Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

PaTraCo: A Framework Enabling the Transparent and Efficient Programming of Heterogeneous Compute Networks

S. Frey and T. Ertl

Visualisierungsinstitut der Universität Stuttgart, Germany

Abstract

We propose PaTraCo (Parallel Transparent Computation), a framework for developing parallel applications for single host or ad-hoc compute network environments incorporating a multitude of different kinds of compute devices including graphics cards. It supports both task parallelism and data parallelism, and is designed for algorithms that can be decomposed into passes. The provided API supports the user in structuring the program accordingly. Only application-specific parts need to be implemented using a set of base classes. Multiple compute kernel implementations can be provided per pass, one for each device class (e.g. CPU, GPU, CELL). The scheduler which is based on the critical path method determines prior to the actual computation which implementation to execute on which device to minimize the overall runtime by considering device speed, availability and transfer cost. This procedure has the additional advantage that data can already be transferred to a compute device before the actual need for it arises and thus network transfers can often be executed parallel to computation. Overall, this results in reduced device idling times (if any) and more efficient device utilization. Thread setup and communication, network data transfers and scheduling are handled transparently to the user. PaTraCo monitors the execution in order to update the cost estimates that are used by the scheduler and to provide the user with visual analysis. We evaluate the framework by means of an interactive distributed volume renderer.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Parallel processing—
I.3.2 [Computer Graphics]: Distributed/network graphics—

1. Introduction

There are numerous frameworks available for programming multi- and many-core computation devices, like CUDA for NVIDIA GPUs and OpenMP for CPUs. OpenCL is even designed for writing programs that execute on different types of devices. However, when a computation needs to be distributed across multiple compute devices, the programmer has to be aware of which and how many devices are available on a machine, in a cluster environment, or even in very heterogeneous environments. Data management, network communication and especially task scheduling, which should optimally consider data locality and compute device specifications amongst others, need to be implemented manually by the application program developer.

In order to support the programmer in developing and providing efficient execution scheduling, we introduce PaTraCo (**Pa**rallel **Tra**nsparent **Co**mputation). Computation tasks need to be divided into one or several passes using the provided API. Passes again are subdivided into pass blocks – each pass block typically operates on a different data chunk – which are scheduled to execute on a device in the cluster such that the estimated computation time over all pass blocks is minimal. This is achieved by an advanced load-balancing heuristic that exploits the information about the overall computation that is implicitly available to PaTraCo and additionally uses performance estimations which may be provided by the user. It enables to predictively copy data to nodes on which it will be needed later on, which is one major advantage over standard work stealing algorithms for instance. Our scheduling heuristic aims to choose the best hardware to optimally process a given workload. However, since the best hardware is not necessarily the hardware that processes the problem fastest, also the cost of transferring the input data to the device as well as the availability of the device are taken into account. Our system allows user-transparent execution of applications on heterogeneous, arbitrary linked

DOI: 10.2312/EGPGV/EGPGV10/131-140

nodes across several device types, steered by our scheduling heuristic for execution planning. Note that the scheduling problem discussed in this paper focusses on assigning task chunks to devices and not on alloting device time to processes (see Fallenbeck et al. [FPSF06] amongst others).

2. Related Work

To our knowledge no framework is publicly available that supports compute-network-transparent programming featuring different classes of devices and automatic scheduling. However, there are tools that enable device-transparent programming and execution of applications on machine level featuring multi-core processors, e.g. RapidMind [Mon08] and HMPP [DBShmcppe07]. When the computation is expressed as a sequence of functions applied to arrays, it is automatically divided among the available processing cores (CPUs, GPUs and CELL). These frameworks base on a runtime system that provides a uniform execution model, scheduling policies and automated data transfers, as discussed by Augonnet et al. [ATNW09] amongst others.

Müller et al. [MFS*09] presented CUDASA, a development environment that additionally supports cluster environments but limits itself to GPUs and CUDA. The underlying communication mechanisms are handled transparently and a data locality aware scheduling mechanism is used. Sunderam [Sun90] proposed a programming environment for parallel applications consisting of many interacting components which is intended to operate on a collection of heterogeneous computing elements. Apart from that, there are frameworks for grid batch computing like BOINC [And04] focusing on large scale computations (e.g. astrophysics). Besides these general frameworks, many application-specific environments for user transparent computation on clusters have been proposed, especially in the area of image and multimedia processing [BDF*93] [LWX02] [PTVvdS09]. Seinstra et al. [SGK*07] proposed a cluster programming library for implementing parallel multimedia applications as fully sequential programs using pre-defined primitives.

One fundamental idea of PaTraCo is that the distribution of parallel applications is based on the graph structure describing the data dependencies. Basic research in this area has been done by Diekmann [Die98] amongst others. He assumed that the vertices of this graph represent data elements and that the edges express the data dependencies. A distribution of the vertices across the available compute devices is achieved by partitioning the graph. Solving the graph partitioning problem while not only considering one but multiple classes of devices (e.g. CPUs and GPUs) is an important feature in our framework. Wang et al. [WzHCyZ08] proposed a simple task scheduling algorithm for single machine CPU-GPU environments that not just uses first idle compute device but chooses the fastest device from all idling devices. Teresco et al. [TFF05] worked on a distributed system in which every CPU requests tasks from the scheduler which are sized according to the device's measured performance score. Resource-aware distributed scheduling strategies for large-scale grid/cluster systems were proposed by Viswanathan et al. [VVR07]. There has also been a lot of research on application specific load balancing strategies. In particular, many volume visualization systems deal with this issue (e.g. Frank and Kaufman [FK09]). Zhou et al. [ZHR*09] propose a multi-GPU scheduling technique based on work stealing to support scalable rendering. In order to allow a seamless integration of load-balancing techniques into an application, object-oriented load-balancing libraries and frameworks were developed [DHB*00] [SZ02].

PaTraCo's scheduler is based on the critical path method, which is a mathematically founded algorithm that was originally developed for scheduling project activities [KW59]. An early application of the critical path scheduling to computation considering resource and processor constraints was presented by Lloyd [Llo82]. Kwok and Ahmad [KA96] discussed the mapping of a task graph to multiprocessors.

3. Programming PaTraCo

PaTraCo is designed to handle the distributed execution of a program on a hardware setup transparently, while the user only needs to implement application-specific functionality.

3.1. Programming Model

The programming model provided by the PaTraCo API is based on the notion of passes. A pass denotes a part of the overall computation of the program. A pass is subdivided into pass blocks such that one pass block can be executed on a device in one step. Pass blocks can have dependencies on any other pass blocks – from the same pass or a different pass – as long as no cycles are implied.

PaTraCo allows the user to provide different implementations for different device classes (CPU, GPU etc.) per pass. The scheduler determines which implementation to use, which mainly depends on the device setup. The hardware setup aware choice of code and execution paths allows for the efficient execution of a program in substantially different environments. Implementations of a pass may be provided using any desired compute API, e.g. CUDA, OpenCL, etc. for GPUs. Our framework only distinguishes between different classes of devices but does not contain specifics of any programming interface. Thus, device and API specific operations like data organization need to be implemented by the user. This is reasonable as the data structures which have to be uploaded by the user heavily depend on the implementation of the respective computation kernels. Usually, special input and output pass blocks are added to model the origin and the target for a computation. These pass blocks can be restricted for scheduling to a specific device to depict the location of the input and output data and letting PaTraCo automatically handle the necessary data transfers.

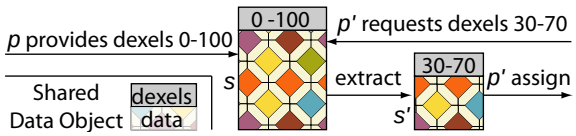

Figure 1: *Example to illustrate the concept of dexels. The outcome of a pass block p is provided as shared data object s and pass block p' requires parts of s (denoted as s') as input. p' defines in terms of dexels what data it needs from s and the corresponding data is extracted to a new shared data object s' which can then be assigned to p'.*

3.2. Programming Interface

The programming interface of PaTraCo can be subdivided into two categories. The first category mainly consists of the PaTraCo master class which is used to integrate PaTraCo in an application. To issue a computation task, the member function `run()` of this object needs to be called (e.g. once for the rendering of every frame for an interactive application). Parameters for this function are the input and output pass blocks as well as parameters used for pass subdivision.

The second category provides the interface to implement the computation passes that should be executed by PaTraCo. For every pass, the following functionality needs to be implemented by overriding the respective functions of the provided abstract base classes:

Passes: Subdivision of the pass into many small pass blocks.

Shared Data: Shared data objects provide a container for the data exchanged between pass blocks.

Pass Blocks: Provide the compute kernels that should be executed. Typically passes are subdivided in such a way that each pass block operates on a fraction of the data. Implementations for different devices may be provided for the same pass.

Required and provided data of pass blocks are specified in terms of dependency elements (called dexels in the following) and exchanged between pass blocks in the form of shared data objects. Dexels consist of a unique identification number and are used for the specification of data relations between pass blocks. Dexels abstract from actual data in a sense that they only denote a certain data chunk (see Figure 1). The mapping between data and dexels is explicitly done by the user in the shared data objects and in pass blocks. The flexible concept of dexels is important to cover the various complex data relations pass blocks might have.

4. PaTraCo Execution Procedure

An overview on the steps that PaTraCo executes is depicted in Figure 2. After starting up the thread infrastructure on each node *(0)(1)(2)*, the user's implementations of the passes

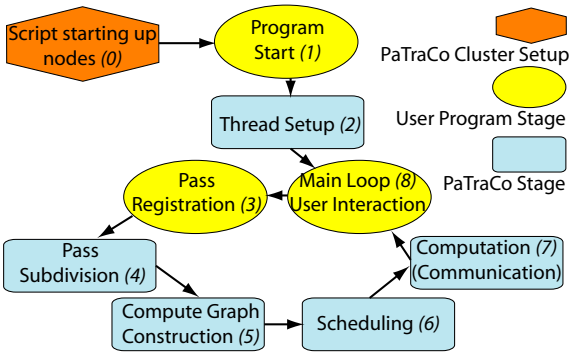

Figure 2: *PaTraCo computation stages using the example of an interactive application.*

that are needed for the next task are registered with the master class *(3)*. The registered passes are then divided into pass blocks with respect to the available compute devices and their available storage resources *(4)*. After that, a compute graph is built that contains the information which dexels a block requires from another block as well as estimates on the cost of processing a pass block with respect to the available device classes *(5)*. Using the compute graph, the scheduler assigns pass blocks to devices by employing a load-balancing heuristic based on critical paths *(6)*. Finally, the computations (and required communication) are executed *(7)*. Depending on the dynamics of an interactive application *(8)*, steps *(3)* to *(7)* need to be executed only once, occasionally, or for every frame. Note that all steps from pass registration to scheduling operate on the same pre-computation input data on all nodes (local node information is automatically distributed like updated device performance data) and execute exactly the same computation.

4.1. Setup and Thread Infrastructure

Before the actual computation can commence, the user's program is started on all nodes that take part in the computation. This is accomplished using a Python startup script that evaluates an XML file which contains the description of all nodes. The same file is also used for building the compute resource graph. For all nodes, the XML file contains the information that is required for the invocation of the program: the host name, the login, and the path to the binary that should be executed. We focused on enabling the ad-hoc construction of a compute network even for very inhomogenous environments. Note that this initial step does not invoke PaTraCo directly.

In order to use PaTraCo, the user program eventually has to create one instance of the PaTraCo master class on every node. Its constructor sets up the computing infrastructure consisting of threads and communication objects as illustrated in Figure 3. It spawns a new thread for each available

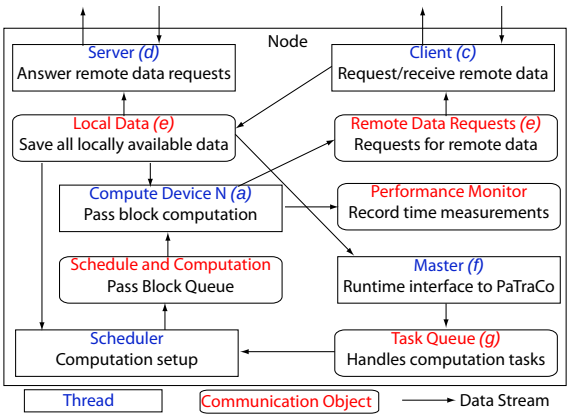

Figure 3: *Data stream (indicated by arrows) between threads and communication objects on a node.*

compute device according to the nodes XML file, which is also used to build the resource graph describing the compute network. Furthermore, a client thread is started that requests and receives data from remote nodes and a server thread is spawned to answer data requests from other nodes.

The threads (Figure 3, *blue*) on a node communicate and exchange data using thread communication objects *(red)*. Before processing the pass blocks, a thread belonging to a compute device *(a)* adds requests to the remote request queue *(b)* for data elements that are required from pass blocks that are scheduled to execute on other nodes. This queue is processed by the client thread *(c)* on the same node which requests and receives data from remote nodes. These data requests are answered by a server thread *(d)* on the respective node, which has access to all local data *(e)*. As soon as the client receives the requested data, it is stored in the local data pool object *(e)*. This object is also used for accessing results that were computed locally. Thus data can be fetched transparently. As soon as all dependencies are satisfied for a pass block, the data is assigned to the respective device and the computation is executed. The result is then stored in form of a shared data object in the local data communication object from which it can be accessed by other threads *(blue)*. The overall computation is finished when the final result data is available to the master thread *(f)* as requested in the invocation by the user program. The computation tasks that need be processed are added to a task queue *(g)* by invoking `run()` in the PaTraCo master class *(f)*. For interactive programs, the queue should maximally contain one element while for offline batch jobs it is used to store all computation tasks of the user's program.

4.2. Pass Subdivision

For every pass, pass blocks are generated with respect to user-defined input data and characteristics of available compute devices. Note that generated pass blocks which are not needed to compute the final result are simply ignored by the scheduler. For subdivision, the member function `determineBlocks()` of `PaTraCoPass` needs to be implemented by the programmer (Listing 1). For optimization purposes, an internal member variable can be used to declare dexel blocks to be stable, if it can be determined using *inputData* that these dexels will not change in this task with respect to the previous task. This indicates to the shared local data object that this data should not be deleted. This can significantly reduce transfer costs and allows the scheduler to skip pass blocks which produce the respective data.

```
class PaTraCoPass {
  virtual void
  determineBlocks(PaTraCoPassBlocks* partitionedBlocks,
                  std::vector<PaTraCoPass*>* prevPasses,
                  PaTraCoSharedData* inputData) = 0;
};
```

Listing 1: *Implementation framework for pass subdivision. Arguments like data sizes and other parameters are provided in inputData and the function has access to all previous pass subdivisions, which helps avoiding redundant subdivision computations for passes with similar partitioning*

4.3. Constructing the Compute Graph

A compute graph is a directed acyclic graph (DAG) in which both the edges and the vertices are weighted (for an example see Figure 4, right). The vertices stand for pass blocks and feature one weight for each device class implementation which is available for the respective pass. This weight is made up of measurements and pass block characteristics, which are specified by the user in pass subdivision (see Section 4.4 for details). Edge weights denote the amount of data that has to be transferred between pass blocks based on the required dexels and their weights as again indicated by the user in pass subdivision (e.g. for a dexel standing for a pixel in a float texture, its weight is the size of a float in bytes).

The compute graph is constructed back-to-front from the pass blocks that provide the result data to the pass blocks that contain the input data. For every visited pass block, edges are created to the vertices (pass blocks) that provide required data and subsequently these are visited. Subsequently, pass blocks which are not required to compute the final result are removed. After that, the resulting graph is checked for cycles to ensure a seamless execution and provide the programmer with details about potential problems.

4.4. Scheduler

In our context, the problem that needs to be solved by the scheduler is the assignment of pass blocks to compute devices considering the compute device speed for the given

pass block, the implied transfer costs, and the availability of the device. The scheduler is a critical component in PaTraCo. It needs to run fast, especially when it is used in the context of highly dynamic, interactive applications that require rescheduling for each frame. However, it also needs to deliver a good solution because the performance of the subsequent computations heavily relies on a good schedule.

Finding an optimal distribution would be very time consuming and not efficient in general, which is why we chose to use a heuristic that is much faster to compute. Our iterative algorithm can even be run with a variable amount of iterations per critical path depending on existing time constraints. The basic idea is to iteratively identify the longest, critical and thus the most time consuming path through the graph until all vertices (pass blocks) are assigned to a compute device. A critical path contains those computations that, when delayed, lead to a later completion of the overall task described by all vertices in the graph. The sooner a pass block is identified as part of a critical path of the remaining graph, the earlier it may allocate the most suitable device for a certain time span.

For determining the longest (critical) paths, the scheduler uses a strongly modified version of the Bellman-Ford algorithm [Bel58]. Bellman-Ford can compute shortest as well as longest paths in a weighted directed graph with running time $O(|V||E|)$ (compared to $O(|E|+|V|log|V|)$ of Dijkstra's algorithm [Dij59] which can only compute shortest paths) with $|V|$ being the number of vertices and $|E|$ denoting the number of edges. We need to extend this algorithm as our scheduler not only needs to consider vertex, but also edge weights both of which depend on the chosen compute device for a vertex. More importantly, edge and vertex weights vary with the compute devices that are assigned to the vertices.

The scheduler is finished when a compute device has been assigned to every vertex, which means that the set of edges that are connected to an open vertex (meaning that no compute device is assigned to it) is empty (Listing 2). Until then, in every iteration the longest critical path is searched considering previously found paths and the remaining set of edges which at least connect to one open vertex. Similar to Bellman-Ford, for every remaining edge we test whether this edge would be part of a critical path passing through the target vertex. If it does, the compute device is chosen that leads to the shortest critical path considering the suitability of the device for the task block, the device availability, and transfer cost from and to the device. Internally, these parameters are computed, combined and evaluated in microseconds. The vertex weight is determined by the product of the average time consumption of task blocks belonging to their respective pass (measured), the relative complexity of the current pass block with respect to other pass blocks in the same pass, general device class suitability for the given pass (both assigned by the user in pass subdivision), and device

```
while not edges.empty()
  for v in vertices
    length[v] = 0, next[v] = none, device[v] = none
  cost.clear()
  for nIterations
    for e in edges with e.target in vertices
      # update cost array , see definition in Listing 3
      updateVicinityCost(cost[e.target], e, next[e.target],
                         device[e.source], device[next[e.target]],
                         length[e.source])
      # cost for the best device choice considering all edges
      # (transfer cost to next node not contained in return value)
      bestWorstCaseCostNoOut = bestWorstCaseNoOut(cost[e.target])
      if length[e.source] + bestWorstCaseNoOut > length[e.target] ||
        next[e.source] = e.target # always update current path
        length[e.target] = length[e.source] +
                           bestWorstCaseNoOut(cost[e.target])
        next[e.source] = e.target
        # return device with the best worst case performance
        # with respect to the direct vicinity of the vertex
        device[e.target] = bestWorstCaseDevice(cost[e.target])
  cricalPathEndVertex = maxValIndex(length[], vertices)
  criticalPathVertices[] = traverseInv(criticalPathEndVertex, next[])
  for v in criticalPathVertices, inverse order # from input to output
    # reserve best device for edge, so that the resulting worse case
    # does not takes longer compared to the lowest possible worst
    # case by more than the factor allowWorseFactor (we chose 0.2)
    reserveDevice(bestWorstCaseDevice(cost[v], inverse(next[])[v],
                  allowWorseFactor), length[v])
    remove(edges, (v, next[v]))
    remove(vertices, v)
```

Listing 2: *Simplified scheduling procedure. Apply devices to the critical paths first, but still do not completely ignore other crossing paths. The multidimensional array* `cost` *stores the induced local cost (in- and outgoing connection and computation cost) of all combinations of incoming edges and compute devices for each vertex.* `nIterations` *can be chosen by the user to adapt to performance constraints. The array* `length` *saves for each vertex the accumulated cost of the longest path from a source to this vertex.*

speed relative to other devices of its class (from the resource graph). Besides, when considering the use of a device, its availability is taken into account by adding the wait time until it can be used according to its schedule when the device is already busy for the requested time frame. In this context, it is also considered that no pass block may be planned for execution before any pass block it depends on (Listing 3). The edge weight is calculated by multiplying the transfer speed between the two devices as depicted by the resource graph (evaluated using Dijkstra's algorithm) with the size in bytes of the required dexel block (implicitly specified by the user in pass subdivision). At the end of each iteration, a critical path is identified and compute devices are allocated.

Note that in our current implementation we chose redun-

```
def updateVicinityCost(costPerVertex, e, nextVertex,
                       prevDevice, nextDevice, from)
  for d in deviceGraph
    wait = d.availableFrom(from, dependencies[e.target]) − from
    lengthConnection = length(prevDevice, d, e.source, e.target)
                       + length(d, nextDevice, e.target, nextVertex)
    costPerVertex[d][e.source] = weight(d, e.target) +
                                 wait + lengthConnection
```

Listing 3: *Simplified algorithm to determine the path weight around the considered vertex for all devices (if no implementation for a given device is available, its weight for the vertex is infinity). The function length returns zero, when required data is already available on a device/node from previous tasks. Otherwise it returns the connection weight, which is determined using the resource graph, multiplied by the edge weight (the transferred data size).*

dant scheduling on all nodes over distributing the schedule with associated information like dependencies over the network in order to save bandwidth. However, if nodes are involved which only have a very slow CPU (but potentially a fast GPU for example), transferring the schedule over the network is the better alternative.

4.5. Communication

The communication between threads can be classified as local or remote, depending on whether they run on the same node or not. Local communication includes the invocation of a computation by the insertion of a task into the task queue by the master thread, assigning this task to the scheduler and providing the device threads with their respective schedule. The data exchange between local device threads is also included. Local communication simply works asynchronously using objects which are shared by threads and guarded with mutexes. Remote communication is necessary for exchanging computation results between device threads on different nodes and providing input data for pass subdivision and scheduling. For this, we implemented a client and a server which are based on sockets and run in separate processes decoupled from the device threads. When there are requests in the client queue, it polls the respective remote server until the data is available, and then receives and saves it in the local shared result data pool.

The shared data pool stores all data that is generated on or transferred to a node and it is deleted automatically when it is not needed anymore. Note that it not only stores data that is available in main memory but also data in device memory. Data is kept in the pool until after the pass subdivision of the subsequent task data determinded which dexels are considered to be stable. All data belonging to non-stable dexels are deleted. Should there be not enough (device) memory available for storage, then the data, which according to the schedule is not needed for the longest time, is downloaded to one storage level below (from device memory to main memory, from main memory to disk) and deleted from the current storage level.

We use sockets for the remote communication. In contrast to higher level APIs like MPI (Message Passing Interface), sockets have no restrictions concerning the construction of an arbitrary compute network. This design choice is important for the flexibility to run applications in very heterogeneous environments on only loosely coupled nodes.

The data exchange between pass blocks, independent whether they are scheduled to execute on the same device, different devices or even different nodes is done using `PaTraCoSharedData` (Listing 4) objects. The result data of a computation must be saved by the programmer in an object whose class is derived from `PaTraCoSharedData`. Three functions need to be implemented here to enable data exchange. `extract` generates a new shared data object with a subset of the data of the original object. This is triggered for every different shared data request that needs to be satisfied by this data object – this is inherently given by the pass subdivision – and saved in the shared data object pool. Note that this does not necessarily mean that the underlying data structure is copied to another location. For many applications saving a pointer to an array and an offset variable is enough. In the case of reusing the same memory chunk, the programmer has to increment and decrement a reference counter manually to prevent a too early destructor call by the local data handler. This highly depends on the application and the programmer has to take care of an efficient implementation. The same is true for both the `serialize` and the `deserialize` functions which are exclusively needed for remote data transfers. The `serialize` call is triggered by the server process for data objects which were requested by another node and the result is stored in the local data communication object for further use. After the transfer, serialized data is stored in the local data communication object as well and `deserialize` is called on demand by the device thread that requires the data.

```
class PaTraCoSharedData {
  virtual void serialize(serialized_t& data) = 0;
  virtual bool deserialize(const serialized_t& data) = 0;
  virtual void extract(PaTraCoSharedData*& sharedData,
                       PaTraCoDexelBlock* dexelBlockRequest,
                       computeDevice_t deviceClass,
                       int localId, bool onDevice) = 0;
};
```

Listing 4: *Shared Data implementation. The parameters for `extract` determine which data to extract from what kind of device, that is uniquely identified by its local id and further it is specified for devices with dedicated memory whether the data stays on the device or is downloaded to main memory.*

4.6. Computation

In the computation step, the device threads process the pass blocks as scheduled. A pass block is an object of a class derived from `PaTraCoPassBlock` (Listing 5), whose functions for assigning data and computation need to be implemented by the programmer. Note that there are many empty functions provided, one for every class of devices like `computeCPU`, `computeGPU`, `computeCELL` and so on. At runtime, the scheduler checks which ones are implemented by the user (using `testonly=true`) and considers them for execution.

```
class PaTraCoPassBlock {
  virtual bool
  assignCPU(int localDeviceId,PaTraCoSharedData* sharedData,
            PaTraCoDexelBlock* dexelBlock, bool onDevice,
            bool testonly=false)
            {return false;}
  virtual bool
  assignGPU(int localDeviceId,PaTraCoSharedData* sharedData,
            PaTraCoDexelBlock* dexelBlock, bool onDevice,
            bool testonly=false)
            {return false;}

  virtual bool computeCPU(int localDeviceId,bool testonly=false)
  {return false;}
  virtual bool computeGPU(int localDeviceId,bool testonly=false)
  {return false;}
};
```

Listing 5: *For providing implementations for a device class, the respective `assign()` and `compute()` functions need to be overwritten. `testonly` is used internally for checking which implementations are available. Due to that, the first line of the actual function definition should be a small macro provided by PaTraCo that returns true if the testonly is true and otherwise continues the function.*

However, before actually computing a pass block, the device thread has to wait until all required dexels are available in the local data communication object. Then, the input data has to be provided in a structure that is required by the computation. This is done by calling the user-defined `assign` function for every block of dexels. The function is device-class-specific, because fundamentally different data structures might be needed (e.g. for a GPU, data structures might need to be serialized and uploaded to the graphics card as textures). If required, input data from main memory is uploaded to device memory using this function. The programmer can create a new shared data object containing the uploaded data, so that it can be accessed by other pass blocks.

Note that the order in which the pass blocks are executed is not finally determined by the scheduler. The pass blocks are executed out of order when the required data is not entirely available for the current block, but for any other pass block in the schedule. After completing its computation, it is checked again in the order determined by the scheduler which is the first pass block in the schedule that can be executed without waiting for results of other devices. Pass block computation results need to be saved such that they can be fetched from requiring subsequent pass blocks directly using shared data objects as discussed in section 4.5. For compute devices with dedicated memory, the data is saved in the same way as data located in main memory. It is downloaded automatically (using the Shared Data `extract()` function) only if requested by another device, or if the current device requires more free memory for other computations.

4.7. Monitoring

Monitoring the performance of all stages of PaTraCo – especially during computation and communication – is very important. First, it is useful for the programmer to be able to analyze the resulting performance as this can give hints, how to improve the program by reducing bottlenecks. After the computation, the monitoring results can be visualized by our monitoring frontend that is based on the 2D graphics library Cairo (see Figure 5). Second, the program requires these for self-calibration by incorporating timing measurements to update cost estimates. Remember that scheduling is based on the resource graph, which contains assumptions of computation speeds of devices relative to different passes. It is crucial for these estimates to be as good as possible to achieve a near-optimal schedule.

5. Results

We implemented an interactive distributed volume renderer incorporating shadow volumes for evaluating our framework, in particular our scheduler. The user interacts with a node called frontend, which displays the rendering result and provides the light position, the view matrix, and the volumetric data set. Our framework was tested in small networks of differently equipped nodes.

5.1. The Volume Rendering Application

Our distributed volume renderer employs object-space data distribution (like e.g. [MSE06]) and uses shadow volumes for determining the illumination contribution of an omnidirectional point light source L with a limited radius (Figure 4). This example application consists of three passes: generation of the shadow volume, volume rendering and compositing. The first pass generates a shadow volume for all volume bricks within the radius of the light source by naively sending a ray for each voxel towards the light source (for a more sophisticated algorithm see Hadwiger et al. [HKSB06]). For all adjacent volume bricks, where a ray exits is originating brick on its way to the light source, a one voxel thin shadow volume layer needs to be available for light value contribution lookup. This results in an in-pass dependency structure. The second pass does standard volume

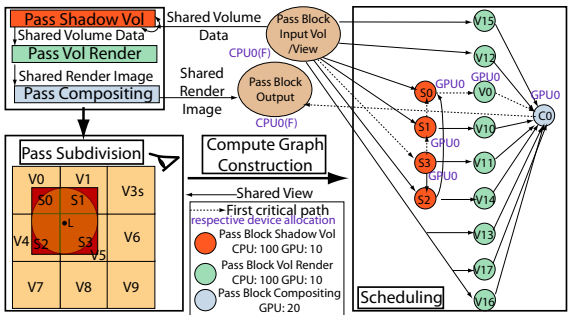

Figure 4: *Chart from pass subdivision to scheduling for the distributed volume rendering example. On the upper left, passes and input/output pass blocks are shown with their data dependency relations. After subdividing the passes and assigning a volume brick to each pass block (bottom left), the compute graph is constructed that includes the data dependencies between the pass blocks (bottom right). The scheduler determines critical passes and assigns devices to pass blocks by using this graph in conjunction with the resource graph and timing measures.*

raycasting and employs the shadow volume for lighting each sample point. The third and final pass takes the renderings of all bricks generated in the previous pass and combines them into the final image. For both shadow volume generation and volume rendering CPU and GPU implementations using CUDA are provided. Compositing is implemented on the GPU only using CUDA as well.

The effort to compute a volume rendering pass block depends on the view matrix (i.e. camera position and orientation) and the size and position of the associated volume brick as well as the screen resolution. Using this input data, the relative cost of rendering a brick is estimated in the pass block generation by computing the rendering size of the brick on the screen. In this case of an interactive application, in contrast to programs which run computations only once, initial costs (here for the first frame) like static volume data distribution might be negligible, when in return the overall computation runs faster in all subsequent frames. Hence we set the weight of the incoming dexel blocks to zero in the pass subdivision of the first frame and thus enable data distribution which better reflects the rendering speed provided by a node. From the second frame on, the input dexel block weights are set normally, in order to allow further volume data transfers only in the case of high load imbalance.

5.2. Evaluation

All nodes we used in the evaluation were connected with Gigabit Ethernet, and equipped with a Intel Core2 Quad 2.4 GHz CPU. They featured three different kinds of GeForce graphics cards: 8600 GT, 8800 GTX and GTX 280. Our frontend node used a 8800 GTX. In our scenario, we rendered a 1600×1024 frame using 16-bit 512^3 data set, which was split into 16 bricks by the the scheduler, so that each brick has the dimension of $128 \times 256 \times 256$. The respective measurements are shown in Figure 5. The timings include all steps from providing input data until the output data is available. However, they do not include scheduling, which took between 5 – 30 ms, depending on the amount of devices given and the number of iterations specified. In our testing scenario, three iterations per critical path were already enough to achieve a good schedule. Even though CPU implementations are available for shadow volume generation and volume rendering, the computation ran on the GPU exclusively due to its vastly superior performance in this context. Using the CPU for the execution of a pass block would have significantly slowed down the overall computation with our hardware setup (the CPU was slower by a factor of ≈ 100 in our implementation than a 8800 GTX). However, note that the relative performance might substantially differ using different setups or slightly different tasks, so in general it is advantageous when a large range of compute devices can be used potentially. Nevertheless, this case shows the importance of considering device speed in the context of the overall computation and not simply using an available device.

In our first series of measurements, no shadow volume is generated because we moved the light source out of the volume and gave it a tiny radius. In the first test, we used two identical machines each equipped with a GeForce 8800 GTX besides the frontend node (Figure 5, top left). It can be seen that the scheduler efficiently assigns task blocks considering the costs and thus a good scaling factor can be achieved. In the heterogeneous setup using a node with a 8600 GT and a GTX 280 (bottom) left, the adaptation of the scheduler to different device speeds can be seen. Most importantly, the node featuring the 8600 GT is not assigned more than two pass blocks, even though the device and further tasks are available because this would result in an overall slowdown. For the second series of measurements, we moved the light source inside the volume and gave it a radius such that four volume bricks are covered. The primary critical path was determined by the scheduler to consist of pass blocks with the following ids: 22 (input) – 0 (shadow) – 1 (shadow) – 3 (shadow) – 14 (render) – 20 (composit) – 23 (output). Figure 5 (top middle) shows that this critical path dominates the total computation time. The volume rendering pass blocks 5 and 7 are also scheduled on the frontend GPU, because they require the shadow volumes generated by the shadow passes 0 and 1 for lighting and the transfer over the network of a shadow volume brick is too expensive. The allocation of other devices for these blocks would have substantially increased the total computation time, even though the other devices idle otherwise. When running the application with a node featuring a 8600 GT and a node featuring a GTX 280 besides the frontend node, it can be seen that the scheduler exploits the fast GTX 280 for execution of the primary

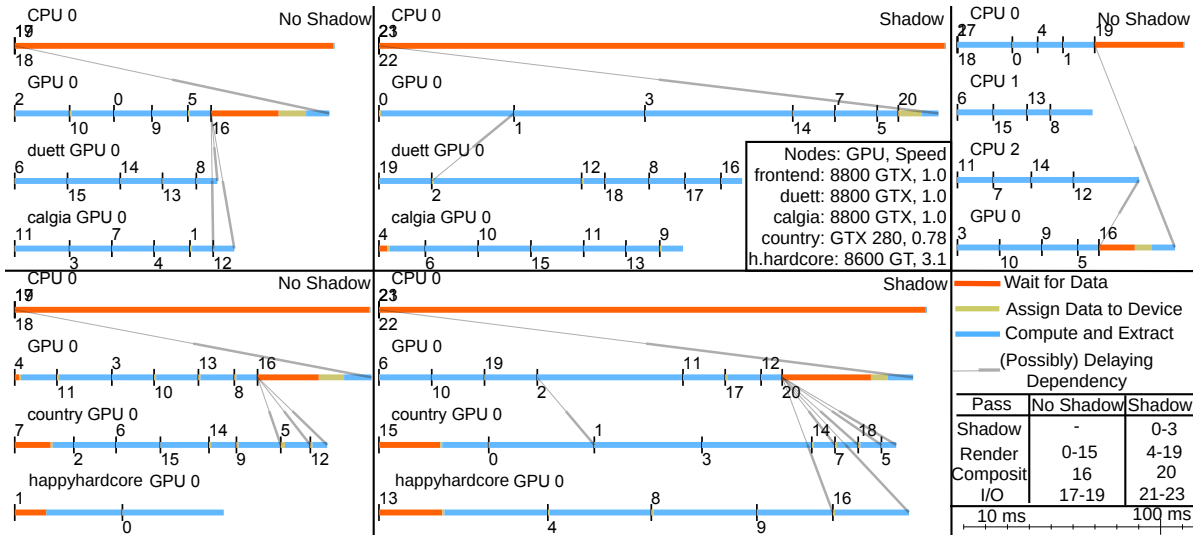

Figure 5: *Five performance monitoring outputs generated by PaTraCo using a timeline for each device that computes at least one pass block. Especially timelines from different nodes can be shifted to each other as computations do not start exactly at the same time. The black bars depict the beginning of a pass block with its identification number. Devices Ids without host information belong to the frontend node. Pass block ids can be matched with passes using the bottom right table.* `wait` *depicts the time until all data is available to start computing a pass block,* `assign` *the time to prepare the data for computation (e.g. GPU upload) and* `compute` *the time for pass block computation and result extraction (e.g. GPU download).*

critical path (Figure 5, bottom middle). The overall computation is not significantly faster compared to the previous case though, because the 8600 GT is significantly slower than the 8800 GTX (the system measured a factor of 3.1) and thus a lot of work has to be done by the other graphics devices. The advantage of pre-copying data also manifests itself in the shadow volume measurements: pass block 2 requires data from pass block 1 and pass block 3 requires data from pass block 2, but there is no waiting time involved even though volumes are processed on different nodes, because result data is sent immediately afterwards. For the last measurement series we again ignored the shadow volume, but this time decreased the CPU cost for volume rendering artificially by sending less rays through the volume. We used the GPU and three CPU cores on one node. The results show the advantage of systems featuring many compute devices locally as opposed to distributed system which cause a lot of network traffic, even though our framework allows to hide latencies by predictively copying data in parallel to ongoing computations (Figure 5, top right).

6. Conclusion and Future Work

We presented a framework for the development of parallel, multi-pass applications for heterogeneous compute environments. The user only has to implement application-specific parts using base classes provided by PaTraCo and to registers them with the framework. Before the actual computation, a device schedule is generated using our critical path heuristic. It takes device availability, suitability and data transfer costs into account for achieving a fast overall computation. Another advantage of the available schedule is that data can already be transferred to a given device before it is actually required thus increasing computation efficiency. PaTraCo is designed to support any API, device dependent or not. Scheduling, thread setup and handling, as well as network communication and data transfer are handled completely transparent to the user. A further benefit of PaTraCo is that it supports the programmer in the development process by providing a basic structure for the application.

For future work, we plan to make the scheduling more flexible during the computation through a work stealing mechanism. In order to enable changes in device assignment, the current architecture could be extended such that the node on which the originally assigned device is located is informed about the changes which would allow a rerouting of requests. We also want to add the possibility to transfer precomputed device schedules to nodes with weak CPUs. Additionally network interconnects should be treated like compute devices, with the ability to schedule them in order to take bandwidth occupancy into account. Additionally, we work on developing a graphical user interface that supports the user in implementing and connecting passes.Furthermore, we want to compare our scheduling algorithm to other scheduling strategies (i.e. first-come-first-serve) in a wide range of computing scenarios. Finally, we plan to add support for interleaved task execution.

References

[And04] ANDERSON D. P.: Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing* (2004), pp. 4–10.

[ATNW09] AUGONNET C., THIBAULT S., NAMYST R., WACRENIER P.-A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science* (Delft, The Netherlands, Aug. 2009), vol. 5704 of *Lecture Notes in Computer Science*, Springer, pp. 863–874.

[BDF*93] BAKER R., DOWNING A., FINN K., RENNISON E., KIM D. D., LIM Y. H.: Multimedia processing model for a distributed multimedia i/o system. In *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video* (London, UK, 1993), Springer-Verlag, pp. 164–175.

[Bel58] BELLMAN R.: On a routing problem. *Quarterly of Applied Mathematics 16*, 1 (1958), 87–90.

[DBShmcppe07] DOLBEAU R., BIHAN, S. B., HYBRID MULTICORE PARALLEL PROGRAMMING ENVIRONMENT. F. H. A.: A hybrid multi-core parallel programming environment. *First Workshop on General Purpose Processing on Graphics Processing Unit.* (2007).

[DHB*00] DEVINE K., HENDRICKSON B., BOMAN E., JOHN M. S., VAUGHAN C.: Design of dynamic load-balancing tools for parallel applications. In *Proc. Intl. Conf. on Supercomputing* (Santa Fe, New Mexico, 2000), pp. 110–118.

[Die98] DIEKMANN R.: *Load Balancing Strategies for Data Parallel Applications*. PhD thesis, Universität Paderborn, 1998.

[Dij59] DIJKSTRA E. W.: A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.

[FK09] FRANK S., KAUFMAN A.: Dependency graph approach to load balancing distributed volume visualization. *The Visual Computer 25*, 4 (2009), 325–337.

[FPSF06] FALLENBECK N., PICHT H.-J., SMITH M., FREISLEBEN B.: Xen and the art of cluster scheduling. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing* (Washington, DC, USA, 2006), IEEE Computer Society, p. 4.

[HKSB06] HADWIGER M., KRATZ A., SIGG C., BÜHLER K.: GPU-accelerated deep shadow maps for direct volume rendering. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (New York, NY, USA, 2006), ACM, pp. 49–52.

[KA96] KWOK Y.-K., AHMAD I.: Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 7*, 5 (1996), 506–521.

[KW59] KELLEY JR J. E., WALKER M. R.: Critical-path planning and scheduling. In *IRE-AIEE-ACM '59 (Eastern): Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference* (New York, NY, USA, 1959), ACM, pp. 160–173.

[Llo82] LLOYD E. L.: Critical path scheduling with resource and processor constraints. *J. ACM 29*, 3 (1982), 781–811.

[LWX02] LI Z., WANG C., XU R.: Task allocation for distributed multimedia processing on wirelessly networked handheld devices. *Parallel and Distributed Processing Symposium, International 1* (2002), 0079.

[MFS*09] MÜLLER C., FREY S., STRENGERT M., DACHSBACHER C., ERTL T.: A compute unified system architecture for graphics clusters incorporating data locality. *IEEE Transactions on Visualization and Computer Graphics 15*, 4 (2009), 605–617.

[Mon08] MONTENEY M.: Rapidmind multi-core development platform, Feb 2008.

[MSE06] MÜLLER C., STRENGERT M., ERTL T.: Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *Eurographics Symposium on Parallel Graphics and Visualization* (2006), Eurographics Association, pp. 59–66.

[PTVvdS09] PARK H., TURAGA D. S., VERSCHEURE O., VAN DER SCHAAR M.: A framework for distributed multimedia stream mining systems using coalition-based foresighted strategies. In *ICASSP '09: Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1585–1588.

[SGK*07] SEINSTRA F. J., GEUSEBROEK J.-M., KOELMA D., SNOEK C. G., WORRING M., SMEULDERS A. W.: High-performance distributed video content analysis with parallel-horus. *IEEE MultiMedia 14*, 4 (2007), 64–75.

[Sun90] SUNDERAM V. S.: Vm: A framework for parallel distributed computing. *Concurrency: Practice and Experience 2*, 4 (1990), 315–337.

[SZ02] STANKOVIC N., ZHANG K.: A distributed parallel programming framework. *IEEE Trans. Softw. Eng. 28*, 5 (2002), 478–493.

[TFF05] TERESCO J. D., FAIK J., FLAHERTY J. E.: Resource-aware scientific computation on a heterogeneous cluster. *Computing in Science and Engineering 7*, 2 (2005), 40–50.

[VVR07] VISWANATHAN S., VEERAVALLI B., ROBERTAZZI T. G.: Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems. *IEEE Trans. Parallel Distrib. Syst. 18*, 10 (2007), 1450–1461.

[WzHCyZ08] WANG L., ZHONG HUANG Y., CHEN X., YAN ZHANG C.: Task Scheduling of Parallel Processing in CPU-GPU Collaborative Environment. *Computer Science and Information Technology, International Conference on 0* (2008), 228–232.

[ZHR*09] ZHOU K., HOU Q., REN Z., GONG M., SUN X., GUO B.: Renderants: interactive reyes rendering on gpus. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers* (New York, NY, USA, 2009), ACM, pp. 155:1–11.