# A Workflow Runtime Environment for Manycore Parallel Architectures

Matthias Janetschek and Radu Prodan
Institute for Computer Science
University of Innsbruck, Austria
{matthias,radu}@dps.uibk.ac.at

Shajulin Benedict
St. Xavier's Catholic College of Engineering
Chunkankadai, Nagercoil, India
shajulin@sxcce.edu.in

## ABSTRACT

We introduce a new Manycore Workflow Runtime Environment (MWRE) to efficiently enact traditional scientific workflows on modern manycore computing architectures. In contrast to existing engines that enact workflows acting as external services, MWRE is compiler-based and translates workflows specified in the XML-based Interoperable Workflow Intermediate Representation (IWIR) into an equivalent C++-based program. This program efficiently enacts the workflow as a stand-alone executable by means of a new callback mechanism that resolves dependencies, transfers data, and handles composite activities. Experimental results on a number of real-world workflows demonstrate that MWRE clearly outperforms existing Java-based workflow engines designed for distributed (Grid/Cloud) computing infrastructures in terms of enactment time, is generally better than an existing script-based engine for manycore architectures (Swift), and sometimes gets even close to an artificial baseline implementation of the workflows in the standard OpenMP language for shared memory systems.

## Keywords

Scientific workflows, heterogeneous manycore parallel architectures, enactment engine, source-to-source compiler.

## 1. INTRODUCTION

Nowadays, computers exhibit an ever higher number of heterogeneous processing cores with a growing trend to combine general-purpose CPUs with specialized computing units. As a result, modern shared memory heterogeneous manycore systems have become increasingly complex to program, since established programming paradigms are no longer able to fully exploit their heterogeneous performance, but need enhancements or alternative solutions in order to do so. Furthermore, there is also a noticeable trend to employ so called multi-objective optimization methods which consider several possibly conflicting objectives like e.g. performance, energy consumption and costs when optimizing an application.

On the other hand, scientific workflows are a widely successful and established paradigm for programming heterogeneous distributed computing infrastructures (DCI) such as Grids and Clouds. Workflows allow to easily create new larger and more complex applications by reusing existing (often legacy and monolithic) software components named activities and interconnecting them by using well-defined control flow and data flow dependencies. Also they facilitate algorithms for full-ahead scheduling (e.g. HEFT [20]) and furthermore multi-objective optimization [3]. Existing DCI workflow engines are currently mature and come with rich ecosystems that support the user in all aspects of a workflow life-cycle including creation, execution, monitoring and steering, interfaced towards the domain scientists and ease of use rather than the computer science underneath.

Because of the similarity in terms of scale and heterogeneity, workflow systems represent today a promising alternative for development and execution of scientific applications on shared memory heterogeneous manycore architectures. Moreover, as large-scale DCI infrastructures are nowadays composed of powerful manycore parallel machines, exploiting them in an efficient fashion becomes an increasingly important requirement. However, existing workflow engines are typically engineered as external services that target loosely-coupled DCI systems prone to high overheads and latencies. While such overheads are acceptable in distributed systems, tightly-coupled manycore parallel machines are much more sensitive to latencies and other sources of performance penalties.

For this purpose, we propose in this paper a new *Manycore Workflow Runtime Engine (MWRE)*, purposely designed to efficiently exploit the low latency characteristics and resources of manycore parallel architectures. The distinguishing characteristic of MWRE is that it *compiles* an input workflow to an imperative *stand-alone workflow program*, instead of interpreting and orchestrating the workflow specification as traditionally done by today's scientific workflow engines acting as external services. MWRE is based on a source-to-source compiler able to process abstract scientific workflows in the Interoperable *Workflow Intermediate Representation (IWIR)* [15], a common workflow specification developed in the European SHIWA project (SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs)[1] that enables translation of workflow across four major scientific workflow systems: ASKALON [4], MOTEUR [5], Triana [17] and WS-PGRADE [7]. Using an activity repository, the compiler generates a stand-alone C++ workflow program that independently executes on the underlying manycore infrastructure with the help of a workflow engine, linked as an external shared C++ library. Our engine uses a novel low-overhead *callback* mechanism to resolve dependencies, transfer data, and handle composite activities, rather than high-overhead reflection and type introspection as done in existing DCI engines. Experimental results on a number of real-world workflows indicate that MWRE clearly outperforms a rep-

[1]http://www.shiwa-workflow.eu/

DOI: http://dx.doi.org/10.1145/2822332.2822333

resentative engine designed for DCIs (ASKALON), is better than lighter script-based one for manycore parallel platforms (Swift), and even comes close an artificial baseline implementation based on OpenMP, the today's de-facto standard for shared memory parallel programs.

The paper is organised as follows. The next section discusses the related work. Section 3 introduces our workflow model, followed by the main architectural design of our new engine in Section 4. Section 5 presents important technical details about our workflow runtime environment that lead to the performance benefits evaluated in Section 6. Section 7 concludes the paper.

## 2. RELATED WORK

Most scientific workflow systems like ASKALON [4], MOTEUR [5], Pegasus [1], Kepler [9], Taverna [11], Triana [17], or WS-PGRADE [7] are targeted at DCIs such as Grids and Clouds. Swift [19] is an exception by being a lighter parallel script-based engine not restricted to DCIs, but open for general use. Swift is the best suited workflow engine for manycore systems aside our proposed system, but is limited to files in modelling data dependencies and provides no ways of calculating complete workflow schedules.

The most prevalent parallel programming API on shared memory systems is OpenMP[2]. Other native parallel programming APIs like MPI[3] or Charm++ [8] target distributed memory systems, but they all have the common drawback of not modelling data dependencies and not providing means of generating a complete workflow execution plan or schedule.

StarSS [14] is a dependency-driven task execution API for shared memory systems, which also supports distributed memory systems through a hybrid MPI/StarSS approach. It provides directives that annotate C/Fortran source code to define tasks and dependencies between them. StarSS does not allow reusing legacy workflows and does not provide any infrastructure for advanced scheduling.

JOpera [12] is a management tool for business workflows that creates Java byte code from the workflow specification through an event driven state machine instead of directly interpreting the specification. JOpera comes with an integrated workflow design and execution environment as an Eclipse plugin hiding the compilation process from the user. JOpera is tailored to business workflows and does not support advanced full-ahead scheduling.

To the best of our knowledge there is no related work which implements a specialised compiled-based scientific workflow engine for manycore parallel architectures.

## 3. WORKFLOW MODEL

In our workflow model, designing and implementing scientific workflows are accomplished in two parts: an *abstract part* and a *concrete part*. A short overview of these two phases of a scientific workflow implementation is described below:

### 3.1 Workflow Design – Abstract Part

In the abstract part of a scientific workflow design, the following information are formulated from the available dependent or independent tasks: ① the structure of the workflow, ② the activities involved in the workflow (identified by a unique name and a type), and ③ the dependency criteria between the workflow activities.

We define the structure of a workflow as a Directed Acyclic Graph (DAG): $W = (A, D, S)$ (see Figure 1), where $A$ represents workflow activities, $D$ represents the data dependencies, and $S$ denotes the initial start activity. We represent the dependencies between workflow activities $A_i$ and $A_j$ as:

$$D = \left\{ \left(A_i, A_j, Data_{ij}\right) | A_i, A_j \in A \right\},$$

where $Data_{ij}$ denotes the data exchanged between $A_i$ and $A_j$, usually modelled as ports (see Figure 1).

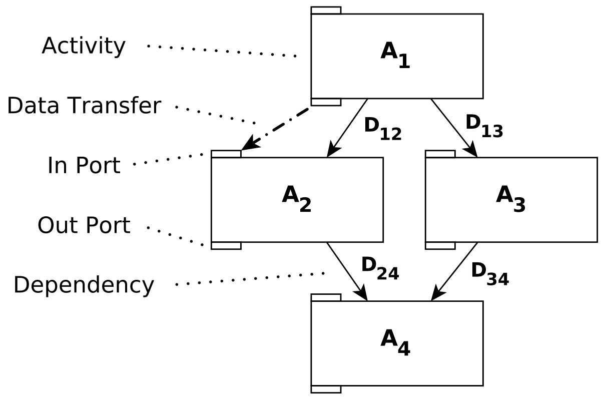


**Figure 1: The abstract part of a scientific workflow.**

There are two different types of workflow activities:

1. *Atomic activities* are basic and indivisible units of computations such as a legacy software;
2. *Composite activities* combine several fine granular workflow activities, including atomic and other composite ones, to form coarse granular activities and impose a control flow on the contained activities.

Typical composite activities are sequential and parallel loops, conditional activities, and sub-workflows.

### 3.2 Workflow Implementation – Concrete Part

The concrete part of a scientific workflow implementation contains detailed information about the atomic activities. This part is often highly specific to each individual workflow system and the underlying DCI and usually contains information about the available activity implementations, locations where they are installed, and how they can be executed.

### 3.3 Workflow Enactment

The workflow activities are mapped to the available resources with the help of scientific workflow enactment engines, optionally in combination with full-ahead DAG schedulers [18]. The responsibility of a workflow enactment engine is to:

- traverse through the structure of a workflow;
- unroll the composite activities whenever required as a preparation for advanced full-ahead scheduling; and
- formulate a *Workflow Execution Plan (WEP)*.

In short, WEPs are represented as DAGs where all composite activities have been unrolled into atomic ones.

[2]http://openmp.org/
[3]http://www.mpi-forum.org/

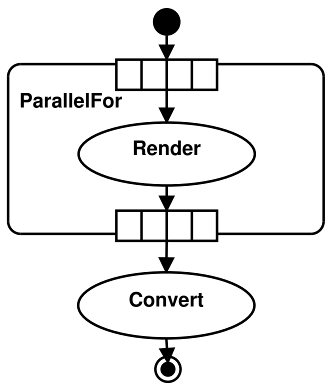


**Figure 2: The POV-Ray workflow.**

**Listing 1** IWIR specification of the POV-Ray workflow.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <IWIR xmlns="http://shiwa-workflow.eu/IWIR" version="1.1" wfname="Povray">
 3   <blockScope name="toplevel">
 4     <inputPorts>
 5       <inputPort name="povFile" type="file" />
 6       <inputPort name="totalFrames" type="integer" />
 7       <inputPort name="framesPerActivity" type="integer" />
 8     </inputPorts>
 9     <body>
10       <parallelFor name="PForLoop">
11         <inputPorts>
12           <inputPort name="numFrames" type="integer" />
13           <inputPort name="totalFrames" type="integer" />
14           <inputPort name="povFile" type="file" />
15           <loopCounter name="frameCounter" from="1" to="" step="" />
16         </inputPorts>
17         <body>
18           <task name="Render" tasktype="RenderTask">
19             <inputPorts>
20               <inputPort name="povFile" type="file" />
21               <inputPort name="startFrame" type="integer" />
22               <inputPort name="numFrames" type="integer" />
23             </inputPorts>
24             <outputPorts>
25               <outputPort name="frames" type="collection/file" />
26             </outputPorts>
27           </task>
28         </body>
29         <outputPorts>
30           <outputPort name="frames" type="collection/collection/file" />
31         </outputPorts>
32         <links>
33           <link from="PForLoop/numFrames" to="Render/numFrames" />
34           <link from="PForLoop/frameCounter" to="Render/startFrame" />
35           <link from="PForLoop/totalFrames" to="Render/totalFrames" />
36           <link from="PForLoop/povFile" to="Render/povFile" />
37           <link from="Render/frames" to="PForLoop/frames" />
38         </links>
39       </parallelFor>
40       <task name="Convert" tasktype="ConvertTask">
41         <inputPorts>
42           <inputPort name="frames" type="collection/collection/file" />
43         </inputPorts>
44         <outputPorts>
45           <outputPort name="outFile" type="file" />
46         </outputPorts>
47       </task>
48     </body>
49     <outputPorts>
50       <outputPort name="finalMovie" type="file" />
51     </outputPorts>
52     <links>
53       <link from="toplevel/povFile" to="PForLoop/povFile" />
54       <link from="toplevel/totalFrames" to="PForLoop/frameCounter/to" />
55       <link from="toplevel/totalFrames" to="PForLoop/totalFrames" />
56       <link from="toplevel/framesPerActivity" to="PForLoop/frameCounter/step" />
57       <link from="toplevel/framesPerActivity" to="PForLoop/numFrames" />
58       <link from="PForLoop/frames" to="Convert/frames" />
59       <link from="Convert/outFile" to="toplevel/finalMovie" />
60     </links>
61   </blockScope>
62 </IWIR>
```

## 4. MANYCORE WORKFLOW RUNTIME ENGINE (MWRE)

Based on the requirements outlined in the introduction, we designed a *Manycore Workflow Runtime Engine (MWRE)* tailored to heterogeneous shared-memory manycore systems. Our main design principle was to provide a set of feature similar to the ones found in the current DCI workflow engines, while addressing the special characteristics and requirements of manycore systems. At first, we briefly describe how we envision the basic architecture of the workflow system, before presenting in detail how our new workflow engine MWRE works. Then, we discuss our workflow representation and at last, we talk about how a workflow is enacted.

We use the Persistence of Vision Raytracer (POV-Ray) workflow as a running example to illustrate the workflow representation and its new enactment mechanism. POV-Ray [13] is a free tool for creating three-dimensional graphics, which is known to be a time-consuming process used not only by hobbyists and artists but also in biochemistry research, medicine, architecture, and mathematical visualization. We modeled a POV-Ray rendering scenario as a workflow, depicted in Figure 2, where the description of a movie can be separated in several scenes, each scene being composed of several frames that can be rendered (for example, in `.png` format) in a `parallel for` loop. Finally, all frames are merged into a `.mpg` movie using a `Convert` activity (e.g. running a `png2yuv` followed by a `ffmpeg` conversion).

Figure 3 presents the overall architecture of the MWRE consisting of three main parts: ① the workflow specification, ② a source-to-source compiler, and ③ the workflow runtime environment. The main difference between MWRE and the traditional DCI workflow engines is the source-to-source compiler that generates a C++ program from the workflow specification. Furthermore, we do the mapping of abstract activity types to concrete activity implementations at compile-time, instead of runtime as done by most DCI engines.

### 4.1 Workflow Specification

The input to MWRE is a workflow specification encoded in the Interoperable Workflow Intermediate Representation (IWIR) [15], an intermediate language that enables interoperability of workflows across different environments. IWIR is currently supported by four workflow systems: ASKALON [4], MOTEUR [5], Triana [17] and WS-PGRADE [7]. Using IWIR for the workflow representation has several advantages. First, because it is designed for interoperability, it captures all concepts and constructs found in most workflow languages, and can therefore be seen as a superset of the major workflow languages. Second, it is well-defined and easy to parse. Third, it allows automatic translation and reuse of workflows across different systems without further modifications. Fourth, it allows integration of new languages and new platforms with O(1) complexity through IWIR front-end or back-end support. The advantage of such an approach is that a scientist can develop and interact with a workflow application by using his favourite language and environment tools, recognised by all interoperable workflow system through automatic IWIR translation.

Listing 1 shows the IWIR specification of our POV-Ray workflow. The `toplevel` composite activity starts at line 3 and ends at line 61. The input ports of the `toplevel` activity (one file and two integers), which also represent the workflow input ports, are defined in lines 4 – 8. A real-world POV-Ray workflow has more input parameters, but we omitted them here brevity reasons. The output ports of the `toplevel` activity (one file), which again represent the workflow output ports, are defined in lines 49 – 51.

### 4.2 Source-to-Source Compiler

The IWIR workflow specification is then given to the source-to-source compiler (②), which translates it into a C++ *workflow program* using the API provided by MWRE. We believe that compiling the workflow application into a native executable program

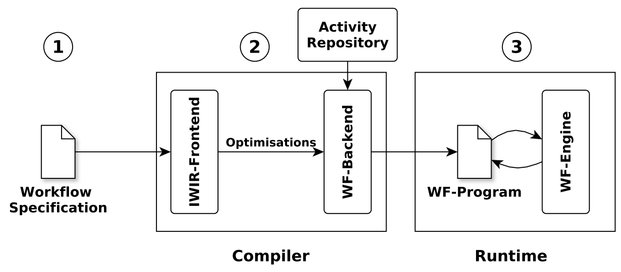


**Figure 3: The MWRE architecture.**

is the way to achieve the "best" performance while minimizing enactment latencies and other middleware overheads. The compiler is also responsible for mapping the abstract workflow activity types to concrete activity implementations. Performing this task at compile-time, instead of runtime as done by most DCI workflow systems, saves additional overhead and eases the configuration. For this purpose, the compiler has access to a repository that contains activity implementations in form of pre-compiled libraries, binary executables, CUDA or OpenCL kernels, source-code snippets, and shell-scripts. The compiler also allows a 1-to-*n* mapping of activity types to implementations, while meta-data stored for each mapping allows to efficiently select a suitable implementation at runtime. Optionally, the mapping phase can also be performed at runtime by employing special stub functions that can dynamically load code.

## 4.3 Runtime Environment

The runtime environment of MWRE (③) invokes and executes the workflow program created by the source-to-source compiler. During the execution of the workflow program, the pre-defined functions interact with the *workflow engine*. The workflow engine is implemented as a C++ library considering several performance concerns, such as overhead, memory footprint, and resource utilization. More detailed information about the MWRE workflow engine and the POV-Ray example is given in Section 5.

## 5. RUNTIME ENVIRONMENT

This section describes the workflow execution methodology, implemented as the runtime environment of MWRE.

## 5.1 Workflow Engine

To meet the requirements of manycore systems, we designed a novel workflow engine that works differently from most traditional workflow engines for DCIs. We paid special attention to efficiently use resources, have low overhead, and be able to use arbitrarily complex data types for ports.

### 5.1.1 Design

Workflow engines for DCIs usually rely on reflection and type introspection to be fully aware of the complete workflow structure and details of the activities. Based on this information, the engine executes the workflow and handles the dependency resolution and the data transfers. One problem with this approach is its high overhead and increased resource use. Furthermore, to reduce the complexity and simplify the engine implementation, only a restricted set of data types for ports is usually supported. While this approach is sufficient for DCIs since complex data structures are usually transmitted via files, on shared memory systems it is more efficient to directly transfer complex data structures via shared memory requiring support for arbitrarily complex data types.

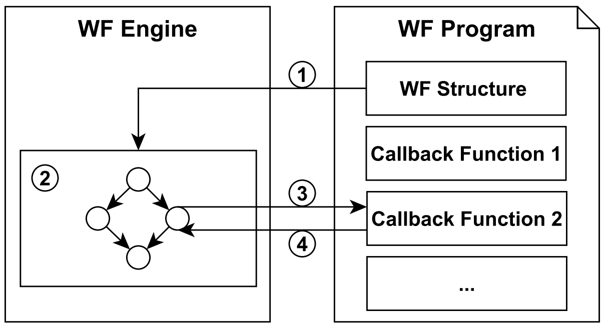


**Figure 4: The MWRE callback design.**

### 5.1.2 Callbacks

To circumvent these problems, we use a novel design which, instead of relying on reflection and type introspection, employs a callback mechanism to resolve dependencies, transfer data, and handle composite activities (see Figure 4). At first (①), the engine reads the workflow specification and constructs a DAG (②). When traversing the DAG, the engine executes so-called *visitor functions* for each node. In traditional engines, the visitor functions are directly responsible for the dependency resolution and data transfers. In contrast, the visitor functions in our engine are only responsible for orchestrating the execution of the associated pre-compiled callback functions (③) that directly modify the state of the associated workflow activities (④). The callback functions are part of the workflow specification and it is in the responsibility of the source-to-source compiler to generate them. Each callback function implements a specific behaviour, such as transferring the input data, collecting the output data from children for composite activities, or resolving the condition of a `while` loop. This approach has the advantage of keeping the workflow engine lightweight and efficient. Instead of knowing all details about the workflow, the engine only knows what it needs to know to traverse the workflow structure and keep track of the execution status, with no need to implement generic functionality for dependency resolution, data transfer and other tasks, that introduce performance overheads. Another advantage is that it allows arbitrary data type support and facilitates extensibility. Every functionality that needs information about data types is encapsulated in a purposely-tailored callback function used in a particular workflow. New functionality can be implemented in callback functions too.

### 5.1.3 Workflow Execution Plan

Our engine implements two modes of evaluating the WEP for full-ahead scheduling.

*Early Evaluation.*

In this mode, the engine evaluates the workflow structure as soon as possible by traversing its structure. For example, it unrolls a `for` loop as soon as it has enough data to evaluate the loop counter instead of waiting for all the input data to be available. This mode has the benefit of generating a complete workflow DAG ahead of time that can be subject to full-ahead optimised scheduling (even though it cannot yet be executed because of the lack of input data). This mode is a prerequisite for IWIR's data streaming extension. Its disadvantage is that it causes additional overheads since every finished

**Listing 2** Visitor function of a `parallel for` loop.

```
 1: function VISITPFOR(activity instance $AI_i$)
 2:     if $\neg AI_i$.ALLINPUTSAVAILABLE then
 3:         $AI_i$.INPUTCALLBACK
 4:     end if
 5:     if $AI_i$.state < ReadyForExecution then
 6:         $AI_i$.FORCOUNTERCALLBACK
 7:     end if
 8:     if $AI_i$.state = ReadyForExecution then
 9:         if $\neg AI_i$.ISUNROLLED then
10:             UNROLLLOOP($AI_i$)
11:         end if
12:         VISITCHILDREN($AI_i$)
13:     end if
14:     if $AI_i$.state $\geq$ ReadyForExecution $\wedge \neg AI_i$.ALLOUTPUTSAVAILABLE then
15:         $AI_i$.OUTPUTCALLBACK
16:     end if
17:     if $AI_i$.state < Finished $\wedge$ all children are finished then
18:         $AI_i$.SETSTATE(Finished)
19:     end if
20: end function
```

**Listing 3** POV-Ray workflow program snippet.

```
 1: ActivityImplemenation RenderImpls[] = {
 2:     /* {type, function-pointer, key-value pairs} */
 3:     PThread, &renderImplFunc, {{"key1", "value1"}, {"key2", "value2"}}
 4: };

 5: ActivityImplemenation ConvertImpls[] = {
 6:     PThread, &convertImplFunc1, {{"key1", "value1"}, {"key2", "value2"}}
 7:     PThread, &convertImplFunc2, {{"key3", "value3"}}
 8: };

 9: ActivityTemplate Activities[] = {
10:     /* (name, ID, parentID, branchID, succs, preds, callbacks, activity-specific) */
11:     AT::Container("wf_container", 1, 0, 0, {}, {}, callbacks, ...),
12:     AT::PForLoop("for", 2, 1, 0, {4}, {}, callbacks, ...),
13:     AT::Atomic("render", 3, 2, 0, {}, {}, callbacks, RenderImpls, 1, ...),
14:     AT::Atomic("convert", 4, 1, 0, {}, {2}, callbacks, ConvertImpls, 2, ...),
15: };

16: Workflow workflow("povray", Activities, 1);

17: int main() {
18:     wf_input wf_int = ...;
19:     wf_output wf_out = WorkflowEngine.startWorkflow(workflow, wf_in);
20: }
```

activity immediately triggers the evaluation of its successors.

*Late Evaluation.*

In this mode, the engine postpones the evaluations of activities until really necessary for continuing the execution. For example, a `for` loop gets unrolled only when all the input data is available. This mode causes lower overheads at the expense of not supporting full-ahead optimised scheduling and data streaming.

### *5.1.4 Scheduling and Activity Execution*

After the WEP has been evaluated, the scheduler is notified of any changes in the WEP. The responsibility of the scheduler is then to map the atomic activities to computing resources. For this purpose, MWRE provides a generic interface designed to plugin new scheduling algorithms, including any information about the WEP and its activities. A scheduler implementation should then use the meta-information of the activity implementations to select an appropriate implementation for execution on a specific device. While the activity implementation meta-information can consist of arbitrary key-value pairs, each execution subsystem provides its own set of well-defined key-value pairs on which the scheduler can base its decisions. After an activity implementation has been mapped onto a computing resource, the activity and its associated information is sent to the appropriate execution subsystem (e.g. pthread subsystem for x86 code, CUDA/OpenCL subsystem for GPU kernels). For this purpose, MWRE provides again a common extensible interface for the execution subsystems designed to support a wide variety of computing resources and to allow the scheduler interact without knowing the details of the underlying subsystem.

### *5.1.5* `Parallel for` *example*

Listing 2 shows the visitor function executed when visiting a `parallel for` loop. At first, it checks whether all input data is already available and if not, calls the input callback function (see Listing 5, line 1) in line 3. When the `for` loop is in a state before `ReadyForExecution`, the `for` counter callback (see Listing 5, line 24) gets executed in line 6. If the activity state is `ReadyForExecution`, indicating that the `for` loop counter has been correctly set up, the loop will be unrolled and an activity instance is created for each child and each iteration in line 10. Afterwards, the visitor functions for these activity instances are called in line 12. In our case, the atomic activity `render` is the only child. The visitor function of an atomic activity executes the input callback function, which sets the activity state to `ReadyForExecution` when all the inputs are available. Afterwards, the visitor function of the `for` loop activity executes the output callback function in line 15 and, when all child activities have finished their execution, the activity state is set to `Finished` in line 18.

When the engine has finished traversing the workflow structure, the (either complete or incomplete) WEP is sent to the scheduler which executes all ready-to-execute atomic activities. When an atomic activity finished its execution, the scheduler notifies the engine which re-evaluates the WEP starting from the recently finished activity. The extend of reevaluation depends on the selected WEP evaluation mode. This process continues until all activities reach the `finished` state and the workflow results can be retrieved from the `toplevel` activity output data structure.

## 5.2 Workflow Program

### *5.2.1 Model*

Generally speaking, a workflow program consists of data structures representing the input and output ports, the callback functions, tables representing activity implementations, activities and their relations, and a main function that starts the workflow engine and passes the activity tables. Mathematically speaking, a workflow $WP = (M_W, A, A_S)$ consists of workflow meta-information $M_W$ (e.g. workflow name), $n$ activities $A = \bigcup_{i=0}^{n} A_i$, and a start activity $A_S$. An activity $A_i = (M_A, IN, OUT, pred(A_i), succ(A_i), I, H)$ consists of activity meta-information $M_A$ (e.g. activity name and activity identifier), an input data structure $IN$, an output data structure $OUT$, immediate predecessors $pred(A_i)$ and successors $succ(A_i)$, $n$ activity implementations $I = \bigcup_{i=0}^{n} I_i$ (empty set in case of composite activities), and $m$ activity-specific callback functions $H = \bigcup_{i=0}^{m} H_i$. An activity implementation $I_i = (M_I, f)$ consists of implementation meta-information $M_I$ (e.g. type of implementation and implementation specific information), and a function pointer $f$ to the actual code.

### *5.2.2 POV-Ray*

Listing 3 shows the tables representing the activities and the workflow itself. The table in line 1 represents the single implementation of the `Render` activity, while the table in line 5 represents the two separate implementations of the `Convert` activity. The supplied key-value pairs are used to select an appropriate implementa-

**Listing 4** Data structures representing the ports of a POV-Ray workflow (we omitted the boolean `"valid data"` flag for each structure component and the structures for the `Convert` activity for brevity reasons).

```
1:  struct toplevel_input {
2:      string povFile;
3:      int totalFrames;
4:      int framesPerActivity;
5:  }
6:  struct toplevel_output {
7:      string finalMovie;
8:  }
9:  struct PForLoop_input {
10:     string povFile;
11:     int totalFrames;
12:     int numFrames;
13: }
14: struct PForLoop_output {
15:     collection frames;
16: }
17: struct RenderTask_input {
18:     string povFile;
19:     int startFrame;
20:     int numFrames;
21: }
22: struct RenderTask_output {
23:     collection frames;
24: }
```

tion at runtime. Line 9 features a table representing the workflow activities. Each table entry contains some meta-information about an activity (e.g. name and activity identifier), dependencies to other activities (e.g. parent activity, successors and predecessors), pointers to the associated callback functions, and other activity type-related information. Line 16 defines the workflow itself through its name, reference to the activity table, and start activity. Lines 17 – 20 finally execute the workflow.

The execution of the POV-Ray workflow starts with the `toplevel` activity (see Listing 3, line 11), whose input and output ports represent the workflow inputs and outputs. The `toplevel` activity usually does not have an input callback, as the input data structure already contains the required valid data. Therefore, its children are visited next starting with the `parallel for` loop.

Listing 4 shows the data structures representing the input and output ports of the POV-Ray workflow activities. Each activity is represented by two data structures, one containing the data fields for each input port and the other for each output port. For example, lines 1 – 5 represent the input ports and lines 6 – 8 represent the output ports of the `toplevel` activity, which correspond to the lines 4 – 8 and lines 49 – 51 in the IWIR specification (see Listing 1). Each data field also includes a boolean flag to indicate whether it contains valid data. They are never accessed by the workflow engine, but only by the callback functions.

Listing 5 shows the callback functions of the `parallel for` loop of the POV-Ray workflow. The first callback (line 1) is responsible for initializing the input data structure. To achieve this, it accesses the input data structure of the parent activity, checks if it contains valid data, saves it into the input structure of the `for` activity, and sets the `allInputsAvailable` flag to `true` if all input data fields contain valid data to avoid unnecessary function invocations. The second callback (line 12) is responsible for filling in the output data structure by accessing the output data structures of its children, checking the validity of the data, storing the data into the output structure of the `for` loop, and setting the `allOutputsAvailable` flag to `true` if all output data fields contain valid data. The third callback (line 24) is responsible for setting the loop counter of the `for` loop activity `PForLoop` by checking if all required data is available, setting the counter, and changing the activity state to `ReadyForExecution`.

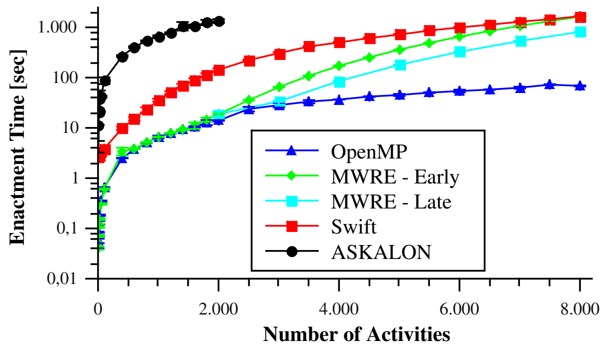
**Listing 5** Callback functions related to the `parallel for` loop activity of the POV-Ray workflow.

```
1:  function FORLOOPINPUTCALLBACK(ForLoopInstance for)
2:      for_in ← RETRIEVEINSTRUCT(for)        ▷ Initialisation of for_in input port
3:      parent_in ← RETRIEVEINSTRUCT(GETPARENTINSTANCE(for))
4:      if ISVALID(parent_in.povFile) then
5:          for_in.povFile ← parent_in.povFile
6:      end if

7:      [...] ▷ Similar code for initializing totalFrames and framesPerActivity ports

8:      if ISVALID(for_in.povFile, for_in.totalFrames, for_in.framesPerActivity)
    then
9:          for.SETALLINPUTSAVAILABLE(true)
10:     end if
11: end function

12: function FORLOOPOUTPUTCALLBACK(ForLoopInstance for)
13:     for_out ← RETRIEVEOUTSTRUCT(for)
14:     for i ← 0 to for.getIterationCount() do
15:         child_out ← RETRIEVEOUTSTRUCT(GETCHILDINSTANCE(for, i, 0))
16:         if ISVALID(child_out.frames) then
17:             for_out.frames.set(i, child_out.frames)
18:         end if
19:     end for
20:     if CONTAINSNVALIDENTRIES(for.GETITERATIONCOUNT, for_out.frames)
    then
21:         for.SETALLOUTPUTSAVAILABLE(true)
22:     end if
23: end function

24: function FORLOOPCOUNTERCALLBACK(ForLoopInstance for)
25:     for_in ← RETRIEVEINSTRUCT(for)
26:     if isValid(for_in.totalFrames, for_in.framesPerActivity) then
27:         for.SETCOUNTER(0, for_in.totalFrames/for_in.framesPerActivity, 1)
28:         for.SETSTATE(ReadyForExecution)
29:     end if
30: end function
```

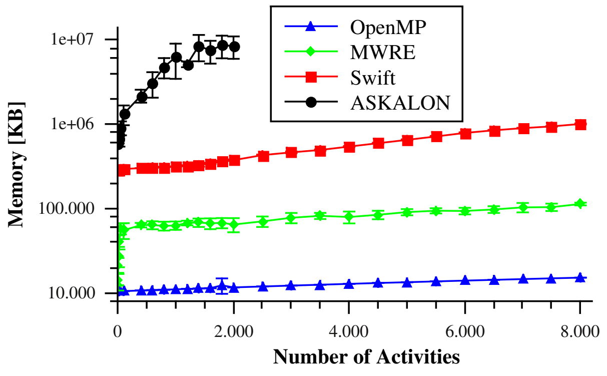
## 6. EXPERIMENTS

We evaluated MWRE by comparing it with two related ones: a fully-fledged workflow environment for Grid and Cloud computing (ASKALON [4]) and a general purpose lightweight workflow scripting language (Swift [19]). The goal of the experiments was to verify the callback-driven approach used in designing the engine and to test its scalability.

To allow a low baseline comparison in the evaluation, we also implemented synthetic OpenMP versions for each workflow application, as OpenMP is the current standard for programming parallel applications on shared memory architectures. We automatically generated the MWRE workflow programs from the IWIR specifications (exported from ASKALON) and the concrete parts from the ASKALON's resource management deployment files using a Java source-to-source compiler that translates IWIR to C++. We conducted the experiments on a system with 4 Intel Xeon E7-4870 10-core CPUs at 2.40 GHz with a total of 128 GB of RAM. We used the GNU C compiler (GCC) version 4.9.1 to compile the MWRE and OpenMP programs with the `-O3` optimization flag. For each experiment, we recorded the enactment time of the workflow programs in both early (labelled `MWRE-Early`) and late (labelled `MWRE-Late`) modes, and their memory utilisation. Enactment time refers to the time spend by the engine while evaluating the WEP and resolving dependencies. Regarding memory, we used the resident size reported by the operating system for MWRE and OpenMP, while for ASKALON and Swift we used the memory statistics delivered by the Java runtime API. Since there is no difference in memory consumption between `MWRE-Early` and `MWRE-`



(a) Enactment time.



(b) Memory utilisation.

**Figure 5: Experimental results for the POV-Ray workflow.**

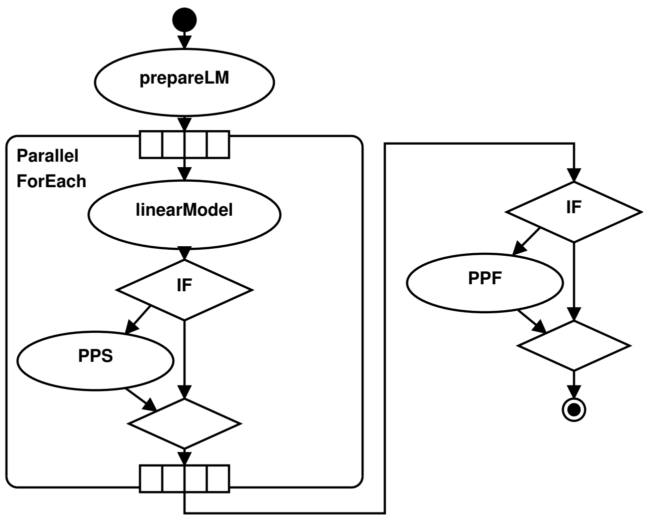


**Figure 6: The RainCloud workflow.**

Late, we report a joint result for both. To obtain a more accurate measure of the workflow enactment overhead and its memory consumption, we replaced the actual atomic activities with dummy implementations that only create empty files to satisfy data dependencies, and recorded the makespan of the workflow executions. We did not use the advanced features provided by the MWRE-Early mode (e.g. streaming and taking advantage of the additional information for scheduling) because we were only interested in evaluating the engine's overhead. The real benefits of these features are highly dependent on the scheduler that is not in the scope of this paper. We used the minimum completion time [10] scheduling algorithm in all experiments as it has low overhead, a low linear complexity and is resilient to prediction inaccuracies.

We use a logarithmic y-axis in all the figures for a better visualisation of the results.

## 6.1 POV-Ray

POV-Ray is our running workflow example with a simple structure presented in detail in Section 4 and Figure 2.

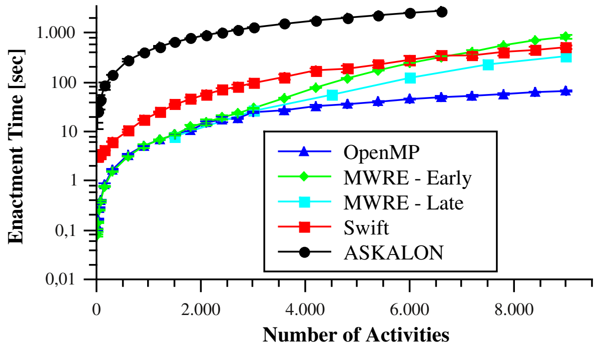
The results in Figure 5 show that until about 2000 activities the enactment time of MWRE is between $0.04-18$ seconds for $5-2000$ activities, which is similar to OpenMP. Above 2000 activities, OpenMP stays between $15-69$ seconds, while MWRE-Early increases up to 1640 seconds showing the same performance as Swift for 8000 activities. MWRE-Late shows a better performance increasing only up to 820 seconds for 8000 activities. Since the POV-Ray workflow consists mostly of a single parallel loop, nearly all activities are scheduled for execution at the same time, which overloads MWRE's engine and limits its scalability. MWRE-Late prevents many reevaluations and gets twice better performance than MWRE-Early. Swift is 50 times worse than MWRE for a low number of activities, but this difference decreases showing the same performance as MWRE-Early for 8000 activities, and twice worse than MWRE-Late. The reason for this result is that Swift does not provide support for maintaining a complete WEP (i.e. it operates in "late" mode with no support for "early"), but is rather designed to efficiently use local resources. ASKALON shows an enactment time between $11-1350$ seconds for $5-2000$ activities. Above 2000 activities, ASKALON's enactment engine begins to exceed the amount of available memory. Like Swift, ASKALON has been designed for heavy-weight distributed systems that do not submit many concurrent sequential tasks to individual cores, but a smaller number of bags of tasks or parallel programs instead. Similar to MWRE, ASKALON generates complete WEPs for full-ahead scheduling (i.e. operates in "early" mode) which brings additional performance and memory penalties.

After a rapid increase in the beginning, MWRE's memory utilization remains relatively constant between $90-110$ megabytes and slowly increases with the number of activities. We consider this a low consumption given that real-world activity implementations typically consume gigabytes of memory. In comparison, OpenMP uses between $10-15$ megabytes, Swift between $280-1000$ megabytes, and ASKALON between $590-8500$ megabytes.

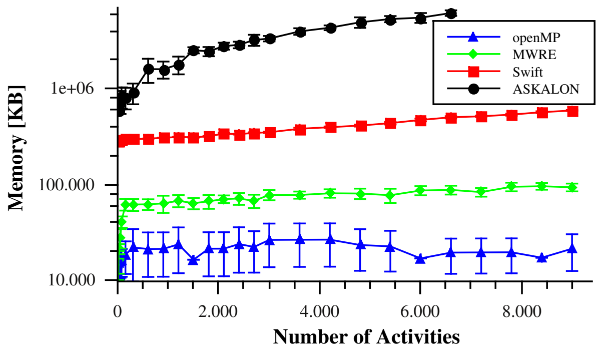
## 6.2 RainCloud

RainCloud is a meteorological workflow for investigating and simulating precipitations in mountainous regions using a simple numerical linear model of orographic precipitations [16]. The workflow is currently used by the Tyrolean avalanche service for their daily avalanche bulletins. Its structure, displayed in Figure 6, is very similar to POV-Ray, but contains a few additional conditional activities. In our experiments, the conditional activities always evaluate to true so that the PPS and PPF activities are executed.

The results in Figure 7 are similar to POV-Ray. In the beginning, the enactment time of MWRE is between $0.8-30$ seconds for $11-3000$ activities, which is similar to OpenMP. Above 3000 activities, OpenMP stays between $25-66$ seconds with $3000-9000$ activities, while MWRE-Early worsens up to 835 seconds for 9000 activities, and even exceeds Swift at around 7000 activities. MWRE-



(a) Enactment time.



(b) Memory utilisation.

**Figure 7: Experimental results for the RainCloud workflow.**

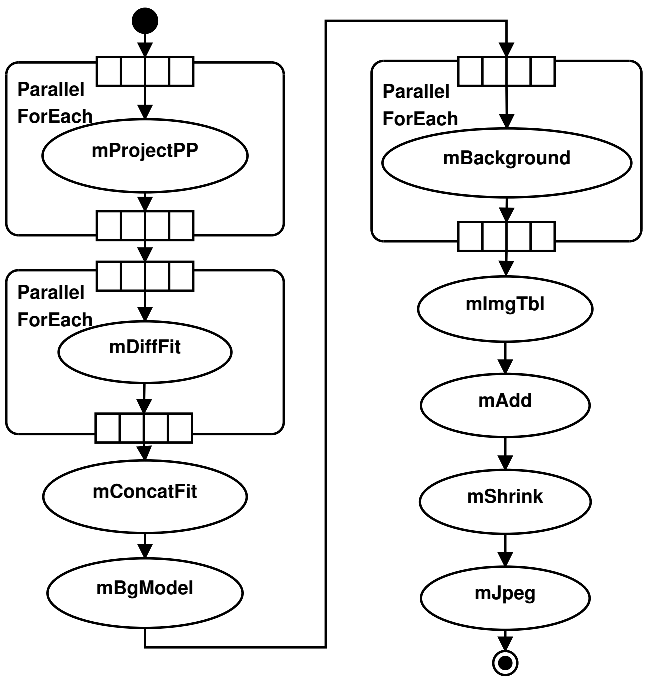


**Figure 8: The Montage workflow.**

Late performs better up to 338 seconds for 9000 activities, which is still worse than OpenMP but better than Swift. The reason for these results is similar to the POV-Ray case because of its similar structure. However, since the `linearModel` and PPS activities are sequentially executed in each `parallel` loop iteration, the number of activities to be simultaneously scheduled is lower than for POV-Ray. Because of this, MWRE's performance only starts to significantly worsen above 3000 activities (instead of 2000). Swift shows an enactment time between 3 – 513 seconds for 5 – 9000 activities. For a low number of activities, it is 40 times higher than MWRE, but this difference decreases to 1.5 times higher than `MWRE-Late` and even 1.6 times faster than `MWRE-Early` for 9000 activities. ASKALON shows an enactment time between 28 – 2813 seconds for 11 – 6600 activities. Above 6600 activities, ASKALON's memory utilisation begins to exceed its allocated size again.

After a rapid increase, the memory utilization of MWRE is relatively constant between 60 – 95 megabytes and slowly increases with the number of activities. In comparison, OpenMP uses between 16 – 21 megabytes, Swift between 283 – 590 megabytes, and ASKALON between 590 – 6200 megabytes.

## 6.3 Montage

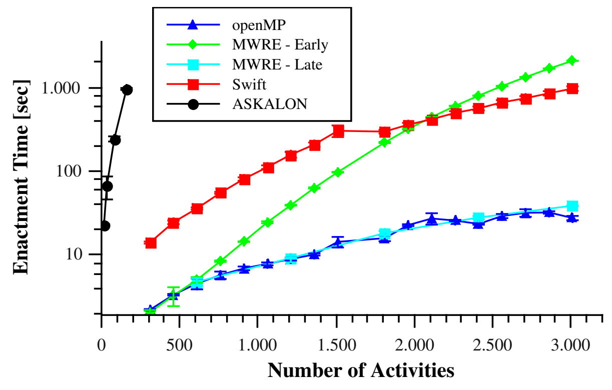
Montage [6] is a well-known workflow in the scientific computing community created by NASA/IPAC, which stitches together multiple images to create mosaics of the sky. Montage has a rather complex structure briefly sketched in Figure 8, making its enactment the most computationally expensive from all our workflows.

The results in Figure 9 show that until about 600 activities the enactment time of MWRE is between 2 – 5 seconds for 300 – 600 activities, which is similar to OpenMP. Above 600 activities, OpenMP stays between 4.6 – 27.6 seconds for 600 – 3000 activities, while `MWRE-Early` increases up to 2074 seconds for 3000 activities, and even exceeds Swift at around 2000 activities due to the high number of reevaluations caused by the complex workflow structure. In contrast, `MWRE-Late` always performs similar to OpenMP with an enactment time of 4.8 – 38.7 seconds for 600 – 3000 activities due to the small number of activities simultaneously scheduled. Swift shows an enactment time between 14 – 968 seconds for 300 – 3000 activities. For a low number of activities, it is seven times higher than `MWRE-Early`, but this difference decreases until Swift gets even twice faster. Compared to `MWRE-Late` and OpenMP, Swift's performance gets up to 35 times worse. ASKALON's enactment time is between 23 – 958 seconds for 16 – 160 activities. Above 160 activities, ASKALON's memory utilisation begins to exceed its allocated size again due to the internal implementation of the collection data type, extensively used for activity ports in Montage.

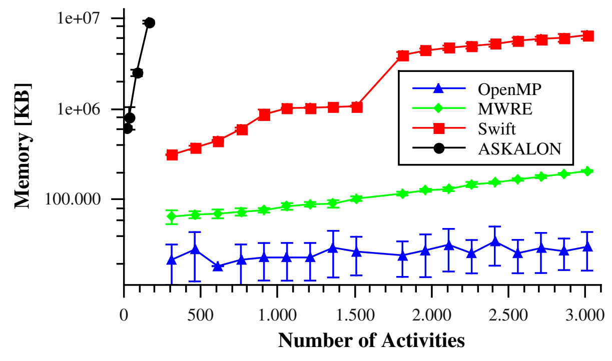
The memory utilisation of MWRE is relatively constant between 65 – 207 megabytes and slowly increases with the number of activities. In comparison, OpenMP uses between 21 – 30 megabytes, Swift between 312 – 6500 megabytes, and ASKALON between 614 – 9000 megabytes.

## 6.4 Sparselu

The Sparselu workflow (see Figure 10) adapted from the `sparselu` program from the BOTS benchmark suite [2] does LU factorisation of sparse matrices. The workflow comes in two flavours: with a `for` and with a `while` sequential outermost loop. In terms of WEP generation in MWRE, there is a big difference between a `while` and a `for` loop. While we can unroll a `for` loop once we know the loop iteration counter and generate the complete WEP, this is not possible for a `while` loop that requires reevaluation of the loop condition after each iteration with no indication on the total number of iterations. The workflow contains three consecutive `parallel`



(a) Enactment time.



(b) Memory utilisation.

**Figure 9: Experimental results for the Montage workflow.**

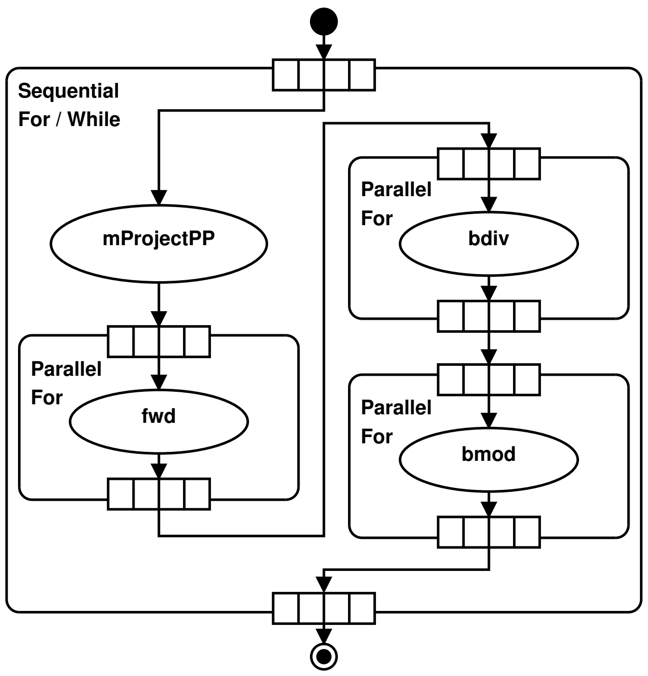


**Figure 10: The Sparselu workflow.**

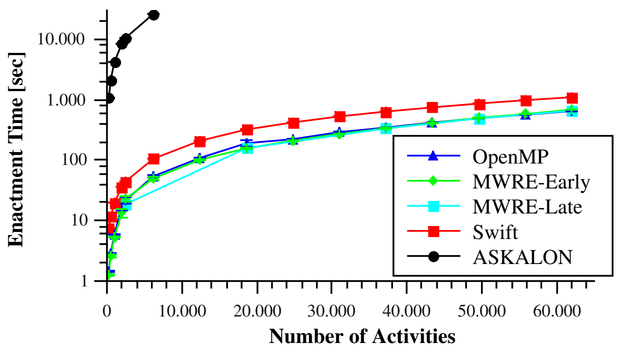
`for` loops with the same number of iterations within the sequential outermost loop. The experiments labelled `*-Length` in Figures 12 and 11 only modify the iteration number of the sequential outermost loop (i.e. the length of the workflow), while the experiments labelled `*-Width` only modify the iteration number of the inner `parallel for` loops (i.e. the width of the workflow). In the first case, we increase of total number of activities while keeping the number of activities scheduled at the same time constant, while in the second case we also modify the number of activities simultaneously scheduled.

The results in Figures 12 and 11 show that the enactment time of MWRE is similar to the one of OpenMP in most cases. The performance of `MWRE-Early` begins to degrade in the `*-Width` experiments above 30000 activities, but is still better than Swift. OpenMP and both MWRE variants have an enactment time of about 1.3 – 680 seconds for 250 – 62000 activities in the `*-Length` experiments and about 2.8 – 300 seconds for 402 – 30000 activities in the `*-Width` experiments. Above 30000 activities, `MWRE-Early` increases up to 830 seconds for 42000 activities in the `*-Width` experiments, while OpenMP and `MWRE-Late` exhibit about 300 – 400 seconds between 30000 – 42000 activities. Because of the nested loop structure, only a small number of activities are simultaneously scheduled for execution (much less than in Montage), which allows MWRE to efficiently execute WEPs containing a very high number of activities. When the number of activities simultaneously scheduled is kept constant (the `*-Length` experiments), there is no performance penalty when increasing the total number of activities. However, when we increase this number (the `*-Width` experiments), we see a performance degradation above 30000 activities. Finally, there is no difference in MWRE's performance between the `For-*` and `While-*` experiments, which indicates that constructing the complete WEP in the beginning or progressively during execution presents no differences in performance. The enactment time of Swift is between 7 – 1200 seconds for 250 – 62000 activities in the `*-Length` experiments, and between 17 – 3600 seconds for 400 – 42000 activities in the `*-Width` experiments. For a low number of activities, it is five times higher than OpenMP and MWRE, but this difference decreases up to 1.7 times higher for 62000 activities in the `*-Length` experiments, and 1.5 times for around 2500 activities in the `*-Width` experiments. Afterwards, it starts to increase again until it gets 8.6 times higher than OpenMP and `MWRE-Late`, or four times higher than `MWRE-Early` for 42000 activities. ASKALON's enactment time is between 1000 – 26000 seconds for 250 – 6000 activities in the `For-Length` experiment, and between 248 – 5800 seconds for 400 – 2600 activities in the `For-Width` experiment. Above 6000 respectively 2600 activities, ASKALON's memory utilisation is exceeds its allocated size again. As ASKALON does not support `while` loops, we report no results for this flavor of Sparselu.

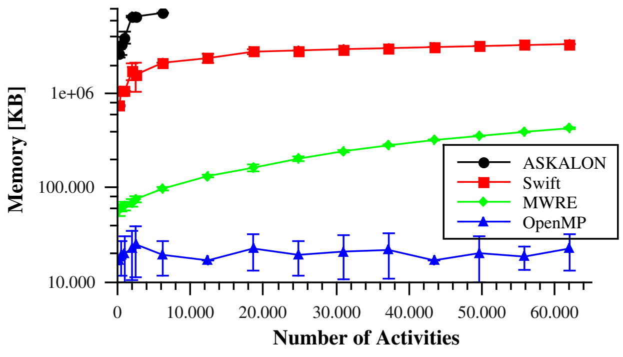
The memory utilisation of MWRE is significantly higher than in the previous experiments (between 15 – 820 megabytes), but comparable if we take into account the higher number of activities involved. In comparison, OpenMP uses between 15 – 32 megabytes, Swift between 550 – 4100 megabytes, and ASKALON between 770 – 7200 megabytes.

## 7. CONCLUSIONS

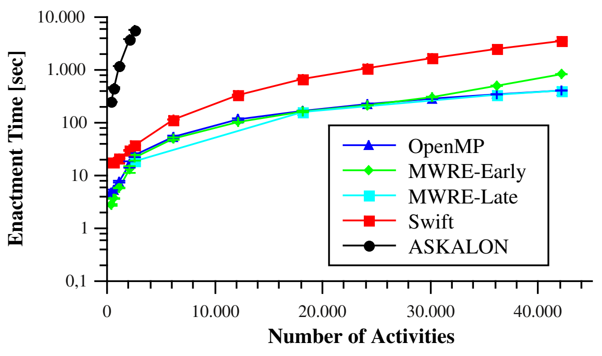
We described a new lightweight Manycore Workflow Runtime Environment (MWRE), purposely designed to efficiently enact traditional scientific workflows on modern manycore computing architectures. In contrast to existing external service-based engines, MWRE is compiler-based and translates workflows represented in the XML-based IWIR specification into an equivalent C++-based



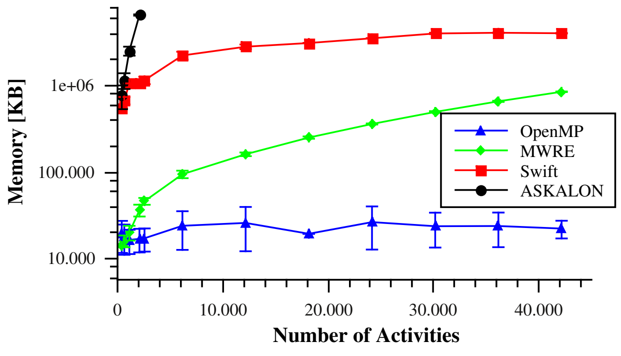
(a) Enactment time (`For-Length`).



(b) Memory utilisation (`For-Length`).

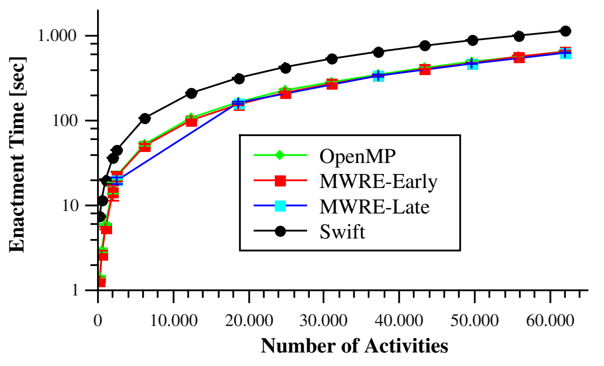


(c) Enactment time (`For-Width`).

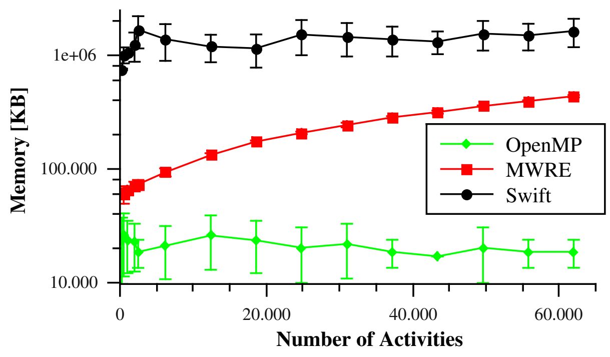


(d) Memory utilisation (`For-Width`).

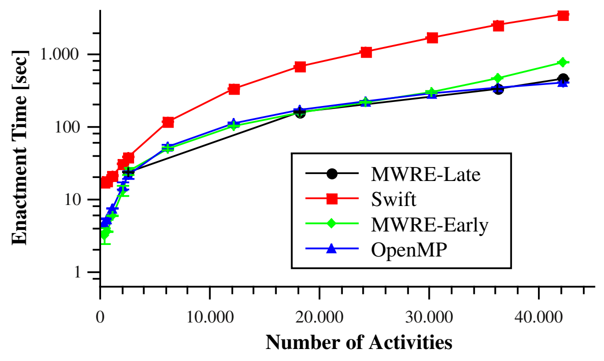
**Figure 11: Experimental results for the Sparselu-`For` workflows.**



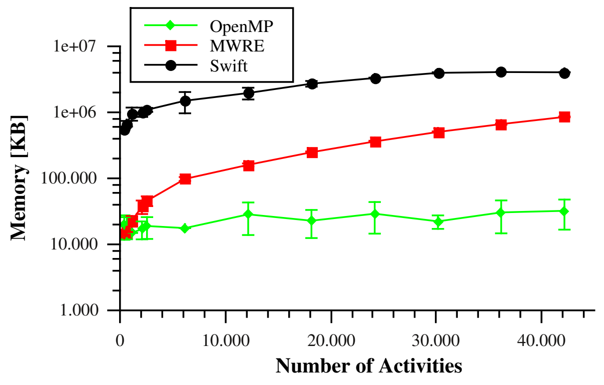
(a) Enactment time (`While-Length`).



(b) Memory utilisation (`While-Length`).



(c) Enactment time (`While-Width`).



(d) Memory utilisation (`While-Width`).

**Figure 12: Experimental results for the Sparselu-`While` workflows.**

workflow program. The workflow program efficiently enacts the workflow as a stand-alone executable by means of a new callback mechanism to resolve dependencies, transfer data, and handle composite activities, rather than high-overhead reflection and type introspection used in existing DCI engines.

We compared MWRE with two representative workflow engines: a Java-based one designed for DCIs (ASKALON) and a script-based one designed for manycore architectures (Swift). Our results demonstrate that employing a compiled workflow program tailored to the needs of manycore platforms exhibits a lower enactment time and less memory consumption. In particular, MWRE efficiently handles complex workflows with a high number of activities, and even achieves a similar enactment time to synthetic OpenMP versions for certain types of workflow applications. The main contributing factor to the performance of MWRE is the length of the WEP and the ratio of the activities simultaneously scheduled to the overall number of activities. The smaller this ratio is, the lower the performance penalty gets. MWRE performs much better than ASKALON in all situations, and it can also perform better than Swift depending on the workflow. It can even come close to the performance of OpenMP. The experiments also showed that MWRE performs especially well with complex workflows which have a long critical path compared to the other workflow engines. Incidentally, this is also the class of applications that will benefit the most from advanced full-ahead scheduling which we plan to investable in future research.

The memory consumption of MWRE is significantly higher than OpenMP, but within low acceptable limits, and much better in other workflow engines designed for distributed systems. While the late WEP evaluation mode exhibits in general less overhead than the early one, early evaluation usually provides more information for full-ahead scheduling and adds support for streaming. The best WEP evaluation mode usually depends on the workflow structure, the number of activities, and the desired feature set.

While the `WEP-Late` evaluation mode exhibits in general less overhead than the early one, `WEP-Early` provides more information that can be used for full-ahead scheduling and adds streaming support. The specific performance difference between the two modes is highly dependent on the workflow structure and size. In future research, we plan to investigate heuristics that allow to automatically select the best evaluation mode for a given workflow, including hybrid modes that dynamically switch between early and late evaluation to gain performance while keeping the benefits of the early evaluation.

## Acknowledgement

This work is supported by the Austrian Science Fund (FWF) project TRP 237-N23: "Workflows on manycore processors" and by the joint FWF – Indian Department of Science and Technology (DST) project I 1523: "Energy-aware autotuning for scientific applications".

## 8. REFERENCES

[1] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto Grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[2] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *International Conference on Parallel Processing*, pages 124–131. IEEE Computer Society, 2009.

[3] J. J. Durillo and R. Prodan. Multi-objective workflow scheduling in Amazon EC2. *Cluster Computing*, 17(2):169–189, 2014.

[4] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, J. Q. Stefan Podlipnig, M. Siddiqui, H.-L. Truong, A. Villazón, and M. Wieczorek. ASKALON: A development and grid computing environment for scientific workflows. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 450–471. Springer, 2007.

[5] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec. Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR. *International Journal of High Performance Computing Applications*, 22(3):347–360, 2008.

[6] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2):73–87, 2009.

[7] P. Kacsuk. P-GRADE portal family for grid infrastructures. *Concurrency and Computation: Practice and Experience*, 23(3):235–245, 2011.

[8] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[9] B. Ludaescher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience, Special Issue on Scientific Workflows*, 18(10):10391065, August 2005.

[10] M. Maheswarana, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131, November 1999.

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

[12] C. Pautasso. Compiling business process models into executable code. *Handbook of Research in Business Process Management*, pages 318–337.

[13] T. Plachetka. POVRAY – Persistence of Vision Parallel Raytracer. In *Proceedings of Computer Graphics International '98*, pages 123–129, 1998.

[14] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[15] K. Plankensteiner, R. Prodan, M. Janetschek, J. Montagnat, D. Rogers, I. Harvey, I. Taylor, Á. Balaskó, and P. Kacsuk. Fine-grain interoperability of scientific workflows in distributed computing infrastructures. *Journal of Grid Computing*, 11(3):429–455, 2013.

[16] F. Schüller, S. Ostermann, M. Janetschek, R. Prodan, and G. Mayr. The raincloud project: Harnessing cloud computing for a meteorological application at the tyrolean avalanche service. In *Geophysical Research Abstracts*, volume 15, page

9710, 2013.

[17] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana applications within grid computing and peer to peer environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[18] M. Wieczorek, A. Hoheisel, and R. Prodan. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Generations Computer Systems*, 25(3):237–256, 2009.

[19] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[20] H. Zhao and R. Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *Euro-Par 2003 Parallel Processing*, pages 189–194. Springer, 2003.