Porting of the OZ++ to the AP1000+ parallel computer

Yoichi Hamazaki　Yuriy Kazakov‡　Michiharu Tsukamoto
hamazaki@etl.go.jp　yuriy@asi.co.jp　tukamoto@etl.go.jp

Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba Ibaraki 305, Japan
‡: During the research, he was AIST fellow.

Abstract　Parallel computers are powerful tools for execution of compute intensive programs. However, programming these machines is still awkward due to the lack of a convenient programming environment. This work is devoted to porting of the OZ++ object-oriented distributed environment to the AP1000+ distributed memory parallel computer. The OZ++ is a middle-ware, now implemented on the base of SunOS operating system. The approach of the porting is to build a library of SunOS system calls to support quick and qualitative porting of OZ++ to AP1000+. This paper presents the design of the SunOS call library and investigates the ways of its implementation.

オブジェクト指向分散環境 OZ++ の並列計算機 AP1000+ への移植

濱崎 陽一　Yuriy Kazakov‡　塚本 享治

電子技術総合研究所
305 茨城県つくば市梅園 1-1-4
‡：前 AIST フェロー

概要　汎用プロセッサを用いた並列計算機は多くのプロセッサと高速のプロセッサ間通信を持つ事から、分散アプリケーションの移植先としても魅力的である。しかしながら、並列計算機のシステムソフトウェアは分散アプリケーションの実装には不都合な場合が多い。
ここではオブジェクト指向分散環境 OZ++ の並列計算機 AP1000+ への移植を試み、その手段として必要なシステムコールをライブラリとして提供方式を提案する。AP1000+ で提供される CellOS+ で提供されていない UNIX のシステムコールは SCLibrary と呼ぶライブラリによって提供し、SCLibrary は AP1000+ のホストコンピュータ上のエージェントと呼ぶデーモンプロセスによって所望の機能を実現する。
また CSLibrary のプロトタイプを実装し、その性能測定を行なった。AP1000+ の持つセル間通信を使わず、ホストコンピュータ経由でセル間の通信を実装したために、その時間的なコストは大きい。しかし、移植するアプリケーションと AP1000+ がほぼ独立したものとなるために、アプリケーションの変更による再移植の手間は小さく、開発中のアプリケーションを速やかに移植する場合に適している。

1 Introduction

Parallel computers which use general purpose processing elements are attractive targets to port distributed software, because they have a lot of processing elements and high-speed communication medium between processing elements. Unfortunately it is not so easy to port distributed applications to parallel computer in general because system software of such parallel computer is not designed to be used for distributed applications but for parallel ones.

In this article we propose a library approach for the porting based on our experience of porting an object-oriented distributed environment OZ++ to the AP1000+ parallel computer. We are not considering parallelizing problems, the OZ++ is a highly distributed software system consists of various components which can work in parallel. So the structure of OZ++ system enables flexible opportunities for distribution of software entities among AP1000+ processors.

Main part of OZ++ system is written in C language, and now it is implemented for a network of Sun workstations. AP1000+ has C language compiler but does not have advanced OS such as Unix. So the main problem of porting of OZ++ to AP1000+ is that OZ++ source codes contain Unix (SunOS) system calls which are not supported on AP1000+ in proper manner.

On the other hand, the OZ++ is a still developing project, its source codes are changed often. The AP1000+ software also develops fast. The porting approach should support the porting of not only one version of OZ++ software. It should be much more a technique for quick and qualitative adoption of every new version of the OZ++ to the new version of basic AP1000+ software.

In this article we propose a library approach for the porting. The library (SCLibrary) supports the SunOS system calls on AP1000+ and enables some level of independence of OZ++ codes from AP1000+.

In section 2 and 3, OZ++ and AP1000+ are overviewed, and in section 4 we presents SCLibrary's architecture and considers various strategies its organization. In section 5, investigation of prototype SCLibrary is presented and conclusion is in section 6.

2 OZ++: an Object Oriented Distributed Environment

OZ++ is a programming environment to support distributed object-oriented applications in heterogeneous distributed computer systems [1]. In OZ++, classes are managed all over the system and objects are transferred among machines. Object methods can be executed at any machine by loading their classes from class management system on demand. This execution mechanisms and dynamic loading of classes over networks make distribution and execution of applications easy.

On every computer one nucleus and any number of executors exist. Executor is a basic unit of OZ++ service environment. Every object is placed on an executor and its methods are executed by the executor. An executor creates, holds and deletes objects. It loads executable codes of classes and layout information from class management system and cache them. It saves persistent objects on secondary storage and loads them on demand. Memory management and garbage collection are functions of executor also.

Executor is a separate unit of OZ++, it can work along on computer. But for communication with the other executors the nucleus have to be used. Nucleus is a daemon process, one nucleus exists on every computer where the OZ++ system goes. Nucleus creates and manages executors on its station. It keeps state and communication address of each executor and resolves communication addresses from executor ID by asking other nucleuses using broadcast.

Several kinds of management objects, object for class management, object for object management on an executor, etc., are placed appropriately on executors. Various application objects are implemented also (object launcher which provides user interface with executor, compiler which produces executable code and other class information from programs written by OZ++ object-oriented language, etc.). All these support flexible, powerful and comfortable environment for programming.

The OZ++ is a middle-ware, so nucleus and executors are implemented as Unix processes. The object methods and other daemon programs are executed using multithreading mechanism inside the executor.

3 AP1000+ massively parallel computer

AP1000+ is a distributed memory parallel computer [2]. SuperSparc processor elements, called cells, are connected by three independent communication networks. These are the torus network for point-to-point communication between cells, the broadcast network for data distribution and collection, and the synchronization network for barrier synchronization.

A host computer is used for the AP1000+ interface with the other world. It sets up cell configurations, generates tasks and supports a user interface with AP1000+. The broadcast and synchronization networks connect all cells and host. A Sparc Station computer is used as host, and one Unix process on host manages the AP1000+ and communicates with it.

CellOS+ is the operating system for the AP1000+. It supports start and parallel execution of cell programs. CellOS+ consists of kernel on each cells, AP1000+ driver on host and libraries for cell programs and host programs, called as cell libraries and host libraries. Host and cell programs are compiled and linked with host libraries or cell libraries at the host computer. The cell libraries implement various kinds of communication and synchronization but they do not support Unix-level of service sufficiently.

The cell libraries contains read() and write() functions but they can be used only input/output on stdin/stdout of host. HAP (Host Access Package) enables cells to access files on host computer but the set of calls is very poor (open(), read(), write(), lseek(), close()) and implementation is very restricted [3]. For example, read() and write() are blocking and synchronous, they always block a cell program until completing of input/output operation.

Recently a work is done in the direction of Unix implementation on AP1000+. Chorus Microkernel IPC have been implemented for AP1000+ [4], the next step is going to be Unix implementation on the base of Chorus/MiX.

4 Porting of OZ++ to AP1000+

Executor is an elementary unit of service environment of OZ++, it can work in parallel with the other parts of OZ++ system. So it is natural to distribute executors on cells of AP1000+ and to leave nucleus on host computer to support communication with the other computers. This architecture corresponds to concepts of nucleus and executor, and doesn't restrict user opportunities.

Executor is developed in C language and implemented for Sun workstations. AP1000+ has C language compiler but does not have advanced OS such as Unix. So the main problem of porting of executor to cell is that executor source codes contain Unix system calls which are not supported on AP1000+ in proper manner.

4.1 SCLibrary approach

We propose approach to use library which provide functions of SunOS system calls on AP1000+ (SCLibrary stands for SunOS system call Library). The library has to support system calls which are used by executor, and it should be special-purpose for efficient work of executor on cell. For example, mmap() is very powerful and universal SunOS system call but an executor uses it only for private, system-free mapping to the file. So it is not necessary to implement other types of mapping in SCLibrary but this kind of mapping should be implemented in the best way.

The library approach implies some level of independence of executor codes from AP1000+. In case of library approach, the porting of new versions of executor to AP1000 can be done more easily because of high independence of executor source codes from AP1000+. A new OZ++ version may require adding of new system calls to the SCLibrary or rewriting of an old implementations of some system calls, and this is all.

In case of Unix implementation on AP1000+, the independence level would be the highest, and porting could be done more easily. But Unix is a rather heavy system, consuming much processor time. An executor does not need in Unix as such, it need only in adequate and fast support for some SunOS system calls. Moreover a user of OZ++ doesn't need in Unix on cell because executor would supply him a flexible and pow-

erful environment for running his applications on cells and his applications are programmed in OZ object-oriented language with class libraries.

4.2 Design of the CSLibrary

The current versions of executor software contains more than 70 SunOS system calls. Most of them are input/output calls, dealing with descriptors and sockets (around 50 system calls), the other work with memory, signals, and Unix system information (such calls as sysconf(), getpid(), etc.).

Some parts of SunOS environment can not be created on cell directly. For example, AP1000+ does not have display and connection to network, so the display and network input/output have to be performed via the host computer. If an AP1000+ does not have a disk memory, then host files or NFS files have to be used. We will call such system calls external system calls.

Some daemon software has to exist on host to support external system calls. We will call a process which runs on host and supports external calls, an agent. In accordance with the AP1000+ architecture all agents must communicate with the AP1000+ by way of one host communication process.

It is the external calls that is of primary concern to SCLibrary design. The other system calls can be executed by SCLibrary faster than by Unix because they do not need in interrupts (they are not real system calls but function calls), and they are specialized for executor. As for external calls, their remote execution may require much time.

Executer is a multithreading program so it is important not to block the executor after system call invocation and allow other threads to execute while the system call is executed. But this contradicts with the Unix system call concept.

Parallelism in Unix is supported by processes. Unix process switching and interprocess communication cost much. So asynchronous system calls and signals are used inside a process for parallel input/output and child management. This internal process parallelism is very restricted and it is hard to understand programs that deal with signals because of their random nature.

Multithreading packages support much more powerful and conceptual model of parallelism inside of process. But Unix kernel doesn't support threads, they are invisible for Unix. So system calls serialize threads, they are compulsory critical sections, and what is more bad they block not only thread invoked system call, but all threads of the process.

As a result, most of multithreading packages has their own libraries which substitute I/O, time, and child management system calls. For example, LWP (Light Weight Processes) package has nonblocking I/O library, the OZ++ executor has nonblocking I/O functions in OzLibrary. The calls of these libraries block thread but the other threads can continue their development. So these calls are blocking and synchronous for thread which invokes them, and they are nonblocking and asynchronous for multithreading process (the other threads are not blocked and can develop further). Threads don't need in asynchronous system calls and the signals because parallelism and asynchrony can be supported naturally by multithreading.

So if system calls support threads, then system call semantics becomes much more simple (signals and asynchronous modes are not used), conceptual and high-level than the Unix one. We propose to use thread supporting external calls for the SCLibrary. The SCLibrary have to be integrated with executor multithreading mechanism to switch threads after external call invocation.

This may provide more fast execution because:

- executors will not wait until external system call completing, the other ready threads continue their development;
- communication expenses between cell and host decrease because only one request for operation and one answer are transferred (no multiple attempts and signal information transferring).

On the other hand, an agent have to buffer external calls and implement them using traditional Unix asynchronous system calls and signals. This means that really the thread-supporting nonblocking functions (of OzLibrary) move from executor on cell to an agent on host.

5 A Prototype of SCLibrary and its investigation

To realize thread supporting external calls, the SCLibrary must have a daemon software on host to support SunOS environment on cell adequately.

This point of the SCLibrary architecture may become most critical for executor performance. The remote execution of external calls costs much so this part of SCLibrary must be implemented carefully and punctually.

For investigation of this question and performance measuring, a prototype version of SCLibrary was implemented. Agents play an important role in this version. All system calls dealing with disk memory, network and Unix data, are considered as external and executed remotely by agents on host. This version presents a SCLibrary for diskless configuration of AP1000+. Host computational power is used in the most intensive manner. The loading of executable codes and layout information, communication with the other parts of OZ++ system, that is to say, all executor input/output is done by way of host.

For execution of external system calls ,it is proposed that each executor on cell has its agent on the host. As long as an executor's input/output is done via its agent, the executor sees this world by agent's "eyes". So it is "of the opinion" that it runs in host SunOS environment. Alternatively, the other processes working on host can see only the agent, the executor on cell is invisible from them. But the agent looks like its executor and behaves as the executor, and the other processes on host consider the agent as the executor. So AP1000+ is transparent for the AP1000+ executors and the host processes.

Figure 1 shows a general structure of the OZ++ system on AP1000+ according the prototype version of SCLibrary.

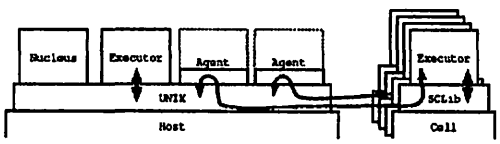

Figure 1: An Overview of the OZ++ on AP1000+

When an external system call is invoked, its code and parameters are packed to a message and sent to agent. The agent receives the message, executes the call in SunOS environment as if it is the executor, and sends results back to cell.

Agents allow to separate ported codes from the the others, only executor source codes have to be changed. Nucleus and the other OZ++ software, i.e. classes for system management and for user applications, are the same as before because they do not know that some executors run on the AP1000+.

In the following subsections we are discussing the influence of the remote execution of external system call on the executor performance. In subsection 5.1 we examines the time of external call execution. Subsection 5.2 considers the influence of external call execution on the time of executor start — one of the most input/output intensive procedure of executor. Subsection 5.3 considers the influence of external call execution on the time of object method invocation. In subsection 5.4 we are discussing a role of thread-supporting system calls for external system call execution.

A Sparc Station 10 was used as a host computer for AP1000+. The executor performance on AP1000+ was compared with performance of ordinary (not-ported) executors on Sparc Station 10. An AP1000+ cell's SuperSparc processor differs slightly from the SuperSparc of Sparc Station 10 (it doesn't have secondary cash and memory access algorithms are a little bit different). Therefore, in average the Sparc Station 10 is slightly faster than a cell of AP1000+.

Measured data used in the discussion below are average of several trial. But measurement environment was not isolated but connected to other computers by ethernet, measured data may be affected external condition.

5.1 Time of remote execution of external calls

Most of external system calls invoked by executor, are input/output system calls. Usually more than 99% of all system call invocations correspond to the following system calls: read(), write(), lseek(), send(), recv(), open(), close().

The time of a system call remote execution (transfer and execution as such) varies from 2000μs to 3500μs if the call does not require a transfer of much data (more than 2 K Byte). The time of execution, as such, on Sparc Station 10 varies from 20 to 100 μs depends on the kind of system call.

Some input/output system call invocations (read(),write(),send(),recv()) may require transfer of much data. This causes increasing of remote execution time. Table 1 gives the dependence of

	Unix	SCLibrary
read (..., ..., 10)	39	2439
read (..., ..., 100)	40	2711
read (..., ..., 1000)	74	3242
read (..., ..., 10000)	900	11950
read (..., ..., 100000)	8736	94266
read (..., ..., 1000000)	89249	945946

Table 1: Execution time (μsec) of read() system call

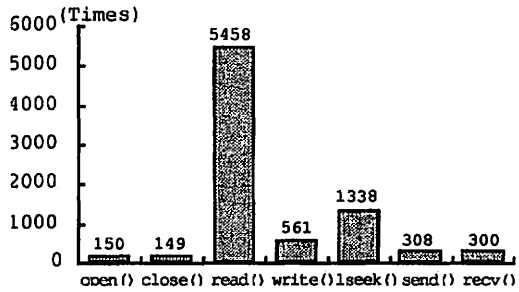

Figure 2: Number of system call invocations during an executor(slave type) start up

execution time values of read() system call from the size of read data.

5.2 Influence of external calls on executor start up

Start up of an executor is a time consuming procedure, especially for executor on AP1000+. During the start up of an executor, it mainly reads much amounts of data from files to memory. Figure 2 shows the numbers of system call invocations during start up of a slave type executor. Each other system call which is not presented in fig. 2, is invoked less than 10 times during a start up.

92% of invoked calls did not transfer much data. The others required transferring of large amounts of data (more than 2Kbyte).

A start up time of an executor is divided into 2 periods:

- classes and codes preloading;
  Classes and their codes which are essential to execute management objects are preloaded first. After start up of OZ++ system, classes and codes can be loaded over network, but it is necessary a special class loading procedure to start up OZ++ system itself.

- objects loading.
  Afterward, management objects which are a part of OZ++ system are loaded from the images on disks to memory and initialized. Object-manager is a such objects which manages objects on an executor. Other objects are loaded when they are accessed with the help of the object-manager.

Figure 3 shows start up times (min) of various types of executors.

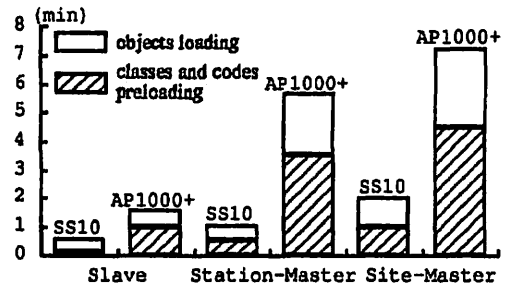

Figure 3: Start up times of various types of executors

Slave executor has two management objects, an object-manager and an application launcher. Station-master executor has a class management object in addition to slave executor. Class management object is large object whose image is more than 2 mega-bytes and it requires many classes. Loading class management object and its classes takes large share of the start up time. Site-master executor has a name directory and a catalog in addition to station-master executor. These two objects are small, but it takes a time to initialize.

Preloading of classes and codes using CSLibrary is 4.5 to 10 times slower than non-porting executor, it is directly influenced by the difference of execution time of read() system call. Object loading using CSLibrary is 1.2 to 4 times slower than non-porting executor, the difference is smaller than that of classes and codes preloading because ini-

tialization of objects takes time and initialization is done without external system calls. In total, start up time of an executor using CSLibrary is 2.7 to 5.4 times longer than that of non-porting executor.

5.3 Influence of external calls on method invocation

When an object invokes method of another object which is located on another executor, this method invocation contains external system calls implicitly.

Figure 4 shows method invocation times (μs) for various locations of invoking and invoked objects. The method to be invoked has no argument and no return value. Ellipses in the figure denote objects which work on the base of executors. Arrows denote invocations, and a number above an arrow is a value of invocation time.

Invocation between two objects on an executor on cell is slower than that on host because of their hardware difference.

Method invocation between objects on other executors, two send() external calls are invoked from invoking object to send invocation request and one send() external calls are invoked from invoked object to send execution result.

Method invocation between executors on other cells takes as two times long as method invocation between executor on cell and executor on host. In the former case, data transfer between executors travels Executor(invoking) —[send()]→ Agent ⇒ Agent —[send()]→Executor(invoked) and in the latter case, data travels Executor(invoking) —[send()]→ Agent ⇒ Executor(invoked). Two external calls are necessary for one data transmission in former case, and this makes time double.

Method invocation from an object on a cell to an object on workstation over networks is little faster than that from an object on a cell to an object on host, because the former does not make process switch from agent to executor on host.

In the prototype version of SCLibrary a communication between AP1000+ executors is made via their agents, through host. Therefore it cost much, for example, more than communication with an executor in network.

The large cost of a host–AP1000+ communication derives from the fact that AP1000+ architecture is designed in paradigm that host is used

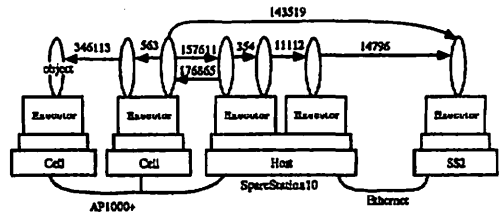

Figure 4: Method invocation times (μsec)

mainly for cell configuration, task generation, input parameters distribution and results gathering. It does not fit well for intensive interaction of cell tasks with processes on host.

5.4 Thread-supporting system call semantics

The discussed above version of SCLibrary is based on thread-supporting system calls. The previously implemented version of SCLibrary was based on process-supporting traditional blocking semantics. It implies the following. After an external call invocation an executor waits for results from cell. In case of asynchronous nonblocking system call, the result coming from agent may indicate that operation is "in progress". Input/output and child signals are transfers by agent to executor on cell via special messages to indicate that the operation can be attempt once more. So this version of SCLibrary implements more complicated algorithms for external call execution but its agents are much more "light weight".

The comparison of executor start up times for two versions of SCLibrary are listed in figure 5. It shows that in average the thread-supporting SCLibrary provides a better executor performance than the process-supporting one.

Generally speaking, we can consider as external calls not only SunOS system calls. Actually, the thread-supporting calls are functions on the base of asynchronous system calls and signals but we decided to consider them as the external calls.

Many other functions of C library or Oz Library can be considered as external calls and executed by agent. However, the strategic line is to leave as much as possible on cell because the purpose of AP1000+ usage is to push computational work

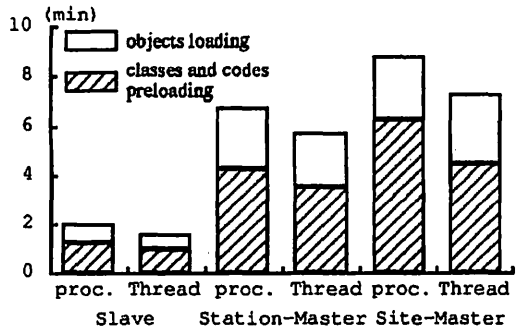

Figure 5: Dependence of start time from threads/proc-supporting system call semantics

from workstation to powerful parallel computer. Only functions that does not do computations but supports communication, are located in agents.

6 Conclusion

The SCLibrary was presented as the basic environment for porting of OZ++ executors to AP1000+. External calls and agent daemon concepts were proposed for SCLibrary support from the host side. The investigation reported above showed that the agent cost is rather expensive. Thread-supporting system call semantics is used to decrease cost of the agent support.

The results of the reported experiments with a prototype version of SCLibrary demonstrated that most of system calls invocations correspond to input/output operations. So the use of AP1000+ disk memory and cell connection networks of AP1000+ suggests reduction of agent expenses radically (for example, in case of starting procedure - more than a hundred times).

What the agents are really necessary for, is communication with nucleus (in particular, for nucleus-executors shared memory emulation) and communication with the executors in network. But executor is a fairly independent entity, it does not need in such communication often. So the cost should not be so high.

OZ++ system have made large version up while developing the prototype of SCLibrary, and it took few days for porting of new version of OZ++ system to AP1000+ only by miner change of executor's source code and recompilation. The size of executors source code is about 20 thousands line in C language.

These show that our library approach is suitable for quick and qualitative porting of systems which still under development, but it is difficult to provide high performance without using special hardware functions which are provided by parallel computer.

During the research, Y. Kazakov worked at Electrotechnical Laboratory and he was supported by Invitation Program for Overseas Visiting Researchers of Agency of Industrial Science and Technology (AIST), Ministry of International Trade and Industry (MITI) of Japan

This research is a part of the R & D program "Open Fundamental Software Technology" of the Information Technology Promotion Agency, Japan (IPA).

References

[1] Tsukamoto M. et al, "The design and implementation of an object-oriented distributed system based on sharing and transferring of classes.", Trans. of Information Processing Society of Japan, vol. 37, No. 5, pp 853-864,(1996).

[2] Shiraki O. et al, "Architecture of Highly Parallel Computer AP1000+", Proc. of the 3rd Parallel Computing Workshop, Fujitsu Laboratory Ltd.,(1994).

[3] Fujitsu Laboratory Ltd., "HAP User's Guide",(1993).

[4] Imamura N. et al, "An Implementation of Chorus Microkernel IPC on AP1000+ and its evaluation", Proc. of the 5th Parallel Computing Workshop, Fujitsu Laboratory Ltd.,(1996).