A Case Study on Multi-Component Multi-Cluster Interaction with an AMR Solver

Atanas Atanasov
Department of Informatics
Technische Universität München
85748 Garching, GERMANY
Email: atanasoa@in.tum.de

Hans-Joachim Bungartz
Department of Informatics
Technische Universität München
85748 Garching, GERMANY
Email: bungartz@in.tum.de

Kristof Unterweger
Department of Informatics
Technische Universität München
85748 Garching, GERMANY
Email: unterweg@in.tum.de

Tobias Weinzierl
Department of Informatics
Technische Universität München
85748 Garching, GERMANY
Email: weinzier@in.tum.de

Roland Wittmann
Department of Informatics
Technische Universität München
85748 Garching, GERMANY
Email: wittmanr@in.tum.de

***Abstract*—The present paper studies an assembly of parallel applications—a controller, a dynamic adaptive Cartesian mesh simulation code for hyperbolic partial differential equations, and a multi-device visualisation server—where the codes interact via procedure calls formalised by an interface definition language. These calls are augmented by target device identifiers. Answering nodes hence are efficiently able to route their data flows through the multi-cluster environment. We study a strategy where answers from multiple nodes are merged dynamically along a virtual tree either on the sender or receiver side. Dynamic merging allows the data flow to adjust to the communication characteristics of the codes, their load balancing, and the hardware topology. The communication between simulation and postprocessing components is handled transparently through well-defined interfaces that guide a code generator. The generated communication code, the data scattering and the data reduction are hidden from the applications and its programmers. The assembly realises concepts of the common component architecture. Results from our case study show reasonable usage of available communication facilities. Hence the present work describes a mature architectural pattern for interactive parallel computing and non-holistic high performance software assemblies.**

***Keywords*—On-demand data exploration, blackbox AMR, coupling, near-data processing, parallel data exchange, parallel components**

I. Introduction

Next generation scientific computing will emancipate the simulation workflows from the one-application-at-a-time paradigm and make them rely on multiple concurrently running applications. Computational steering with its on-the-fly visualisation and change of parameters pioneered here and still remains an important challenge. Other examples are embedded simulation codes interacting with real-time measurements, or simply non-monolithic applications assembled from clearly separated components. The increase of application concurrency faces an increase of hardware concurrency: multiple applications running on multiple cores on multiple machines or machine parts interact.

In the sketched application landscape, communication between two codes includes multiple data sources (on N nodes), request sources (on M nodes) and multiple destinations or addressees (on $\hat{M}$ nodes). A single-point-of-contact realisation—making all requests pass through one node, distribute requests, gather results, and redistributing them again—is not an option: It requires significant temporary memory conflicting with the stagnating memory per core [1], it requires bandwidth and communication resources for the gathering and scattering on both communication partners already suffering from low bandwidth and high latency [2], [3], it requires high IO throughput on the distinguished application communication nodes while many architectures are already IO bound [3]–[6], and it finally induces a bulk synchronisation among the exchanging applications. Moreover, asynchronism of the individual cores—some deliver data faster than others—is not exploited. Thus, a $N : M$ communication scheme is desired. From a programmer's point of view it is however often preferable to interpret a code as an atomic building block with a well-defined single-point interface hiding the application's parallelisation.

In data postprocessing, visualisation, and in particular computational steering, the safe-load-cycle with single synchronisation points already has been rendered old fashioned and replaced by workflows instantaneously postprocessing distributed results [5], [7]–[10]—either in-situ or out-of-core. The recent message passing interface (MPI) [11] provides non-blocking collective operations and its implementations come along with sophisticated and effective routing. The common component architecture (CCA) [12], [13] allows for direct-connected and distributed frameworks. Direct-connected frameworks as described in [14] make each node hold the whole functionality of the application, i.e. components and their connections. The set of same components running on different nodes defines a group called a cohort. Outbound interactions between a cohort and other components are defined by CCA and can be realised by function calls. The interaction inside the cohort remains undefined by the standard. Such kind of setups are not well-suited for non-monolithic applications combining heterogeneous parallel components running on different

clusters (e.g. , parallel simulation and parallel visualisation). Distributed frameworks preserve the one-process-per-node-per-application splitting (MPMD) and have to maintain mappings from distributed arrays to other distributed arrays, i.e. global routing tables. Linearisation, native distributed arrays, and distributed array descriptions, i.e. meta data exchange, for $N : M$ are discussed, e.g., in [14] and references therein. Also CORBA provides extensions for parallel objects similar to CCA by augmenting the interface definition language [15]. The present paper's use case architecture falls into the latter category and tackles the tedious shared data layout, data redistribution, and consistency challenge. It introduces one particular realisation pattern for $N : M$ communication challenge that avoids shared memory data structures and relies exclusively on method calls. We assume that the addressee's data ownership pattern in general is unknown to the method caller (and irrelevant as it changes anyway), while the caller knows how the result data is to be scattered or replicated among on the receiver side. Passing this meta information, the called component can, in collaboration with the receiver, distribute incoming data and assemble answers internally. It acts as the single instance to the caller.

For our case study, we start from a classical computational steering scenario in a high performance computing environment where multiple steering nodes ask multiple compute nodes solving a hyperbolic partial differential equation (PDE) for particular solution data which then is to be sent to multiple visualisation nodes. The work extends [9], [10] with a running simulation code rather than a static data source, and this code realises dynamic load balancing. It changes data ownership frequently. Different to data-driven approaches, our steering nodes explicitly ask for data and at the same time label the requests with destination addresses, i.e. tell the addressee where to send the results to. It is an on-demand data exchange not driven by data updates. Different to publish-subscribe approaches, our data source nodes know who accepts the data and which records are of interest. Filtering is avoided and publish-subscribe relations are volatile. The domain decomposition of the PDE solver, i.e. the information which node can answer particular requests concerning a particular spatial data, is hidden from the requesting nodes. Requests are routed through the PDE solver instances organised in a (virtual) spatial tree describing the decomposition of the computational domain. Each compute node contributing with subanswers sends its results back asynchronously. Prior to feeding the results into the rendering pipeline of the visualisation nodes, they have to be merged into one answer per receiving instance. These merges are split up into two stages distributed among both the data source nodes and the receivers. The distribution of the merges takes into account the solver's domain decomposition, the tree topology mentioned before, the available bandwidth, and the receiver's address. The reduction is data flow- and IO-aware. All communication is hidden behind generated code that stems from a simple interface description realising a subset of the Scientific Interface Description Language (SIDL) [13]. Technical communication details of the components are not visible to the programmer.

As a result, the present case study describes an architectural pattern for high performance computing landscapes consisting of multiple parallel applications. It suggests one way to extend the existing interface specifications with data flow annotations. Here, neither global shared data structures nor bookkeeping of shared data structure mappings are required which makes the approach well-suited if the components reside on different machines. Finally, it introduces a pattern realising the specified data flows such that the communication overhead is small and interferes minimally with the communication demands of the involved solvers.

The remainder is organised as follows: We first introduce our use case from which we derive non-functional requirements for the data exchange. This allows us to describe and motivate our system architecture in Section III. The exact data flow and data merge mechanism in Section IV deserves special attention as it is at the heart of the present case study. Some notes on the use case realisation and experimental results illustrate the architecture's behaviour before a brief summary and outlook in Section VII close the discussion.

II. Use case

The base of the present work is a three component software assembly. It simulates the shallow water equations describing how the water waves spread throughout the computational domain for a given initial condition (Fig. 1). We are interested in a visualisation of the solution behaviour as well as the solution value at given probing points, while the user's view frustum changes as she alters her position, her view direction, and the zoom level, i.e. the region of interest in the computational domain. Such a use case arises when we simulate tsunamis, observe the predicted wave, and at the same time compare the simulation results with buoys in the ocean. An extension to other scenarios such as acoustic waves in a three-dimensional setting is (technically) straightforward. From an high level point of view, our setup realises a model-view-control architecture with a simulation acting as model.

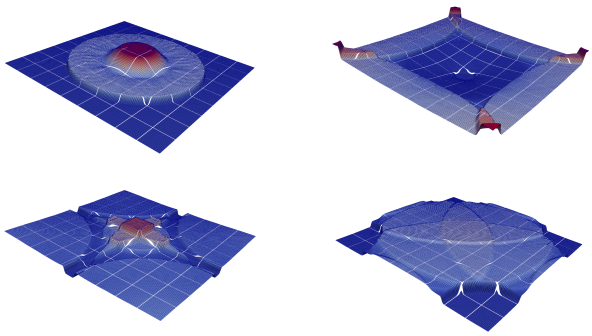

Fig. 1. Breaking dam simulation for the shallow water equation at four different times.

The choice of a proper state-of-the-art hyperbolic simulation code has severe implications: An accurate result depends on adaptive mesh refinement (AMR) where the mesh width locally adapts to the current solution. In general, regions around spreading waves are resolved more accurately than regions with a calm water surface. As the computational mesh changes all the time, also the domain decomposition, i.e. how the subdomains are distributed among the individual compute nodes, changes due to dynamic load balancing. If we track values or regions of the solution interactively, the data source, i.e. where the appropriate data comes from, changes in time. As

the FLOPS on the individual compute nodes are cheap and tend to become cheaper or even free in the future [1], the individual compute nodes are communication-bound. As a result, it is desirable to protect the communication devices from additional work—as few as possible additional data should run through the simulator's interconnects to create the visualisation. The accuracy of a hyperbolic solver correlates with the mesh width. The finer the computational mesh, the better the accuracy. As a result, it is desirable to use as much memory as possible per compute node to simulation the wave spreading—almost no memory should be spent on additional tasks such as the visualisation or data tracking. In particular data required for the visualisation should leave the individual cores as soon as possible.

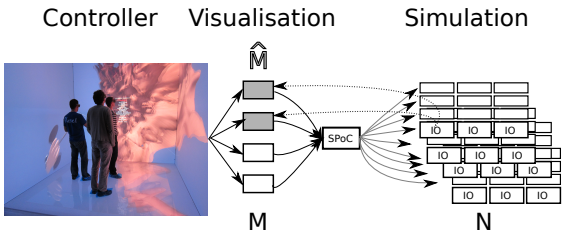

Fig. 2. Three component architecture: Thin workstation for controlling, visualisation, and simulation.

Our simulation runs on a supercomputer on multiple nodes. It is a massively parallel component. Besides this component, a small steering client runs on the user's work station. It provides the human-computer interface, accepts user control (such as a change of the viewer's position), or bookkeeps the water surface displacement at the probing points. These two components are complemented by a visualisation component. This one runs on a distinguished postprocessing cluster, i.e. is a parallel code as well (Fig. 2). The individual postprocessing nodes typically correspond to different visualisation walls/tiles and require data to be seen and rendered on these devices. As the user's view frustum changes, the mapping which data is required where is time dependent. We run into a routing challenge as data sources and destinations are volatile. Finally, the interconnects between the three individual components, i.e. the clusters hosting them, can be weak while the network for the supercomputer application can be assumed to be a high-end product: The assembly has to cope with a low IO density on the coarsest communication level.

All three involved software components were written independently of each other. A tight coupling of them is important. However, this tight coupling shall not induce significant code changes. It should be minimally invasive.

III. System architecture and design rationale

Our data flow follows the query retrieval mechanism from [9], [10], as streaming data from the compute nodes to the postprocessing devices and filtering it there, is not an option due to the hard IO limitations: Whenever the user or the probe tracking component require data, they specify a region of interest, desired resolution and physical property (wave height here) to be studied : all of our data requests encode spatial information and filter in the unknown space. The control component sends this request to a distinguished receiver of the simulation code. This is by default the global master node of the hyperbolic solver running on a supercomputer. Such a request is mapped to a remote procedure call. Hereby, the communication realises a single point of contact scheme. This request only includes meta information about the query, so this scheme does not imply a bottleneck due to its small memory footprint. As the requests refer to spatial data, each request is (recursively) broken down into subrequests mirroring the fact that some nodes handling subdomains of the wave simulation can handle selected requests while other nodes cannot contribute with data. This request decomposition is called splitting mechanism: Let a user ask for data samples $x_0, x_2, \cdots \in \mathbb{R}^d$ passed as input parameters to a procedure call. A node handling Ω_i with $x_i \in \Omega_i$ can answer to this data sample. Others can not. The domain decomposition on the simulation side decomposes the remote procedure calls. This renders the data request a $M : N$ call where calls are split/forwarded internally by the simulation software.

As the answers of the calls are processed by those nodes holding the respective subdomains Ω_i, the answering mechanism is asynchronous: If one postprocessing node expects multiple data entries from a function, these entries might become available at different times as they stem from different sources. We label the individual data requests with markers identifying which postprocessing node on the receiver side expects the results. Split requests inherit the routing information from the original request. This way, the data sources can send results directly to the right data sink. Data routing is encoded in the data requests.

A direct implementation of the resulting $M : N : \hat{M}$ data flow (Fig. 2: the receivers $\hat{M}$ do not have to be the nodes M requesting the data) that merges all N results on a receiver $\hat{M}$ does not mirror the architectural constraints; neither does a merge of all results on the supercomputer prior to the send to the postprocessing devices. In the latter case, the supercomputer's internal communication connections are stressed, and the node finally merging the data and sending it out is assigned significant temporary memory overhead and workload. In bulk synchronous environments, the latter induces workload peaks polluting any load decomposition. The other way round, if all data merges are realised on the receiver side, multiple data sources and, hence, communication channels compete for the IO interconnections between the machines. This emphasises the impact of latency and communication context switches, and it increases the IO memory footprint if multiple nodes send out redundant data. The latter argument for example gains impact for overlapping domain decompositions. We hence break down the merge/reduction into a sequence of merge steps acting on partial answers, and allow to distribute the merge steps among the nodes both on the N and $\hat{M}$ side. This distribution is tailored to both the domain decomposition and the communication resources available.

The realisation of the data exchange comes along with technical proxy code based upon MPI ports and intercommunicators or sockets, and logically can be seen as remote procedure calls. For the given application landscape, it has to support Fortran or C/C++ with the latter offering a base for Python calls on the controller. We decided to take the scientific interface definition language (SIDL), augment the

Algorithm 1 The query interface provides methods to extract data from the simulation. Implementations on the simulation side either fill data into the result arguments (out) or forward the calls to other simulation nodes. Depending on a target annotation, proxy code based upon different communication libraries is created.

```
/**
 * Interface for data retrieval through
 * queries.
 */
interface QueryRequest {
  /**
   * Asks for data on a regular Cartesian
   * grid specified by offset, bounding box,
   * and resolution.
   */
  void getData(
    in int        dataId,
    in double[]   boundingBoxOffset,
    in double[]   boundingBox,
    in int[]      resolution,
    out double[]   data
  );
}
@target=socket
class Controller uses QueryRequest as source{
}

@target=MPI, socket
class Simulation implements-all QueryRequest
uses QueryRequest as destination{
}
```

simple data types required for the present applications, and to write a small stand-alone code generator to create all proxy codes from provided SIDL definitions. Once the three involved applications provide SIDL interfaces, embedding them into the data flow pattern does not induce any code changes anymore. Thereby, each piece of software can act as an out-of-the-box CCA component. An example for such SIDL description is shown in Algorithm 1.

Per class hierarchy, our compiler generates C++ interfaces and three different types of classes. First, there are wrappers around the user-defined components or glue code around non-C++ components, respectively. Second, it generates communication classes realising either plain function calls or remote calls based upon MPI or TCP/IP sockets. A target annotation determines which components interact with which communication classes due to which connector. Finally, the compiler creates interface realisations that are able to break down requests or merge them in turn. The latter realise a composite pattern [16].

IV. Hierarchical master-worker merges

Our simulation code realises a state-of-the-art domain decomposition interacting with the data flows triggered by requests from other components. As our use case brings along huge data flows from the simulation to the visualisation cluster while the requests themselves exhibit a small memory footprint, we focus on the data flow following a request from hereon. However, an efficient realisation of incoming requests follows the same pattern.

We rely on a logical tree topology: Up to k neighbouring domain partitions that are connected via common faces can be joined into one virtual subdomain. Whenever such a scheme is applied bottom-up recursively, we end up with a logical tree topology: Starting top-down, the grid is recursively split up into subdomains. If a subdomain is not split up further, i.e. it is a real domain and not a virtual one, it is the actual compute domain Ω_i of a simulation node. In our code, the logical tree topology stems directly from a recursive subdivision of the computational domain. But the pattern is more general and can be applied to non-recursive domain decompositions. For the data treatment, we assign each virtual domain uniquely to a compute node that is responsible for one of its subdomains. This definition again is applied bottom-up recursively.

If a node requests data, each node can analyse which part of the requested data can be filled with data. As the data requests correspond to spatial information (e.g. , probes at a given location), such an analysis can be performed top-down along the logical tree topology: If a virtual domain does not contribute to the answer to a request, none of its subpartitions contributes. This scheme describes a tree broadcast with built-in filtering.

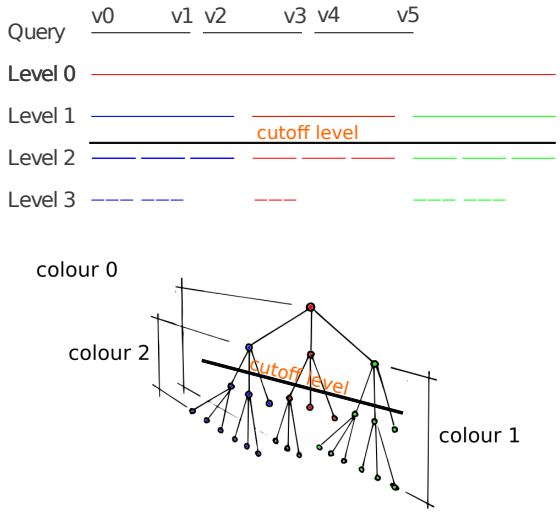

Fig. 3. Tree-based merge using a cutoff level to control the depth up to which the merges take place. Illustration follows [17].

If both a node and one of its children own entries of a result data structure that is to be sent to the same destination node, we can either make the two nodes stream their data directly to the destination and merge the two results there into one record, or we merge the data locally on the node responsible for the coarser tree level and send the merged data back then. The latter approach describes a destination-dependent reduction along the virtual tree topology. Obviously, hybrid strategies are possible where we merge data up to ℓ levels or at most up to a level ℓ within the tree (Fig. 3).

In [10], we propose to merge all data belonging to one receiver prior to any sends on the supercomputer. For these merges, we use the following responsibility strategy: merge data on the compute node that already holds most of the answer

results. For data uniquely owned by one node, this pattern equals owner-compute strategies. Otherwise, it minimises data movement. To identify the responsible node, we need one global reduction. The whole merge algorithm is executed only on the data source side. This strategy synchronises due to the global communication, requires significant memory on the selected nodes to gather the data for the individual answers, and requires high bandwidth on the merging node.

The present work relies on a given logical tree topology and does not take into account the data cardinality. Due to the fact that each virtual domain is assigned to one of the compute nodes responsible for its children, we however still preserve data locality. Since good load balancing algorithms try to preserve physical locality, i.e. to place neighbouring partitions on devices also physically close to each other, data locality is implied for the fully asynchronous merge processes as well—we expect required data exchanges for example to be realised via fast shared memory. The same argument makes one expect good IO efficiency as usually multiple neighbouring physical nodes share IO resources (psets). Due to the fact that tree topologies often are given explicitly or can be reconstructed cheaply, the potential merge paths can be evaluated efficiently. A tree-constrained restriction path still can merge all data prior to sending them out. It then mirrors the concept of [10] without a dynamic decision where to merge—instead, the compute node being assigned to the topological tree root then conducts all merges.

In realistic scenarios, a deep logical tree topology faces spatial queries affecting only few compute nodes. Consequently, only the first few restriction steps modify the result data, i.e. really conduct merges. All further reductions then basically act as identity on the results, i.e. it does not make sense to merge. If multiple subtrees come into play for multiple queries, those subtrees can answer the queries in parallel. If they belong to different psets, they do not compete for resources. While the individual nodes typically are synchronised by the simulation, a subtree can send out its results as soon as the merges terminate. This makes different answers being sent out asynchronously. Such an communication shift in time circumnavigates bandwidth constraints.

If answers to a request for one destination node are not merged completely by the simulation component, multiple pieces of data asynchronously arrive on the receiver side and have to be merged there. The number of concurrent merges has to be selected carefully to fit to the receiver's compute power. In most scenarios however the constraints induced by IO are more severe.

Tracking the answer data routes, we end up with one tree topology per receiving node. The root is the receiver itself. Its children are the $\hat{N}$ nodes of the virtual simulation tree topology that actual send data to the destination. These children in turn have other nodes that feed them with data. These data routing trees (one per destination node) build up on-the-fly for each data request. Their shape is determined by the request, the domain decomposition, i.e. the load balancing, the actual cluster topologies, and the level constraints on ℓ, and, for subsequent requests for the same region, may change frequently as does the grid. Given an interface where procedure call arguments are annotated with destination markers encoding ranks, IP addresses, and so forth, the code generator translating the interface definition into proxy code can automatically split, distribute, merge, and forward data if we equip it with hook-in points to change configuration parameters such as the coarsest merge level ℓ and a spatial distribution logic. Data routing then is hidden from the application.

V. Realisation

A. Hardware description

We conducted all experiments on the CoolMAC cluster at the Leibniz Supercomputing Centre. Our tests used the AMD partition consisting of 19 quad socket AMD Bulldozer Opteron 6274 nodes with 256 GB RAM as simulation host and made the Intel partition of the cluster—28 dual socket Intel SandyBridge-EP Xeon E5-2670 nodes with 128 GB RAM each—running the visualisation codes. Both partitions are connected through a QDR InfiniBand interconnect. In total the Intel partition offered 32 processes per node or 896 processes on the whole system, while the AMD partition made 64 processes per node and 1216 processes in total available. To estimate the capabilities of the interconnect between the two systems we employ the Intel (R) MPI Benchmark Suite V3.2.4, MPI-1 part [18] and the b_{eff} benchmark [19]. They yield around 100 Gbps (Gigabit per second) as effective bandwidth in the b_{eff} benchmark and around 150 Gbps for the $Alltoall$ communication in the Intel MPI benchmark for our setup.

B. Software details

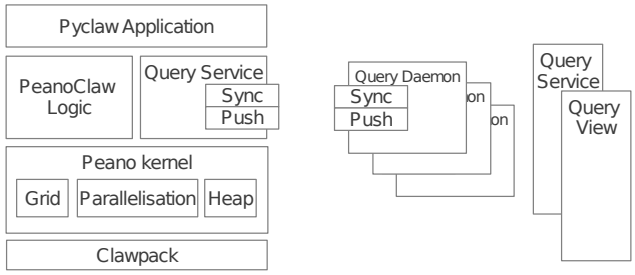

Fig. 4. System architecture of the example software.

Our component firing the function calls was a small hand-written demo code controlling the experiments. The visualisation component is based upon the VTK library [20] and the parallel Paraview server [21]. We wrapped a thin data retrieval layer around the server.

As simulation code, we relied on the package PeanoClaw [22] combining the Python-based PyClaw software [23] realising hyperbolic equation system solvers on regular Cartesian grids with the parallel AMR framework Peano [24] written in C++. We simulated a simple breaking dam scenario (Figure 1) where our refinement criterion is based on the gradient of the solution. It makes the fine mesh follow the wave distribution and, due to the well-known wave distribution characteristics, we can balance the grid dynamically and equally. The system architecture is summarised in Figure 4.

The compiler translating the SIDL-like interfaces into proxy code as well as the whole application assembly code were plain Java applications with a front-end realised as the Eclipse plug-in ASCoDT [25]. All generated code is stand-alone realising the communication via direct function calls

(within the same process space), MPI, or TCP/IP sockets (between the simulation and the visualisation cluster partition).

VI. Results

For our measurements, we ran the two dimensional hyperbolic equation solvers with around $4.3 \cdot 10^7$ cells on $N = 730$ MPI ranks. We made the controller ask the simulation to write simulation snapshots to regular Cartesian grids (queries). The snapshot grids are coarser than the actual computational mesh, so the answering simulation nodes map their simulation data to these grids due to bilinear interpolation. The regular grids holding interpolated data then were fed into the visualisation component. Two request types with a 4096×4096 grid were studied: they either cover the whole domain, i.e. each compute node contributes to the result, or cover only half of the domain. We chose the layout of the visualisation server such that we can control due to the covered domain size of the queries whether all $\hat{M}$ nodes receive data or only half of them. While we fix N, $\hat{M}$ is one of the experiment parameters. The interplay of different query characteristics and resolutions is studied in [10]. In the present experiments, we focused on effectively used bandwidth in-between simulation and visualisation. For all experiments, we stopped to merge results data on the levels $\ell = \{1, 2, 3\}$. To cut off the merge on level $\ell = 1$ implies that all result data is merged at the root of the virtual simulation tree, while no merges have to be conducted on the visualisation component side. $\ell = 2$ allows up to $k = 9$ simulation nodes to send their data in parallel. $\ell = 3$ induces up to $k = 81$ nodes if the data has not been merged completely before.

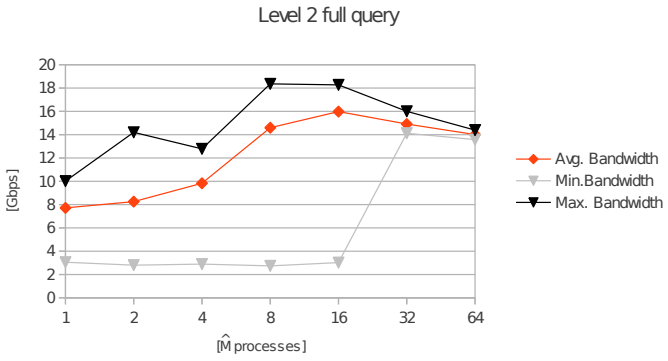

Fig. 5. Bandwidth for a query over the whole domain with different numbers of visualisation nodes $\hat{M}$. The data is reduced on the simulation side up to a tree level of $\ell = 2$, i.e. up to nine simulation nodes contact directly the visualisation side.

Our measurements first study the minimal, average, and maximal effective throughput for the data retrieval from the whole domain (Figures 5 and 6) with the TCP/IP protocol. We observe that the more data sources we use, the better the overall throughput. Aggressive restriction on the simulation side does not pay off in general. The best throughput also depends on the number $\hat{M}$ of receivers. We observe that for very small $\hat{M}$ counts, more aggressive restriction pays off (e.g. , cmp. $\hat{M} = 2$ for $\ell = 2$ and $\ell = 3$). $\hat{M}$ correlates directly to a good choice of ℓ: the best results are achieved if the number of senders is close to the number of receivers yet does not exceed this number dramatically. The huge gap between the measured

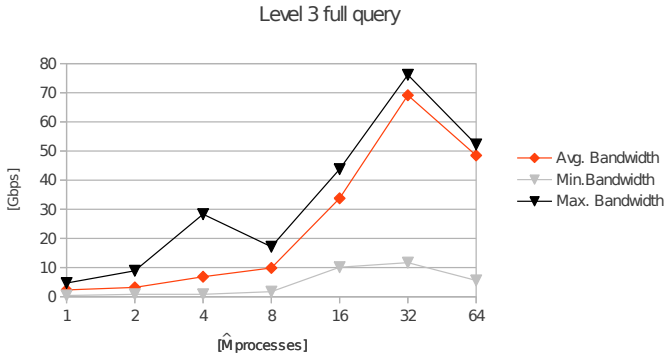

Fig. 6. Bandwidth for a query over the whole domain with different numbers of visualisation nodes $\hat{M}$. The data is reduced on the simulation side up to a tree level of $\ell = 3$, i.e. up to 81 simulation nodes contact directly the visualisation side.

minimal bandwidth and the average or maximal bandwidth can be explained by the used TCP/IP protocol and the underlying congestion window algorithm.

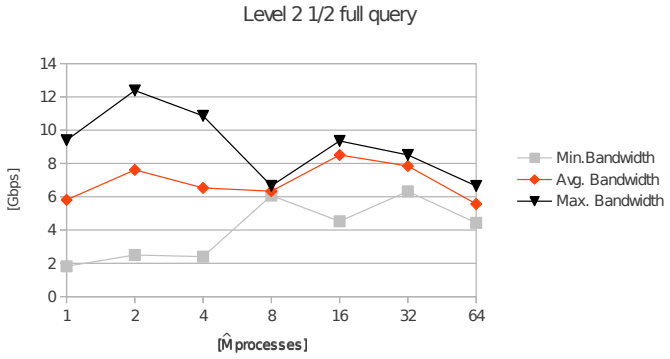

Fig. 7. Bandwidth for a query covering half of the whole domain with different numbers of visualisation nodes $\hat{M}$. Only $\hat{M}$ of these nodes as well as half of the simulation nodes are involved in the data transfer. The data is reduced on the simulation side up to a tree level of $\ell = 2$, i.e. up to five simulation nodes contact directly the visualisation side.

We next study the data retrieval with the same memory footprint for requests that involve only half of the visualisation nodes as well as half of the simulation nodes, as it refers only to half of the whole domain (Figures 7 and 8). Again, the more data sources send their results to the postprocessing cluster, the better the total peak. This time, this observation holds independent of $\hat{M}$. At the same time, the sensitivity with respect to the total number of $\hat{M}$ (where always only half of them are sent data) is higher. The more spatially specific the query, the fewer the number of reductions on the simulation side.

In practice, the only parameter we can control is ℓ. It determines how the data merges are deployed among the simulation and the visualisation nodes. All other system settings (M, $\hat{N}$, type of queries) are prescribed by the use case or user, respectively. If requests are passed to the simulation system where all or many simulation nodes are involved in answering, the restriction should be chosen such that most of the merges

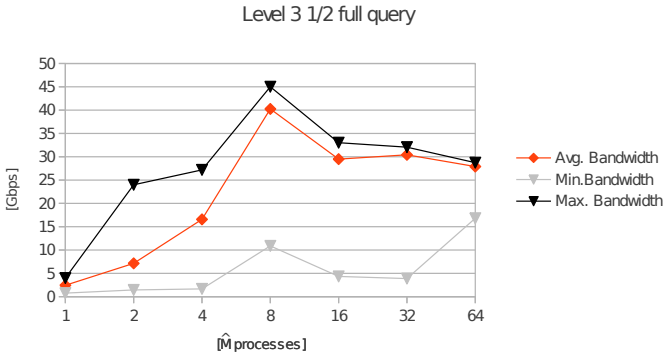

Fig. 8. Experiment of Figure 7. This time, the data is reduced on the simulation side up to a tree level of $\ell = 3$, i.e. up to 41 simulation nodes contact directly the visualisation side.

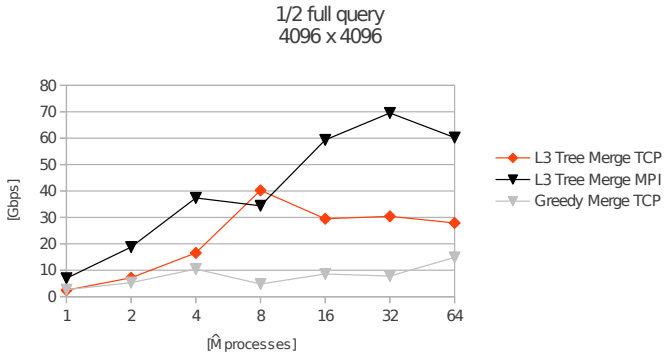

Fig. 10. Bandwidth comparison for a full resolution query over half of the domain using TCP/IP greedy merge [10] and level 3 tree merge with TCP/IP and MPI Ports.

with respect to each receiver are done on the simulation side. Overbooking with respect of the receiver number does not pay off. This way, the receiver basically achieves stream throughput while the merge workload is distributed equally. We achieve up to 70 percent of stream benchmark throughput. If requests are handed in that affect only few simulation nodes, restriction on the simulation side should be applied carefully. For the present experiments, merging the results on the receiver side is faster, as such merges do not interfere with the application behaviour, i.e. does not slow down particular nodes. The latter result however holds if and only if the load balancer yields well-balanced partitions on the simulation side.

We finally study three different software realisations of the merge: a TCP/IP version of the greedy merge from [10], and TCP/IP sockets and MPI ports realising the present tree merge with cutoff $\ell = 3$. All comparisons are done for the full and the half query (Figs. 9 and 10). The MPI ports outperforms the TCP/IP version of the algorithm by factor of two. The greedy approach exhibits worse scaling than the MPI ports variant and does not scale beyond four destination nodes. By using more than one client connections per visualisation server process the tree algorithm manages to improve the overall throughput.

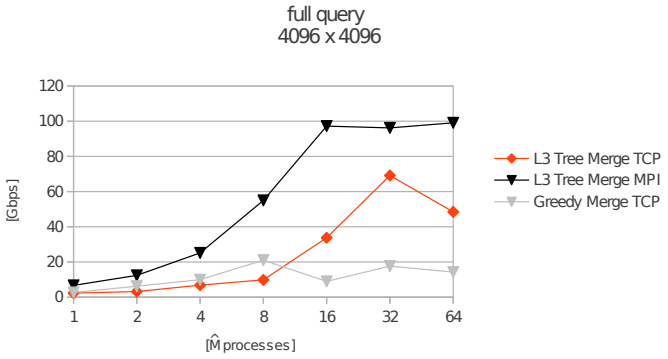

Fig. 9. Bandwidth comparison for a full resolution query over the whole domain using TCP/IP greedy merge and level 3 tree merge with TCP/IP and MPI Ports.

We omit the actual rendering times of the visualisation due to the fact that in performed experiments the overall performance of the rendering was orders of magnitude faster, thus the communication had the biggest influence on the actual "time-to-solution".

VII. Summary and Outlook

The present work combines three software components into one high performance solution with an AMR solver with dynamic load balancing and an on-the-fly visualisation. While the software assembly tackles an application toy problem and exhibits prototype character, the work addresses fundamental challenges of multi-component interaction and computational steering.

It becomes obvious that on-demand data retrieval (as well as data distribution/splits if their memory footprint becomes large) is one software pattern that allows to couple multiple parallel components. Here, it pays off to integrate routing information, i.e. which node has to send which data where, into the communication signatures. Another successful pattern is the distribution of data reduction/merge and distribution/split kernels among different component instances and different components and make this distribution fit to the physical topology of the machine and the application's communication characteristics.

Though realised as technical proof-of-concept and as stand-alone tools, the individual building blocks—in particular the ideas underlying the SIDL-like compiler—next can be integrated into larger software endeavours though the low overhead due to the lack of any middleware also is one of the selling points of the present solution. Natural next steps comprise the usage of the present technologies for larger applications and real-world scenarios. Here, we hope that the vision of real-time interactive high performance computing came one step closer. Technologically challenging is an autotuning of the reduction levels ℓ to the query characteristics on-the-fly. The dependency of the tuning parameter on the setting has been elaborated in the present work. In the future, it should be chosen automatically.

Acknowledgements

This work partially is based on work supported by Award No. UK-c0020, made by the King Abdullah University of

Science and Technology (KAUST). The support of the Leibniz Supercomputing Centre under award pr63no is highly appreciated. Additional thanks are due to Robert Guder for implementing several software components used in this work.

REFERENCES

[1] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[2] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable i/o forwarding framework for high-performance computing systems," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[3] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. Papka, R. Ross, and K. Yoshii, "Accelerating i/o forwarding in ibm blue gene/p systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov., pp. 1–10.

[4] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/o performance challenges at leadership scale," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 40:1–40:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654100

[5] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. Jansen, "The ParaView coprocessing library: A scalable, general purpose in situ visualization library," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, oct. 2011, pp. 89 –96.

[6] C. Johnson, R. Ross, S. Ahern, J. Ahrens, W. Bethel, K. Ma, M. Papka, J. van Rosendale, H. Shen, and J. Thomas, "DOE visualization and knowledge discovery: Report from the doe/ascr workshop on visual analysis and data exploration at extreme scale," October 2007. [Online]. Available: http://www.sci.utah.edu/publications/crj07/DOE-Visualization-Report-2007.pdf

[7] K.-L. Ma, "In situ visualization at extreme scale: Challenges and opportunities," *Computer Graphics and Applications, IEEE*, vol. 29, no. 6, pp. 14 –19, nov.-dec. 2009.

[8] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, ser. EG PGV'11. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2011, pp. 101–109. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV11/101-109

[9] A. Atanasov and T. Weinzierl, "Query-driven multiscale data post-processing in computational fluid dynamics," in *Proceedings of the International Conference on Computational Science, ICCS 2011*, ser. Procedia Computer Science, M. Sato, S. Matsuoka, G. D. van Albada, J. Dongarra, and P. M. Sloot, Eds., vol. 4, Jun. 2011, pp. 332–341.

[10] A. Atanasov, M. Srinivasan, and T. Weinzierl, "Query-driven parallel exploration of large datasets," in *Large Data Analysis and Visualization (LDAV), 2012 IEEE Symposium on*, Oct. 2012, pp. 23 –30, doi 10.1109/LDAV.2012.6378972.

[11] M. P. I. Forum, "MPI: A message-passing interface standard," Tech. Rep., 2012.

[12] CCA—The Common Component Architecture Forum. http://www.cca-forum.org.

[13] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high-performance scientific computing," in *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*. IEEE, 1999, pp. 115–124.

[14] F. Bertrand, R. Bramley, A. Sussman, D. E. Bernholdt, J. A. Kohl, J. W. Larson, and K. B. Damevski, "Data redistribution and remote method invocation in parallel component architectures," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 2005, pp. 40b–40b.

[15] C. Rene and T. Priol, "MPI code encapsulating using parallel CORBA object," in *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, 1999, pp. 3–10.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: elements of reusable object-oriented languages and systems," 1994.

[17] T. Weinzierl, *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. München: Verlag Dr. Hut, 2009. [Online]. Available: http://www.dr.hut-verlag.de/978-3-86853-146-6.html

[18] "Intel MPI benchmarks 3.2.4," http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[19] R. Rabenseifner and A. E. Koniges, "The parallel communication and i/o bandwidth benchmarks: b_eff and b_eff_io," in *Proc. of 43rd Cray User Group Conference, Indian Wells, California, USA*. Citeseer, 2001.

[20] "VTK the visualization toolkit," http://http://www.vtk.org/.

[21] "ParaView - open source scientific visualization," http://www.paraview.org/.

[22] K. Unterweger, T. Weinzierl, D. I. Ketcheson, and A. Ahmadia, "Peanoclawa functionally-decomposed approach to adaptive mesh refinement with local time stepping for hyperbolic conservation law solvers," *Technische Universität München, Technical Reports*, 2013.

[23] K. T. Mandli, D. I. Ketcheson *et al.*, "Pyclaw software," 2012, numerics.kaust.edu.sa/pyclaw. [Online]. Available: http://numerics.kaust.edu.sa/pyclaw

[24] T. Weinzierl *et al.*, "Peano—a Framework for PDE Solvers on Spacetree Grids," 2012, www.peano-framework.org. [Online]. Available: http://www.peano-framework.org

[25] A. Atanasov, H.-J. Bungartz, and T. Weinzierl, "A toolkit for the code development in advanced computing," Tech. Rep. TUM-I1330, 2013.