Advanced Data Flow Support for Scientific Grid Workflow Applications [*]

Jun Qin[†] and Thomas Fahringer
Institute of Computer Science, University of Innsbruck
Technikerstr. 21a, 6020 Innsbruck, Austria
{ jerry | tf }@dps.uibk.ac.at

ABSTRACT

Existing work does not provide a flexible dataset-oriented data flow mechanism to meet the complex requirements of scientific Grid workflow applications. In this paper we present a sophisticated approach to this problem by introducing a data collection concept and the corresponding collection distribution constructs, which are inspired by HPF, however applied to Grid workflow applications. Based on these constructs, more fine-grained data flows can be specified at an abstract workflow language level, such as mapping a portion of a dataset to an activity, independently distributing multiple datasets, not necessarily with the same number of data elements, onto loop iterations. Our approach reduces data duplication, optimizes data transfers as well as simplifies the effort to port workflow applications onto the Grid. We have extended AGWL with these concepts and implemented the corresponding runtime support in ASKALON. We apply our approach to some real world scientific workflow applications and report performance results.

Keywords

Grid Workflow, Data Flow, Data Collection, Data Distribution

1. INTRODUCTION

With the advent of Grid technologies, scientists and engineers are building more and more complex applications to manage and process large datasets and to execute scientific experiments on distributed Grid resources. Grid workflow systems play a paramount role in this process. It enables scientists to configure available application components into a workflow of tasks and submit them for execution on the Grid. Nowadays, many Grid workflow frameworks and tools have been developed for supporting scientific workflow applications.

A Grid workflow application can be seen as a collection of computational tasks that are processed in a well-defined order to accomplish a specific goal. Many control flow constructs have been identified and developed in Grid workflow systems to enable users to define the exact execution order of tasks. These constructs can be divided into four categories: sequential, parallel, conditional and iterative constructs. With each of these constructs, different data flows can be specified. Data flows in scientific Grid workflow applications are commonly complex because datasets are involved. For instance, a scientific application consumes a portion of a dataset produced by another application, a parallel iterative construct consumes multiple datasets and each of its loop iteration processes a variable number of elements in the datasets. However, how datasets and the corresponding data elements can be specified in data flow links, especially how datasets can be distributed onto the parallel loop iterations, is a problem not sufficiently addressed by most Grid workflow languages. Many Grid workflow systems solve the problem by replicating the entire dataset to activities or loop iterations, or by restricting file names used in datasets which reduces the workflow reusability. Fig. 1 illustrates the problem through an example of a *parallelFor* loop construct which accepts a dataset consisting of six data elements as input to its four loop iterations. Each loop iteration produces an output data element. The *parallelFor* construct has a subsequent task *A* which requires only the output data elements produced by iterations 0 and 2. Obviously, more flexible dataset-oriented data flow mechanisms are needed to avoid redundant data transfers.

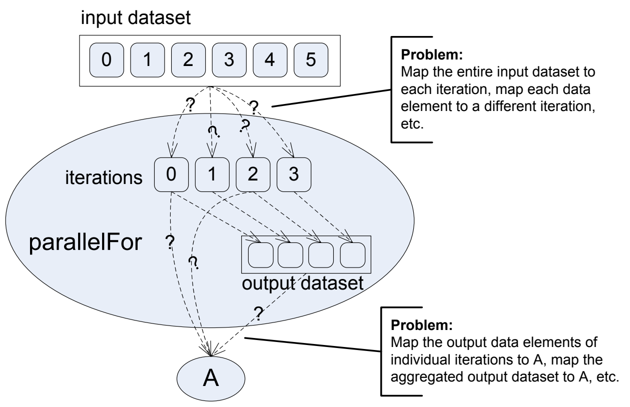

input dataset
0
1
2
3
4
5
Problem:
Map the entire input dataset to each iteration, map each data element to a different iteration, etc.
iterations
0
1
2
3
parallelFor
output dataset
Problem:
Map the output data elements of individual iterations to A, map the aggregated output dataset to A, etc.
A

Figure 1: Data flow problems

In this paper we present a sophisticated approach as part of the Abstract Grid Workflow Language (AGWL) [7] to solve the problem by introducing the concept of *data collection* and the corresponding collection distribution constructs, which are inspired by

[*]This work is partially funded by the European Union through the IST-034601 edutain@grid and FP6-004265 CoreGRID projects.
[†]Corresponding author

High Performance Fortran (HPF) [9]. The corresponding runtime support has been implemented in the ASKALON [6] Grid application development and computing environment. By using our approach, more fine-grained data flows can be specified, data transfers are optimized and thus performances are improved. The effort to port scientific applications onto Grid can also be simplified.

The remainder of this paper is organized as follows. Section 2 presents important related work. A brief overview of AGWL is provided in Section 3. Section 4 discusses our approach in detail. We apply the collection distribution constructs to some real world Grid workflow applications in Section 5 and report performance results in Section 6. The paper ends with a short conclusion and an outline of the future work.

2. RELATED WORK

Many workflow languages and systems are developed for supporting scientific Grid workflow applications. We limit this section to selected work and compare them against our solution.

The Chimera Virtual Data System (VDS) [8] aims to abstract the workflow from the details of implementation. Its workflows expressed in Chimera's Virtual Data Language (VDL) are converted into Condor's DAGMan [4] format for execution. While VDS supports iterations over datasets, it is limited to operating datasets (or slices of datasets) one element by one element, through the *foreach* statement in VDL. Taverna [13] is a data centric workflow development environment and has no explicit iterative constructs. Its workflow language SCUFL (Simple Conceptual Unified Flow Language) provides support for processing data collections through an implicit iteration mechanism, which is limited to the cross or dot product of lists (collections) being processed. The Kepler project [2] is based on an actor-oriented modeling paradigm where actors correspond to re-usable workflow components. The workflow specification is based on the workflow modeling language MoML (Modeling Markup Language). Kepler uses a *map* operator to apply a function, which operates on singletons, to collections. It is limited to processing one collection at a time. More recently published work [11] from the Kepler project focuses on nested, especially heterogeneous data collections. It uses *read scope* to specify portions of collections and *iteration scope* to control iterations of actions. Its collection operations are limited to invoking actions once for each item, or once for the entire collection. Triana [20] is an integrated and generic workflow-based graphical problem solving environment. Triana uses an XML based language similar to Web Services Description Language (WSDL). The Triana workflow language has no explicit support for control flow constructs. Loops and execution branching are handled by specific components. The Karajan [21] workflow language supports parallel iterations. ICENI [10] (Imperial College e-Science Network Infrastructure) is a system for workflow specification and enactment on the Grid. However, no flexible dataset-oriented data flow is supported in either of these systems. Parallel Computing Patterns [14] identified a data parallelism pattern and its variants *static/dynamic/adaptive* data parallelism for Grid workflows. These patterns have a close relationship to the classical *multi-instance* workflow pattern [16] and its variants. However, none of these patterns reflect *BLOCK*, *BLOCK(S)*, *BLOCK(S,L)* and *REPLICA(S)* collection distribution constructs as presented in this paper.

In a word, existing work commonly suffers by one or several of the following drawbacks: no iterative constructs, no support for processing datasets or portions of datasets, and no flexible dataset-oriented data flow support for iterative constructs. In contrast, AGWL supports a rich set of control flow constructs, including iterative constructs, as well as mapping a portion of a *data collection* to activities or loop iterations. Furthermore, one iterative construct in AGWL (e.g. the `parallelFor` or `parallelForEach` construct, etc.) can also process multiple *data collections* and each *data collection* can be processed in an independent way in terms of how many data elements are processed in one loop iteration. All of these features can be specified at the workflow language level by domain users to control how datasets are processed by activities or iterative constructs.

3. OVERVIEW OF AGWL

AGWL [7] is an XML-based language for describing Grid workflow applications at a high level of abstraction. It is the main interface to the ASKALON [6] Grid application development and computing environment and has been applied to numerous real world Grid workflow applications. AGWL has been designed such that users can concentrate on specifying Grid workflow applications without dealing with either the complexity of the Grid or any specific implementation technology (e.g. Web or Grid Service, software components or Java classes, etc.). AGWL workflows consist of activities, control flow constructs, data flow links and properties and constraints.

An AGWL activity can be an *atomic activity* or a *compound activity*. An atomic activity is represented by an *activity type* and input and output data ports. The input and output data ports are logical representations of the corresponding input and output data. The number and the types of the input and output data ports are determined by the activity type. An activity type is an abstract description of a group of *activity deployments* (concrete implementations of computational entities deployed in the Grid) which have the same functionality, probably different performance behaviors. Activity types shield the implementation details of activity deployments from the AGWL programmer. Locating and invoking activity deployments based on activity types are done by the underlying runtime system. An invocation of an activity deployment is called an *activity instance*. AGWL activities are connected by control flow constructs and data flow links.

AGWL allows a programmer to define a graph of activities that refer to computational tasks or user interactions. A rich set of control flow constructs (compound activities) has been provided in AGWL to simplify the specification of Grid workflow applications such as `sequence`, `parallel`, `if`, `switch`, `while`, `doWhile`, `for`, `forEach`, `parallelFor` and `parallelForEach` with semantics similar to comparable constructs in high-level programming languages. The `dag` construct is also provided for describing a directed acyclic graph of activities. AGWL also supports sub-workflows, which are similar to procedures in high-level programming languages. Sub-workflows are used to modularize, encapsulate and reuse code regions. In the remaining text, activities refer to both atomic activities and compound activities if not explicitly stated.

The data flow in AGWL is expressed by data flow links from source data ports to sink data ports. A source data port can be an input data port of the whole workflow, an input data port of a compound activity (e.g. a `parallelFor` construct), or an output data port of an atomic activity. A sink data port can be an output data port of the whole workflow, an output data port of a compound activity, or an input data port of an atomic activity.

Properties and constraints can be defined in AGWL to provide additional information for workflow runtime systems to optimize and steer the executions of workflow applications. Properties provide hints about the behavior of activities and constraints should be complied with by the underlying workflow runtime system. The user can specify properties and constraints for both activities and

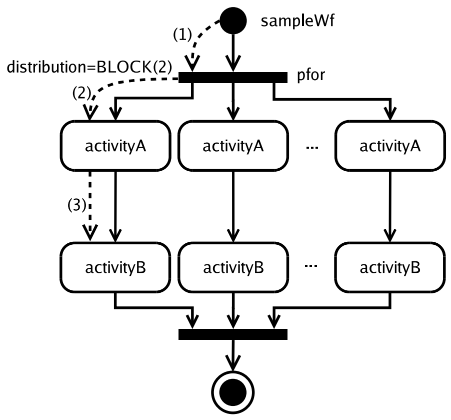


Figure 2: Three AGWL data flow links

data ports.

To enable readers to understand the collection distribution constructs presented in this paper, the AGWL data flow model is further explained in the following paragraph (refer to [7] for more detailed information on AGWL).

Data Flow Model Unlike many other workflow languages, e.g. the SCUFL used in Taverna [13], the workflow language of Triana [20], the WSBPEL [12] from OASIS, etc., AGWL does not have an explicit XML tag like `<link>` or `<connection>`. Instead, data flow links are specified by setting the `source` attributes of sink data ports to source data ports, which is easier to use but provides similar expressive power. The `source` attribute of a sink data port can be specified in the form of `activity-name/source-data-port-name`, where the `activity-name` can be the name of the workflow, the name of a compound activity or the name of an atomic activity. Ex. 1 illustrates three data flow links and one constraint (the corresponding graphical representation is illustrated in Fig. 2 with the dashed lines showing the three data flow links and the text beside the second data flow link showing the constraint): the data flow link (1) flows from the input data port *inWf* (line 3), the logical representation of the files *gsiftp://host//dir/file1*, ..., *gsiftp://host//dir/filen*, of the workflow *sampleWf* to the input data port *inPfCol* (line 10) of the `parallelFor` compound activity *pfor*. The `source` attribute of the sink data port *inPfCol* is set to *sampleWf/inWf* (line 11). In the data flow link (2), the source data port *inPfCol* with type of `agwl:collection` has a constraint *agwl:distribution=BLOCK(2)* (line 13-14), which specifies that the data elements of the collection (i.e. the files) are mapped pairwise to the corresponding parallel loop iterations and consumed by the activity *activityA* through its input data port *inAct* (line 23). The constraint *agwl:distribution* and its value *BLOCK(2)* as well as the data type `agwl:collection` are explained in Section 4. The data flow link (3) is similar to (1). The data flow links from/to sub-workflows, which are similar to the ones from/to a compound activity like a `parallelFor`, are omitted here.

Example 1. Three AGWL data flow links

```
1  <agwl name="sampleWf">
2    <workflowInput>
3      <dataIn name="inWf" type="agwl:collection"
4              source="gsiftp://host//dir/file1,...,
5                      gsiftp://host//dir/filen" />
6    </workflowInput>
7    <workflowBody>
8      <parallelFor name="pfor">
9        <dataIns>
10         <dataIn name="inPfCol" type="agwl:collection"
11                 source="sampleWf/inWf">
12           <constraints>
13             <constraint name="agwl:distribution"
14                         value="BLOCK(2)" />
15           </constraints>
16         </dataIn>
17       </dataIns>
18       <loopCounter name="index" type="xs:integer"
19                    from="1" to="10" step="1"/>
20       <loopBody>
21         <activity name="activityA" type="...">
22           <dataIns>
23             <dataIn name="inAct" type="agwl:collection"
24                     source="pfor/inPfCol" />
25           </dataIns>
26           <dataOuts>
27             <dataOut name="outAct" type="agwl:file" />
28           </dataOuts>
29         </activity>
30         <activity name="activityB" type="...">
31           <dataIns>
32             <dataIn name="inAct" type="agwl:file"
33                     source="activityA/outAct"/>
34           </dataIns>
35           <dataOuts>
36             <dataOut name="outAct" type="agwl:file" />
37           </dataOuts>
38         </activity>
39       </loopBody>
40       <dataOuts .../>
41     </parallelFor>
42   </workflowBody>
43   <workflowOutput .../>
44 </agwl>
```

4. DATA COLLECTIONS AND COLLECTION DISTRIBUTION CONSTRUCTS

Scientific workflows usually involve large and complex dataset processing. For instance, a scientific application consumes a portion of a dataset produced by another application, a parallel iterative construct produces datasets based on their input, or a parallel iterative construct consumes multiple datasets and each of its loop iteration processes a variable number of data elements of different datasets. In this section, we describe AGWL *data collections* and explain how *data collections* can be mapped to activities or distributed onto loop iterations by using collection distribution constructs.

4.1 Data collections

Datasets in scientific workflows may contain static or dynamic (unknown at composition time) number of data elements. Modeling at the Grid workflow language level each element in datasets with a logical data port can be awkward and often impossible. To solve this problem, AGWL introduces the concept of *data collection* (Fig. 3) to model datasets in scientific Grid workflows at a high level of abstraction.

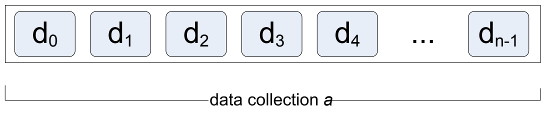


Figure 3: A data collection *a* with n data elements

AGWL *data collection* is a logical data representation of physical data. It is defined as a data type `agwl:collection` in AGWL, where `agwl` is the namespace used in XML representations of AGWL workflows. A data port with type of `agwl:collection` represents an ordered list of data elements provided by domain users as the initial input of a workflow or produced by workflow activities as an intermediate result. The number of the data elements contained in a data collection may be dynamic. The data elements in a data collection are logical representations of physical data, which can be files in a file system, data retrieved from a relational database, or primitive data such as *integers* or *strings* in the memory of the underlying workflow runtime system. Files will be used in the following sections to demonstrate our approach. Data elements can be accessed with their indices in the enclosing data collection.

To port scientific applications, especially the ones producing dynamic output datasets, onto the Grid, one of the common approaches is to write some wrapping code such as to *tar* a set of output files into a *tar* file. This kind of wrapping code is not flexible in the case where only a portion of the dataset is required. The reusability of the wrapping code is also limited. In contrast, the AGWL data collection presented here provides a more flexible solution and avoids this kind of wrapping code. Thus our approach is a valuable contribution to simplify porting workflow applications onto the Grid.

4.2 Collection distribution constructs

AGWL provides two built-in constraints for specifying collection distribution constructs: *agwl:element-index* and *agwl:distribution* (The namespace *agwl* is omitted in the following text to avoid redundancy). The constraint *element-index* is used to specify portions of data collections and it can be used for data ports of activities. The constraint *distribution* is used to partition data collections into blocks which are then distributed onto loop iterations. While the constraint *distribution* can be used for both sequential and parallel iterative constructs, we only focus on the parallel iterative constructs in the remainder of this paper.

The value of the constraint *element-index* is a list of coma separated colon expressions. The syntax is defined by the following grammar, where e denotes the element index, c a colon expression, s_1 a start index, s_2 a stop index, s_3 a stride:

$$\begin{aligned} e &::= c[, c]* \\ c &::= s_1[: s_2[: s_3]\,] \end{aligned}$$

For example, the constraint *element-index=1,3,6:10:2* specifies the data elements associated with index 1,3,6,8,10 in the source data collection. Note that in the absence of the constraint *element-index*, the entire data collection is specified.

To distribute data collections onto parallel loop iterations, we reused ideas from High Performance Fortran (HPF) [9], where some directives are used to map a data array into a processors array. In AGWL, a data collection is an ordered list of data elements, i.e. a one-dimensional array. Parallel loop iterations can also be considered as a one-dimensional array, which we denote by *iteration array*. Thus the problem can be formulated as how to map a one-dimensional array of data elements (a data collection) to another one-dimensional array of iterations (an iteration array). AGWL supports the following collection distribution constructs: *BLOCK*, *BLOCK(S)*, *BLOCK(S, L)* and *REPLICA(S)*, which are specified through the constraint *distribution* of input data ports of parallel iterative constructs (see Ex. 1). These four collection distribution constructs are processed at runtime to determine which elements of a data collection are distributed onto which iteration. We explain these four collection distribution constructs in detail in the following sections assuming that all data elements in data collections are distributed onto at least one iteration. It is possible that some iterations may not be assigned to any data element. In the case where not all data elements in a data collection are required to be distributed onto parallel loop iterations, a subset of the data collection can be obtained through the constraint *element-index*. The constraint *element-index* has higher priority than the constraint *distribution* when both of them are specified for the same data port.

In order to express collection distributions, we assume that any collection C with $|C|$ data elements is associated with an *index domain* $\boldsymbol{J}^C$ which is defined by a set of integers $\{i|0 \leq i < |C|\}$. The index domain $\boldsymbol{J}^C$ provides an unambiguous *name* for the data elements in the data collection. Let $\boldsymbol{J}^C$ denote an index domain of a data collection C, $\boldsymbol{K}^I$ an index domain of an iteration array I, the collection distribution problem can be further formulated as how to map $\boldsymbol{J}^C$ to $\boldsymbol{K}^I$. The notations used in the explanation of the four collection distribution constructs are summarized in Table 1.

C	a data collection
$\lvert C\rvert$	the element number of C
$\boldsymbol{J}^C$	the index domain of C
I	an iteration array
$\lvert I\rvert$	the iteration number of I
$\boldsymbol{K}^I$	the index domain of I
i	an index
$[i_1 : i_2]$	a set of indices, defined by $[i_1 : i_2] := \{i \mid i_1 \leq i \leq i_2\}$
$\delta(i)$	a function mapping indices of C to indices of I

Table 1: Notations

4.2.1 *BLOCK distribution*

BLOCK distribution partitions a data collection C into equal sized, contiguous blocks and distributes each of them onto a different iteration of an iteration array I. The size of each iteration's block is determined by the element number $|C|$ and the iteration number $|I|$.

Definition 1. ***BLOCK distribution*** *of a data collection C is a function δ: $\boldsymbol{J}^C \rightarrow \boldsymbol{K}^I$ that partitions the data collection C into $b = \left\lfloor \frac{|C|}{\left\lceil \frac{|C|}{|I|} \right\rceil} \right\rfloor$ contiguous blocks of size $s = \left\lceil \frac{|C|}{|I|} \right\rceil$ which are distributed onto the first b iterations, with $0 \leq b < |I|$, and, if $|C| \bmod s \neq 0$, one additional block with $|C| \bmod s$ elements which are distributed onto the last iteration with index b. The function is given by:*

$$\delta(i) = \left\{ \left\lfloor \frac{i}{\left\lceil \frac{|C|}{|I|} \right\rceil} \right\rfloor \,\middle|\, 0 \leq i < |C| \right\}$$

Fig. 4 illustrates the distribution of a data collection with $|C| = 12$ data elements onto $|I| = 4$ loop iterations based on the collection distribution construct *BLOCK*, where each iteration is assigned to a block of three data elements.

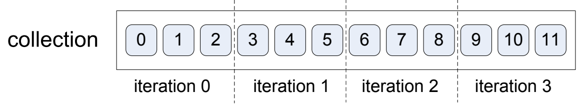


Figure 4: The *BLOCK* distribution

4.2.2 *BLOCK(S) distribution*

While *BLOCK* distribution partitions a data collection into equal sized blocks (except that the last one may have a smaller size) based on the iteration number, it is more common in scientific applications to partition a data collection into fixed sized blocks. For example, three files are produced in each time step of a simulation process, and the three files of one time step are required by the subsequent computation which is enclosed in the loop body of a parallel loop construct. *BLOCK(S)* is provided for this purpose, with the integer parameter S specifying the fixed block size.

Definition 2. ***BLOCK(S) distribution*** *of a data collection C is a function δ: $\boldsymbol{J}^C \rightarrow \boldsymbol{K}^I$ that partitions the data collection C into $b = \left\lfloor \frac{|C|}{S} \right\rfloor$ contiguous blocks of size S which are distributed onto the first b iterations, with $0 \leq b < |I|$, and, if $|C| \bmod S \neq 0$, one additional block with $|C| \bmod S$ elements which are distributed onto the next iteration with index b. We require $S \geq \left\lceil \frac{|C|}{|I|} \right\rceil$ to ensure that all data elements in the collection C are distributed onto at least one iteration. All other iterations (if any) are not assigned to any data elements. The function is given by:*

$$\delta(i) = \left\{ \left\lfloor \frac{i}{S} \right\rfloor \mid 0 \leq i < |C| \wedge S \geq \left\lceil \frac{|C|}{|I|} \right\rceil \right\}$$

Fig. 5 illustrates the distribution of a data collection with $|C| = 12$ elements onto $|I| = 3$ iterations based on the collection distribution construct *BLOCK(5)*, where iteration 0 is assigned to a block of five data elements with index $[0 : 4]$, iteration 1 a block of five data elements with index $[5 : 9]$, and the last iteration a block of two data elements with index $[10 : 11]$, respectively.

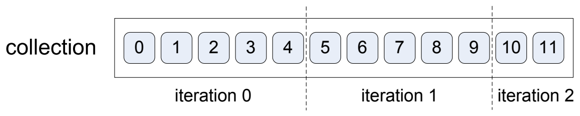


Figure 5: The *BLOCK(5)* distribution

4.2.3 BLOCK(S,L) distribution

BLOCK distribution and *BLOCK(S)* distribution partition a data collection into equal sized, contiguous blocks (except that the last one may have a smaller size) which does not involve replication. For some applications, an overlap between neighboring blocks is required. For example, an application which processes the files of a certain time step may also need the files from the previous time step. For this purpose, *BLOCK(S,L)* is provided with the integer parameter S specifying the block size, and the integer parameter L specifying the size of the overlapped part between neighboring blocks.

Definition 3. ***BLOCK(S,L) distribution*** *of a data collection C is a function δ: $\boldsymbol{J}^C \rightarrow \boldsymbol{K}^I$ that partitions the data collection C into $b = \left\lfloor \frac{|C|-L}{S-L} \right\rfloor$ overlapped blocks of size S (with an overlap of size L between each two neighboring blocks) which are distributed onto the first b iterations, with $0 \leq b < |I|$, $L < S$, and, if $(|C| - L) \bmod (S - L) \neq 0$, one additional block with $|C| - b \times (S - L)$ elements are distributed onto the next iteration with index b. We require $\left\lceil \frac{|C|-L}{S-L} \right\rceil \leq |I|$ to ensure that all data elements in the collection C are distributed onto at least one iteration. All other iterations (if any) are not assigned to any data elements. The function returning a set of iteration indices is given by:*

$$\begin{aligned} \delta(i) \quad =& \Big\{ [\delta(i)_{min} : \delta(i)_{max}] \Big| \; 0 \leq i < |C| \\ & \wedge \; L < S \; \wedge \; \left\lceil \frac{|C| - L}{S - L} \right\rceil \leq |I| \Big\} \\ where & \end{aligned}$$

$$\begin{aligned} \delta(i)_{min} =& \Big\{ \max\left(0, \left\lfloor \frac{i - L}{S - L} \right\rfloor\right) \Big| \; 0 \leq i < |C| \\ & \wedge \; L < S \; \wedge \; \left\lceil \frac{|C| - L}{S - L} \right\rceil \leq |I| \Big\} \\ \delta(i)_{max} =& \Big\{ \min\left(\left\lfloor \frac{i}{S - L} \right\rfloor, |I| - 1\right) \Big| \; 0 \leq i < |C| \\ & \wedge \; L < S \; \wedge \; \left\lceil \frac{|C| - L}{S - L} \right\rceil \leq |I| \Big\} \end{aligned}$$

Fig. 6 illustrates the distribution of a data collection with $|C| = 12$ elements onto $|I| = 3$ iterations based on the collection distribution construct *BLOCK(6,3)*, where iteration 0 is assigned to a block of six data elements with index $[0 : 5]$, iteration 1 a block of six data elements with index $[3 : 8]$, and iteration 2 a block of six data elements with index $[6 : 11]$, respectively. The data elements with index 3,4,5 are distributed onto both iteration 0 and iteration 1 and the data elements with index 6,7,8 are distributed onto both iteration 1 and iteration 2.

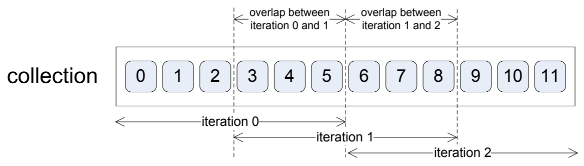

overlap between iteration 0 and 1
overlap between iteration 1 and 2
collection
0
1
2
3
4
5
6
7
8
9
10
11
iteration 0
iteration 1
iteration 2

Figure 6: The *BLOCK(6,3)* distribution

4.2.4 REPLICA(S) distribution

BLOCK distribution, *BLOCK(S)* distribution and *BLOCK(S,L)* distribution are normally used when $|C| \geq |I|$. However, in some cases, a smaller number of data elements are needed to be replicated onto a larger number of loop iterations. To support this kind of data distribution, *REPLICA(S)* distribution (we consider it as a special distribution) is provided with the integer parameter S specifying onto how many iterations each data element in a data collection should be replicated.

Definition 4. ***REPLICA(S) distribution*** *of a data collection C is a function δ: $\boldsymbol{J}^C \rightarrow \boldsymbol{K}^I$ that replicates each data element in the data collection C, S times which are distributed onto the first $b = S \times |C|$ iterations, with $0 \leq b < |I|$. We require $S \leq \left\lfloor \frac{|I|}{|C|} \right\rfloor$ to ensure that all replicated data elements are distributed onto at least one iteration. All other iterations (if any) are not assigned to any data elements. The function returning a set of iteration indices is given by:*

$$\delta(i) = \left\{ [S \times i : S \times (i + 1) - 1] \mid 0 \leq i < |C| \wedge S \leq \left\lfloor \frac{|I|}{|C|} \right\rfloor \right\}$$

Fig. 7 illustrates the distribution of a data collection with $|C| = 3$ elements onto $|I| = 12$ iterations based on the collection distribution construct *REPLICA(4)*, where iterations with index $[0 : 3]$ are assigned to the data element with index 0, iterations with index $[4 : 7]$ the data element with index 1, iterations with index $[8 : 11]$ the data element with index 2, respectively.

Note that both the constraint *element-index* and the constraint *distribution* can be specified for an input data port of a parallel iterative construct, and a parallel iterative construct can have as many input data ports as needed. Thus AGWL supports processing of multiple collections with one parallel iterative construct and each

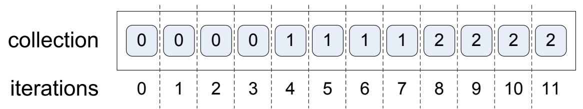


Figure 7: The *REPLICA(4)* distribution

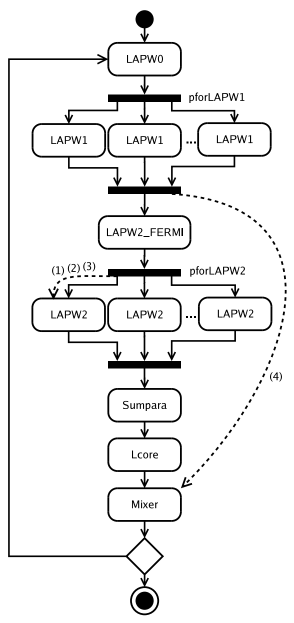


Figure 8: The WIEN2k workflow

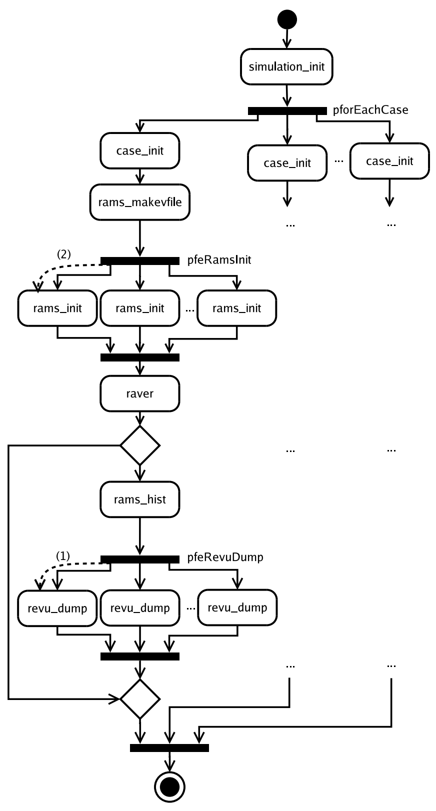


Figure 9: The MeteoAG workflow

collection, which may have different number of data elements, can be processed independently based on the associated collection distribution constructs.

5. CASE STUDY

In this section, we apply the collection distribution constructs presented in Section 4 to four real world scientific Grid workflow applications: (1) the material science application WIEN2k [3], (2) the meteorology application MeteoAG [18], (3) the astrophysics application AstroGrid [17], and (4) the astrophysics application GRASIL [19].

5.1 WIEN2k

WIEN2k [3] is a program package for performing electronic structure calculations of solids using density functional theory. The programs which compose the WIEN2k package are typically organized in the workflow illustrated in Fig. 8. The activity *LAPW1* and the activity *LAPW2* can be executed in parallel, specified by parallel iterative constructs. The number of parallel loop iterations of each parallel iterative construct is determined by *kpoint*, which is the output of the activity *LAPW0*. We only focus on the data ports with type of `agwl:collection` in the workflow here. A detailed explanation of the whole workflow can be found in [7]. In this workflow, the `parallelFor` construct *pforLAPW1* produces three data collections: *energyFileCol*, *vectorFileCol*, and *scf1FileCol*. The atomic activity *LAPW2_FERMI* consumes the data collection *energyFileCol* and produces another data collection *weighFileCol*. The four data flow links (the labeled dashed lines in Fig. 8) with their associated collection distribution constructs specify how these data collections are consumed by subsequent activities. The data flow links (1), (2) and (3) each has a constraint *distribution=BLOCK(1)* in its source data port. These constraints specify that for each loop iterationof the `parallelFor` construct *pforLAPW2*, only one data element of the corresponding data collection (*energyFileCol*, *vectorFileCol* or *weighFileCol*) is required. This example shows how a `parallelFor` construct can process multiple data collections. Note that, in this case, it is not required that all data collections must have the same number of data elements. The iteration number of the `parallelFor` construct *pforLAPW2* is decided neither by the iteration number of the `parallelFor` construct *pforLAPW1* nor by the element number of the data collection *weighFileCol*. Instead, the `parallelFor` construct *pforLAPW2* has its own loop counter for this purpose. We believe this approach is more flexible than those described in related work. The data flow link (4) has a constraint *element-index=0* in its sink data port because the activity *Mixer* only requires the first data element in the data collection *scf1FileCol*.

Note that without the constraint *distribution*, all data elements in the collections *energyFileCol*, *vectorFileCol* and *weighFileCol* would have to be transferred at runtime to the Grid site where each activity *LAPW2* will be executed, which would result in redundant data transfers. The same holds for the data collection *scf1Collection*. The AGWL representation of the WIEN2k workflow and the performance improvement with these collection distribution constructs are discussed in Section 6.1.

5.2 MeteoAG

MeteoAG [18] is a meteorology simulation application based on the numerical model RAMS [5]. The simulations produce precipitation fields of heavy precipitation cases over the western part of Austria at a spatial and temporal grid in order to resolve most

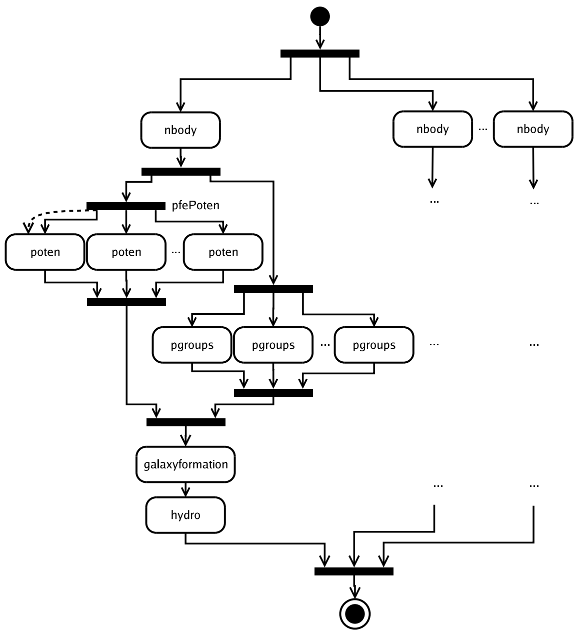


Figure 10: The AstroGrid workflow

alpine watersheds and thunderstorms. Fig. 9 illustrates the workflow structure with two labeled dashed lines showing the interesting data flow links. Considering the data flow link (1) in this workflow, the activity *rams_hist* produces in each time step two *grid* files (based on the input parameter *NGRID* which is specified in the input file, NGRID=2 in this case) and one *head* file, which we denote by two data collections: *gridFiles*, *headFiles*. For each run of an iteration of the `parallelForEach` construct *pfeRevuDump*, the activity *revu_dump* requires the files of one time step (two *grid* files and a *head* file) produced by the activity *rams_hist*. The collection distribution constructs *BLOCK(2)* and *BLOCK(1)* are used here to fulfill the data flow requirements.

The data flow link (2) in this workflow is a good example to demonstrate *BLOCK(S,L)* distribution. Similar as the activity *rams-_hist*, the activity *rams_makevfile* produces two *grid* files and one *tag* file in each time step. Again, we denote them by two data collections: *gridFiles*, *tagFiles*. For each run of an iteration of the `parallelForEach` construct *pfeRamsInit*, the activity *rams_init* requires not only the files of the current time step but also those of the previous time step. Therefore, the collection distribution constructs *BLOCK(4,2)* and *BLOCK(2,1)* can be used respectively for the distribution of the two data collections *gridFiles* and *tagFiles* onto the parallel loop iterations.

Both examples in the workflow MeteoAG also show that how multiple data collections can be processed by one parallel iterative construct independently in terms of how many data elements of each data collection are processed in one loop iteration. The performance results with these collection distribution constructs are discussed in Section 6.2.

5.3 AstroGrid

The AstroGrid [17] application is about numerical simulations of the movements and interactions of galaxy clusters based on N-Body systems. The computation starts with the state of the universe at some time in the past and is done to the current time. Galaxy potentials are computed for each time step. Then the hydrodynamic behavior and processes are calculated. The workflow structure is illustrated in Fig. 10. Considering the data flow link illustrated with the dashed line, the activity *nbody* produces two data collections: *t00Files*, consisting of multiple *t00* files, and *dataFiles*, consisting of multiple *data* files. One *t00* file is produced at each time step. One *data* file is produced when every four *t00* files are available. For each run of an iteration of the `parallelForEach` construct *pfePoten*, the activity *poten* requires one *t00* file and the corresponding *data* file. While the data collection *t00Files* is iterated over by the `parallelForEach` construct *pfePoten*, the data collection *dataFiles* is distributed based on the collection distribution construct *REPLICA(4)*, which specifies that every *data* file should be replicated four times then distributed onto the corresponding loop iterations.

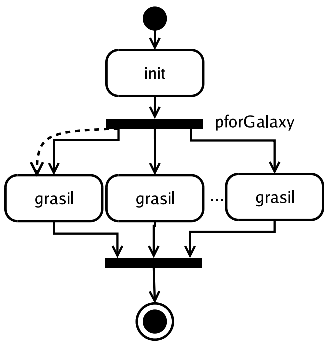


Figure 11: The GRASIL workflow

5.4 GRASIL

GRASIL [19] is an application to calculate the spectral energy distribution (SED) of galaxies lying in a certain field of view (light cone) ranging from now back to shortly after the beginning of the universe. The workflow structure is very simple and is illustrated in Fig. 11. The activity *init* produces a data collection, in which each data element corresponds to the data about a specific galaxy. The data collection is then processed by a `parallelFor` construct *pforGalaxy*, which uses the constraint *distribution=BLOCK* to distribute the input collection onto its loop iterations. The activity *grasil* is designed to accept a data collection with a variable number of galaxies and calculates each of them, which make the workflow immune to the changes of the element number of the data collection produced by the activity *init*. Thus the workflow reusability is improved.

6. EXPERIMENTAL RESULTS

We have implemented all collection distribution constructs described in this paper and integrated them in the ASKALON Grid environment, which is the main Grid application development and computing environment for the Austrian Grid infrastructure [1]. Through our UML based Grid workflow modeling tool [15], the domain users can develop AGWL based Grid workflow applications, including the selection of correct collection distribution constructs. We have conducted the experiments of the WIEN2k workflow and the MeteoAG workflow. The sizes of the files in the data collections used in our experiments range from several kilobytes to several megabytes. In both experiments, we measured the number of file transfers and the execution time and compared them for two cases: (1) without collection distribution constructs specified (denoted by *without data collection distribution*), and (2) with collection distribution constructs specified (denoted by *with data collection distribution*). The corresponding results are presented in the

Grid Site	*CPU*	*#*	*GHz*	*JobMgr*	*Location*
karwendel	Dual Core AMD Opteron	8	2.4	SGE	Innsbruck
c703-pc2201	Pentium 4	8	2.8	Torque	Innsbruck
c703-pc2509	Pentium 4	8	2.8	Torque	Innsbruck
schafberg	Itanium 2	8	1.4	PBS	Salzburg
altix1	Itanium 2	8	1.4	PBS	Innsbruck
c703-pc450	Pentium 4	8	1.8	Torque	Innsbruck
hydra	AMD Athlon	8	1.67	Torque	Linz

Table 2: The Austrian Grid testbed

following sections. A subset of the computational resources which have been used for the experiments is summarized in Table 2.

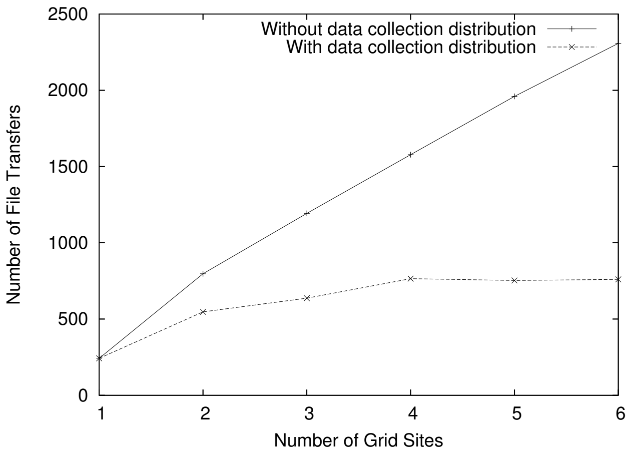
6.1 Results of the WIEN2k workflow

The WIEN2k workflow structure is illustrated in Fig. 8 in Section 5.1, and the corresponding AGWL representation is depicted in Ex. 2 (for simplicity, details not related to collections are omitted). Since the integer data port *kpoint* produced by the activity *LAPW0* determines the iteration number of the `parallelFor` constructs *pforLAPW1* and *pforLAPW2* (line 16 and line 69), which further determines the size of the data collections mentioned in Section 5.1, we performed two series of experiments for the WIEN2k application, corresponding to two different problem sizes: $kpoint = 116$ and $kpoint = 252$. The experiments were conducted on six Grid sites: *karwendel*, *c703-2201*, *c703-2509*, *schafberg*, *c703-pc450* and *hydra*. For each problem size, we first executed the workflow on the fastest Grid site *karwendel*. Then, we incrementally added new Grid sites to investigate whether we can improve the performance of the workflow application by increasing the available computational Grid resources.

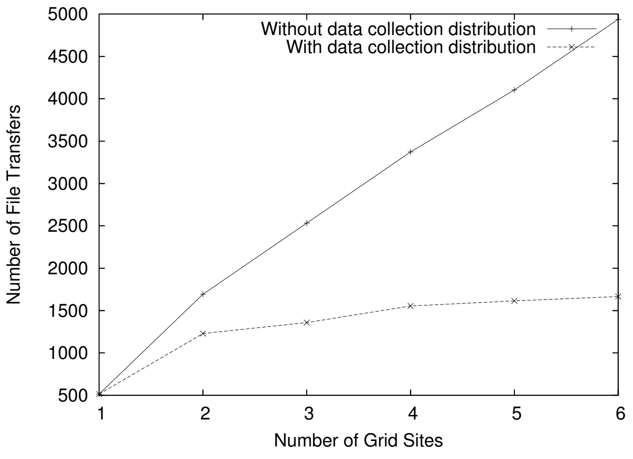
Example 2. The AGWL representation of the WIEN2k workflow

```
  1 <?xml version="1.0" encoding="UTF-8"?>
  2 <agwl name="WIEN2k">
  3     <workflowInput .../>
  4     <workflowBody>
  5         <doWhile name="Conv">
  6             <loopBody>
  7                 <activity name="LAPW0"
  8                           type="wien:lapw0">
  9                     <dataIns .../>
 10                     <dataOuts>
 11                         <dataOut name="kpoint"
 12                                  type="xs:integer" saveto=""/>
 13                     </dataOuts>
 14                 </activity>
 15                 <parallelFor name="pforLAPW1">
 16                     <loopCounter name="taskNumber"
 17                                  type="xs:integer"
 18                                  from="1"
 19                                  to="LAPW0/kpoint"
 20                                  step="1"/>
 21                     <loopBody>
 22                         <activity name="LAPW1"
 23                                   type="wien:lapw1" .../>
 24                     </loopBody>
 25                     <dataOuts>
 26                         <dataOut name="engergyFileCol"
 27                                  type="agwl:collection"/>
 28                         <dataOut name="vectorFileCol"
 29                                  type="agwl:collection"/>
 30                         <dataOut name="scf1FileCol"
 31                                  type="agwl:collection"/>
 32                     </dataOuts>
 33                 </parallelFor>
 34                 <activity name="LAPW2_FERMI"
 35                           type="wien:lapw2FERMI">
 36                     <dataIns .../>
 37                     <dataOuts>
 38                         <dataOut name="weighFileCol"
 39                                  type="agwl:collection"/>
 40                     </dataOuts>
 41                 </activity>
 42                 <parallelFor name="pforLAPW2">
 43                     <dataIns>
 44                         <dataIn name="energyFileCol"
 45                                 type="agwl:collection"
 46                                 source="pforLAPW1/engergyFileCol">
 47                             <constraints>
 48                                 <constraint name="distribution"
 49                                             value="BLOCK(1)"/>
 50                             </constraints>
 51                         </dataIn>
 52                         <dataIn name="vectorFileCol"
 53                                 type="agwl:collection"
 54                                 source="pforLAPW1/vectorFileCol">
 55                             <constraints>
 56                                 <constraint name="distribution"
 57                                             value="BLOCK(1)"/>
 58                             </constraints>
 59                         </dataIn>
 60                         <dataIn name="weighFileCol"
 61                                 type="agwl:collection"
 62                                 source="LAPW2_FERMI/weighFileCol">
 63                             <constraints>
 64                                 <constraint name="distribution"
 65                                             value="BLOCK(1)"/>
 66                             </constraints>
 67                         </dataIn>
 68                     </dataIns>
 69                     <loopCounter name="taskNumber"
 70                                  type="xs:integer"
 71                                  from="1"
 72                                  to="LAPW0/kpoint"
 73                                  step="1"/>
 74                     <loopBody>
 75                         <activity name="LAPW2"
 76                                   type="wien:lapw2TOT" .../>
 77                     </loopBody>
 78                     <dataOuts .../>
 79                 </parallelFor>
 80                 <activity name="Sumpara"
 81                           type="wien:sumpara" .../>
 82                 <activity name="Lcore"
 83                           type="wien:lcore" .../>
 84                 <activity name="Mixer"
 85                           type="wien:mixer">
 86                     <dataIns>
 87                         <dataIn name="scf1FileCol"
 88                                 type="agwl:collection"
 89                                 source="pforLAPW1/scf1FileCol">
 90                             <constraints>
 91                                 <constraint name="element-index"
 92                                             value="1"/>
 93                             </constraints>
 94                         </dataIn>
 95                     </dataIns>
 96                     <dataOuts .../>
 97                 </activity>
 98                 <activity name="testconv"
 99                           type="wien:testconv" .../>
100             </loopBody>
101             <condition .../>
102         </doWhile>
103     </workflowBody>
104 </agwl>
```

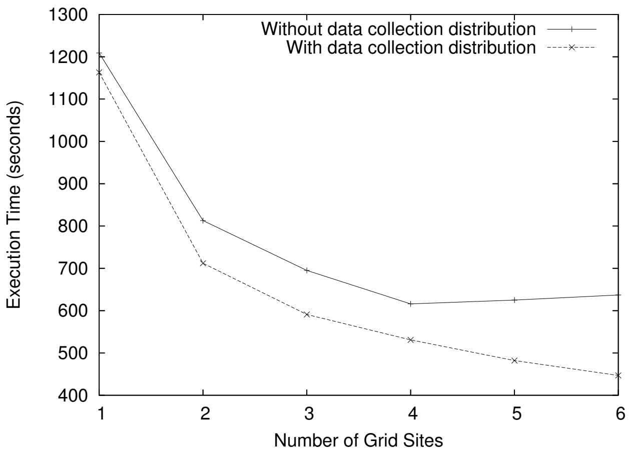
As shown in Fig. 12, we significantly improved the performances for both problem sizes by using collection distribution constructs. Specifically, when executing the workflow with $kpoint = 116$ on 6 Grid sites, the number of file transfers was reduced by 67% and the execution time was reduced by 30% compared with the execution without collection distribution. Accordingly, we achieved the speedup of 2.60 on 6 Grid sites, compared with the maximum speedup of 1.96 achieved on 4 Grid sites when no collection distribution constructs are used. Similarly, in the case where *kpoint* is 252, there was a 68% reduction in the number of file transfers and a 42% reduction in the execution time when executing on 6 Grid sites. And the corresponding speedup was 2.58 on 6 Grid sites, compared with the maximum speedup of 1.74 achieved on 4 Grid sites without collection distribution. By using the collection distribution constructs, the scalability of the workflow was also improved: because of redundant file transfers, the workflow did not scale for more than 4 Grid sites for experiments without collection distribution. Furthermore, the reduction of the file transfer number and the execution time increased with the increase of the number



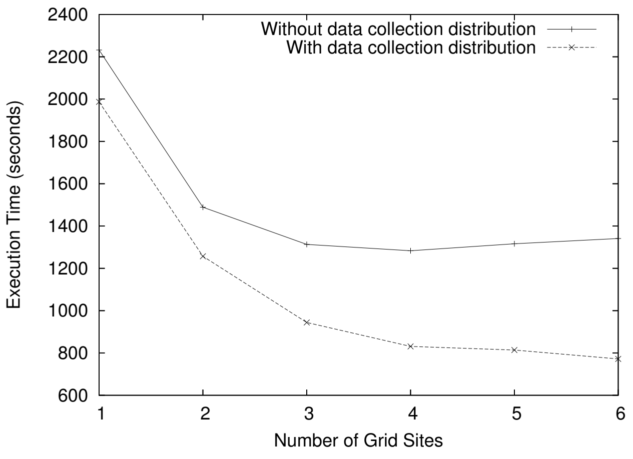
(a) File transfer (kpoint=116)



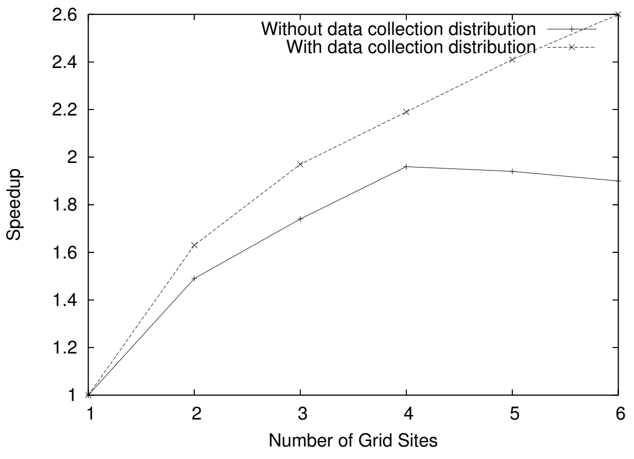
(b) File transfer (kpoint=252)



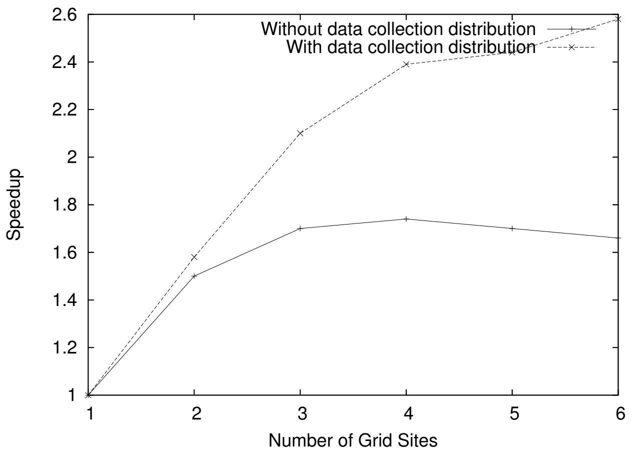
(c) Execution time (kpoint=116)



(d) Execution time (kpoint=252)

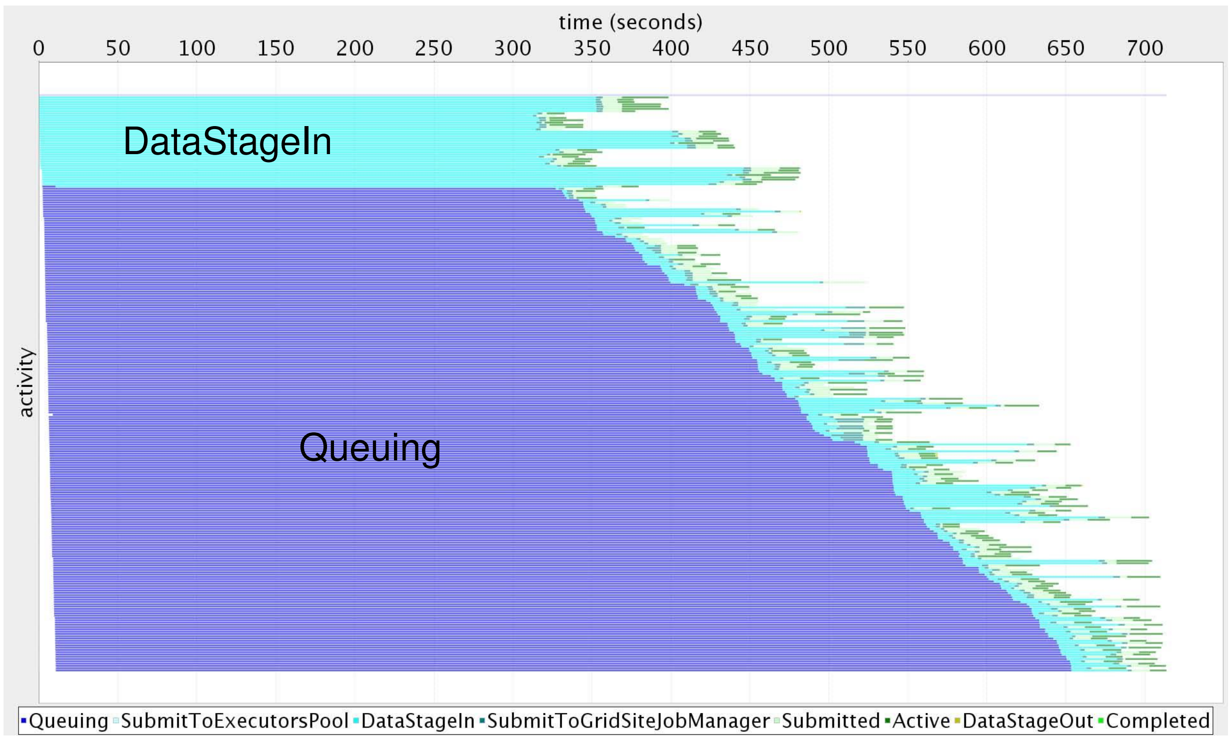


(e) Speedup (kpoint=116)

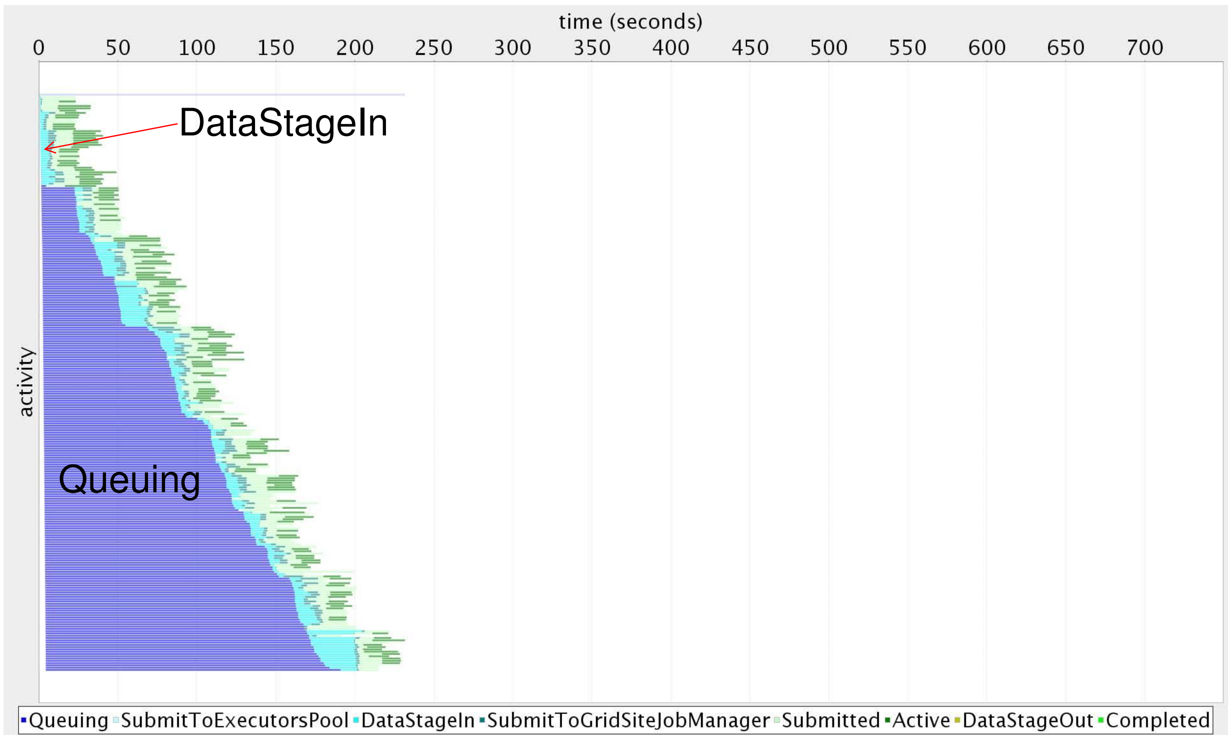


(f) Speedup (kpoint=252)

Figure 12: Experimental results of the WIEN2k workflow



(a) Without data collection distribution



(b) With data collection distribution

Figure 13: Performance analysis of the execution of the `parallelFor` construct *pforLAPW2* on 5 Grid sites (kpoint=252)

of Grid sites involved, especially when *kpoint* is 252. This is because when adding one more Grid site, the entire collections must be transferred to that Grid site for the execution of the loop iterations on that Grid site in case no collection distribution constructs are used. However, if we use the corresponding collection distribution constructs for the data collections in the workflow, only the files required by the iterations scheduled on that Grid site need to be transferred. When increasing the problem size to $kpoint = 252$, the number of file transfers and the corresponding execution time were further reduced due to an increased collection size for increasing problem size.

Fig. 13 illustrates two stacked bar charts which show the performance analysis of the execution of the `parallelFor` construct *pforLAPW2* on 5 Grid sites when $kpoint = 252$. The horizontal axis is the time line and the vertical axis represents activity instances. For each activity instance, the times consumed at different stages (e.g. Queuing, DataStageIn, Active, etc.) are illustrated in a horizontal bar. As illustrated in Fig. 13(a), there exists significant DataStageIn time for the `parallelFor` construct *pforLAPW2*, which results in its total execution time 713.61 seconds. The total execution time of *pforLAPW2* is only 231.70 seconds when using collection distribution constructs (see Fig. 13(b)). The bottom-left blue parts in the two charts represent the time consumed by the activity instances of the activity *LAPW2* for waiting in the job queue of the workflow enactment engine due to the lack of CPU resources.

6.2 Results of the MeteoAG workflow

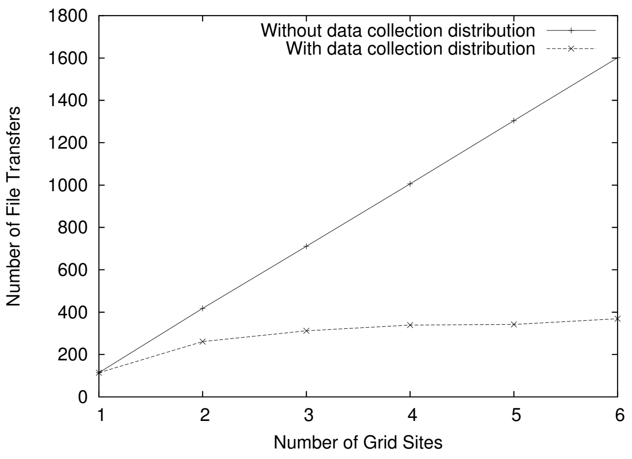
The MeteoAG workflow structure is illustrated in Fig. 9. The corresponding AGWL representation is omitted here to avoid redundancy. In this experiment, we ran the `parallelForEach` construct *pforEachCase* with two parallel loop iterations (corresponding to two simulation cases), each of which has two `parallelFor` constructs with 48 parallel loop iterations (corresponding to 48 simulation time steps). The experiments were conducted on six Grid sites: *karwendel*, *altix1*, *schafberg*, *c703-2201*, *c703-pc450* and *hydra*. We conducted the experiments in the same way as for the WIEN2k workflow: first running the workflow on the fastest Grid sites, then incrementally adding the slower Grid sites.

As illustrated in Fig. 14, we significantly improved the performance of this workflow application when using collection distribution constructs. Compared with the execution on 6 Grid sites without collection distribution, the number of file transfers was reduced by 77% and the execution time was reduced by 53% when using collection distribution constructs. Accordingly, we achieved the speedup of 2.31 on 6 Grid sites, compared with the maximum speedup of 1.70 achieved on 4 Grid sites without collection distribution. And the workflow scalability was also improved when using collection distribution constructs.

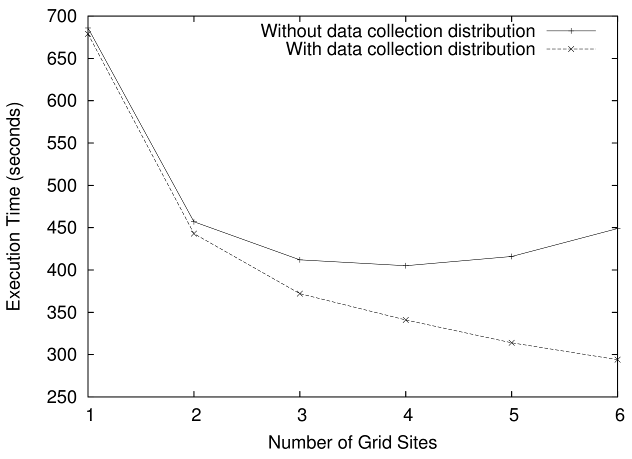
By using collection distribution constructs, the total number of file transfers is reduced and thus the performance is improved. We observed similar behavior for the other two applications, which are omitted here to avoid redundancy.

7. CONCLUSIONS AND FUTURE WORK

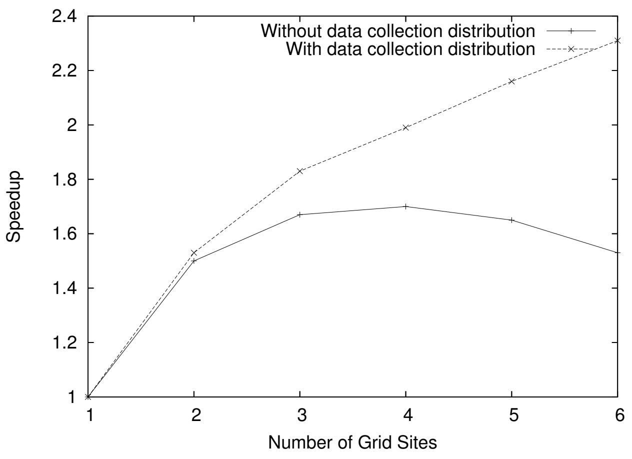
Existing work does not provide a flexible dataset-oriented data flow mechanism to meet the complex data flow requirements of scientific Grid workflow applications. In this paper, we presented an approach as part of AGWL to solve this problem by introducing the concept of *data collection* and the sophisticated collection distribution constructs. A data collection is used to model a static or dynamic dataset at a high level of abstraction. The collection distribution constructs are used to map data collections to activities and to distribute data collections onto loop iterations.



(a) File transfer



(b) Execution time



(c) Speedup

Figure 14: The MeteoAG experimental results

The collection distribution constructs are specified through AGWL constraints *agwl:element-index* and *agwl:distribution*. Five collection distribution constructs, i.e. coma separated colon expressions, *BLOCK*, *BLOCK(S)*, *BLOCK(S,L)* and *REPLICA(S)*, are discussed. With this approach, AGWL enables the specification of fine-grained dataset-oriented data flows in various scientific workflow domains, such as mapping portions of data collections to activities, distribution of data collections onto loop iterations, processing multiple data collections with one parallel iterative construct independently in terms of how many data elements of each collection are processed in one loop iteration. Our approach reduces data duplication, optimizes data transfers between workflow activities, and thus improves workflow performances. It also simplifies the effort to port scientific applications onto the Grid. We demonstrated our approach by applying it to four real world Grid workflow applications and reported the experimental results.

With collection distribution constructs, more advanced data flow like *data stream* can be supported: as soon as the required data elements (instead of the entire data collection) are produced and ready to be processed, the corresponding subsequent activity (the consumer) can start. We are implementing collection-based data stream support in ASKALON. Future extensions to allow more elaborate data collection distribution mechanisms such as *CYCLIC* and *CYCLIC(S)* distributions as well as distributions of nested collections are also being investigated.

8. REFERENCES

[1] The Austrian Grid Project. http://www.austriangrid.at, 2007.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th Int'l Conf. on Scientific and Statistical Database Management (SSDBM'04)*, Santorini Island, Greece, June 21-23, 2004. IEEE Computer Society Press.

[3] P. Blaha, K. Schwarz, G. Madsen, D. Kvasnicka, and J. Luitz. WIEN2k: An Augmented Plane Wave plus Local Orbitals Program for Calculating Crystal Properties, Institute of Physical and Theoretical Chemistry, Vienna University of Technology, Vienna, Austria, 2001.

[4] Condor Team. DAGMan: A Directed Acyclic Graph Manager. http://www.cs.wisc.edu/condor/dagman/, July 2005.

[5] W. R. Cotton, R. A. Pielke, R. L. Walko, G. E. Liston, C. J. Tremback, H. Jiang, R. L. McAnelly, J. Y. Harrington, M. E. Nicholls, G. G. Carrio, and J. P. McFadden. RAMS 2001: Current status and future directions. *Meteorology and Atmospheric Physics*, 82:5–29, 2003.

[6] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. In *6th International Workshop on Grid Computing (Grid 2005)*, Seattle, USA, November 2005. IEEE Computer Society Press. http://www.askalon.org.

[7] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005)*, Cardiff, UK, May 9-12, 2005. IEEE Computer Society Press.

[8] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *14th International Conference on Scientific and Statistical Database Management (SSDBM'02)*, Edinburgh, Scotland, July 2002.

[9] C. H. Koelbel, D. B. Loveman, and R. S. Schreiber. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. The MIT Press, November 1993.

[10] A. Mayer, S. McGough, N. Furmento, J. Cohen, M. Gulamali, L. Young, A. Afzal, S. Newhouse, J. D. V. Getov, and T. Kielmann. *Component Models and Systems for Grid Applications*, volume 1 of *CoreGRID series*, chapter ICENI: An Integrated Grid Middleware to Support e-Science, pages 109–124. Springer, June 2004.

[11] T. McPhillips, S. Bowers, and B. Ludaescher. Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data. In *3rd international Conference on Data Integration for the Life Sciences (DILS)*, Hinxton, UK, November 2006. LNCS/LNBI.

[12] OASIS WSBEPL TC. Web Services Business Process Execution Language (WSBPEL). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, December 2005.

[13] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics Journal*, 20(17):3045–3054, June 2004.

[14] C. Pautasso and G. Alonso. Parallel Computing Patterns for Grid Workflows. In *Proceedings of the Workshop on Workflows in Support of Large-Scale Science*, Paris, France, June 19-23, 2006.

[15] J. Qin, T. Fahringer, and S. Pllana. UML Based Grid Workflow Modeling under ASKALON. In *Proceedings of 6th Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2006)*, Innsbruck, Austria, September 21-23, 2006. Springer Verlag.

[16] N. Russell, A. ter Hofstede, D. Edmond, and W. van der Aalst. Workflow Data Patterns (Revised Version). Technical Report Technical Report FIT-TR-2004-01, Queensland University of Technology, Brisbane, Australia, 2004.

[17] S. Schindler, W. Kapferer, W. Domainko, M. Mair, E. van Kampen, T. Kronberger, S. Kimeswenger, M. Ruffert, O. Mangete, and D. Breitschwerdt. Metal Enrichment Processes in the Intra-Cluster Medium. *Astronomy and Astrophysics*, 435:L25–L28, May 2005.

[18] F. Schüller, J. Qin, F. Nadeem, R. Prodan, T. Fahringer, and G. Mayr. Performance, Scalability and Quality of the Meteorological Grid Workflow MeteoAG. In *2st Austrian Grid Symposium*, Innsbruck, Austria, September 21-23, 2006. OCG Verlag.

[19] L. Silva, G. L. Granato, A. Bressan, C. Lacey, C. M. Baugh, S. Cole, and C. S. Frenk. Modelling Dust in Galactic SEDs: Application to Semi-Analytical Galaxy Formation Models. *Astrophysics and Space Science*, 276(2-4):1073–1078, 2001.

[20] I. Taylor, I. Wang, M. Shields, and S. Majithia. Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience*, 17(9):1197–1214, 2005.

[21] G. von Laszewski, M. Hategan, and D. Kodeboyina. Java CoG Kit Workflow. http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-workflow-book.pdf, 2006.