User-level Process towards Exascale Systems

Akio Shimada[1,a)] Atsushi Hori[1,b)] Yutaka Ishikawa[1,c)] Pavan Balaji[2,d)]

Abstract: The process oversubscription, which binds multiple parallel processes to one CPU core, can hide the communication latency and reduce CPU idle time. However, the lightweight OS kernels for Exascale systems may no longer support OS task scheduling. Without OS task scheduling, only one parallel process per CPU core is allowed, and then the process oversubscription is impossible. Even if the OS task scheduling is supported, the overhead of the context switch between parallel processes decreases the application performance and prevents the process oversubscription from being utilized. By assigning a role of a parallel process to a user-level thread, high-performance process oversubscription can be achieved without OS task scheduling. However, the process oversubscription utilizing the user-level thread dramatically changes the programming model of the parallel application. In this paper, we propose *user-level process*. The user-level process is a "process", which can be scheduled in the user-space. The user-level process has the beneficial features of the user-level thread. Meanwhile, it has its own program code and data like a traditional process. By assigning a role of a parallel process to an user-level process, high-performance process oversubscription can be achieved without OS task scheduling. Moreover, the process oversubscription utilizing the user-level process does not change the programming model of the parallel application.

1. Introduction

Parallel processes running on a HPC cluster communicate with each other to exchange data for processing parallel computation. If a parallel process is blocked in order to wait for completion of a communication, a CPU core, to which it is bound, can be idle state during the communication. Then, the communication latency between parallel processes results in inefficient use of CPU resources. Network systems of HPC clusters will be larger and more complicated towards Exascale systems. Therefore, the communication latency and the disbenefit due to it continue to be an important problem.

One of the techniques for reducing CPU idle time is to use non-blocking communication. The non-blocking communication enables a parallel process to overlap communication with computation. As a result, the communication latency can be hidden, and efficient use of CPU resources can be enhanced. However, there are not always computations, which can be overlapped with communications. In such situations, a parallel process has to wait for completion of a communication after all, even if the no-blocking communication is used. Further technique for reducing CPU idle time is to use process oversubscription, which binds multiple processes to one CPU core and makes them run as a parallel process. With the process oversubscription, if a parallel process is blocked in order to wait for completion of a communication, any other ready parallel process bound to the same CPU core can preempt it. As a result, communication latency can be hidden and CPU idle time can be reduced.

Although the process oversubscription is beneficial, some problems prevent HPC system users from utilizing it. The lightweight OS kernels, such as Mckernel[4][12] and Argo[1], may no long support OS task scheduling, because it is a resource consuming and noisy operation. Without OS task scheduling, only one parallel process per CPU core is allowed, and then the process oversubscription is impossible. Even if the OS task scheduling is supported, the overhead, which arises at the context switch between parallel processes, prevents the process oversubscription from being utilized. Parallel processes must be scheduled by the OS kernel. Thus, the overhead of jumping into the kernel-context arises at every context switch. This overhead spoils the performance benefit of the process oversubscription in some cases[5].

Assigning a role of a parallel process to a user-level thread solves above problems. In this scheme, the process oversubscription is achieved by invoking multiple user-level threads within a process on behalf of binding multiple parallel processes to one CPU core. If a user-level thread, which plays a role of a parallel process, is blocked, any other ready user-level thread within the same process preempts it. Then, CPU idle time is reduced. The context switch between user-level threads within the same process can be done in user-space, which means that task scheduling of the user-level threads within the same process can be operated in user-space. Therefore, OS task scheduling is not necessary for the process oversubscription when assigning a role of a parallel process to a user-level thread. Moreover, the overhead of jumping into kernel-context does not arise at the context switch between user-level threads within the same process. Therefore,

[1] RIKEN AICS
[2] Argonne National Laboratory
[a)] a-shimada@riken.jp
[b)] ahori@riken.jp
[c)] yutaka.ishikawa@riken.jp
[d)] balaji@anl.gov

© 1992 Information Processing Society of Japan

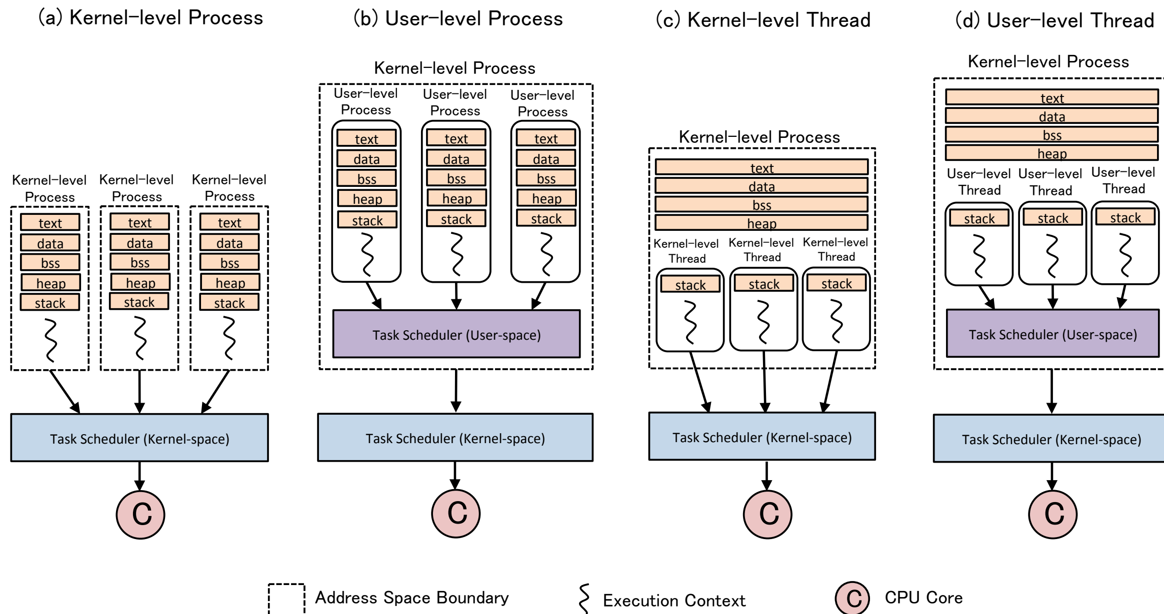

Fig. 1 Semantics views of kernel-level process, user-level process, kernel-level thread and user-level thread

high-performance process oversubscription can be achieved. Currently, a variety of high-performance user-level thread libraries are proposed[13][14][8][2].

However, this scheme dramatically changes the programming model of the parallel application. In general, programmers implement parallel applications on the assumption that a parallel process has its own program code (text) and data (data, bss and heap). Nevertheless, when assigning a role of a parallel process to a user-level threads, a user-level thread playing a role of a parallel process shares the same program code and data with others. When using this scheme, existing programs cannot be executed without any modifications, and programmers must implement a new application with unfamiliar manner.

In this paper, we propose *user-level process*. The user-level process is a ”process”, which can be scheduled in the user-space. The user-level process has the beneficial features of the user-level thread. Meanwhile, it has its own program code and data like a traditional process. By assigning a role of a parallel process to a user-level process, all of following purposes can be achieved.

- Enabling the process oversubscription without OS task scheduling.
- Enabling high-performance process oversubscription by avoiding the overhead, which arises at the context switch between parallel processes.
- Enabling the process oversubscription without changing the programing model of the parallel application.

The rest of this paper is organized as follows. Next section provides an overview of the user-level process and shows its capability. Section 3 describes implementation of the user-level process. The preliminary evaluation results are presented in Section 4. Related work is discussed in Section 5. Finally, this paper is summarized in Section 6.

2. Overview of User-level Process

2.1 Process and Thread

Process can be defined as a set of program code (text), data (data, bss and heap), stack and execution context (register states). Meanwhile, Thread can be defined as just a set of stack and execution context. Thread is a subset of process, and then process has one or more threads within itself. Threads within the same process share the program code and data.

When doing the process oversubscription, multiple processes are bound to the same core. Then, task scheduling is required. The context switch between processes requires address space switching, and only OS kernel can conduct it. Thus, task scheduling of processes must be operated in kernel-space.

Another way to do the process oversubscription is to invoke multiple threads within a process and to assign a role of a parallel process to each of them. Threads within the same process run in the same address space. Then, address space switching is not required at the context switch between them. Therefore, task scheduling of the threads within the same process can be operated in both the kernel-space and the user-space. There are two types of threads, *kernel-level thread* and *user-level thread*. The kernel-level thread is a thread scheduled in the kernel-space. On the other hands, the user-level thread is a thread scheduled in the user-space. When OS task scheduling is supported, process has one kernel-level thread at least. The context switch between user-level threads is faster than the context switch between processes and the context switch between kernel-level threads, because the overhead of jumping into the kernel-context can be avoided.

2.2 User-level Process

User-level process is a ”process”, which can be scheduled in user-space. To avoid any ambiguity, we use the term *kernel-level process* when referring to traditional processes that are sched-

© 1992 Information Processing Society of Japan

Table 1 User-level process capability

	kernel-level process	user-level process	kernel-level thread	user-level thread
Enabling the process oversubscription without OS task scheduling	No	**Yes**	No	**Yes**
Enabling high-performance process oversubscription by avoiding the overhead, which arises at the context switch between parallel processes	No	**Yes**	No	**Yes**
Enabling the process oversubscription without changing the programing model of the parallel application	**Yes**	**Yes**	No	No

uled in the kernel-space, and the term *process* to indicate both the kernel-level processes and the user-level processes.

Figure 1 describes semantics views of the kernel-level process, the user-level process, the kernel-level thread and the user-level thread. The user-level process is similar to the user-level thread. However, it has its own program code and data unlike the user-level thread. In this paper, ”process” is defined as a set of program code, data, stack and execution context. Therefore, we equate the user-level process with ”process”. The user-level process is a subset of the kernel-level process, and so the user-level processes within the same kernel-level processes runs in the same address space. Then, address space switching is not required at the context switch between them. Therefore, task scheduling of the user-level processes within the same kernel-level process can be operated in the user-space.

2.3 Capability of User-level Process

As described before, task scheduling of the user-level processes within the same kernel-level process can be operated in the user-space. Therefore, the process oversubscription can be done without OS task scheduling by invoking multiple user-level processes within the same kernel-level process and assigning a role of a parallel process to each of them. Moreover, the context switch between the user-level processes within the same kernel-level process is faster than the context switch between kernel-level processes, because the overhead of jumping into the kernel-context can be avoided. Thus, high-performance process oversubscription can be achieved.

The key feature of the user-level process is that it has its own program code and data. When doing the process oversubscription by assigning a role of a parallel processes to a user-level thread, the programming model of the parallel application must be changed, because the program code and data are shared among parallel processes. In general, programmers implement parallel applications on the assumption that one parallel process has its own program code and data. On the other hands, when doing the process oversubscription by assigning a role of a parallel process to a user-level process, the programming model of the parallel application does not need to be changed, because a user-level process playing a role of a parallel process have its own program code and data respectively. Existing programs can be executed without any modifications, and programmers can implement parallel applications with familiar manner. The capability of the user-level process is summarized and compared with the kernel-level process, the kernel-level thread and the user-level thread in table 1.

The capability of the user-level process is supposed to be embedded in runtimes for parallel computation, such as MPI runtimes[9][7]. By modifying a process manager of a runtime to invoke a user-level process playing a role of a parallel process and implementing a task scheduler for the user-level processes into the runtime, the process oversubscription utilizing the user-level process can be achieved.

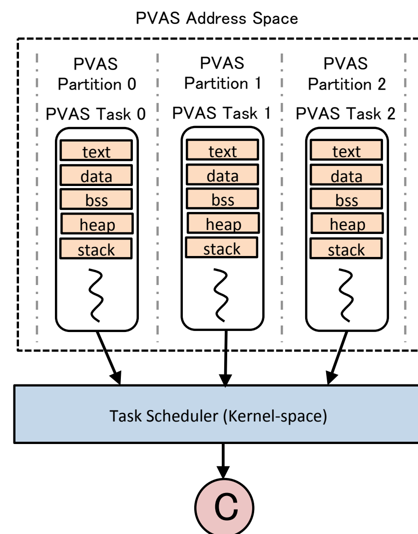

Fig. 2 An overview of the PVAS

3. Implementation

Currently, the prototype implementation of the user-level process works on the Linux for the x86_64 architecture. In this section, implementation of the user-level process is described.

3.1 Partitioned Virtual Address Space

The user-level process was implemented by diverting our former research, Partitioned Virtual Address Space (PVAS) [10][11]. Originally, The PVAS was developed to enhance the intra-node communication between kernel-level processes within a computing node. The PVAS enables kernel-level processes to share the same address space, so that kernel-level processes sharing the same address space can directly access the data of other kernel-level processes. Figure 2 shows an overview of the PVAS. The idea behind the PVAS is to partition a virtual address space and assign one partitioned region to one kernel-level process instead of assigning a whole virtual address space. In the PVAS, the virtual address space that is partitioned is called PVAS address space, and the partitioned region is called PVAS partition. The kernel-level process that is assigned to the PVAS partition is called PVAS task. A PVAS task can have the program code (text), data (data, bss and heap) and stack within its PVAS partition, and thus a PVAS task can load a program into its PVAS partition and execute it. A PVAS task can directly access the data

© 1992 Information Processing Society of Japan

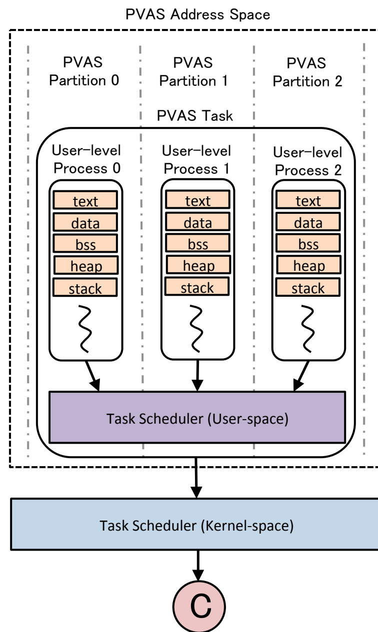

Fig. 3 An overview of the extended PVAS which supports the user-level process

of other PVAS tasks, because no address space boundaries lie between the PVAS tasks within the same PVAS address space.

3.2 User-level Process Support

Here, our PVAS was extended to support the capability of the user-level process. Our implementation of the user-level process uses the address space partitioning mechanism of the PVAS. Figure 3 shows an overview of the extended PVAS that supports the user-level process capability. In the extended PVAS, a user-level process runs as a subset of a PVAS task. In our implementation, one PVAS partition is assigned to one user-level process, and then a user-level process can have its own program code, data and stack within its PVAS partition. A user-level process can load a program into its PVAS partition, and the loaded program can be executed as the user-level process. One PVAS task can be the owner of multiple user-level processes and execute them. Then, the context switch between user-level processes means that the PVAS task switches the user-level process that it is currently executing to another user-level process. The context switch between user-level processes within the same PVAS task does not require address space switching, because they are running in the same address space. Therefore, task scheduling of the user-level processes within the same PVAS task can be operated in the user-space. Having multiple PVAS tasks in the same PVAS address space is allowed. However, one user-level process cannot belong to multiple PVAS tasks, and the context switch between user-level processes that belong to different PVAS tasks is prohibited. Each user-level process has a unique ID that is called the PVAS ID. In this paper, user-level process N means the user-level process whose PVAS ID is N. The size of the PVAS partition is fixed, and so the start address of the PVAS partition possessed by the user-level process N is calculated by the following expression.

$$(N + 1) \times [size\ of\ PVAS\ \ partition]$$

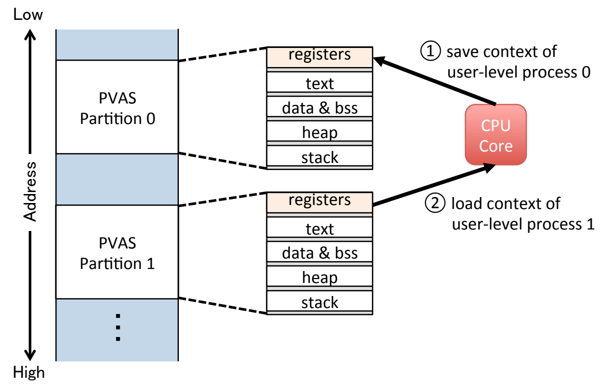

Fig. 4 Context switch from user-level process 0 to user-level process 1 (In this figure, PVAS partition 0 is assigned to user-level process 0, and PVAS partition 1 is assigned to user-level process 1)

The PVAS ID is zero-origin. Thus, to prevent the start address of the PVAS partition possessed by user-level process 0 from becoming NULL, the PVAS ID is shifted by one.

3.3 Context Switch

The context switch between user-level processes is done in the user-space by saving the execution context of the current user-level process and loading the execution context of the next user-level process. In this case, execution context means the status of the registers. Figure 4 describes the context switch from user-level process 0 to user-level process 1. The context switch is implemented by loading the execution context of user-level process 1 after saving the execution context of user-level process 0. The memory region for saving the execution context of a user-level process is pre-allocated on top of its PVAS partition when it is created. Then, the routine for context switching can easily locate the address to which the execution context of the current user-level process should be saved and the address from which the execution context of the next user-level process should be loaded by their PVAS ID and the expression in Section 3.2.

3.4 User-level Process API

We developed the library for leveraging the capability of user-level process. In this section, specification of the user-level process API is described.

3.4.1 PVAS Address Space Creation

- `int pvas_create(void)`

This function creates a PVAS address space where user-level processes are created, and returns the descriptor of the created PVAS address space. This function calls `sys_pvas_create()` which is a system call for creating a PVAS address space.

3.4.2 PVAS Address Space Destroy

- `int pvas_destroy(pvd)`

This function destroys the PVAS address space indicated by *pvd*. *pvd* is a descriptor of the PVAS address space to be destroyed. This function calls `sys_pvas_destroy()` which is a system call for destroying a PVAS address space.

© 1992 Information Processing Society of Japan

3.4.3 PVAS Task Spawn

- `pid_t pvas_spawn(int` *pvd*, `int` *pvid*, `char` **filename*, `char **`*argv*, `char **`*envp*)

This function spawns a new PVAS task within the PVAS address space indicated by *pvd* and creates a new user-level process that is executed by the spawned PVAS task. *pvid* is a PVAS ID of the new user-level process. The new user-level process loads the program indicated by *filename* into its PVAS partition. *argv* is a list of arguments passed to the program. *envp* is a list of environmental variables. At the same time as loading the program, the memory region for saving the execution context of the new user-level process is allocated on top of its PVAS partition. The address at which this region should be allocated is calculated by *pvid* following the expression in Section 3.2. The loaded program is executed as a user-level process by the spawned PVAS task. This function returns the process ID of the new PVAS task. This function calls `sys_pvas_spawn()` which is a system call to do the above operations.

3.4.4 User-Level Process Creation

- `int pvas_ulp_exec(int` *pvid*, `char` **filename*, `char **`*argv*, `**`*envp*)

`pvas_ulp_exec()` create a new user-level process. This function must be called by a user-level process. A new user-level process is created into the PVAS address space where the calling user-level process is running. The PVAS task that is the owner of the calling user-level process also becomes the owner of the new user-level process. *pvid* is a PVAS ID of the new user-level process. The new user-level process loads a program indicated by *filename* into its PVAS partition. *argv* is a list of arguments passed to the program. *envp* is a list of environmental variables. At the same time as loading the program, the memory region for saving the context of the new user-level process is allocated on top of its PVAS partition in the same manner as `pvas_spawn()`. After creating a new user-level process, the context of the current user-level process is saved. Then, the loaded program is executed as the new user-level process. This function returns 0 on success, error code on error. This function calls `sys_pvas_ulp_exec()` which is a system call to do the above operation.

3.4.5 Context Switch

- `int pvas_ulp_switch(int` *pvid*)

This function executes context switch from current user-level process to another one. *pvid* indicates the PVAS ID of the target user-level process which will be switched to. This function performs a context switch by saving the execution context of the current user-level process and loading the execution context of the target user-level process. This function returns 0 on success, error code on error.

Most of this function is executed in the user-space. However, a part of the operation is delegated to the OS kernel. Some libc libraries utilize thread local storage (TLS) to implement thread-safe functions. The TLS enables a kernel-level thread to allocate its own private data region (TLS region). The GCC compiler[3] for the x86_64 architecture uses `fs` register that is one of the segment registers to implement the TLS. In this implementation, the TLS region is pointed to by the `fs` register. The `fs` register is not accessible from the user-space, so the operation for the save and the load of this register must be delegated to the OS kernel. This delegation causes the overhead of jumping into the kernel-context. The value of the `fs` register is constant while a program is executed, so the save of the fs register is required only on the first context switch. However, the load of the `fs` register is required on every context switch. This problem can be avoided by building libc libraries with disabling the TLS and making user-level processes use them. In this case, `pvas_ulp_switch()` can ignore the save and the load of the `fs` register.

ulpspawn

```
#include <stdio.h>
#include <pvas.h>

int main (int argc, char **argv) {
      int error;
      int pvd, status;

      /* Create a new PVAS address space */
      pvas_create(&pvd);
      /* Spawn a new PVAS task */
      pvas_spawn(pvd, 0, "ulpexec", argv, NULL, NULL);
      wait(&status);
      /* Destroy the PVAS address space */
      pvas_destroy(pvd);
      return 0;
}
```

ulpexec

```
#include <stdio.h>
#include <pvas.h>

#define PNUM 2

int main (int argc, char **argv) {
      int pvd, i;

      /* Create a new user-level process */
      for(i=1; i<PNUM+1; i++)
            pvas_ulp_exec(pvd, i, "ulpprogram", argv, NULL, NULL);
      return 0;
}
```

ulpprogram

```
#include <stdio.h>
#include <pvas.h>

      int main (int argc, char **argv) {
      int pvid;

      /* Get PVAS ID */
      pvas_get_pvid(&pvid);

      printf("I am user-level process %d.\n", pvid);

      /* Context Switch */
      pvas_ulp_switch(0);

      return 0;
}
```

Fig. 5 Sample program of the user-level process

3.4.6 Sample Program

Figure 5 describes the sample programs of the user-level process written by C language, and Figure 6 shows the image of executing it. This example consists of three programs, *ulpspawn*,*ulpexec* and *ulpprogram*.

ulpspawn creates a new PVAS address space by calling `pvas_create()`. Then it spawns a new PVAS task and creates user-level process *0* by calling `pvas_spawn()`. User-level process *0* loads *ulpexec* into its PVAS partition, and it is executed as user-level process *0* by the created PVAS task.

ulpexec executed as user-level process *0* creates user-level pro-

© 1992 Information Processing Society of Japan

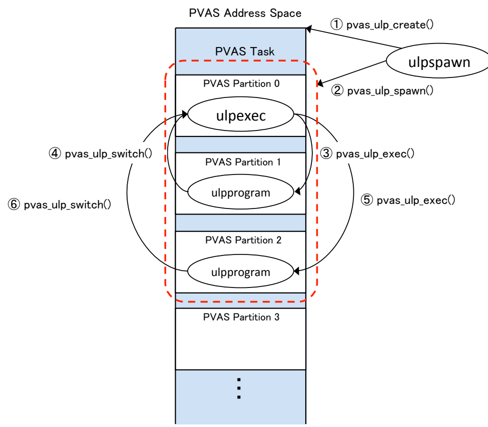

Fig. 6 Image of executing the sample program

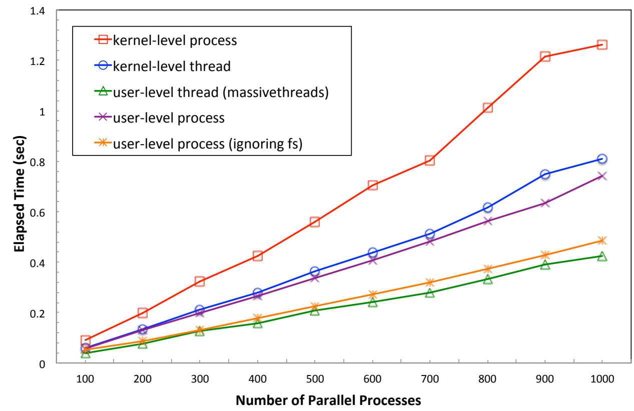

Fig. 7 Context switch performance

cess *1* by calling `pvas_ulp_exec()`. User-level process *1* loads *ulpprogram* into its PVAS partition and it is executed as user-level process *1*.

ulpprogram executed as user-level process *1* performs context switch from user-level process *1* to user-level process *0* by calling `pvas_ulp_switch()`. User-level process *0* creates user-level process *2*, and then user-level process *2* repeats the operation performed by user-level process *1*.

When embedding the capability of the user-level process to the rutimes for parallel computation, these sample programs can be an example of the implementation. For example, supposing that the capability of the user-level process is embedded in the MPI runtimes, `mpiexec` or `mpirun` can be implemented by consulting *ulpexec* and *ulpspawn*, and *ulpprogram* may be a counterpart of a MPI program executed by `mpiexec` or `mpirun`.

3.5 Compatibility Issues

The specification of the user-level process is not equal to that of the traditional kernel-level process. For example, file descriptors cross user-level processes within the same kernel-level process, and transmission of signals takes place between kernel-level processes (not between user-level processes). Therefore, those compatibility issues must be considered when embedding the capability of the user-level process in the runtimes for parallel computation.

4. Preliminary Evaluation

When doing the process oversubscription, context switch performance affects the overall performance of the parallel application, because so much number of parallel processes may be invoked on a single CPU core. Then, the context switch performance of the user-level process was measured in this preliminary evaluation. The evaluation was performed on the Intel Xeon CPU X5670 (2.93GHZ).

The context switch performance was evaluated by using a simple micro-benchmark. This benchmark invokes multiple parallel processes on a single CPU core and makes each of them perform a context switch 1000 times. Then elapsed time until all context switch operations are performed was measured. Four implementations of this benchmark are prepared. First implementation treats one kernel level process as one parallel process. This implementation invokes multiple kernel-level processes and makes each of them perform a context switch by calling `sched_yield()`. Second implementation treats one kernel-level thread as one parallel process. This implementation invokes multiple kernel-level threads and makes each of them perform a context switch by calling `sched_yield()`. Third implementation threats one user-level thread as one parallel process. This implementation invokes multiple user-level threads and makes each of them perform a context switch. We used massivethreads[8], which is one of the high-performance user-level thread libraries. When using massive threads, a user-level thread is invoked by calling `myth_create()`, and the invoked user-level thread can perform a context switch by calling `myth_yield()`. Final implementation treats one user-level process as one parallel process. This implementation invokes multiple user-level processes and makes each of them perform a context switch by calling `pvas_ulp_switch()`. In this way, the context switch performance of user-level process is compared with those of kernel-level process, kernel-level thread and user-level threads.

The execution results of this benchmark are shown in Figure 7. As described in the graph, the context switch performance of kernel-level process is slower than those of others, because the context switch between kernel-level processes requires address space switching. The address space switching needs TLB flush and some other heavy operations, then the context switch performance can be lower. The performance of user-level process is competitive with that of the kernel-level thread. This is why the save and load of the *fs* register on every context switch between user-level processes introduces the overhead of jumping from the user-space into the kernel-space as described in Section 3.4. Then we implemented `pvas_ulp_switch()` which ignores the save and load of the *fs* register. The performance result of this implementation is competitive with that of the user-level thread.

We think that there still remains a room to improve the context switch performance of user-level process. For example, current implementation saves all general purpose registers on every context switch, although they all do not necessarily have to be saved for some situations. If we further examine this problem, the context switch performance of the user-level process will be improved.

© 1992 Information Processing Society of Japan

5. Related Work

FG-MPI[6] is an implementation of MPI runtime, which binds a MPI rank to a user-level thread. By assigning a role of an MPI process to a user-level thread, high-performance process oversubscription can be achieved. The process oversubscription hides the communication latency and results in better load balance, then MPI application performace can be improved.

The problem of the FG-MPI is that it dramatically changes a programming model of MPI. In general, programmers implement MPI applications on the implicit assumption that a MPI process has its own program code (text) and data (data, bss and heap). However, programmers must consider those objects are shared among MPI processes when using the FG-MPI. If FG-MPI binds a MPI rank to a user-level process proposed in this paper, MPI applications can enjoy the benefit of the process oversubscription without changing the programming model of MPI.

6. Summary

In this paper, we propose the user-level process. The user-level process is a "process", which can be scheduled in the user-space. By assigning a role of a parallel process to a user-level process, the process oversubscription can be achieved, even if the OS kernel does not support task scheduling. This is an important feature, because the lightweight OS kernels for Exascale systems may no longer support the task scheduling, although the process oversubscription is beneficial. Moreover, the user-level process enables high-performance process oversubscription. The context switch between user-level processes is faster than the context switch between traditional kernel-level processes, because it is operated in the user-space, and the overhead of jumping into the kernel-context is not required at every context switch.

The capability of the user-level process is similar to that of the user-level process. By assigning a role of one parallel process to one user-level thread, high-performance oversubscription can be achieved without OS task scheduling. However, the process oversubscription utilizing the user-level thread changes the programming model of the parallel application. Meanwhile, the process oversubscription utilizing the user-level process does not change the programing model of the parallel application, because a user-level process has its own program code and data like a traditional kernel-level process.

The capability of the user-level process is supposed to be embedded in the runtimes for parallel computation, such as MPI runtimes. Our future work is to enhance those runtimes to leverage the capability of the user-level process and evaluate their performance.

7. Acknowledgments

This research was partially supported by the CREST project of the Japan Science and Technology Agency (JST).

References

[1] Argonne National Laboratory: Argo: An exascale operating system, http://www.mcs.anl.gov/project/argo-exascale-operating-system.

[2] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. and Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System, *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, New York, NY, USA, ACM, pp. 207–216 (online), DOI: 10.1145/209936.209958 (1995).

[3] GCC, the GNU Compiler Collection: GCC online documentation, https://gcc.gnu.org/onlinedocs/.

[4] Gerofi, B., Shimada, A., Hori, A. and Ishikawa, Y.: Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures, *CCGRID*, pp. 360–368 (2013).

[5] Iancu, C., Hofmeyr, S., Blagojevic, F. and Zheng, Y.: Oversubscription on multicore processors, *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IEEE, pp. 1–11 (online), DOI: 10.1109/IPDPS.2010.5470434 (2010).

[6] Kamal, H. and Wagner, A.: An Integrated Runtime Scheduler for MPI, *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, Berlin, Heidelberg, Springer-Verlag, pp. 173–182 (2012).

[7] MPICH: High-Performance Portable MPI, http://www.mpich.org/.

[8] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, *Concurrent Objects and Beyond From Theory to High-Performance Computing (Festschrift)*.

[9] Open MPI: Open Source High Performance Computing, http://www.open-mpi.org/.

[10] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: PGAS Intra-node Communication towards Many-Core Architecture, *In PGAS 2012: 6th Conference on Partitioned Global Address Space Programing Model*, PGAS'12 (2012).

[11] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: Proposing A New Task Model towards Many-Core Architecture, *Proceedings of the ACM international workshop on manycore embedded systems 2013*, MES'13, Tel-Aviv, Israel, ACM (2013).

[12] Si, M., Ishikawa, Y. and Takagk, M.: Direct MPI Library for Intel Xeon Phi co-processors, *The 3rd Workshop on Communication Architecture for Scalable Systems in conjunction with IPDPS2013* (2013).

[13] von Behren, R., Condit, J., Zhou, F., Necula, G. C. and Brewer, E.: Capriccio: Scalable Threads for Internet Services, *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, New York, NY, USA, ACM, pp. 268–281 (online), DOI: 10.1145/945445.945471 (2003).

[14] Wheeler, K. B., Murphy, R. C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads., *IPDPS*, IEEE, pp. 1–8 (2008).

© 1992 Information Processing Society of Japan