Policies for Swapping MPI Processes

Otto Sievert[1] Henri Casanova [1,2]

[1] Department of Computer Science and Engineering
[2] San Diego Supercomputer Center
University of California at San Diego

[osievert,casanova]@cs.ucsd.edu

Abstract

Despite the enormous amount of research and development work in the area of parallel computing, it is a common observation that simultaneous performance and ease-of-use are elusive. We believe that ease-of-use is critical for many end users, and thus seek performance enhancing techniques that can be easily retrofitted to existing parallel applications. In a previous paper we have presented MPI process swapping*, a simple add-on to the MPI programming environment that can improve performance in shared computing environments. MPI process swapping requires as few as three lines of source code change to an existing application. In this paper we explore a question that we had left open in our previous work: based on which policies should processes be swapped for best performance? Our results show that, with adequate swapping policies, MPI process swapping can provide substantial performance benefits with very limited implementation effort.*

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 [7, 37, 32] by which users can write applications with well-defined Application Programming Interfaces (API) and use various parallel platforms. In this paper we focus on message passing, and in particular on the Message Passing Interface (MPI) standard [20]. MPI provides the necessary abstractions for writing parallel applications and harnessing multiple processors, but the primary parallel computing challenges of application scalability and performance remain. While these challenges can be addressed via intensive performance engineering and tuning, end users often lack the time and expertise required. As a result, parallel computing often enjoys ease-of-use or high performance, but rarely both at the same time. We believe that 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.

In [35] we have presented such a technique, called *MPI Process Swapping*. MPI process swapping improves performance by dynamically choosing the best available resources throughout the execution of a "long-running" 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 desires N processors to run, due to memory and/or performance considerations. Our approach *over-allocates* $N + M$ processors so that the application only runs on N processors, but has the opportunity to swap any of these processors with any of M spare processors. We impose the restriction that data redistribution is not allowed: for simplicity and ease-of-use the application is "stuck" with the initial data distribution. Although MPI swapping will often be sub-optimal, it is a practical solution for practical situations and it can be integrated into existing applications easily.

Why keep M spare processors? While one could run on all $N + M$ processors, most data-parallel applications have speedup curves that are parabolic with the number of processors (due to scaling overheads). Swapping allows an application to run with its optimal number of processors N, while keeping a few processors in reserve in case performance falls on these N. Furthermore, in a dynamic environment, using more processors can increase the risk of slowdown due to competition for computing resources.

For the moment we target the broad class of iterative ap-

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

plications. Process swapping can be added to an existing iterative application with as few as three lines of source code change. We target 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, as evidenced by the continued rise of commercial distributed computing solutions offered by Entropia [16], Avaki [8], Microsoft [13], United Devices [25], Platform [31], and others. Although our approach could be used when resource reclamations and failures occur, in this work we focus solely on performance issues. We target a usage scenario in which only a few parallel applications run on the platform simultaneously, the idea being for these applications to benefit from mostly unloaded resources (if the workload consists predominantly of parallel applications then the platform of choice should be a batch-scheduled cluster).

In [35] we described an architecture and prototype implementation of MPI process swapping, and we validated this in a production environment. In this paper we investigate a fundamental question that had been left open in that work: based on which policies should process swapping decisions be made? Our initial implementation used a naïve and greedy swapping policy, simple to implement but of unknown efficacy. In this paper we propose a number of policies for deciding when and how process swapping should occur. To evaluate and compare these policies we use simulation as it enables reproducible experiments for comparing competing strategies. Based on our results we conclude that, with appropriate swapping policies, process swapping is as beneficial as other performance enhancing techniques, with much lower implementation cost.

2. Motivation and related work

Dynamic Load Balancing (DLB) is one of the best known methods for achieving good parallel performance in unstable conditions. DLB techniques have been developed and used for scenarios in which the application's computational requirements change over time [9, 10, 15, 24] and scenarios in which the platform changes over time [43, 27, 44]. In this work we target the latter scenario and DLB is thus an attractive approach, but we find that it has limitations. First, DLB requires an application that is amenable, in the limit, to arbitrary data partitioning. Some algorithms demand fundamentally rigid data partitioning. Second, 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. Lastly, 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. In this last case, we note that a DLB implementation could further improve performance through the use an over-allocation mechanism similar to the one used in our approach.

Another way for an application to adapt to changing conditions is Checkpoint/Restart (CR). While CR is usually used for fault-tolerance, we discuss how it can be used for performance by adapting to changing resources. 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, parallel (heterogenous) checkpoint/restart of MPI applications is a difficult task; it remains the subject of several active research projects [38, 3, 17, 5]. However, note that application-level checkpointing can be implemented with limited effort for iterative applications as demonstrated in [2, 40]. Finally, checkpointing may incur significant overheads depending on the application and compute platform (e.g. the time to save application state can be significant, and startup costs are incurred for each restart).

We claim that MPI process swapping can potentially achieve high performance while being straightforward to integrate into existing applications, and especially significantly easier to implement than DLB.

Our work is related to a number of efforts to enhance the MPI runtime system. Our implementation of MPI process swapping is a sleight-of-hand played in MPI user space, rather than a true infrastructure feature. Checkpointing facilities such as those provided by fault-tolerance extensions to MPI [38, 3, 17, 5] provide better-integrated support and improve the capabilities of the MPI system. These checkpointing/migration mechanisms could be combined with our process swapping services and policies, improving the robustness and generality over the current process swapping solution. In particular, a checkpointing facility would allow a better process swapping implementation by (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 techniques and policies with the cycle-stealing facilities of desktop computing systems like Condor [30], XtremWeb [18] or other commercial systems [16, 25] would yield a powerful system. These systems evict application processes when a resource is reclaimed by its owner. By combining our swapping poli-

cies 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 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 [5] could be used.

MPI process swapping shares performance ideas and methodologies with traditional application schedulers such as those found in the AppLeS [4] and GrADS [26] projects. These systems are also concerned with achieving high performance in the face of dynamic parallel execution environments. Additionally, they strive for ease-of-use, knowing that common users such as disciplinary scientists are often not parallel computing experts. The performance measurement and prediction techniques used in process swapping have much in common with these projects; all use application and environmental measurements (e.g. via the NWS [41], Autopilot [33], or MDS [19]) to improve application performance.

3. MPI process swapping

In this section we briefly review our implementation of MPI process swapping. See [35] for more details.

MPI over-allocation – Over-allocated, spare processors are left idle (i.e. blocking on an I/O call) and thus an application does not consume more resources because of over-allocation. We use this over-allocation technique because MPI-1.2 does not support adding processes to (or removing processes from) communicators. In this work we have used MPICH version 1.2.4 [22] and implement over-allocation with two private MPI communicators. All inter-process communication uses standard MPI calls, over these two private MPI communicators and over `MPI_COMM_WORLD`.

MPI-2 has support for adding and removing processors during application execution [21]. However, MPI-2 is not widely supported and this mechanism is more invasive to user code. Note that the latest Grid-enabled implementation of MPI, MPICH-G2 [39], supports the dynamic addition and removal of processes as specified in the MPI-2 standard; this could remove the need for over-allocation.

MPI process swapping runtime architecture – We have architected a runtime system that supports process swapping and automatically determines the best processors to use for a run of an MPI application. During execution a number of runtime services cooperate to (i) periodically check the performance of the processors; (ii) make swapping decisions; and (iii) enact these decisions. Each MPI process is accompanied by a *swap handler* which is a separate process responsible for coordination with other processes in the runtime system. The *swap manager* is a possibly remote process that is responsible for collecting information and making swapping decisions. All details on this architecture, its implementation, and the interactions between components can be found in [35].

Impact on application source code – For maximum transparency, our MPI process swapping implementation *hijacks* many of the MPI function calls. There are three types of modifications to the application code that must be done by the user. First, the code must include a `mpi_swap.h` header file. Second, a call to the `MPI_Swap()` function must be inserted inside the iteration loop of the application. Third, the user must register static variables that need to be saved and communicated when a swap occurs. This is done via a series of calls to the `swap_register()` function. Conceivably these steps could be automated with a compiler but they are straightforward for a user to implement. In [35] we give examples and describe our work with a real-world particle dynamics code for which only 4 lines of the original source code were modified.

The `MPI_Swap()` call, as currently implemented, demands a full application barrier, i.e., no communication messages can be outstanding at the point the `MPI_Swap()` call is made. An improved system has been designed, but not implemented as of this writing, to addresses this limitation through message forwarding. It is important to also note that this swapping implementation does not transfer non-MPI I/O channels; it is assumed that the only I/O in the main program iteration loop are MPI calls.

Verification and validation – In [35] we verified our design and validated our prototype implementation on a production intranet at a Hewlett-Packard research and development facility. Most of the workstations in this platform are used as personal computers and exhibit various levels of load variations. We observed and reported the effect of swapping throughout runs spanning several hours. Our focus was on testing our approach and we used a naïve and greedy swapping policy. As a result, swapping decisions were often inadequate (e.g. high frequency of swaps).

4. Process swapping 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. We describe these parameters in Section 4.1, then extract three interesting policies for further study in Section 4.2.

4.1. Policy parameters

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* (see Section 5 for a detailed discussion of payback distance).

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*.

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 hoarde fast processors.

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*.

4.2. 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 ***safe*** policy uses a low payback threshold (0.5 iterations), a high minimum 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 sleight 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).

5. Process swapping payback

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 [40, 34], 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:

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

The swap time in this equation is the time required to transfer process state to another processor over a communication link modelled 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.

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 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 several reasons: (i) often, we do not know how many iterations are

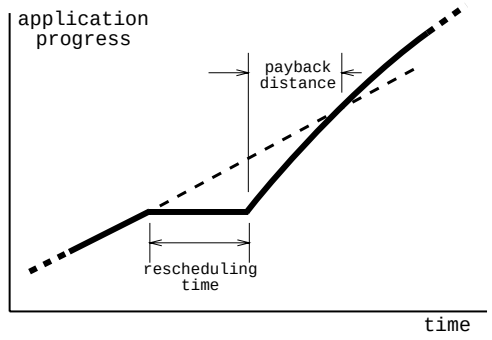

Figure 1. Payback distance.

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 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.

6. Simulation methodology

Since we target long-running applications on non-dedicated platforms that are by their very nature dynamic, it is infeasible to perform back-to-back experiments or to obtain reproducible results using real systems. Consequently, we use simulation and have implemented a simulator using the SIMGRID simulation toolkit [28]. Our simulator models the execution environment, the iterative application, and the different approaches for running the application, all of which are described in detail below.

Execution environment – We simulate a heterogeneous platform that consists of workstations connected via a 100-baseT ethernet LAN. More specifically, we simulate processors in the hundreds-of-megaflops performance range that 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.

CPU load – CPU load characterization is a challenging task [11] and no widely accepted model has been identified.

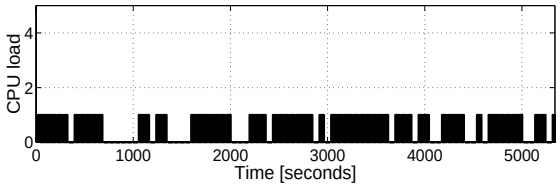

Figure 2. ON/OFF CPU load example.

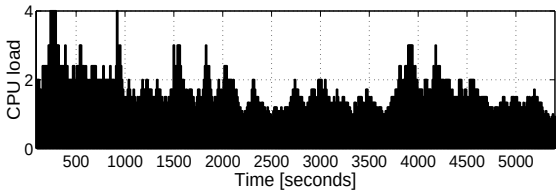

Figure 3. Hyperexponential CPU load example.

One approach is to "replay" traces of CPU load measurements obtained from monitoring infrastructures [42, 12]. When used in our previous work [6, 36] we found that 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. 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 "chaotic"). The trade-off is that the generated CPU loads may not be completely realistic. Nevertheless, we took this approach as it allows for a clearer understanding of simulation results.

We model CPU load in two ways. Our 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. Using this model we generate traces of CPU loads that take value 1 (ON, i.e. loaded with one competing compute-intensive process) or 0 (OFF, i.e. unloaded). We only simulate one competing process as it is typical of the environment that we target. More complex loads can be easily generated by aggregating ON/OFF sources. Figure 2 shows a typical CPU load trace generated using the ON/OFF source model (using $p = .3$, $q = .08$).

The second model used to simulate competing process load uses a degenerate hyperexponential distribution of process run times, as in [14]. Compared to the ON/OFF source

model, this model should better predict the heavy-tailed nature of the process lifetime distribution [29, 23]. As in the previous model, process arrival adheres to a uniform random distribution. Unlike in the ON/OFF model, we allow multiple simultaneous competing processes per processor. An example trace is shown in Figure 3.

The ON/OFF and hyperexponential CPU load models have limitations. There are other models, used by [29] for example, that even more accurately matche real CPU load. However, we claim our models are simple and conservative, and are sufficient to obtain the necessary first-order comparisons between the different algorithms and policies. We leave more complex models and the use of CPU load traces for future work.

Application – We simulate iterative applications with a range of 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 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. It is important to note that we are simulating one application in a shared environment; we are not analyzing arbitrary process migration.

Communication – We simulate a single, shared network link with latency α and bandwidth β. Thus messages compete for a fixed amount of communication bandwidth, and collisions delay message transmission.

Initial schedule – For all simulated application runs we must compute an initial application schedule. For load balancing we partition the work into unequal size chunks to balance processor iteration times. For other techniques we partition the application workload into equal size chunks. The initial schedule always uses the fastest performing processors at the time of application startup.

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. We simulate the overhead of starting up the application. We do not account for the overhead of doing the actual load balancing (i.e. exchanging data among processors) and assume that it is instantaneous. Consequently, the application execution times we obtain in our simulation for DLB are lower bounds on what could be obtained in practice.

Checkpoint/restart (CR) – The CR strategy is simulated as follows. 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, the application is checkpointed. We simulate the overhead of starting up the application. We assume that application state information is written to a central location. Upon application restart, the checkpoint is read by each process, and execution resumes. Our simulations account for the overhead of writing and reading the checkpoint. We 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).

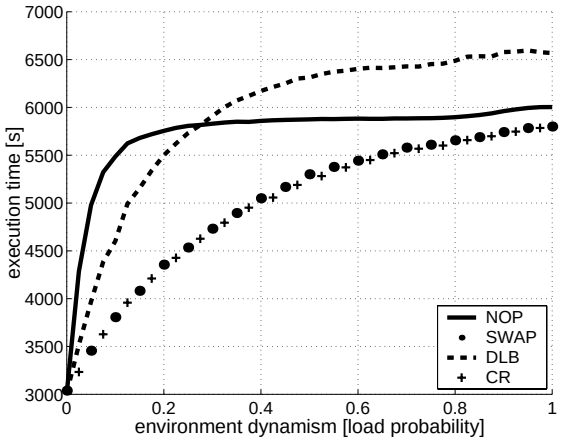

Figure 4. Execution time of various performance enhancing techniques across the full range of environment dynamism.

Process swapping – We simulate the application startup cost (including over-allocation). At each application iteration, if a swap occurs, we simulate the cost of transferring application state from an active to an inactive processor.

7. Simulation results

7.1. Evaluation of swapping vs. competing approaches

We examine four techniques: (a) do nothing (NOP); (b) process swapping using the greedy policy (SWAP); (c) dynamic load balancing (DLB); and (d) checkpoint/restart (CR). We will show that SWAP generally performs favorably as compared to the other techniques across a range of application characteristics and environment dynamism using the (more conservative) ON/OFF load model.

Swapping provides benefit in moderately dynamic environments. Figure 4 shows application execution time for the four techniques as a function of environment dynamism. In quiescent environments, shown on the left side of the figure, there is little difference between the techniques. Sim-

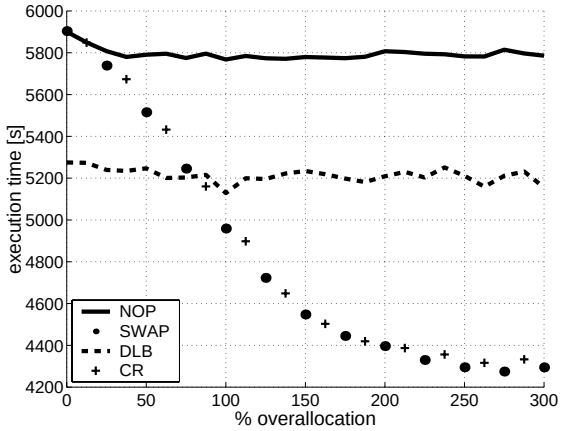

Figure 5. Execution time of various performance enhancing techniques across a range of over-allocation (8 active processes).

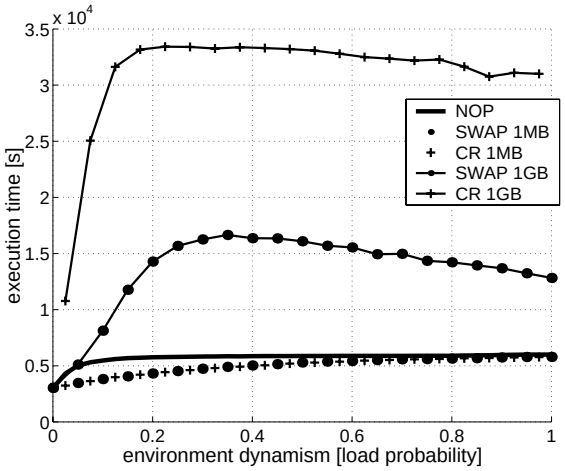

Figure 6. Execution time of various performance enhancing techniques for two selected process sizes.

ilarly, in highly dynamic environments, shown on the right side of the graph, the techniques tend to converge 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.

It is interesting to note that DLB does not perform very well in dynamic environments. When the environment becomes dynamic, 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.

Swapping performs better with more over-allocation. Figure 5 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 requires 100% over-allocation. 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.

The effectiveness of SWAP drops quickly as process size increases. Figure 6 shows the effect of process size on the performance techniques. The process size is the amount of information that needs to be saved during swapping or checkpoint. 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 harmful at a process size of 1GB. In the example shown, the swap time at 1 gigabyte is 120 seconds, while the application iteration time is 50 seconds.

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: process migration of any kind suffers as migration costs escalate.

SWAP has other limitations. For example, for very short-running applications, the additional cost of over-allocation causes SWAP to perform worse than other techniques. An over-allocation of 30 processors adds approximately 20 seconds to the application startup time.

7.2. Evaluation of three swapping policies

The greedy policy provides the largest performance boost. Figure 7 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 chaotic environments, almost keeping pace with the greedy policy. In more chaotic 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 chaotic environments the safe pol-

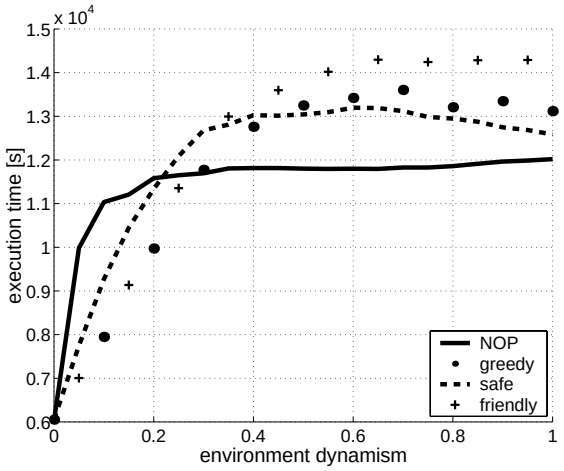

Figure 7. Execution time for various swapping policies across a range of environment dynamism.

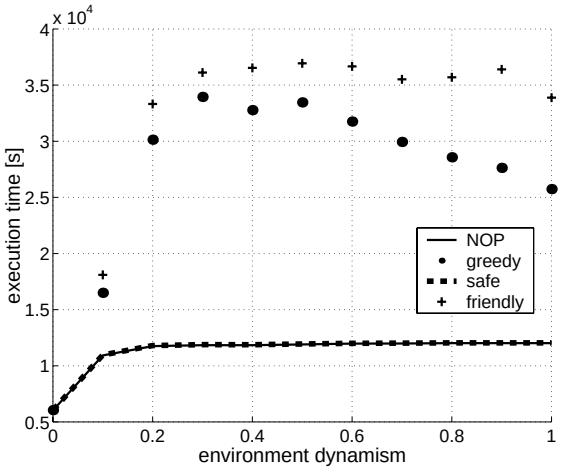

Figure 8. Execution time for various swapping policies where process size is large.

icy 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 8 shows application execution time for the various swapping policies when the 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.

7.3. Effect of CPU load distribution

Figure 9 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 the hyperexponential model increases the dynamism range over which swapping is beneficial.

7.4. Summary

Using an ON/OFF load model, swapping provides performance benefit on par with dynamic load balancing and checkpoint/restart. In moderately dynamic environments all three techniques outperform the NOP technique. In highly dynamic environments, where the load changes dramatically during each application iteration, all techniques can hurt performance. The conclusion is that process swapping is comparable to competing approaches in terms of performance while requiring virtually no implementation effort by the user (at least for iterative applications).

We observed that process swapping is viable for situations where one can over-allocate twice as many processors as the application actually requires. Also, swapping is viable for applications whose iteration times are at least as long as the time required to transfer process state from one processor to another.

The greedy swapping policy provides the greatest potential benefit, but also the greatest risk of poor performance. The safe swapping policy does not guarantee performance improvement, and while its potential benefit is lower than the greedy approach, it has a lower risk of poor performance. The friendly policy shows that you don't have to be greedy to receive some performance benefit.

These results were obtained using an ON/OFF CPU load model. Studies using a more heavy-tailed process lifetime distribution indicate that for these environments swapping (as well as DLB and CR) is even more beneficial.

8. Conclusion

In this paper we have presented a performance analysis of process swapping, a simple technique to enhance the performance of a parallel application in a time-shared environment consisting of a network of workstations. This analysis was done using a simulation environment, and complements the actual process swapping implementation described previously [35]. The regime within which swapping

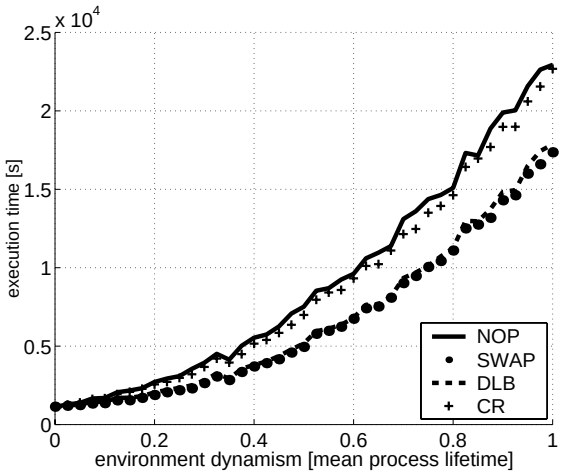

Figure 9. Execution time for various performance enhancing techniques under a hyperexponential load model.

is beneficial, relative to dynamic load balancing and checkpoint/restart, was discussed.

We developed three policies used to swap MPI processes, and presented a novel payback metric used to tune these policies. We showed 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 our policies are beneficial. As we work to improve the actual implementation, we can combine swapping with emerging middleware infrastructure that will extend the reach outside of the area of iterative applications. Currently, work is underway to integrate process swapping in the GrADS [26] architecture.

Our main conclusion in this paper is that 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. Furthermore, the simplicity of design allows programs to be easily retrofitted, making this technique accessible to new and existing applications.

References

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

[2] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[3] R. Batchu, J. Neelamegam, Z. Cui, 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] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Proceedings of Supercomputing 1996*, 1996.

[5] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Proceedings of SC'02*, 2002.

[6] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00)*, pages 349–363, May 2000.

[7] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, October 2000.

[8] A. Corporation. Avaki 2.0 concepts and architecture. Technical report, 2003.

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

[10] R. Diekmann, B. Monien, and R. Preis. Load balancing strategies for distributed memory machines. In H. Satz, F. Karsch, and B. Monien, editors, *Multiscale Phenomena and Their Simulation*, pages 255–266. World Scientific, 1997.

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

[12] P. Dinda, T. Gross, R. Karrer, B. Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The Architecture of the Remos System. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 2001)*, August 2001.

[13] The microsoft .NET framework. `http://www.microsoft.com/net/basics/whatis.asp`.

[14] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. *Conf. on Measurement & Modelling of Comp. Syst., (ACM SIGMETRICS)*, pages 63–72, May 1988.

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

[16] The Entropia Home Page. `http://www.entropia.com`.

[17] G. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the Euro PVM/MPI User's Group, Berlin, Germany*, pages 346–353, 2000.

[18] G. Fedak, C. Germain, V. Nri, and F. Cappello. XtremWeb : A Generic Global Computing System. In *Proceedings of the Workshop on Global Computing on Personal Devices*, May 2001.

[19] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-performance Distributed Computations. In *Proceedings of the 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.

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

[21] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.

[22] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Supercomputing Applications*, 22(6):789–828, 1996.

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

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

[25] The United Devices Home Page. `http://www.ud.com`.

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

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

[28] A. Legrand, L. Marchal, and H. Casanova. 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.

[29] W. E. Leland and T. J. Ott. Load-balancing heuristics and process behavior. *ACM SIGMETRICS*, pages 54–69, May 1986.

[30] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)*, 1988.

[31] Platform computing. `http://www.platform.com`.

[32] J. Protic, M. Tomaevic, and V. Milutinovic. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press and John Wiley & Sons, Inc., July 1997.

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

[34] G. Shao, R. Wolski, and F. Berman. Modeling the cost of redistribution in scheduling. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[35] O. Sievert and H. Casanova. MPI process swapping: Architecture and experimental verification. Technical Report CS2003-0735, Dept. of Computer Science and Engineering, University of California, San Diego, 2003.

[36] S. Smallen, H. Casanova, and F. Berman. Applying Scheduling and Tuning to On-line Parallel Tomography. In *Proceedings of Supercomputing 2001 (SC'01)*, November 2001.

[37] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1998.

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

[39] K. Toonen and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 2003. to appear.

[40] S. Vadhiyar and J. Dongarra. A Metascheduler for the Grid. In *Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing*, July 2002. To appear.

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

[42] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

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

[44] W. Zhu and C. Steketee. An experimental study of load balancing on Amoeba. In *First Aizu International Symposium on Parallel Algorithms/ Architecture Synthesis*, pages 220–226, 1995.