this case, performance will suffer even though twice as many resources are used. Doubling work units within the application, using the first available results, and abandoning the other results, can also in general be hindered by slow processors. This method also requires significant modification to the application itself.

Swapping automatically determines the best processors to use for a run of an MPI application. During execution, the swapping run-time services periodically check the performance of the machines in its pool, and swap the application to run on the fastest available machines. Inactive MPI processes utilize very little computational power; aside from periodic active performance measurement, they block on I/O calls and wait to become active.

4. Swapping Architecture

MPI process swapping is implemented as a set of run-time services that interact with a modified MPI library interface. The run-time architecture for a swappable application comprises five main components: the swap-enabled MPI application itself, swap handlers, a swap manager, a swap dispatcher, and the swap tools.

4.1. Process Swapping Run-time Architecture

Figure 2 shows the swap run-time architecture, and describes the communication patterns between the swap components. The swap handler modules are transient network services; a

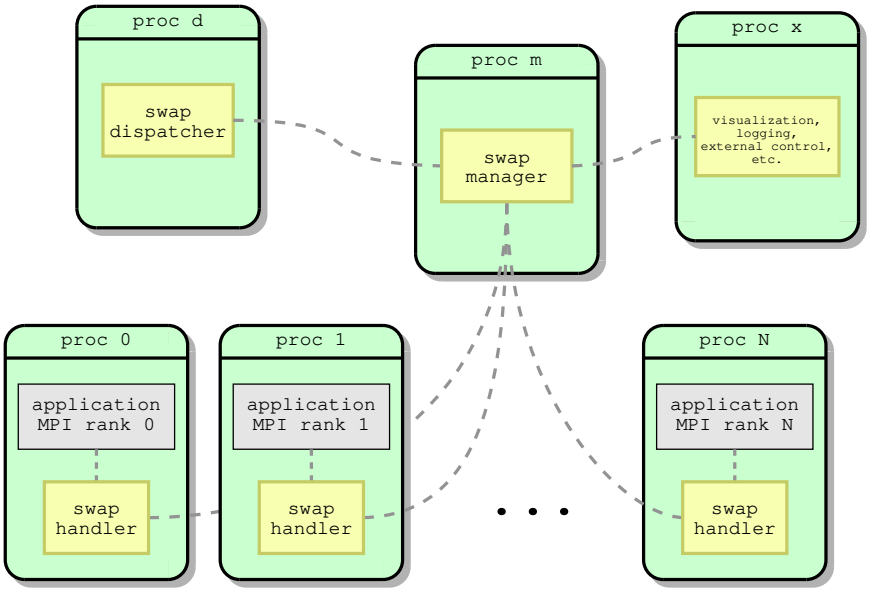

Fig. 2. Swap run-time architecture.

swap handler module is started for each MPI process (active and inactive) in an MPI application. It lives only as long as the MPI application lives. The swap handler module is the main communication link between the application and the other swap components. Because it resides on the same host as the MPI process that it shepherds, swapping-related communication delays are minimized. In addition to being the communication portal between the application and the swap services, the swap handler also contains performance measurement capabilities.

Each application is associated with one swap manager. The swap manager is the intelligence of the swapping operation. Information from each MPI process and each processor is sent to the swap manager. Using its swap policy, the manager analyzes this information and determines when and where to swap processes.

The swap dispatcher is an always-on remote service at a well-known location (network host/port). The dispatcher fields requests for swapping services and launches a swap manager for each application. Additional services may contact the dispatcher in order to establish communication with existing swap managers.

The swap tools are a collection of utilities designed to improve the usability of the swap environment. Facilities such as swap information logging and swap visualization connect to the swap manager (possibly through the swap dispatcher), and track an application's progress. The swap actuator provides a simple interface to manually force a swap to occur.

The swap services interact with the MPI application and with each other in a straightforward asynchronous manner, as illustrated in Figure 3. Walking through an example application execution will further describe these interactions. First, from machine `u` a user launches an MPI application that uses `N` total processes, a subset of which will be active at any given time. The root process (the process with MPI rank zero) on machine `0` contacts the always-on swap dispatcher (running on machine `d`) during initialization, and requests swap services. The swap dispatcher launches a swap manager on machine `m`. The swap dispatcher waits for the swap manager to initialize, then tells the root process how to contact this personalized swap manager. The root process passes

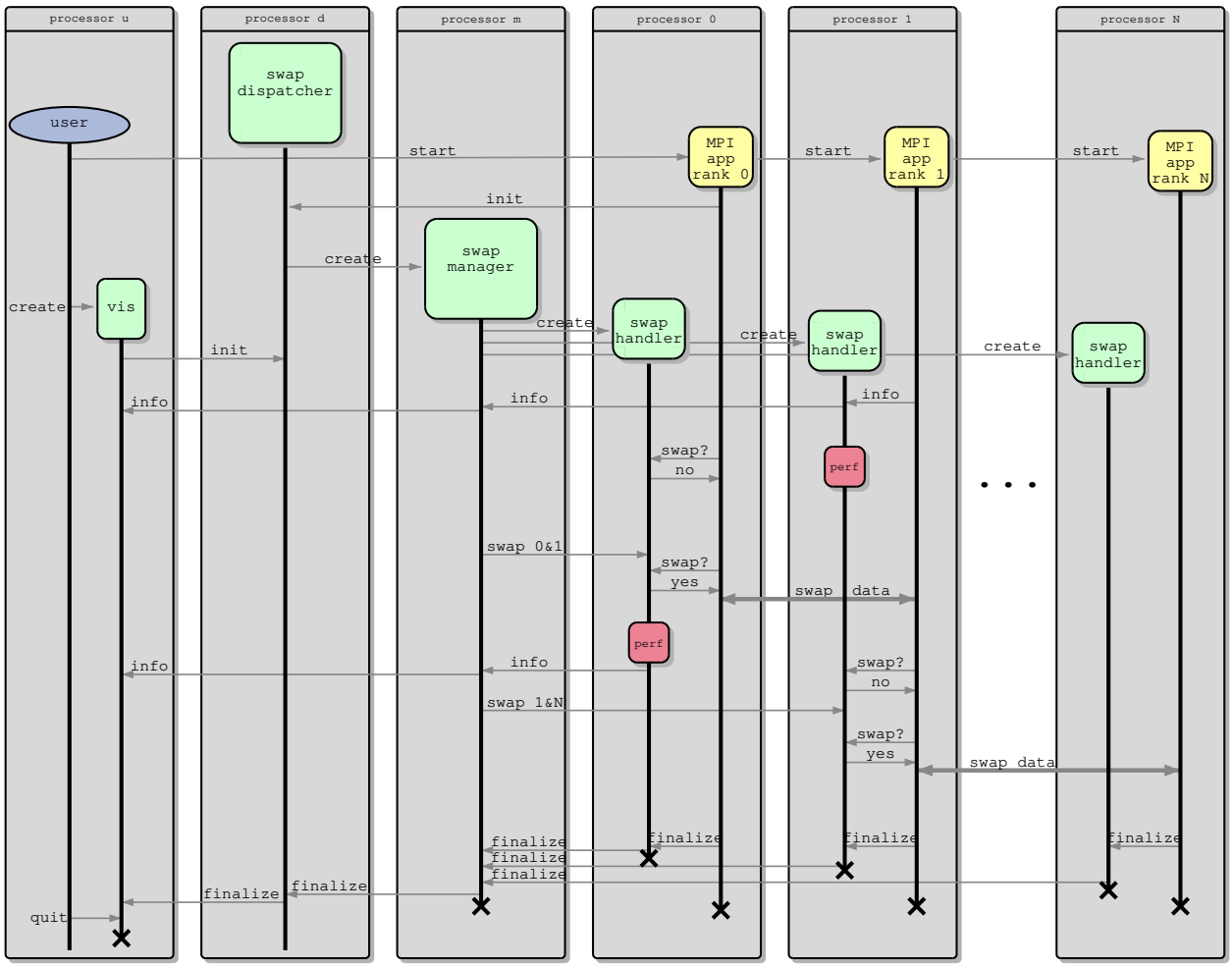

Fig. 3. Interaction diagram of a swappable MPI application.

this information to all MPI processes in the application. From this point onward, the swap dispatcher plays a minimal role; the swap manager becomes the focal point.

For each MPI process, the swap manager starts a swap handler on the same machine. Once the swap handlers are initialized, the application begins execution. While the application is executing, the swap handlers are gathering application and environment (machine) performance information and feeding it to the swap manager. Some of this information is passive, like the CPU load or the amount of computation, communication, and barrier wait time of the application. Other times the performance information is gathered via active probing, which uses significant computational resources for a short period of time but provides more accurate information. The swap manager analyzes all of this information and determines whether or not to initiate a process swap.

The *active root process*, the MPI process that is the root process in the group of active processes, contacts its swap handler periodically (at an interval of some number of iterations, during the call to `MPI_Swap()`). In this case, the active root starts out as the process on machine `0`. The first time this process asks if a swap is needed, the swap handler replies "no". The application continues

to execute, and information continues to be fed to the swap manager. Eventually, the swap manager decides that process `0` and `1` should swap, so it sends a message to the swap handler that cohabitates with the active root process. The next time the application asks if it should swap, the swap handler answers "yes". Processes `2` through `N` continue to execute the application while processes `0` and `1` exchange information and data. The process on machine `0` will become inactive, while the process on machine `1` becomes active.

When the swap is complete, process `1` is now the active root process, so the next swap message from the swap manager is sent to the process on machine `1`. This time, process `1` and process `N` swap. The execution continues in this fashion until it completes. As the MPI application shuts down, each MPI process sends finalization messages to its swap handler before quitting. The swap handler in turn registers a finalization message with the swap manager, then quits. Once all the swap handlers have unregistered with the swap manager, it sends a quit message to the swap dispatcher, and shuts down.

In this case, all during the application execution the user monitored the progress of the application. Shortly after the application began to execute, the user started the swap visualization tool.The visualization tool contacted the swap dispatcher, which told it where the swap manager lived. The visualization tool registered itself with the swap manager, and from that time forward the swap manager kept the visualization tool informed directly. After the application shut down, and the swap manager also shut down, the user closed the visualization tool.

This example illustrates the distributed nature of the swap services. However, all of these swap services could have been running on one machine, if the user had all of her MPI processes on that machine, had an interactive console on that machine, and the swap dispatcher and swap manager were launched on that machine.

4.2. Process Swapping Source Code Architecture

MPI process swapping is simple and minimally invasive to existing iterative MPI applications. In order to minimize the impact to user code, and yet still provide automated swapping functionality, MPI process swapping *hijacks* many of the MPI function calls. To illustrate how this is done, let us first examine a typical MPI application, as shown in Figure 4. This C-like pseudo-code contains the MPI calls from an actual MPI application that computes Van der Waals forces between particles in a two-dimensional grid [16]. In this typical scenario, a user's C source code includes the `mpi.h` header file, and makes several MPI function calls throughout the code. To build the application, the user compiles their source code and links to the MPI library, as shown in Figure 5. Note that MPICH [9] has built-in facilities for hijacking. Process swapping uses a different, but similar, mechanism so it is portable to any MPI implementation.

```
#include "mpi.h"

main()
{
  MPI_Init();
  MPI_Type_contiguous();
  MPI_Type_commit();
  MPI_Comm_size();
  MPI_Comm_rank();
  MPI_Bcast();  /* X 8 */


  MPI_Barrier();

  for (a lot of loops)
  {

    (MPI_Send() || MPI_Recv());
    MPI_Bcast();
    MPI_Allreduce();
  }

  MPI_Barrier();
  MPI_Finalize();
}
```

Fig. 4. Standard portable or vendor MPI C source.

In the swapping scenario, as few as three lines of code are changed from the previous scenario. First, the user's code includes the header file

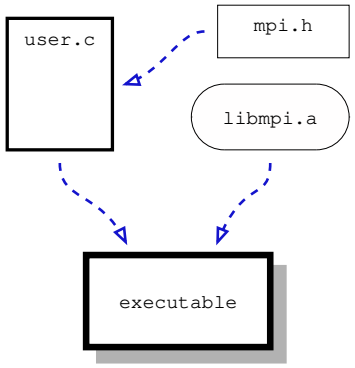

Fig. 5. Standard portable or vendor MPI usage.

```
#include "mpi_swap.h"  /* instead of mpi.h */

main()
{
  MPI_Init();
  MPI_Type_contiguous();
  MPI_Type_commit();
  MPI_Comm_size();
  MPI_Comm_rank();
  MPI_Bcast();  /* X 8 */

      swap_register(iteration variable); /* new */
  MPI_Barrier();

  for (a lot of loops)
  {
          MPI_Swap(); /* new */
    (MPI_Send() || MPI_Recv());
    MPI_Bcast();
    MPI_Allreduce();
  }

  MPI_Barrier();
  MPI_Finalize();
}
```

Fig. 6. Swappable MPI C source.

`mpi_swap.h` instead of `mpi.h`. Secondly, the user must register the iteration variable using the `swap_register()` function call. This is necessary in order for the swap code to know which iteration a particular MPI process is executing at any given time. Other variables may be registered, if it is important that their contents be transferred when swapping processors. Finally, the user must insert a call to `MPI_Swap()` inside the iteration loop to exercise the swapping test and actuation routines. Figure 6 highlights these changes.

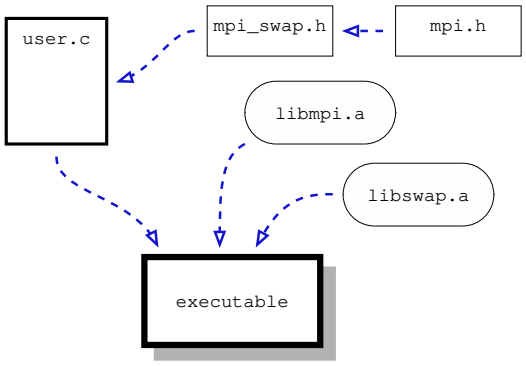

Fig. 7. Swapping resides on top of portable or vendor MPI implementations.

Figure 7 illustrates how a user would compile a swap-enabled application. The user includes the `mpi_swap.h` header file provided by the swap package, and links against both the standard MPI library, called `libmpi.a` here, and the swap library `libswap.a` that is provided by the swap package.

Swapping is implemented using private MPI communicators. An active communicator contains all the MPI processes that are actively participating in the application, and an inactive communicator contains all the inactive processes. To hide this complexity from the user, the swapping library hijacks MPI function calls, as shown in Figure 8.

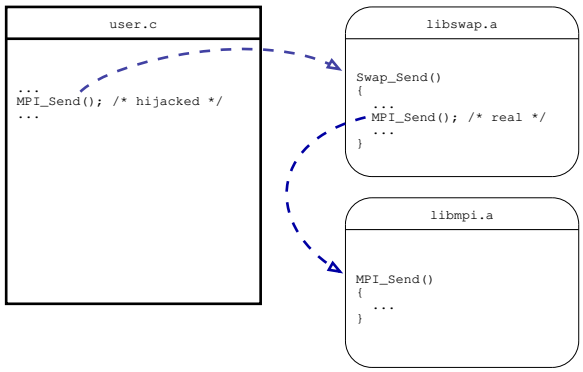

Fig. 8. Swapping *hijacks* standard MPI communications.

5. Experimental Results

A set of MPI process swapping experiments were performed on a production intranet at a Hewlett-Packard research and development facility. This NOW comprises several hundred high performance PA-8700 series RISC workstations in three buildings, connected to a central data server room via several subnets of 10-baseT and 100-baseT Ethernet. These workstations run HP-UX 11.11i exclusively. Most of the workstations are used as personal computers for Computer Aided Design (CAD), digital Application Specific Integrated Circuit (ASIC) design, embedded system design, and other research and product development activities. The experiments capture the natural variation found within this environment.

In one experiment, the *fish* MPI program from Fred Wong and Jim Demmel was used [16]. An example of the type generally found in the field of particle dynamics, this application computes Van der Waals forces between particles in a two-dimensional field. As the particles interact, they move about the field. Because the amount of computation depends on the location and proximity of particles to one another, this application exhibits a dynamic amount of work per processor even when the data partitioning is static and the processors are dedicated. From the original code, four source lines were added/changed in order to add the process swapping capability to this application.

Four processors were used in the experiment (two of them active). The application execution eclipsed thirty minutes. Figure 9 shows the relevant execution behavior from this run. There are four charts in this figure; each chart contains information about one processor. The vertical axis of these charts is a measure of processor performance. Process swapping supports several active and passive performance measures; the simplest of these, the inverse of the CPU load (as measured by the `uptime` facility), was used for these experiments. The horizontal axis of the charts is time. The broken line plots the instantaneous

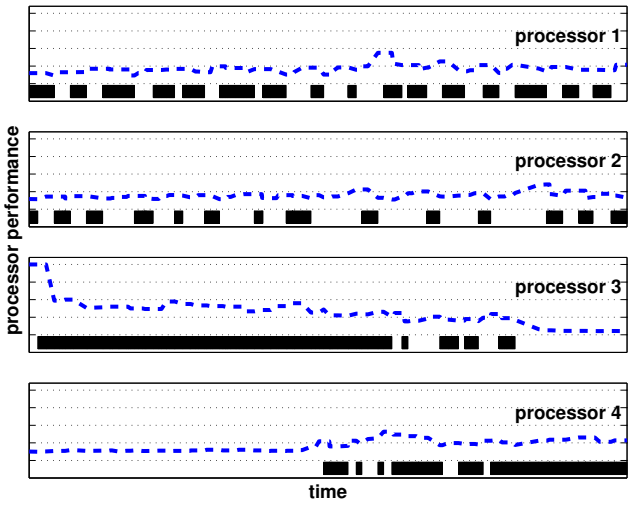

Fig. 9. Behavior of a swapping-enabled particle physics application. The y-axes are processor performance (higher is better); the x-axes are time. Broken lines show processor performance; the bars below show when processors were active.

computational performance as measured by the swap services, over the duration of the application (the higher the better). The solid black bars below the performance measurements indicate active/inactive status. At any given time, the presence of a black bar indicates the processor was active.

At the beginning of execution, processors 1 and 2 were active. Shortly after, however, processor 3 began a long duration of activity because its performance was very good. Thus the initial schedule, as computed by the off-line pre-execution scheduler, was quickly modified due to observed performance. During the first half of the execution, processors 1 and 2 shared an MPI process and processor 3 hosted the second active MPI process. In the later half of the execution, the performance of processor 3 continued to decline, and processor 4 became more desirable. Approximately forty swaps occurred during execution of the application.

Another experiment, illustrated in Figure 10, used a toy MPI application that was designed to

quickly and simply evaluate the implementation robustness of the process swapping services. Using eight active (out of sixteen total) MPI processes, this application run lasted thirty minutes. In addition to generally illustrating how swapping gravitates toward the machines with the highest performance, this run also shows the natural dynamism of a typical production environment.

For both of these experiments, a very simple swapping policy was used. Each time a new piece of information was delivered to the swap manager, it computed whether to swap or not based on only the most recent information. No hysteresis was applied. No knowledge of the volatility of a particular processor was taken into account. In fact, in this policy only environmental information (the computational performance of each processor) was used; no application information, e.g., barrier wait time, computation time, communication time, was used.

The swapping policy is a critical, but delicate, part of the process swapping system. Because optimal scheduling is typically NP-hard, many schedulers are laden with heuristics; the process swapping policy is no different. It is clear from the figures that swapping is occurring too often in these experiments. The *hot-potato* exchange between processors 1 and 2 in the fish run (Figure 9) was unnecessary given how similarly these two processors were performing.

One reason for this hot-potato activity could be the use of the (admittedly naïve) cpuload-based performance measure. This measure is fundamentally unable to separate load due to the swap application from load due to another source. For two otherwise evenly loaded processors, this will cause the kind of swap bouncing seen between processors 1 and 2. While running on processor 1, the observed load increases, causing a swap to processor 2. But when executing on processor 2, the load increases, so we swap back to the processor 1. And so on. Other performance measures employed by the swap handler are not susceptible to this kind of influence.

6. Future Work

In the near future additional work will be done to develop a set of general purpose swapping policies. Critical to this development is a thorough analysis of the performance of process swapping.

Especially in a dynamic environment such as the one used in the experiments described in this paper, real application runs are insufficient to prove anything about the efficacy of a swapping policy. Changes in the environment from one run to the next could have more effect on the results than a swapping policy change. In order to develop and evaluate swapping policies, a swapping simulation environment has been built. Using this environment, several swapping policies will be developed and cost/benefit models will be evaluated. Some of the resulting policies will then be introduced to the actual swap implementation, where they will be tested for general applicability in real world environments. These findings will be reported in an upcoming paper.

Another interesting future direction, that incidentally is not currently planned, would be to merge the swapping run-time services with a different swapping mechanism, for example the MPICH-V checkpointing facility described earlier.

Finally, the focus of this work to date has been on local area parallel computing. MPI process swapping could be applied to wide area parallel computing (grid computing) using MPICH-G2 [8]. In the wide-area environment, the cost of swapping can be much higher. However, the swapping implementation will function on the grid with only slight modification, and could have benefit in that arena as well.

7. Conclusion

The architecture of a system to improve performance of iterative MPI applications has been presented. By hijacking MPI calls, this user-level infrastructure can add dynamic performance steering to existing MPI applications with as few as

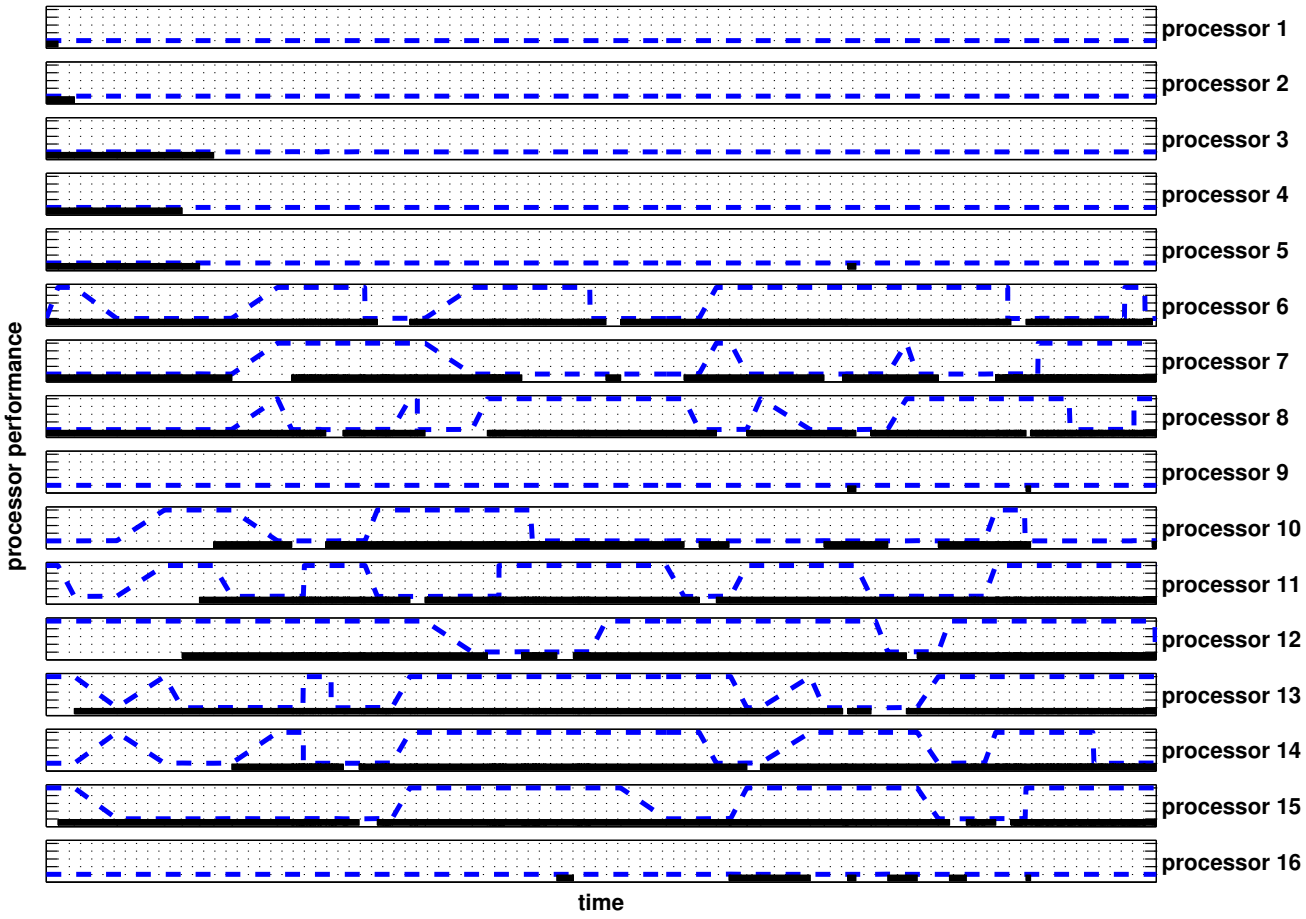

Fig. 10. Behavior of a swapping-enabled toy application. The y-axes are processor performance (higher is better); the x-axes are time. Broken lines show processor performance; the bars below show when processors were active.

three lines of source code change. During execution, the MPI application over-allocates MPI processes and uses only a subset of these, bypassing limitations in MPI 1.1 and MPI 2. A supporting set of run-time services provides information and support during application execution, and determines when and where to actively execute the application.

This system has been implemented, and initial testing has been done in a commercial production environment. The swapping system works, perhaps too well, as the swapping policy used in these runs tended to swap more than it should. A simulation environment will be utilized to further develop and tune swapping policies. The resulting policies will be tried by fire again in a real production environment.

ACKNOWLEDGMENTS

The Hewlett-Packard Company has provided extended access to their computing facilities, which were used for the production runs described in this paper.

The idea of developing a light-weight MPI process swapping system did not happen all at once, or in a vacuum. Discussions among members of the GrADS project, in particular Holly Dail, Ruth Aydt, and Celso Mendez, were critical to the for-

mulation of a need for a run-time performance system. MPI Process Swapping shares architectural ideas with GrADS and with the AutoPilot adaptive resource control system [13].

REFERENCES

[1] F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, pages ??–??, 1996.

[2] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of Supercomputing 2002*, pages ??–??, 2002.

[3] G. Cybenko. Load balancing for distributed memory processors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[4] R. Diekmann, B. Monien, and R. Preis. Load balancing strategies for distributed memory machines, 1997.

[5] E. Elsässer, B. Monien, and R. Preis. Diffusion schemes for load balancing on heterogeneous networks. *Theory of Computing Systems*, 35:305–320, 2002.

[6] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.

[7] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[8] I. Foster and N. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of SC'98*. ACM Press, 1998.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Supercomputing Applications*, 22(6):789–828, 1996.

[10] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a framework for preparing and executing adaptive grid programs. In *Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium 2002), Fort Lauderdale, FL, April 2002*, 2002.

[11] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988.

[12] M. P. I. F. MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[13] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. In *HPDC*, pages 172–179, 1998.

[14] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[15] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1):119–132, 1998.

[16] F. Wong and J. Demmel. UC Berkeley CS 267 course programming assignment 4 at `http://www.cs.berkeley.edu/~fredwong/cs267_Spr99/assignments/assignment4.html`.