Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS

Woo-Chul Jeun

Electrical Engineering and Computer Science
Seoul National University
Seoul 151-742 Korea
Tel : +82-2-880-7292
Fax : +82-2-879-1532
e-mail : wcjeun@iris.snu.ac.kr

Soonhoi Ha

Electrical Engineering and Computer Science
Seoul National University
Seoul 151-742, Korea
Tel : +82-2-880-7292
Fax : +82-2-879-1532
e-mail : sha@iris.snu.ac.kr

Abstract - It is attractive to use the OpenMP as a parallel programming model on a Multiprocessor System-On-Chip (MPSoC) because it is easy to write a parallel program in the OpenMP and there is no standard method for parallel programming on an MPSoC. In this paper, we propose an effective OpenMP implementation and translation for major OpenMP directives on an MPSoC with physically shared memories, hardware semaphores, and no operating system.

I Introduction

Two major models for parallel programming are the message-passing model and the shared address space model. In the message-passing model, each processor has a private memory and communicates data to other processors by a message. Though programmers can get high performance with this model, it is difficult to write a program to optimize data distribution and data movement for the performance. The Message Passing Interface (MPI)[1] is a de facto standard interface of this model. It is mainly used for a cluster that has physically distributed memories. In the shared address space model, all processors share a memory and communicate data through the access to the shared memory. Though this model needs a memory consistency model to keep a memory consistency, it is easy to write a program and to port a serial program to a parallel program. The OpenMP [2] is a de facto standard interface of this model. It is mainly used for a shared memory multiprocessor (SMP) machine [3][4]. Because it is easy to write a parallel program, there are some attempts to use the OpenMP as a parallel programming model on other parallel-processing platforms such as System-On-Chips and clusters [5][6][7][8].

An OpenMP implementation and an OpenMP translator are necessary to use the OpenMP on a target platform. The OpenMP interface is a specification standard to represent a programmer's intension with compiler directives. The standard does not define how to make an OpenMP implementation on a parallel-processing platform. When we choose a target platform, available Application Programming Interface (API) on the platform is different from API on other platforms. So, an OpenMP implementation varies from platform to platform. There can be several different OpenMP implementations on the same target platform. Then, a customized OpenMP translator for a target platform is necessary to translate an OpenMP program into a parallel program with API of the target platform. There are many works to translate an OpenMP program into a parallel program on various platforms [3][4][5][6][7][8][9][10].

Though a Multiprocessor System-On-Chip (MPSoC) platform can vary with a target application program, previous works do not cover these various platforms. A platform can have a shared memory or a distributed memory and use an operating system or not. When we customize an OpenMP implementation and an OpenMP translation to a target platform, we can get high performance on the platform. There are some attempts to use the OpenMP on an MPSoC because there is no standard method for parallel programming on an MPSoC and it is easy to write a parallel program in the OpenMP [7][8][9]. When a platform uses an operating system (OS) such as an SMP Linux Kernel and a thread library such as the POSIX thread library on an MPSoC board, the OpenMP implementation and translation on the platform are similar to those on an ordinary SMP machine [8]. However, when a platform does not use an operating system, we need different approach. Though there are previous works without using an OS, they mainly focused on OpenMP directive extension and OpenMP implementations using special hardware to improve the performance on the board [7][9]. Moreover, their implementations of the synchronization directive have the possibility that the synchronization directives work wrong.

This paper proposes an effective OpenMP implementation and translation for major OpenMP directives on a target MPSoC with physically shared memories, hardware semaphores, and no operating system and analyzes the performance. We focus on effective translation of global '*shared*' variables and effective implementation of the 'reduction' clause to improve the performance without using special hardware and OpenMP directive extension. We improved the implementation of synchronization directives and removed the possibility that the synchronization directives work wrong. We made the proposed OpenMP implementation and OpenMP translator. Then, we translated and ran some OpenMP programs. Experiment results show that the proposed OpenMP implementation and translation improve the performance of OpenMP programs.

This paper is organized as follows. In Section 2, we briefly

1-4244-0630-7/07/$20.00 ©2007 IEEE.

introduce a target MPSoC board and explain the OpenMP overview. We propose an effective OpenMP implementation and OpenMP translation for major OpenMP directives on the target MPSoC board in Section 3. Then, we present the experimental results in Section 4. Finally, we draw a conclusion with some idea of future research direction.

II. Background

A. CT3400: Cradle MPSoC Board

The CT3400 [11] is our target MPSoC board made by Cradle Technologies, Inc. Fig.1 shows the simplified block diagram of the board. In the figure, we only present some components that our OpenMP implementation uses. The architecture of the board mainly consists of an MPSoC chip and a 256MB global data memory. The chip has 64 global hardware semaphores, four 230MHz RISC-like processors, 32 local hardware semaphores, 32KB instruction cache, and 64KB local data memory. Processors can share variables in the local data memory and the global data memory. Each processor can access the memory exclusively.

The C language is a programming language for the processors and the '*cragcc*' [12] is GNU-based C-compiler. Each processor communicates data to other processors with the shared memory.

B. OpenMP Overview

The OpenMP defines a specification for the C/C++ language and the FORTRAN language and uses compiler directives. It is easy to write a parallel program and to port a serial program to a parallel program with inserting OpenMP directives into code segments that we wish to run in parallel. Fig.2 shows an example OpenMP code to sum from 0 to 9999. The runtime execution model of an OpenMP program is the fork/join model. The program executes serially with a single thread, referred to as the '*master thread*'. When the master thread encounters the '*parallel*' directive, the '*master thread*' creates some additional child threads and each child thread divides and executes the computation workload. After all child threads complete their own workload and the synchronization procedure, all child threads join the '*master thread*'. The 'master thread' resumes the execution again.

III. OpenMP Implementation and Translation for MPSoC

Because an OpenMP implementation and translation are dependent on a target platform, we present an OpenMP implementation and translation for our target platform, the CT3400.

A. The Clause determining attributes of variables

In the OpenMP, the variables in a parallel region have attributes such as '*shared*' and '*private*'. We can determine the attributes with the 'clause' of the OpenMP such as '*private (var)*', '*shared (var)*', and '*default (shared)*'. The '*shared*' variable means that all threads can directly access the variable. The '*private*' variable means that each thread has its own copy of the variable. The '*default*' clause sets the default attributes

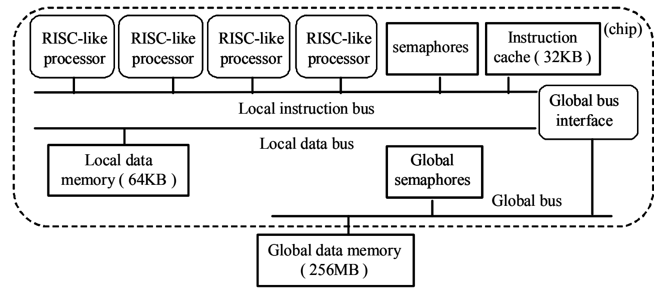

Fig. 1. Simplified block diagram of CT3400 board.

```
#pragma omp parallel default(shared) private(i)
{
      #pragma omp for reduction(+:sum)
       for(i=0;i<10000;i++)
      {
            sum += i;
       }
}
```

Fig. 2. Example OpenMP code to sum from 0 to 9999.

of variables. Because these attributes are only specification, we have to implement these attributes. For the '*private*' clause, we declare a variable that has the same name of the variable in the parallel region, again. Newly declared variable hides the original variable.

B. Global 'Shared' Variables

When we declare a '*shared*' variable as a global variable, all processors can access the variable. There are two approaches to allocate variables: static memory allocation and dynamic memory allocation. For the static memory allocation, we declare global variables like '*int data[10000]*' and the variables reside in the global data area. For the dynamic memory allocation, we declare a global pointer variable like '*int *data*' and allocate memory to the pointer with a memory allocation function such as '*malloc()*'. Though the pointer, '*data*', resides in global data area, real array resides in the heap area.

Though the memory allocation approach on PC does not cause large difference in the performance, the static memory allocation on the CT3400 can improve the performance. The '*cragcc*' compiler can efficiently process global variables when the global variables reside in global data area on the CT3400 [12]. Because this performance improvement depends on the compiler, the static memory allocation for global '*shared*' variables cannot always improve the performance for all application programs. However, when we use the dynamic memory allocation, we cannot inform the compiler that the variable is the global variable. Then the compiler cannot even get the opportunity to improve the performance.

C. Non-global 'Shared' Variables

If we do not declare a '*shared*' variable as a global '*shared*' variable, processors cannot directly share the variable on the CT3400. Each processor has its own copy of the '*shared*' variable similar to the '*private*' variable. So, our OpenMP implementation uses the shared memory as a communication channel. When one processor writes a value to its own copy of

the '*shared*' variable, our OpenMP implementation copies the value to a structure-type variable of the C language in the shared memory. The member variables of the structure-type variable are '*shared*' variables used in the parallel region. After our OpenMP implementation makes other processors copy the value of the structure-type variable to their own copies of the '*shared*' variable, other processors can read the value of the '*shared*' variable from their own copy of the '*shared*' variable.

D. '*Parallel*' Directive

Programmers can denote a parallel region by inserting '*parallel*' directive into the code segment that we wish to run in parallel. If a platform supports a thread library, the OpenMP implementation of the '*parallel*' directive is to use the thread library. However, we cannot use the fork/join model for the '*parallel*' directive because the CT3400 has no OS and no thread library. The master processor has to load other processors with an executable image and to invoke them at the beginning of the program. The master processor has to complete this procedure before the main function of the application program starts. So, our OpenMP translator changes the name of original main function to '*app_main*' and inserts the initialization procedure before calling the original main function, '*app_main*'. Previous works chose this approach on other parallel-processing platforms [6][7][9][10]. The master processor executes the program and other processors wait until the master thread arrives at a parallel region. All processors divide and execute the computation workload in the parallel region. After a synchronization step at the end of the parallel region, the master processor resumes the execution and other processors wait for a parallel region again [3][6][10].

Our OpenMP translator moves a parallel region to another place and defines a function including the parallel region code. The translator inserts a code calling the function instead of the original parallel region. This translation method has an advantage to make an OpenMP translator with small effort by reusing prior OpenMP translators for a platform that support a thread library.

E. Synchronization Directive

The synchronization directives are '*critical*', '*atomic*', and '*barrier*'. The '*barrier*' directive is a representative directive of the synchronization directives. Programmers can use the directive like '*#pragma omp barrier*'. This directive means that all processors have to wait here until all processors arrive at this synchronization point. Other synchronization directives are based on the '*barrier*' directive. Then, we present our implementation of the '*barrier*' directive and compare our implementation and previous implementation.

Fig.3(a) is the previous implementation [7][9] of the '*barrier*' directive and Fig.3(b) is our proposed implementation of the '*barrier*' directive. Both use hardware semaphores on the CT3400 to implement the synchronization directives. The '*PES*' means the total number of processors. The '*my_peid*' means unique processor ID from 0 to *n*-1 when there are *n* processors. The '*done_pe*' is a counter of processors that arrive at the synchronization point. The basic algorithm of the '*barrier*' implementation is that each processor gets the lock and increases the counter by one and releases the lock. When the counter equals the total number of processors, it means that all processors arrive at the synchronization point. Then, we have to initialize the counter to reuse it at next synchronization point. There is a problem related with this initialization step in the previous implementation. In Fig.3(a), the master processor initializes the counter to 0 at the end of the '*barrier*'. Assume that the master processor is not the last processor that arrives at the synchronization point. Until the last processor gets the lock and executes the statement of '*done_pe++*', other processors wait at the statement of '*while(done_pe < PES)*'. When the last processor completes the statement of '*done_pe++*', the master processor can escape the statement of '*while(done_pe < PES)*' and execute the statement of '*done_pe = 0*' before the last processor releases the lock. Then, the counter is initialized to 0 before the last processor evaluates the counter at the statement of '*while(done_pe < PES)*'. Then, the last processor cannot escape the statement of '*while(done_pe < PES)'*. Though we cannot always encounter this wrong execution result, there is a possibility for the program to work wrong because the execution result of multiprocessors is non-deterministic.

We propose our new implementation in Fig.3(b) to solve the problem. We introduce a '*phase*' variable and toggle the value of the variable at each '*barrier*' directive. With the '*phase*' variable, we can initialize the counter for the next '*barrier*' and keep the value of the counter for the present '*barrier*'. We use two counter variables, '*done_pe[2]*'. One is for the present '*barrier'* and another is for the next '*barrier*'. The last processor initializes the counter for the next '*barrier*' before the last processor increases the counter for the present '*barrier*'. Then, other processors cannot escape the statement of '*while(done_pe[phase]<(PES))*' until the last processor increases the counter for the present '*barrier*'. Because the '*phase*' of the present '*barrier*' is different from the '*phase*' of the next '*barrier*', the last processor can escape the statement of '*while(done_pe < PES)'* even when other processors arrive at the next '*barrier*'.

```
semaphore_lock(Sem.p);
done_pe++;
semaphoer_unlock(Sem.p);
while(done_pe<(PES))
      _pe_delay(1);
if(my_peid == 0)
      done_pe = 0;
```

(a) Previous '*barrier*' implementation

```
semaphore_lock(Sem.p);
phase = (phase + 1) % 2;
if(done_pe[phase]+1 == PES)
      done_pe[(phase+1)%2] = 0;
done_pe[phase]++;
semaphoer_unlock(Sem.p);
while(done_pe[phase]<(PES))
      _pe_delay(1);
```

(b) Proposed '*barrier*' implmentation

Fig. 3. OpenMP implementations of the '*barrier*' directive.

F. Work-sharing Directive

The '*for*' directive is a work sharing directive. Programmers can use this directive with a target for-loop as presented in Fig.2. This directive distributes the iterations of the target for-loop to processors. The '*reduction*' clause can be used with the 'for' directive. If we set a variable and a reduction operation with the 'reduction' clause, each processor computes a partial result for the reduction operation on the variable and the OpenMP implementation integrates all partial results into a total result. Fig.4 shows a translation example of

the '*for*' directive and the '*reduction*' clause presented in Fig.2. Because processors divide the iterations of the target for-loop, each processor has a different loop condition from loop conditions of other processors. Then, each processor declares some variables for own loop condition and own loop index and executes some procedure to determine the loop condition for the processor. After then, each processor runs the original for-loop region and gets a partial sum. For example, if there are four processors, the first processor gets a partial sum from 0 to 2499 and the second processor gets a partial sum from 2500 to 4999. Each processor saves the partial sum to the reduction structure-type variable, '*_t_red*' as presented in Fig.4. The '*reduce()*' function gathers all partial results and integrates them into a total result. In this example, the total result is the total sum of partial sums.

We can use two methods to implement the '*reduce()*' function. One method is to use a temporary variable in the shared memory. It is similar to the previous reduction implementation [7][9]. Fig.5 shows the code of '*reduce()*' function implementation using the method. Each processor gets a lock and directly accesses the temporary variable, '*reduction_buffer*' in the figure. The 'reduce' function gets the information of the partial sum, '*_t_red*' presented in Fig.4, as function parameters, '*buf*' and '*size*'. If a processor is the first processor that executes the '*reduce()*' function, the processor saves its partial sum to the temporary variable. After then, other processors update the temporary variable with the reduction function, '*function*' as presented in the figure. After all processors execute the '*reduce()*' function, the temporary variable, '*reduction_buffer*', is the final result. The disadvantage of this method is that while a processor gets a lock and executes a reduction operation, other processors wait to get the lock.

Another method is to use a temporary buffer array as a message queue. Fig.6 shows the code of '*reduce()*' function implementation using the method. We declare a temporary buffer array as '*char reduction_buffer[PES][4096]*' in the shared memory. Each processor writes its own partial sum to the fixed position for the processor, '*reduction_buffer [my_peid]*', in the buffer array without a lock. The '*barrier*' part in this implementation makes sure that all processors write their partial sums to the buffer array. After then, each processor can add all partial sums from the fixed positions in the buffer array and get the final result without a lock. Because we just use a lock for the synchronization, we can run the reduction operation in parallel and reduce the waiting time for a lock.

```
int _t_i, _t_n, i;    // loop initial condition, loop termination condition, loop index
int *_rdl_sum = &(*(_t_shared->_sh_sum)); // variable for final reduction result
int _rdf_sum = (*(_t_shared->_sh_sum)); // initial value of the reduction variable
int sum = 0;
int _rd_sum = 0;
// some procedure to determine loop conditions for this processor
 for ( i = _t_i ; i < _t_n ; i++ ) {     //    original for-loop region
    sum += i;
 }
 _rd_sum = 0;                  // initializing temporary variable for reduction result
 _rd_sum = _rd_sum + sum;       // saving partial sum of this processor
 { reduction_0 _t_red;
    _t_red._rd_sum = _rd_sum; // moving partial sum to the reduction 'structure'
   reduce(&_t_red, sizeof(reduction_0), reduce_0); // reduction between processors
  *_rdl_sum = _rdf_sum + _t_red._rd_sum;      // getting the final reduction result
}
barrier();
```

Fig. 4. Translation for a '*for*' directive and a '*reduction*' clause.

```
void reduce(void *buf, int size, parade_reduce_function function) {
semaphore_lock(Sem.p);
phase = (phase + 1) % 2;
if( done_pe[phase] + 1 = = PES )
      done_pe[(phase + 1) % 2] = 0;
if( done_pe[phase] = = 0)    // First processor initializes the temporary var.
       memcpy(reduction_buffer, buf, size);
else function(reduction_buffer, buf); // other processors execute reduction.
done_pe[phase]++;
semaphore_unlock(Sem.p);
while(done_pe[phase] < PES)
       _pe_delay(1);
memcpy(buf, reduction_buffer, size); // copy final result to all processors
}
```

Fig. 5. '*reduce*' implementation using a temporary variable.

```
void reduce(void *buf, int size, parade_reduce_function function) {
int i;
memcpy(reduction_buffer[my_peid], buf, size); // copy partial sum to buffer
semaphore_lock(Sem.p);   // 'barrier'
phase = (phase + 1) % 2;
if( done_pe[phase] + 1 = = PES )
      done_pe[(phase + 1) % 2] = 0;
done_pe[phase]++;
semaphore_unlock(Sem.p);
while(done_pe[phase] < PES)
      _pe_delay(1);
for( i = 0; i < PES ; i++) // add all partial sums of other processors in buffer
      if( PES != i ) {
            function(buf, reduction_buffer[i]);
      }
}
```

Fig. 6. '*reduce*' implementation using a temporary buffer array.

IV. Experiments

We customized the Omni C compiler [5] to our OpenMP implementation and our OpenMP translation. Then, we translated and ran some OpenMP programs on the CT3400 board.

The 'Inspector' is a cycle accurate simulator that the Cradle Technologies, Inc. made for the CT3400 board. Like previous works [7][9], we present results on the simulator as it provides detailed profiling information.

We used a program to sum from 0 to 9999 shown in Fig.2 and an N*N matrix multiplication program that the CT3400 development kit provides. Though there are only a serial program and a hand-written parallel program of N*N matrix multiplication in the kit, we wrote an OpenMP program from the serial program. As there is no standard OpenMP benchmark for an MPSoC, previous works used the matrix multiplication program, too [7][9]. Because the matrix multiplication program has no '*reduction*' clause, we used the 'sum' program for the experiment of the '*reduction*' clause. We replaced the statement of '*sum += i*' with the statement of '*sum += data[i]*' to test the effect of memory allocation method for the global shared array, '*data[]*'. We allocate all variables on the global data memory in this paper. We used the

EPCC benchmark [13] to measure the overhead of OpenMP directives.

Table I shows a result of 24*24 matrix multiplication. As the number of processors increases, the execution time is reduced. The static memory allocation for global shared variables shows better performance than the dynamic memory allocation. Both the serial program and the hand-written program use the static memory allocation. Then, our OpenMP implementation shows similar performance to the hand-written code when we used the static memory allocation for the global shared variables. On the other hand, the serial program spends 5,207,562 cycles in another experiment when we modify the serial program to use the dynamic memory allocation. Table II and Table III show the execution time of the matrix multiplication with various size of matrix. In all cases except for 4*4, the static memory allocation is about 21%~31% better than the dynamic memory allocation. For 4*4, the computation workload is small enough to equal the overhead of OpenMP directives and the difference of the execution time between two allocation methods is much smaller than other cases.

Table IV shows the overhead of major OpenMP directives that we measured with the EPCC benchmark. The overhead of OpenMP directives in the matrix multiplication program is about 20,000 cycles. This overhead is constant, regardless of the size of matrix and the number of processors. Our '*barrier*' implementation needs additional 200~700 cycles compared to previous implementation. However, we removed the problem of previous implementation and the additional overhead is much smaller than the overhead of other directives. For the '*reduction*' clause, we implemented and compared our two methods. The method using a temporary buffer array as a message queue is about 10% better than the method using a temporary variable in the shared memory.

Table V shows the execution time of the 'sum' program. The program spends about 350,000 cycles in serial part, regardless of the number of processors. However, as the number of processors increases, the execution time for the parallel part is reduced. The '*variable*' and '*array*' mean two methods to implement the '*reduction*' clause. In this program, the static memory allocation is 2% worse than the dynamic memory allocation. Though the OpenMP translator gives the compiler the hint that the variable is a global variable, the compiler does not efficiently use the hint. The method using a temporary buffer array as a message queue reduced the execution time by about 1000~2700 cycles, compared to the method using a temporary variable in the shared memory. Because this program executes the '*reduction*' clause once, the benefit is small.

IV. Summary and Conclusions

In this paper, we proposed effective OpenMP implementation and translation for major OpenMP directives without special hardware or OpenMP directive extension on an MPSoC with physically shared memories, hardware semaphores, and no operating system. We made our proposed OpenMP implementation and OpenMP translator. OpenMP directives work well on the MPSoC board simulator.

When we translate the global variable declaration into static

TABLE I
Execution time of 24*24 matrix multiplication (cycles)

Processors	Serial	Parallel (hand-written)	OpenMP	
			Dynamic	Static
1	3,664,513	3,653,761	5,221,225	3,674,336
2	N/A	1,827,537	2,622,901	1,845,050
4	N/A	914,127	1,320,474	933,549

TABLE II
Execution time of the matrix multiplication program with dynamic memory allocation (cycles)

Processors	The number of processors		
	1	2	4
4*4	38,291	31,120	25,413
8*8	208,399	116,330	68,174
16*16	1,564,136	794,121	409,248
32*32	12,373,765	6,198,865	3,108,500
64*64	98,731,357	49,375,560	24,698,132

TABLE III
Execution time of the matrix multiplication program with static memory allocation (cycles)

Processors	The number of processors		
	1	2	4
4*4	41,314	32,756	29,233
8*8	164,922	92,587	56,983
16*16	1,118,376	566,935	294,333
32*32	8,650,047	4,332,926	2,177,484
64*64	68,346,647	34,181,382	17,101,868

TABLE IV
Overhead of major OpenMP directives (cycles)

Processors	1	2	4
Previous 'barrier' implementation	1,136	1,698	2,848
Our 'barrier' implementation	1,404	2,128	3,581
'parallel' directive	1,645	5,094	8,109
'for' directive	7,340	8,933	12,039
'reduction', temporary variable	1,713	8,790	14,028
'reduction', temporary buffer array	1,713	7,805	12,631

TABLE V
Execution time of the program to sum from 0 to 9999 (cycles)

Processors	Dynamic allocation		Static allocation	
	Variable	Array	Variable	Array
1	1,678,840	1,706,669	1,678,740	1,707,169
2	1,014,402	1,028,935	1,013,134	1,027,983
4	680,179	694,081	677,409	692,457

memory allocation instead of dynamic memory allocation, a compiler can know whether the shared variables are global variables. It is possible to make a compiler that can efficiently process the global variable region at a platform. Even though the compiler cannot always improve the performance of all kinds of applications with this information, this information gives the compiler the opportunity of performance improvement. When the computation workload is sufficiently larger than the overhead of OpenMP directives and a compiler efficiently process the global shared variables, the matrix multiplication results show that static memory allocation for global shared variables can reduce the execution time by 21%~31% compared to dynamic memory allocation. The 'sum' program result shows that the performance degradation is about 2% even when the compiler does not efficiently process the global shared variables.

For the reduction implementation, the EPCC benchmark and the 'sum' program results show that using a temporary buffer array as a message queue is 10% better than using a temporary variable in the shared memory.

We improved the implementation of synchronization directives and removed the possibility that the synchronization directives work wrong. The EPCC benchmark result shows that our implementation needs about additional 200~700 cycles compared to the implementation of the previous work. However, the overhead is smaller than the overhead of other directives.

We plan to make the OpenMP translator support for OpenMP 2.0[14] and to research about an OpenMP implementation and translation for an MPSoC with physically distributed memory.

Acknowledgements

This work was supported by National Research Laboratory Program (No.M1-104-00-0015), Brain Korea 21 Project, SystemIC 2010, IT-SoC, and IT leading R&D Project funded by Korean MIC. The ICT and ISRC at Seoul National University and IDEC provide research facilities for this study.

References

[1] Message Passing Interface Forum, "MPI: A message-passing interface standard", *International Journal of Supercomputer Applications and High Performance Computing*, Vol.8, No.3/4, pp159-416, 1994.

[2] OpenMP Architecture Review Board, "OpenMP C and C++ application program interface," *http://www.openmp.org*, Version 1.0, Oct. 1998.

[3] Christian Brunschen and Mats Brorsson, "OdinMP/CCp – a portable implementation of OpenMP for C," *EWOMP'1999*, Lund, Sweden, Sept. 1999, pp.21-26.

[4] Vassilios V. Dimakopoulos and Elias Leontiadis, "A portable C compiler for OpenMP V.2.0", *EWOMP'2003*, Aachen, Germany, Sept. 2003, pp.5—11.

[5] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka, "Design of OpenMP compiler for an SMP cluster," *EWOMP'99*, Lund, Sweden, Sept. 1999, pp.32-39.

[6] Yang-Suk Kee, Jin-Soo Kim, and Soonhoi Ha, "ParADE: An OpenMP programming environment for SMP cluster systems," *ACM/IEEE Supercomputing (SC'03)*, Nov 12-15, 2003.

[7] Feng Liu and Vipin Chaudhary, "A practical OpenMP compiler for system on chips," *WOMPAT 2003, LNCS 2716*, pp. 54-68, 2003.

[8] Yoshihiko Hotta, Mitsuhisa Sato, Yoshihiro Nakajima, Yoshinori Ojima, "OpenMP implementation and performance on embedded renesas M32R chip multiprocessor," *EWOMP 2004*, Stockholm, Sweden, Oct. 2004, pp. 37-42.

[9] Feng Liu and Vipin Chudhary, "Extending OpenMP for Heterogeneous Chip Multiprocessors," *ICPP'2003*, Kaohsiung, Taiwan, Oct. 2003, pp.161-.

[10] Woo-Chul Jeun, Yang-Suk Kee, and Soonhoi Ha, "Improving performance of OpenMP for SMP clusters through overlapped page migrations," *International Workshop on OpenMP (IWOMP'06)*, Reims, France, 2006, in press.

[11] Cradle Technologies, Inc., *CT3400 Multi-core DSP datasheet*, http://www.cradle.com

[12] Cradle Technologies,Inc., *CRAGCC Compiler Addendum*, http://www.cradle.com

[13] EPCC OpenMP Microbenchmarks 1.0, *http://www.epcc.ed.ac.uk/research/openmpbench*

[14] OpenMP Architecture Review Board, "OpenMP C and C++ application program interface," *http://www.openmp.org*, Version 2.0, Mar. 2002.