Data-Localization for Fortran Macro-Dataflow Computation Using Partial Static Task Assignment

Akimasa YOSHIDA, Kenichi KOSHIZUKA and Hironori KASAHARA
Department of Electrical, Electronics and Computer Engineering
Waseda University
3-4-1 Ohkubo Shinjuku-ku, Tokyo 169, Japan
E-mail: {yoshida,kosizuka,kasahara}@oscar.elec.waseda.ac.jp

Abstract

This paper proposes a data-localization compilation scheme for macro-dataflow computation, in which coarse-grain tasks such as loops, subroutines and basic blocks in a Fortran program are automatically processed in parallel on a multiprocessor system. The data-localization scheme reduces data transfer overhead for passing shared data among coarse-grain tasks composed of Doall loops and sequential loops by using local memory effectively. In this scheme, a compiler partitions coarse-grain tasks, or loops, having data dependences among them into multiple groups by a *loop aligned decomposition* so that data transfer among groups can be minimum, generates dynamic scheduling routine with partial static task assignment to assign decomposed tasks in a group to the same processor at run-time, and generates parallel machine code to pass shared data inside the group through local memory. A compiler has been implemented for an actual multiprocessor system OSCAR having centralized shared memory and distributed shared memory in addition to local memory on each processor. Performance evaluation on OSCAR shows that macro-dataflow computation with the proposed data-localization scheme can reduce the execution time by 10% to 20% average compared with ordinary macro-dataflow computation using centralized shared memory.

1 Introduction

Most Fortran parallelizing compilers for multiprocessor systems have been using loop parallelization techniques, such as Doall and Doacross[1, 2]. Currently, many types of Do-loops can be parallelized with support of strong data dependence analysis techniques[3, 4, 5, 6, 7]. There still exist, however, sequential loops which cannot be parallelized efficiently because of complex loop-carried data dependences and conditional branches to the outside loops. Also, parallelism outside Do-loops, for example, coarse grain parallelism among loops, subroutines and basic blocks, and (near) fine grain parallelism[8] inside a basic block[9] or a sequential loop, has not been effectively exploited by automatic compilers for multiprocessor systems.

Therefore, in order to improve the effective performance of multiprocessor systems, it is important to exploit the coarse grain parallelism[10] and also the (near) fine grain parallelism inside a sequential loops and a basic block[8], in addition to the medium grain parallelism among loop iterations exploited by the conventional loop parallelization. The coarse grain parallel processing on a multiprocessor system is also called the macro-dataflow computation[11, 12, 10, 13, 14, 15]. The macro-dataflow computation can be efficiently combined with the loop concurrentization and the near fine grain parallel processing hierarchically[16, 17, 15].

In parallel processing schemes like macro-dataflow where coarse grain tasks (macrotasks) are dynamically scheduled to processors (PEs) or processor clusters (PCs), shared data among macrotasks are generally allocated onto centralized shared memory (or ordinary common memory) and data transfers among macrotasks are performed via centralized shared memory. However, data transfer via centralized shared memory causes large overhead. Therefore, in order to reduce data transfer overhead, it is necessary that a compiler automatically decomposes data and computation and allocates them to local memory on each processor so that data transfer overhead can be minimum.

To this end, Tu and Padua[18], Eigenman[19], Li[20] proposed Array Privatization method, in which temporal array variables in a loop are allocated to local memory to reduce data transfer overhead. However, it can be applied only inside a loop.

Another popular approach for data decomposition and assignment on distributed memory multiprocessor systems is that user specifies distribution of data by using extended Fortran such as High Performance Fortran (HPF)[21] and Fortran D[22]. However, it is difficult for ordinary users to optimize both parallelism and data locality.

Considering this fact, recently, many researchers have been studying on automatic data partitioning. Li and Chen[23] showed how explicit communication can be synthesized and how communication costs are estimated by analyzing reference patterns in the source program. Ramanujam and Sadayappan[24] focused on partitioning a nested Doall loop so that the partitioned loops can be executed without communication overhead. Chen and Sheu[25] also presented communication-free partitions for a nested loop. However, these methods can be applied only to a nested loop. Meanwhile, Gupta and Banerjee[26] proposed automatic data decomposition for a whole program. Anderson and Lam[27] proposed automatic data and computation de-

ICS'96, Philadelphia, PA,USA

composition among loops when a compiler can allocate data and computation to processor using a linear transformation matrix. However, these schemes cannot be applied to a processing scheme in which computation and data are allocated dynamically, such as macro-dataflow computation.

Considering the above fact, this paper proposes a data-localization scheme to transfer data via local memory among macrotasks composed of Doall and sequential loops in macro-dataflow computation. The proposed scheme decomposes loops considering data dependence among iterations over different loops into multiple groups by *loop aligned decomposition* method. Then the compiler generates dynamic scheduling routine with partial static task assignment to schedule the decomposed loops inside a group which causes large data transfer to the same processor. Next the compiler generates parallel machine codes in which shared data among the decomposed loops scheduled to the same processor are transferred via local memory. A compiler has been implemented and its performance is evaluated on an actual multiprocessor system OSCAR(Optimally SCheduled Advanced multiprocessoR)[8].

Section 2 describes the macro-dataflow computation for a Fortran program. Section 3 proposes data-localization scheme among macrotasks composed of Doall and sequential loops. Section 4 evaluates performance of the proposed data-localization scheme on OSCAR.

2 Macro-Dataflow Computation

The macro-dataflow compilation scheme [10, 13, 17, 14, 28, 29, 30, 15, 31] mainly consists of the following four parts.

2.1 Generation of macrotasks (MTs)

In the macro-dataflow computation, a Fortran program is decomposed into three kinds of macrotasks (MTs), such as a Block of Pseudo Assignment statements (BPA), a Repetition Block (RB) and a Subroutine Block (SB).

A BPA is composed of a basic block or multiple basic blocks. A BPA composed of multiple basic blocks is generated by fusing small basic blocks. To the contrary, BPAs are also defined by decomposing a basic block into several blocks if a basic block has independent data dependence graphs inside. A RB is a Do-loop or a loop generated by a backward branch, namely, an outermost natural loop[9]. RBs are restructured by the proposed *loop aligned decomposition* method mentioned in section 3.1. As to subroutines, subroutines to which the in-line expansion technique cannot efficiently be applied, are defined as SBs.

2.2 Generation of macro flow graph (MFG)

Before detection of parallelism among macrotasks, control flow and data flow among macrotasks are analyzed and are represented by *Macro-Flow-Graph* (MFG)[10, 13, 17, 14] as shown in Figure 1. MFG is generally a directed acyclic graph since RBs contain all back edges inside them. In MFG, a node represents a macrotasks, a small circle inside a node shows a conditional branch, and a solid edge and a dotted edge represent data flow and control flow respectively.

2.3 Generation of macrotask graph (MTG)

The MFG explicitly represents the control flow and data flow among macrotasks though it does not show any parallelism among macrotasks. Generally, the control dependence graph, or the program dependence graph[32], represents maximum parallelism if there are not data dependences among macrotasks[33]. In practice, there exist, however, data dependences among macrotasks. Therefore, in order to effectively extract parallelism among macrotasks from a macro-flow-graph, the control dependences and the data dependences should be analyzed together.

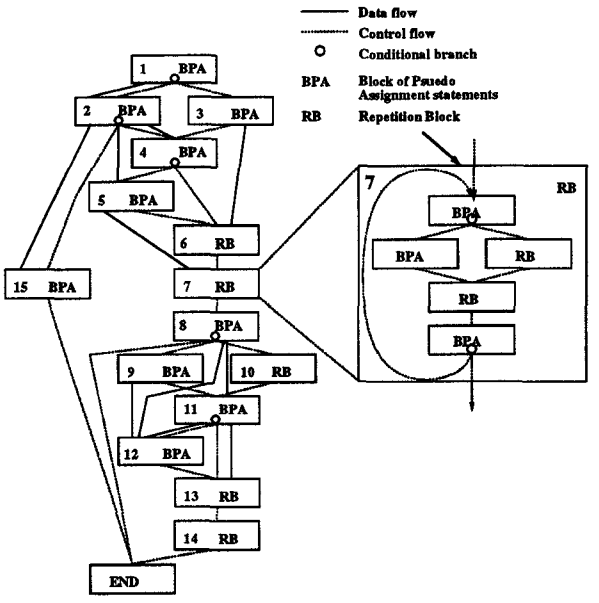

Figure 1: MFG of a sample program

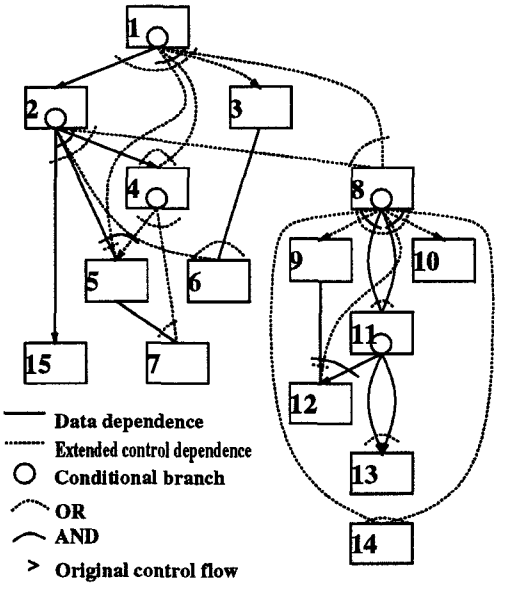

Figure 2: MTG of a sample program

In this paper, an *earliest-executable-condition* [10, 13, 17, 15] of each macrotask is used to find the maximum parallelism among macrotasks considering control dependences and data dependences. The earliest-executable-condition of a macrotask i (MT_i) is a condition on which MT_i may begin its execution earliest after data and control depen-

dences are satisfied. For example, an earliest-executable-condition of MT_6 in Figure 1 is "MT_3 completes execution OR MT_2 branches to MT_4". Girkar and Polychronopoulos [34] modified the original earliest executable condition analysis[10, 13, 17] assuming a conditional branch inside a macrotask is executed in the end of the macrotask[3].

The earliest-executable-conditions of macrotasks are represented by a directed acyclic graph called *MacroTask-Graph* (MTG) [10, 13, 17, 14] as shown in Figure 2.

2.4 Generation of dynamic scheduling routine

Next, the compiler generates a dynamic scheduling routine to schedule macrotasks onto processors (PEs) or processor clusters (PCs) at run-time. Dynamic scheduling is adopted to cope with runtime uncertainties, such as conditional branches among macrotasks and a variation of macrotask execution time. As a dynamic scheduling algorithm, Dynamic-CP algorithm[17], which uses scheduling priority based on estimation of longest path length from each node to the exit node on MTG, is applied.

3 Data-Localization among Macrotasks Composed of Doall and Sequential Loops

This section proposes a data-localization scheme to reduce data transfer overhead among macrotasks composed of Doall and sequential loops. In this paper, data-localization means to decompose multiple loops, or array data, and to assign them to processors so that shared data among the macrotasks can be transferred through local memory on the processors.

This compilation method consists of the following three steps: *loop aligned decomposition* which decompose loop index and arrays to minimize data transfer among processors, *generation of dynamic scheduling routine with partial static task assignment* to assign a set of decomposed loops among which large data transfer may occur onto the same processor, and *generation of parallel machine-code* to transfer data via local memory among the decomposed loops assigned onto the same processor.

3.1 Loop aligned decomposition

In the definition of RB in section 2.1, a Doall loop is assigned to a processor as a macrotask. To avoid this situation and to exploit parallelism of a Doall loop, the compiler decomposes the Doall loop into n small Doall loops (or macrotasks), where n is a number or multiple numbers of processors. These decomposed small Doall loops are executed on processors in parallel.

However, in a set of Doall and sequential loops which are connected by data dependence edge in MTG, if Doall loops are decomposed and executed on processors and sequential loops are not decomposed and are executed on a single processor, then a large amount of data should be transferred among processors.

Therefore, this section proposes *loop aligned decomposition* considering sequential loops, which decomposes sequential loops as well as Doall loops so that data can be transferred via local memory (LM) among decomposed Doall and sequential loops.

3.1.1 Detection of target-loop-group for loop aligned decomposition

First, we find a set of RBs or *Target-Loop-Group* (TLG) to which *loop aligned decomposition* is applied. The TLG is composed of RBs satisfying the following conditions.

(i)RBs are connected by a single data dependence edge related with array variables on MTG. Here, a data dependence edge from small BPA which initializes scalar variables used in RB may exist.

(ii) Each RB (Outermost loop) is a Doall loop, a reduction loop, or a sequential loop with loop carried data dependence. Also, it is assumed each RB has been normalized to have stride 1.

(iii) Array subscript in RB is expressed by a linear function of an index variable.

In the above conditions, it is assumed that suitable loop restructuring techniques[2, 35] such as loop interchange, loop fusion, loop distribution and so on have been applied to each RB before this TLG detection phase. A set of RB_1, RB_2 and RB_3 in Figure 3(a) is an example of TLG.

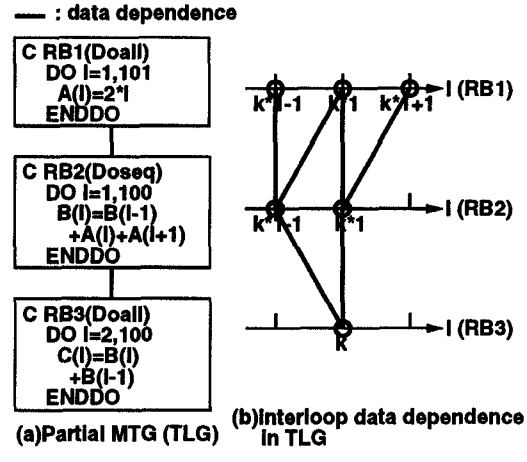

Figure 3: Target-loop-group (TLG) with a sequential loop

3.1.2 Interloop data dependence analysis in target-loop-group

For each TLG composed of "m" macrotasks or $RB_i(1 \leq i \leq m)$, the compiler analyzes data dependence among iterations of RB_i $(1 \leq i \leq m)$. For example, in case of the TLG in Figure 3(a), k-th iteration of RB_3 (e.g. 34th iteration in Figure 4) is data dependent on $(k-1)$-th (e.g. 33rd) and k-th (e.g. 34th) iterations of RB_2 by array B as shown in Figure 3(b). Also, k-th (e.g. 34th in Figure 4) iteration of RB_2 is data dependent on k-th (e.g. 34th) and $(k+1)$-th (e.g. 35th) iteration of RB_1 by array A.

Here we call this kind of data dependence *Direct Inter-Loop Data dependence* and represent it as $DirILD(RB_i, RB_j, k)$, which means the loop indices of iterations in RB_i on which k-th iteration in RB_j is data dependent. Each elements of $DirILD(RB_i, RB_j, k)$ is represented by linear function of k. In the example of Figure 3(a), $DirILD(RB_2, RB_3, k) = \{k-1, k\}$, and $DirILD(RB_1, RB_2, k) = \{k, k+1\}$ as shown in Figure 3(b).

Next, the compiler analyzes index ranges of iterations in $RB_i(1 \leq i \leq m-1)$ on which k-th iteration of RB_m (exit node in TLG) is directly

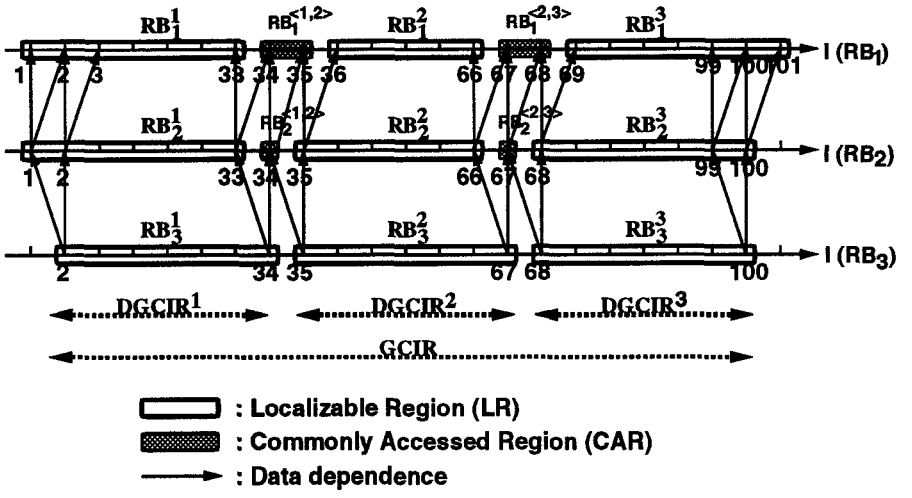

Figure 4: An example of loop aligned decomposition

or indirectly data dependent. Then, the compiler represents the result of analysis by *direct and indirect InterLoop Data dependence* expressed by $ILD(RB_t, RB_m, k)$. Note that loop carried data dependence inside sequential loop (RB) is not considered while this analysis is performed. Each element of $ILD(RB_i, RB_m, k)$ is also represented by a set of linear function of k.

The algorithm to analyze $ILD(RB_t, RB_m, k)$ is shown in the following. Here, $SucDtDepRB(RB_i)$ represents a set of succeeding RBs in TLG each of which is data dependent on RB_t.

$$ILD(RB_m, RB_m, k) = \{k\}.$$

for $i := m-1$ *to* 1 *do*

$$ILD(RB_t, RB_m, k) = \bigcup_{RB_j \in SucDtDepRB(RB_i)} \left(\bigcup_{t \in ILD(RB_j, RB_m, k)} DirILD(RB_t, RB_j, t) \right).$$

In an example in Figure 3(a), $ILD(RB_i, RB_3, k)$ $(1 \leq i \leq 3)$ are analyzed as follows; $ILD(RB_3, RB_3, k) = \{k\}$, $ILD(RB_2, RB_3, k) = \bigcup_{t \in ILD(RB_3, RB_3, k)} DirILD(RB_2, RB_3, t) = \{k-1, k\}$, and $ILD(RB_1, RB_3, k) = \bigcup_{t \in ILD(RB_2, RB_3, k)} DirILD(RB_1, RB_2, t) = \{k-1, k, k+1\}$. Namely, k-th (e.g.34th in Figure 4) iteration of RB_3 is data dependent on $(k-1)$-th and k-th (e.g. 33rd and 34th) iterations of RB_2 and on $(k-1)$-th through $(k+1)$-th (e.g. 33rd through 34th) iterations of RB_1.

In the calculation of $DirILD$ and ILD, lower-bound and upper-bound of elements in $DirILD$ and ILD are used instead of a set of elements to reduce calculation time. Also, if several iterations of RB_t exist between $ILD(RB_t, RB_m, k)$ and $ILD(RB_t, RB_m, k+1)$, those iterations are joined into $ILD(RB_i, RB_m, k)$.

3.1.3 Calculation of group-converted-index-range in target-loop-group

Next, the compiler calculates *Group-Converted-Index-Range* ($GCIR$), which represents index range of array data used (or defined) in all RBs inside the TLG as a loop index range of RB_m.

First, the compiler converts loop index range of $RB_i (1 \leq i \leq m)$, or $IR(RB_t)$, into loop index range of the RB_m, or *Converted-Index-Range* ($CIR(RB_t)$). Namely, $CIR(RB_i)$ is determined as a set of indices to satisfy the following equation.

$$IR(RB_i) = \bigcup_{t \in CIR(RB_t)} ILD(RB_i, RB_m, t).$$

Secondly, the compiler calculates $GCIR$ as follows;

$$GCIR = \bigcup_{1 \leq i \leq m} CIR(RB_t).$$

For example, in case of Figure 3(a), $CIR(RB_1) = CIR(RB_2) = CIR(RB_3) = [2 : 100]$ as shown in Figure 4. Note that $[x : y]$ denotes the range of x-th index (lower-bound) through y-th index (upper-bound). Therefore $GCIR = CIR(RB_1) \cup CIR(RB_2) \cup CIR(RB_3) = [2 : 100]$.

3.1.4 Decomposition of RBs in target-loop-group

By using $GCIR$ and $ILD(RB_t, RB_m, k)$, the compiler decomposes each $RB_t (1 \leq i \leq m)$ into *Localizable-Regions* (LRs) and *Commonly-Accessed-Regions* (CARs) as follows. LR is a partial set of iterations in RB_i on which only a succeeding decomposed RB is data dependent when RBs are decomposed. CAR is a set of iterations in RB_t on which plural succeeding decomposed RBs are commonly data dependent. Note that CAR is not generated if there are not iterations in RB_t on which several succeeding decomposed RBs are commonly data dependent.

Concretely, first, the compiler evenly decomposes $GCIR$ into n partial ranges, or $DGCIR^p (1 \leq p \leq n)$, where n is a number or a multiple number of processors. For example,

when RBs in Figure 3(a) are executed on 3 processors and $n = 3$, $GCIR(= [2 : 100])$ is decomposed into $DGCIR^1(= [2 : 34])$, $DGCIR^2(= [35 : 67])$ and $DGCIR^3(= [68 : 100])$ as shown in Figure 4.

Secondly, the compiler generates *CARs*, or $RB_i^{<p,p+1>}$ $(1 \leq p \leq n-1)$, for each $RB_i(1 \leq i \leq m)$. Loop index range of $RB_i^{<p,p+1>}$, or $IR(RB_i^{<p,p+1>})$, is calculated as follows.

$$IR(RB_i^{<p,p+1>}) = \left(\bigcup_{t \in DGCIR^p} ILD(RB_i, RB_m, t) \right) \cap \left(\bigcup_{t \in DGCIR^{p+1}} ILD(RB_i, RB_m, t) \right).$$

Thirdly, the compiler generates *LRs*, or $RB_i^p(1 \leq p \leq n)$, for each $RB_i(1 \leq i \leq m)$. The compiler calculates loop index range of RB_i^p, or $IR(RB_i^p)$ as follows.

$$IR(RB_i^p) = \left(\bigcup_{t \in DGCIR^p} ILD(RB_i, RB_m, t) \right) - IR(RB_i^{<p-1,p>}) - IR(RB_i^{<p,p+1>})$$

where compiler initializes $IR(RB_i^{<0,1>}) = \emptyset$, $IR(RB_i^{<n,n+1>}) = \emptyset$ in advance.

Fourthly, the compiler examines whether loop index range of each decomposed RB (e.g. $IR(RB_i^p)$ or $IR(RB_i^{<p,p+1>})$) is included in loop index range of RB_i. If $IR(RB_i^p)$ or $IR(RB_i^{<p,p+1>})$ (the decomposed RB for RB_i) includes iterations outside $IR(RB_i)$, these iterations are removed from the index range of decomposed RB.

In an example in Figure 4, as to generation of *CAR*, iterations of RB_1 (or RB_2) on which both a set of iterations of a partial RB_m (whose loop index range is $DGCIR^1$) and a set of iterations of the other partial RB_m (whose loop index range is $DGCIR^2$) are commonly data dependent are defined as $RB_1^{<1,2>}$ (or $RB_2^{<1,2>}$) respectively. Next, as *LRs*, RB_1^1, RB_2^1 and RB_3^1 are generated by using $DGCIR^1$, and RB_1^2, RB_2^2 and RB_3^2 are generated by using $DGCIR^2$. Loop index ranges of each decomposed RB are shown in Figure 4.

3.1.5 Generation of data-localization-group

After *loop aligned decomposition* is applied, the compiler defines a set of *LRs*, among which large data transfer are required, as a *Data-Localization-Group* (*DLG*) where array data are passed through local memory. At this time, a *CAR* is fused into an adjacent *LR* to reduce dynamic scheduling overhead. Figure 5 shows *DLGs* corresponding to Figure 4.

As shown later, these macrotasks inside *DLG* are scheduled to the same processor at run-time by dynamic scheduling routine with partial static task assignment. Moreover, among the macrotasks inside *DLG* assigned to the same processor, shared data are transferred via LM as described later.

3.2 Generation of dynamic scheduling routine with partial static task assignment

As mentioned before, in order to transfer shared data among macrotasks via local memory, macrotasks inside *DLG* are

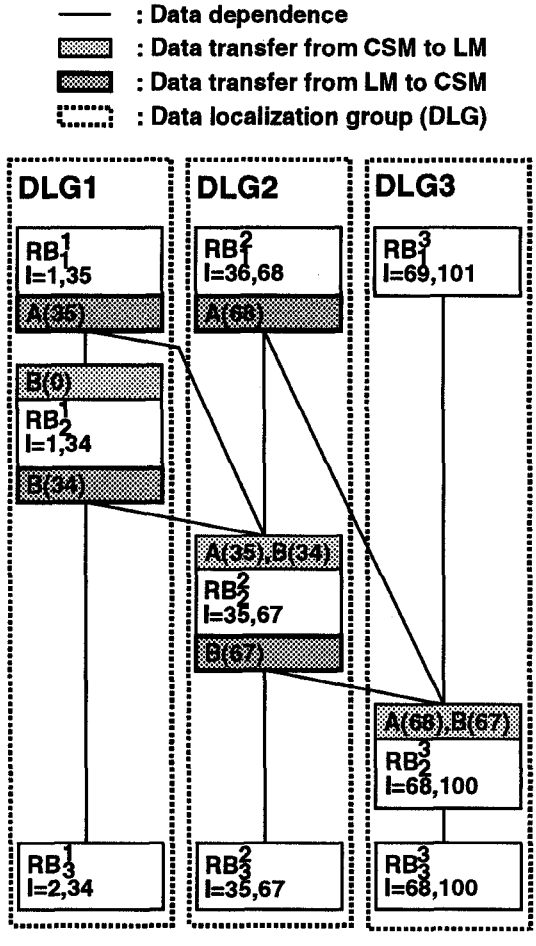

Figure 5: Data-localization-group (DLG) and data transfers among macrotasks

scheduled to the same processor at run-time. However, in ordinary Dynamic-CP method[15][31] which is used as dynamic scheduling algorithm in macro-dataflow computation, macrotasks in DLG are not always scheduled to the same processor. Therefore, the proposed dynamic scheduling method uses a partial static task assignment scheme to schedule macrotasks in DLG to the same processor at run-time. In the following, the proposed dynamic scheduling using Dynamic-CP algorithm with partial static task assignment is explained more concretely.

The execution order for macrotasks in each DLG is decided by data dependence among them uniquely. So, after entrance macrotask ($MT_{(d,entrance)}$) in a DLG_d is scheduled to a processor PE_p at run-time, scheduler assigns the other macrotasks ($MT_{(d,i\neq entrance)}$) in the DLG_d to the same PE_p. To realize this procedure, when the compiler generates a dynamic scheduling routine, the compiler specifies succeeding $MT_{(d,i\neq entrance)}$ should be scheduled to the same processor with $MT_{(d,entrance)}$. In this approach, to avoid load unbalance among processors, dynamic scheduler assigns $MT_{(d,entrance)}$ in DLG to a processor having smallest load. Here, load of processor is estimated considering processing time for both a running macrotask on the processor and macrotasks in DLG to be scheduled to the processor.

In current implementation, it takes about 90 clocks to schedule a macrotask to a processor by using the proposed dynamic scheduling routine with partial static task assignment. Namely, the overhead of dynamic scheduling with partial static task assignment is very small.

3.3 Generation of Data-transfer code via local memory in data-localization-group

First, the compiler estimates data transfer time via local memory (LM), or $t^x_{localize}$, and data transfer time via centralized shared memory (CSM), or t^x_{CSM}, for each array variables x which are used or defined in loops inside DLG. For array x satisfying $t^x_{localize} < t^x_{CSM}$, the compiler generates data transfer code via LM as follows. Meanwhile, for array x not satisfying $t^x_{localize} < t^x_{CSM}$, the compiler generates data transfer code via CSM.

Next, for each array to be localized among loops in a DLG, the compiler generates store or load instructions to LM in these loops. Also, the compiler generates machine code to transfer data on CSM to LM before these data are used by processor. For instance, when data-localization is applied to arrays A and B in the TLG shown in Figure 5, data transfer code from CSM to LM is generated as shown by light shaded part of each macrotask inside DLG.

Also, if data on LM in DLG are used by another DLG, the compiler inserts data transfer code from LM to CSM. For example, the compiler generates data transfer code as shown by dark shaded part of each macrotask inside DLG in Figure 5.

4 Performance Evaluation on OSCAR

This section describes performance evaluation of the proposed data-localization scheme on OSCAR[17, 8, 36].

4.1 OSCAR's architecture

OSCAR was developed in 1987 by the authors and Fuji Facom Corp.. It is a multiprocessor system having centralized and distributed shared memories in addition to local program and data memories as shown in Figure 6. Its processor elements (PEs) and a Control and I/O processors are uniformly connected to centralized shared memory (CSM) through three buses. Each PE has a custom-made 32 bit RISC processor, distributed shared memory (DSM) and the local memories (LM). On OSCAR, it takes 4 clocks to store (or load) one word data to CSM or DSM on the other PE, and 1 clock to store (or load) to LM or DSM inside PE.

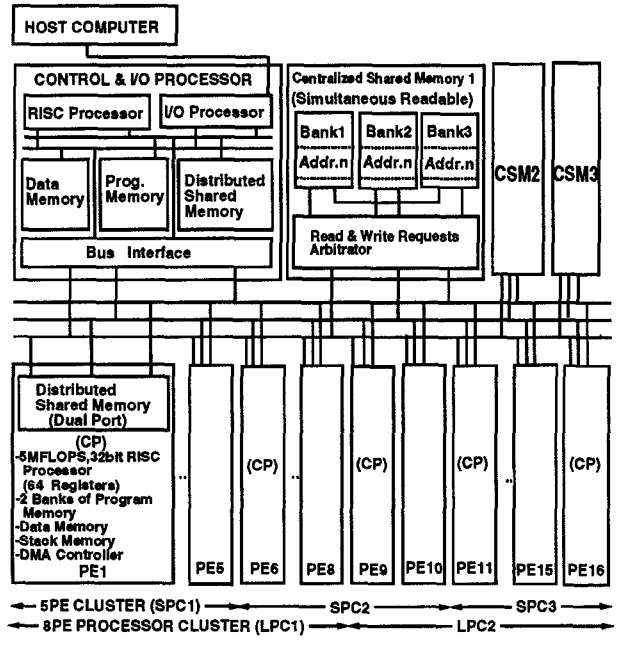

Figure 6: OSCAR's architecture

4.2 Performance evaluation using Spline Interpolation program

In this performance evaluation, a Fortran program for Spline Interpolation, having 9 Doall loops, 2 sequential loops with loop carried data dependence and 3 basic blocks is used. Figure 7 shows macrotask-graph (MTG) for this program. Table 1 shows the execution result of this program on OSCAR.

Table 1: Performance evaluation using Spline Interpolation program

PE	Processing Schemes	time [ms]	speed up vs. 1 PE
1	Sequential processing	632	1/1.00
3	Doall processing	284	1/2.23
3	Macro-dataflow	246	1/2.57
3	Macro-dataflow with data-localization	187	1/3.38
6	Doall processing	218	1/2.90
6	Macro-dataflow	188	1/3.36
6	Macro-dataflow with data-localization	152	1/4.16

Conventional Doall processing reduces execution time from 632ms for 1 PE to 284ms (1/2.23) for 3 PEs and to

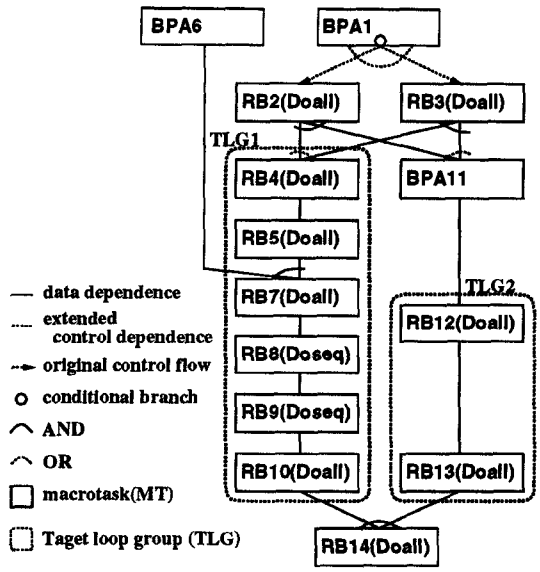

Figure 7: MTG of a program for Spline Interpolation

218ms (1/2.90) for 6 PEs. On the other hand, ordinary macro-dataflow computation without data-localization reduces execution time to 246ms (1/2.57) for 3 PEs and to 188ms (1/3.36) for 6 PEs because parallelism among sequential loops and the other macrotasks can be exploited.

Furthermore, when the proposed data-localization method is applied to the target-loop-group on MTG in Figure 7, execution time on 3 PEs is reduced to 187ms (1/3.38) for 3 PEs and to 152ms (1/4.16) for 6 PEs. In other words, speedup of 23.9% for 3 PEs and 19.1% for 6 PEs are obtained by the data-localization compared with ordinary macro-dataflow computation.

4.3 Performance evaluation using CFD program

Next, we evaluate the proposed scheme by using a Computational-Fluid-Dynamics (CFD) program based on IAF(Implicit Approximate Factorization) algorithm. CFD program is a 1400-line Fortran program. In this evaluation, a part of the program having 7 loops is used.

Table 2: Performance evaluation using CFD program

PE	Processing Schemes	time [s]	speed up vs. 1 PE
1	Sequential processing	4.452	1/1.00
2	Macro-dataflow	2.231	1/1.99
2	Macro-dataflow with data-localization	1.868	1/2.36
4	Macro-dataflow	1.176	1/3.78
4	Macro-dataflow with data-localization	0.987	1/4.51
6	Macro-dataflow	0.815	1/5.46
6	Macro-dataflow with data-localization	0.721	1/6.17

The execution time of this program on OSCAR is indicated in Table 2. The sequential execution time of this program on 1 PE is 4.452[s]. In this case, all array data are initially allocated onto centralized shared memory because array data size is larger than local memory size on 1 PE. Ordinary macro-dataflow computation reduces processing time to 2.231s (1/1.99) for 2 PEs, to 1.176s (1/3.78) for 4 PEs, to 0.815s (1/5.46) for 6 PEs. On the other hand, macro-dataflow computation with data-localization reduces to 1.868s (1/2.36) for 2 PEs, to 0.987s (1/4.51) for 4 PEs, to 0.721 (1/6.17) for 6 PEs. In other words, speedup of 16.2% for 2 PEs, 16.0% for 4 PEs and 11.5% for 6 PEs are obtained by the data-localization compared with ordinary macro-dataflow computation.

In the above evaluation, OSCAR needs only 4 clocks to access CSM and 1 clock to access LM. However, since ratio of CSM access time to LM access time is large on multiprocessor system available in the market, the proposed data-localization scheme may be more effective on these machines.

Currently, the authors are rewriting the prototype compiler to a practical version which can compile large scale application programs and generate parallelized program written in VPP Fortran, KSR Fortran, MPI etc. in addition to OSCAR parallel machine code.

5 Conclusions

This paper proposes the data-localization scheme for macro-dataflow computation on multiprocessor system having shared memory and local memory. The data-localization scheme is composed of *loop aligned decomposition*, generation of dynamic scheduling routine with partial static task assignment, and generation of machine-code to pass shared data among loops through local memory.

Performance evaluations on OSCAR showed that macro-dataflow computation with data-localization could reduce execution time by 23.9% for Spline Interpolation program and by 16.2% for a partial program of Computational Fluid Dynamics program compared with ordinary macro-dataflow computation without data-localization. From these results, effectiveness of the proposed data-localization scheme using partial static task assignment was confirmed.

Currently, the authors are researching on combination of data-localization technique and data pre-loading and data post-storing technique to hide data transfer overhead by overlapping computation and data transfer for macro-dataflow computation.

Acknowledgements

This research was partly supported by the Ministry of Education through the Grant-in-Aid for Scientific Research No.(B)05452354 and No.(C)07680372.

References

[1] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for super computers. *Commun. ACM*, 29(12):1184–1201, 1986.

[2] M. Wolfe. Optimizing supercompilers for supercomputers. *MIT press*, 1989.

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic program parallelization. *Proc. of IEEE*, 81(2):211–243, Feb. 1993.

[4] D.A. Padua, D.J. Kuck, and D.H. Lawrie. High-speed multiprocessor and compilation techniques. *IEEE Trans. Comput.*, C-29(9):763–776, 1980.

[5] M. Wolfe. High performance compilers for parallel computing. *Addison-Wesley Publishing Company*, 1996.

[6] U. Banerjee. Dependence analysis for supercomputing. *Kluwer Academic Pub.*, 1988.

[7] U. Banerjee. Loop parallelization. *Kluwer Academic Pub.*, 1994.

[8] H. Kasahara, H. Honda, and S. Narita. Parallel processing of near fine grain tasks using static scheduling on OSCAR. *IEEE ACM Supercomputing'90*, 1990.

[9] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers (principles, techniques, and tools). *Addison Wesley*, 1988.

[10] H. Honda, M. Iwata, and H. Kasahara. Coarse grain parallelism detection scheme of Fortran programs. *Trans. IEICE(in Japanese)*, J73-D-I(12):951–960, 1990.

[11] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. Cedar. *Report UIUCDCS-R-83-1123, Dept. of Computer Sci., Univ. Illinois at Urbana-Champaign*, Feb. 1983.

[12] D.D. Gajski, D.J. Kuck, and D.A. Padua. Dependence driven computation. *Proc. of COMPCON 81 Sprint Computer Conf.*, pages 168–172, 1981.

[13] H. Kasahara, H. Honda, M. Iwata, and M. Hirota. A compilation scheme for macro-dataflow computation on hierarchical multiprocessor systems. *Proc. Int. Conf. on Parallel Processing*, 1990.

[14] H. Honda, K. Aida, M. Okamoto, A. Yoshida, W. Ogata, and H. Kasahara. Fortran macro-dataflow compiler. *Proceedings of Fourth Workshop on Compilers for Parallel Computers*, Dec. 1993.

[15] H. Kasahara. Parallel processing technology. *Corona Pub. in Japan*, 1991.

[16] H. Kasahara, H. Honda, K. Aida, M. Okamoto, and S. Narita. OSCAR Fortran compiler. *Proc. Workshop on Compilation of (Symbolic) Languages for Parallel Computers in 1991 Int. Logic Programming Symposium*, 1991.

[17] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K.Fujiwara, and S.Narita. Multi-grain parallelizing compilation scheme for OSCAR. *4th Workshop on Language and Compilers for Parallel Computing*, 1991.

[18] P. Tu and D. Padua. Automatic array privatization. *6th Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.

[19] R. Eigenmann. Toward a methodology of optimizing programs for high-performance computers. *Proc. of ACM International Conference on Supercomputing'93*, pages 27–36, Jul. 1993.

[20] Z. Li. Array privatization for parallel execution of loops. *Proc. of the 1992 ACM Int. Conf. on Supercomputing*, pages 313–322, 1992.

[21] High Performance Fortran Forum. High performance Fortran language specification draft ver.1.0. *High Performance Fortran Forum*, 1993.

[22] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran D programming system. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, 1991.

[23] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Trans. on Parallel and Distributed System*, 2(3):361–376, 1991.

[24] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE trans. on parallel and distributed systems*, 2(4), 1991.

[25] T.-S. Chen and J.-P. Sheu. Communication-free data allocation techniques for parallelizing compilers on multicomputers. *IEEE trans. on parallel and distributed systems*, 5(9), 1994.

[26] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. on Parallel and Distributed System*, 3(2):179–193, 1992.

[27] J.M. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. *Proc. of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, 1993.

[28] L. Bic, A. Nicolau, and M.Sato (ed). Parallel language and compiler research in japan. *Kluwer Academic Pub.*, 1995.

[29] K. Aida, K. Iwasaki, H. Kasahara, and S. Narita. Performance evaluation of macro-dataflow computation on shared memory multiprocessors. *Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, 1995.

[30] M. Okamoto, K. Yamashita, H. Kasahara, and S. Narita. Hierarchical macro-dataflow computation scheme on a multiprocessor system OSCAR. *Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, 1995.

[31] H. Honda, K. Aida, M. Okamoto, and H. Kasahara. Coarse grain parallel execution scheme of a Fortran program on OSCAR. *Trans. IEICE(in Japanese)*, J75-D-I(8):526–535, 1992.

[32] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Syst.*, 9(3):319–349, 1987.

[33] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for determining useful parallelism. *Proc. 2nd ACM Int. Conf. on Supercomputing*, 1988.

[34] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on Parallel and Distributed System*, 3(2):166–178, 1992.

[35] U. Banerjee. Loop transformations for restructuring compilers. *Kluwer Academic Pub.*, 1993.

[36] W. Ogata, K. Fujimoto, M. Oota, and H. Kasahara. Compilation scheme for near fine grain parallel processing on a multiprocessor system without explicit synchronization. *Proceedings of IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, 1995.