UNIVERSITY of CALIFORNIA, SAN DIEGO

MPI Process Swapping: Performance Enhancement for Tightly-coupled Iterative Parallel Applications in Shared Computing Environments

A thesis submitted in partial satisfaction of the
requirements for the degree of Master of Science

in

Computer Science

by

Otto K. Sievert

Committee in charge:

Professor Henri Casanova, Chair
Professor Francine Berman, Co-chair
Professor Scott Baden

2003

The thesis of Otto K. Sievert is approved:

Scott B. Baden 6/6/03

Date

Francine Berman 6/6/03

Co-chair Date

[illegible] 6/3/03

Chair Date

University of California at San Diego

2003

For Aaren.

It is not best to swap horses crossing the river.

– Abraham Lincoln, *in a reply to the National Union League, 1864*

Contents

List of Tables

List of Figures

Acknowledgements

I would like to acknowledge a number of people who have provided support, given assistance, and shown understanding throughout the development of this work. All the Grail members at UCSD, but especially Holly, Shava, and Graziano, provided the best environment anyone could ask for. This work grew out of a desire for a mid-execution rescheduler for the GrADS project; the members of GrADS, especially Ruth, Celso and Rich, engaged me in many interesting discussions that influenced the swapping design. The Hewlett-Packard Company supported me throughout my graduate experience, and folks there like Larry and Craig who have themselves traveled this road were always generous sounding boards. Tiffany and Thomas tolerated my mental and physical absence as I was engaged in thesis activities, often at odd hours of the day. Finally, Fran exposed me to the expanse of thesis possibilities and to the many wonders and personalities in the parallel and distributed computing community, and Henri gave me tremendous attention and (through the use of some techniques not seen since the Inquisition) encouraged me to focus on one small part of this universe.

Thank you.

Portions of this work were previously published in the Proceedings of the High Performance Distributed Computing Conference and the International Journal of High Performance Computing Applications.

Abstract

MPI Process Swapping:
Performance Enhancement for Tightly-coupled Iterative Parallel Applications in Shared Computing Environments

by

Otto K. Sievert

Master of Science in Computer Science
University of California, San Diego, 2003

Professor Henri Casanova, Chair
Professor Francine Berman, Co-chair

Simultaneous performance and ease-of-use is difficult to obtain for many parallel applications. Despite the enormous amount of research and development work in the area of parallel computing, performance tuning remains a labor intensive and artful activity. A form of process migration called *MPI Process Swapping* is presented as one way to achieve higher application performance without sacrificing ease-of-use. Process swapping is a simple add-on to the MPI programming paradigm, and can improve performance in shared computing environments. MPI process swapping is easy to use, requiring as few as three lines of source code change to an existing iterative application, yet it provides performance benefits on par with techniques that require significant application modification.

Chapter 1

Introduction

While parallel computing has been actively pursued for several decades, it remains a daunting proposition for many end users. A number of programming models have been proposed [9, 39, 34] by which users can write applications with well-defined Application Programming Interfaces and use various parallel platforms. This thesis focuses on message passing, and in particular on the Message Passing Interface (MPI) standard [24]. MPI provides the necessary abstractions for writing parallel applications and harnessing multiple processors. However, the primary parallel computing challenges of application scalability and performance remain, especially in a shared computing environment. While these parallel challenges can be addressed via intensive performance tuning and careful application scheduling, typical end users often lack the time and expertise required. As a result, many end users sacrifice some performance in exchange for ease-of-use (the user is relatively happy as long as parallel application performance is better than serial execution, even if the overall parallel performance is quite inefficient). This is a general trend as parallel computing often enjoys ease-of-use or high performance, but rarely both at the same time. In this situation, a simple technique that provides a sub-optimal (but still beneficial) performance improvement can be more appealing in practice than a near optimal solution that requires substantial effort to implement.

MPI Process Swapping *is an easy-to-use application performance enhancement that provides benefit in dynamic environments. With careful attention to swapping policies, this thesis shows that process swapping can perform as well as other performance enhancing techniques such as Dynamic Load Balancing and Checkpoint/Restart.*

Process swapping is useful for applications that execute slowly because of resource contention. Using process swapping, application execution time can be reduced. This is done by continually moving the application computation from slow processors to fast processors.

Process swapping improves performance by dynamically choosing the best available resources throughout the execution of an application, using MPI process *over-allocation* and real-time performance measurement. The basic idea behind MPI process swapping is as follows. Say that a parallel iterative application needs N processes to run, due to memory and/or performance considerations. Process swapping *over-allocates* $N + M$ processes so that the application only runs on N processes, but has the opportunity to swap any of these processes with any of M spare processes. Process swapping imposes the restriction that data redistribution is not allowed: the application is "stuck" with the initial data distribution, which limits the ability to adapt to fluctuating resources. Although MPI process swapping will often be sub-optimal, it is a practical solution for practical situations and it can be integrated to existing applications easily.

Process swapping can be thought of as a partial application checkpoint and migration. Of the N active MPI processes, only those that show poor performance are checkpointed and have their work responsibilities transferred to another MPI process. Because the checkpointing used by MPI process swapping is only semi-automated (the user must define quiescent, checkpoint-able states in the code that guarantee no outstanding communication messages; the user also must register process state information that is transferred during a migration), this technique is most applicable to iterative applications.

Iterative applications lend themselves to MPI process swapping in two ways. First, the loop structure of an iterative code execution is a natural place for a command that checks for a process swap. Modifying only one place in the code gives many swap opportunities during execution. Second, iterative applications typically have natural barrier points where synchronization occurs. These barrier points often (but not always) have the required communication quiescence required for MPI process swapping. For these reasons, iterative applications are the focus of this work.

The target execution environment is heterogeneous time-shared platforms (e.g. networks of desktop workstations) 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). This type of platform has steadily gained in popularity in arenas such as enterprise computing.

In the target environment, it is assumed there are only a few parallel applications running on the platform simultaneously amidst a field of interactive (serial) jobs, creating performance “hot spots” that should be avoided. If the workload instead consists primarily of parallel applications with no interactive usage then the platform of choice should be a batch-scheduled cluster. Although our target usage scenario may appear limiting, workloads consisting of few parallel applications with interactive desktop users are not uncommon in academic and commercial environments. Many research labs at the University of California, San Diego have this characteristic, as do many commercial development facilities such as those used by the Hewlett-Packard Company, for example.

The remainder of this thesis is organized as follows:

- Chapter 2 provides background information and describes related approaches to improve application performance.
- Chapter 3 describes process swapping in detail.
- Chapter 4 describes experiments run to validate the implementation.
- Chapter 5 introduces three swapping policies.
- Chapter 6 compares these swapping policies to other performance enhancement techniques using simulation.
- Chapter 7 concludes the work.

Chapter 2

Related Work

This chapter relates process swapping to other performance enhancing techniques:

- work replication
- Dynamic Load Balancing (DLB)
- Checkpoint / Restart (CR)
- process migration
- pre-execution scheduling

In typical shared environments, the computing power of each individual workstation can vary for many reasons. For example, the resources themselves are usually heterogenous due to varying requirements or because purchases are staggered in time. Also, the resources are not uniformly used — between individual users and an occasional parallel job, each machine's load can vary. This variation can be dramatic and unpredictable, resulting in an unfriendly environment for parallel computing.

End users tolerate the inconsistency of such a computing environment. One day their applications may run quickly, but another day may bring slow progress. As long as there is some minimal amount of performance benefit to parallelizing an application, typical users today will be most concerned with how easy it is to run their parallel application. As a general group, end users are not likely to invest significant time and effort tuning their application schedules for performance beyond some notion of "good enough". Often they don't have the time, or the experience, or even the desire, required to perform detailed performance and scheduling analyses.

There are several well-known techniques to harness the power of a heterogeneous computing environment. Four of these techniques are described in this chapter: work replication, dynamic load balancing, checkpoint/restart, process migration, and pre-execution scheduling. It is claimed that, when the product of implementation effort and performance gain are compared, that MPI Process Swapping compares favorably with these techniques. Figure 2.1 illustrates this abstract claim. The remainder of this thesis provides information supporting this hypothesis.

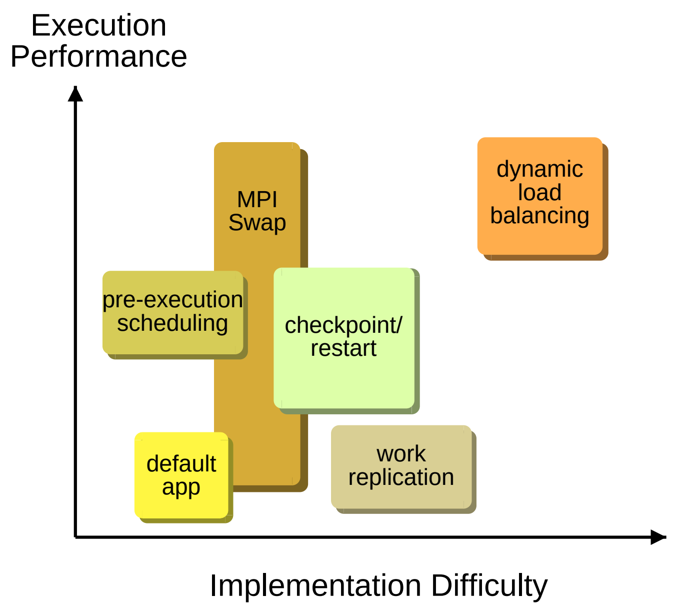


Figure 2.1: Claim: swapping brings potential performance benefits with relatively low effort.

2.1 Work Replication

Because process swapping intelligently decides which processors actively participate in program execution, the over-allocation technique used by 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. Both of these work replication methods wastefully use twice as much computational power as is required. In contrast, process swapping makes efficient use of the pool of available processors.

2.2 Dynamic Load Balancing

Dynamic Load Balancing (DLB) is one of the best known methods for achieving good parallel performance in unstable conditions. Dynamic load balancing repartitions application work during execution to balance load and minimize execution time. DLB techniques have been developed and used for scenarios in which the application's computational requirements change over time [10, 14, 18, 26] and scenarios in which the platform changes over time [49, 30, 50]. This work targets the latter scenario and DLB is thus an attractive approach — but it has limitations.

DLB often requires substantial effort to implement. Support for uneven, dynamic data partitioning adds complexity to an application, and complexity takes time to develop and effort to maintain. An application that supports arbitrary data decomposition is more likely to break down, and is harder to debug when it does fail. It should be noted that this difficulty is recognized; there have been efforts to provide middleware services that hide some of this complexity from the user and the programmer [22, 29, 5].

DLB requires an application that is amenable, in the limit, to arbitrary data partitioning. Many parallel algorithms demand fundamentally rigid data partitioning. For example, ScaLAPACK requires fixed block sizes (although these blocks can be somewhat creatively assigned to processors for performance reasons, as in [7]).

The performance of an application that supports dynamic load balancing is limited by the achievable performance on the processors that are used. A perfectly load-balanced execution can still run slowly if all the processors used operate at a fraction of their peak performance. It should be noted that a DLB implementation could further improve performance in this case through the use an over-allocation scheme such as the one used by process swapping.

2.3 Checkpoint / Restart

Another way for an application to adapt to changing conditions is Checkpoint/Restart (CR). Checkpoint/restart halts an entire application during execution, saves important application state information, and restarts the application on possibly new hosts with a possibly new data partitioning. CR is traditionally used for fault-tolerance, but it can be used for performance considerations by choosing appropriate restart hosts [2, 45].

CR does not limit the application to the processors on which execution is started, so it does not have to remain running on a set of processors that have become loaded. It also does not require a sophisticated data partitioning algorithm, and can thus be used with a wider variety of applications/algorithms. Unfortunately, generalized heterogenous checkpoint/restart of parallel applications is a difficult task; it remains the subject of several active research projects [40, 3, 20, 6] and is not widely supported today.

Checkpointing may incur significant overheads depending on the application and compute platform. For large applications using substantial amounts of memory, running on a set of machines where only one or two nodes are loaded and slow, the overhead required to checkpoint the entire application, possibly to a central remote checkpoint drydock, is substantial. After the application has been checkpointed, a new application schedule must be calculated. This can take some time, especially if the scheduler needs to “cool down” for a period of time before the resource performance predictions are accurate enough to compute the new schedule (because of hysteresis effects, cpuload-based performance estimates often require several minutes to fully dissipate the effect of a running application). Once this new schedule is completed, the entire application must bootstrap itself and read the appropriate data and process state from the checkpoint file. Only then can computation proceed.

Application-level checkpointing can be implemented with limited effort for iterative applications. The Cactus worm [2] uses standard Cactus checkpointing facilities to checkpoint a Cactus application to a central remote location, allowing the application schedule to be recomputed.

The CoCheck project provides a checkpointing tool for PVM and MPI applications, allowing checkpoint/restart [40]. CoCheck is implemented on top of MPI, and provides parallel checkpointing and message forwarding.

The SRS library recently developed to add checkpoint capability to ScaLAPACK provides similar user-level checkpointing (without the outstanding message bookkeeping) [44].

2.4 Process Migration

Process swapping is also related to a number of efforts to add migration support to the MPI runtime system. Migration facilities such as those provided by fault-tolerance extensions to MPI provide better-integrated support and more generally improve the capabilities of the MPI system. For example, MPI/FT provides self-checking parallel threads, offering a true migration infrastructure [3]. Similarly, FT-MPI adds fault-tolerance to MPI [20]. MPICH-V [6] is very similar to FT-MPI, rising out of the volatility exhibited by global computing platforms. MPICH-V provides uncoordinated checkpointing as part of the MPI core implementation, and tracks undelivered messages through a distributed message logging facility. Designed primarily for very large systems whose mean time between failure is low, these techniques could conceivably be used to migrate for performance reasons.

These migration mechanisms could be combined with the process swapping services and policies developed in this thesis, improving the robustness and generality over the current process swapping solution. In particular, a checkpointing facility might allow a better process swapping implementation by (i) removing the restriction of working only with iterative applications; (ii) further reducing the already minimal source code invasiveness; and (iii) reducing or removing the need to over-allocate MPI processes at the beginning of execution.

Combining MPI process swapping policies with the cycle-stealing facilities of high throughput desktop computing systems like Condor [33], XtremWeb [21] or other commercial systems [19, 27] would yield a powerful system. These systems currently evict application processes when a resource is reclaimed by its owner. By combining swapping policies with this eviction mechanism, a process might also be evicted and migrated for application performance reasons. Such a combined system would not only provide high system 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 [6] could be used.

2.5 Pre-execution Scheduling

MPI process swapping can be categorized as a mid-execution scheduler, and can be compared to pre-execution application schedulers such as those found in the AppLeS [4] and GrADS [28] projects. These projects are also concerned with achieving high performance within dynamic parallel execution environments. Additionally, they strive for ease-of-use, an important attribute for disciplinary scientists. The performance measurement and prediction techniques used in process swapping have much in common with these projects; all use application and environmental measurements (e.g. via the NWS [46], Autopilot [35], or MDS [23]) to improve application performance.

AppLeS schedulers exist to promote application performance (as opposed to system throughput or system performance). Many application-specific AppLeS schedulers have been built, for example [38, 41, 12]. By and large, these schedulers can be categorized as pre-execution schedulers, matching an application to appropriate computing resources before the application is run. Recently, application-class AppLeS schedulers have been pursued, most notably AMWAT and APST, which focus on master/worker and parameter sweep application classes respectively [36, 8].

In related work, the GrADS project seeks grid ease-of-use through performance-aware grid infrastructure from compilation through execution. This effort includes an application-generic pre-execution scheduler that promotes application performance, but is modular [11]. Using a rich application requirements description and resource capabilities, this scheduler matches an application and execution resources for a variety of application classes. Also part of this effort is a mid-execution rescheduler, which evaluates the application schedule and alters it during execution to improve performance [13]. The work in this thesis has been integrated into the GrADS system as the first implementation of this rescheduler.

Pre-execution scheduling is advantageous because it requires little or no application modifications. The pre-execution scheduler determines resources and application configuration information without access to application source code.

While pre-execution scheduling is broadly similar to mid-execution scheduling (for example, when applied in the middle of an application run both schedulers will discard slow processors and acquire fast processors), a mid-execution scheduler has more constraints that make simple application of a pre-execution scheduler difficult. These constraints are categorized below:

- **affinity –** After executing for some time on a particular set of processors, an application develops an affinity for these resources. The use of different resources therefore has an additional cost that is not present before the start of execution.

- **the Heisenberg Principle –** Using traditional pre-execution performance estimation techniques, it is difficult to separate the performance effect of the application itself, as it is currently running, from other performance drains. Blindly using these traditional performance estimates will result in frivolous scheduling decisions, as the currently used resources will appear slow and overloaded, even when the load is caused by the application itself.

- **relative performance is not sufficient –** Before execution, the pre-execution scheduler chooses the best resources for the application, often relying on relative performance rankings. However, during execution there is a substantial cost for changes to the schedule. Any mid-execution change must be evaluated using absolute, not relative, performance measures or one cannot reason about the benefits of the new schedule relative to the cost of implementing the schedule.

Chapter 3

MPI Process Swapping

This chapter presents MPI process swapping in detail, including:

- the swap library, and its impact on application source code;
- the swap run-time services, and how they interact with a swap-enabled application;
- the advantages and limitations of the swapping design and implementation, including performance considerations.

3.1 The Swap Library

In order to minimize the impact to user code, and yet still provide automated swapping functionality, MPI process swapping *overloads* many of the MPI function calls through a combination of `#define` macros and function calls. Over-allocation is implemented with two 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 overloads MPI function calls, as shown in Figure 3.1.

The application developer is still required to make a handful of changes to an existing application. This is best illustrated through an example. Figure 3.2(a) contains C-like pseudo code for a typical MPI application. This example shows the communication from an actual MPI application that computes Van der Waals forces between

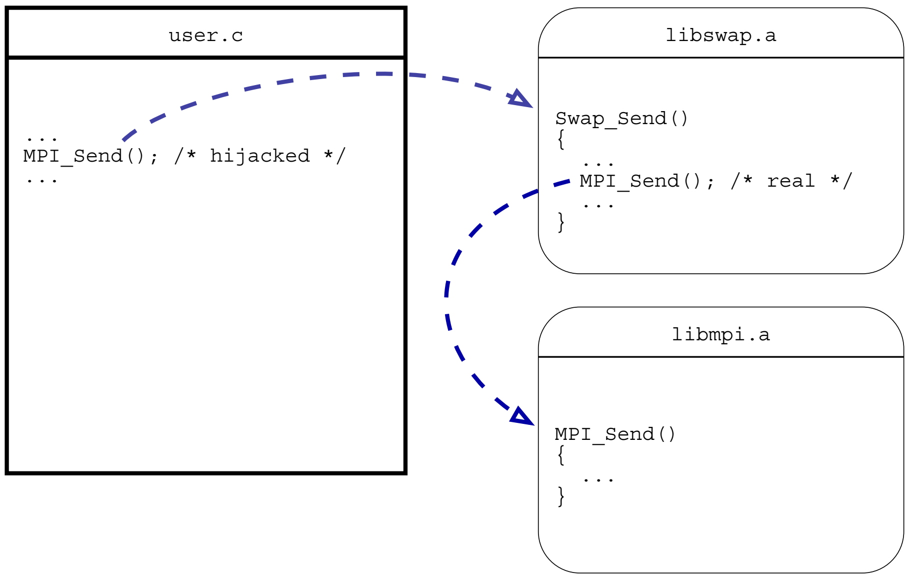


Figure 3.1: Swapping *overloads* standard MPI communications.

particles in a two-dimensional grid [48]. In this example, the original 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 3.3(a).

To swap-enable this application, the following three lines of code are changed (see Figure 3.2).

1. The user's code includes the header file `mpi_swap.h` instead of `mpi.h`.

2. 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.

   The `swap_register()` function is used to register statically allocated memory that is important to be swapped. All statically allocated memory that must be transferred during a swap must be registered with the `swap_register()` function call; for example, the iteration variable above is registered this way. The `swap_register()` call is considered collective and must be issued across all processes.

3. The user must insert a call to `MPI_Swap()` inside the iteration loop to exercise the swapping test and actuation routines. The `MPI_Swap()` function call acts like a barrier to active processes. It must be placed inside the application's iteration loop. The current implementation requires that no communication messages be outstanding when `MPI_Swap()` is called. In theory, outstanding messages could be allowed by forwarding them to the new active process. This enhancement has been designed, but not implemented to date.

Figure 3.3(b) illustrates how to 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.

```
#include "mpi.h"

main()
{
  MPI_Init();


  for (a lot of loops)
  {

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

  MPI_Finalize();
}
```

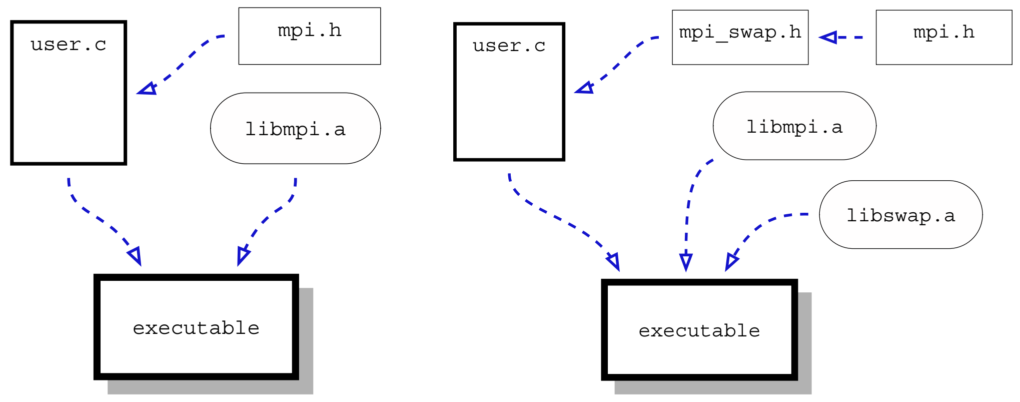
(a) Standard MPI C source.

```
#include "mpi_swap.h"

main()
{
  MPI_Init();
  swap_register(iteration variable);

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

  MPI_Finalize();
}
```

(b) Swap-enabled MPI C source.

Figure 3.2: The minimum source changes to swap-enable an application.

In a similar way to how the standard MPI calls are overloaded, the standard C library calls to dynamically allocate memory are overloaded. In this way, the swap library



(a) Standard MPI.

(b) Swap-enabled MPI.

Figure 3.3: The build changes to swap-enable an application.

tracks memory that needs to be communicated during a swap. In the rare instance that some local memory is dynamically allocated (and does not need to be communicated during a swap), passthrough functions are provided that allocate memory but do not register the memory for swap communication. This default overloading of dynamic memory allocation can cause lower application performance. However, swapping is much less likely to fail for novice users if all dynamic memory is trapped and registered. Furthermore, the advanced user who is concerned about every little bit of performance will take the time to sort out these local memory pools and prevent them from being registered.

3.2 Run Time Services

The swap run time services, depicted in Figure 3.4, are independent processes that together manage the execution of a parallel application. The swap services monitor the application and the resources on which it runs, and are responsible for determining when and where to swap.

Below, each of these services is described in more detail.

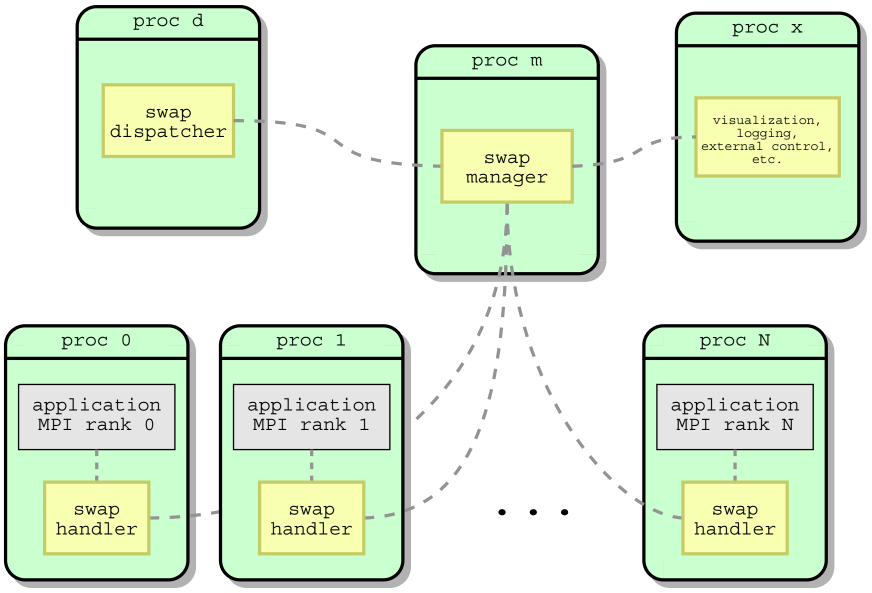


Figure 3.4: Swap run-time architecture.

swap dispatcher – The swap dispatcher service is an always-on service, listening at an advertised host/port. As shown in Figure 3.5, the swap dispatcher is contacted during application bootstrap, in the `MPI_Init()` call. At this time the dispatcher launches a swap manager on the processor with MPI rank 0. Until the application terminates, and the swap manager unregisters itself with the swap dispatcher, the only additional action provided by the dispatcher is to forward communication inquiries to the appropriate swap manager. For example, the swap dispatcher can be contacted for information about a particular application, and this request will be forwarded on to the swap manager responsible for the application.

swap manager – When it starts, the swap manager reports to the swap dispatcher the host and port number on which it receives messages. This information is passed back to the application. The application communicates this information to all the MPI processes in the parallel app. Each of these processes contacts the swap manager and requests a swap handler. The swap manager remotely starts a swap handler for each application process.

During application execution, the swap manager plays an active role, tracking the performance of active and inactive processors, tracking application progress, evaluating this information, and making process swapping decisions.

swap handler – A swap handler is started for each MPI process, during each process' call to `MPI_Init()`. Each MPI process requests a swap handler from the swap manager. The swap manager remotely starts a swap handler on the same processor as the MPI process. This handler returns to the swap manager the host and port on which it communicates. The swap manager gives this information to the MPI process, which then only talks with the swap handler for the remainder of the application. Upon application termination, the swap handler unregisters itself with the swap manager, then exits. The swap handler exists as a separate process (or thread) for several reasons:

- For efficiency all swapping communication between the application and the swap services are made on the local host. During the bulk of the application run, the overhead of swapping is minimized because all communication is local to the processor.
- The swap manager needs to have asynchronous dialog with the swap handler, and this could not happen if the swap handler were written inline in the MPI process.
- The swap handler performs some active resource measurements from time to time, and this is very difficult to do on inactive MPI processes where the handler code is inline with the application.

Swap utilities – The swap utilities are 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 admin provides a simple interface to manually configure or control the swapping system.

When a swap is determined by the swap manager, it communicates this to the swap handler associated with the MPI process with local rank zero (the "local root"). At the next application iteration, during the call to `MPI_Swap()`, the swap handler indicates to the local root process that a swap is necessary. The local root communicates this information to all the MPI processes, and the appropriate processes initiate a swap. The other processes continue on with the application, doing as much work as possible in parallel with the swap.

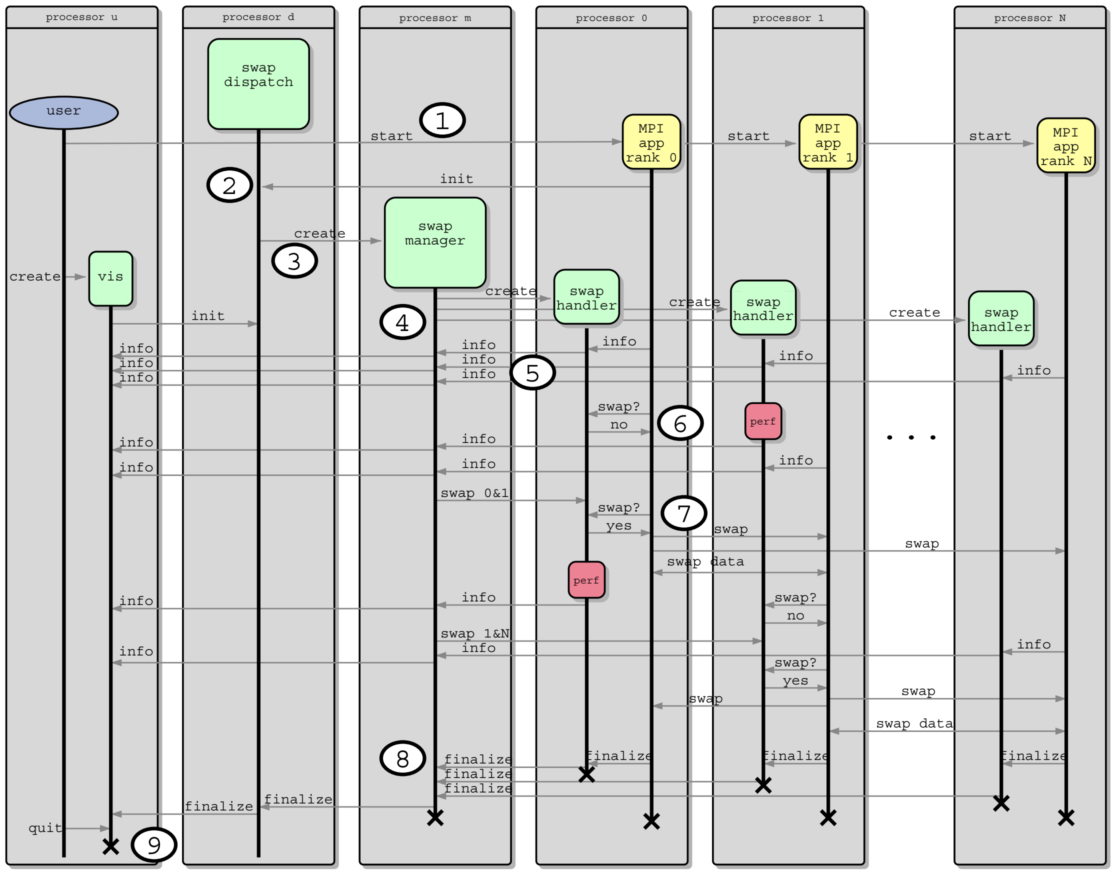


Figure 3.5: Interaction diagram of a swappable MPI application.

The swap services interact with the MPI application and with each other in a straightforward asynchronous manner, as illustrated in Figure 3.5. Walking through an example application execution will further describe these interactions. For ease of reference, the numbers below are matched in the figure.

1. 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.

2. 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.

3. 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 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.

4. 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.

5. 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.

6. 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.

7. 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.

8. 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.

9. 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.

3.3 Advantages of the Design

The advantages of this design are briefly discussed below.

Minimal impact on application source code – MPI process swapping works by *overloading* many of the MPI function calls. The goal is to achieve as much transparency as possible for the application programmer and end-user. There are several modifications to the application source code that must be done by the user. Conceivably these steps could be automated with a compiler but they are straightforward for a user to implement. As few as three lines of source code need to be changed in order to swap-enable an existing iterative MPI application.

Support for MPI-1 applications – Process swapping is tightly integrated with MPI, but it does not require advanced MPI-2 features. MPI-1 does not support adding processors to communicators, so process swapping relies on over-allocation of processes at the beginning of execution to get its pool of processors. Swapping chooses the best subset of available processors to actively participate in the application execution; the rest remain inactive until needed. These inactive processes utilize very little computational power; aside from periodic active performance measurement, they block on I/O calls and wait to become active.

MPI-2 has support for adding and removing processors to an application during execution. However, MPI-2 is not as widely supported as MPI-1. Furthermore, the functionality to add/remove processes is not transparent. The programmer must manage new communicators, requiring significant source code modification for existing MPI-1 applications. By contrast, MPI-1 with process swapping requires minimal source code changes. So while MPI-2 dynamic process management functionality such as that supported by the latest grid-enabled MPI implementation, MPICH-G2 [43], minimizes the need for over-allocation, it requires significant modification to existing applications.

Profiling support – By using a custom interface to overload MPI functions, the end user may still use any MPI profiling, logging, or debugging instrumentation they are familiar with. MPI does not have language support for multiple levels of interception, which means that (without special arrangement) only one profiling instrumentation can be done. Had the swap library used the standard MPI profiling interface for its own use, this would have precluded the use of another profiling or debugging tool.

Scalable run-time architecture – By organizing the run-time system as services, and minimizing the communication with a central service, the run-time system can scale to large numbers of simultaneous swap-enabled applications running at the same time.

3.4 Limitations

There are a number of limitations to the current implementation. Some of these limitations are fundamental design limitations, but most are simply features that (for expediency) were not implemented. A laundry list of limitations is given below.

Iterative applications – The swap system was designed to target the class of iterative applications.

Language – The swap library only has C bindings. C++ and Fortran bindings have not been implemented.

Message forwarding – The swap library does not perform message forwarding; in particular, this means that the call to `MPI_Swap()` must be at a natural application barrier; there can be no outstanding asynchronous communications when this function is called.

User-defined communicators – The swap library does not currently do bookkeeping necessary to properly handle user-defined communicators.

Overhead of MPI overloading – There is a slight performance overhead in each MPI communication call. The cost of an additional function call is incurred for each call. These overheads, while small, can accumulate in an application with a lot of MPI calls. The function call overloading could have been implemented with macros, but this limits the readability and maintainability of the swap library itself.

Inactive process blocking – Inactive MPI processes block on an MPI communication receive. On most MPI implementations, this means that these inactive processes consume virtually no system resources. This is what allows an application to dramatically over-allocate processors without incurring prohibitive penalties or causing problems for other users. However, it must be noted that some MPI implementations use spinning in their implementation of `MPI_Recv()`, for example TMPI[42]. On these machines, with the current implementation, the inactive processes would needlessly consume a significant amount of resources.

It is straightforward to implement the inactive processes' pause in a fashion that does not consume resources, for example through the use of the `select()` facility. This was not done in this implementation for expediency and because the systems on which swapping was evaluated did not spin wait, and so they did not consume significant resources while processes were inactive.

Scalability – The swap services, written in Python, do not scale well above roughly one hundred processes. Above this level, communication traffic causes dropped messages and reset socket connections. This could be addressed through a focused redesign. Similarly, the mpich system could not launch more than two hundred processes before running out of process space.

Chapter 4

Experimental Results

Swapping has been implemented and verified in a commercial production environment. In this chapter, this validation is presented. The overheads of swapping are measured, and the effectiveness of the system is shown anecdotally (a more rigorous discussion of swap efficacy is presented in Chapter 6.

4.1 Environment

The Hewlett-Packard Company maintains a large network of workstations in San Diego, California. These machines are a heterogenous collection of HP PA-RISC processors running HP-UX 11.11i, connected through 100baseT Ethernet using multiple sub-nets, routers, and switches. The majority of the HP machines are used as individual workstations for commercial research and development activities, such as Computer Aided Drafting, Very Large Scale Integrated digital circuit development, and software/firmware development. A handful of these machines are not used interactively; instead, these machines are used as a cluster. Three hundred of the HP machines were included in MPI process swapping machine pool, though due to scalability concerns fewer than 100 participated in experiments at any given time.

Figure 4.1 shows an example of the CPU load on this system. The load in this figure is typical of most of the interactively-used processors in the system. The batch processors tend to have much fewer jobs that run much longer.

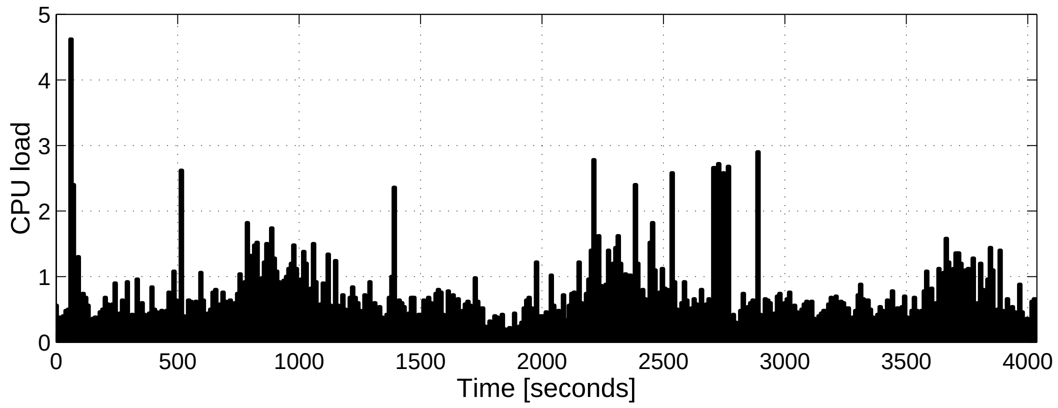


Figure 4.1: Example of CPU load in the experimental environment.

4.2 Applications

A number of applications have been swap-enabled. They are described here.

fish – Fish is a simple particle physics parallel application written by Fred Wong and Jim Demmel [48]. Typical of the codes 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. This iterative code uses simple MPI calls using only the `MPI_COMM_WORLD` communicator. It has a per-iteration barrier point, which is a natural place to insert a call to `MPI_Swap()`. From the original code, four source lines were added/changed in order to add process swapping capability to this application.

iterative stencil applications – This unified application, written by Holly Dail, performs the computation and communication for (a) a Jacobi relaxation solver for a system of equations, and (b) for the game of life algorithm. Both sub-applications are iterative and have natural barrier points suitable for `MPI_Swap()` insertion. These applications use straightforward synchronous MPI communication calls, and only communicate via `MPI_COMM_WORLD`. From the original code, only three lines of code were changed to enable swapping.

synthetic – This application was designed specifically to emulate applications like fish, jacobi, and the game of life. It has the same basic application elements of configuration, iteration, finalization. Unlike these real applications, the synthetic application does not implement a real algorithm, or communicate real data. Instead, this code is able to be tuned to any computation/communication demands desired. The advantage is that the synthetic application allows quick and efficient exploration of entire classes of applications without actually writing them. The synthetic application does perform actual floating point calculations (additions and multiplies), and it does communicate buffers full of random data. Four lines of source code were altered to make this application swap-enabled.

ring – This well-known introductory MPI application sends a message from process to process in a ring communication. The message finally returns to the initial sending process. This simple application reports the total message travel time, and as such is useful for measuring various swap overheads.

mpi_sleep – This utility application was designed to allow exploration of swapping capabilities without degrading the experience of other users. Instead of performing actual computation and communication, the payload of this iterative application is simply a sleep command. When configured with a particular sleep time and number of sleep iterations, this application swaps in between sleeping. This application was swap-enabled with modifications to three lines of source.

4.3 Swap Policy

The policy used in the experiments took every opportunity to acquire and use faster processors, even when those processors were only marginally faster than the current processor set. This greedy policy periodically evaluated the performance of all the machines in the application's pool (a subset of the entire HP cluster), and swapped processes if the slowest active machine had lower performance than the fastest inactive machine. Only one process swap was allowed per application iteration.

4.4 Results

4.4.1 Cost of over-allocation

Over-allocation adds overhead to the execution of a parallel application. Because more processes are involved, some collective MPI operations are more time consuming. Figures 4.2 – 4.4 show the elapsed time for the `MPI_Init()`, `MPI_Barrier()`, and `MPI_Finalize()` calls, as a function of the number of MPI processes. While the data shown are from actual performance measurements taken on the GRAIL[1] cluster at UCSD, these numbers are indicative of the performance on the HP system as well.

The data in these figures represent the time required to perform various MPI tasks as a function of the number of processes. The actual data are plotted as individual points on the graphs, and are accompanied by best-fit first-order (linear) equations describing the data. These best-fit models were created using a least-squares regression on the data. Because of tree or other hierarchical implementations of common MPI communications, one might not expect the models to be linear with the number of MPI processes. In fact, one might expect the overheads to be linear with the log of the number of MPI processes. While higher-order polynomials and logarithmic models have slightly better fit to the data, in the range of concern the linear models match sufficiently well.

Looking first at Figure 4.2, the process startup time is noticeably impacted by over-allocation. Approximately $3/4$ second is spent initializing each MPI process. For an application that naturally desires eight processes for optimal parallel speedup, the total cost for a 100% over-allocation is not high (about six seconds total, or a 100% increase above a nominal three second startup without over-allocation). However, for a large parallel application that naturally desires one hundred processes, doubling the allocation costs more than a minute. Doubling the allocation of a massively parallel application

[1] The Grid Research And Innovation Laboratory (GRAIL) is a parallel and distributed computing lab at the San Diego Supercomputing Center under the direction of Henri Casanova. The GRAIL cluster comprises thirteen Intel and AMD desktop machines running the Linux operating system.

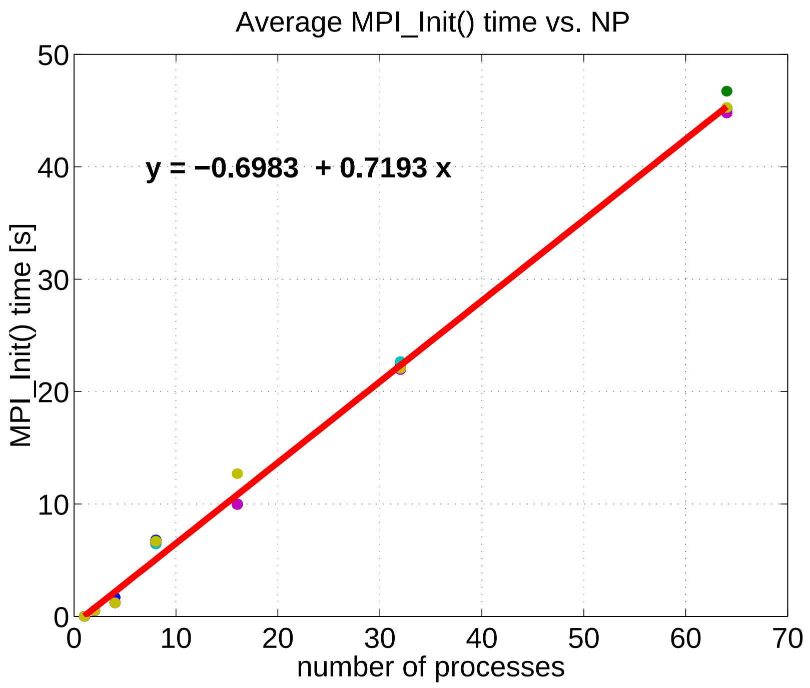


Figure 4.2: Time costs of over-allocation for `MPI_Init()`.

that actively uses one thousand processes adds twelve minutes to the process startup time according to this model. For a long-running application in a dynamic environment, it is possible that this startup time, even for a massively parallel application, can be recouped through swapping. However, this cost is guaranteed, while the performance benefits of swapping are not.

Note that the `MPI_Init()` model derived from the data predicts near zero cost with one process. This reflects the measurement technique used, which does not account for the cost to stage and start the first MPI process. Measuring the startup time for the first process is difficult due to clock skew issues. Since the startup time for the first process is immaterial to the affect of over-allocation (a “first” process must be started whether or not the execution is over-allocated), the startup time for the first process is not discussed.

Figure 4.3 shows that barrier wait times are not substantially increased with more MPI processes. The slope of this best-fit model indicates that each additional MPI

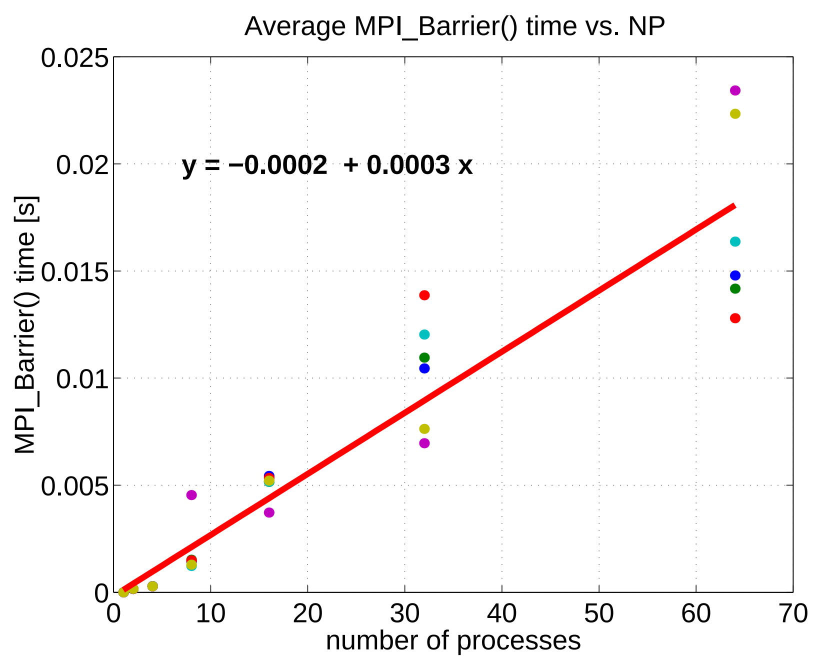


Figure 4.3: Time costs of over-allocation for `MPI_Barrier()`.

process adds 300 microseconds to the barrier time. This is, even for massively parallel applications, negligible. It can be observed that the variation in barrier wait time increases with the number of processes. With more individual processes and communications, there is increased likelihood that a process will be paused (by the local machine task scheduler) or that a communication will be delayed (possibly by the other barrier communications), resulting in larger variation.

Figure 4.4 shows that `MPI_Finalize()` costs are not substantially increased with more MPI processes. The slope of this best-fit model indicates that each additional MPI process adds less than two milliseconds to the total time of this collective MPI operation. As with barrier wait time, the effect is negligible.

Because process swapping utilizes private communicators to associate processes that are actively performing application work, point to point communications and collective communications other than `MPI_Init()` and `MPI_Finalize()` incur no performance drop due over-allocation.

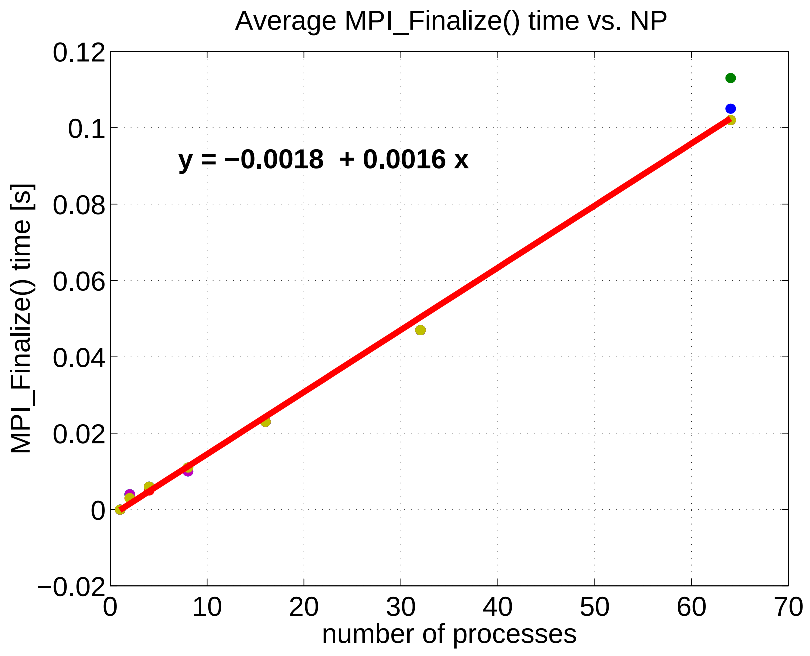


Figure 4.4: Time costs of over-allocation for `MPI_Finalize()`.

4.4.2 Gross swapping behavior

In one experiment, the *fish* program was executed using four processors (two of them active per the greedy swapping policy). The application execution eclipsed thirty minutes (wall clock time). Figure 4.5 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 (higher is better). 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 computational performance available to an application over the duration of the application, i.e., if the application were to actively use a processor, this is the performance it would achieve. The solid black bars below the performance measurements indicate active/inactive status. At any given instant in time, the presence of a black bar indicates the processor was active.

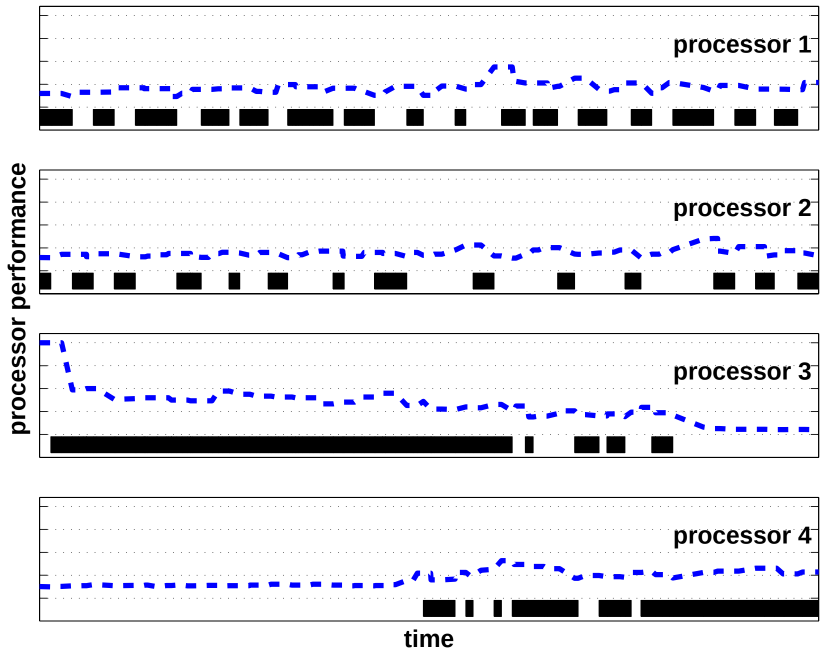


Figure 4.5: Behavior of a 4 process (2 active) swapping-enabled particle physics application. The y-axes are processor performance (higher is better); the x-axes are time. Broken lines show processor performance; the bars below show when processors were active.

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 modified within one application iteration 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.

It is clear from the figure that swapping is occurring too often in this experiments. The *hot-potato* exchange between processors 1 and 2 was unnecessary given how similarly these two processors were performing. This is one of the problems with the greedy policy.

This needless hot-potato swapping activity could be exacerbated by the use of the (admittedly naïve) cpuload-based performance measure. This measure is fundamentally unable to separate load caused by the swap-enabled application from load caused by another job. 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.

Another experiment, illustrated in Figure 4.6, used the synthetic 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 also lasted thirty minutes. In addition to generally illustrating how swapping moves applications toward the machines with the highest performance, this run also shows the natural variability of a typical production environment.

This execution also used the greedy swap policy. However, because the loaded resources were so heavily loaded, needless hot-potato swapping did not occur. In fact, at any instant in time you can see the swap system acting to favorably improve the performance of the application. The only time the application remained on poorly performing processors was when there were no better processors to choose. During these times the application made the progress it could while continuing to search out better resources.

Figure 4.7 show a portion of a 64 processor run. In this experiment, 42 processes were active. Even as the number of processes grows, the swapping system still generally maintains enough control to avoid the slowest processors. The broken line shows the processor performance throughout the run, and the solid horizontal line indicates when a processor was active. The swapping system as a whole can scale up to nearly 100 processors before losing stability. The MPICH implementation runs out of process space with between 40 and 200 processes, on the systems used. The swapping system suffers communication failures due to heavy message traffic around 80 to 100 processors. The result is a system that can scale to the tens of processors range easily, but can not support hundreds of processors or more.

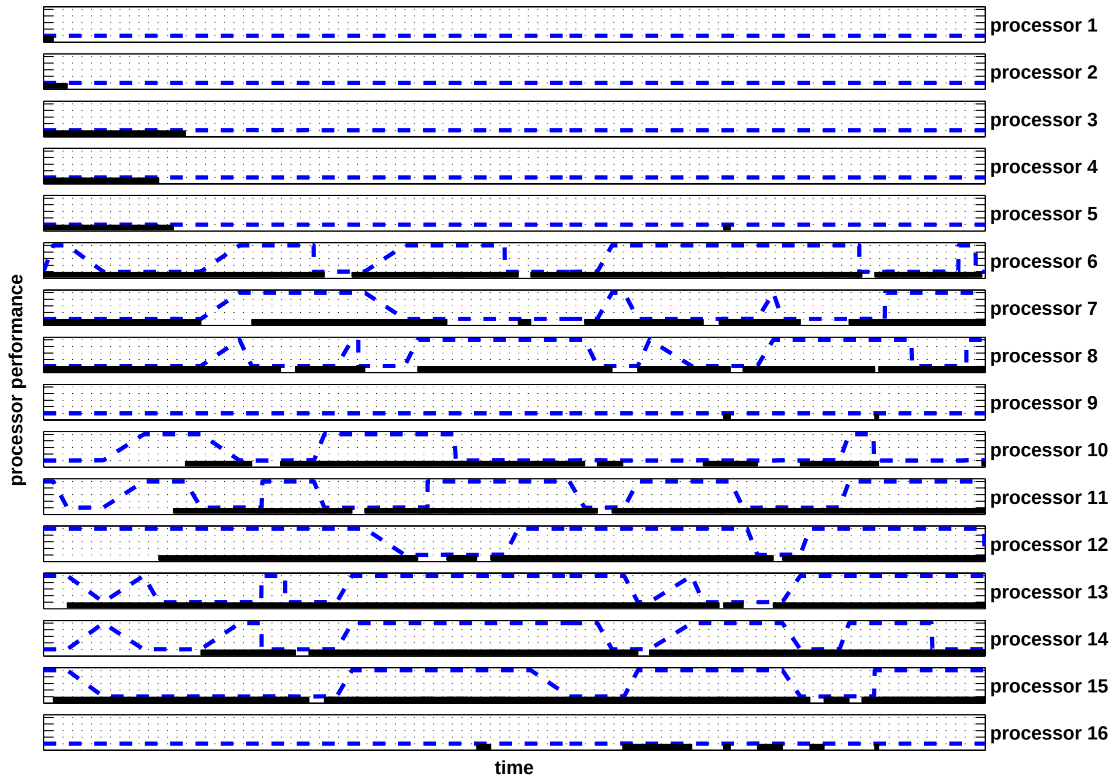


Figure 4.6: Behavior of a 16 process (8 active) swapping-enabled synthetic application. The y-axes are processor performance (higher is better); the x-axes are time. Broken lines show processor performance; the bars below show when processors were active.

4.5 Conclusion

The process startup overhead of over-allocation is noticeable at approximately $3/4$ second per process. This overhead is incurred whether or not a swap happens, and so must be a factor in determining whether or not to swap-enable an application. The effect of over-allocation on other MPI communications is negligible.

Swapping has been implemented and works on real systems for real applications of moderate size. However, the greedy swap policy combined with the CPU load performance metric does not function well under conditions where the performance of processors is similar, resulting in excessive swapping. Clearly, more careful study of the policies that govern swapping is needed.

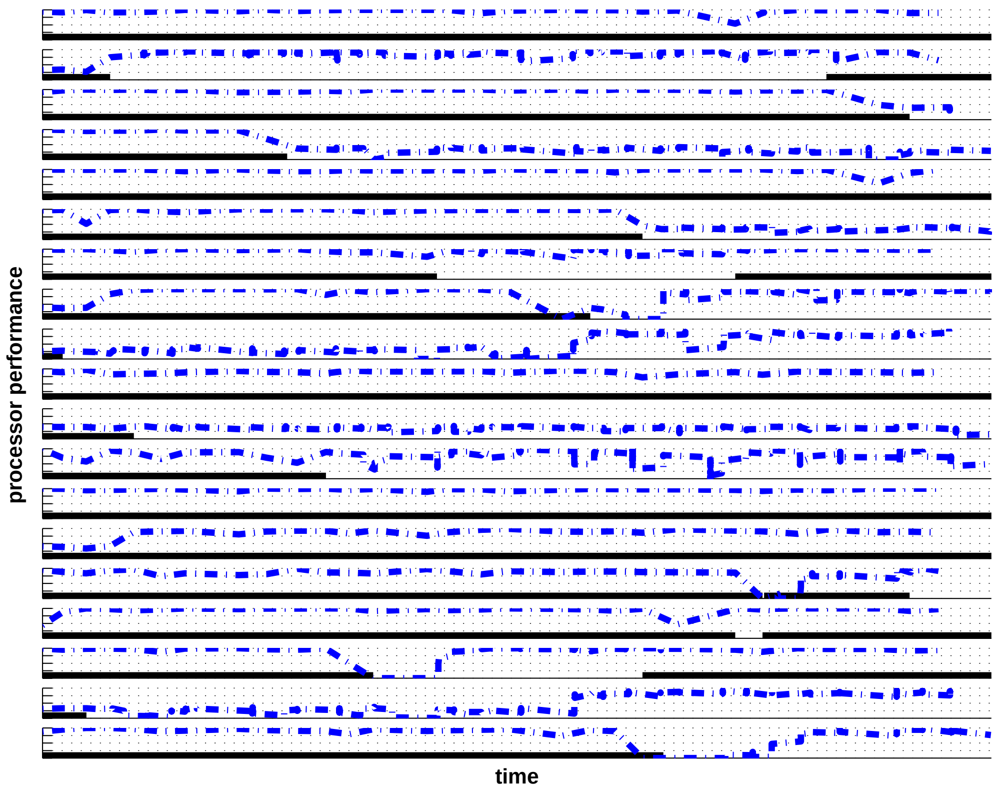


Figure 4.7: Behavior of a 64 process (42 active) swapping-enabled synthetic application. The y-axes are processor performance (higher is better); the x-axes are time. Broken lines show processor performance; the bars below show when processors were active.

Chapter 5

Swap Policies

In order to better gauge the efficacy of swapping, three swapping policies have been developed:

- greedy
- safe
- friendly

This chapter defines these policies, and introduces the concept of payback distance.

In order to better gauge the efficacy of swapping, several swap policies were developed. This chapter carefully defines these policies.

Swapping policies can be categorized by what kind of information they use, how much of that information is used, and how the information is used. The policies discussed here use application-intrinsic information such as iteration time, environmental information such as CPU availability, and a set of policy heuristics. Our swapping system parameterizes the swapping behavior so different policies can be created. After introducing a cost/benefit concept called *payback distance* in Section 5.1, we describe these parameters in Section 5.2, then extract three interesting policies for further study in Section 5.3.

5.1 Payback Algebra

With process swapping, the application must be paused for process state transfers, and the cost of halting progress may outweigh the performance advantage. As others have done [44, 37], we define a cost/benefit algebra that helps determine if process swapping will lead to a net benefit. The unique aspect of our process swapping algebra is the introduction of a ***payback*** distance, indicating the number of iterations (at an increased performance rate) required to offset the swapping cost:

$$\text{payback distance} = \frac{\text{swap time}}{\text{old iteration time} \times \left(1 - \frac{\text{old performance}}{\text{new performance}}\right)}$$

The swap time in this equation is the time required to transfer process state to another processor over a communication link modeled with latency α and bandwidth β:

$$\text{swap time} = \alpha + (\text{process size})/\beta$$

The performance metric in the payback equation can be any measure that increases with increased application performance, e.g., flop rate. The process swapping system has been tested with the following two performance measures: CPU availability (as derived from CPU load) and flop rate (as computed through a microbenchmark).

Consider an example. Say that the iteration time and swap time are both 10 seconds. If the new performance, after swapping, is twice the old performance then the payback distance is 2 iterations. In other words, it will take two iterations after swapping before the cumulative application progress will exceed that obtainable at the pre-swap rate. If the new performance is four times the old performance, the payback distance is 1 1/3 iterations. The greater the performance increase, the smaller the payback distance. Note that payback distance is by definition not linearly proportional to the performance increase.

Instead of calculating the potential performance benefit of a swapping decision over the entire remaining application execution time, we compute the number of iterations at the improved performance rate required to offset the swap cost. If the payback distance

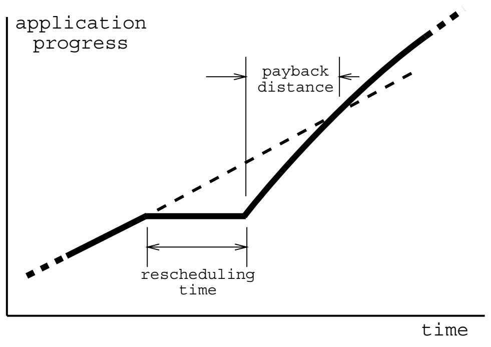


Figure 5.1: Payback distance.

is negative, there is no benefit. If the payback distance is positive, there is a potential benefit. The larger the payback distance, the longer it takes to recoup the swap overhead. Payback distance is useful for three reasons: (i) often, we do not know how many iterations are left in an application execution, e.g., the application runs until "convergence"; (ii) our environment is by definition not quiescent, so we cannot hope to realize the increased performance benefit forever; and (iii) a payback distance gives a parameter (the payback threshold) that we can tune to be more or less risk-averse in our swap policy.

Figure 5.1 illustrates the payback concept. The vertical axis of this figure is application progress, e.g., number of iterations completed, and the horizontal axis is time. During a process swapping event, the application pauses while the swap occurs, as indicated by the horizontal line segment. After swapping, increased application performance erases the swap cost. The time required to recoup the swapping overhead is the payback distance. It is worthwhile to note that if increased performance is not realized, there can be a net performance drop.

5.2 Policy Parameters

The following metrics are composed to create swapping policies.

payback threshold – The number of iterations, at the increased performance rate achieved after swapping, required to recover the cost of swapping is called the payback distance. Swap policies have a payback threshold that controls swapping: if the payback distance of a potential swap is less than the payback threshold, the swap is allowed. Smaller values of the payback threshold indicate more *risk-aversion*.

minimum process performance threshold – The performance gain of an individual process after a swap must be greater than a minimum improvement threshold, or swapping will not occur. Higher threshold values require more potential benefit from a swap, and indicate increased reluctance to swap for very small benefit. This parameter provides *swapping stiction*.

minimum application performance threshold – The performance gain of the overall application after a swap must be greater than a minimum improvement threshold, or swapping will not occur. Higher threshold values mean that the application will be less likely to needlessly hoard fast processors.

history – The amount of performance history used to predict processor performance can be tuned. Increasing the amount of history reduces the chance of being fooled by a transient load event, but can cause the application to miss good swapping opportunities. This parameter enables *swap frequency damping*.

5.3 Three Swapping Policies

The ***greedy*** policy has an infinite payback threshold, no minimum process improvement threshold, no minimum application improvement threshold, and uses no performance history. This policy swaps processes if there is any indication that application performance will increase. This policy does not care how great or little the performance is increased, nor does it care how long it will take to amortize the swap overhead. The greedy approach will use any and all processors available to it to promote application performance.

The ***safe*** policy uses a low payback threshold (0.5 iterations), a high minimum process improvement threshold (20%), no minimum application improvement threshold, and a large amount of performance history (5 minutes). This policy swaps processes only if the benefit is significant and the potential downside to the application is minimal. This policy looks at a significant amount of history so it is not fooled by instantaneous performance behavior. The safe policy requires that the overhead of swapping be recovered in a short amount of time, or swapping will not happen.

The ***friendly*** policy has no minimum process improvement threshold, a slight overall application improvement threshold (2%), and uses a moderate amount of performance history (1 minute). The friendly policy does not use computational resources unnecessarily. If swapping to a faster processor will not measurably increase the overall application performance, the swap will not occur. This policy promotes application performance, but judiciously uses compute resources, leaving more computing power available to other applications.

All three policies, when they decide to swap, swap the slowest active processor(s) for the fastest inactive processor(s), where fast and slow are defined by the performance metric used in the payback equation.

Chapter 6

Simulation

In this chapter, swapping is compared against three other performance enhancing techniques:

- do nothing
- dynamic load balancing
- checkpoint / restart

and different swapping policies are compared:

- greedy policy
- safe policy
- friendly policy

all using a simulation environment.

Since the target swapping usage scenario is a long-running application on a non-dedicated platform that is by its very nature dynamic, it is difficult to obtain reproducible results using experiments on real systems. Consequently, the efficacy of swapping is studied in a virtual environment. The environment is simulated using the SIMGRID toolkit [31]. This simulator models an execution environment, an iterative application of interest, one or more competing applications, and different performance-enhancing approaches for running the application, all of which are described in detail below.

6.1 Environment

The execution environment is simulated as a heterogeneous platform that consists of workstations connected via a 100-baseT Ethernet LAN. More specifically, simulated processors have computational performance in the hundreds-of-megaflops performance range, and are connected via a low latency shared communication link capable of transferring 6MB/s. MPI startup is assumed to be 3/4 second per process, which we have measured and found to be typical in such environments (see Section 4.4.1).

CPU load – CPU load characterization is a challenging task [15] and no widely accepted model has been identified. One approach is to "replay" traces of CPU load measurements obtained from monitoring infrastructures [47, 16]. This method, although realistic, makes it difficult to obtain a clear understanding of the simulation results. Indeed, it can be challenging to decouple the relative effectiveness of competing scheduling algorithms from idiosyncrasies of real CPU traces. Furthermore, it is difficult to gather enough traces for exploring a wide range of scenarios. Another approach is to use a simple stochastic model to simulate CPU load. The intent is to have a way to precisely tune the dynamics of CPU load (from "stable" to "dynamic"). The trade-off is that the generated CPU loads may not be completely realistic. Nevertheless, this is the approach used, as it allows for a clearer understanding of simulation results.

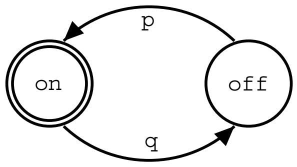


Figure 6.1: On-Off CPU load model.

CPU load is modeled in two ways. The first, simpler, model assumes a uniformly distributed process arrival, where the process run times are exponentially distributed. This model uses simple ON/OFF sources, which have been used extensively in other domains such as networking [1]. An ON/OFF source is a two-state Markov chain with fixed probabilities p and q of exiting each state, as depicted in Figure 6.1. Using this

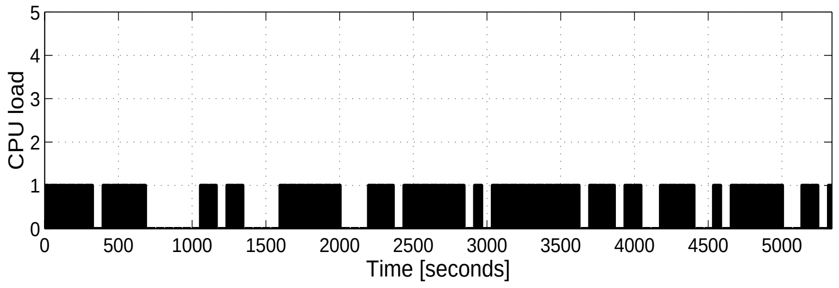


Figure 6.2: ON/OFF CPU load example.

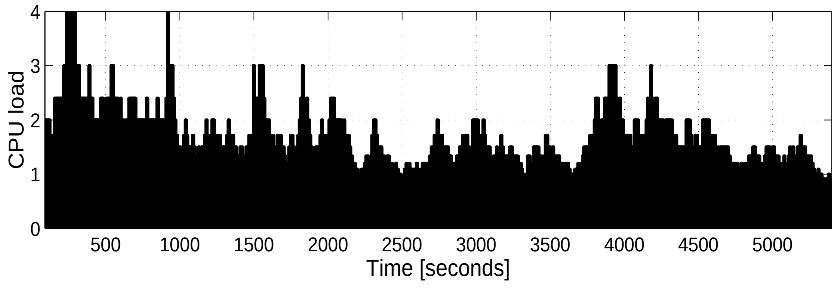


Figure 6.3: Hyperexponential CPU load example.

model, traces of CPU loads that take value 1 (ON, i.e. loaded with one competing compute-intensive process) or 0 (OFF, i.e. unloaded) were generated. This model simulates only one competing process as it is typical of the target environment. More complex loads can be easily generated by aggregating multiple ON/OFF sources. Figure 6.2 shows a typical CPU load trace generated using the ON/OFF source model (using $p = .3$, $q = .08$). The distribution of process lifetimes using this model are shown in Figure 6.4 in linear and log-log space.

The second model used to simulate competing process load uses a degenerate hyperexponential distribution of process run times, as in [17]. Compared to the ON/OFF source model, this model better predicts the heavy-tailed nature of the process lifetime distribution [32, 25]. As in the previous model, process arrival adheres to a uniform random distribution. Unlike in the ON/OFF model, multiple simultaneous competing processes per processor are allowed. An example trace is shown in Figure 6.3. The

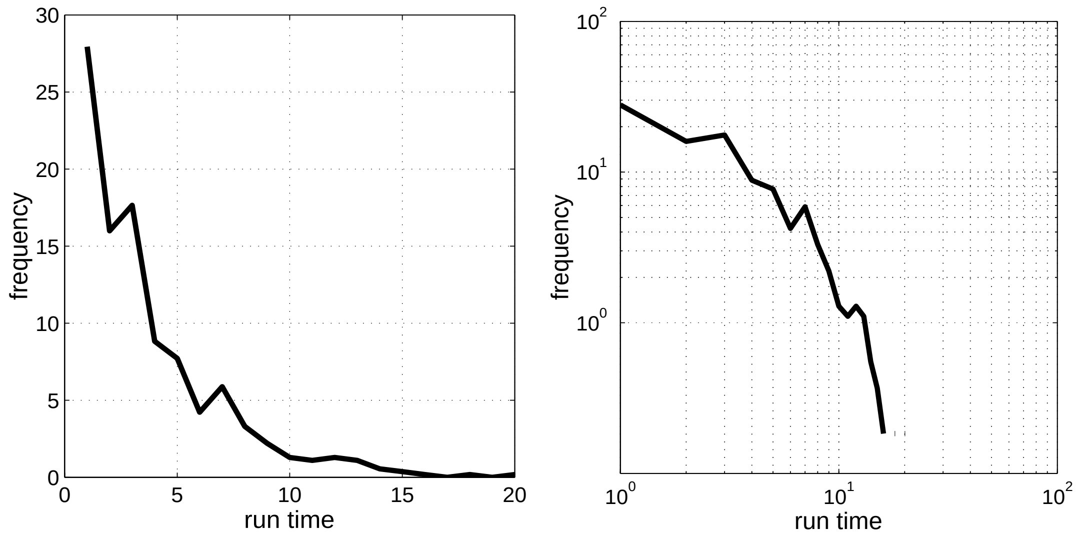


Figure 6.4: Example ON/OFF CPU load run length distribution. Linear and log-log scales shown.

distribution of process lifetimes using this model are shown in Figure 6.5 in linear and log-log space. Note the similarity between the hyperexponential CPU load and the measured CPU load on the Hewlett-Packard NOW shown in Figure 4.1.

The ON/OFF and hyperexponential CPU load models have limitations. There are other models, used by [32] for example, that even more accurately match real CPU load on some systems. However, the simple models used are conservative, and are sufficient to obtain the necessary first-order comparisons between the different algorithms and policies. More complex models and the use of CPU load traces are left for future work.

Communication – A single, shared network link with latency α and bandwidth β is assumed. Messages compete for a fixed amount of communication bandwidth, and collisions delay message transmission.

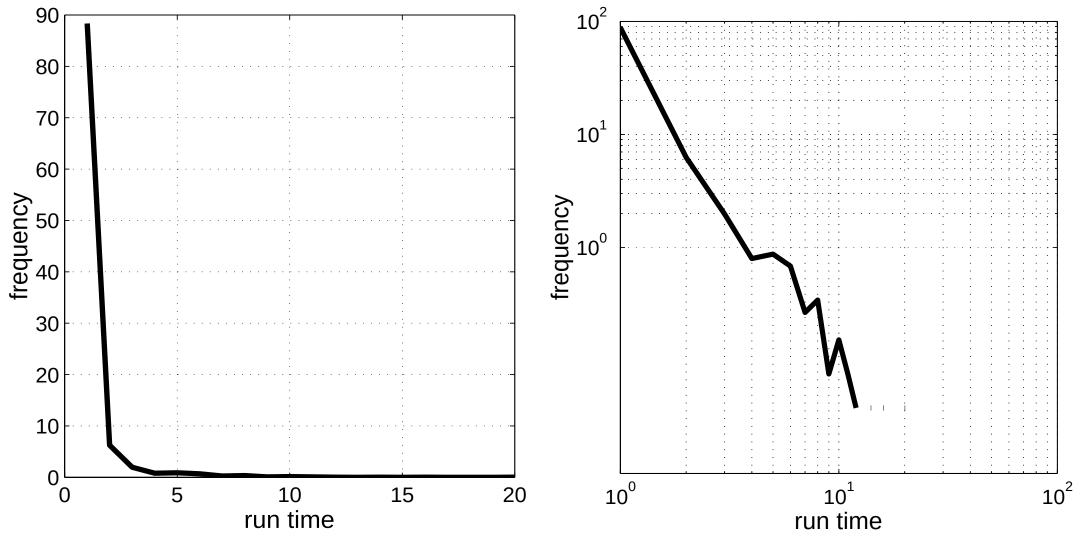


Figure 6.5: Example hyperexponential CPU load run length distribution: linear and log-log scales shown.

6.2 Application

In our simulations the iterative application of interest has a range of possible execution characteristics: (i) computation time per iteration on an unloaded processor are in the 1-5 minute range; (ii) the amount of data that a processor must communicate in each iterations is in the 1KB-1GB range; (iii) the process size, or amount of application state information (process state) that needs to be transferred during a process swap (or a checkpoint/restart), ranges from 1KB to 1GB, per processor. These values were chosen to span the range of most parallel applications, and because the balance of application iteration time to process swap time is a key factor to the effectiveness of swapping (as will be shown later, long swap times relative to iteration times can cause performance degradation).

For each application run, a constant amount of work, amount of communication data, and amount of application state were chosen. The application execution time is then dictated by the variability in resource availability and the performance-enhancing optimization chosen.

Initial schedule – The simulated application computes an optimal initial application schedule. For load balancing the work is partitioned into unequal size chunks to balance processor iteration times at the beginning of execution. For other techniques the application workload is partitioned into equal size chunks. The initial schedule always uses the fastest performing processors at the time of application startup.

6.3 Performance Enhancing Techniques

The simulation environment supports the four performance enhancing techniques, described below.

Do nothing – As an experimental control, the execution of an application without any performance enhancing techniques is noted.

Process swapping – The application startup cost, including over-allocation, is simulated. At each application iteration, if a swap occurs, the simulation computes the cost of transferring application state from one or more active processes to one or more inactive processes. Each application iteration incurs a slight (0.1 second) delay at each iteration to account for the communication with the swap handler.

Dynamic load balancing (DLB) – The DLB strategy redistributes work at each iteration so that the iteration times of all the processors are perfectly balanced given their respective performance. The overhead of doing the actual load balancing (i.e. exchanging data among processors) is not accounted for in simulation — the assumption is that it is instantaneous. Consequently, the DLB application execution times obtained in simulation are lower bounds on what could be obtained in practice. Initial application startup is accounted for.

Checkpoint/restart (CR) – The CR strategy is simulated as follows. Application startup cost is simulated. Then, at each iteration, the execution rate is analyzed. If performance can be increased by using another set of processors, based on the same criteria used to evaluate process swapping decisions (and including the same 0.1 second per iteration delay), the application is checkpointed. It is assumed that application state

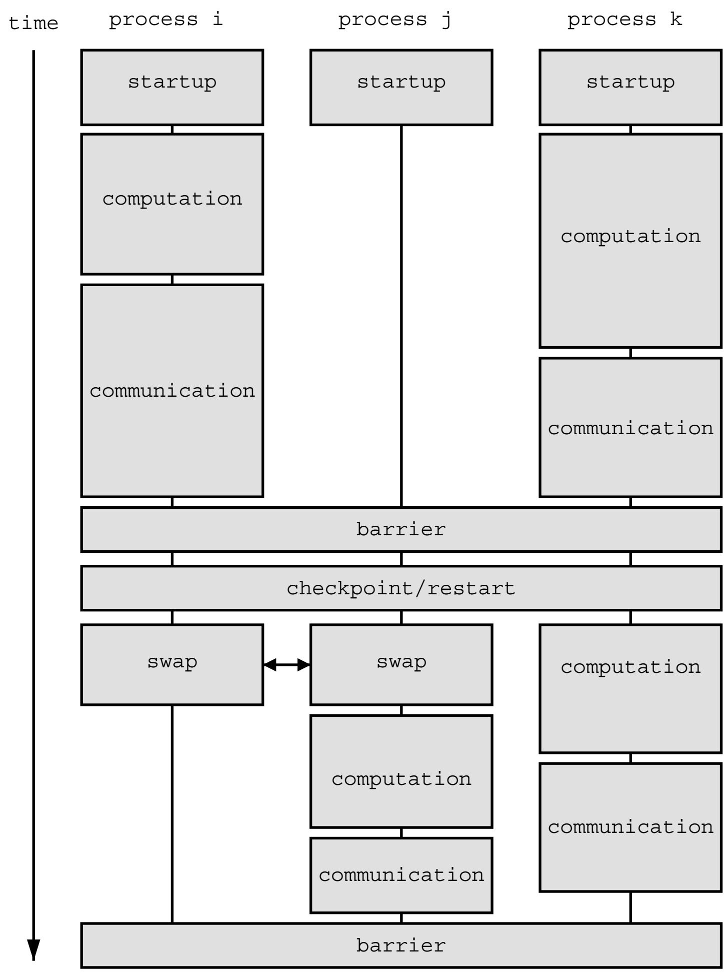


Figure 6.6: Simulation architecture

information is written to a central location. Upon application restart, the checkpoint is read by each process, and execution resumes. The simulations account for the overhead of writing and reading the checkpoint. They do not account for the delay incurred in computing a new application schedule, nor is there any “cool off” period to wait for the execution environment to become quiescent (which may be needed to compute a new schedule).

6.4 Simulation Architecture

The SIMGRID application is shown in Figure 6.6 for the case of two active processes and three total processes. In this figure, time increases downward. Each process performs a cyclic set of activities. First, the application is started. Then each active process performs some computation, followed by communication. In this example, process j is inactive initially. All communication occurs over a single shared network link. After communicating, the application reaches a barrier. The length of the barrier is zero for the NOP and DLB strategies, and 0.1 second for the SWAP and CR strategies. Following the barrier are two optional activities. If the strategy is CR and the application is checkpointed and restarted, there is another barrier-like activity that represents the checkpoint and subsequent restart of the application. Other strategies do not perform this activity. If the strategy is SWAP, and a swap is required, then the processes participating in the swap exchange process state information. The non-swapping processes proceed to compute the next application iteration. This sequence continues until the application ends.

6.5 Experimental Results

6.5.1 Evaluation of swapping vs. competing approaches

Four techniques are examined using the (more conservative) ON/OFF load model: (a) do nothing (NOP); (b) process swapping using the greedy policy (SWAP); (c) dynamic load balancing (DLB); and (d) checkpoint/restart (CR). It will be shown that SWAP generally performs favorably as compared to the other techniques across a range of application characteristics and *environment variability*. Environment variability is defined by the frequency and magnitude of change in available resource performance. For example, a quiescent, or low variability, environment has slowly changing performance characteristics. Note that this does not mean that all processors are unloaded; rather, it means that the load of any given processor is constant. Similarly, a highly dynamic environment (high variability) has frequent and dramatic changes in available

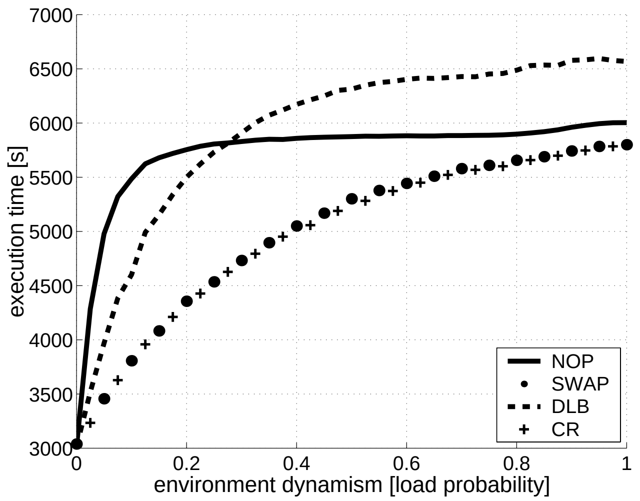


Figure 6.7: Execution time of various performance enhancing techniques across the full range of environment variability.

performance, caused by frequent, heavy, and short jobs.

Swapping provides benefit in moderately dynamic environments. The goal of our first experiment is to determine how well swapping works relative to the other performance-enhancing techniques. The performance of the four techniques is compared across a range of environments. The environment is varied from completely quiescent (defined as no changes in load during the entire application run) to completely dynamic (defined as a change in the performance of each processor multiple times per application iteration).

Figure 6.7 shows application execution time for the four techniques as a function of environment variability. In quiescent environments, shown on the left side of the figure, there is little difference between the techniques. Similarly, in highly dynamic environments, shown on the right side of the graph, the techniques tend toward convergence because the environment is too chaotic for any technique to do well. However, in moderately dynamic environments we see that DLB, CR, and SWAP all perform better than NOP (up to 40% better). The number of active processors used in this data is 4, the total number of processors 32, and the process size is 1MB.

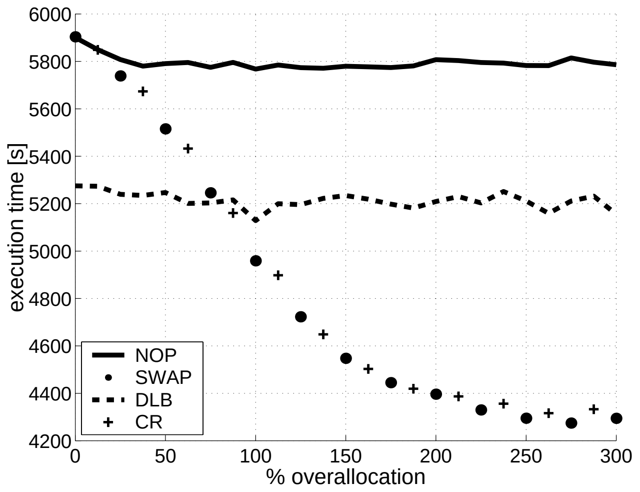


Figure 6.8: Execution time of various performance enhancing techniques across a range of over-allocation (8 active processes).

It is interesting to note that DLB does not perform very well in highly dynamic environments. When the environment becomes dynamic due for example to frequent interactive jobs, DLB chooses uneven work sizes, but the performance changes quickly and the application is left computing a lot of work on a (suddenly) slow processor. In this condition, load balancing actually results in more load imbalance.

Swapping performs better with more over-allocation. A second experiment shows the importance of over-allocation to process swapping, and the importance of a large available processor pool to checkpoint/restart.

Figure 6.8 shows application execution time over a range of over-allocation. As more spare processors are available, SWAP and CR performance both improve. Practically speaking, substantial benefit from SWAP requires 100% over-allocation. However, in this environment swapping can match the performance benefits of DLB with only 75% over-allocation. Similarly, for this environment CR requires an available processor pool as large as the number of active application processes. DLB consistently outperforms NOP. However, both SWAP and CR double the performance gain of DLB when the over-allocation is substantial. The slight drop in NOP execution time is due to the fact that the pre-execution scheduler has more options for initial process placement. In this case, the environment has a load probability of 0.2, which is moderately dynamic. The process size is 1 megabyte.

Ideally, the NOP and DLB performance should be constant across all over-allocation amounts. The fact that there is slight variation is due to the randomization used in the simulation.

The effectiveness of SWAP drops as process size increases. The goal of a third experiment is to find the break-even condition where the cost of process swapping outweighs the benefits.

The cost of swapping is directly related to the amount of data to be transferred when swapping. Figure 6.9 shows the effect of process size on the performance techniques. Since NOP and DLB do not need to save process state, their performance does not depend on process size. However, in the environment studied, both SWAP and CR transition from being beneficial at a process size of 1MB to somewhat harmful at a process size of 60MB (in highly dynamic environments). As process size increases from 60MB to 1GB, the SWAP and CR curves continue to rise, indicating increasing performance loss as process size increases. This is shown in Figure 6.10. The swap time for a 1GB process size is 120 seconds. This is large compared to the 50 second iteration time in this example.

In general, SWAP shows a performance drop when the ratio of application iteration time to swap time becomes small. When this happens, in the best case swapping does not happen and the performance matches the NOP case. In the worst case, swapping happens but never provides a net benefit, ultimately hurting application performance. As a general rule, for SWAP to be beneficial the swap time should be shorter than the application iteration time. It should be noted that this is an expected result: as shown in [37, 25, 32], process migration of any kind suffers as migration costs escalate.

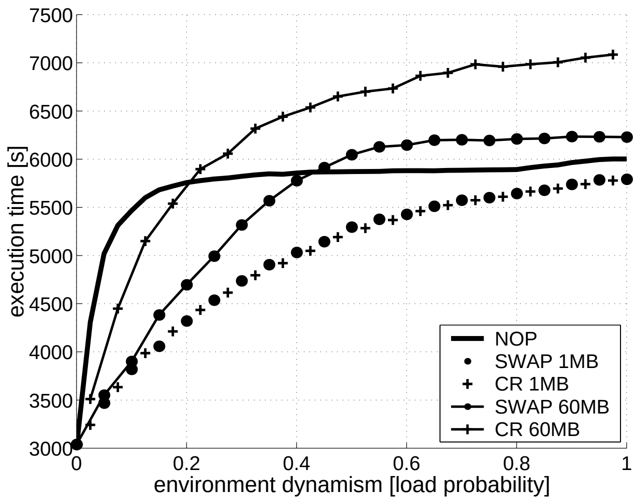


Figure 6.9: Execution time of various performance enhancing techniques for two selected process sizes.

6.5.2 Evaluation of three swapping policies

The greedy policy provides the largest performance boost. This experiment compares various process swapping policies, across a range of environments. The goal of this experiment is to determine if any single policy consistently performs better than the other policies, and to determine the improvement relative to the NOP condition. Refer to Section 5.3 for the definitions of the swap policies.

Figure 6.11 shows application execution time for the NOP technique, and for three swapping policies. For moderately dynamic environments, the greedy policy provides a maximum 40% performance increase. The friendly policy does surprisingly well in moderately dynamic environments, almost keeping pace with the greedy policy. In more dynamic situations, however, friendly application performance decreases dramatically. The safe policy, as expected, provides lower performance benefit than the greedy approach, but at slightly lower risk — in dynamic environments the safe policy outperforms the greedy policy. In this example, the total number of processors is 32, the active number of processors is 4, and the process size is 100MB.

When the process size becomes large, only the safe policy is appropriate. Figure 6.12 shows application execution time for the various swapping policies when the

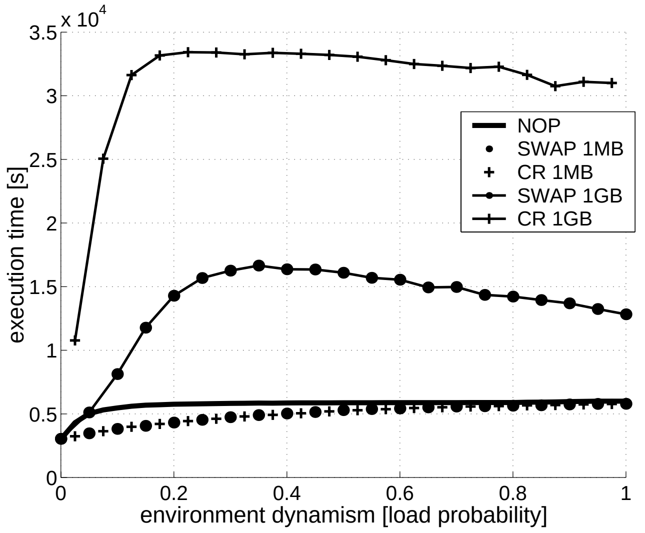


Figure 6.10: Execution time of various performance enhancing techniques for two selected process sizes.

process size is large. At 1 GB, the process swap time is twice that of the application iteration time in this example. By the time the process state has been swapped, the environment has changed, requiring another swap. The application spends all its time swapping, chasing an unobtainable performance; performance suffers. This example is for two active processes out of 32 total processes.

6.5.3 Effect of CPU load distribution

It has been observed that some unix process lifetimes follow a hyperexponential model [32, 25], which predicts the long-tailed distribution resulting from a few very long running processes better than an exponential model. The presence of these very long-running jobs effectively smoothes processor performance over time, allowing swapping to be more effective.

Figure 6.13 shows the performance of swapping versus NOP, DLB, and CR assuming a hyperexponential load model. This model predicts more long-running competing applications than with the ON/OFF model. Note that swapping remains viable under this CPU load model. In fact, the larger percentage of long-running jobs created under

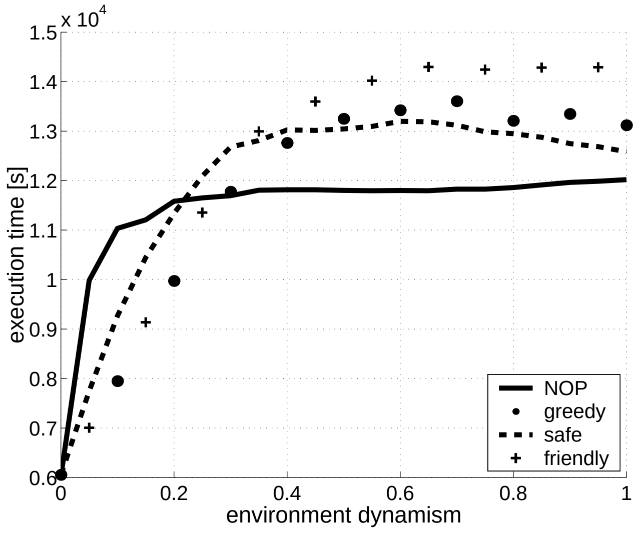


Figure 6.11: Execution time for various swapping policies across a range of environment variability.

the hyperexponential model increases the range of environments over which swapping is beneficial.

As previously noted, this model may be better at predicting some observed processes. However, it is less conservative. The bulk of the analysis in this these was done with the exponential on-off load model because it represents a more difficult environment. If swapping can perform well in an exponential world, it is certainly well-suited to an environment where process lifetimes follow a hyperexponential trend.

6.6 Summary

In this chapter two CPU load models were developed. The first (exponential process time distribution) is simple and conservative, and represents a more difficult environment in which to prove the effectiveness of swapping. The second (hyperexponential) is shown to be a reasonable match to the CPU load measured on the Hewlett-Packard NOW. The first model is used for most of the analysis, however, because the second model is shown to create an easier environment in which to operate. By showing that process swapping is effective in difficult environments, perhaps more difficult than is

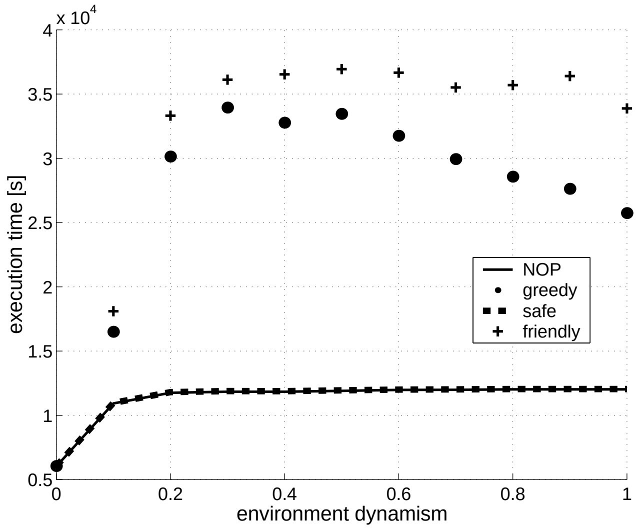


Figure 6.12: Execution time for various swapping policies where process size is large.

experienced in practice, we build confidence that the technique is broadly useful.

In an environment simulating a collection of personal workstations in a typical local area network, process swapping was compared to dynamic load balancing and checkpoint/restart. This comparison spanned a range of application characteristics and a range of environments from quiescent to highly dynamic. In general, process swapping outperforms checkpoint/restart and performs comparably with dynamic load balancing.

Specifically, it was shown that process swapping becomes very effective with 100% process over-allocation, and can match dynamic load balancing with 75% over-allocation. It was also shown that process swapping is more beneficial for lower ratios of process swap time to application iteration time. Process swapping (and checkpoint/restart) become harmful when the swap time (checkpoint/restart time) is as large or larger than the application iteration time.

Three swap policies were analyzed, and it was shown that through policy design the risk of decreased performance can be balanced against the reward of improved performance. The safe policy did not yield as large a performance increase as the greedy policy. However, the safe policy was not prone to large performance loss that the greedy policy could give.

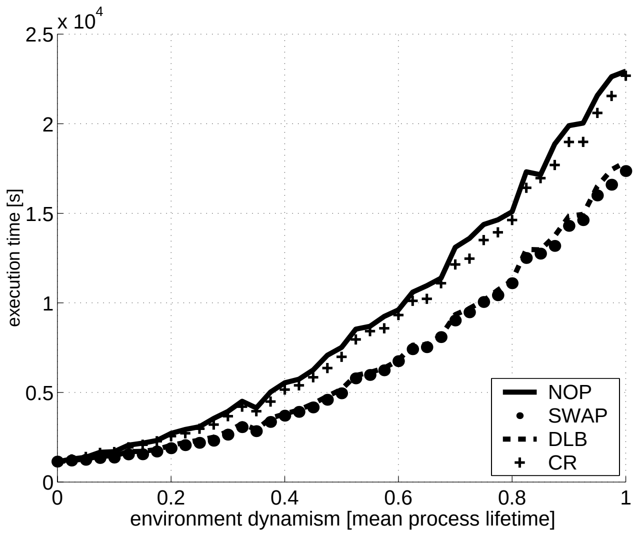


Figure 6.13: Execution time for various performance enhancing techniques under a hyperexponential load model.

An environmentally friendly policy that does not frivolously use resources was shown to give some performance benefit, even though it does not hoard all of the fast processors.

Chapter 7

Conclusion

The architecture of a system to improve performance of iterative MPI applications has been presented. By overloading MPI calls, this user-level infrastructure can add run-time scheduling to existing MPI applications with as few as 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 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 validation has been done on desktop resources within a research environment and a production commercial environment. The greedy policy used showed benefit, but also showed a tendency to swap needlessly.

A performance analysis of process swapping was presented. This analysis was done using a simulation environment, complementing the actual process swapping implementation. The regime within which swapping is beneficial ($>$ 100% over-allocation, swap time $<$ iteration time) relative to dynamic load balancing and checkpoint/restart, was discussed.

Three policies used to swap MPI processes were developed, along with a novel payback metric used to tune these policies. It was shown that greedy swapping policies have the best performance potential, but also have the most risk as performance can also be reduced. Risk-averse swapping policies provide reduced performance benefit, but are not susceptible to the performance loss that greedy policies are. Finally, swapping policies that avoid hogging fast processors are shown to still have some application performance benefit.

This work is extensible in several directions. Augmenting the simulation with CPU load traces that better reflect actual environments will help ensure that swap policies are beneficial. The actual implementation could be improved to add message forwarding (and accommodate asynchronous MPI calls) and to increase robustness and scalability. Swapping could be combined with emerging middleware infrastructure that will extend the reach outside of the area of iterative applications. Currently, work has been done to integrate process swapping into the GrADS [28] architecture.

Process swapping is a valuable performance enhancing technique. It provides performance on par with dynamic load balancing and checkpoint/restart, and does so simply and easily by changing as few as three lines of source code in an iterative application. This combination of performance and ease-of-use is an important contribution as it applies to an entire class of MPI applications. The simplicity of design allows programs to be easily retrofitted, making this technique accessible to both new and existing applications.

Appendix A

User's Guide

These instructions are specific to MPICH version 1.2.4. Other MPI implementations should act similarly. The instructions are specific to code written in C; no specific provision has been made to support Fortran or C++.

A.1 Getting Started

A.1.1 Compiling a swap-enabled application

In the simplest case, three lines of code need to be added or modified in order to swap-enable an application:

1. Include `mpi_swap.h` instead of `mpi.h`.
2. Register the iteration loop variable with `swap_register()`.
3. Add a call to `MPI_Swap()` just inside the iteration loop.

Then, you must modify your makefile, to include

- the location of the swap include files with `-I`*dir*;
- the location of the swap library with `-L`*dir*;
- the linkage of the swap library with `-lswap`.

Depending on your application, you may need to (or want to) further customize your source code:

- Any statically allocated variable that needs to be communicated when swapping must be registered with `swap_register()`.
- The symbols `global_rank` and `global_size` are predefined and contain the global size and rank of each MPI process. If you only want the true global root process to perform some task (like parsing command line options, or reading or writing configuration or data files) then you will have to check the global rank; do not trust the response from `MPI_Comm_rank()` and `MPI_Comm_size()`, as they will return the active (or inactive) rank and size.
- By default, all dynamically allocated memory is communicated to the other process during a swap. If you want local-only memory (for example, for a scratch computation area), then use the commands `real_malloc()`, `real_calloc()`, and `real_realloc()`.

A.1.2 Running a swap-enabled application

There are many ways to run a swap-enabled application. The easiest, however, is to use the swap dispatcher. First, ensure that the swap dispatch is running. If you do not have a system-wide always-on swap dispatch dæmon, then start your own:

```
% swap_dispatch
```

Next, launch your application. The two suggested command line options are `-ap` and `-swd`. Say you have an application called `foo` that you want to run on twelve processors total and want seven of these to be active. You launched your own swap dispatch on machine `my.host.com` and it bound itself to port 5505 (the default port). You would launch your application as follows:

```
% mpirun -np 12 foo -ap 7 -swd my.host.com:5505
```

You can log some informational output by using the `-slog` and -sv options, for example:

```
% mpirun -np 12 foo -ap 7 -swd host:5505 -slog foo.log -sv 3
```

To stop the swap dispatch, you can use the `swap_admin` utility:

```
% swap_admin -p 5505 quit
```

You can even use the `swap_admin` utility to stop a running swap manager, if you know its port number:

```
% swap_admin -p <port> quit
```

and the swap handlers also operate the same way; you can stop a running swap handler, if you know its port number, with

```
% swap_admin -p <port> quit
```

For a list of common swap services communication commands, see the tables later in this guide.

Table A.1: Important swapping files

name	description
`swap_dispatch`	The swap dispatcher — this always-on service must be running before starting a swap-enabled application, unless the user has self-started an application manager
`swap_mgr`	The swap manager — this is the application-specific manager. A swap manager must exist for each application. The swap manager is automatically started if the swap-enabled application contacts the swap dispatch. If a swap manager is manually started, it must be told the application name and the number of MPI processes.
`swap_hdl`	The swap handler – one of these is run per MPI process; this handles all the direct communication with the application. The swap manager launches these normally, although they can be launched manually.
`swap_admin`	The swap admin utility allows easy access to all the swap services by acting as a communication portal.
`swap_log`	The swap log utility — this utility allows logging of swap service information to a possibly remote file location.
`swap_vis`	The swap visualization utility is a graphical window onto the current performance of the MPI processes.
`mpi_swap.h`	The swap header file must be included in user code instead of `mpi.h`.
`libswap.a`	The swap library is linked with the application object code to create a swap-enabled executable.

A.2 Command Line Options

Table A.2 lists the command line options available to swap-enabled applications. Figures A.3-A.5 list the command line options for the swap services.

Table A.2: Run time command line options for swap-enabled applications

option	default	description
`-ap` #	# processes	The number of active processes.
`-swd` *host:port*	none	The location of the swap dispatcher.
`-swm` *host:port*	none	The location of the swap manager. Only necessary if the swap dispatch location is not given.
`-slog` *fname*	none	The file name of a local file for info logging.
`-sv` #	0	The swap verbose level. Swap information with value less than or equal to this level will be shown.
`-sd` #	0	The swap debug level. Swap information with value that mask positively against this value will be shown.

Table A.3: Swap Dispatcher command line options

option	default	description
`-p` #	5505	Starting port number. If this port is not available, climb until an available port is found.
`-d`	0	Display debug info.

Table A.4: Swap Manager command line options

option	default	description
`-n` *name*	none	Name of application (required).
`-s` #	0	Size of application (total number of processes) (required).
`-h`	0	Swap dispatch hostname.
`-p` #	none	Swap dispatch port.
`-d`	0	Display debug info.

Table A.5: Swap Handler command line options

option	default	description
`-n` *name*	none	Name of handler (required).
`-h`	localhost	Swap manager hostname.
`-p` #	6505	Swap manager port.
`-d`	0	Display debug info.

Table A.6: Swap Admin command line options

option	default	description
`-h`	localhost	Swap manager hostname.
`-p` #	6505	Swap manager port.
command	0	The command to send.

Bibliography

[1] ADAS, A. Traffic models in broadband networks. *IEEE Communications Magazine 35*, 7 (July 1997), 82–89.

[2] ALLEN, G., ANGULO, D., FOSTER, I., LANFERMANN, G., LIU, C., RADKE, T., SEIDEL, E., AND SHALF, J. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *International Journal of High Performance Computing Applications 15*, 4 (2001), 345–358.

[3] BATCHU, R., NEELAMEGAM, J., CUI, Z., AND ET AL. MPI/FT: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid* (May 2001).

[4] BERMAN, F., WOLSKI, R., CASANOVA, H., CIRNE, W., DAIL, H., FAERMAN, M., FIGUEIRA, S., HAYES, J., OBERTELLI, G., SCHOPF, J., SHAO, G., SMALLEN, S., SPRING, N., SU, A., AND ZAGORODNOV, D. Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel and DIstributed Systems* (2003), 369–382.

[5] BHARGAVA, R., FOX, G., OU, C.-W., RANKA, S., AND SINGH, V. Scalable libraries for graph partitioning. In *Scalable Libraries Conference* (1993).

[6] BOSILCA, G., BOUTEILLER, A., CAPPELLO, F., DJILALI, S., FEDAK, G., GERMAIN, C., HERAULT, T., LEMARINIER, P., LODYGENSKY, O., MAGNIETTE, F., NERI, V., AND SELIKHOV, A. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Proceedings of SC'02* (2002).

[7] BOUDET, V., PETITET, A., RASTELLO, F., AND ROBERT, Y. Data allocation strategies for dense linear algebra kernels on heterogeneous two-dimensional grids. Tech. Rep. PP99-31, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France, 1999.

[8] CASANOVA, H., OBERTELLI, G., BERMAN, F., AND WOLSKI, R. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of Supercomputing 2000* (2000), pp. 75–76.

[9] CHANDRA, R., MENON, R., DAGUM, L., KOHR, D., MAYDAN, D., AND MCDONALD, J. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, October 2000.

[10] CYBENKO, G. Load balancing for distributed memory processors. *Journal of Parallel and Distributed Computing* (1989), 279–301.

[11] DAIL, H., BERMAN, F., AND CASANOVA, H. A Modular Scheduling Approach for Grid Application Development Environments. In *Journal of Parallel and Distributed Computing* (2002). to appear, Available as UCSD CSE Tech Report CS2002-0708.

[12] DAIL, H., OBERTELLI, G., BERMAN, F., WOLSKI, R., AND GRIMSHAW, A. Application-aware scheduling of a magnetohydrodynamics application in the legion metasystem. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)* (May 2000).

[13] DAIL, H., SIEVERT, O., BERMAN, F., CASANOVA, H., YARKHAN, A., VADHIYAR, S., DONGARRA, J., LIU, C., YANG, L., ANGULO, D., AND FOSTER, I. *Resource Management in the Grid*. Kluwer, 2003, to appear.

[14] DIEKMANN, R., MONIEN, B., AND PREIS, R. Load balancing strategies for distributed memory machines. In *Multiscale Phenomena and Their Simulation*, H. Satz, F. Karsch, and B. Monien, Eds. World Scientific, 1997, pp. 255–266.

[15] DINDA, P. The Statistical Properties of Host Load. *Scientific Programming 7*, 3-4 (1999), 211–229.

[16] DINDA, P., GROSS, T., KARRER, R., LOWEKAMP, B., MILLER, N., STEENKISTE, P., AND SUTHERLAND, D. The Architecture of the Remos System. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 2001)* (August 2001).

[17] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. *Conf. on Measurement & Modelling of Comp. Syst., (ACM SIGMETRICS)* (May 1988), 63–72.

[18] ELSÄSSER, E., MONIEN, B., AND PREIS, R. Diffusion schemes for load balancing on heterogeneous networks. *Theory of Computing Systems 35* (2002), 305–320.

[19] Entropia, Inc. `http://www.entropia.com`.

[20] FAGG, G., AND DONGARRA, J. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the Euro PVM/MPI User's Group, Berlin, Germany* (2000), pp. 346–353.

[21] FEDAK, G., GERMAIN, C., NRI, V., AND CAPPELLO, F. XtremWeb : A Generic Global Computing System. In *Proceedings of the Workshop on Global Computing on Personal Devices* (May 2001).

[22] FINK, S. J., BADEN, S. B., AND KOHN, S. R. Flexible Communication Mechanisms for Dynamic Structured Applications. In *Proceedings of Workshop on Parallel Algorithms for Irregularly Structured Problems* (1996), pp. 203–215.

[23] FITZGERALD, S., FOSTER, I., KESSELMAN, C., VON LASZEWSKI, G., SMITH, W., AND TUECKE, S. A Directory Service for Configuring High-performance Distributed Computations. In *Proceedings of the 6th IEEE Symp. on High Performance Distributed Computing* (1997), IEEE Computer Society Press, pp. 365–375.

[24] Forum, M. P. I. MPI: A Message-Passing Interface Standard. Tech. Rep. UT-CS-94-230, Dept. of Computer Science, University of Tennessee, Knoxville, 1994.

[25] Harchol-Balter, M., and Downey, A. B. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems 15*, 3 (1997), 253–285.

[26] Heirich, A., and Arvo, J. A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing. *The Journal of Supercomputing 12*, 1–2 (1998), 57–68.

[27] United Devices, Inc. `http://www.ud.com`.

[28] Kennedy, K., Mazina, M., Mellor-Crummey, J., Cooper, K., Torczon, L., Berman, F., Chien, A., Dail, H., Sievert, O., Angulo, D., Foster, I., Gannon, D., Johnsson, L., Kesselman, C., Aydt, R., Reed, D., Dongarra, J., Vadhiyar, S., and Wolski, R. 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).

[29] Kohn, S. R., and Baden, S. B. Parallel software abstractions for structured adaptive mesh methods. *Journal of Parallel and Distributed Computing 61*, 6 (June 2001), 713–736.

[30] Le Sergent, T., and Berthomieu, B. Balancing Load under Large and Fast Load Changes in Distributed Computing Systems - A Case Study. In *Conference on Algorithms and Hardware for Parallel Processing* (1994), pp. 854–865.

[31] Legrand, A., Marchal, L., and Casanova, H. Scheduling Distributed Applications: The SimGrid Simulation Framework. In *Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03), Tokyo, Japan* (May 2003). to appear.

[32] LELAND, W. E., AND OTT, T. J. Load-balancing heuristics and process behavior. *ACM SIGMETRICS* (May 1986), 54–69.

[33] LITZKOW, M., LIVNY, M., AND MUTKA, M. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)* (1988).

[34] PROTIC, J., TOMAEVIC, M., AND MILUTINOVIC, V. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press and John Wiley & Sons, Inc., July 1997.

[35] RIBLER, R. L., VETTER, J. S., SIMITCI, H., AND REED, D. A. Autopilot: Adaptive control of distributed applications. In *Proceedings of the 8th IEEE Symposium on High-Performance Distributed Computing* (1998), pp. 172–179.

[36] SHAO, G., BERMAN, F., AND WOLSKI, R. Master/slave computing on the grid. In *Proceedings of the 2000 Heterogeneous Computing Workshop* (2000), pp. 3–16.

[37] SHAO, G., WOLSKI, R., AND BERMAN, F. Modeling the cost of redistribution in scheduling. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing* (1997).

[38] SMALLEN, S., CASANOVA, H., AND BERMAN, F. Applying Scheduling and Tuning to On-line Parallel Tomography. In *Proceedings of Supercomputing 2001 (SC'01)* (November 2001).

[39] SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., AND DONGARRA, J. *MPI: The Complete Reference*. MIT Press, 1998.

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

[41] SU, A., BERMAN, F., WOLSKI, R., AND MILLS STROUT, M. Using AppLeS to Schedule Simple SARA on the Computational Grid. *The International Journal of High Performance Computing Applications 13*, 3 (1999), 253–262.

[42] TANG, H., AND YANG, T. Optimizing threaded mpi execution on smp clusters. In *Proceedings of the 15th ACM International Conference on Supercomputing* (2001), pp. 381–392.

[43] TOONEN, K., AND FOSTER, I. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing* (2003). to appear.

[44] VADHIYAR, S., AND DONGARRA, J. A Metascheduler for the Grid. In *Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing* (July 2002). To appear.

[45] VADHIYAR, S., AND DONGARRA, J. SRS - a framework for developing malleable and migratable parallel applications for distributed systems. *International Journal of High Performance Applications and Supercomputing* (2003). To appear.

[46] WOLSKI, R. Dynamically forecasting network performance using the Network Weather Service. *Cluster Computing 1*, 1 (1998), 119–132.

[47] WOLSKI, R., SPRING, N., AND HAYES, J. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems 15*, 5–6 (1999), 757–768.

[48] WONG, F., AND DEMMEL, J. UC Berkeley CS 267 course programming assignment 4 at
`http://www.cs.berkeley.edu/~fredwong/`
`cs267_Spr99/assignments/assignment4.html`.

[49] ZHOU, S. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Transactions on Software Engineering 14*, 9 (September 1988), 1327–1341.

[50] ZHU, W., AND STEKETEE, C. An experimental study of load balancing on Amoeba. In *First Aizu International Symposium on Parallel Algorithms/ Architecture Synthesis* (1995), pp. 220–226.