Minimizing Data Size for Efficient Data Reuse in Grid-enabled Medical Applications*

Fumihiko Ino[1], Katsunori Matsuo[1], Yasuharu Mizutani[2], and Kenichi Hagihara[1]

[1] Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
ino@ist.osaka-u.ac.jp
[2] Faculty of Information Science and Technology, Osaka Institute of Technology

Abstract. This paper presents a data minimization method that aims at reducing overhead for data reuse in grid environments. The data reuse here is designed to promote efficient use of grid resources by avoiding multiple executions of the same computation in a collaborative community. To promote this at the program block level, our method minimizes the data size of attribute values, which are used for identification of computation products stored in a database (DB) server. Because attribute values are specified in queries used for store, search, or retrieval of computation products, their reduction leads to less communication between computing nodes and the DB server, minimizing the runtime overhead of data reuse. We also show some experimental results obtained using a time-consuming medical application. We find that the method successfully reduces the data size of a query from 683 MB to 52 B. This reduction allows our data reuse framework to reduce execution time from approximately 9 minutes to 27 seconds.

1 Introduction

With the rapid advance of network technology, the computational grid [1] is emerging as an attractive platform for computational scientists. For example, grid technology allows us to build high performance computing environments in virtual organizations. The key role of grid technology here is to make it possible to share computational resources and database (DB) contents in a specific virtual community constructed over the grid.

In contrast, some researchers are trying to share computation products in addition to hardware and software resources mentioned above. This data sharing approach avoids multiple executions of the same computation submitted by (usually different) users in the collaborative community. Therefore, grid resources are dedicated to produce new data, achieving highly efficient use of resources. For example, Quantum Chemistry Grid [2] provides a grid-enabled problem solving environment capable of accumulating computation products obtained by a computational chemistry program. Since this environment focuses on a single program, data reuse can easily be realized by constructing a DB where each of computation products is associated with attributes, namely the inputs

* This work was partly supported by JSPS Grant-in-Aid on Priority Areas (170320007), for Scientific Research (B)(2)(18300009), and for Young Researchers (17700060).

given to the program. Attribute values are then specified in queries to store, search, or retrieve computation products in the DB.

On the other hand, Pegasus [3, 4] integrates a data reuse functionality into a workflow mapping system. A workflow here abstracts the processing sequence of data in discrete steps. It consists of vertices and edges, representing application components and their flow dependencies, respectively. Before mapping a workflow onto grid resources, the system eliminates vertices in the workflow if the corresponding application components have previously been executed with the same inputs. This elimination achieves higher efficiency by replacing repetitive executions with data retrieval from the DB.

Thus, data reuse capabilities are useful to avoid wasteful executions in a virtual community. However, data reuse is usually done at the program level. Therefore, the efficiency can be further improved if data is reused within a program, for example, at the block level. The problem addressed in the paper is to realize this block-level reuse at a low overhead. We think that data reuse should work at a finer granularity to perform data reuse at the appropriate granularity such as blocks, functions, or programs, chosen according to the tradeoff between its repeatability and time saved by reuse.

In this paper, we present a data minimization method that aims at reducing overhead for block-level reuse in grid environments. Our method requires users to specify the program code for data reuse, and then minimizes the data size of attribute values, which are used for identification of computation products stored in a DB server. The key idea of our method is data dependence analysis that aims at replacing the initially specified code with an extended code that requires less amount of attribute values for data reuse.

2 Related Work

Similar to Pegasus [3, 4], the GriPhyN virtual data system (VDS) [5] also provides a data reuse capability to data-intensive applications. The VDS allows users to discover and share virtual data products, compose workflows, and monitor workflow executions. However, their data reuse model does not consider complex workflows having branches or iterations. Dealing with these control flows is required to reuse computation products for program blocks.

Altintas et al. [6] also realize a data reuse functionality for grid workflow systems. Their functionality is designed to reuse workflows rather than computation products. Therefore, their concern is the share of knowledge on effective workflows across different scientific fields.

The AppLeS (Application Level Scheduling) Parameter Sweep Template (APST) [7] maximizes reuse of shared data files by using adaptive scheduling techniques for parameter sweep applications. It employs a replication strategy to minimize data transmission between the client machine and computing nodes. Thus, this reuse functionality works at the scheduling level, trying to place data files to maximize data reuse. With respect to minimization of data transmission, our method also addresses the same problem. However, our method differs from APST, which does not minimize the data size itself. Another replication-based strategy is also presented in [8].

Some researchers resolve the problem of data reuse in caches. Strout et al. [9] performs data reuse to accelerate Gauss-Seidel methods. Their method improves intra-

Table 1. Notations.

Notation	Explanation
V	A set of vertices
E	A set of edges
G	$G = (V, E)$ defines a directed acyclic graph representing a workflow
(u, v)	$(u, v) \in E$ represents an edge from vertex u to vertex v
$label(v)$	The name labeled on vertex v
$label(u, v)$	The name labeled on edge (u, v)
$value(u, v)$	Values (contents) of name $label(u, v)$
I_v	$I_v = \{(u, v) \mid \exists u$ such that $(u, v) \in E\}$ defines the set of edges incoming to vertex v
O_v	$O_v = \{(v, w) \mid \exists w$ such that $(v, w) \in E\}$ defines the set of edges outgoing from vertex v
$\mathcal{S}_v$	$\mathcal{S}_v = \{\langle label(u, v), value(u, v)\rangle \mid (u, v) \in I_v\}$ defines the set of pairs of labels and values passed to vertex v
$\mathcal{T}_v$	$\mathcal{T}_v = \{\langle label(v, w), value(v, w)\rangle \mid (v, w) \in O_v\}$ defines the set of pairs of labels and values passed from vertex v
R	$R \subset V$ denotes the set of vertices initially marked as the target code for data reuse
C_R	$C_R \subseteq R$ denotes the set of critical vertices that have attribute information for R, where $C_R = \{v \in R \mid \exists u$ such that $u \notin R \land (u, v) \in E\}$
$\mathcal{A}_R$	$\mathcal{A}_R = \bigcup_{v \in C_R} \mathcal{S}_v$ denotes the initial attributes for user-specified R

iteration and inter-iteration data locality in iterative solvers. Issenin et al. [10] performs data reuse in a more explicit manner. They make copies of frequently used data in a small local memory. Although these kinds of data reuse contribute to acceleration of specific applications, their computation products are shared during a single execution. In contrast, we focus on data sharing across multiple executions submitted from different grid users.

Thus, our work tries to enhance data reuse in workflow-based systems and caches by coupling the advantages of these two data reuse concepts.

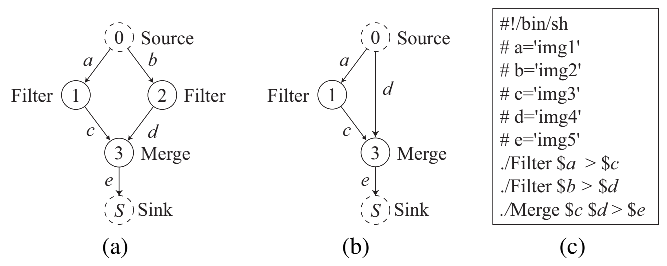
3 Preliminaries

To decribe our problem clearly, we first introduce the program-level reuse addressed by prior systems. Table 1 summarizes notations used in the paper.

Let $G = (V, E)$ be a directed acyclic graph (DAG), where V and E represent a set of vertices and that of edges in the graph, respectively. Graph G here has two special vertices, source and sink, each having only outgoing edges and incoming edges, respectively. Then, as shown in Fig. 1(a), prior systems [3, 4] regard a vertex and an edge as an application component and a flow dependency between components, respectively. Each vertex is labeled with a component name while each edge is labeled with a file name given to the next component (destination vertex). In the following, let (u, v) denote the edge connected from vertex u to vertex v.

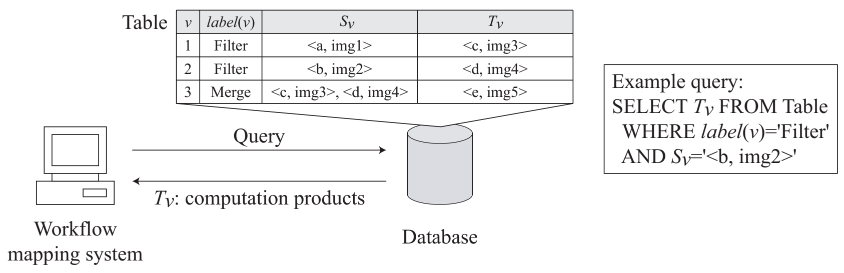
Note here that communities should have a single name space to give a unique name to identical data. Otherwise, data contents as well as fine names must be checked to prevent inappropriate reuse of wrong data. Thus, prior systems assume a single name space where identical components or files have a unique name. Let $label(v)$ and $label(u, v)$ denote the name labeled on vertex v and on edge (u, v), respectively. Let $value(u, v)$ also denote the values (contents) of the name $label(u, v)$.

Suppose that we have two DAGs $G = (V, E)$ and $G' = (V', E')$, where G and G' represent a workflow being considered for execution and that executed in the past, respectively. Suppose vertex $v \in V$ has a set I_v of incoming edges and that O_v of



```
#!/bin/sh
# a='img1'
# b='img2'
# c='img3'
# d='img4'
# e='img5'
./Filter $a > $c
./Filter $b > $d
./Merge $c $d > $e
```

Fig. 1. (a) A directed acyclic graph (DAG) representing a workflow, (b) its reduced DAG for data reuse, and (c) a shell script for the workflow. Vertex 2 and its edges $(0, 2)$ and $(2, 3)$ are replaced with $(0, 3)$, meaning that the second filter program is omitted by reuse of file 'img4.'



v	*label*(v)	S_V	T_V
1	Filter	<a, img1>	<c, img3>
2	Filter	<b, img2>	<d, img4>
3	Merge	<c, img3>, <d, img4>	<e, img5>

Example query:
SELECT T_V FROM Table
WHERE *label*(v)='Filter'
AND S_V='<b, img2>'

Fig. 2. Overview of data reuse in prior systems.

outgoing edges, where $I_v = \{(u, v) \mid \exists u$ such that $(u, v) \in E\}$ and $O_v = \{(v, w) \mid \exists w$ such that $(v, w) \in E\}$. Let $\mathcal{S}_v$ be the set of pairs of labels and values passed to vertex v: $\mathcal{S}_v = \{\langle label(u, v), value(u, v)\rangle \mid (u, v) \in I_v\}$. Let $\mathcal{T}_v$ also be the set of pairs of labels and values passed from vertex v: $\mathcal{T}_v = \{\langle label(v, w), value(v, w)\rangle \mid (v, w) \in O_v\}$. Then, any vertex v and its every edge $(u, w) \in I_v \cup O_v$ can be eliminated if and only if

C1. $\exists x \in V'$ such that $label(x) = label(v) \wedge \mathcal{S}_x = \mathcal{S}_v$.

Such vertices v and x represent a program with the same inputs, and thus they are identical computation. In this case, we already have computation products of x in the DB. Therefore, graph G can be reduced for data reuse as follows (see Fig. 1(b)). (1) Vertex elimination: Remove vertex v and its every incoming/outgoing edge $(u, w) \in I_v \cup O_v$ from set V and set E, respectively. (2) Vertex connection: For all w such that $(v, w) \in O_v$, add edge $(0, w)$ to E. Edge $(0, w)$ here has $label(v, w)$, representing loading of computation products from the DB.

According to condition C1, data reuse capabilities require a DB that stores a set of 4-tuples $(v, label(v), \mathcal{S}_v, \mathcal{T}_v)$, as shown in Fig. 2. Given such a DB, identical computations can be detected by comparing the two attributes $label(v)$ and $\mathcal{S}_v$ to obtain computation products $\mathcal{T}_v$ stored in the DB.

Prior systems typically reuse computation products in the following steps (Fig. 3).

S1. Workflow submission: The system receives a workflow, namely a DAG, from users.

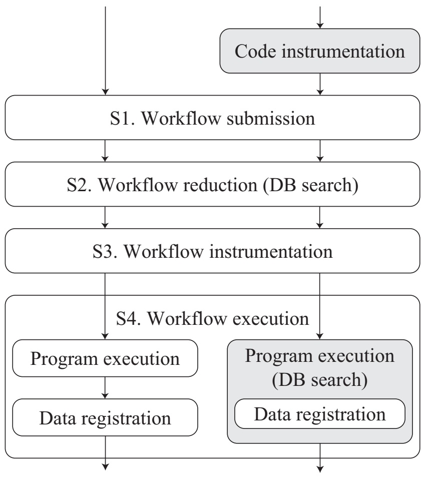


Fig. 3. Procedure for data reuse. Left-hand side shows the procedure for prior program-level reuse while right-hand side shows that for our block-level reuse.

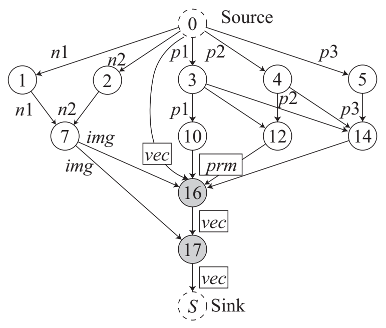
S2. Workflow reduction: The DAG is reduced to exploit data reuse based on the DB.
S3. Workflow instrumentation: Some instrumentation vertices are added to the reduced DAG for later registration of newly produced data.
S4. Workflow execution: This step consists of two substeps: Program execution step, which executes each application component using grid resources; Data registration step, which registers computation products to the DB after program execution.

4 Problem Description

In contrast to the program-level reuse, which prior systems support, our final goal is to realize the block-level reuse during program execution. To achieve this, we must tackle the following technical issues (see Fig. 3).

I1. Model extension: We must adapt the data reuse model mentioned in Section 3 to our case, because we focus on blocks rather than programs. The extended model must handle program structures and control-flow dependencies in programs.
I2. Code instrumentation: In addition to the workflow component, the source code in programs must be instrumented to record computation products for each block. The key issue here is to assist users in selecting the code to be instrumented for data reuse.
I3. Runtime registration: Since we focus on program blocks, their computation products must be registered during program execution. Therefore, reducing runtime overhead is a critical issue to obtain higher program performance. Note here that prior systems are allowed to register data after program execution.

With respect to issue I2, we assume that users know which code takes most of execution time, and thus such bottleneck code can be initially specified as the instrumented



(a)

```
 1: readln(n1); // File name 1
 2: readln(n2); // File name 2
 3: readln(p1); // Parameter 1
 4: readln(p2); // Parameter 2
 5: readln(p3); // Parameter 3
 6: h = 0; // Hierarchy
 7: img = load(n1, n2);
 8: while h<5 {
 9:       if h == 0 {
10:             prm = p1;
11:       } else if h == 1 {
12:             prm = p1 * p2;
13:       } else {
14:             prm = p1 * p2 * p3;
15:       }
16:       vec = registration(prm, img, vec);
17:       vec = resample(img, vec); // Increase resolution
18:       h += 1;
19: }
```

(b)

Fig. 4. Overview of (a) registration algorithm and (b) its DAG. Statements at lines 16–17 are initially specified as the target code for data reuse. Enclosed labels on edges represent flow-influenced variables.

code, which may save significant time and resources. Once such initial code is given by users, our method tries to suggest better (extended) code with smaller attribute values. Thus, we assist users in selecting appropriate attributes with smaller data size. In summary, the problem of data minimization can be defined as follows:

P1. The data minimization problem addressed in the paper is to minimize the data size of attribute values required for the target code initially specified by users.

Note here that the data minimization contributes to resolve issue I3, because attribute values are transmitted to the DB server as a part of queries. It also minimizes accesses to storage devices on the DB server. Thus, data minimization will reduce runtime overhead of data reuse.

5 Block-Level Data Reuse

We now present our data reuse framework based on an extension of the previous reuse model and the data minimization method.

5.1 Model Extension

To resolve issue I1, we extend the model for our case as follows (see Fig. 4).

– On program structures. In our interpretation, a vertex and an edge in a DAG represent a statement in a program and a flow dependency between statements, respectively. The label on a vertex and that on an edge represent the line number of the corresponding statement and the variable names relevant to the dependency, respectively.

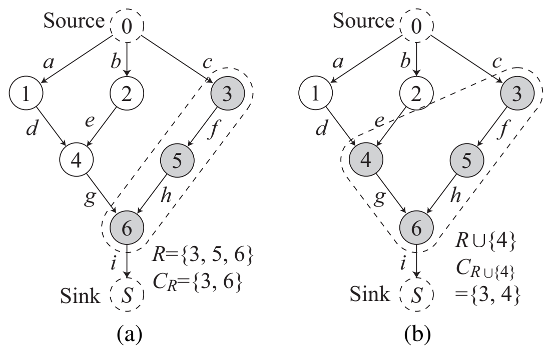


Fig. 5. Data minimization. Given a DAG with a set R of vertices, $\{3, 5, 6\}$, our method returns pairs of labels and values on edges $(0, 3)$ and $(4, 6)$ as the initial attributes. It then tries to replace some edges to others with smaller data size. In this case, (a) edge $(4, 6)$ is replaced with (b) edges $(1, 4)$ and $(2, 4)$ if the data size of attribute g is larger than total size of attributes d and e.

- On control-flow dependencies. Since programs consist of branches and iterations, which workflows in prior systems do not have, we associate labels with additional properties to represent flow-influenced variables. For such variables, we consider variables with loop-carried dependencies [11] or control-flow dependencies.

Figure 4 shows an example of the extended model. In this example, branch statements at lines 10, 12, and 14 depends on runtime conditions, and thus their outputs, namely the labels 'prm' on outgoing edges in the DAG, are marked as flow-influenced variables. Similarly, variable 'vec' with a loop-carried dependency is also marked as a flow-influenced variable.

5.2 Principles of Data Minimization

Suppose that we have a DAG $G = (V, E)$ with a set $R \subset V$ of vertices initially marked as the target code for data reuse (see Fig. 5). Then, the initial attributes $\mathcal{A}_R$ for R are given by their inputs:

$$\mathcal{A}_R = \bigcup_{v \in C_R} \mathcal{S}_v, \tag{1}$$

where C_R represents the set of critical vertices that have attribute information for R:

$$C_R = \{v \in R \mid \exists u \text{ such that } u \notin R \wedge (u, v) \in E\}. \tag{2}$$

Let $u \in V$ be a vertex such that $u \notin R$. Given R by users, the proposed method tries to extend R to include u to minimize the data size of inputs, namely attribute values. Such an extension is allowed if

C3. $v \in R$, for all v such that $(u, v) \in O_u$.

This extension tries to replace set O_u of edges outgoing from u with set I_u of edges incoming to u. If this extension is allowed, the initial attributes $\mathcal{A}_R$ can be replaced with those for $R \cup \{u\}$ given by:

$$\mathcal{A}_{R\cup\{u\}} = \bigcup_{v \in C_{R\cup\{u\}}} \mathcal{S}_v, \tag{3}$$

where

$$C_{R\cup\{u\}} = C_R \cup \{u\} - \{w \mid w \in C_R \wedge (u, w) \in O_u\}. \tag{4}$$

Condition C3 is not sufficient if vertex u corresponds to a flow-influenced statement, such as branches and loops. For example, suppose that we have if-else statements followed by the initial target code, as shown in Fig. 4(a). In this case, the extension mentioned above may include branches at lines 9–15 to the target code for data reuse. This implies that computation products vary depending on runtime conditions. In this case, we must record branching results as well as computation products. Otherwise, the replacement may result in wrong data reuse, because it does not have information on the actual flow that have yielded the products. To avoid such wrong reuse, we store actual flows by recording the value history list for flow-influenced variables.

In summary, data minimization is performed by extending R initially given by users. Extended part here is allowed to include flow-influenced statements, however, the actual flow must be recorded as attributes in the DB. We record this by the value history list of flow-influenced variables to perform data reuse only for identical computation.

5.3 Data Minimization Algorithm

Figure 6 shows our algorithm, which tries to extend the target code R to obtain the smallest attributes $\mathcal{A}$. Basically, it backtraces flow dependencies from a vertex in set R. The algorithm consists of the following three phases.

Phase 1. Extract the initial attributes $\mathcal{A}_R$. To do this, the algorithm computes a set C_R of critical vertices according to Eq. (2).

Phase 2. Extend the target code for data reuse. The algorithm then performs the code extension to compute set $\mathcal{L}$ containing possible sets of critical vertices. This code extension is done by recursive calls of function Extension(), as shown in Fig. 7. This function backtraces flow dependencies from a start vertex $u \in R$. During the backtracing of flows, candidates for critical vertices are added to set $\mathcal{L}$ at each visited vertex. The backtracing procedure stops when it reaches (1) the source or (2) a flow-influenced vertex. For the former case, it returns the current candidates $\mathcal{L}$. For the latter case, it stops further backtracing of flows because flow-influenced variables cannot be removed from attributes, as we mentioned in Section 5.2.

Phase 3. Select the smallest attributes $\mathcal{A}$. The attributes $\mathcal{A}$ are selected from set $\mathcal{L}$ of sets of critical vertices. This phase is executed at runtime if flow-influenced variables are included in $\mathcal{A}$, because the data size of such variables cannot be determined before program execution. That is, the value history list for such variables grows during program execution.

```
Algorithm DataMinimization($G$, $R$)
// Input #1. DAG $G = (V, E)$.
// Input #2. Set $R$ of vertices representing the initial target code.
// Output. Attributes $\mathcal{A}$ extended from the initial attributes $\mathcal{A}_R$.
begin
    // Phase 1: Extract the initial attributes $\mathcal{A}_R$
    $C_R \leftarrow \emptyset$; // initialize set $C$ as an empty set
    foreach vertex $v \in R$ do begin
        foreach edge $(u, v) \in I_v$ do begin
            if vertex $u \notin R$ then begin
                Add vertex $v$ to set $C_R$;
            end
        end
    end
    Compute the initial attributes $\mathcal{A}_R$ by using $C_R$ // see Eq. (1)
    $\mathcal{A} \leftarrow \mathcal{A}_R$;
    // Phase 2: Extend the target code by backtracing
    $\mathcal{L} \leftarrow \emptyset$; // $\mathcal{L} = \{C_R\}$: A set of sets of critical vertices
    Add $C_R$ to set $\mathcal{L}$;
    foreach vertex $v \in R$ do begin
        $\mathcal{L} \leftarrow$ Extension($v$, $C_R$, $\mathcal{L}$); // extend $R$ from vertex $v$
    end
    // Phase 3: Select the smallest attributes
    foreach set $C_{R'} \in \mathcal{L}$ do begin
        Compute the attributes $\mathcal{A}_{R'}$ by using $C_{R'}$ // see Eq. (1)
        if $\mathcal{A}_{R'}$ is smaller than $\mathcal{A}$ then begin
            $\mathcal{A} \leftarrow \mathcal{A}_{R'}$;
        end
    end
end
```

Fig. 6. Data minimization algorithm. See Fig. 7 for function Extension().

6 Experimental Results

We now show how the method reduces data size and execution time to evaluate the impact of block-level data reuse.

For experiments, we employ a cluster of 16 PCs, each having two Xeon 2.8 GHz CPUs and 2 GB of main memory. PCs are interconnected by a Myrinet switch [12], providing a bandwidth of 2 Gb/s. A DB system is constructed on a file server. This DB server is accessible from PCs through Gigabit Ethernet network.

The application used for experiments is nonrigid image registration [13], which computes point correspondences between two deformable objects. This application is written using the C++ language and the Message Passing Interface (MPI) standard [14]. It requires three parameters and a pair of three-dimensional images as inputs. The image size is $512 \times 512 \times 295$ voxels, which is equivalent to 148 MB per image in file size. From the viewpoint of program structures, this application have the following characteristics (see Fig. 4).

- Computation intensive. Sequential implementations take several hours to process the core functions, registration() and resample() at lines 16 and 17, respectively.
- Hierarchical algorithm. Registration is hierarchically performed in a coarse-to-fine manner to reduce the computational amount. Each hierarchy requires a parameter to control object deformations.
- Parametric study. While registration is controlled by three parameters ‘p1,’ ‘p2,’ and ‘p3,’ better parameter values are still unknown. Therefore, parametric study is

```
Function Extension($v$, $C_R$, $\mathcal{L}$)
// Input #1. Vertex $v$ representing the current point for backtracing.
// Input #2. Current set $C_R$ of critical vertices.
// Input #3. Current set $\mathcal{L}$ of sets of critical vertices.
// Output. Updated set $\mathcal{L}$.
begin
    $\mathcal{L}_{local} \leftarrow \mathcal{L}$;
    if $I_v \neq \emptyset$ then begin // $v$ has a predecessor
        foreach edge $(u, v) \in I_v$ do begin // try to include vertex $u$
            if $(u, v)$ is not a flow-influenced variable then begin
                $\mathcal{L}_{local} \leftarrow$ Extension($u$, $C_{R \cup \{u\}}$, $\mathcal{L}_{local}$);
                Add $C_{R \cup \{u\}}$ to $\mathcal{L}_{local}$; // see Eq. (4)
            end
        end
    end
    return $\mathcal{L}_{local}$;
end
```

Fig. 7. Code extension algorithm.

Table 2. Data size of attribute values required to reuse data at each of hierarchy H1–H5. Attribute 'img' is replaced with 'n1' and 'n2' while attribute 'vec' is replaced with 'n1,' 'n2,' and 'prm.'

Prior method w/o data minimization						Our method w/ data minimization					
Attribute	H1	H2	H3	H4	H5	Attribute	H1	H2	H3	H4	H5
img	680 M					n1	6				
vec	58 K	408 K	3 M			n2	6				
prm	8					prm	8	16	24	32	40
Total	680 M	680 M	683 M			Total	20	28	36	44	52

needed to know better parameter values, and thus, registration tasks are repeatedly submitted usually with the same pair of images but with slightly different combinations of parameters.

Due to the last characteristic, we think that data reuse is effective to accelerate parametric study of nonrigid registration algorithms on the grid.

We first determine the target code for data reuse. Two functions registration() and resample() are selected for data reuse, because these core functions take most (98%) of execution time. We then manually applied our method to the registration program.

Table 2 shows data reduction results. We can see that the prior method requires three attributes, namely all inputs directly given to the target function: 'prm,' 'img,' and 'vec.' Attributes 'prm' and 'img' are fixed-size variables containing 8 B and 680 MB of data, respectively. On the other hand, the data size of attribute 'vec' increases as the algorithm moves up the hierarchy, and thus it ranges from 58 KB to 3 MB. Summing up these sizes, the prior method requires approximately 680 MB and 683 MB of attribute values for the coarsest hierarchy H1 and for the finest hierarchy H5, respectively.

In contrast to the prior method, our method requires at most 52 B of attribute values. This reduction can be explained as follows.

– Attribute 'img' is replaced with 'n1' and 'n2,' because it has larger data size and satisfies condition C3. Vertex 7 in Fig. 4 intuitively explain this. In the example code, this replacement means that image data can be replaced with its file name.

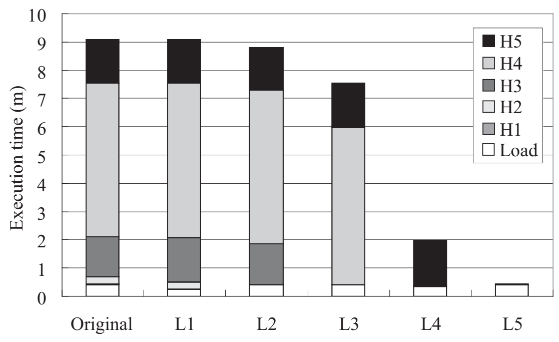


Fig. 8. Timing results. Each bar shows execution time and its breakdown for variation Li ($1 \leq i \leq 5$). Data reuse is applied to the program in a stepwise manner.

– Attribute 'vec' is replaced with 'n1,' 'n2,' and the value history list of 'prm.' Figure 4 indicates that 'img' and 'prm' must be included to attributes to remove 'vec,' because they are given as inputs to vertices 16 and 17, which output 'vec.' The former attribute 'img' here is already replaced with 'n1' and 'n2.' On the other hand, the latter attribute 'prm' is marked as a flow-influenced variable, and thus it must be recorded with its value history list. Since 'prm' is a double variable, its size is increased by 8 B at each hierarchy.

In order to activate data reuse in the application, we added function calls before and behind the target functions. The added function here requires attributes and their values to deal with 4-tuple data stored in the DB. We specified the smallest attributes in Table 2: 'n1,' 'n2,' and value history list of 'prm.'

Figure 8 shows timing results for the original program and its five variations L1–L5. Here, variation Li perform data reuse from hierarchy H1 to Hi, where $1 \leq i \leq 5$. In this figure, we can see that the original time of 9 minutes is reduced to approximately 30 seconds if all computations are allowed to be omitted by data reuse. Therefore, if all of computation products are already stored in the DB, other programs can use grid resources for approximately 8 minutes in this case.

Note here that hierarchy H3 takes most of execution time in the original program. Therefore, data reuse is effective if the computation products of this hierarchy is reused during program execution. In other words, data reuse at hierarchies H1, H2, and H3 was not so effective in terms of performance.

7 Conclusion

We have presented a data minimization method for efficient data reuse in grid environments. As compared with prior methods, our method focuses on reusing computation products at smaller granularities such as program blocks. The novelty of our method is the data dependence analysis that minimizes the data size of attribute values by extending the target code of data reuse. The proposed method allows us to transmit less amount of data between computing nodes and the DB server, and thus contributes to reduce runtime overhead of data reuse.

In experimental results, we find that the method reduces the data size from 683 MB to 52 B, achieving a small overhead for data reuse in a medical application. Our reuse framework also accelerates the application from approximately 9 minutes to 27 seconds, achieving shorter response time with higher efficiency.

Future work includes the development of a tool that automates the instrumentation of program code.

References

1. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: Enabling scalable virtual organizations. Int'l J. High Performance Computing Applications **15** (2001) 200–222
2. Nishikawa, T., Nagashima, U., Sekiguchi, S.: Design and implementation of intelligent scheduler for gaussian portal on quantum chemistry grid. In: Proc. 3rd Int'l Conf. Computational Science (ICCS'03), Part III. (2003) 244–253
3. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Blackburn, K., Lazzarini, A., Arbree, A., Cavanaugh, R., Koranda, S.: Mapping abstract complex workflows onto grid environments. J. Grid Computing **1** (2003) 25–39
4. Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A., Jacob, J.C., Katz, D.S.: Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Scientific Programming **13** (2005) 219–237
5. Zhao, Y., Wilde, M., Foster, I., Voeckler, J., Dobson, J., Gilbert, E., Jordan, T., Quigg, E.: Virtual data grid middleware services for data-intensive science. Concurrency and Computation: Practice and Experience **18** (2006) 595–608
6. Altintas, I., Birnbaum, A., Baldridge, K.K., Sudholt, W., Miller, M., Amoreira, C., Potier, Y., Ludaescher, B.: A framework for the design and reuse of grid workflows. In: Proc. 1st Int'l Workshop Scientific Applications of Grid Computing (SAG'04). (2004) 120–133
7. Casanova, H., Obertelli, G., Berman, F., Wolski, R.: The AppLeS parameter sweep template: User-level middleware for the Grid. In: Proc. High Performance Networking and Computing Conf. (SC2000). (2000)
8. Santos-Neto, E., Cirne, W., Brasileiro, F., Lima, A.: Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In: Proc. 10th Int'l Workshop Job Scheduling Strategies for Parallel Processing (JSSPP'04). (2004) 210–232
9. Strout, M.M., Carter, L., Ferrante, J., Freeman, J., Kreaseck, B.: Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In: Proc. 15th Workshop Languages and Compilers for Parallel Computing (LCPC'04). (2002) 90–110
10. Issenin, I., Brockmeyer, E., Miranda, M., Dutt, N.: Data reuse analysis technique for software-controlled memory hierarchies. In: Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE'04). (2004) 202–207
11. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Computing Surveys **26** (1994) 345–420
12. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.K.: Myrinet: A gigabit-per-second local area network. IEEE Micro **15** (1995) 29–36
13. Ino, F., Ooyama, K., Hagihara, K.: A data distributed parallel algorithm for nonrigid image registration. Parallel Computing **31** (2005) 19–43
14. Message Passing Interface Forum: MPI: A message-passing interface standard. Int'l J. Supercomputer Applications and High Performance Computing **8** (1994) 159–416