Load Balancing in the Bulk-Synchronous-Parallel Setting using Process Migrations*

Olaf Bonorden

Heinz Nixdorf Institute, Computer Science Department,
University of Paderborn, 33095 Paderborn, Germany,
bono@uni-paderborn.de

Abstract

The Paderborn University BSP (PUB) library is a powerful C library that supports the development of bulk synchronous parallel programs for various parallel machines. To utilize idle times on workstations for parallel computations, we implement virtual processors using processes. These processes can be migrated to other hosts, when the load of the machines changes. In this paper we describe the implementation for a Linux workstation cluster. We focus on process migration and show first benchmarking results.

1. Introduction

Motivation. Nowadays there are workstations in nearly every office spending most of their time on the idle task waiting for user input. There exist several approaches to use this computation power. First, there are programs for solving specialized problems, SEIT@home (search for extraterrestrial intelligence, [26]) and distributed.net (e. g., breaking cryptographic keys, [12]) are two of the most famous ones. These systems scan a large data space by dividing it into small pieces of data, sending them as jobs to the clients, and waiting for the results. There is no communication during the calculation, or the communication is combined with the job data and sent through the central server as in the BSP implementation Bayanihan [24]. These loosely coupled systems can efficiently be executed on PCs connected by the Internet.

Achieving fault tolerance is very easy: If a client does not return the result, a timeout occurs and the job is assigned to another client. Although only a small, fixed number of problems can be solved, these systems are very powerful in terms of computation power because of the huge number of clients taking part.

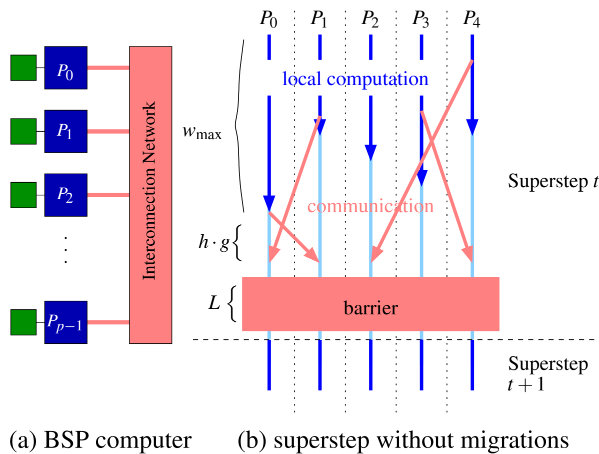
Second, there are job scheduling and migration systems. Here you can start jobs, either directly or using a batch system. These jobs are executed on idle nodes and may migrate to other nodes if the load in the system changes. Condor [20] is one example for such a system implemented in user space, MOSIX [3] is a module for the Linux kernel which allows migrations of single processes. Kerrighed [21] goes beyond this and provides a single system image of a workstation cluster. It supports migration of processes and also communication via shared memory.

Cao, Li, and Guo have implemented process migration for MPI applications based on coordinated checkpoints [9]. They use an MPI checkpointing system for the LAM MPI implementation [23] based on Berkeley Lab's Linux Checkpoint/Restart [14], a kernel level checkpointing system. As MPI stores the location of the processes, they have to modify the checkpoints before restarting.

An implementation for migration of single threads, even in a heterogenous network with different CPU types, is proposed by Dimitrov and Rego in [11]. They extend the C language and implement a preprocessor which automatically inserts code for keeping track of all local data. Additionally, the state in each function, i. e., the actual execution point, is stored and a switch statement, which branches to the right position depending on the state if a thread is recreated, is inserted at the beginning of each function. These changes lead to an overhead of 61 % in their example, mainly due to the indirect addressing of local variables. A similar approach for Java is JavaGo [22], used in the BSP web computing library PUBWCL [7].

Most of these systems support only independent jobs (Condor can schedule parallel jobs using MPI or PVM, but these jobs do not take part in load balancing and migration) or affect parts of the operation system kernel like Kerrighed, thus the user do not only need administrative access to the

*Partially supported by DFG SFB 376 "Massively Parallel Computation" and by EU FET Integrated Project AEOLUS, IST-2004-15964
1-4244-0910-1/07/$20.00 ©2007 IEEE.



(a) BSP computer (b) superstep without migrations

Figure 1. BSP model

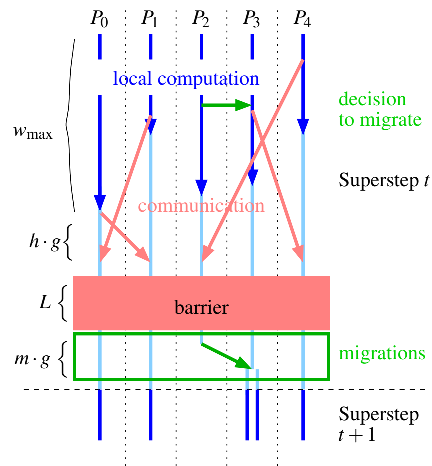


Figure 2. BSP model: extended by migrations

machines, they furthermore have to install a Linux kernel with special patches, which has some drawbacks (usable kernel versions are limited, e. g., not up-to-date).

There exists some work about checkpointing threaded processes [10], which is useful to migrate parallel programs for symmetric multiprocessor (SMP) machines, but as far as we know there was no system for distributed parallel applications in user space.

Our contribution. We have extended the PUB library [8], a C library for writing parallel programs for the Bulk Synchronous Parallel Model [27], using virtual processors executed on idle computers. These virtual processors can migrate through the network to adapt to load changes. Our implementation runs in user space and does not need to be installed on the used computers.

Organization of paper. The paper is organized as follows: In Section 3 we describe the procedure of migration. In Section 4 we present different strategies for load balancing. In Section 5 we show performance results from some test programs.

2. The Bulk Synchronous Parallel Model with migrations

In this section we describe how we extend the *Bulk Synchronous Parallel* (BSP) Model [27] by migrations. We start with a short introduction of the model (see [4] for a textbook with a more detailed view and example programs). The BSP model was introduced by Leslie G. Valiant as a bridge between the hardware and the software to simplify the development of parallel algorithms. It gives the developer an abstract view of the technical structure and the communication features of the hardware to use (e. g., a parallel computer, a cluster of workstations, or a set of PCs interconnected by the internet).

A *BSP computer* is defined as a set of processors with local memory, interconnected by a communication mechanism (e. g., a network or shared memory) capable of point-to-point communication, and a barrier synchronization mechanism (cf. Figure 1 (a)). A *BSP program* consists of a set of *BSP processes* and a sequence of *supersteps*—time intervals bounded by the barrier synchronization. Within a superstep each process performs local computations and sends messages to other processes; afterwards it indicates by calling the sync method that it is ready for the barrier synchronization. When all processes have invoked the sync method and all messages are delivered, the next superstep begins. Then the messages sent during the previous superstep can be accessed by its recipients. Figure 1 (b)) illustrates this.

In the classical BSP model these BSP processes are executed on different processors, e. g., hosts in a workstation cluster. To deal with the problem that these hosts are used by others, i. e., they are not always available for our tasks, we migrate the BSP processes (also known as virtual processors) to suitable hosts. In the normal setting we assume that we will always get a little amount of computation power (like Linux processes started with nice-priority), so we are able to do some calculations and leave a computer in a regular way but there is also a fault tolerant version that uses checkpoints to restart aborted task.

Due to some technical reasons (cf. Section 3.2) migrations are performed during the synchronization only, thus we insert a migration phase after each synchronization (cf. Figure 2). The length of this phase is $m \cdot g$, whereas m is the maximal sum of the sizes of all processes a processors

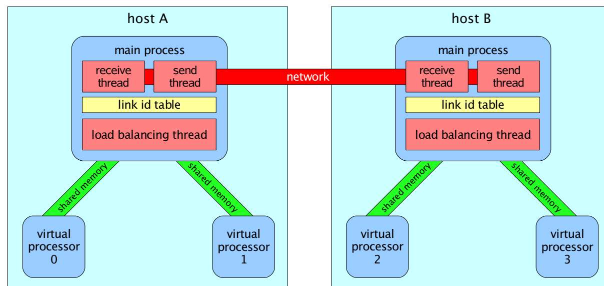


Figure 3. Structure of PUB with migratable processes

will send or receive. To take migration latencies, caused by the network and the migration of the memory used by PUB, into account, we count each process memory smaller than a value m_0 as m_0 (similar as hiding the latency for a h relation in the standard BSP model, cf. [27])

3. Migration

Using PUB you can start virtual processors with the function `bsp_migrate_dup` [5]. This will start one new process for each virtual processor. Figure 3 gives an overview of the processes, their threads and the connections. The main process consists of three additional threads: The *receive thread* reads all messages from the network and sends them to the right destination by either calling the proper message-handler or putting the message into a shared memory queue to a virtual processor. The *send thread* sends all messages from a shared memory send queue to the network. The *load balancing thread* gathers information about the load of the system and decides when to migrate a virtual processor. We have implemented several algorithms for load balancing (cf. Section 4). The virtual processors are connected to the main process by two shared memory message queues, one for each direction.

The link-table is the routing table. For each virtual processor of the system it stores the actual execution host and the link id (used to distinguish the processes). This table does not have to be up-to-date on all hosts, for example if a process was migrated, but then the destination host stored in the table knows the new location and will forward the message to it. Thus a message sent to the location stored in the table will reach its destination, although it might take some additional hops.

3.1 Procedure of the migration

When the load balancer decides to migrate one process to another host, the following steps are executed:

1. The load balancer writes the destination host id to a migration queue.

2. The first process which enters the synchronization dequeues this id. It sends a message to the main process.

3. The receive thread reads this message, marks the virtual processor as *gone* and sends an acknowledgment to it. From now on it will not send any messages to the process. All messages for this destination will be stored in a buffer in the main process.

4. The process to migrate reads this acknowledgment, opens a TCP/IP server socket and sends a message to the destination host.

5. The destination host creates two shared memory queues and a new child process.

6. This new process connects directly via TCP/IP to the old virtual processor.

7. The context of the process is transfered from the old host to the new destination. The details of this step are described in Section 3.2. After this the old process terminates.

8. The new process sends its new link id to the main process on the original host.

9. This host updates its link table and sends all messages arrived during the migration to the new destination. If any further messages arrive for this virtual processor, they will be forwarded to the new location. Optionally the migration can be broadcasted to reduce the number of forwarded messages.

3.2 Migration of Linux Processes

This section deals with the technical aspects of migrating a Linux process from one computer to another. We consider the parts of a process and look at its interaction with the operating system kernel. The *configuration* of a process consists of a user space and a kernel space part. As we do not extend the kernel, we cannot access the data in the kernel space unless we use standard system functions, in particular we are not allowed to access the kernel stack.

In the following we have a closer look to the migratable parts in the user space, and the dependencies to the kernel, namely system calls and thread local storage.

Linux system calls. The access to the operating system is done by *syscalls*. A syscall is like a normal function call, but additionally the following steps are performed: the CPU switches to *supervisor mode* (also known as *kernel mode* or *ring 0*) and sets the stack pointer to the kernel stack. This

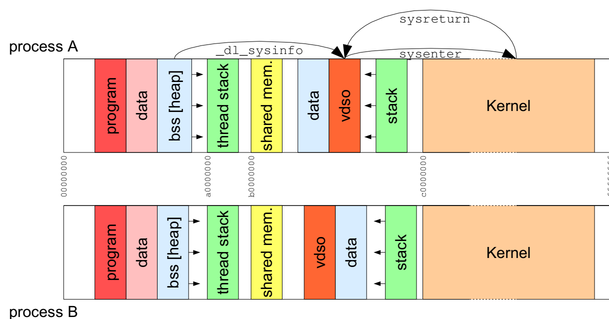


Figure 4. Virtual address space

is needed to allow the kernel to use all instructions and deal with stack overflows of the user stack. There are two different ways for a system call in Linux: Traditionally Linux uses a software interrupt, i. e., the number of the system function and its arguments are stored into processors registers, and an interrupt is generated by the instruction `int 0x80`. This interrupt call is very slow on Intel Pentium IV processors, the time for a system call is much higher on a 2 GHz Pentium IV than on a 850 MHz Pentium III [15] (see [25] for a detailed view on measuring Linux kernel execution times). Intel has introduced a more efficient mechanism for privilege switches, the `sysenter` instruction, available since the Pentium Pro processors. But there was a hardware bug in early CPUs, so it took a long time until operating systems started using sysenter. Linux supports sysenter since version 2.5.53 from December, 2002.

To provide a portable interface to different CPUs, Linux uses a technique called *vsyscall*. The kernel maps a page of memory (4096 bytes) into the address space of user programs. This page contains the best implementation for system calls, thus programs just do a simple subroutine call to this page. The page is called *VDSO*, its address can be found in the file maps in the proc-filesystem (cf. Table 1). The address is also noted in the ELF header of dynamically linked programs and can be viewed with the command `ldd`, here the page looks like a dynamic library called linux-gate.so.

Before Linux version 2.6.18 (September, 2006) the address of the page was fixed to 0xffffe000. This is the last usable page in the 32 bit address space; the page at 0xfffff000 is not used because illegal memory accesses (pointer underruns) should lead to a page fault. Recent Linux versions use a random address[19] to complicate some kind of buffer overflow attacks. The C library stores the address in a variable, `_dl_sysinfo`, the kernel also stores the address because sysenter does not save a return address, so the `sysreturn` instruction used to return to the userspace needs this address as an argument, i. e., no user process can change the address of the VDSO page, because the kernel will always jump back to the old address.

If we migrate a process, we cannot guarantee that the source and the destination use the same address, but we have to keep the old VDSO page. So we do not migrate this page and adapt the address in the C library, such that it uses the old page. If this is not possible because of user data is mapped to the same address (cf. Figure 4), we free the VDSO page, and change the `_dl_sysinfo` address to our implementation for system calls using the old interrupt method.

Randomization of the VDSO page can be disabled with the sysctl command (sysctl -w kernel.randomize_va_space=0) or directly using the proc filesystem (echo 0 > /proc/sys/kernel/randomize_va_space).

Thread local storage. Although virtual processors are not allowed to use threads, we have to deal with *thread local storage* (TLS), because the ancestor of our processes, the main process, uses threads, i. e., all the processes in PUB are forked of a threaded process. TLS is used for global variables (accessible from all functions), that have different values for each thread. The C library uses this feature among other things for the errno variable: errno is a variable that is set to the error code of operations. Instead of using the function return value, the POSIX IEEE Std 1003.1 [18] uses a global variable.

As all threads share the same address space, these variables must have different addresses for different threads, so the linker cannot insert a normal memory reference here. The Linux implementation uses a Thread Control Block (TCB), which stores all the information about TLS data. A pointer for the actual thread is accessible at offset 0 of the segment stored in the GS segment register (see [13] for details). This segment is mapped to a part of the stack of the actual running thread. We create new processes for the migration by forking the receive thread, i. e., although we do not use threads inside virtual processors and have our own stack, we cannot free the stack of the receive thread, and furthermore we have to guaranty that the stack is at the same address on all hosts. Thus we allocate our own stack at a fixed address and assign it to the receive thread with the function `pthread_attr_setstack` from the POSIX thread library.

Migratable resources. When we migrate a process, we transfer all its resources to the new destination, namely the state of the processor, the used memory, and some other data stored inside the kernel, e. g., open files.

Processor state: The processor state contains all data stored in the processor, e. g., the registers. These data have to be migrated. We save the registers in the data segment (we use the word “segment” for parts of memory, not to be mixed up with segments in the context of Intel processors) of our process. Litzkow et. al. suggest to use a signal

Table 1. Example of memory a map read from maps of the proc-filesystem

virtual address	perm	offset			mapped file
08048000-08108000	r-xp	00000000	03:05	4432424	/tmp/migtest
08108000-08109000	rw-p	000bf000	03:05	4432424	/tmp/migtest
08109000-081c9000	rw-p	08109000	00:00	0	[heap]
a0000000-a0100000	rwxp	a0000000	00:00	0	
b0000000-b2000000	rw-s	00000000	00:07	305790982	/SYSV00000000
b7e00000-b7e21000	rw-p	b7e00000	00:00	0	
b7e21000-b7f00000	---p	b7e21000	00:00	0	
b7ffc000-b7ffd000	r-xp	b7ffc000	00:00	0	[vdso]
bffb7000-bffcd000	rw-p	bffb7000	00:00	0	[stack]

handler for the migration [20]. Inside a signal handler one can use all registers freely, i. e., the operating system has to store and restore all registers by itself and the code can become more portable. Older versions of our implementation use this technique, too. But this is not compatible with the handling of signals in recent Linux kernels. If a signal is sent to a process, the kernel saves some data onto the kernel stack, and writes a system call to the kernel function `sys_sigret` onto the user stack. So when the signal handler returns, the kernel function restores the data from the kernel stack, but we cannot modify or migrate this data on the kernel stack. See [2] for more details.

Memory: The virtual address space of a Linux process contains different parts which must be handled separately. Table 1 shows a memory mapping of one virtual processor. This mapping can be read from the maps file of the Linux proc filesystem. First there is the program code segment, which is marked as read-only and is the execution file mapped into the address space. We do not need to migrate this because the program is known to all the nodes. Second, there are two data segments: One is for the initialized data; it is mapped from the execution file, but only copied on first write access. Next there is the real data segment; this data segment grows when the application needs more memory and calls `malloc`. The size of this segment can be adjusted directly by `brk`. If the user allocates large blocks of data, the system creates new segments. We have to migrate the data segments and all these new segments.

The last segment type is the stack. It used to start at address `PAGE_OFFSET` defined in `page.h` of the Linux kernel, normally 0xc0000000 (32 bit Linux splits the 4 GB address space into 3 GB user space from 0x00000000–0xbfffffff and 1 GB reserved for the kernel), and grows downwards. For security reasons Linux uses address space randomization [1] since kernel version 2.6.12, i. e., the stack starts at a random address in the 64 KB area below `PAGE_OFFSET`, so we need to find the stack and restore it at the same address on the destination node. During the migration we use a temporary stack inside the shared memory.

Data stored in the Linux kernel: There is some data stored in the kernel, e. g., information about open files, used semaphores, etc. We have added support for regular files which exists on all the hosts, e. g., on a shared file system like the network file system (NFS). We gather information about all open files and try to reopen them on the destination host.

All other information stored in the kernel will be lost during migration, so you must not use semaphores, network connections etc. during the migration (i. e., during the synchronization of BSP). If such resources are needed over more than one superstep, the migration of that virtual processor can temporarily be disabled.

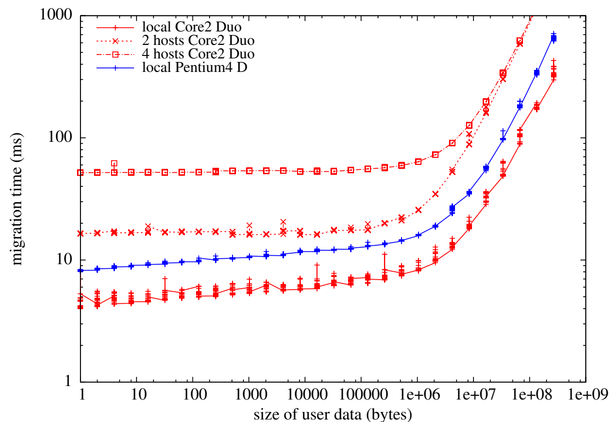


Figure 5. Time needed for migrations

3.3 Performance

To measure the performance of migrations we disable load balancing and force one process to migrate. Due to different local clocks, we send the process via all hosts in the network and back to the original location, i. e., like a ping-pong test for two nodes and a round trip for more hosts.

For our benchmark we use a Linux cluster at our institute (pool for students). We choose up to 4 idle PCs with Intel Core2 Duo 2.6 GHz processors connected by a Cisco Gigabit Ethernet Switch. To evaluate the influence of the local system (CPU, mainboard, memory), we also used one PC with an Intel Pentium D 3 GHz CPU.

The benchmark program allocates a block of memory and migrates one virtual processor from host 0 to $1,2,\ldots p-1$ and back to host 0. These migrations are repeated 20 times. Figure 5 shows the mean time for one migration. The bandwith grows up to 111 MB/s for large data sets, which is nearly optimal for Gigabit Ethernet. Without network we achieve 850 MB/s (Core2Duo, 412 MB/s Pentium4 D) for migrations whereas a plain memory copy obtains 3.123 GB/s (resp. 1.926 GB/s).

4. Load balancing

Load balancing consists of three decisions: 1. Do we need to migrate a virtual processor? 2. To which destination node? 3. Which virtual processor? The library contains some loadbalancing strategies, but it is also possible to add new strategies for the first and second question. The virtual processor for a migration is always chosen by PUB: When the load balancer decides to migrate a virtual processor, it writes the destination to a queue and PUB will migrate the first virtual processor that looks into this queue, i. e., the first executed on this computer that reaches the synchronization.

In the following we describe the implemented strategies: one centralized one with one node gathering the load of all nodes and making all migration decisions, and some distributed strategies without global knowledge.

The global strategy: All nodes send information about their CPU-Power and their actual load to one master node. This node calculates the least ($P_{\min}$) and the most ($P_{\max}$) loaded node and migrates one virtual processor from $P_{\max}$ to $P_{\min}$ if necessary.

Simple distributed strategy: Every node asks a constant number c of randomly chosen other nodes for their load. If the minimal load of all the c nodes is smaller than the own load minus a constant $d \geq 1$, one process is migrated to the least loaded node. The waiting time between these load balancing messages is chosen at random to minimize the probability that two nodes choose the same destination at exactly the same time. If a node is chosen, it will be notified by a message and increases its load value by one to indicate that the actual load will increase soon.

Conservative distributed strategy: As in the simple strategy, c randomly chosen nodes are asked for their load l_i and for their computation power c_i. Next we compute the expected computation power per process if we would migrate one process to the node i for $i = 1, \ldots, c$: $\frac{c_i}{l_i+1}$. If the maximal ratio of all c nodes is greater than the local ratio, the program would run faster after the migration. But this strategy only migrates a process, if the source is the most loaded node of the $c+1$ ones, i. e., that $\frac{c_i}{l_i}$ is greater than the own ratio. This conservative strategy gives other more loaded nodes the chance to migrate and thus leads to less migrations than the simple strategy.

Global prediction strategy: Each node has an array of the loads of all other nodes but these entries are not up to date all the time, but each node sends its load periodically to k uniformly at random chosen nodes in the network. In [16] Hill et al. analyse this process using Marcov chains: the mean age of an entry in the array is $O(p/k)$.

All these strategies use the one minute load average increased by one for each expected task. This is needed because migrations are executed at the synchronization only, and an unloaded machine should not be chosen by many other nodes without noticing that other processes will migrate to the same destination.

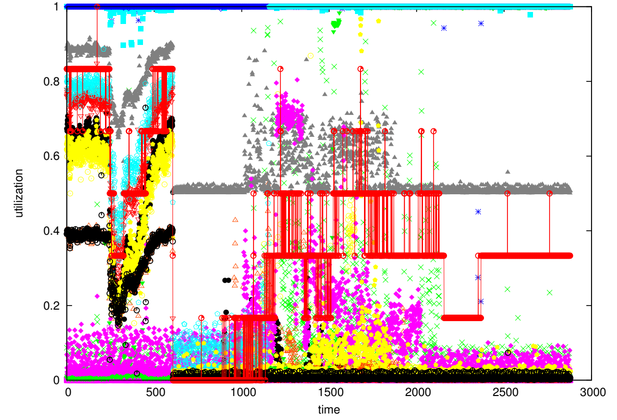


Figure 6. Simulated load profile

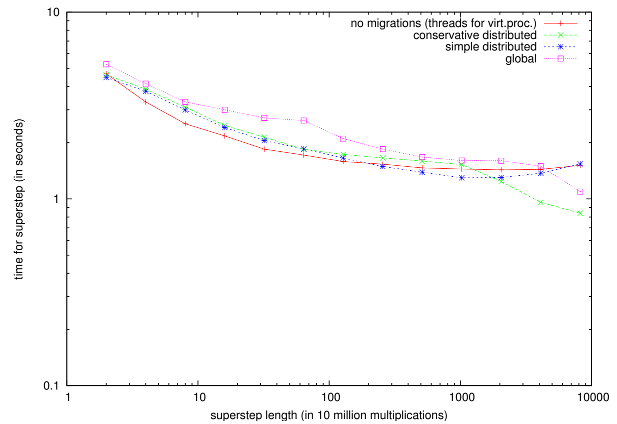
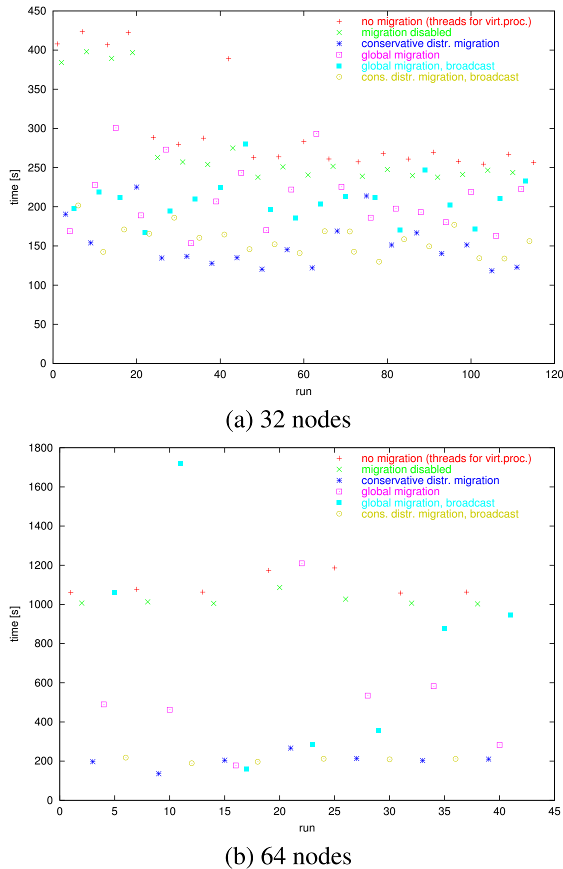


Figure 7. Computation benchmark

5. Experiments

Hardware. For our experiments we used the PCs in our office and in the students pools. There are computers of different speed: Intel Pentium III with 933 MHz and one or two processors (each processor 1867 bogomips[1]), Intel Pentium IV with 1.7 GHz (3395 bogomips), and 3.0 GHz (5985 bogomips). During the first experiment the computers were partially used by their owners, for the second experiment we chose unused computers and simulated the external workload by generating artificial load on the computers with a higher scheduling priority than our jobs. We used the same

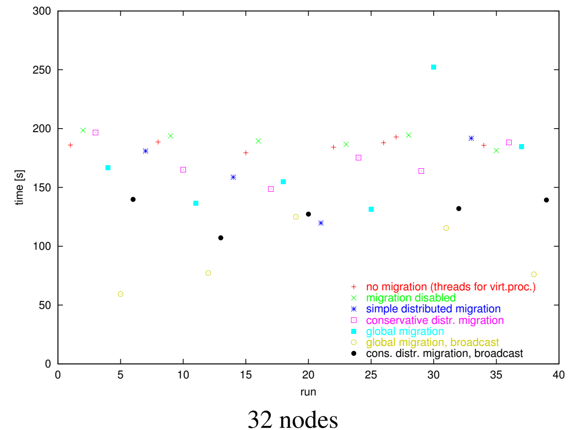
[1]bogomips is a simple measuring unit for the speed of CPUs provided by the Linux kernel. We show this unit here because our loadbalancing algorithms use it for approximating the power of the processors.



(a) 32 nodes

(b) 64 nodes

Figure 8. execution time of 3-SAT with different load balancing strategies



32 nodes

Figure 9. execution time of total exchange with different load balancing strategies

load profiles as in [6]. One load profile is shown in Figure 6. Each entry represents the load of one host, the red curve aggregates the total utilization of the workstations.

Applications. Our first example application is a BSP implementation of a divide and conquer algorithm for the 3-SAT problem ([17], pp. 169–173). In the first step processor 0 executes the algorithm sequentially until $c \cdot p$ subproblems are created for some constant c (we use $c = 4$ in our experiments). These branches are distributed evenly among the processors. Then all processors try to find a solution sequentially for their subproblems. Every 10000 divide-steps the processors enter a new superstep. If a solution is found, it is broadcasted and the computation stops at the end of the actual superstep. On the one hand these synchronizations are necessary to detect whether a solution is found, on the other hand they are needed because migrations only occur during synchronizations. The second example is a simple all-to-all-communication. Since in the 3-SAT solver the calculations dominate the communication, its benefit from migration is natural. In an all-to-all comunication the network is the bottle-neck, so we have to minimize the number of forwarded messages to reduce the network congestion. The last benchmark is a simple computation test: each host calculates an increasing number of multiplications (from 10^7 up to 10^{11}).

Results. The running time for 3-SAT on 32 and 64 processors is shown in Figure 8. The 3-SAT solver benefits from migrations in our office cluster as expected. There were some nodes in the system with some load of other users greater than 1, so it was reasonable to migrate the processes from such nodes to other idle nodes. The cost of the migrations itself is small due to the small memory contexts, and updating the routing tables is not of big importance.

Figure 9 shows the results for the communication benchmark all-to-all on 32 nodes. Here, updating the routing tables by broadcasting migrations is necessary.

Figure 7 shows that migrations are profitable for large supersteps. The total execution time of our benchmark decreases from 8 hours to 4 hours with migrations.

6. Concluding Remarks

In this paper we have presented a system to use idle times in workstation clusters for parallel computations. Process migrations are a powerful technique to cope with changing availability of the machines. First experiments show the benefit using migrations. We are working on more experiments with different load profiles to learn more about the behavior of our load balancing algorithms in practice. Furthermore we are implementing different checkpointing

strategies (When to create checkpoints? At which computers should the data be stored?) for fault tolerance.

One can also extend the library to be able to use new unloaded computers, e. g., when a faulty system is working again or new computers are available. For some applications it seems to be reasonable that the application itself can control the number of virtual processors to adapt to changing environments; currently the number of virtual processors stays constant or increases, whereas the number of used computers can only go down.

References

[1] Address space randomization in 2.6. http://lwn.net/Articles/121845/, 2005. LWN.net.

[2] M. Bar. The Linux Signals Handling Model. *Linux Journal*, 2000. http://www.linuxjournal.com/article/3985.

[3] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, March 1998.

[4] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, Oxford, UK, Mar. 2004.

[5] O. Bonorden and J. Gehweiler. PUB-Library - User Guide and Function Reference. Available at http://www.upb.de/~pub, Jan. 2007.

[6] O. Bonorden, J. Gehweiler, and F. Meyer auf der Heide. Load balancing strategies in a web computing environment. In *Proceeedings of International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 839–846, Poznan, Poland, September 2005.

[7] O. Bonorden, J. Gehweiler, and F. Meyer auf der Heide. A web computing environment for parallel algorithms in Java. *Scalable Computing: Practice and Experience*, 7(2):1–14, June 2006.

[8] O. Bonorden, B. H. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[9] J. Cao, Y. Li, and M. Guo. Process migration for MPI applications based on coordinated checkpoint. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems*, volume 1, pages 306–312, 2005.

[10] W. R. Dieter and J. E. Lumpp, Jr. User-level checkpointing for LinuxThreads programs. In *Proceedings of the 2001 USENIX Technical Conference*, http://www.engr.uky.edu/~dieter/publications.html, June 2001.

[11] B. Dimitrov and V. Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):459–469, 1998.

[12] distributed.net project homepage. http://www.distributed.net.

[13] U. Drepper. ELF Handling For Thread-Local Storage. http://people.redhat.com/drepper/tls.pdf, Dec. 2005. Version 0.20.

[14] J. Duell. The design and implementation of Berkeley Lab's Linux checkpoint/restart. Technical Report LBNL-54941, Apr. 2005.

[15] M. Hayward. Intel P6 vs P7 system call performance. http://lkml.org/lkml/2002/12/9/13, 2002. LWN.net.

[16] J. M. Hill, S. R. Donaldson, and T. Lanfear. Process migration and fault tolerance of BSPlib programs running on networks of workstations. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, pages 80–91, London, UK, 1998. Springer-Verlag.

[17] J. Hromkoviĉ and W. M. Oliva. *Algorithmics for Hard Problems*. Springer-Verlag New York, Inc., 2001.

[18] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. IEEE, New York, NY, USA, 1996.

[19] Kernel Newbies: Linux 2.6.18. http://kernelnewbies.org/Linux_2_6_18.

[20] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, April 1997.

[21] C. Morin, P. Gallard, R. Lottiaux, and G. Vallée. Towards an efficient single system image cluster operating system. *Future Generation Computer Systems*, 20(4):505–521, 2004.

[22] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in Java. Technical report, University of Tokyo, 2000.

[23] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[24] L. F. G. Sarmenta. An adaptive, fault-tolerant implementation of BSP for JAVA-based volunteer computing systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 763–780, London, UK, 1999. Springer-Verlag.

[25] S. Schneider and R. Baumgartl. Unintrusively measuring Linux kernel execution times. In *7th RTL Workshop*, pages 1–5, Nov. 2005.

[26] SETI@home project homepage. http://setiathome.ssl.berkeley.edu.

[27] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

Biography

Olaf Bonorden is a research assistant at the Algorithms and Complexity research group of Prof. Dr. Meyer auf der Heide at the Heinz Nixdorf Institute and the Computer Science Department, University of Paderborn, Germany. He received his diploma degree at the University of Paderborn in 2002. Main research topics are parallel models, especially the Bulk Synchronous Parallel Model, for all kind of parallel architectures—from a parallel system on chip to web computing. Since 1999 he also takes part in the development of the Paderborn University BSP (PUB) library.