Analyzing One-Sided Communication Using Memory Access Diagrams

Olaf Krzikalla[1], Arne Rempke[1], and Ralph Müller-Pfefferkorn[2]

[1] German Aerospace Center, Dresden, Germany
{olaf.krzikalla,arne.rempke}@dlr.de
[2] Technische Universität, Dresden, Germany
ralph.mueller-pfefferkorn@tu-dresden.de

Abstract. In recent years, one-sided communication has emerged as an alternative to message-based communication to improve the scalability of distributed programs. Decoupling communication and synchronization in such programs allows for more asynchronous execution of processes, but introduces new challenges to ensure program correctness and efficiency. The concept of memory access diagrams presented in this paper opens up a new analysis perspective to the programmer. Our approach visualizes the interaction of synchronous, asynchronous, and remote memory accesses. We present an interactive tool that can be used to perform a post-mortem analysis of a distributed program execution. The tool supports hybrid parallel programs, shared MPI windows, and GASPI communication operations. In two application studies taken from the European aerospace industry we illustrate the usefulness of memory access diagrams for visualizing and understanding the logical causes of programming errors, performance flaws, and to find optimization opportunities.

Keywords: One-Sided Communication · PGAS Programming Models · Memory Access Analysis.

1 Introduction

Today, distributed applications can create a shared global address space across nodes of an HPC system by using asynchronous, one-sided memory accesses to remote memory. This global address space has given rise to new programming models like one-sided MPI or GASPI[1], which allow new ways to develop powerful applications, but also introduces new challenges to ensure program correctness and efficiency.

In a one-sided communication model a process can directly access memory areas of another target process. For the target process, such accesses are transparent, leading to a decoupling of data transfer and process synchronization. In addition, these accesses can even be asynchronous to the executing process, allowing an overlap of computation and communication. However, the potential interaction of direct, asynchronous and remote memory accesses to the same memory region increases the software complexity. The software developer must ensure race-free memory accesses across the entire process space. On the other hand, process synchronization must not be stricter than necessary, otherwise the performance and scalability benefits of one-sided communication will be lost.

The paper contributes to the systematic understanding of parallel distributed applications that use one-sided, asynchronous communication. The introduced visualization concept opens up a new analysis perspective to the programmer. We present an interactive visualization tool for the memory access analysis of GASPI programs with an extension for MPI shared windows. Our tool supports the analysis of hybrid parallel programs, i.e. programs combining distributed and shared-memory parallelization. Two application studies on the analysis of industrial codes from the European aerospace industry complete the paper.

2 Terms, Operations, and Execution Model

Throughout this paper, we distinguish between two types of memory accesses. A *direct memory access* is executed synchronously by a thread, usually by a CPU instruction. It always accesses process-local memory. An *asynchronous memory access* is caused by a one-sided communication operation and is usually part of a data transfer between processes. It can access local memory as well as remote memory. In addition, we use the term *remote memory access* for asynchronous memory accesses to the local memory of one process caused by a communication operation of another process.

A one-sided communication operation copies data from the memory of one process to the memory of another process. It is usually accompanied by synchronization operations. For example, the `write_notify` operation specified by the GASPI standard copies data from local source memory to target remote memory. It also sets a notification flag on the remote side signaling the completion of the data transfer. Source and remote memory reside in dedicated regions. In GASPI these regions are called *segments*, in MPI *windows*. Asynchronous memory accesses can only occur to these regions.

We model a program execution as a *task graph*. A task executed by a thread represents either a direct memory access or a communication operation. We compute the synchronization relations between threads and processes using a replay algorithm. This algorithm replays the recorded synchronization operations in an arbitrary order respecting the synchronization relations of the operations. As proved in [7] this leads to a logically sound task graph, which includes all happens-before relations across threads and processes.

In addition, the task graph is extended with *virtual* tasks. Virtual tasks are not assigned to a thread and therefore run in parallel to all threads. However, they are forked by thread tasks or other virtual tasks and eventually joined by thread tasks. Fig. 1 shows a task graph of a `write_notify` operation initiated by process 1, that writes data to the memory of process 2. The `write_notify` call is modeled as the task WN. It forks three virtual tasks R_L, W_R, and W_F. R_L represents the asynchronous read from the local memory. W_R represents the remote writing to the memory of process 2. Due to the data dependency, we model

R_L as happening before W_R. The third virtual task forked by `write_notify` is the notification task W_F. It sets a flag on the remote process signaling the completion of the write access. So W_R happens before W_F. The task WT represents a call to an operation waiting for the completion of the local asynchronous read. Note that this task waits only for the local read task, but not for the remote write task. The remote side waits for the data to arrive. The corresponding operation is modeled by the NW task. This task is triggered, when the flag is set by W_F.

Virtual tasks can be used to model all kinds of one-sided communication and synchronization operations. Many examples for the modeling of MPI, GASPI, and openShmem operations are given in [6]. Virtual tasks play a central role in our analysis approach, because they abstract from concrete APIs while accurately modeling the logical relationships of events.

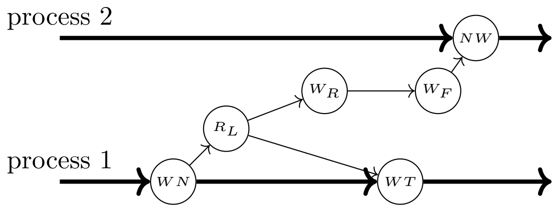


Fig. 1. Virtual tasks of a `write_notify` operation.

3 Visualization Concept and Realization

In programs with one-sided communication, the interactions between direct memory accesses and remote memory accesses form the central element of communication between processes. In order to analyze these interactions we have developed a visualization concept based on memory access diagrams.

A memory access diagram is a two-dimensional Cartesian coordinate system. It represents the course of the memory accesses related to the execution of a particular thread. The x-axis denotes the program time of the visualized thread as logical time steps. A logical time step is either a direct memory access or another event represented in the task graph (e.g. a synchronization operation). The y-axis denotes the local address space of the process to which the visualized thread belongs. Each direct memory access is represented as a vertical line, which marks the access interval at the time of the access. The access type is determined by color: read accesses are marked green and write accesses red (Fig. 2).

Fig. 2 illustrates this visualization concept. It displays the direct memory accesses to the `source` and `target` arrays as they happen over time. During each logical time step one direct memory access is performed.

Asynchronous memory accesses, like direct memory accesses, have an address interval and an access type. However, unlike direct memory accesses, such a memory access has a potentially very large address interval. A single memory access to individual data words within this interval is not represented by

```
int* source = 0x10;
int* target = 0x20;
for (int i=0; i<4; ++i)
{
  target[i] = source[i];
}
```

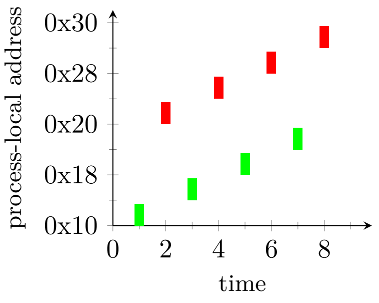


Fig. 2. Visualization of direct memory accesses in a memory access diagram.

our diagrams. Instead, each memory access by a communication operation is always represented as access to the entire address interval. This interval is plotted directly on the y-axis of the diagram.

Specifying a concrete logical time for such a memory access from the point of view of the visualized thread is usually not possible, since it occurs asynchronously to the thread execution. However, both the earliest time at which the access can take place and the latest time at which the access can take place can be specified. Thus, our diagrams draw asynchronous memory accesses as rectangles. The width of the rectangle represents the time range, at which the shown memory access can happen. The height represents the address range, and the color of the rectangle represents the access type again.

Fig. 3 illustrates this concept. The vertical sequence of tasks in the task graph are executed by a thread. The first task represents a loop similar to the one of Fig. 2. It performs four direct memory writes to address `0x10-0x20`. The second task is a write operation, which transfers the data just written to another process. This operation initiates an asynchronous read access – the virtual task `R` – to the local memory at address `0x10-0x20`. Depending on `R` is the asynchronous write access `W` (see also Fig. 1). Asynchronous accesses initiated by GASPI operations are specified by the GASPI segment id `seg`, an offset `o` into `seg`, and a byte size `sz`. For remote accesses the target rank `r` is added. The third task of the thread again represents a loop, this time writing to address `0x0-0x10`. The fourth task represents a wait operation, which synchronizes with the local read of the former write operation. The final task again represents a loop, this time reading from address `0x0-0x10`. The green rectangle in the associated memory access diagram represents the asynchronous read access from the perspective of the thread. This access starts with the execution of the write operation, and it lasts until the wait operation is executed. During that time, all memory locations inside the rectangle can be read by the asynchronous read access. The representation as a rectangle corresponds to the usual specifications of one-sided communication operations, since these specifications neither specify any access order nor further timing constraints.

In order to visualize all kinds of asynchronous memory accesses in the described way we have generalized the computation of the time range of the access as seen by a particular thread based on the task graph of the program execution. An asynchronous memory access starts with the latest thread task, which has a

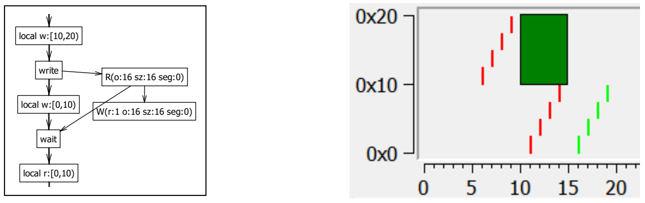


Fig. 3. Task graph and corresponding memory access diagram of a thread.

happens-before relation to the corresponding virtual task. And it ends with the first thread task, to which this virtual task has a happens-before relation.

Fig. 4 visualizes both an asynchronous local read access and a remote write access of process 1. Two processes, each consisting of one thread, exchange data via `write_notify`. The virtual notification tasks are specified by the target rank `r`, the target segment `s` and the flag id `f`. Both processes have organized its memory in the same way: the region from `0x0-0x10` is only locally accessed, the region from `0x10-0x20` is locally written and then sent to the remote process, and the region from `0x20-0x30` acts as receive buffer, from which the data received is later read. The green rectangle in the memory access diagram represents the local asynchronous read access similar to Fig. 3. The red rectangle represents the remote write access, which is caused by the other process. The red arrow in the task graph outlines how this remote access is timed in relation to process 1. From the point of view of process 1 the remote write access can occur as soon as the barrier before the data exchange is entered (`barrier(E)`). And it is not finished until the corresponding notification flag is received by the `notify_reset` operation, which corresponds to the *NW* task in Fig. 1.

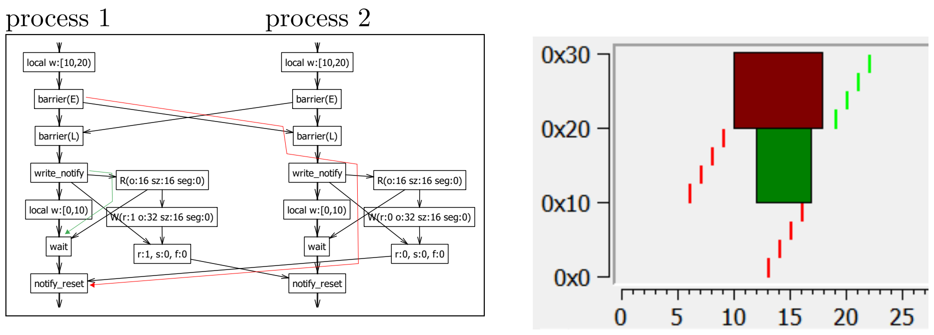


Fig. 4. Representation of a local asynchronous read access and a remote write access to the memory of process 1.

The tool presented in this paper consists of two components. The recording tool traces a program execution. The analysis and visualization tool evaluates the recorded program execution after its completion.

The recording tool is implemented on top of the dynamic instrumentation API PIN [10]. We trace function calls including the values of their arguments and the return result, and we use the fine-grained instrumentation capabilities of PIN to record the addresses of direct memory accesses. With our implementation, practical applications are slowed down by a factor of 2-10 due to instrumentation and recording, and a native 10-second run of a CFD program on 24 processes over a typical mesh, for example, will generate a total of 10 GB of analysis data [5]. However, memory access patterns do not usually depend on the length of the program run. Also, the number of processes usually does not affect the interaction between local computation and communication, only the amount of data exchanged. Therefore, when applying our analysis, one should limit oneself to a rather short program run with few processes. In our experience, this is also perfectly sufficient to obtain the desired insights.

The second component of our tool suite is the analysis and visualization tool. It reads a recorded program execution and displays a task graph for all threads of all processes. The task graph provides a global view of the program execution, including its synchronization relationships. From there, a user can select a thread and view its memory access diagram. The memory access diagram provides a view of the process-local memory as observed by the selected thread. The user may zoom in, and hide address ranges. By default, address ranges without accesses are hidden. Moreover, our tool automatically checks for data races. A memory access diagram is well suited for visualizing such a race, since races are intuitively recognizable as an overlap of multiple memory accesses.

Fig. 5 shows a case very similar to Fig. 3. The only difference is that the direct memory accesses in the third task have been shifted by 8 bytes. In this case the memory from address `0x10-0x18` is written by direct memory accesses, while at the same time the `write` operation reads data from this region. This constitutes a data race, which can also be recognized by the two direct write accesses inside the green rectangle.

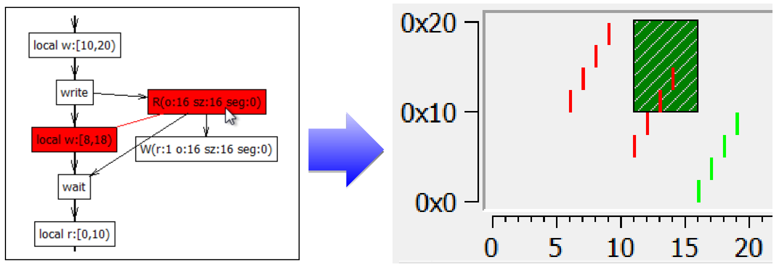


Fig. 5. Selection of a task representing an asynchronous memory access that causes a data race.

Fig. 5 also demonstrates the interactive capabilities of our tool to understand the relationships between program synchronization and memory access patterns. All tasks involved in data races are marked red in the task graph display. Red lines are drawn to the tasks representing conflicting memory accesses. The user can click on such a task to switch to the memory accesses diagram. The selected memory access is then shown hatched. Thus, a user who clicks on the mouse pointer position shown in the task graph of Fig. 5 will see the right memory access diagram with the hatched asynchronous read access.

The interaction also works the other way around. A user can select an asynchronous memory access in the diagram to see its temporal embedding in the displayed thread. The left part of Fig. 6 shows a memory access diagram of a stencil code using double buffering. Clicking at the shown mouse pointer position in the remote write access leads to the task graph on the right side. In this case, not only is the access event highlighted in gray in the task graph, but also the paths to those tasks are marked blue, that lead to the logical time limit of the memory access with respect to the displayed thread. This allows you to analyze asynchronous memory accesses not only in the context of other memory accesses, but also in the context of the triggering and synchronizing functions.

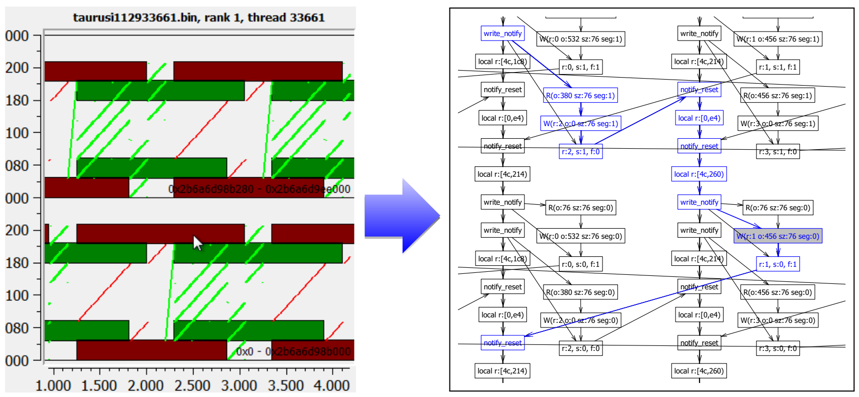


Fig. 6. Selection of a remote memory access – the causal relationship to the operations of the thread associated with the memory access diagram (here: the left thread) becomes clear.

4 Application Studies

Case studies on the application of memory access diagrams to small and commonly used code patterns are explained in [5]. In this paper we describe the analysis of two industry codes that we have evaluated using our trace and visualization tools. Both codes originate from the European aerospace industry.

4.1 Checking correctness and performance during development - Spliss

Spliss[8] is a linear solver library, which supports a wide range of linear operators typically used in computational fluid dynamics (CFD) applications. This includes sparse block matrices of variable block sizes and different scalar types as well as matrix-free operators. It is actively developed at the German Aerospace Center (DLR) and used by various research institutions and industry partners.

Since a key design goal of Spliss is HPC efficiency and scalability, a one-sided asynchronous communication strategy using GASPI was implemented. During the development we used memory access diagrams to ensure, that the communication patterns behave as intended, and to spot optimization opportunities.

Fig. 7 shows the memory access diagram of a Jacobi solver run in Spliss. Only accesses to the memory area of the halo segment of the vectors were traced, since the local part is never accessed by asynchronous or remote memory accesses. Before timestamp ~230 some initialization accesses occur. At ~230 the first Jacobi iteration starts. From then on, a regular pattern is visible with alternating direct and remote accesses.

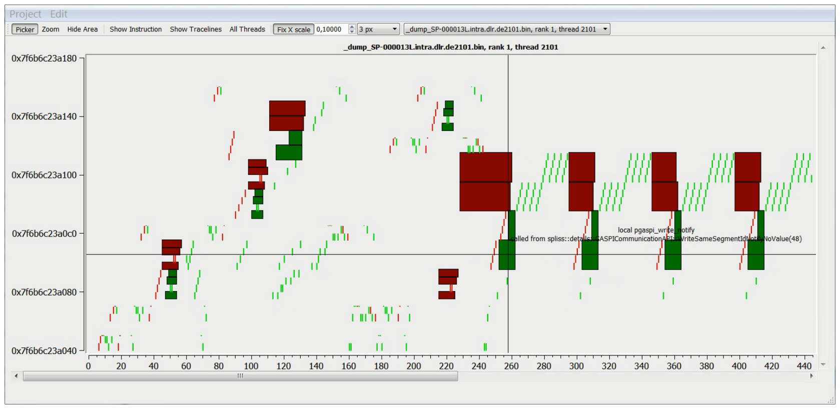


Fig. 7. Memory access diagram of the halo segment of a Jacobi solver run in Spliss.

What stands out during the iterations is the rather short duration of asynchronous read accesses and the large temporal gaps from those accesses to the following direct write accesses. This points to a very tightly coupled synchronization pattern. By clicking in one of the rectangles representing such an asynchronous read access (in the figure the cross-hair is already in the right place) the user can switch to the task graph view to examine the reason for the synchronization pattern (Fig. 8). In the task graph view the task representing the clicked read access is marked gray and the synchronization relations to the formerly watched thread are highlighted by blue arrows.

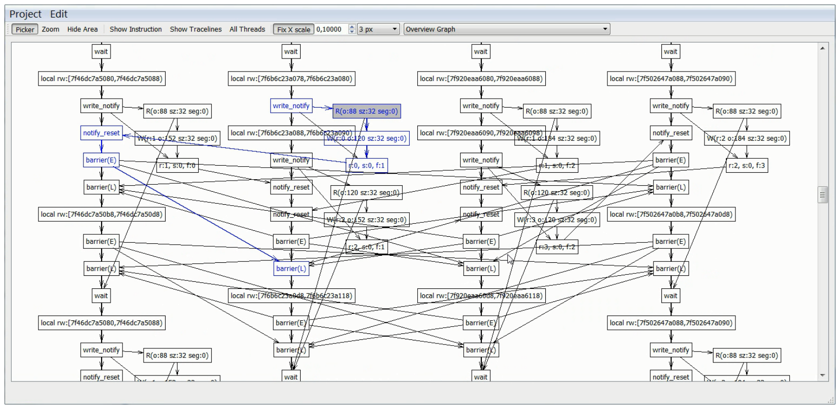
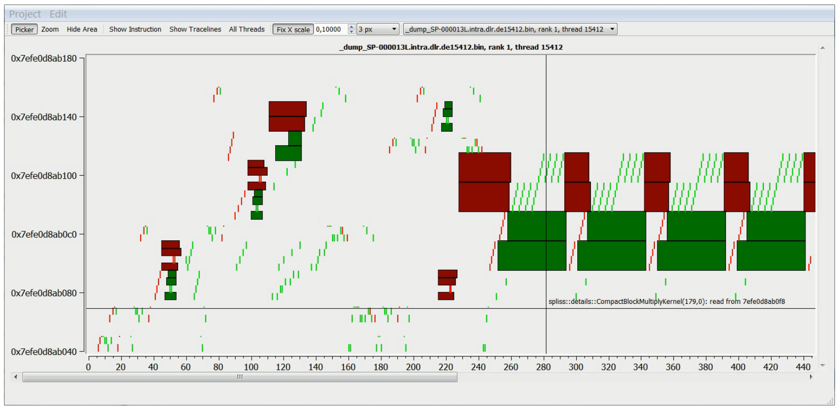

Project Edit
Picker Zoom Hide Area Show Instruction Show Tracelines All Threads Fix X scale 0,10000 3 px Overview Graph
wait
local rw:[7f46dc7a5080,7f46dc7a5088)
write_notify
R(o:88 sz:32 seg:0)
notify_reset
barrier(E)
barrier(L)
local rw:[7f6b6c23a078,7f6b6c23a080)
local rw:[7f920eaa6080,7f920eaa6088)
local rw:[7f502647a088,7f502647a090)
R(o:120 sz:32 seg:0)
r:1, s:0, f:0
r:0, s:0, f:1
r:2, s:0, f:3
local rw:[7f46dc7a50b8,7f46dc7a50d8)
local rw:[7f502647a0b8,7f502647a0d8)

Fig. 8. Task graph for a Jacobi solver run in Spliss showing the blue marked synchronization path of an asynchronous memory access.

The actually intended synchronization task for the examined read access is the `wait` task at the very bottom of the task graph view (the one which is directly connected to the gray marked read task). However, the blue synchronization path joins the watched thread already before the next direct memory access due to a global barrier (`barrier(E)`: enter barrier, `barrier(L)`: leave barrier). It turned out, that this particular barrier was not necessary during the solver loop. After we removed it, the memory access pattern changed as depicted in Fig. 9. In this diagram the temporal gaps have disappeared. On the other hand there are still no overlaps of direct and asynchronous memory accesses proofing that the change doesn't cause a data race.



```
spliss::details::CompactBlockMultiplyKernel(179,0): read from 7efe0d8ab0f8
```

Fig. 9. Memory access diagram of the halo segment of a Jacobi solver run after the removal of a gratuitous barrier.

This case demonstrates one strategy to utilize our memory access diagrams to find optimization opportunities. The developer can look for temporal gaps between asynchronous and direct memory accesses to the same addresses. By closing such gaps communication operations get more time to finish before a process actually waits for the corresponding data. Thus, the program becomes more asynchronous and hence can scale better.

In the explained example the optimization improved the performance of the solver loop by up to 30% even for few processes, since an entire global barrier could be removed. However, for this kind of optimization one should usually expect measurable improvements especially for many processes. For instance, a similar optimization approach of a stencil code only significantly increased the parallel efficiency above 256 processes [4]. A method for automatically finding such temporal gaps is also described there. To reduce the search space, such gaps are only searched for along the critical path.

4.2 Assessing memory accesses of an established code - HYDRA

Rolls-Royce's CFD solver framework HYDRA[9] uses a pure distributed parallelization scheme. Our task was to assess an evaluation version, which uses a one-sided communication scheme. In that version node-local communication is organized by shared MPI windows. Therefore, we have extended our tool suite to handle direct memory accesses to remote process memory. We have done this by mapping tasks for direct memory accesses to shared MPI windows to the process address space of the just visualized thread and handle them as virtual tasks. In this way, the synchronization relationships to other tasks are preserved.

Fig. 10 shows a memory access diagram of a sample HYDRA run. Data in the GASPI segment memory is sent and received by processes from remote nodes. Data in the SHAN segment memory is directly read and written by other processes running on the same node. Those accesses still appear as rectangles similar to remote accesses to the GASPI segment, since the access and synchronization patterns are the same for each process independent of its node location.

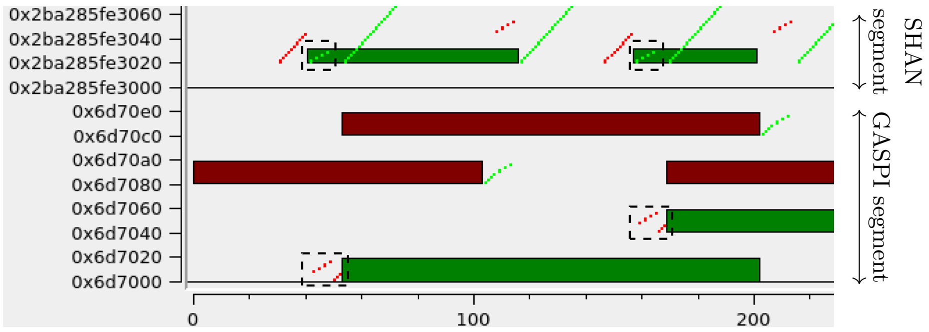


Fig. 10. Memory access diagram of halo segments of a HYDRA run. The dashed rectangles mark a copy operation that can be saved.

As you can see in the diagram, many direct memory accesses are close to remote accesses. Still, there are some temporal gaps between direct and remote accesses. During our assessment, we found that the synchronization patterns that cause these gaps are difficult to change. However, another access pattern caught our attention. The spots marked by the dashed rectangles represent a copy operation from the SHAN segment to the GASPI segment. From the GASPI segment the data is then sent to processes on remote nodes. But this data could also be transferred directly from the SHAN segment to remote nodes via GASPI operations. Thus, the copy operation can be optimized away. Then the green rectangle at address `0x6d7000` would move to address `0x2ba285fe3020`, because the data will be read from there. The diagram also shows, that in this case the GASPI remote access represented by that green rectangle needs a stricter synchronization. This prevents an overlap and hence a data race with the direct write accesses to address `0x2ba285fe3020` starting at timestamp ~140.

As a result of our assessment we have shown, that there are no data races in HYDRA regarding the interaction on direct memory accesses, accesses to shared MPI windows and remote asynchronous accesses. We have also identified several optimization opportunities. The asynchronicity of memory accesses could be increased on several occasions. In addition, a superfluous copy operation in the solver loop has been identified.

5 Related Work

Memory access analysis tools are used for a wide variety of problems. We limit our brief overview to tools targeting distributed platforms and visualization tools.

Data race detection tools that include direct memory accesses in their analysis, are *MC-Checker*[2] and *UPC-Thrille*[11]. *MC-Checker* checks one-sided MPI communication operations; and *UPC-Thrille* checks UPC programs. A performance analysis tool for MPI programs is described in [14], which traces memory accesses and provides a dataflow and dependency graph.

A visualization tool for memory accesses is described in [13]. It focuses on visualizing performance characteristics of direct memory accesses. The space-time diagrams used are very similar to our visualization concept. *MemAxes*[3] uses a radial layout to visualize the performance of direct memory accesses.

To the best of our knowledge, our tool is the only one that combines direct, asynchronous, and remote memory accesses in its visualization and analysis.

6 Conclusion

This paper extends the portfolio of available analysis methods for distributed programs with one-sided communication. The introduced tool offers new possibilities to examine all memory accesses in their respective logical context and draw conclusions about the correctness and efficiency of the program. The interactive integration of memory access diagrams with the task graph model makes it possible to study the logical relationships in a program in detail. The visualization can also contribute to a better understanding of the often complex interrelationships of threads and processes in development meetings, documentation or even in teaching.

Our tool combines the analysis of local memory accesses, accesses via shared MPI windows, and GASPI operations. However, the underlying model is API-independent. One-sided MPI operations can easily be supported and will be integrated as the need arises. We expect the popularity of one-sided communication to continue to grow as a remote completion operation similar to GASPIs `write_notify` is integrated into MPI [12].

The recent addition of shared MPI windows to our tool suggests another improvement. In principle, this method can also be used to examine shared memory programs. The visualization can reveal basically the same information about the access patterns of a shared memory program. Overlaps would still indicate data races, and it would still be important to have as few gaps as possible between memory accesses of the watched thread and the memory accesses from other threads to the same region.

Acknowledgements We would like to thank the Center for Information Services and High Performance Computing (ZIH) Dresden, esp. Prof. Wolfgang Nagel for supporting this work. We would also like to thank the German Aerospace Center (DLR) and Rolls-Royce for their support and the permission to publish analysis results of their codes.

References

1. Alrutz, T., Backhaus, J., et al.: Gaspi – a partitioned global address space programming interface. In: Facing the Multicore-Challenge III: Aspects of New Paradigms and Technologies in Parallel Computing. pp. 135–136. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
2. Chen, Z., Dinan, J., et al.: Mc-checker: Detecting memory consistency errors in mpi one-sided applications. In: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 499–510 (Nov 2014)
3. Gimenez, A.A., Gamblin, T., et al.: Memaxes: Visualization and analytics for characterizing complex memory performance behaviors. IEEE Transactions on Visualization and Computer Graphics **PP**(99), 1–1 (2017)
4. Herold, C., et al.: Optimizing one-sided communication of parallel applications using critical path methods. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 567–576 (May 2017)
5. Krzikalla, O.: Neue Ansätze zur Speicherzugriffsanalyse paralleler Anwendungen mit gemeinsam genutztem Adressraum. Ph.D. thesis, Technische Universität Dresden (2018)
6. Krzikalla, O., Knüpfer, A., et al.: On the Modelling of One-Sided Communication Systems. In: Proceedings of the 7th International Conference on PGAS Programming Models. pp. 41–53. Edinburgh, UK (October 2013)
7. Krzikalla, O., Müller-Pfefferkorn, R., Nagel, W.E.: Synchronization debugging of hybrid parallel programs. In: Dutot, P.F., Trystram, D. (eds.) Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings. pp. 37–50. Springer International Publishing, Cham (2016)
8. Krzikalla, O., Rempke, A., et al.: Spliss: A sparse linear system solver for transparent integration of emerging hpc technologies into cfd solvers and applications. In: STAB-Symposium 2020. pp. 635–645. Notes on Numerical Fluid Mechanics and Multidisciplinary Design, Springer International Publishing (Juli 2021)
9. Lapworth, L.: Hydra: A framework for collaborative cfd development. In: International Conference on Scientific and Engineering Computation. Singapore (2004)
10. Luk, C.K., et al.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 190–200. PLDI '05, ACM, New York, NY, USA (2005)
11. Park, C.S.: Active Testing: Predicting and Confirming Concurrency Bugs for Concurrent and Distributed Memory Parallel Systems. Ph.D. thesis, EECS Department, University of California, Berkeley (December 2012)
12. Sergent, M., Aitkaci, C.T., et al.: Efficient notifications for mpi one-sided applications. In: Proceedings of the 26th European MPI Users' Group Meeting. EuroMPI '19, Association for Computing Machinery, New York, NY, USA (2019)
13. Servat, H., Llort, G., González, J., Giménez, J., Labarta, J.: Low-overhead detection of memory access patterns and their time evolution. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings. pp. 57–69. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
14. Subotic, V., Ferrer, R., Sancho, J.C., Labarta, J., Valero, M.: Quantifying the potential task-based dataflow parallelism in mpi applications. In: Proceedings of the 17th International Conference on Parallel Processing - Volume Part I. pp. 39–51. Euro-Par'11, Springer-Verlag, Berlin, Heidelberg (2011)