混合式多處理器系統可程式度之改善

A Programmability Improving Scheme for Hybrid Multiprocessor Architectures

李良德
Liang-Teh Lee
大同大學資訊工程系
ltlee@ttu.edu.tw

張鴻源
Hung-Yuan Chang
大同大學資訊工程系
hychang@ntist.edu.tw

林秋旺
Chiu-Wang Lin
大同大學資訊工程系
g9406013@ms2.ttu.edu.tw

劉岡遠
Kang-Yuan Liu
大同大學資訊工程系
d9306005@ms2.ttu.edu.tw

潘昆祺
Kun-Chi Pan
大同大學資訊工程系
g9506039@ms2.ttu.edu.tw

摘要

多處理機系統的可程式度對於程式設計者而言，是個普遍面臨到的重要議題。使用多核心處理器的運算節點所連結而成的叢集電腦，是一個包含訊息傳遞與記憶體分享架構的混合式多處理機系統，而這樣的系統也將會隨著多核心處理器的流行越顯普遍。本篇論文提出了在混合式多處理機系統下的可程式度改善之方法。論文中提出了一套以 SUIF 為基礎的編譯系統，能將使用 C 程式語言所撰寫的循序式程式轉換成適用於混合式多處理機系統下進行平行處理的程式。而最後的實驗結果，也顯示出此套編譯系統的可運行性並能提升程式執行的效能。此篇論文也提出了一個加強型 MPICH 多重通訊協議裝置，可以增進此裝置在分享式記憶體通訊模式下的頻寬表現。
關鍵詞: 可程式度、迴圈轉換、混合式多處理器系統

Abstract

The programmability of a multiprocessor system is generally recognized to be the major issue confronting designers. A PC cluster with multi-core computing nodes to form a Hybrid Multiprocessor System which consists of both message passing and shared memory multiprocessing will become popular. The proposed SUIF-based compiler system with software methodologies for improving the programmability of a hybrid multiprocessor has been built to transform the sequential program which is written in C language to running in parallel. Moreover, an enhanced MPICH multi-protocol device has also been proposed, and it can improve the bandwidth when using the MPICH multi-protocol device with shared memory communication mode. The experimental results show that the proposed system is workable and has a better performance.
Keywords : *Programmability, Loop Transformation, Hybrid Multiprocessor Architecture*

1. Introduction

In scientific or commercial application is demand huge computing power to execute tasks. The multiprocessing architecture is the better solutions of improving the computing power for parallel computing. This system can operate tasks at the same time on difference process units. The classification of the multiprocessing architecture can be divided in accordance with different ways, such as the symmetry of the processor, the type of instruction and data streams and the processor coupling. However, it is not difficult to use this multiprocessing architecture to improve the computing power with the prevalence of the PC cluster [1] [2] [3]. In recent years, with the emergence of the multi-core processor constructed personal computer, the performance of the PC cluster can further be improved.

Multi-core processor (MCP), chip-level multiprocessing (CMP), has become more popular. There will be more and more PC clusters made up of CMP or SMP computers. The mixed model called 2-*level Hybrid Multiprocessing Architecture* (2LHMA). There is 2-level memory architecture, the one is the shared memory (SM) and the other one is distributed memory (DM). The most existing automatic parallelizing compilers are designed for SM, but the DSM libraries stilly have the problem of inefficiency [8] [9].

Different multiprocessing architectures have different methods to write parallel program. According to the memory architecture, the SM can adopt the *threading technique*, and the DM can use the *message passing technique*. There are two methods to write parallel programs with message passing technique. The one is the parallel program such as Fortran D [2] or HPF [3]. The other is the sequential program in the implementation mostly used MPI or PVM [6]. Nevertheless, this is a hindrance for general users, some the automatic parallelizing compilers have proposed to reduce the threshold, e.g. Polaris [7], PGI PGF77/PGCC and JAVAR-KAI and so forth, for the SM. The DSM library such as Bert 77, PARAGUIN/PARADIGM, PGI PGHPF, and VAST-HPF etc allows programmers to write parallel program with SM programming style on the DM.

In this paper, we propose a SUIF-based automatic parallelizing compiler system for handling the C language. A sequential program can be transformed to a parallelable one for running on HMA without considering the problem of what type of the memory system is. This hybrid memory model will use the message passing as the communication foundation, and each core in cluster is regarded as an independent processing unit. These units will exchange data by MPICH. In order to reduce the communication time, units that belong to the same computers can just use local memory for exchanging data by using the MPI shared-memory device. Therefore, an Enhanced MPI Multi-Protocol Device (EMPIMPD) proposed to handle the communication of inter-node and intra-node, respectively.

This paper is organized as follows: Section 2 is relative work to describe some knowledge of techniques. Our approach for constructing an automatic parallelizing compiler system will present in section 3. Section 4 introduces the environment of the experiment to set up and show the experimental results. Finally, concluding remarks and future direction are given in section 5.

2. Related Work

Loop Transformation (LT) means to restructure the loop of the source code. The actions of LT are unrolling the loop, iterations separated and regrouped, and mapping into processing units. LT can be regarded as a set of optimizations and have three purposes. First, it can increase the degree of the parallelism in a source code [11] [12]. Second, it can utilize the advantage of the locality concept effectively. Third, it can reduce the required communication or synchronous time [13]. However, we need to analyze the data dependence in the source code before restructuring in order to avoid something wrong due to data dependence.

The *Stanford University Intermediate Format version 2* (SUIF2) can be described as two major parts: the front-end and the back-end. The front-end consist of lexical analysis, syntax analysis, semantic analysis, and generation of the SUIF intermediate format file. In the back-end, the operations are code optimization and code generation. SUIF2 compiler system, first, gives full supports of handling the SUIF intermediate format file, next allows user to develop new modules or compiler passes according to their specific requirements, final supports in current programming languages such as FORTRAN, C, C++ and Java.

3. Design and Implementation

3.1 System Model

The concepts of proposed automatic parallelizing compiler system designed for high-level programming language as diagramed in Figure 1 are listed as follows:

In order to make easy to analyze the program, system restructures the sequential program from input. Then system will analyze the program and look for the data dependence. Next system will optimize the program and increase the degree of parallelism. Repeat the step 2 and 3 until the program has already reached the maximum degree of parallelism. Final system inserts MPI function calls into a program and outputs the parallelable one.

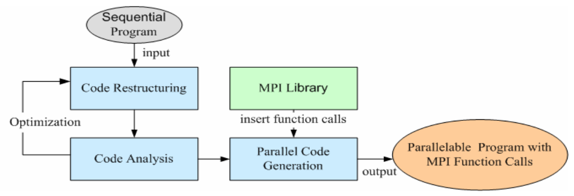


Figure 1: The concept of the proposed automatic parallelizing compiler system

A sequential C program can be compiled for running on a hybrid multiprocessing system. The proposed compiler system shows in Figure 2 that includes *front-end* and *back-end* passes. The front-end transforms the input program into the SUIF intermediate format file through the C front-end compiler, and then sends the SUIF intermediate format file to the back-end. The back-end can further be separated into three passes. The first pass is "*code restructuring and analysis"*. In order to maximize the degree of parallelism, the loop is restructured for finding and reducing data dependence. The second pass is "*parallel code generation"* that determines the degree of parallelism of loop structures, and then decomposes the computation units and maps these computation units into each processing unit. It gives program the ability of executing in parallel through inserting the MPICH function calls. The final pass is "*C back-end compiler"*, it transforms the SUIF intermediate format file into a parallelable C program with MPICH function calls.

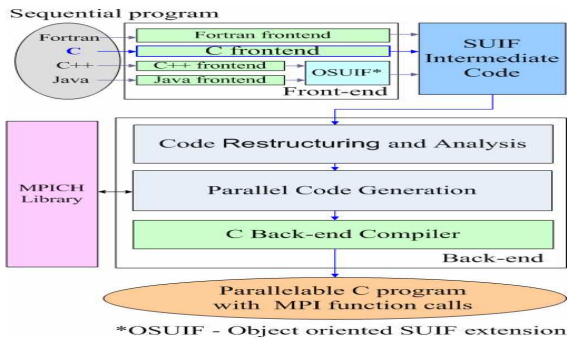


Figure 2: The automatic parallelizing compiler system based on the SUIF2 compiler system

3.2 Enhanced MPICH Multi-protocol Device

Proposed HMA needs an MPICH multi-protocol device to handle inter-node communication and intra-node communication. Two or more processors belong to different level computers can exchange data via the TCP/IP device. Processors that belong to the same computer can use the shared-memory device to exchange data. The MPICH provides a multi-protocol device that supports both devices, i.e., the TCP/IP device and shared-memory device, and this multi-protocol device is called *ch_p4*. One of the problems with the *p4_shmem* device is owing to the default soft processor affinity; since the OS scheduler attempt to maintain adequate load balancing, processes will move between processors of a computer. However, related data will not move to corresponding private memory due to the hard memory affinity. Therefore, all the processes may almost access the memory on certain processors and this will be a performance limiter.

Another problem of the *p4_shmem* device, a message could end up using a packet cached in the global queue of available packets while being prepared. While a packet allocated previously is unavailable, the process will allocate a new memory area for the new packet from SM. Owing to the page-alignment of the allocated memory area, if the new memory area is part of a memory page that has already been partially used for other packets, and this new memory area will belong to other process that allocates the memory page originally. This makes the performance of any given message transaction hard or impossible to predict.

The EMPIMPD is proposed solution these problem. First, in order to make sure that processes will not migrate away from the related data allocated in the private memory, processes should be locked in corresponding processors. Therefore, processes will be given the same hard affinity as memory and this can be accomplished with the "*sched_setaffinity( )*" system call. Before any private memory has been allocated, the affinity must be set as early as possible in the startup process. In order to make sure that the processes are evenly distributed among the available processors, the affinity cannot be set until the process has found its own rank. After all the processes have gotten their own ranks, they will be scheduled and set affinities.

Second, the queue of available packets needs to be split up, and each process has its own queue. When a packet is received by a process, it will be moved into the queue of available packets that belongs to the sender.

Third, the packet/message allocation will happen in page-aligned memory chunks or the per-process page-aligned shared memory segment. The SM allocator will be modified for both types of allocations; it will allocate page-sized memory blocks that are page aligned, and will take process IDs as the name of each memory page so that these named pp. can be managed effectively.

Finally, the lists *"avail_buffs->*buff", implemented by "struct p4_msg"* of cached available message buffers *"p4_global->avail_buffs[]"* will also need to be divided into page-aligned per-process partitions. A buffer for the message is fetched from the cached available message buffers, or allocated from the SM segment if no free buffer of sufficient size can be found. Then the data will be move into the list of cached available message buffer, which belongs to the sender.

3.3 Code Restructuring and Analysis

3.3.1 Loop Detection

Before unrolling loops, it is needed to detect the loop statements from code body. The operations are as follows: First, we will get the procedure definition of the input source code, i.e., part (1), and this procedure definition can be regarded as a handler of the source code. Then we can obtain the code body by using the method "*get_body*" via the procedure definition, i.e., part (2). Referring to the definition of code body in the "*SUIF Infrastructure Guide*", the code body of a procedure is formed with a lot of "*Statement*". Therefore, it is needed to transform the code body into "*Statement Description Format*" in order to find out the loop statement, i.e., part (3).

3.3.2 Loop Unrolling

After finding out the loop statements, the loops will be unrolled. In loop unrolling process, it will generate the information of iteration space that can be used to analyze the data dependence. After transforming the "*ForStatament*" into "*CollectObjects*", then we can use template "*list<ForStatement *>*" to convert the "*CollectObjects*" into enumerable type to see part (2). After taking all the loops out by using part (3) to enumerate every "*ForStatament"* in the input source code, all the loops will be unrolled in next step.

For unrolling a loop, e.g. *ForStatement*, it requires to obtain the upper bound, lower bound, and step of the loop. The array index changed with the loop index in the loop body will be replaced by a constant. Then the label and index of this array will be written into a table called "*Iteration Space Table"* (IST). An "*Iteration Space Table List"* can be used to maintain every IST in a code body. In the IST for iteration there are two entries: *LHS List* and *RHS List*, to record the array variables on the left hand side and the right hand side of a statement respectively.

3.3.3 Data Dependence Analysis and Elimination

In the proposed system, an iteration of a loop is a basic partition of a *computation unit*, thus, it is only required to analyze the loop carried data dependence. Loop carried data dependence can be detected by IST.

There are four searching directions to find out the four types of data dependence, as shown in Figure 3 and Table 1.

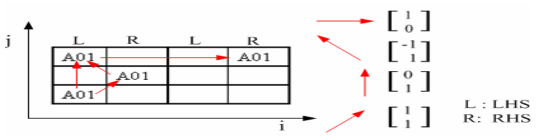
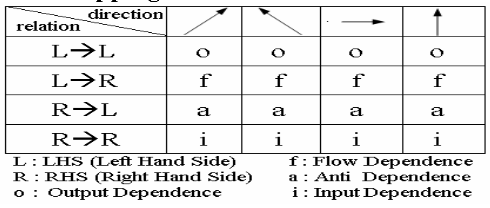


Figure 3: Four searching directions

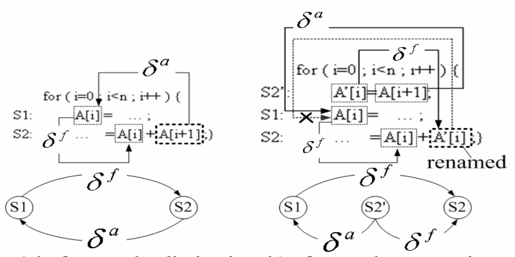
Table 1: Searching direction to data relation mapping table



relation \ direction	↗	↖	→	↑
L→L	o	o	o	o
L→R	f	f	f	f
R→L	a	a	a	a
R→R	i	i	i	i

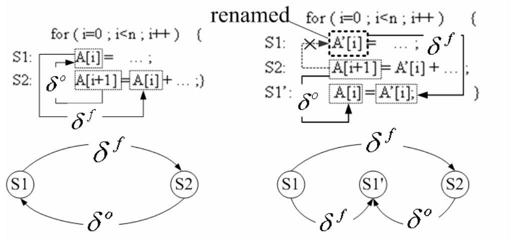
L : LHS (Left Hand Side) f : Flow Dependence
R : RHS (Right Hand Side) a : Anti Dependence
o : Output Dependence i : Input Dependence

The data dependence in a loop can be represented with a dependence graph. If there exists any cycle in the graph, i.e., *data dependence cycle* as illustrated in Figure 4(a) and 5(a), it will decrease the degree of parallelism in a loop. We can use the renaming methodology to eliminate the anti-dependence or output dependence to see in Figure 5(b) and 6(b) which may cause cycles. After removing the anti-dependence and output dependence, the SUIF intermediate format file will be sent to the "*loop unrolling*" step and "*data dependence analysis*" step again. Finally, the SUIF intermediate format file will be sent to the "*degree of parallelism determination*" step.



a) before cycle elimination b) after cycle elimination

Figure 4: Dependence graph with anti-dependence removing



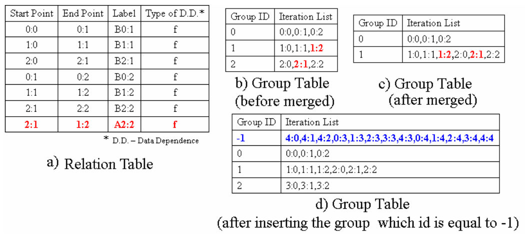
a) before cycle elimination b) after cycle elimination

Figure 5: Dependence graph with output dependence removing

3.4 Parallel Code Generation

3.4.1 Degree of Parallelism Determination

The degree of parallelism refers to the number of *computation units* a loop can be sliced up into. These *computation units* mapped into processors will be executed concurrently without any error caused from data dependence occurring. *Group Table* shown in Figure 6 is created to record a set of iterations which will be mapped into the same processor. There exist data dependence between iterations in one group, and these iterations must be mapped into the same processor, except the group which id is assigned to value *-1*. In this case, the maximum degree of parallelism is *16*.



Start Point	End Point	Label	Type of D.D.*
0:0	0:1	B0:1	f
1:0	1:1	B1:1	f
2:0	2:1	B2:1	f
0:1	0:2	B0:2	f
1:1	1:2	B1:2	f
2:1	2:2	B2:2	f
2:1	1:2	A2:2	f

* D.D. – Data Dependence

a) Relation Table

Group ID	Iteration List
0	0:0,0:1,0:2
1	1:0,1:1,1:2
2	2:0,2:1,2:2

b) Group Table (before merged)

Group ID	Iteration List
0	0:0,0:1,0:2
1	1:0,1:1,1:2,2:0,2:1,2:2

c) Group Table (after merged)

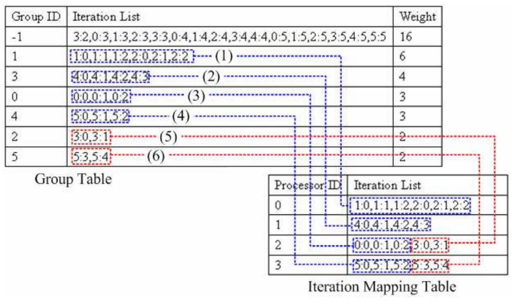
Group ID	Iteration List
-1	4:0,4:1,4:2,0:3,1:3,2:3,3:3,4:3,0:4,1:4,2:4,3:4,4:4
0	0:0,0:1,0:2
1	1:0,1:1,1:2,2:0,2:1,2:2
2	3:0,3:1,3:2

d) Group Table (after inserting the group which id is equal to -1)

Figure 6: Group Table

3.4.2 Loop Decomposition and Allocation

The maximum degree of parallelism can be found from the group table. Iterations which have the data dependence, group id > *-1*, will be allocated first. Next, these groups, with id > *-1,* will be allocated to processors in sequence according to the *Group Table*. For example, if there are four processors then allocation steps will be steps *(1)*, *(2)*, *(3)*, and *(4)* as shown in Figure 7. If each processor has been assigned a mapping group, the remaining groups, except the group with id=*-1*, will be allocated in sequence to processors which have the smallest number of mapping iterations, i.e., allocating step *(5)* and *(6)*. Finally the group with id=*-1* will be allocated to the processor with the similar process. At this stage, an *Iteration Mapping Table* is generated to record the mapping between iterations and processors for indicating which processor will perform which iterations.



Group ID	Iteration List	Weight
-1	3:2,0:3,1:3,2:3,3:3,0:4,1:4,2:4,3:4,4:4,0:5,1:5,2:5,3:5,4:5,5:5	16
1	1:0,1:1,1:2,2:0,2:1,2:2 (1)	6
3	4:0,4:1,4:2,4:3 (2)	4
0	0:0,0:1,0:2 (3)	3
4	5:0,5:1,5:2 (4)	3
2	3:0,3:1 (5)	2
5	5:3,5:4 (6)	2

Group Table

Processor ID	Iteration List
0	1:0,1:1,1:2,2:0,2:1,2:2
1	4:0,4:1,4:2,4:3
2	0:0,0:1,0:2,3:0,3:1
3	5:0,5:1,5:2,5:3,5:4

Iteration Mapping Table

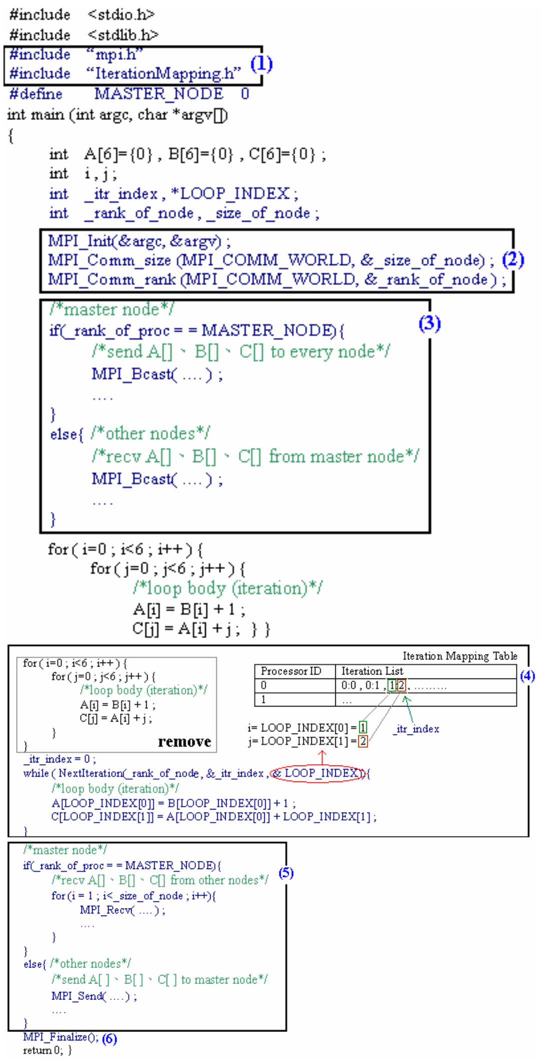
Figure 7: Allocating iterations to processors

3.4.3 Explicit Function Call Insertion

The action of inserting enhanced MPI function calls into the source code can be divided into six parts in Figure 8.

(1) It needs to declare the MPICH library and the proposed IterationMapping library in the source

code.
(2) The proposed system will insert the statements of initial settings for MPI environment into the source code.
(3) The array variables will be sent to other processing nodes, slave nodes, from the master node by using the *MPI_Bcast( )* function.
(4) Now the original nested loop structure will be removed and is replaced with the function "*NextIteration()*" which can return the index value of the array variable in the loop body.
(5) The slave nodes will send array variables that have accomplished to master node by using *MPI_Send()* and *MPI_Recv( )*.
(6) Finally, *MPI_Finalize()* is inserted.



```
#include   <stdio.h>
#include   <stdlib.h>
#include   "mpi.h"
#include   "IterationMapping.h"
#define      MASTER_NODE    0
int main (int argc, char *argv[])
{
      int   A[6]={0} , B[6]={0} , C[6]={0} ;
      int   i , j ;
      int   _itr_index , *LOOP_INDEX ;
      int   _rank_of_node , _size_of_node ;
      MPI_Init(&argc, &argv) ;
      MPI_Comm_size (MPI_COMM_WORLD, &_size_of_node) ;
      MPI_Comm_rank (MPI_COMM_WORLD, &_rank_of_node) ;
      /*master node*/
      if(_rank_of_proc = = MASTER_NODE){
            /*send A[] 、B[] 、C[] to every node*/
            MPI_Bcast( ....) ;
            ....
      }
      else{ /*other nodes*/
            /*recv A[] 、B[] 、C[] from master node*/
            MPI_Bcast( ....) ;
            ....
      }
      for ( i=0 ; i<6 ; i++ ) {
            for ( j=0 ; j<6 ; j++ ) {
                  /*loop body (iteration)*/
                  A[i] = B[i] + 1 ;
                  C[j] = A[i] + j ;  } }
for ( i=0 ; i<6 ; i++ ) {
      for ( j=0 ; j<6 ; j++ ) {
            /*loop body (iteration)*/
            A[i] = B[i] + 1 ;
            C[j] = A[i] + j ;
      }
}
_itr_index = 0 ;
while ( NextIteration(_rank_of_node , &_itr_index , & LOOP_INDEX)){
      /*loop body (iteration)*/
      A[LOOP_INDEX[0]] = B[LOOP_INDEX[0]] + 1 ;
      C[LOOP_INDEX[1]] = A[LOOP_INDEX[0]] + LOOP_INDEX[1] ;
}
/*master node*/
if(_rank_of_proc = = MASTER_NODE){
      /*recv A[] 、B[] 、C[] from other nodes*/
      for (i = 1 ; i<_size_of_node ; i++){
            MPI_Recv( ....) ;
            ....
      }
}
else{ /*other nodes*/
      /*send A[ ] 、B[ ] 、C[ ] to master node*/
      MPI_Send( ....) ;
      ....
}
MPI_Finalize();
return 0;  }
```

Iteration Mapping Table

Processor ID	Iteration List
0	0:0 , 0:1 , 1:2 , ...
1	...

```
i= LOOP_INDEX[0] = 1
j= LOOP_INDEX[1] = 2
```

Figure 8: Six parts of Function calls insertion

4. Experimental Results

Two case studies are selected in this section. The first case uses a MPI benchmark called *MPBench* to evaluate the performance of MPICH with proposed enhanced multi-protocol device. The MPI benchmark compiled with GCC compiler will run on a dual-processor computer. Six bandwidth benchmark item of the MPI benchmark is selected to execute such as *bandwidth*, *bidirectional bandwidth*, *all-to-all*, *broadcast*, *reduce* and *all reduce*. More detailed information about the test platform can be found in Table 2.

Table 2: Test Platform for Case I

Processor	Intel Xeon (Prestonia) LV @ 2.4 GHhz × 2	
Memory	1 Gigabytes × 1 (DDR 266 MHz with ECC registered)	
OS	Red Hat release 9 (Red Hat Linux 3.2.2-5)	
	Linux version	2.4.20-8smp
	compiler	gcc 3.2.2
MPI	MPICH version 1.2.7p1 (with *ch_p4* or enhanced *ch_p4* device)	
Benchmark	LLCbench–MPBench(six bandwidth benchmark items are selected)	

The second case to list in Table 3 uses the *Livermore Loops* program compiled with proposed automatic parallelizing compiler system to evaluate the feasibility and performance. The transformed *Livermore Loops* program will be executed on a PC cluster that consists of *8* dual-processor nodes in parallel as shown in Table 4.
After all bandwidth benchmark items are accomplished, the average bandwidth with different message size, from 512 bytes to 1024 Kbytes, are illustrated in Figure 9 and it can be found that the MPICH with proposed enhanced *ch_p4* device gives better performance than with original *ch_p4* device. Table 5 shows the speedups when running the different bandwidth benchmark items with proposed enhanced ch_p4 device instead of running with the original ch_p4 device. The maximum speedup 13.7% is given when executing the *bidirectional bandwidth benchmark*. While running the *broadcast benchmark* or the *all reduce benchmark* system gains the minimal speedup about 0.5%. Furthermore, the average speedup about 3.7% is gained. Therefore, the average speedup may be grater than 3.7% while most of the MPI operations used in a MPI program are bidirectional send/recv operations.
Table 6 shows the execution time of the parallelized Livermore loops program executed on the PC cluster with 8 dual-processor nodes. In this table, the problem size means the number of matrix operations, or the matrix size. Each loop kernel is executed twice with different problem sizes. The number of outmost loop iterations is fixed to 10000, besides loop kernel 5 and kernel 6.

Table 3: Test Platform for Case II

Processor	Intel Xeon (Prestonia) LV @ 2.4 GHhz × 2		× 8
Memory	1 Gigabytes × 1 (DDR 266 MHz with ECC registered)		
Network	1 Gigabit Ethernet		
OS	Red Hat release 9 (Red Hat Linux 3.2.2-5)		
	Linux version	2.4.20-8smp	
	compiler	proposed compiler system	
MPI	MPICH version 1.2.7p1		
Benchmark	Livermore Loops (8 loop kernels are selected)		

Table 4: Kernels of Livermore Loops

# of loop kernel	kernel name
1	hydro fragment
2	ICCG excerpt (Incomplete Cholesky Conjugate Gradient)
3	inner product
4	banded linear equations
5	tri-diagonal elimination, below diagonal
6	general linear recurrence equations
7	equation of state fragment
8	ADI integration

Table 5: Speedups of MPICH/enhanced ch_p4 device

Benchmark Items	**Mean Bandwidth (KB/s) (original *ch_p4* used)**	**Mean Bandwidth (KB/S) (enhanced *ch_p4* used)**	**Speedups**
bandwidth	56259.089929	56543.403745	0.51%
*Bibw**	116264.403108	117894.788383	1.40%
all-to-all	159057.839674	164780.347571	3.60%
broadcast	137161.926630	155961.733866	13.71%
reduce	174961.047554	175900.320313	0.54%
all-reduce	58006.535878	59516.728133	2.60%
		Average	3.73%

**bibw-bidirectional bandwidth*

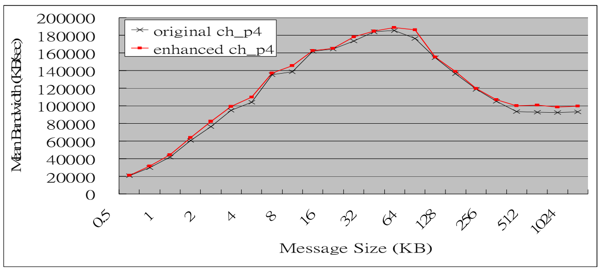


Figure 9: Mean bandwidth of MPICH/enhanced ch_p4 device

Table 6: Execution time of Livermore Loops

Kernel	Num.of procs. / Problem size	1	2	4	8	16
1	10000	0.696309	0.395374	0.198575	0.099585	0.065298
	50000	11.21486	5.929563	3.015653	1.512052	0.892601
2	10000	0.114364	0.060198	0.030958	0.015316	0.019145
	50000	0.571860	0.299505	0.149950	0.075417	0.058565
3	10000	1.115342	0.560758	0.284350	0.143190	0.114957
	50000	6.129068	2.881989	1.405347	0.701462	0.447979
4	10000	0.359792	0.203924	0.099732	0.063580	0.034578
	50000	1.802904	0.978933	0.496880	0.262015	0.153400
5	10000	7.478875	4.338958	1.975020	1.137458	0.582670
	50000	38.028135	20.998818	9.905799	5.056606	2.614763
6	1000	11.660749	6.232663	3.132081	1.563517	0.827019
	5000	58.231953	29.567091	15.134578	7.381455	3.778215
7	10000	2.062370	1.318247	0.663898	0.352114	0.229885
	50000	14.499141	7.406826	3.916108	2.164031	1.256460
8	1000	5.179509	2.873689	1.437425	0.817788	0.514371
	5000	53.762438	27.515169	14.541558	7.427022	4.574638

(sec.)

5. Conclusions

An automatic parallelizing compiler system has been proposed for improving the programmability of HMA. The compiler system performs the loop transformation for loop structures in a sequential program for parallel execution. The experimental results show that the compiler system is workable and a better system performance can be achieved. An EMPICHMPD has also been proposed and can improve the bandwidth when using the MPICH *ch_p4* device with SM communication mode.

6. Reference

[1] T. L. Sterling, J. Salmon, D. J. Backer, and D. F. Savarese, “How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters”, 2nd Printing, MIT Press, Cambridge, Massachusetts, USA, 1999.

[2] R. Buyya, “High Performance Cluster Computing: System and Architectures”, Vol.1, Prentice Hall PTR, NJ, 1999.

[3] B. Wilkinson and M. Allen,”Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”, Prentice Hall PTR, NJ, 1999.

[4] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng, “Compiling Fortran D for MIMD distributed-memory machines,” *Communications of the ACM*, Vol. 35, No. 8 (Aug. 1992), pp. 66-80.

[5] High Performance Fortran Language Specification Version 2.0, Houston Texas: Rice University, 1997.

[6] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skellum, “A high-performance, portable implementation of the MPI message-passing interface standard,” *Parallel Computing*, 1996, pp. 789-828.

[7] D. A. Padua et al., “Polaris: A new-generation parallelizing compiler for MPPs,” *Technical Report CSRD-1306*, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, June 1993.

[8] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel, “Evaluating the performance of

software distributed shared memory as a target for parallelizing compilers," *In the Proc. of the 11th International Parallel Processing Symposium*, Geneva. Switzerland, Apr. 1997, pp. 475-482.

[9] P. J. Keleher, "Update Protocols and cluster-based shared memory," *Computer Communications*," Vol. 22, No.11, July 1999, pp. 1045-1055.

[10] Yijun Yu, E.H. D'Hollander, "Partitioning loops with variable dependence distances," 2000 International Conference on Parallel Processing, 2000. Proceedings, Aug. 2000, pp. 209-218.

[11] M. E. Wolf, D. E. Maydan, Ding-Kai Chen, "Combining loop transformations considering caches and scheduling," Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, Dec. 1996, pp. 274-286.

[12] M. Kandemir, J. Ramanujam, and A. Choudary, "Compiler Algorithms for Optimizing Locality and Parallelism on Shared and Distributed Memory Machines," *Journal of Parallel and Distributed Computing*, 2000, pp. 924-965.

[13] G. Goumas, M. Athanasaki, and N. Koziris, "An efficient code generation technique for tiled iteration spaces," *IEEE Transaction on Parallel and Distributed Systems*, Vol. 14, Oct. 2003, pp.1021–1034.