Many-Core Event Evaluation

Jean-Eudes Marvie* Technicolor
Patrice Hirtzlin† Technicolor
Pascal Gautron‡ Technicolor

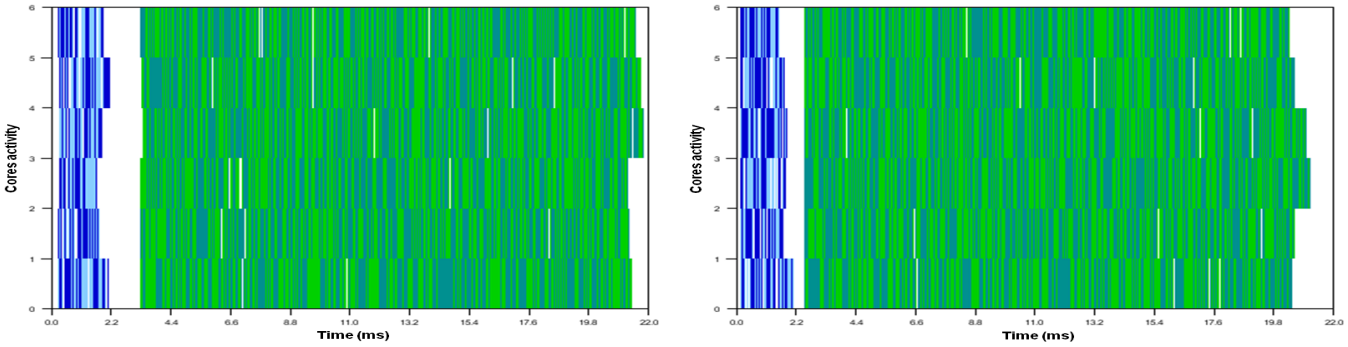


Figure 1: *We introduce a novel event evaluation framework using task parallelism on multi/many-core CPU architectures for real-time animation of complex scenes. Our framework supports 5 dynamic scheduling strategies based on OpenMP and custom Pthread-based approaches. Parallel evaluation timings for an animation frame on a 6-core CPU are displayed for the implemented custom Pthread-based (left) and OpenMP-based (right) static schedulers for a population of 1K humanoids generating 110K events per frame. Each rectangle corresponds to the evaluation of a single scene graph node (bones transformations and skeletal constraint evaluation at left and right figure parts respectively). The color of each consecutive rectangle is modified automatically for readability purpose. The gaps correspond to the time spent either in sequential execution or idle.*

Abstract

We present a Many-Core Event Evaluation framework for real-time execution of many complex animation schemes applicable to a wide range of domains such as gaming and interactive pre-visualization in studio production. Our technique takes advantages of task parallelism on many-core CPU architecture using a two-level scheduling approach. Our generic approach can deal with tens of thousands event-processing nodes, event loops, non-deterministic and interaction-driven animation modifications at runtime. Versatility is further enforced through a native support of hierarchical animation graphs using *prototypes* and *inline files*. Our implementation based on the X3D event-based animation model exhibits performances approaching the theoretical upper bounds of parallelization.

1 Introduction

Virtual worlds feature more and more complex animation schemes simulating various phenomena such as crowd displacement or cloth deformation. An animation scheme can be modeled by a graph whose nodes process the events carried by the edges. In the context of realistic character motion and object deformation, such graph may contain tens of thousands event-processing nodes.

Besides the challenge of executing one of these animations in real-time, interactive virtual worlds also require both non-deterministic and interaction-driven modifications of animation graph at runtime. Non-deterministic events can be generated dynamically by a scripting node. Interaction-driven events can be generated by sensor nodes to execute new animation schemes.

After a review of event processing and task parallelism (Section 2), we introduce in Section 3 a generic Many-Core Event Evaluation (MCEE) framework based on a two-level scheduling approach. Our framework provides real-time execution of many complex animation schemes, each of them handling tens of thousands event-processing nodes and supporting both non-deterministic and interaction-driven animation modifications at runtime. Moreover, our framework supports cyclic animation graphs.

Modeling these complex animation schemes can be simplified by reusing existing sub-animation schemes described in user-defined (*prototype*) and sub-file (*inline file*) nodes. These additional hierarchies in the animation graph also have to be accounted for when parallelizing the event evaluation. As shown in Section 4, our framework seamlessly provides a consistent speedup for each individual animation sub-graph.

In Section 5 we illustrate the performance of our framework by proposing a parallel event evaluation implementation based on X3D.

2 Related work

2.1 Event-based execution model

Numerous multimedia systems such as X3D, Qt, MPEG4 system or Flash Action Script rely on the event-based model. The events are propagated along paths between specific fields of the scene graph nodes in charge of the animation. The input fields trigger the node

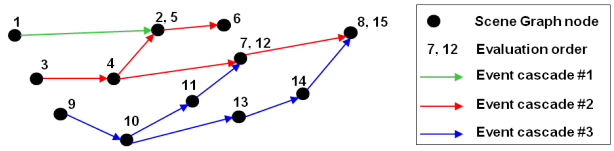


Figure 2: *In an event cascade principle, initial events trigger node evaluations which generate new events toward other nodes. A node belonging to n cascades is evaluated n times.*

*e-mail: jean-eudes.marvie@technicolor.com
†e-mail: patrice.hirtzlin@technicolor.com
‡e-mail: pascal.gautron@technicolor.com

evaluation, modifying its states accordingly. The output fields are then the events generated by those state changes. Initial events can either be generated for each frame (e.g. Time Sensor node) or be generated occasionally through user interaction or sensor activation.

The event-based execution model is based on the event cascade principle (Figure 2). An event, such as a change of input fields, immediately triggers the node evaluation. This evaluation may generate new events (e.g. a change in its output fields) towards other nodes, which are then evaluated. A node belonging to n cascades is then evaluated n times. A cascade may contain cycles where an event E is received by a node which generates an event that results in E being generated again. In that case, the cycle shall be detected and broken to execute a single animation sequence per frame and avoid infinite looping.

Following the dataflow programming paradigm derived from Dennis and Misunas [1975], the event evaluation can be seen as task executions in a certain order to satisfy their dependencies. The animation can be then expressed as a directed dependency graph called *animation graph*, in which the event-processing nodes of the scene graph are linked by the event paths representing the dependencies.

The event-based execution model requires a depth-first traversal of the *animation graph*. Parallelism can be directly achieved using a thread per cascade but raises 2 major issues. First, if a node belongs to more than one cascade, a time-consuming synchronization step is required at each evaluation to ensure that all input fields are updated (Figure 3a). Second, parallelism cannot be easily achieved for animation sub-graphs enclosed within *prototype* nodes (Figure 3b).

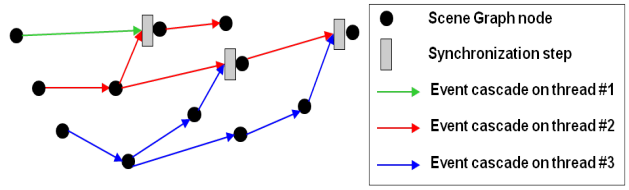
Another issue for parallel event evaluation is the support of events having non-deterministic side effects on the *animation graph* itself at runtime, such as the insertion/removal of nodes and paths (Figure 3c).

2.2 Task parallelism

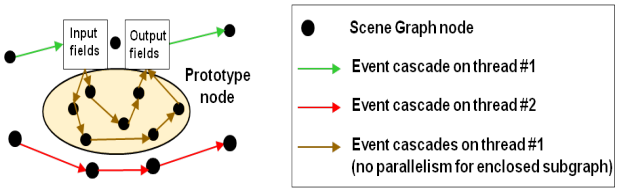
The dataflow programming paradigm is composed of two main steps. The task partitioning step is performed prior to execution to cluster tasks according to the dependency graph. Ideally the tasks within a cluster are independent and can be executed in parallel to reduce synchronization overhead. The order of execution of these clusters is set to meet the dependencies requirements.

The scheduling step is then performed at runtime by iterating on the clusters in the defined order. The tasks of a cluster are distributed among the available processing units. The distribution strategies are numerous, from simple static equal-size task allocation to complex dynamic allocation based on heuristics such as task execution time.

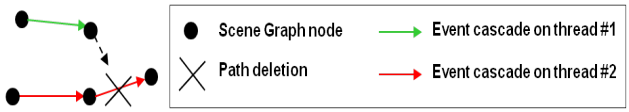
The parallelization of a large number of heterogeneous time-consuming tasks can be achieved using a cluster of processors. The related frameworks rely on underlying dataflow models provided by a distributed middleware. The dataflow model is based on modules, each of them running on a processing unit, exchanging data through connections. DRONE [Repplinger et al. 2009] relies on the Network-Integrated Multimedia Middleware (NMM) [Lohse et al. 2008]. FlowVR Render [Allard and Raffin 2005] relies on FlowVR [Allard et al. 2004] for the distributed rendering and display application. This kind of distributed architecture is also used for collision detection and physics simulation. Allard and Raffin [2006] perform physics simulations using FlowVR [Allard et al. 2004]. Hermann *et al.* [2009; 2010] perform parallel physics simulations at the body level using the KAAPI middleware [Gautier et al. 2007] for dynamic load balancing using a work-stealing mechanism over a cluster of multiprocessors.



(a) Synchronization steps for single node evaluation



(b) Hierarchical animation graph leading to sequential sub-graph evaluation



(c) Node evaluation leading to a path deletion in the animation graph

Figure 3: *Issues for parallel event evaluation. A per-cascade parallelism introduces time-consuming synchronization steps (a) and prevents parallelism for prototype evaluation (b). The support of non-deterministic side effects on the animation graph at runtime (c) requires further synchronization mechanisms.*

These frameworks target an unified solution for parallel execution on both multi-core and cluster environments, hence integrating network management at their lowest level. Our MCEE framework has been designed from ground-up to be executed on a many-core CPU processor of an unique machine leading to optimized lightweight thread management.

More lightweight frameworks exploiting task parallelism have been demonstrated for data visualization [Vo et al. 2010] and physics simulation [Thomaszewski et al. 2008]. However, they do not address hierarchical scene graphs and do not support dynamic side effects on the dataflow.

Related to dataflow in interactive general-purpose Virtual Reality (VR) applications, Figueroa [2010] presents an abstraction for parallel execution within the InTml interaction technique markup language. It provides a mechanism for node replacement in the dataflow allowing references to dynamic objects. However, the insertion and removal of paths, as well as hierarchical scene graphs are not addressed.

Dataflow techniques have already been used for some animation applications. Klein *et al.* [2012] propose a framework for real-time mesh interpolation and skeletal animations. In combination with XML3D, this framework provides high performance general-purpose data processing for real-time or interactive Web applications using a declarative language to describe the dataflow. However, the described execution model uses a pull-based execution phase and hence does not support continuous multidirectional changes. This framework hence does not support graph cycles, hierarchical graphs and nodes whose evaluation may dynamically alter the dataflow. Watt *et al.* [2012] introduce a parallel character animation framework handling thousands of nodes in studio production. This approach provides two levels of parallelism: at animation de-

pendency graph level using Intel Concurrent Collections [Budimlic et al. 2010] and at node level for the expensive nodes such as deformers, tessellators and solvers using Intel Threading Building Blocks [Reinders 2007]. While providing convincing results, this framework supports a limited number of unique evaluation paths (up to 100). Moreover, it also does not support side effects on the dataflow.

Compared to the previous work, our MCEE framework provides the main contributions:

- Support of an arbitrary number of nodes and event routes
- Support of non-deterministic side effects on the *animation graph* at runtime
- Preserved parallelism in hierarchical *animation graphs*
- Support of cycles in the *animation graph*

3 The MCEE framework

3.1 Principle

Two levels of parallelism can be exploited for the event evaluation on many-core CPU architectures. Node level parallelism is suited to the evaluation of time-consuming nodes, such as mesh deformers. This approach is efficient when handling only few nodes as it introduces concurrency among the threads to access to the available processing units. Graph level parallelism is adapted to the evaluation of numerous nodes. The addition of a node-level parallelism to a graph-level parallelism only provides a marginal speedup [Watt et al. 2012]. Therefore, the MCEE framework exploits only the graph level parallelism to achieve real-time animation of complex scenes.

While graph level parallelism can be directly achieved using a thread per event cascade, this approach becomes intractable due to synchronization issues. Instead, our framework uses a dataflow-based approach by partitioning the node evaluation into independent batches in such a way to avoid any thread synchronization. As detailed in Section 3.2, the order of these batches corresponds to a breadth-first traversal of the *animation graph*. Our framework also creates as many threads as the available processing units to avoid thread concurrency.

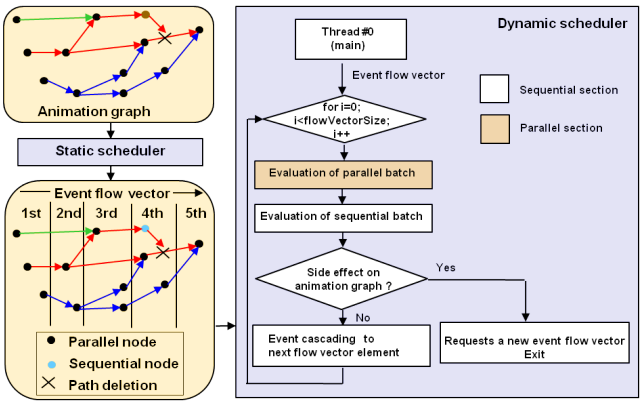


Figure 4: *The MCEE framework is based on a two-level scheduling approach. From the animation graph, the static scheduler creates an event flow vector containing the batches of nodes to be evaluated. At runtime, the dynamic scheduler iterates on the vector elements and creates a succession of parallel and sequential steps.*

Our framework is composed of the task partitioning and scheduling steps (Figure 4). Task partitioning is performed by the *static scheduler* prior to the execution when the *animation graph* has been modified (insertion/removal of nodes and paths). The *static scheduler* (Section 3.2) defines the batches of nodes to be evaluated from the *animation graph* and inserts them into an *event flow vector*. Each element of the *event flow vector* contains two batches for efficient scheduling: The *parallel batch* contains the nodes incurring no side effect on the *animation graph*. The other nodes are processed in the *sequential batch*.

The scheduling step is performed at runtime by the *dynamic scheduler* running on the main thread. This scheduler iterates on the *event flow vector* elements to create a succession of parallel and sequential steps meeting the dependency constraints.

The *parallel batch* is then divided into sub-batches and distributed among the available processing units. These sub-batches can be evaluated independently, without any synchronization. We implemented both static and dynamic allocation strategies (Section 3.3).

The nodes of the *sequential batch* are evaluated sequentially to handle the potential side effects on the *animation graph*. Upon detection of a side effect, the *dynamic scheduler* stops the entire frame evaluation, requests a new *event flow vector* from the *static scheduler* for the current animation frame and re-executes that frame from the beginning. If no side effect has been detected, the event is cascaded towards the next element of the *event flow vector*.

In practice most of the nodes do not modify the *animation graph* and are therefore processed in the parallel batches of the *event flow vector*. This allows the *dynamic scheduler* to achieve significant parallelism speedup.

3.2 Static scheduling

Starting from the *animation graph* the *static scheduler* creates an *event flow vector* containing batches of nodes meeting the event evaluation dependencies.

The *event flow vector* is created by iterating over the *animation graph* based on the node dependencies (Figure 4 and Algorithm 1). For each node, we first initialize a *dependency counter* with its number of direct predecessors. Each node inserted in the *event flow vector* decrements the *dependency counters* of its direct successors.

The nodes having no predecessor are placed at the first element ($i = 0$) of the *event flow vector*. These nodes have no dependency and hence are ready to be evaluated. Further elements i of the *event flow vector* are created as follows:

$$\begin{aligned} &\forall node \in EFV \;/\; EFV.rank(node) = i - 1, \\ &\forall node' \in Succ(node) \;/\; node'.c = 0 \\ &\Rightarrow EFV.rank(node') = i, \end{aligned} \tag{1}$$

where $node$, and $node'$ are nodes of the *animation graph*, EFV is the *event flow vector*, $rank(node)$ is the element of the *event flow vector* which contains $node$, $Succ(node)$ is the set of direct successors of $node$ in the *animation graph* and c is the node *dependency counter*.

Nodes incurring side effects on the *animation graph* can be easily identified using a flag or the type of the event path. The evaluation of these nodes requires synchronization to update the *animation graph* and hence cannot be performed in parallel. Therefore, each element of the *event flow vector* contains a *sequential batch* for these specific nodes and a *parallel batch* for all the other nodes.

Algorithm 1 Creation of the Event Flow Vector (EFV)

```
EFV.clear()
nodeList.clear() // list of nodes to be sorted
for all node ∈ animationGraph do
  if node.hasPredecessor() then
    // node.visited = FALSE // loop check init
    nodeList.pushBack(node)
  else
    if node.isSequential() then
      // add to the sequential batch
      EFV[0][1].insert(node)
    else
      // add to the parallel batch
      EFV[0][0].insert(node)
    end if
  end if
end for

i = 0
while nodeList.size() != 0 do
  // insert Algorithm 2 for loop handling
  for all node ∈ EFV[i] do
    for all node' ∈ Succ(node) do
      // node'.visited = TRUE // loop check set
      node'.c = node'.c - 1
      if node'.c = 0 then
        // add node' to the (i+1) element of EFV
        if node'.isSequential() then
          // add to the sequential batch
          EFV[i+1][1].insert(node')
        else
          // add to the parallel batch
          EFV[i+1][0].insert(node')
        end if
        // remove node' from nodeList
        nodeList.erase(node')
      end if
    end for
  end for
  i++
end while
```

As shown in Figure 4 the nodes contained within a *parallel batch* of an element of the *event flow vector* are independent and can be processed in parallel, avoiding the need for intra-element synchronization.

Handling event loop detection and breaking Without proper detection, event loops result in a failure of Algorithm 1. We therefore enrich this algorithm to detect and break any event loop during the *event flow vector* creation (Figure 5).

Our technique (Algorithm 2) is based on the observation that the element $i-1$ of the *event flow vector* is empty when a loop exists between the nodes to be inserted in the i^{th} element due to the condition of a null *dependency counter*. Therefore, all the successor nodes which have been previously visited but not inserted in the *event flow vector* due to this condition are tagged. Then, once a loop is detected (empty element $i-1$ of the *event flow vector*), it is broken by forcing the addition of a tagged node to this empty element $i-1$.

Algorithm 2 Loop handling

```
if EFV[i].size() == 0 then
  // loop detected
  node = NULL
  isNodeFound = FALSE
  index = 0
  while !isNodeFound do
    node = nodeList[index]
    if node.visited then
      isNodeFound = TRUE
    end if
    index++
  end while
  if node.isSequential() then
    EFV[i][1].insert(node)
  else
    EFV[i][0].insert(node)
  end if
  nodeList.erase(node)
end if
```

3.3 Dynamic scheduling

Running on the main thread, the centralized *dynamic scheduler* iterates linearly over the *event flow vector* (Figure 6) to evaluate the *parallel* and *sequential batches*. The events are then cascaded to the next element of the *event flow vector*.

Parallel batch evaluation We implemented two approaches for the node distribution among the available processing units. A custom Pthread-based approach using a static thread pool and an OpenMP-based approach using a *for* loop iterating over the nodes of each *parallel batch*. We use as many threads as available processing units to avoid thread concurrency.

Static and dynamic node allocation strategies have been tested for both approaches. The static strategy creates as many sub-batches as available processing units. Each sub-batch contains approximately the same number of nodes and is assigned to a thread. This strategy requires less thread synchronization but can lead to unbalanced evaluation time among threads. The dynamic strategy initially allocates one node per thread. Threads then pick unevaluated nodes in a queue. This strategy requires more thread synchronization but improves load balancing (Figure 13).

We also implemented OpenMP-guided strategy with a default chunk size of 1 for benchmark purposes. This strategy provides a tradeoff between the static and dynamic strategies.

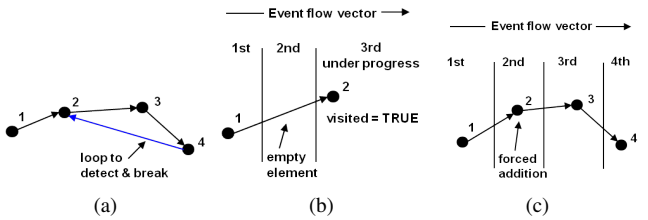


Figure 5: *An event loop (a) is detected when creating the third element of the event flow vector as the second vector element is empty (b). The loop is broken by forcing the addition of the previously visited node to the second vector element (c).*

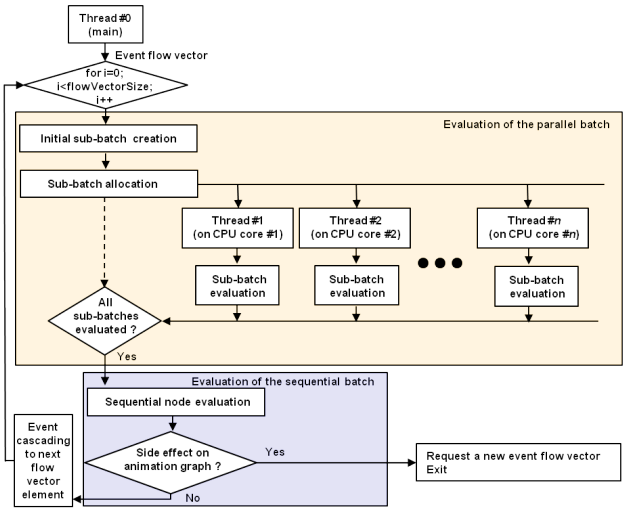


Figure 6: *The dynamic scheduling algorithm is executed on the main thread. Parallel and sequential batch evaluation steps are executed for each element of the event flow vector.*

Sequential batch evaluation This step deals with the evaluation of nodes potentially altering the *animation graph*, including the addition or removal of nodes and event paths. In order to be able to stop the entire node evaluation process upon detection of a side effect, each node of the *sequential batch* is successively evaluated by the main thread. The *event flow vector* is then rebuilt by the *static scheduler* for the current animation frame (see Algorithms 1 and 2). Finally, the *dynamic scheduler* executes the new animation frame from the beginning.

Local event cascading The evaluation of the nodes belonging to the batches of the *event flow vector* elements can also generate new events. Such events are cascaded to the input fields of the nodes belonging to the next element of the *event flow vector* (Figure 7).

Parallel event cascading involves time-consuming synchronization steps before accessing to the input fields of nodes belonging to several cascades. Cascading also requires sequential read and write operations for each event. Therefore we choose to perform event cascading sequentially as no significant parallelism gain can be obtained.

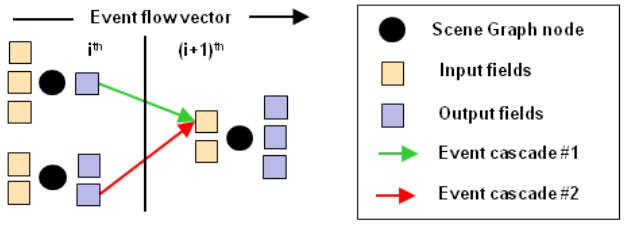


Figure 7: *The evaluation of nodes belonging to the element i of the event flow vector generates a change in their output fields. These new events are cascaded to next element of the event flow vector.*

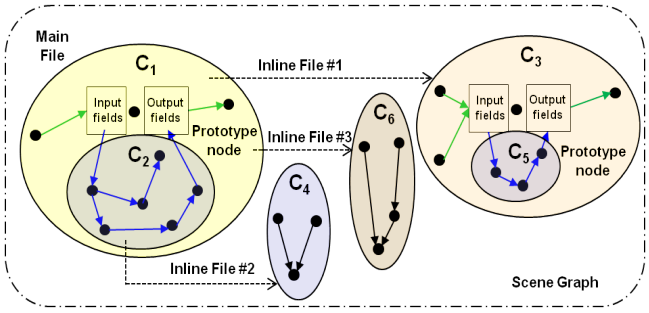


Figure 8: *Animation contexts C_i are created for the main file and for each prototype or inline file node of the hierarchical scene graph.*

4 Hierarchical animation graphs

Many complex animation schemes reuse some existing sub-schemes using *prototypes* and *inline files*, introducing hierarchies in the *animation graph*. We propose an enrichment of our framework to provide a consistent parallelism speedup throughout the entire animation.

4.1 Prototype and inline file mechanisms

A *prototype* is a powerful mechanism for creating user-defined reusable objects. Each *prototype* node instantiates the reusable objects in the scene graph with specific input data values. A *prototype* node contains a scene graph with an associated *animation graph*. This *prototype* scene graph can be considered as an enclosed scene graph interfaced externally through the *prototype* input and output declaration fields.

An *inline file* node also contains a scene graph with an associated *animation graph*. This scene graph is completely independent and can be considered as an adjacent scene graph.

4.2 Hierarchical MCEE framework

We support hierarchical graphs by introducing *animation contexts* created for the main file of the scene graph and for each *prototype* or *inline file* node (Figure 8). Each *animation context* C_i contains its own *animation graph* and its own *static scheduler* generating an *event flow vector*.

The hierarchical MCEE framework shall ensure that each *animation context* is evaluated once per frame while enforcing parallelism across all levels.

Per-frame unique animation context evaluation The *dynamic scheduler* now contains a list of *animation contexts*. For each frame, we evaluate the animation by evaluating each *animation context* if needed (Figure 9).

Many-core animation context evaluation *Animation contexts* can be considered as adjacent in the case of *inline files* or enclosed when *prototypes* are used. Adjacent *animation contexts* can be simply evaluated sequentially by the *dynamic scheduler*, using all the processing units. Enclosed *animation contexts*, however, require an extension of both schedulers to optimize the usage of the computation resources. To this end, we add a *hierarchical batch* to the elements of the *event flow vector*. This batch contains the nodes having

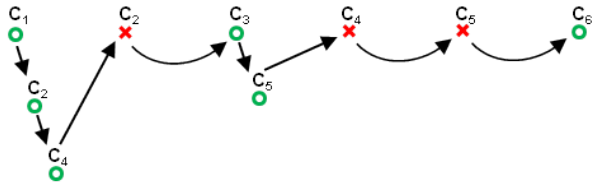


Figure 9: *Per-frame unique animation context evaluation of the scene graph of Figure 8. The dynamic scheduler iterates on its list of animation contexts and triggers the evaluation only if the current animation context was not previously evaluated.*

an enclosed sub graph. A *prototype* node is then inserted into the *hierarchical batch* as well as the *parallel* or *sequential batch*.

At runtime, the *dynamic scheduler* evaluates the *hierarchical batch* between the *sequential batch* evaluation and the cascading steps (Figure 10). The nodes belonging to the *hierarchical batch* are evaluated sequentially, allowing the execution of each sub-animation graph with all available processing units. The execution of each sub-animation graph (triggered by the associated node) is again composed of all the parallel and sequential evaluation steps detailed in Section 3.3. The event cascading step finally cascades the events to the output fields of the *prototype* node, allowing their propagation in the enclosing *animation graph*.

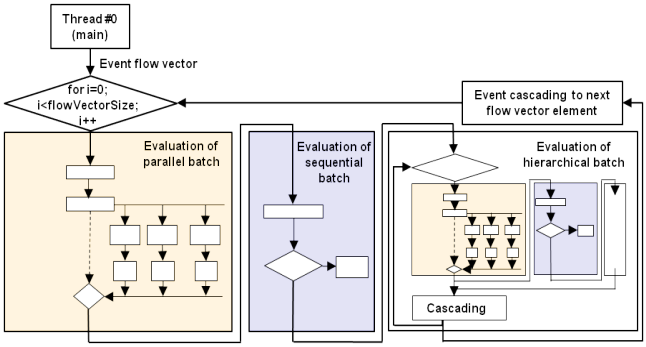


Figure 10: *The dynamic scheduler is extended by adding a new hierarchical evaluation step between the sequential batch evaluation and the cascading steps. This new step evaluates the prototype nodes sequentially, allowing an execution of the animation sub-graphs with all available processing units. Once evaluated, the events are cascaded to the output fields of the prototype node, allowing their propagation in the enclosing animation graph.*

5 Implementation and Results

5.1 X3D event-based animation model implementation

The X3D event-based animation model provides features to support complex animation schemes. The event evaluation module is called the *execution engine*. The event paths are called *routes* and the *animation graph* is called *route graph*.

We implemented our framework within the X3D *execution engine* (Figure 11), whose underlying model enables both continuous and user-activated events. It also specifies *prototypes* and *inline files* and allows the definition of event loops.

Side effects on the *route graph* itself are also allowed at runtime. In order to provide variety in the animation, non-deterministic side

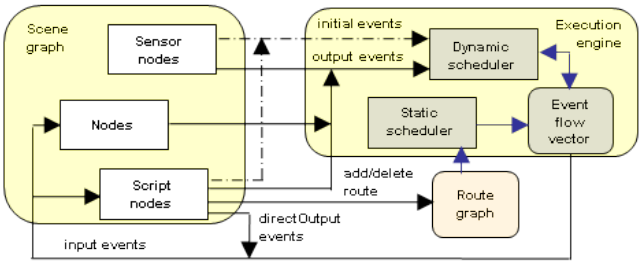


Figure 11: *Our static and dynamic schedulers have been implemented within the X3D execution engine.*

effects (node or path insertion/removal) can be generated through the evaluation of directOutput-enabled Script nodes. The evaluation of any node having routed SFNode/MFNode fields may also lead to node removal. As these characteristics can easily be detected, the nodes having potential side effects are inserted in the sequential node batches when constructing the *event flow vector*.

5.2 Results

We benchmarked our technique on X3D scenes, running on a 6-core 2.67GHz Intel Xeon CPU processor.

Scene 1 is composed of 3,200 objects being continuously translated and rotated. This scene uses a single TimeSensor node to generate an initial event. Then, 3,200 OrientationInterpolator and 1,248 PositionInterpolator nodes cascade the initial event to 3,200 Transform nodes to modify the object positions. A total of 5708 events are generated per frame.

Scene 2 is composed of one animated biped creature. The creature comprises 48,855 vertices split into 63 objects implemented by IndexedFaceSet nodes. The vertices and normal vectors of the geometry are modified using CoordinateInterpolator and NormalInterpolator nodes. A single TimeSensor node is also used to generate an initial event. A total of 252 events are generated per frame.

Scene 3 is composed of a population of 1K humanoids represented using HAnimHumanoid nodes. Two TimeSensor, 54 OrientationInterpolator and one PositionInterpolator nodes are used to animate each humanoid. A total of 110K events are generated per frame.

We compare the speedups measured in Scene 3 (Table 1) to the theoretical upper bounds S given by Amdahl [1967] :

$$S = \frac{1}{1 - p + \frac{p}{n}}, \tag{2}$$

where n is the number of available processors and p is the fraction of job that can be executed in parallel. In Scene 3, p is 1 minus the proportion of the execution time spent in the cascading step.
We achieve speedups close to the Amdahl Law due to the avoidance of inter-thread synchronization. The difference between the measured and theoretical speedups can be explained by the sequential memory accesses and by the data and code cache faults at the operating system level.

Our test scenes are composed of two-fold event cascades. In other words, the size of the *event flow vector* equals to three.

The activity of the 6 cores over a single frame of each scene is charted in Figures 1 and 13 for various scheduling strategies. Each rectangle corresponds to the evaluation of a single scene graph node. The color of each consecutive rectangle is modified automatically for readability purpose. The evaluation time of the TimeSensor is negligible and is not visible. The evaluations of the nodes

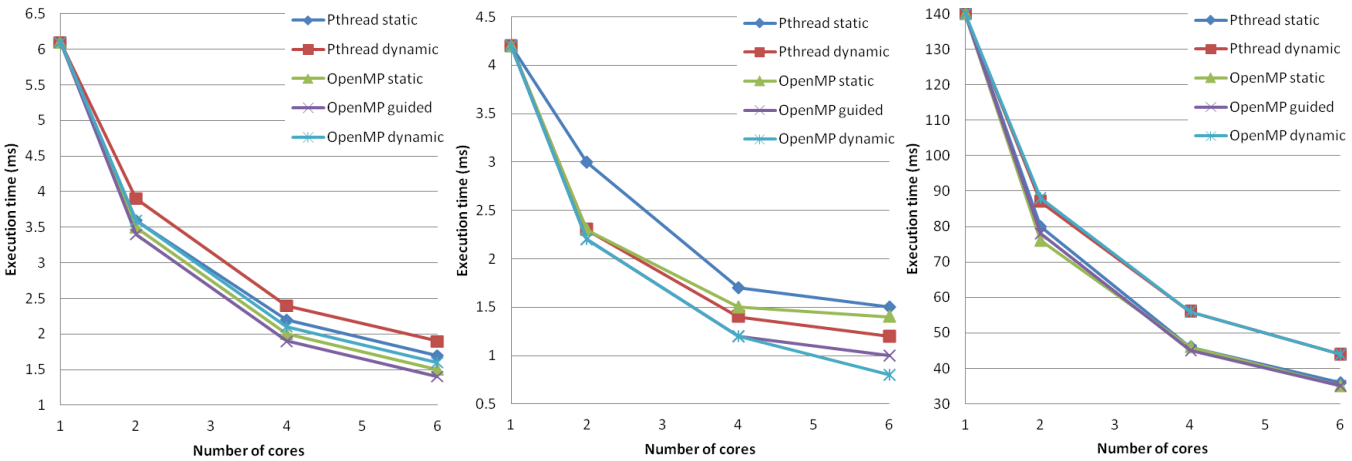


Figure 12: *Measured execution time for a single frame of Scenes 1 (left), 2 (middle) and 3 (right). These values include the time spent in the sequential cascading step, respectively 0.3ms, negligible and 11.5ms.*

	2 cores	4 cores	6 cores
Amdahl Law	1.85	3.21	4.25
Pthread static	1.75	3.04	3.89
Pthread dynamic	1.61	2.50	3.18
OpenMP static	1.84	3.04	4.00
OpenMP guided	1.79	3.11	4.00
OpenMP dynamic	1.59	2.50	3.18

Table 1: *Measured speedups for Scene 3 using 2, 4 and 6 cores, compared to the theoretical Amdahl Law.*

belonging to the second and the third elements of the *event flow vector* are drawn at left and right figure parts respectively. The gaps correspond to the time spent in the sequential event cascading from the output fields of the nodes belonging to the current element to the input fields of the nodes belonging to the next element of the *event flow vector* combined with the idle time.
The OpenMP-based scheduling strategies exhibit better performance compared to our custom Pthread-based implementation as they avoid thread wake-up using spinlocks. However, having all the available processing units performing spin-locking within the event evaluation can also be a drawback if the cores needs to simultaneously perform other tasks. Moreover, spinlocks also reduce battery life and raise the temperature of the devices.

We compare various scheduling strategies with respect to the number of processing units (Figure 12).
A significant parallelism speedup is achieved for all scenes and scheduling strategies. The main speedup is obtained when switching from a mono-core to a dual-core configuration. Executing the animations over more than 6 cores do not significantly improve the speedup. The performance of each scheduling strategy depends on the type of animation. For animations involving the motion of numerous objects or characters with lightweight animation (Scenes 1 and 3), the static scheduling approaches exhibit better performance. Conversely, dynamic scheduling is more adapted to heavy mesh deformation (Scene 2) as the node evaluation times are heterogeneous. The OpenMP-guided scheduling strategy offers a tradeoff which can be chosen for intermediate animation types.

Therefore, all five scheduling strategies have their own benefits and drawbacks. The final choice depends on the user requirements in term of animations and hardware platforms. However, in any context our two-level scheduling scheme will bring the benefits of unhindered parallel computing architectures.

6 Conclusion

We presented a novel lightweight framework for parallel event evaluation on many-core platforms. Our formulation exhibits significant speedups yielding real-time evaluation of many complex animation schemes handling numerous (tens of thousands) events per frame. In some cases we achieve close-to-linear speedups using a 6-core architecture.

The proposed framework is also highly generic, handling complete animation models including nodes having runtime side effects on the *animation graph*, *prototype* and *inline file* mechanisms as well as cyclic event routing.

References

ALLARD, J., AND RAFFIN, B. 2005. A shader-based parallel rendering framework. *Proceedings of IEEE Visualization*, 127–134.

ALLARD, J., AND RAFFIN, B. 2006. Distributed physical based simulations for large VR applications. *Proceedings of IEEE Virtual Reality*, 89–96.

ALLARD, J., GOURANTON, V., LECOINTRE, L., LIMET, S., MELIN, E., RAFFIN, B., AND ROBERT, S. 2004. FlowVR: a middleware for large scale virtual reality applications. *Proceedings of Euro-Par*, 497–505.

AMDAHL, G. 1967. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of American Federation of Information Processing Societies*, 483–485.

BUDIMLIC, Z., BURKE, M., CAV, V., KNOBE, K., LOWNEY, G., NEWTON, R., PALSBERG, J., PEIXOTTO, D., SARKAR, V., SCHLIMBACH, F., AND TASIRLAR, S. 2010. Concurrent collections. *Scientific Programming 18*, 3-4, 203–217.

DENNIS, J., AND MISUNAS, D. 1975. A preliminary architecture for a basic data-flow processor. *Proceedings of the Symposium on Computer Architecture*, 126–132.

FIGUEROA, P. 2010. Insights on the design of InTml. *Presence 19*, 2 (April), 118–130.

GAUTIER, T., BESSERON, X., AND PIGEON, L. 2007. KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. *Proceedings of International Workshop on Parallel Symbolic Computation*, 15–23.

HERMANN, E., RAFFIN, B., AND FAURE, F. 2009. Interactive physical simulation on multicore architectures. *Proceedings of Eurographics Workshop on Parallel Graphics and Visualization*, 1–8.

HERMANN, E., RAFFIN, B., FAURE, F., GAUTIER, T., AND ALLARD, J. 2010. Multi-GPU multi-CPU parallelization for interactive physics simulations. *Proceedings of Euro-Par on Parallel Processing: Part II*, 235–246.

KLEIN, F., SONS, K., RUBINSTEIN, D., AND BYELOZYOROV, S. 2012. Xflow - declarative data processing for the Web. *Web3D symposium*, 37–45.

LOHSE, M., WINTER, F., REPPLINGER, M., AND SLUSALLEK, P. 2008. Network-integrated multimedia middleware (NMM). *Proceedings of ACM Multimedia*, 1081–1084.

REINDERS, J. 2007. Intel threading blocks. *O'Reilly*.

REPPLINGER, M., LOFFLER, A., RUBINSTEIN, D., AND SLUSALLEK, P. 2009. DRONE: A flexible framework for distributed rendering and display. *Proceedings of ISVC*, 975–986.

THOMASZEWSKI, B., PABST, S., AND BLOCHINGER, W. 2008. Parallel techniques for physically based simulation on multi-core processor architectures. *Computer Graphics Forum 32*, 1, 25–40.

VO, H., OSMARI, D., SUMMA, B., COMBA, J., PASCUCCI, V., AND SILVA, C. 2010. Streaming-enabled parallel dataflow architecture for multicore systems. *Computer Graphics Forum 29*, 3, 1073–1083.

WATT, M., CUTLER, L., POWELL, A., DUNCAN, B., HUTCHINSON, M., AND OCHS, K. 2012. LibEE: A multithreaded dependency graph for character animation. *Proceedings of Digital Production Symposium*, 59–66.

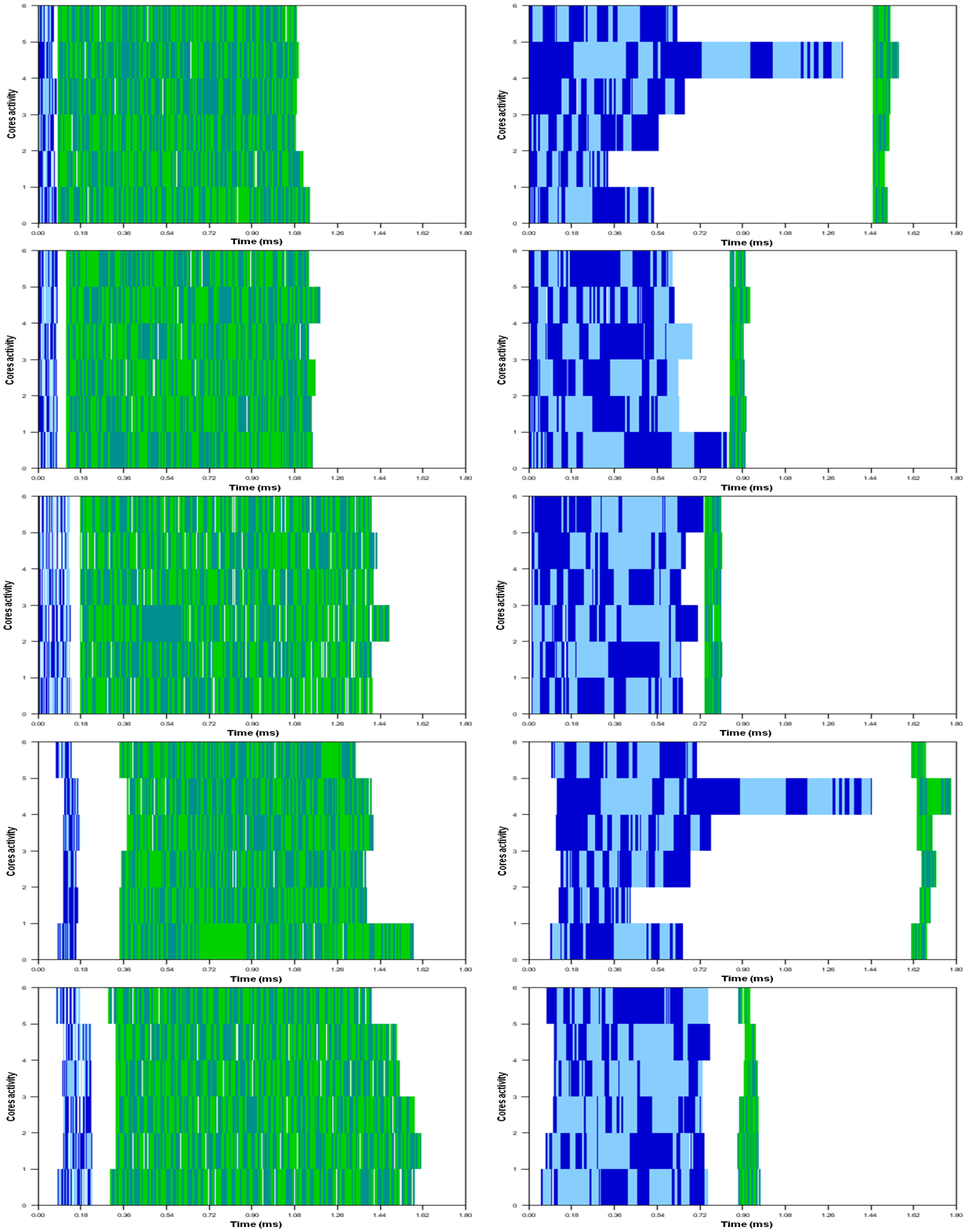


Figure 13: *6-core evaluation of a single frame of the Scene 1 (left) and Scene 2 (right). From top to bottom: static, guided and dynamic scheduling using OpenMP, static and dynamic scheduling using custom Pthread. Each rectangle represents a single animated node. The color of each consecutive rectangle is modified automatically for readability purpose. The evaluation of the nodes belonging to the second and the third elements of the event flow vector are drawn at left and right figure parts respectively. The gaps correspond to either sequential steps or idle time.*