An Implementation of OpenMP Compiler for PC Clusters based on Array Section Descriptor

Naoki Yonezawa
Department of Information and Computer Science
Faculty of Science, Kanagawa University
Hiratsuka, Kanawaga, Japan
Email: yonezawa@info.kanagawa-u.ac.jp
Telephone: +81–463–59–4111
Fax: +81–463–58–9684

Koichi Wada
Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba, Ibaraki, Japan
Email: wada@is.tsukuba.ac.jp
Telephone: +81–29–853–5512
Fax: +81–29–853–5512

***Abstract*— In this paper, we propose an implementation of OpenMP compiler for distributed memory environment. While OpenMP provides a notion of shared address space, distributed memory environment does not have a physical shared memory. One of the approaches to implement OpenMP on distributed memory environment is communication code generation, in which a producer sends appropriate data to the consumer. Our compiler finds accesses to shared data and represents them by using quad, which is our proposed array section descriptor. To identify data to be sent, intersection operation is performed between quads representing written and read data. Since a quad can concisely represent stride accesses to an array section, our compiler can generate efficient code in the case which OpenMP directive divides a for-loop in block-cyclic manner. As a preliminary evaluation, we parallelized a matrix-multiply program by inserting an OpenMP directive and executed it on a PC cluster. In result, we achieved a speedup of 7.82 with 8 processors.**

I. Introduction

Several parallel programming languages, such as High Performance Fortran (HPF) [1] and Split-C [2], provide a notion of shared memory or global address space. Shared memory has been also used in thread libraries, such as POSIX thread [3], in which threads can communicate via global variables. Recently, OpenMP [4] has emerged as a standard for shared memory programming and mainly is used on multiprocessor system which has a physical shared memory, e.g., Symmetric Multi-Processor.

In this paper, we propose an implementation of OpenMP compiler for distributed memory environment, such as PC clusters. One of the approaches to implement OpenMP on distributed memory environment is *communication code generation*, in which a producer sends appropriate data to the consumer. To identify data to be sent, intersection operation is performed between the sets representing written and read data. Such sets can be represented by using an *array section descriptor*. Several array section descriptor has been proposed so far [5], [6], [7], [8], [9].

In order to balance load, OpenMP have facilities to divide and assign iterations of a loop to processors in a fashion of *cyclic* or *block-cyclic*. An OpenMP program using these facilities tends to access the array sections periodically with strides. The above conventional array section descriptors cannot represent such access patterns efficiently. In the former literature[10], we have proposed a new array section descriptor, called *quad*, which can concisely represent the access patterns typically observed in parallel programs.

We use quad as a component of our compiler proposed in this paper. In analysis phase, our compiler identifies the segments in the source code that access the shared data and recognizes the access type (i.e., read or write) as well as the accessed section of the shared array, which is represented by quads. Our compiler generates not only intersection operation between quads representing written and read array section, but also communication codes to send the array section represented by the resulting quads which will be yielded by the operation. In result, the data consumer can receive the necessary data for correct execution.

The rest of this paper is organized as follows. Firstly, we describe quad and operations among quads in Section II. Section III describes implementation of our OpenMP compiler using quad in detail. Section IV presents the results of performance evaluations. Finally, we conclude in Section V.

II. Array Section Descriptor for Parallel Computing — quad

An array section assigned to a processor in block-cyclic manner is mainly accessed by the processor on which the array section is placed. Those accessed sections consist of iteratively accessed subsections that include non-accessed subsections, as shown in Fig. 1(a). In the case that the processor accesses the slightly wider section than the assigned section, the shape will be similar to the above case (Fig. 1(b)). Quad can represent these access patterns concisely, which typically occur in parallel programs.

A. quad

Quad, as its name implies, is a quadruplet of integers. A quad (a, b, c, d) consists of the following parameters:

- a: first subsection's offset from the head of the array,
- b: the length of accessed subsection,
- c: the length of non-accessed subsection,

0-7803-7978-0/03/$17.00 ©2003 IEEE

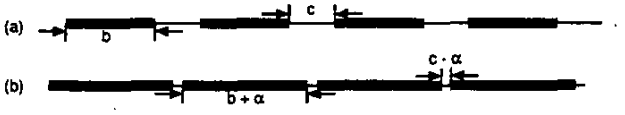

Fig. 1. Typical Access Patterns Observed in Parallel Programs

```
int a[n], c[n];
#pragma omp parallel for schedule(static, bs)
for (i = 0; i < bs * nproc * 5; i++)          /* (a) */
   a[i] = i * i;
#pragma omp parallel for schedule(static, bs)
for (i = 0; i < bs * nproc * 5; i++) {        /* (b) */
   if (i <= 0 || i >= n - 1) continue;
   c[i] = a[i-1] + a[i] + a[i+1];
}
```

Fig. 2. An Example of OpenMP Program

d: the number of repetition.

For example, in the program shown in Fig. 2(a) that shows a piece of an OpenMP program, each processor accesses section represented by the following quad:
$(pid * bs, bs, bs * (nproc - 1), 5)$
(where pid is processor ID, $nproc$ is the number of processors, and bs is block size).

Because quad always represents array section with four parameters, the range of represented array section does not affect the size of the descriptor and the cost of operations among descriptors.

B. Operations among quads

The operations we have defined are intersection operation and union operation.

In the code of Fig. 2(b), a processor reads slightly wider section than assigned to the processor. In this case, two boundary elements of the individual read subsection should be transferred to the processor because they are written by other processors. To obtain a quad that represents data to be transferred, intersection operation between quads can be used. For example, data to be transferred from Processor 2 to Processor 3 is $(3*bs-1, 1, bs*(nproc-1)+bs-1, 5)$, that is the result of $(2*bs, bs, bs*(nproc-1), 5) \cap (3*bs-1, bs+2, bs*(nproc-1)-2, 5)$.

Union operation can be used to merge several quads into fewer quads. In Fig. 3, first loop writes to $(pid*25, 25, 0, 1)$ and second loop writes to $(100+pid*25, 25, 0, 1)$. These two quads can be unified to $(pid*25, 25, 75, 2)$ by using union operation.

III. OpenMP Compiler Using quad

Our compiler consists of code translator and communication library, as shown in Fig. 4. The code translator analyzes an OpenMP program and locates the shared data accesses, which are represented by several quads. If the translator finds

```
omp_set_num_threads(4);
#pragma omp parallel for nowait
for (i = 0; i <100; i++)
   a[i] = i * i;
#pragma omp parallel for
for (i = 100; i <200; i++)
   a[i] = i * i + 1;
```

Fig. 3. Two Loops that Access Exclusive Parts of the Same Array

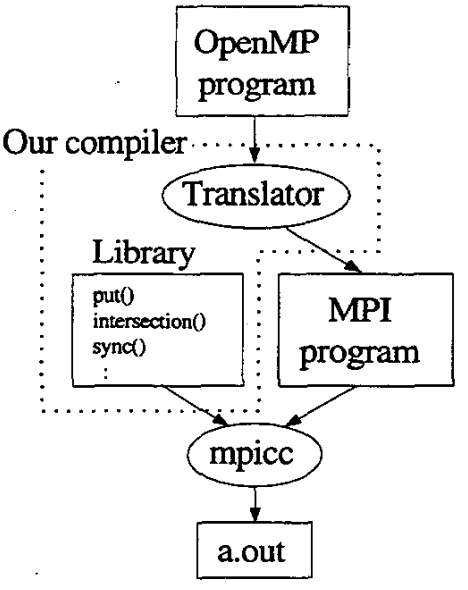

Fig. 4. A Compiling Flow

a synchronization, it generates communication code. In the communication code, data to be transferred will be identified by intersection operation among quads at runtime and will be sent by message passing call such as MPI_Send(). The final code generated by our translator is compiled with `mpicc` and can be executed on PC cluster.

A. Generating Communication Code

In analyzing an OpenMP program, the code translator locates the shared data accesses. If the access is write, it is represented by quads and a *quad register function* is generated in the location on which the write access occurs. This function registers the quads to a *quad table*. The quad table is indexed by access type (i.e., read or write) and processor ID. If the access is read, a quad register function is generated in the location on which the corresponding write access occurs, which produces data read by the read access.

Before a synchronization point, the code translator inserts communication code. To investigate data dependency, the communication code firstly extracts two quad sequences from the quad table. The first sequence represents the array section written by the processor that is to execute the communication

```
#pragma omp parallel for
for (i = 0; i <16; i++)
    a[i] = i * i;
b = a[2] + a[3] + a[4] + a[5];
```

Fig. 5. An OpenMP Program

```
register_quad(pid, write, a(pid * 16 / nproc, 16 / nproc, 0, 1));
register_quad(p0, read, a(2, 4, 0, 1));
for (i = pid * 16 / nproc; i < (pid + 1) * 16 / nproc; i++)
    a[i] = i * i;
foreach p.(all_other_processors)
    push(intersection(written_by_me, read_by_p));
sync();
if (pid == 0) b = a[2] + a[3] + a[4] + a[5];
```

Fig. 6. Inserting quads and Communication Code

code. The second one represents the array section read by one of other processors. Then, the communication code invokes intersection operation between these two quad sequences and consequently a new quad sequence representing data to be transferred are obtained. If the quad sequence is not empty, the data represented by the sequence are transferred by push() function.

B. Updating to Home Processor

Our compiler equally divides arrays into subarrays and assigns them to processors. A processor to which the subarray is assigned is called a *home processor* of the subset. If write accesses by non-home processor occurred, the written data should be transferred to the home processor by the next barrier. Our compiler maintains the quad table about home assignment as well as write accesses. The data to be transferred can be computed by intersection operations between these quad sequences.

C. An Example of Code Generation

Fig. 6 depicts an example of code segment which is generated as a result of translating the original OpenMP program shown in Fig. 5. In Fig. 6, inserted quad register functions and communication codes are emphasized in italics. Firstly, the code translator finds for-loop that is to be divided and assigned to all processors. An array a is written in this for-loop and quad register function is generated with respected to each processor.

Our translator generates an explicit barrier, that is included in our communication library, because OpenMP requires that all processors synchronize after parallel-for implicitly.

Several elements of array a are read by a single processor after barrier. Our translator generates if-statement so that the only processor 0 executes write access to variable b. Additionally, our translator inserts quad register function with respect to the processor 0. Note that this function should be called by all processors so that write processors can push data to the processor 0.

TABLE I
THE EXECUTION TIME OF MATRIX-MULTIPLY

no. of procs.	comp. (sec)	speedup	overall time (sec)	speedup
1	93.67	1.00	96.07	1.00
2	46.95	2.00	48.67	1.97
4	24.21	3.87	26.62	3.61
8	11.97	7.82	14.65	6.56

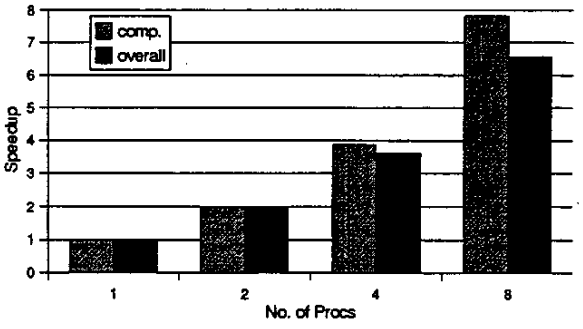

Fig. 7. Speedups of Matrix-Multiply

IV. EVALUATION

For evaluation of our compiler, we executed a matrix-multiply program on a PC cluster. Our PC cluster has 8-node of PCs that are connected by Myrinet whose throughput is 2Gbps. Each PC has PentiumIII 800MHz, 512MB memory, and Linux 2.4.10 running.

We also have evaluated the performance of operations among quads. However, we do not describe it here because it is described in detail in the former literature[10].

A. Matrix-Multiply

Parallel matrix-multiply program is obtained by adding directives of OpenMP to the sequential program. In this evaluation, a single line of the directive have been inserted at the for-loop that executes multiplication.

According to the inserted directive, the compiler divides the matrix in block-row partitioning scheme, and assigns the consecutive rows to each home processor.

In this evaluation, the size of matrix is 1024×1024.

B. Results

The matrix-multiply program firstly initializes the matrices by using a single processor. After initialization, these data are transferred to home processors and these submatrices are multiplied by home processors. We measured the execution time including and excluding time for initialization.

Table I shows the execution time and the speedup of the matrix-multiply program. Fig. 7 also depicts the speedup. We achieved a speedup of 7.82 with 8 nodes for the part of computation. When the time for the initialization is included, 6.56-fold speedup have been obtained. The reason for this slight degradation of speedup when the time for initialization

is included, is as follows. When the number of processors increases, the size of the partitioned work that is assigned to each processor decreases. Therefore, the initializing processor keeps the small part of the matrix and pushes the other parts to each home processor. In other words, the more the number of processors, the more amount of data transferred. Additionally, compared with the time for computation, the time for data transfer is relatively longer for more processors.

V. Conclusions and Future Works

In this paper, we described the implementation of OpenMP compiler for PC clusters. Our compiler uses an array section descriptor, called quad, to identify data which should be transferred to the consumers. Quad can represent stride accesses to an array section so that our compiler can generate efficient code in the case which a for-loop is directed to divide in block-cyclic manner.

As a preliminary evaluation, we compiled a matrix-multiply program which is parallelized by an OpenMP directive and executed it on a PC cluster. As a result, we achieved 7.82-fold speedup with 8 processors.

Our future works are as follows:

- evaluating our compiler with a various kind of application programs.
- cooperating with a system which fetches data on demand.
- supporting interprocedural analysis.

At present, our compiler assumes that all of the bindings between data produced and data consumed can be known at compile time and cannot compile the programs which intend to balance load dynamically by using task queue and whose behavior depends on inputs. To cope with these programs, we are developing a system which fetches the fresh data on demand. This dynamic coherency management feature enables clusters to execute any shared memory programs. However, we believe that if this feature cooperates with our compiler, a large number of data can be pushed to consumer in advance rather than fetched by consumer on demand and thus the program can be executed more efficiently.

Additionally, our compiler should support interprocedural analysis. In the interprocedural analysis, if a procedure is called in a loop body, the compiler tries to summarize the information on accesses in the procedure using quads. The quads will be used to yield new quads that represent the access summary about the loop.

Acknowledgment

This research was supported by a Grant-in-Aid for Young Scientists (B-14780184) from the Japan Society for the Promotion of Science.

References

[1] H. P. F. Forum, "High performance fortran language specification version 2.0," 1997.

[2] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick, "Parallel programming in Split-C," in *Supercomputing*, 1993, pp. 262–273.

[3] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*, ser. Nutshell handbook. 981 Chestnut Street, Newton, MA 02164, USA: O'Reilly & Associates, Inc., 1998.

[4] O. A. R. Board, *OpenMP C and C++ Application Program Interface*, 1998.

[5] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," *SIGPLAN Notices*, vol. 21, no. 7, pp. 162–175, July 1986, *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*.

[6] Z. Li and P.-C. Yew, "Efficient interprocedural analysis for program parallelization and restructuring," in *Proceedings ACM/SIGPLAN PPEALS 1988 Parallel Programming: Experience with Applications, Languages and Systems, July 1988, SIGPLAN Notices*, Sept. 1988, pp. 85–99.

[7] V. Balasundaram, "A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 154–170, June 1990.

[8] R. Triolet, F. Irigoin, and P. Feautrier, "Direct parallelization of call statements," in *ACM SIGPLAN 1986 Symposium on Compiler Construction*, June 1986, pp. 176–185.

[9] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, July 1991.

[10] N. Yonezawa and K. Wada, "*quad*: an array section descriptor for parallel computing," in *IASTED International Conference on Networks, Parallel and Distributed Processing, and Applications*, October 2002, pp. 46–52.