Automatic Parallelization and Optimization for Irregular Scientific Applications *

Minyi Guo
Dept. of Computer Software, The University of Aizu
Aizu-Wakamatsu City, Fukushima, 965-8580, Japan

Abstract

In this paper, some automatic parallelization and optimization techniques for irregular scientific computing are proposed. These techniques include communication cost reduction for irregular loop partitioning, interprocedural optimization techniques for communication preprocessing when the irregular code has the procedure call, global vs. local indirection arrays remapping methods, and OpenMP directive extension for irregular computing.

Keywords *Parallelizing compilers, Irregular scientific application, Communication optimization, Loop transformation, Loop partitioning, Interprocedural optimization, OpenMP.*

1 Introduction

Many codes in scientific and engineering computing involve sparse and unstructured problems in which array accesses are made through a level of indirection or nonlinear array subscript expressions. This means that the data arrays are indexed either through the values in other arrays, which are called *indirection arrays/index arrays*, or through non-affine subscripts. The use of indirect/nonlinear indexing causes the data access patterns, i.e. the indices of the data arrays being accessed, to be highly irregular. Such a problem is called *irregular problem*, in which the dependency structure is determined by variable causes known only at runtime. Irregular applications are found in unstructured computational fluid dynamic (CFD) solvers, molecular dynamics codes, diagonal or polynomial preconditioned iterative linear solvers, n-body solvers, and so forth.

Exploiting parallelism for irregular problems becomes very difficult due to their irregular data access pattern. A typical example is shown below. Here, elements are moved across the columns of a 2D array based on the information provided in the indirection arrays `prev_elem` and `next_elem`. The elements of array `cell` are shuffled and stored in array `new_cell`.

```
    DO 100 i = 1, rows
C   size(i) is the number of ele-
ments in the i-th row
    DO 200 j = 1, size(i)
      prev_elem(i,j) = new_elem(0,i,j)
      next_elem(i,j) = new_elem(1,i,j)
      new_cell(i, prev_elem(i,j)) =   &
                          f(cell(i,j))
      new_cell(i, next_elem(i,j)) =   &
                          g(cell(i,j))
200 CONTINUE
100 CONTINUE
```

Figure 1. A typical irregular loop

In this paper, we propose automatic parallelization and optimization techniques for irregular parallel code. These techniques include reducing communication cost for indirection array loop partitioning, global-local array transformation and index array remapping, inter-procedural communication optimization for irregular loops, and OpenMP directive extension if an irregular code uses OpenMP for its parallelization. These methods can experimentally achieve better performance.

2 Reducing Communication cost for Indirection Array Loop Partitioning

In this section, we propose a communication cost reduction technique for indirection array loop partitioning. In the following discussion, we assume that the indirection array loop body has only loop-independent dependence, but no loop-carried dependence (it is very difficult to test irregular loop-carried dependence since dependence testing methods

*This research was supported in part by the Grant-in-Aid for Scientific Research (C)(2) 14580386.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

for linear subscripts are completely disabled), because most of practical irregular scientific applications have this kind of loops.

Generally, in distributed memory compilation, loop iterations are partitioned to processors according to the owner computes rule [1]. This rule specifies that, on a single-statement loop, each iteration will be executed by the processor which owns the left hand side array reference of the assignment for that iteration.

However, owner computes rule is often not best suited for irregular codes. This is because use of indirection in accessing left hand side array makes it difficult to partition the loop iterations according to the owner computers rule. Therefore, in CHAOS library, Ponnusamy et al. [13, 14] proposed a heuristic method for irregular loop partitioning called *almost owner computes rule*, in which an iteration is executed on the processor that is the owner of the largest number of distributed array references in the iteration.

Some HPF compilers employ this scheme by using EXECUTE-ON-HOME clause [15]. However, when we parallelize a fluid dynamics solver ZEUS-2D code by using almost owner computes rule, we find that the almost owner computes rule is not optimal manner in minimizing communication cost — either communication steps or elements to be communicated. Another drawback is that it is not straightforward to choose optimal owner if several processors own the same number of array references.

We propose a more efficient computes rule for irregular loop partition [8]. This approach partitions iterations on a particular processor such that executing the iteration on that processor ensures

- the communication steps is minimum, and
- the total number of data to be communicated is minimum

In our approach, neither owner computes rule nor almost owner computes rule is used in parallel execution of a loop iteration for irregular computation. A communication cost reduction computes rule, called least communication computes rule, is proposed. For a given irregular loop, we first investigate for all processors $P_k, 0 \leq k \leq m$ (m is the number of processors) in which two sets $FanIn(P_k)$ and $FanOut(P_k)$ for each processor P_k are defined. $FanIn(P_k)$ is a set of processors which have to send data to processor P_k before the iteration is executed, and $FanOut(P_k)$ is a set of processors which have to send data from processor P_k after the iteration is executed. According to these knowledge we partition the loop iteration to a processor on which the minimal communication is ensured when executing that iteration. Then, after all iterations are partitioned into various processors. Please refer to [8] for details.

3 Global-local Array Transformation and Index Array Remapping

There are two approaches to generating SPMD irregular codes after loop partitioning. One is receiving required data in an iteration every time before the iteration is executed, and send the changed values (which are resident in other processors originally) to other processors after the iteration is executed. Another is gather all remote data from other processors for all iterations executed on this processor and scatter all changed remote values to other processors after all iterations are executed. Because message aggregation is a main communication optimization means, obviously the later one is better for communication performance. In order to perform communication aggregation, this section discusses redistribution of indirection arrays.

3.1 Index Array Redistribution

After loop partitioning analysis, all iterations have been assigned to various processors: $iter(P_0) = \{i_{0,1}, i_{0,2}, \ldots, i_{0,\alpha_0}\}, \ldots, iter(P_{m-1}) = \{i_{m-1,1}, i_{m-1,2}, \ldots, i_{m-1,\alpha_{m-1}}\}$. If an iteration i_r is partitioned to processor P_k, the index array elements $ix(i_r), iy(i_r), \cdots$ may not be certainly resident in P_k. Therefore, we need to redistribute all index arrays so that for $iter(P_k) = \{i_{k,1}, i_{k,2}, \ldots, i_{k,\alpha_k}\}$ and every index array ix, elements $ix(i_{k,1}), \ldots, ix(i_{k,\alpha_k})$ are local accessible.

As mentioned above, The source BLOCK partition scheme for processor P_k is $src\_iter(P_k) = \{\lceil \frac{g}{m} \rceil * k + 1..(\lceil \frac{g}{m} \rceil + 1) * k\}$, where g is the number of iterations of the loop. Then in a redistribution, the elements of an index array need to communicate from P_k to P'_k can be indicated by $src\_iter(P_k) \cap iter(P'_k)$. Back to the Example 1, same as Example 3, let the size of data array and index arrays be 12, after loop partitioning analysis, we can obtain $iter(P_0) = \{1, 5, 8, 9, 10\}, iter(P_1) = \{2, 3, 4\}$, and $iter(P_2) = \{6, 7, 11, 12\}$. The index array elements to be redistributed are shown in Figure 2.

3.2 Scheduling in Redistribution Procedure

A redistribution routine can be divided into two part: subscript computation and interprocessor communication. If there is no communication scheduling in a redistribution routine, communication contention may occur, which increases the communication waiting time. Clearly in each communication step, there are some processors sending messages to the same destination processor. This leads to node contention. Node contention will result in overall performance degradation.[5, 6] The scheduling can avoid this contention.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

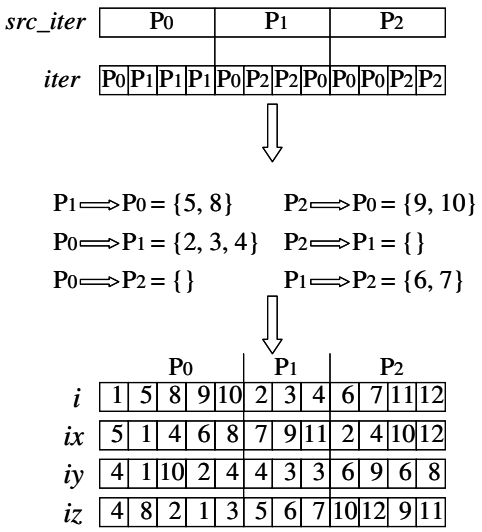

Figure 2. Subscript computations of index array redistribution in Example 1.

Another consideration is to align messages so that the size of messages as near as possible in a step, since the cost of a step is likely to be dictated by the length of the longest message exchanged during the step. We use a sparse matrix $\mathcal{M}$ to represent the communication pattern between the source and target processor sets where $\mathcal{M}(i,j) = M$ expresses a sending processor P_i should send an M size message to a receiving processor Q_j. The following matrix describes a redistribution pattern and a scheduling pattern.

$$\mathcal{M} = \begin{bmatrix} 0 & 3 & 0 & 10 & 3 \\ 10 & 0 & 2 & 1 & 2 \\ 2 & 2 & 0 & 3 & 9 \\ 10 & 0 & 2 & 0 & 0 \\ 0 & 4 & 3 & 3 & 0 \end{bmatrix}, CS = \begin{bmatrix} 1 & 4 & 3 & \\ 4 & 2 & 0 & 3 \\ 0 & 3 & 4 & 1 \\ & 0 & 2 & \\ 3 & 1 & & 2 \end{bmatrix}$$

We have proposed an algorithm that accepts $\mathcal{M}$ as input and generates a communication scheduling table CS to express the scheduling result where $CS(i,k) = j$ means that a sending processor P_i sends message to a receiving processor Q_j at a communication step k. The scheduling satisfies:

- there is no node contention in each communication step; and
- the sum of longest message length in each step is minimum.

4 Inter-procedural Communication Optimization for Irregular Loops

In some irregular scientific codes, an important optimization required is communication preprocessing among procedure calls. In this section, we extend a classical data flow optimization technique – Partial Redundancy Elimination – to an Interprocedural Partial Redundancy Elimination as a basis for performing interprocedural communication optimization [2]. Partial Redundancy Elimination encompasses traditional optimizations like loop invariant code motion and redundant computation elimination.

For irregular code with procedure call, initial intraprocedural analysis (see [8]) inserts pre-communicating call (including one buffering and one gathering routine) and post-communicating (buffering and scattering routine) call for each of the two data parallel loops in two subroutines. After interprocedural analysis, loop invariants and can be hoisted outside the loop.

Here, data arrays and index arrays are the same in loop bodies of two subroutines. While some communication statement may not be redundant, there may be some other communication statement, which may be gathering at least a subset of the values gathered in this statement.

In some situations, the same data array A is accessed using an indirection array IA in one subroutine $SUB1$ and using another indirection array IB in another subroutine $SUB2$. Further, none of the indirection arrays or the data array A is modified between flow control from first loop to the second loop. There will be at least some overlap between required communication data elements made in these two loops. Another case is that the data array and indirection array is the same but the loop bound is different. In this case, the first loop can be viewed as a part of the second loop.

We divide two kinds of communication routines for such situations. A common communication routine takes more than one indirection array, or takes common part of two indirection arrays. A common communication routine will take in arrays IA and IB producing a single buffering. Incremental preprocessing routine will take in indirection array IA and IB, and will determine the off-processor references made uniquely through indirection array IB and not through indirection array IA. While executing the second loop, communication using an incremental schedule can be done, to gather only the data elements which were not gathered during the first loop.

5 OpenMP Extensions for Irregular Applications

OpenMP's programming model uses fork-join parallelism: master thread spawns a team of threads as needed. Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program. Hence, we do not have to parallelize the whole program at once. OpenMP is usually used to parallelize loops. A user finds his most time

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

consuming loops in his code, and split them up between threads.

OpenMP is a model for programming any parallel architecture that provides the abstraction of a shared address space to the programmer. The purpose of OpenMP is to ease the development of portable parallel code, by enabling incremental construction of parallel programs and hiding the detail of the underlying hardware/software interface from the programmer. A parallel OpenMP program can be obtained directly from its sequential counterpart, by adding parallelization directives.

If the loop shown in Fig. 1 is split across OpenMP threads then, although threads will always have distinct values of j, the values of `prev_elam` and `next_elem` may simultaneously have the same values on different threads. As a result, there is a potential problem with updating the value of `new_cell`. There are some simple solutions to this problem, which include making all the updates atomic, or having each thread compute temporary results which are then combined across threads. However, for the extremely common situation of sparse array access neither of these approaches is particularly efficient.

5.1 Directive Extension for Irregular Loops

This section provides the new extensions to OpenMP, the `irregular` directive. The `irregular` directive could be applied to the `parallel do` directive in one of the following situations:

- When the parallel region is recognized as an irregular loop: in this case the compiler will invoke a runtime library which partitions irregular loop according to a special computes rule.
- When an `ordered` clause is recognized in the parallel region where the loop is irregular: in this case the compiler will treat this loop as a partial ordered; that is, some iterations are executed sequentially while some others may be executed in parallel.
- When an `reduction` clause is recognized in the parallel region where the loop is irregular: in this case the compiler will invoke an inspector/executor routines to perform irregular reduction in parallel.

The irregular directives in extended OpenMP version may have the following patterns:

```
  $!omp parallel do sched-
ule(irregular, [irarray1,...,irarrayN])
```

This case designates that the compiler will encounter an irregular loop, where `irarray1,...,irarrayN` are possible indirection arrays, or

```
  $!omp parallel do re-
duction|ordered irregu-
lar([expr1,...,exprN])
```

This case designates that the compiler will encounter a special irregular reduction or irregular ordered loop where `expr1, ..., exprN` are expressions such as loop index variables.

5.2 Partially Ordered Loops

A special case for the example in Fig. 1 occurs when the shared updates need not only to be performed in mutual exclusion, but also in an ordered way. The use of the `irregular` clause in this case tells the compiler that for those iterations which may update the same data in the different threads, they need to be executed in an ordered way. Other iterations are still executed in parallel. An example of code using the indirect clause in this manner is the following:

Example 1

```
$!omp parallel do ordered irregu-
lar(x, y)
do i = 1, n
   x[i] = indirect(1,i)
   y[i] = indirect(2,i)
$!omp ordered
  a(x[i]) = a(y[i]) ......
$!omp end ordered
end do
$!omp end parallel do
```

6 Conclusions

The efficiency of loop partitioning influences performance of parallel program considerably. For automatically parallelizing irregular scientific codes, the owner computes rule is not suitable for partitioning irregular loops. In this paper, we have presented an efficient loop partitioning approach to reduce communication cost for a kind of irregular loop with nonlinear array subscripts. In our approach, runtime preprocessing is used to determine the communication required between the processors. We have developed the algorithms for performing these communication optimization. We have also presented how interprocedural communication optimization can be achieved. Furthermore, if irregular codes are parallelized in OpenMP, we also proposed to extend OpenMP directives to be suitable for compiling and executing such codes. We have done a preliminary implementation of the schemes presented in this paper. The experimental results demonstrate efficacy of our schemes.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

References

[1] R. Allen and K. Kennedy. Optimizing compilers for Modern Architectures. Morgan Kaufmann Publishers, 2001.

[2] G. Agrawal and J. Saltz. Interprocedural compilation of Irregular Applilcations for Distributed memory machines. *Language and Compilers for Parallel Computing*, pp. 1-16, August 1994.

[3] R. Das, M. Uysal, J. Saltz, and Y-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[4] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May, 1999.

[5] M. Guo, I, Nakata, and Y. Yamashita. Contention-free communication scheduling for array redistribution. *Parallel Computing*, 26(2000), pp. 1325-1343, 2000.

[6] M. Guo and I. Nakata. A framework for efficient array redistribution on distributed memory machines. *The Journal of Supercomputing*, Vol. 20, No. 3, pp. 253-265, 2001.

[7] M. Guo, Y. Pan, and C. Liu. Symbolic Communication Set generation for irregular parallel applications. To appear in *The Journal of Supercomputing*, 2002.

[8] M. Guo, Z. Liu, C. Liu, L. Li. Reducing Communication cost for Parallelizing Irregular Scientific Codes. In *Proceedings of The 6th International Conference on Applied Parallel Computing*, Finland, June 2002.

[9] E. Gutierrez, R. Asenjo, O. Plata, and E.L. Zapata. Automatic parallelization of irregular applications. *Parallel Computing*, 26(2000), pp. 1709-1738, 2000.

[10] Y.-S Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software-Practivce and Experience*, Vol.25(6), pp. 597-621,1995.

[11] J.M. Stone and M. Norman. ZEUS-2D: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions: The hydrodynamic algorithms and tests. *Astrophysical Journal Supplement Series*, Vol. 80, pp. 753-790, 1992.

[12] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, pp. 3(3):5-40, 1989.

[13] R. Ponnusamy, Y-S. Hwang, R. Das, J. Saltz, A. Choudhary, G. Fox. Supporting irregular distributions in Fortran D/HPF compilers. Technical report CS-TR-3268, University of Maryland, Department of Computer Science, 1994

[14] R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang, and G. Fox. Runtime support and compilation methods for user-specified data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8), pp. 815-831, 1995.

[15] M. Ujaldon, E.L. Zapata, B.M. Chapman, and H.P. Zima. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*. 8(10), Oct. 1997. pp. 1068 -1083.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE