Meta Process Model and its Portable Parallel Programming Interface MpC

Hiroko Midorikawa
Department of Information Sciences
Seikei University
3-3-1 Kichijouji kita-machi, Musashino-shi, Tokyo, 180-8633, Japan
midori@is.seikei.ac.jp

Abstract

This paper proposes a new portable parallel programming interface MpC, Meta process C, for Meta Process Model. The Meta Process Model is a parallel programming padadigm based on a hierarchical shared memory model and an explicit description of parallelism On these points, this model is different from either the strict Shared Memory Model (SMM) or the Message Passing Model (MPM). The Meta Process Model introduces *shared* data that can be accessed by all processes in one Meta Process and distinguishes process-local and process-shared data explicitly with a hierarchical data scope.

A programmer describes process interactions explicitly using shared data accesses and synchronizing operations, such as a barrier and a lock/unlock. So it enables us to write a wide variety of parallel programs, not only SPMD and data parallel applications written in OpenMP and HPF but also asymmetric and asynchronous MPMD applications written in MPI. The Meta Process Model provides us with both flexible parallelism description of the MPM and good program readability/writability of the SMM. Its target machines include clusters as well as shared memory parallel machines. Some execution results of MpC programs on clusters and shared memory machines are shown, and they are compared with OpenMP and UPC programs. It also proves good portability of MpC programs and MpC compilers because they use user-level SDSM library, pthread library and gcc, which are available for various OS's and architectures.

Key Word
Programming Model, Parallel Language, Cluster computing, SDSM

1 Introduction

The two most representative parallel programming models are the message passing (MPM) and the shared memory (SMM) models. In MPM, MPI has greatly contributed to portability and popularization of parallel programs. On the other hand, the SMM has an advangtage in easy extention 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[1][2]. However, a typical SMM environment, such as pthread and OpenMP, is not available to or not so efficient for clusters. Clusters are important platforms for parallel processing because of their good cost effectiveness.

Usual clusters have relatively low bandwidth communication links between computing nodes as compared with their CPU performance. They are categorized into a distributed memory system.

Another effort to derive high performance from clusters with the SMM instead of the MPM is the development of software distributed shared memory systems (SDSMs), such as TreadMarks[3], JIAJIA[4] and SMS[5]. These systems use some type of the relaxed memory consistency models [6] to reduce communication overhead between computing nodes. In SDSM programs, programmers must describe the parallelism explicitly using shared data access and synchronizing operations like a barrier and a lock/unlock. This explicit parallelism description is similar as in pthread programming.

Using OpenMP on clusters, which means using the SMM on a distributed memory system, usually causes more performance degradation than using SDSMs directly[7]. It is caused by the fact that the most of OpenMP implementation for clusters use a SDSM as an underlying system. Without special hardware support to access remote memory, it is difficult to derive high performance on clusters comparable to the performance on SMP machines and NUMA type shared memory machines such as SGI origin.

The OpenMP is originally designed based on fork-join parallel thread model for shared memory systems. Since it is not for distributed memory systems, it places much importance on easy extention from sequential programs. Its API tends to hide details 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.

To overcome such overhead, some dialects of OpenMP add extended facilities to specify data mapping[8] [9]. Others explore a hybrid model with OpenMP and MPI[10][11]. Though, the more elaborated facilities are incorporated in the extended OpenMP, the more its simplicity and integrity of SMM seems to be deteriorated. It seems that system's implicit processing, such as synchronizations and data copying between local from/to shared data, and user-specified explicit ones 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 of underlying memory configurations, actually it is difficult to achieve high performance on clusters unless programmers pay attention whether data are local or global, private or shared. Currently, 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 interface MpC, Meta Process C, for an API of Meta Process Model. The Meta Process Model [12] is based on a hierarchical shared memory model and employs an explicit parallelism description paradigm. The motivation to the model is to improve the performance in realistic way with keeping a consistent SMM interface, instead of a hybrid API of MPM and SMM like an MPI/OpenMP, even on distributed memory systems.

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-shared data from process-local one with a hierarchical data scope. The reason why the execution entity of the model is called a process, not a thread, is that the processes in one Meta Process have separate address spaces for process-internal data, global data, but they share an additional address space for process-shared data. Shared data are maintained by a relaxed memory consistency model. Basically processes do not share resources, such as files and memory, and threads share all resources. So we chose a process model instead of thread model even if the model implementation uses threads for actual execution.

A programmer describes process interactions explicitly using shared data accesses and synchronizing operations, such as a barrier and a lock/unlock. So it enables us to write a wide variety of parallel programs, not only SPMD and data parallel applications often found in OpenMP and HPF programs, but also asymmetric and asynchronous MPMD applications usually written in MPI. The Meta Process Model provides us flexible parallelism description of the MPM. It also keeps good program readability/writability of SMM, because it hides communication details between computing nodes. The MpC programs have a good portability. Its target machines include not only clusters but also shared memory parallel machines. This is because the implementation of the Meta Process Model uses only user-level software DSM library or pthread library. The MpC compiler uses gcc which is widely available in various machine architectures. This is a reason why MpC programs and the compiler can be widely accepted and portable.

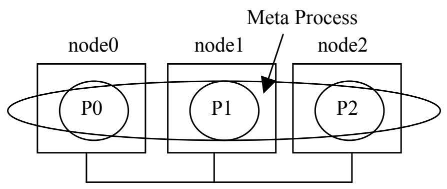

Figure 1: Meta Process and its constituent processes

2 Meta Process Model

Usually, parallel processing on clusters requires the cooperation of multiple processes on each computing node as shown in Figure 1. The Meta Process model treats such cooperative processes distributed over the nodes as one execution entity, called Meta Process. Each process in Meta Process is a traditional process, which is identified by the node's OS and possesses some resources. Although the Meta Process is not identified by general OS's, it is identified as a user's single execution entity by some system software, such as SDSMs. As far as a user is concerned, it is a single application program execution entity on clusters. So the model can be easily extended for shared memory parallel computers. It is similar that the MPI is available both for shared and distributed memory machines.

2.1 Single Shared Address Space for shared data

This model introduces two hierarchical execution entities, process and Meta Process, and it also provides two storage types of data, which are available in the each scope level. The first type is local data in a process, the second one is global data shared among the processes in a single Meta Process. For newly introduced shared data, the same address used in each process refers to the same shared data. So users need no message passing statements in their programs. Runtime data scopes are shown in Figure 2. The lifetime and the scope for process-local data are controlled by the description of C programs, in accordance with a declaration of automatic, global or static variables. Other processes can't access process-local data directly. Only *shared* data, which newly defined in the model, are accessible from any process in the Meta Process.

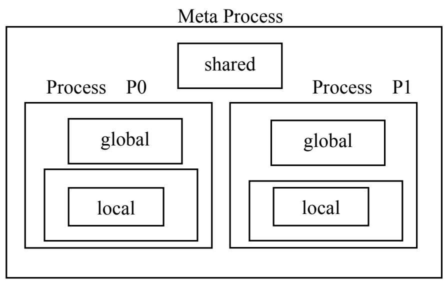

Figure 2: Shared variables and the hierarchical data scope in execution time

2.2 Relaxed Memory Consistency

This model assumes that shared data are maintained by a relaxed memory consistency model[6], often used in SDSMs. The premise gives an advantage in performance for distributed memory machines such as clusters and even for NUMA, and no harm for shared memory machines.

2.3 Shared Data Layout

This model provides an API for associating shared data with a process in the Meta Process. Specifying the most closely related process that accesses the shared data frequently enables to achieve good performance on distributed memory systems. If such explicit association does not exist in an application or a user has no knowledge of the application nature, the association specifiers can be omitted. In that case, an implementation of this model associates the shared data to an arbitrary process depending on the implementation systemn's policy. The association specifier is a hint for the underlying implementation, like *register* used in C language. For array data, flexible cyclic data allocations to a sub group of processes are supported. They include line, band, tile, and cube mapping, etc. A user can declare shared data with suitable association specifiers for their application. Both static declarations and dynamic shared data allocations are possible.

2.4 Good Portability

This model is portable to both distributed memory systems and shared memory systems. Typical SDSMs and pthread have similar APIs, such as lock and unlock. Incorporating these similar APIs into the MpC API, it is possible to port MpC programs from a SDSM to other different SDSM or from a shared memory system to a SDSM.

3 MpC Language

MpC is a portable parallel programming interface of the Meta Process model for DSMs and shared memory systems. It is a parallel extension of ANSI C.

3.1 Shared Storage Class Specifier

In the MpC, *shared* is newly added to the storage class specifier of ANSI C. The data prefixed with *shared* must be declared outside of functions in at least one of the program files that are linked into a single executable. The scope of the shared data is the whole Meta Process, and the data are visible and accessible from all processes in the Meta process. 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 from external to internal at runtime is shared, global and local, in order, as shown in Figure 2. If the same variable name is used for different scope level data, the most internal variable data are effective for that name. In our implementation of MpC for clusters, shared data are allocated dynamically when a Meta Process is started and deallocated at the time of its exit. So the lifetime of shared data can be considered to be the same as the static data in C.

The MpC treats pointers in the same way of original C, where a storage type of the content referenced by a pointer is not cared . It ignores whether the content pointed by the pointer is shared or local. It only takes account of data type specifiers. The MpC simplifies pointer expressions. For example, only 2 expressions are allowed for integer pointers.

1. a local pointer variable which points to integer:
   int *p1;

2. a shared pointer variable which points to integer:
   shared int *p2;

3.2 Distributed Shared Data Mapping

To associate shared data with processes, the MpC supports data mapping specifiers. The mapping specifier consists of two parts, divide information and owner information. The mapping specifier follows *shared* prefixed variable names, accompanied with ::. Using mapping specifiers, users can inform an underlying implementation of hints to map the shared data distributedly over processes. Typical mapping examples for 2d-arrays and 3d-arrays are shown in Figure 3. The MpC data mapping interface is more straightforward and flexible than one of UPC described in section 3.5.

3.3 MpC Constants

The MpC allows runtime constants called MpC constants as well as ordinary constants to be used in distributed shared data mapping specifiers and othre program codes. The MpC constants include two constants, 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 the each process in the Meta Process. The value of MYPID is from 0 to NPROCS-1. The values of

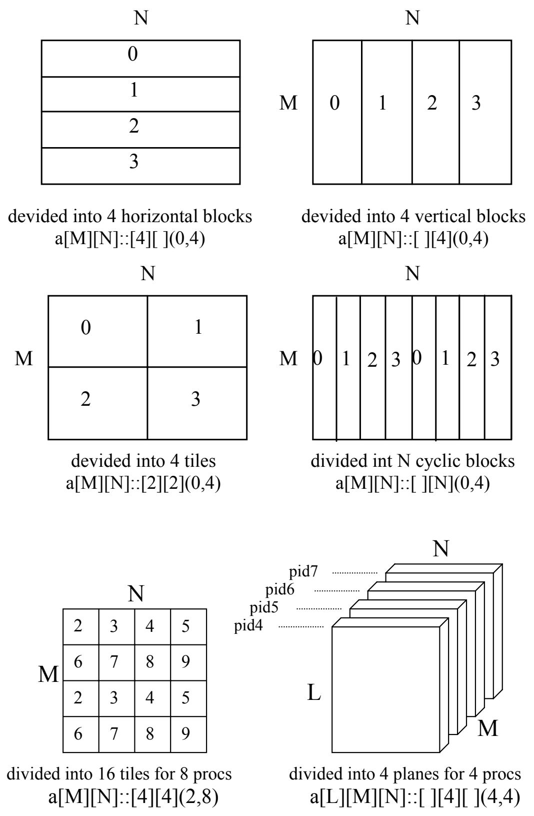

Figure 3: Shared variable distributed mapping examples

the MpC constants can't be determined in compile time, but they are resolved and fixed when a Meta Process execution starts.

3.4 MpC Standard Library Functions

The MpC has standard MpC library functions, such as barrier, lock/unlock, condition signal/wait, Meta Process initialize/finalize and shared data dynamic allocation, shown in Table1. These standard function calls are translated into underlying runtime system's library function calls.

Figure 4 shows a MpC sample program that uses basic MpC functions. It is a simple SPMD program. It has no message passing statements.

3.5 UPC and MpC

One of the other related works is the UPC[13]. The UPC is a language for the distributed shared memory model, but its target is only SPMD programs as in OpenMP. It incorporates parallel statements and synchronizations into its language level and supports two memory consistency models, strict/relaxed. It differs MpC in data mapping I/F and pointers etc.

	MpC Library Functions
Initialization	mpc_init(int argc, char *argv)
Termination	mpc_exit(int value)
Error Termination	mpc_err(int value, char *msg)
Barrier	mpc_barrier(int value)
Lock	mpc_lock(int lockid)
Unlock	mpc_unlock(int lockid)
Condition signal	mpc_cond_signal(int condid)
Cond. broadcast	mpc_cond_broadcast(int condid)
Condition wait	mpc_cond_wait(int condid, int lockid)
Data allocation	void *mpc_alloc(size)
Data allocation	void *mpc_alloc(char *declare)

Table 1: MpC Standard library functions

A *shared* prefix is also introduced in UPC but it is implemented as the type qualifiers of ANSI C. As a result, there are 4 kinds of integer pointers as follows.

1. a local pointer variable which points to local integer:
   int *q1;
2. a local pointer variable which points to shared integer:
   shared int *q2;
3. a shared pointer variable which points to local integer:
   int *shared q3;
4. a shared pointer variable which points to shared integer:
   shared int *shared q4;

However, it is considered that such pointer distinction is not necessary for programmers in actual use and only gives unnecessary complexity to a compiler and programmers. Dynamically content-changing pointers by casting and multiple indirected pointers are actually impossible to determined whether actual data are shared or not. Moreover, most of SDSMs support no mechanisms to improve memory access performance using such information. So its effectiveness seems to be limited in actual programs. It is more natural to regard *shared* as one of the storage classes instead of the data type specifier as in UPC.

The distributed mapping declaration for shared data in UPC is also different from one in MpC. The simplest declarations in UPC and MpC are shown in Figure 5. Originally UPC and the former base language Split-C is tailored to finer grain parallel machines than clusters and its main target machines seem to be shared memory machines. So its default declaration of shared data array employs one element cyclic mapping over all threads as shown in Figure 5(b). It is extremely inefficient for clusters. Moreover the default data type in UPC is *shared*, because it employs a thread model.

In UPC, the number of threads available for data mapping must be fixed number, all threads at runtime, because

```
#include <stdio.h>
#include <mpc.h>
#definr M 1024
#define N 2048

shared double matrix[M][N]::[NPROCS][ ];
shared double sum::(0);

int main(int argc, char **argv)
{
      FILE fp;
      double mysum=0;
      int start, end, i, j;

      mpc_init(argc, argv);
      if(MYPID == 0){
            fp=fopen("initial.dat", "r");
            for(i=0; i<M; i++)
                  for(j=0; j<N; j++) fscanf(fp,"%lf", &matirx[i][j]);
            sum = 0;
      }
      mpc_barrier(0);

      start = M/NPROCS*MYPID;
      end = start+M/NPROCS;
      for(i=start; i<end; i++)
            for(j=0; j<N; j++) mysum += process(matrix[i][j]);

      mpc_lock(0);
         sum += mysum;
      mpc_unlock(0);

      mpc_barrier(0);
      if(MYPID == 0)    printf("Result=%f¥n",sum);
      mpc_exit(0);
}
```

Figure 4: A MpC program sample

```
MpC:   shared int mat[M][N];
UPC:   shared [ ] int mat[M][N];
          (a) no division allocated in proc0

MpC:   shared int mat[M][N]::[M][N];
UPC:   shared int mat[M][N];
          (b) 1 element cyclic allocation

MpC:   shared int mat[M][N]::[NPROCS][ ];
UPC:   shared [M*N/THREADS] int mat[M][N];
          (c) horizontal division

MpC:   shared int mat[M][N]::[ ][NPROCS];
UPC:   shared [N/THREADS] int mat[M][N];
          (d) vertical division
```

Figure 5: Distributed mapping difference between UPC and MpC

	MPC	UPC
Process specification for data mapping	start proc: arbitrary num of procs: arbitrary	array data start proc: fixed 0 num of procs: fixed all procs scalar data fixed 0
Flexibility for data mapping	Enough flexible	Less Flexible
Memory consistency model	relaxed	strict / relaxed
Pointer category	2 types	4 types
Syncro. functions	Library	Construct, Library
Parallel constructs	no	forall, sizeof, threadof etc.
Target program type	SPMD / MPMD	SPMD

Table 2: Comparison between UPC and MpC

of its SPMD model. In the case of scalar data, they must be mapped to the first thread fixedly. In MpC, on the other hand, arbitrary number and subset of processes is available for various types of data mapping. The comparison between UPC and MpC is summarized in Table2.

4 MpC Compiler

Our MpC compiler for SDSMs consists of 2 phases: a MpC-to-C translator and a Meta Process runtime module builder. In the first phase, the MpC compiler preprocesses and translates MpC programs into C programs, converting all MpC parallel APIs into underlying runtime system's library function calls. Users can specify the underlying runtime system, pthread or one of the SDSMs, in their compiler parameter.

First, the MpC compiler calls a C preprocessor to process header files and macros. After that it checks hierarchical data scope to introduce a *shared* data type, renames several variable names according to standard C grammar, and inserts necessary program codes to the original program.

In the second phase, it calls a native C compiler to compile the translated C programs into binary codes for a target machine. The compiler links it with runtime system library, a SDSM or pthread. Our MpC compiler assumes that a target C compiler is gcc, because it is widely used for compiling programs on various types of machine architecture. In our MpC compiler, the icc is also possible to use for x86 machine codes generation. As a result, our MpC compiler is very portable.

So MpC programs can be executed with a SDSM library on clusters and with pthread library on shared memory machines.

5 Meta Process Model for clusters

5.1 Implementation for clusters

The Meta Process Model implementation for clusters uses representative user-level software distributed shared memory (SDSM) system, such as TreadMarks[3], JIAJIA[4] and SMS[5]. Users can specify one of the SDSMs as an underlying system in compile time. Each SDSM's API has a little difference in its style and its function, but their basic functions, such as barrier, lock/unlock and alloc, are almost the same. So the most of the existing SDSM programs can be translated into other SDSM's programs with a little modification.

Moreover these SDSMs have high portability because of user-level software implementation. TreadMarks runs on

Progs	Parameters	Data Size	Barrier /proc	Lock /proc
ep	M=28,MK=10	44B	2	1
tsp	19cities(19b)	100MB	4	75-122
lu	2048 x 2048 double, 32bloks	34MB	135	0
mm	2048 x 2048 double	96MB	3	0

Table 3: Benchmark programs parameters

CPU	Intel Pentium III -S 1.13GHZ
Memory	512MB
Network	Intel Proc 1000T 3Com SuperStack3 switch
OS	RedhatLinux 7.1.2 kernel 2.4.7.10

gcc version 2.96 -O3
SMS 0.4.16
JIAJIA 2.2
TreadMarks 1.0.3.2

Table 4: Experiment environment

IBM, DEC, SUN, HP, x86 (running FreeBSD or Linux) and SGI hardware [14]. JIAJIA runs on SUN SPARC stations, IBM SP2, X86-based Linux 2.0, etc[15]. The MpC programs linked with such SDSMs' libraries can be executable on various types of OSs and architectures. This is a reason why MpC programs are highly portable.

5.2 Experimental Results on SDSMs

The Meta process performance evaluation for several programs on three SDSMs, TreadMarks, JIAJIA and SMS, is investigated. Benchmark programs for this experiment include ep from NPB, lu from Splash benchmark, tsp from TreadMarks distribution and mm, matrix multiply, from JIAJIA distribution. These benchmark programs are rewritten in MpC from individual SDSM programs. Their program parameters are shown in Table3. The experiment hardware and software environments are shown in Table4. Figure 6,7,8,9 show speedup of the benchmark programs on TreadMarks, JIAJIA and SMS.

MpC programs can be executed on three different SDSMs with no modification. Compute-dominate applications, such as ep and lu have good scalability. The tsp shows performance difference depending on underlying SDSMs, because it is distinguished from other programs by having many lock/unlock calls.

The MpC API is similar to these SDSMs' APIs, so MpC program execution suffers from little additional overhead to direct SDSM program execution.

5.3 The Comparison of MpC and OpenMP

Performance comparison of MpC and OpenMP programs on clusters was also examined. Omni OpenMP[16] on SCore5.6.1[17] and MpC on SMS were used for this measurement. The Omni OpenMP is implemented on

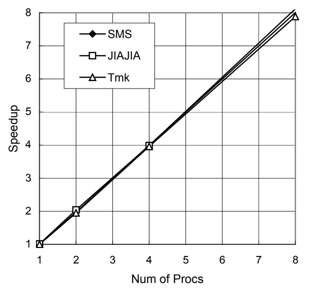

Figure 7: Speedup of lu on SDSM

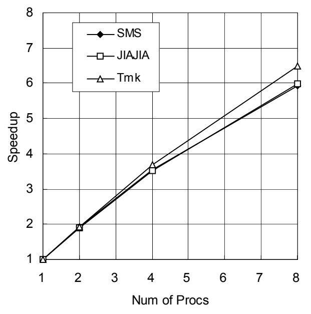

Figure 8: Speedup of mm on SDSM

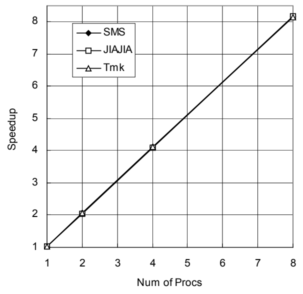

Figure 6: Speedup of ep on SDSM

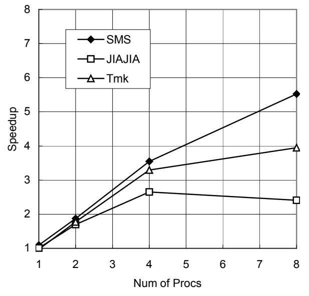

Figure 9: Speedup of tsp on SDSM

(sec)

Program Code		MpC	OpenMP			SCASH	
Compiler		mpcc	omcc		gcc		
Network		giga	giga	myri		giga	myri
ep	1	11.57	14.45	14.45	13.99		
S class	2	5.79	7.69	7.28			
	4	2.90	4.57	3.71			
	8	1.45	2.76	1.95			
Floyd	1	66.65	65.21	65.12	66.80		
1024x1024	2	35.01	69.80	46.57			
	4	19.42	44.94	25.30			
	8	11.38	41.62	14.71			
laplace	1	3.36	3.18	3.24	3.26	3.79	3.83
1024x1024	2	2.06	2.73	2.18		2.65	2.33
50ite	4	1.10	1.61	1.21		1.84	1.25
	8	0.75	1.40	0.63		1.39	0.66
mandel	1	1.39	1.44	1.44	1.37		
1024x1024	2	0.87	0.84	0.79			
dynamic	4	0.54	0.50	0.42			
	8	0.43	0.35	0.23			
mm	1	12.18	13.50	14.41	12.32		
blocked	2	5.16	5.75	5.22			
1024x1024	4	3.22	3.87	2.92			
	8	2.16	3.60	2.20			
galaxy	1	8.52	9.02	9.02	8.75		
1000-body	2	4.32	2.98	2.47			
10steps	4	1.88	1.61	1.25			
100time	8	0.96	1.08	0.65			

Table 5: Performance in MpC and OpenMP

SCASH, a SDSM developed on the SCore. Our system is based on user-level software but the SCore requires OS kernel modification. Table5 shows execution times of 6 programs. They include floyed:minimum path search, laplace:Laplace transformation, mandel:Mandelbrot plot, mm:matrix multiplication and galaxy:n-body problem. All have regular program structure, data parallel with for loop, which is effective for OpenMP's SPMD model. The clusters have 8 single-CPU nodes. The Omni uses PM for gigabit Ethernet and Myrinet, and MpC uses UDP sockets for gigabit Ethernet.

In the case of gigabit Ethernet, MpC performance is better than Omni's for most of the programs. The floyd and the mm, where shared data size are large, the MpC's giga Ethernet performance is comparable or even better than the Omni's Myrinet performance.

6 Meta Process Model for Shared Memory Machines

6.1 Implementation for Shared Memory Machines

The implemnatation on shared memory machines uses pthread for the *processes* in the Meta Process, and global data in each *process* are translated to thread specific data. Only the *shared* data are shared among the threads. To make MpC programs run on shared memory machines, the MpC compiler translates MpC programs into pthread programs. As shown in Figure 10, the Meta Process is converted to a process and the *processes* in the Meta Process are converted to threads in a shared memory machine.

The MpC compiler changes original main routines into a subroutine for each thread and adds a new main routine, which creates threads that execute that subroutine, and wait the termination of all threads. It also translates global variable data in the original MpC programs into thread specific data and initializes locks and condition variables. Actual implementation of hierarchical data and execution entities in the Meta Process Model for shared memory machines and their counterparts for clusters are shown in Table6.

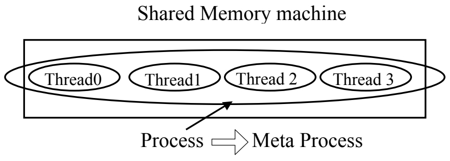

Figure 10: Meta Process Model for shared memory machines

Meta Process Model	SDSM installation for cluster computers	pthread installation for shared memory machines
a Application (Meta Process)	A Group of Processes	A Process
Unit of parallel execution entity	A Process	A Thread

Data Type in Meta Process Model	SDSM installation for cluster computers	pthread installation for shared memory machines
shared	Process shared data (using SDSM functions)	Thread shared data (Global variables)
global	Process local data (Global variables)	Thread local data (Local variables)
local	Block local data (Local variables)	Block local data (Local variables)

Table 6: The implementation of Meta Process Model for SDSM and pthread

6.2 Experimental Results on Shared Memory Machines

Some MpC benchmark programs, such as ep, mm, galaxy and mandel are run on SMP machines as pthread programs. Table7 shows execution times of MpC programs and gnuUPC[18] programs on the SMP machine (Pentium3 2CPUs LINUX). It also shows MpC program results for a shared memory computing server (IBM rs6000 4CPUs AIX5.2) and pc cluster (Pentium3 LINUX 8 nodes) with SMS.

It shows that MpC programs are ported to other type of machines with no modification and the performance of these programs are good for each platform.

The execution time of UPC programs such as galaxy and mm are extremely slow. These programs use large amount of shared data and their memory-access/computation ratio are bigger than the other pro-

(sec)

programs	Num of Procs	MpC	MpC	MpC	gnuUPC 3.2.3.5
		pthread smp 2CPUs Pentium3 LINUX2.4.20 -6smp RedHat9	pthread rs6000 4CPUs AIX5.2	SMS GigaEther pc cluster LINUX 2.4.20-6 RedHat9	smp 2CPUs Pentium3 LINUX2.4.20-6smp RedHat9
	1	10.82	10.45	11.19	10.15
ep	2	5.54	5.23	6.09	5.09
S class	4		2.65	2.56	
	8			1.28	mapping 1ele
galaxy	1	9.02	8.81	8.17	64.05/62.96
1000body	2	4.52	4.42	4.53	32.89/31.5
10 steps	4		2.24	2.53	
100time	8			1.63	mapping div/0proc
mandel d	1	1.35	1.03	1.39	1.56/1.47
1024x1024	2	0.68	0.53	0.87	0.79/0.75
0.3<x<0.4	4		0.38	0.54	
0.5<y<0.6	8			0.43	mapping 1ele/0proc
mm512	1	0.77	0.61	0.80	15.2/15.2/35.26/9.21
512x512	2	0.40	0.31	0.47	8.16/7.69/17.83/4.64
double array	4		0.17	0.31	mapping
blocked	8			0.29	vb/hb/1ele/0proc
mm1024	1	6.18	4.9	6.90	132.63
1024x1024	2	3.17	2.47	3.47	66.35
double array	4		1.49	2.04	
blocked	8			1.57	mapping 0proc

Table 7: Performance in MpC and UPC

grams. Various data mapping specifications, hb:horizontal band mapping, vb:vertical band mapping, 1ele:one element cyclic mapping, which is UPC default, and 0proc:no data distributed mapping/allocated in THREAD0, are tried for the mm UPC program, but their results are much slower than MpC one. The details of UPC implementation are not clear, but some of the mechanism of UPC implementation or the gnuUPC compiler may not fit to the SMP machines. Current gnuUPC compiler for x86 architecture targets for only SMP machines and it is not available for pc clusters. Other programs, ep and mandel, are comparable performance to the MpC program.

7 Conclusion

This paper proposes a new portable parallel programming interface MpC as the API of Meta Process Model. The design of MpC is presented and its performance on shared memory machines and clusters has been examined. It shows the good portability of MpC programs and better performance than existing UPC programs and OpenMP programs on gigabit Ethernet. Moreover, MpC programs have good readability/writability than MPI programs. It shows that MpC has a possibility of becoming one of the alternatives for parallel programming interfaces/languages.

Acknowledgements

The author wishes to thank Professor Hajime Iizuka for his valuable advice and Shingo Katano and Yoshihito Watanabe for their contribution to the development of the MpC compiler.

References

[1] H. Jin, M. Frumkin and J. Yan: The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance, NAS Technical Report NAS-99-011, 1999.

[2] 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.

[3] 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., pp.115-131, 1994.

[4] 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, pp.463-472, 1999.

[5] 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, 299-302, 2001.

[6] S.V.Adve, K.Gharachorloo : Shared Memory Consistency Models: A Tutorial, IEEE Computer vol.29, no.12, pp.66-76, 1996.

[7] 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.

[8] 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.

[9] V. Schuster and D. Miles. Distributed OpenMP, Extensions to OpenMP for SMP Clusters, Proc. of the Workshop on OpenMP (WOMPAT2000), 2000.

[10] Edmond Chow and David Hyson: Assessing Performance of Hybrid MPI/OpenMP programs on SMP Clusters, Tech.Report UCRL-JC-143957, Lawrence Livermore National Laboratoty, 2001.

[11] S.Dong and G.E.Karniadakis: Dual-Level Parallelism for Deterministic and Stochastic CFD Problems, Proc. of IEEE/ACM Super Computing 2002, 2002.

[12] H.Midorikawa : Meta Process Model: A New Distributed Shared Memeory programming Model, Proc. of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems, pp.295-300, 2003.

[13] 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.

[14] http://www.cs.rice.edu/ willy/TreadMarks/overview.html

[15] http://www.ict.ac.cn/chpc/dsm/index.html

[16] http://phase.hpcc.jp/Omni/home.html

[17] http://www.pccluster.org/index.html.en

[18] http://www.intrepid.com/upc/