META PROCESS MODEL: A NEW DISTRIBUTED SHARED MEMORY PROGRAMMING MODEL

H. MIDORIKAWA
Seikei University
3-3-1 Kichijojikita-machi Musashino-shi, Tokyo, Japan
midori@is.seikei.ac.jp

Abstract

This paper proposes a new parallel programming model named Meta Process model and MpC language for its API. The Meta Process model is based on a hierarchical shared memory model and is characterized by its explicit parallelism description. On these points, this model is different from either the shared memory model (SMM) or the message passing model (MPM). Meta Process is a coined term that represents a group of cooperative processes to achieve a single application. The Meta Process model introduces *shared* data that can be accessed by all processes in the Meta Process and distinguishes process-local and process-shared data clearly with using scopes. Processes in a Meta Process share a single address space for shared data. Shared data are maintained by a relaxed memory consistency model. A programmer describes process interactions explicitly using shared data access and synchronizing operations, such as a barrier or a lock/unlock. So this model enables us to write a wide variety of parallel programs, not only SPMD programs like OpenMP/HPF but also irregular and asymmetric programs. The Meta Process model provides us both flexible parallelism description of the MPM and good program readability/writability of the SMM. An actual installation scheme for this model on the software DSM is also described.

Key Words

Distributed Shared Memory, Parallel Programming, Programming Model, Parallel Language

1. Introduction

The two most representative parallel programming models are the message passing programming model (MPM), known as PVM[1] and MPI[2], and the shared memory programming model (SMM), such as OpenMP[3] and pthread[4]. In MPM, MPI became the first standard API for parallel programming and it enables general programmers to write parallel programs easily without paying attention to an underlying computer configuration. It greatly contributes to the portability and the popularization of parallel programs. On the other hand, the SMM has a benefit in good continuity from existing sequential programs. Moreover, it alleviates the difficulty of reading/writing parallel programs, because it needs no bothersome statements for message passing in the programs. In this background, OpenMP was proposed as a standard API for the SMM, and it showed its effectiveness in some shared memory systems[5][6]. However, a typical SMM environment, such as pthread and OpenMP, is not available to or is not so efficient for cluster computers. Cluster computers are widely used recently in parallel processing because of their good cost performance. Usual cluster computers only have relatively low speed connections between computing nodes with regard to their CPU performance. They are categorized into a distributed memory system. Another effort to derive high performance in cluster computers with the SMM instead of the MPM is the development of software distributed shared memory systems (SDSMs), such as TreadMarks[7], JIAJIA[8] and SMS[9]. These systems use one of the relaxed memory consistency models [10-12] to reduce communication overhead between computing nodes. In SDSM programs, programmers should be conscious of the relaxed memory consistency model employed in the underlying SDSM. The pure SMM such as a pthread programming differs in this point In addition, programmers should describe the parallelism explicitly using shared data access and synchronizing operations like a barrier or a lock/unlock. This explicit parallelism expression is a similar way to the one used in pthread programming.

Using OpenMP on cluster computers, in other words, using the SMM on a distributed memory system, usually causes more performance degradation than using SDSMs [13]. The main reason is OpenMP is originally designed for a fork-join parallel thread model in a shared memory not for a distributed memory and it places much importance on the continuity of sequential programs. Its API tends to hide a detail of actual parallel executions and data layouts over distributed memory. So it supports no distributed data mapping facilities to each cluster node and causes implicit and redundant memory consistency synchronizations to maintain a single shared memory illusion on a distributed memory system. OpenMP supports several directives for *data environment*, such as *threadprivate*, *shared*, *private* and *first/lastprivate*, and it can copy private to/from shared data. Though programmers cannot control data layout explicitly. Even if users do not specify these directives, data attributes are changed/copied by OpenMP automatically and implicitly when entering/leaving a parallel region. Moreover default data attribute is *shared*, which is expensive for SDSMs. To overcome such degradation, some dialects of OpenMP add extended facilities to specify data mapping [14][15]. Others explore a hybrid model with OpenMP and MPI. Though, the more elaborated facilities are incorporated in the extended OpenMP, its simplicity and integrity of SMM seems to be more deteriorated. Using such facilities causes chaos, where system's implicit executions and user-specified explicit executions are complicatedly mixed, and it makes programmers more confused. Strictly speaking, these extended SMM API is not a SMM API any more. Although it is ideal that the same SMM API can be available on any systems with high performance independently from underlying memory configurations, it is actually difficult to achieve high performance on cluster computers or NUMA systems if programmers pay no attention whether data are local or global, private or shared. At current stage, there is a limit to obtain high performance using the pure SMM that deals all data in the same way without data layout.

In these backgrounds, this paper proposes a new parallel programming model named Meta Process model. It is based on a distributed shared memory model, not a shared memory model, and it employs an explicit parallelism description paradigm. It can handle the parallel execution on a cluster computer more efficiently compared to the SMM. Meta Process is a newly coined term that represents a group of cooperative processes to achieve a single application. Meta Process model introduces *shared* data that can be accessed by all processes in a Meta Process and it distinguishes process-local and process-shared data distinctly with a hierarchical scope concept. For this purpose, a new parallel language MpC, an extension of ANSI C, is also proposed as its API. In this model, processes in a Meta process share a single address space for shared data. A relaxed memory consistency model maintains shared data. Each process in a Meta process executes its program independently. A programmer describes process interaction unambiguously in a program using shared data access and synchronizing operations. So it enables us to write various types of parallel programs, not only SPMD programs in OpenMP, HPF[16] or UPC[17-19], but also irregular and asymmetric programs. The Meta Process model enables us to describe program's parallelism flexibly like the MPM and it makes programmers read/write parallel programs easily like the SMM. An actual execution scheme of this model on SMS, which is a software DSM developed by us, is also described.

2. Explicit Parallelism Description

The APIs and languages for the SMM can be categorized into two groups. One group places much value on the continuity of sequential programs. Another is characterized by its explicitly parallel programming paradigm. In OpenMP and HPF, one of the most appealing points to users is supporting incremental parallelism. The incremental parallelism is that a programmer can easily extends an original sequential program to a parallel program by adding *pragma* statements. If removing these statements, the program can execute as a sequential program. In other words, these APIs target to sequential algorithms that can be executed in partially parallel by adding simple parallelizing directives, *parallel for*, for example. Regular and symmetric processing, such as array calculations and loop iterations, are typical targets for them. They were not designed to implement parallel algorithms, which is completely different from sequential ones in some cases. Even if people are not familiar to parallel algorithms and parallel programs, they can run their sequential programs in parallel with their minimum costs. Systems for these APIs implicitly parallelize users' programs, so users need no detail knowledge of the parallel execution. However, these API are not adequate for implementing irregular and asymmetric parallel algorithms, which is usually neither SPMD nor data-parallel.

On the other hand, in pthread programming, programmers should specify parallel execution declaratively. It is the API for implementing parallel algorithms, not for expanding sequential programs to parallel ones. For parallel programmers, it is important to control parallelism, load balancing and shared data management, differently from sequential programmers. Though they favor a good readability and writability of the SMM, which has no message passing statements, they do not need implicit parallel executions like OpenMP or HPF, which degrade the readability of their explicit parallel description and their algorithms.

OpenMP and pthread are usually categorized into the same programming model, the SMM, but, roughly speaking, their main targets are different. One is for a community that engages in scientific computations using arrays and iterations with affinity to Fortran sequential programs. Other is for parallel program developers with affinity to C programs. The Meta Process model targets the later community like pthread, and it is the model where parallel programmers describe parallelism explicitly. To implement parallel algorithms, some parallel primitives are needed besides existing sequential statements. So MpC employs well-known API, such as a barrier and a lock/unlock, which is used in pthread and representative SDSMs. Moreover MpC supports facilities for shared data distributed mapping, which usually do not or should not exist in the SMM.

3. Meta Process Model

Usually, parallel processing on cluster computers is the cooperation of multiple processes on each computing node as shown in figure 1. Newly proposed programming model deals such cooperative processes distributed over the nodes as one execution entity, called Meta Process. Each process in a Meta Process is a traditional process, which is identified by the node's OS and associated to some resources, such as memory and files. Though Meta Process is not identified by ordinary OSs, it is identified as a user's single execution entity by some system software, which supports a DSM function, for example. For a user's view, it is a single application program execution on cluster computers.

3.1 Asymmetric Parallel Process Execution

Meta Process has a flexibility that allows both a SPMD style execution, where all processes execute the same program, and a MPMD style execution, where each process executes a different program. Each process in Meta Process executes a specified program independently. The interaction between processes in a Meta Process is specified by a programmer's explicit directions by barrier/lock and shared data access written in each program.

What kind of processing is done in each process of a Meta Process is specified in a *Meta Process description file*. It includes associations between a logical process name and a program module with its arguments. The logical process name has no relation to a physical host or node. The association of the logical process names and actual nodes are done in run time according to a *Processor file*. So *Meta Process description file* is independent from an underlying computer configuration and it only depends on a parallel algorithm created by a user.

3.2 Single Shared Address Space

This model introduces two hierarchical execution entities, process and Meta Process, and it also provides two storage types of data, which can be available in the each scope level. First type is local data in a process, the second one is global data shared among the processes in a Meta Process. For newly introduced *shared* data, the same address used in each process points to the same shared data. So users need no message passing statements in their programs. Run time data scopes are shown in figure2. The scope and the lifetime of process-local data are controlled by ANSI C rules, which is automatic, global or static. Other processes cannot access these process-local data

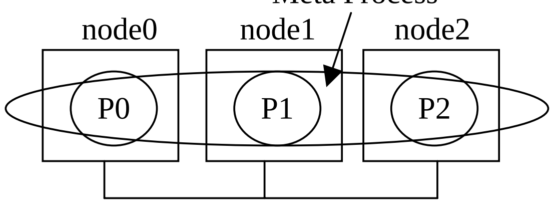

Figure 1 Meta Process and its constituent processes

directly. Only *shared* data, which newly defined in the model, can be accessible from any process in a Meta Process.

3.3 Relaxed Memory Consistency

This model is designed on the premise that shared data is maintained by one of the relaxed memory consistency models [10], which is often used in SDSMs. So programs for this model can be executable both on distributed memory systems and on shared memory systems efficiently.

3.4 Shared Data Layout

This model provides an API for associating shared data with logical processes. To specify the most closely related shared data for the process that access them frequently enables to achieve high performance on distributed memory systems. If such definite association does not exist in applications or a user has no knowledge of application nature, association specifiers are omissible. In that case, an installation of this model associates the shared data to an arbitrary process according to the installation's policy. The association specifier is a hint for the installation system, like *register* in C language. For array data, flexible cyclic data allocations to a sub group of processes are supported, which include line, band, tile, and cube mapping, etc. A user can declare shared data with any association specifiers that appropriate for their application. Not only static declarations but also dynamic data allocations are possible.

3.5 High Portability

This model can be portable among distributed memory systems and shared memory systems. Representative SDSMs and pthread have similar APIs, such as a barrier and a lock/unlock. Standardizing these similar APIs into MpC API standard, it is possible to port MpC programs from some SDSM to some other SDSM, or from some SDSM to some shared memory system.

4. MpC Language

MpC (Meta Process C) is designed to realize the Meta Process model on DSMs and shared memory systems. It is a parallel extension of ANSI C.

4.1 *Shared* Storage Class Specifier

In the MpC, *shared* is newly added to the storage class specifier of ANSI C. The data prefixed with a keyword *shared* must be declared outside of functions in at least one of the program files that are linked into a single execution program module. The scope of shared data is the whole Meta Process, and all processes in the Meta process can view and access the data in run time. Like global variables in C, an external reference to shared data defined in other files also requires *extern shared* or simply *shared* declaration. The scope hierarchy is *shared, global* and *local,* in order from external to internal as shown in Figure2. If the same variable name is used for different scope level data, the most internal variable data are effective by the name. In our installation of MpC, shared data are allocated dynamically when a Meta Process is started, and they are deallocated when the Meta Process exits. So the lifetime of shared data can be considered to be the same as the static data in C.

The MpC deals pointers like original C, which don't care storage type of the content referenced by a pointer. It ignores whether the content pointed by the pointer variable is shared or local. It only takes account of type specifiers, which is int, double or char, for example. The MpC simplifies pointer expressions. For an integer pointer, only 2 expressions are allowed.

• a local pointer variable which points to integer:
  int *p1;
• a shared pointer variable which points to integer:
  shared int *p2;

In contrast, the other similar language, UPC[17-19] supports 4 possibilities as follows.

• a local pointer variable which points to local integer:
  int *q1;
• a local pointer variable which points to shared integer:
  shared int *q2;
• a shared pointer variable which points to local integer:
  int **shared* q3;
• a shared pointer variable which points to shared integer:
  shared int **shared* q4;

In the UPC, *shared* is one of the type qualifiers in ANSI C. The MpC regards such 4-pointer distinction is not necessary in actual use, and it is natural to deal *shared* as one of the storage classes, which includes *auto, register* and *static* etc., rather than dealing *shared* as one of the type qualifiers, which includes *const* and *volatile.*

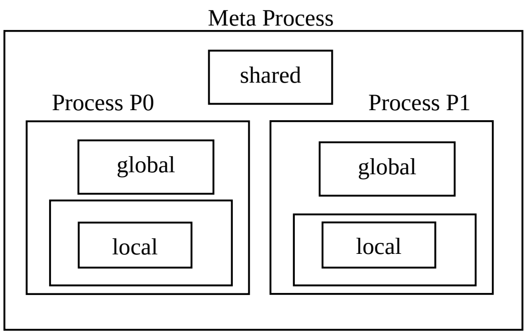

Figure 2 The hierarchical data scope

4.2 Distributed Shared Data Mapping

To realize shared data associations with processes, the MpC supports data *mapping specifiers*. The *mapping specifier* consists of 2 parts, *divide information* and *owner information*. The *mapping specifier* follows *shared*-prefixed variable names accompanied with :: . Figure 3 shows an informal definition of shared data with a *mapping specifier.* Figure 4 also shows some examples of shared data declarations. Using a *mapping specifier,* users can inform an underlying installation of hints to shared data mapping on processes. Typical mapping examples for 2d-array are shown in figure 5.

shared typespecifier* name *mapping specifier
***shared typespecifier* name [sn]..[si]..[s0] :: [dn]..[di]..[d0] (st, n)**

name[sn]..[si]..[s0]	variable name with optional dimensions
mapping specifier	*:: divide information owner information*
divide information	[dn]..[di]..[d0] optional
di	the number of division in i'th dimension (default: 1, not divide)
owner information	(st,n) optional
st	starting process id for mapping (default: arbitrary process)
n	the number of processes for cyclic mapping (default: all processes in a Meta Process , NPROCS)

Figure 3 An informal definition of shared data declaration

shared int x::(s); scalar data mapped to process st
shared double y[M]; the whole array is mapped to an arbitrary process
shared float z[M]::[n] (st); 1d-array is divided into n blocks and each block is mapped to each process from st to st+n-1
shared int q[M][L][N]::[ ][n][ ] (0,n) ; 3d-array is divided into n layers in y axis and mapped to processes from 0 to n-1 individualy

Figure 4 Shared data declaration examples

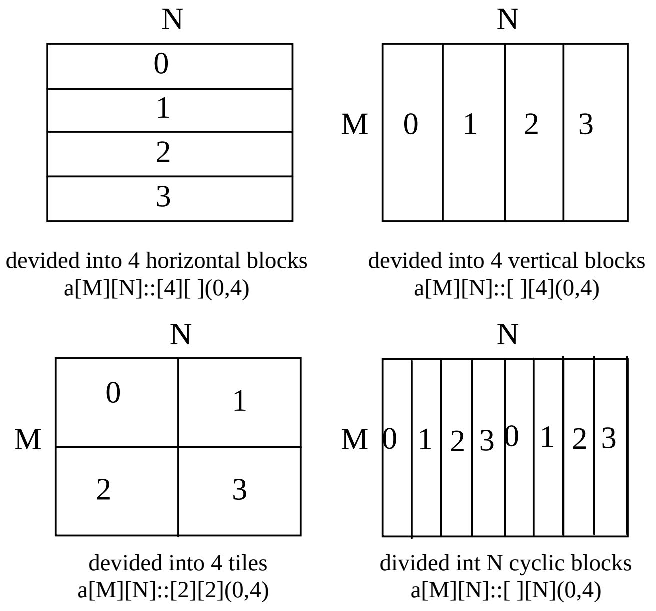

Figure 5 Typical shared 2d-array data mapping examples

As parameters for the *owner information* and the *divide information*, the MpC allows both ordinary constants and runtime constants called *MpC constants*, which are **NPROCS** and **MYPID**. The **NPROCS** is the number of processes that consist of a Meta Process in execution. The **MYPID** is a unique id number of process that uses **MYPID** in runtime. The values of the MpC constants cannot be determined in compile time, but they are fixed when a Meta Process execution starts and they are not changed during its execution in the current version.

4.3 MpC Standard Library Functions

The MpC has standardized MpC library functions, such as barrier, lock/unlock, condition signal/wait, Meta Process initialize/finalize, shared data dynamic allocation, etc. These standard function calls are translated into underlying runtime system's library calls.

5. MpC Compiler

Our MpC compiler for SDSMs has 2 components: the MpC-to-C translator and the Meta Process runtime module builder shown in Figure 6. In the first phase, the MpC compiler preprocesses and translates MpC program files into ANSI-C code files, with all MpC parallel APIs converted into calls of underlying runtime system's library. Users can select a target runtime system as a compiler parameter. It enables us to execute MpC programs on representative SDSMs. The translator also produces a *shared data information file* per program module, which is used in the second phase. The translated C codes are then compiled using a target machine's C compiler and linked to a program module for each process of a Meta Process.

In the second phase, the MpC compiler checks the consistency and validity of shared data usage among constituent processes, using *shared data information files* and a *Meta Process description file.* The *Meta Process description file* specifies an individual program module name and its arguments for each process. For a SPMD type execution, the *Meta Process description file* includes a single program module name, its arguments set and the number of parallelism. If no inconsistency is found, it builds a *Meta Process execution file* that is tailored for the selected runtime systems. Depending on the runtime systems, the actual content of the *Meta Process execution file* becomes different, but it is usually implemented as a script file.

6. Implementation for SDSMs

We have already developed shared-prefixed data declarations with distributed shared data mapping for our SDSM, SMS [20]. The SMS is a user-level SDSM system that runs on most Unix. It provides a global shared address space on top of physically distributed memories. It is a pege-based DSM and employs newly proposed memory consistency model, IBEC (Implicit Binding Entry Consistency) [9]. In execution, a *Meta Process execution file*, program modules and a *processor file* are used on the SMS as shown in Figure 7. Mapping to logical processes to physical computing nodes are done in runtime using a *processor file.* The *processor file* includes available host names or node names for the Meta Process. For the other user-level SDSMs such as the TreadMarks and the JIAJIA, the MpC compiler can translate MpC programs to C codes with their native library function

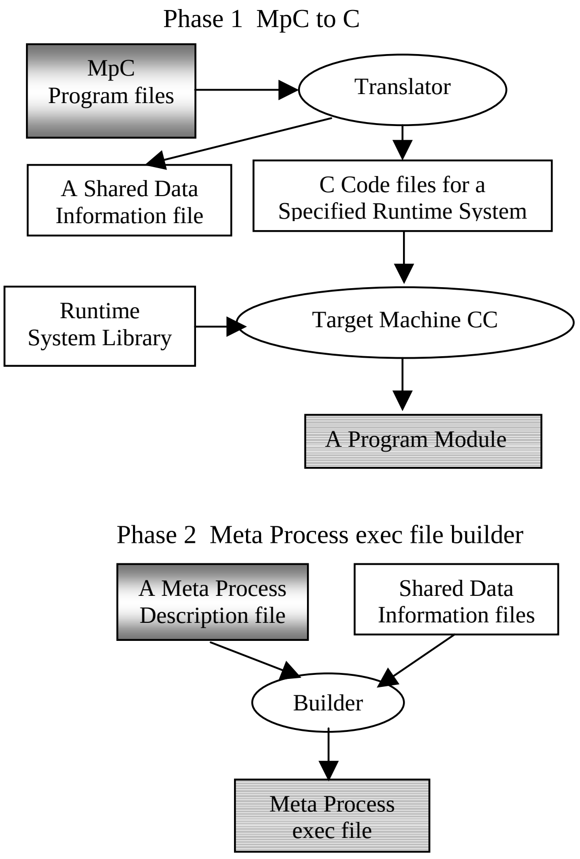
Phase 1 MpC to C
MpC Program files
Translator
A Shared Data Information file
C Code files for a Specified Runtime System
Runtime System Library
Target Machine CC
A Program Module
Phase 2 Meta Process exec file builder
A Meta Process Description file
Shared Data Information files
Builder
Meta Process exec file

Figure 6 The structure of MpC Compiler for SDSMs

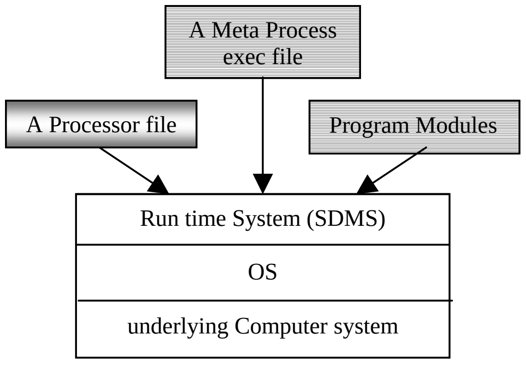

Figure 7 Meta Process runtime environment for user-level SDSMs

calls. If one of other SDSMs is used as an underlying runtime system, some constraints may occur for MpC programs and Meta Process. In some SDSMs, a Meta Process execution may be limited to SPMD style only. Other systems may support no condition signal/wait, or they may have only insufficient data layout APIs to reflect mapping specifiers in MpC programs. Actually, the most of the existing programs written for such SDSMs use basic core of MpC APIs, and extending the SDSMs to support such functions is not so difficult.

7. Conclusion

The Meta Process model, a new distributed shared memory programming model, introduces *shared* data and distinguishes process-local and process-shared data clearly with a scope concept. Its policy is based on the explicit parallelism description. It eliminates the implicit parallelism by systems, which is often used in the extended SMM and usually causes performance degradations. The Meta Process model enables us to get better performance and a diversity of parallel programs, not only SPMD programs like OpenMP, HPF and UPC, but also irregular and asymmetric programs. The Meta Process model provides us both flexible parallelism description of the MPM and good program readability/writability of the SMM.

The Meta Process model is similar to the pthread model, if a pthread is replaced with a process. Though, the pthreads share resources, such as files and sockets, etc. To implement our model on ordinary computer clusters with a general OS using threads instead of processes, some global migratory file descriptors etc. are required. So we implement our model with processes.

Other related works includes the UPC. The UPC is also a language for the distributed shared memory model, but its target is only a SPMD program like OpenMP. It incorporates parallel statements, i.e. forall, etc., and synchronizing as its language level, and employs two memory consistency models, strict/relaxed. UPC differs from MpC in data mapping API and the treatment of pointers.

8. Acknowledgement

I would like to thank Professor Hajime Iizuka of Seikei University for his encouragement and valuable advice. I also thank Shingo Katano for creative discussion and his aid in implementation of MpC preprocessor.

References

[1] PVM http://www.epm.ornl.gov/pvm/
[2] MPI http://www.unix.mcs.anl.gov/mpi/
[3] OpenMP http://www.openmp.org
[4] David R.Butenhof, *Programming with POSIX Threads*, Addison-Wesley,1997
[5] H. Jin, M. Frumkin and J. Yan, The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance: *NAS Technical Report NAS-99-011*,1999
[6] D. Takahashi, M.Sato, T.Boku, Performance Evaluation of the Hitachi SR8000 Using OpemMP Benchmarks:Workshop on OpenMP, *Int'l Workshop on OpenMP (WOMPEI'02),* 2002
[7] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems: *Proc. of the Winter 94 Usenix Conf.*, 1994, 115-131.
[8] Weiwu Hu, Weisong Shi, Zhimin Tang, JIAJIA: An SVM System Based on A New Cache Coherence Protocol: *Proc. of the High Performance Computing and Networking (HPCN'99),* LNCS 1593, 1999, 463-472.
[9] H.Midorikawa, U.Ohashi, H.Iizuka, The Design and Implementation of User-Level Software Distributed Shared Memory System: SMS Implicit Binding Entry Consistency Model -: *Proc. of IEEE Pacific Rim Conf. on Communications Computers and Signal Processing,* 2001, 299-302. http://sirius.is.seikei.ac.jp/~midori/paper/rim01.pdf
[10] S.V.Adve, K.Gharachorloo,: Shared Memory Consistency Models: A Tutorial, *IEEE Computer vol.29,no.12*, 1996, 66-76.
[11] P. Keleher, A.L. Cox, and W. Zwaenepoel, Lazy Release Consistency for Software Distributed Shared Memory: *Proc. of the 19th Symp. on Computer Architecture,* 1992, 13-21.
[12] Liviu Iftode, Jaswinder Pal Singh and Kai Li., Scope Consistency: A Bridge between Release Consistency and Entry Consistency: *Theory of Computing Systems*, *31,* 1998, 451-473
[13] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann, Towards OpenMP execution on software distributed shared memory systems, *Int'l Workshop on OpenMP(WOMPEI'02),* 2002.
[14] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson, and C. Offner, Extending OpenMP for NUMA Machines. *Proc. of the IEEE/ACM Supercomputing 2000 (SC2000),* 2000.
[15] V. Schuster and D. Miles. Distributed OpenMP, Extensions to OpenMP for SMP Clusters, *Proc. of the Workshop on OpenMP (WOMPAT2000),* 2000.
[16] High Performance Fortran Language Specification High Performance Fortran Forum January 31, 1997 , Version 2.0, http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20/index.html
[17] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren., Introduction to UPC and Language Specification: *CCS-TR-99-157, IDA Center for Computing Sciences*, 1999
[18] T.El Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study: *Proc. Super computing2002 (SC2002),* 2002
[19] UPC http://upc.gwu.edu/
[20] H.Midorikawa, S.Katano, H.Iizuka, The APIs for the shared data distributed mapping on software DSM SMS: *Procs. of FIT2002 vol. B-42*, 2002, 171-172 (in Japanese)