# Visualization of Open Community Runtime Task Graphs

Jiri Dokulil
University of Vienna, Austria
jiri.dokulil@univie.ac.at

Jana Katreniakova
Comenius University, Bratislava, Slovakia
katreniakova@dcs.fmph.uniba.sk

**Abstract**

*The emergence of new types of high performance hardware also drives the need for new programming models. The Open Community Runtime (OCR) proposal uses a task-based programming model to target some of these architectures. In OCR, the whole program from start to end needs to be expressed using tasks and synchronized using task-to-task dependences, significantly limiting the applicability and usefulness of existing approaches to application development and debugging. In this paper, we present our approach to visualizing tasks and their synchronization, based on trace data from application execution. This way, the application developer may compare the intended organization of the tasks with the actual dependences as they are seen by the OCR runtime system.*

## 1 Introduction

The Open Community Runtime (OCR, [9]) is a task-based distributed runtime system. It is by far not the first task-based programming model, but unlike many of the alternatives, like the Intel Threading Building Blocks (TBB, [8]), it requires the whole application to be expressed as tasks. In TBB, the application developer only uses tasks to parallelize the performance critical parts of the code. In OCR, even the application entry point is a task. The execution of tasks is synchronized using task dependences. From a high-level point of view, a dependence specifies that one task (source of the dependence) needs to finish before another task (destination of the dependence) is allowed to start. The tasks and their dependences form a DAG.

The OCR memory model requires all data to be stored in data blocks managed by the OCR runtime. To make a data block accessible inside a task, the runtime needs to be made aware of this before the task starts. If the computation needs access to a new piece of data, it is necessary to end the running task and create a new task, which is given access to the new data. As a result, the applications are likely to be split into many small tasks, creating a long chain. Furthermore, a task contains only serial code. To allow for parallel execution, multiple tasks need to be running at the same time. So, there isn't just one, but many of the (long) task chains.

All tasks and dependences are created and managed dynamically, while the application is running. Combined with parallel (or even distributed) execution, this makes the OCR application codes very difficult to debug. Visualization may provide some assistance to the application developers. In this paper, we deal with visualization of the tasks and their dependences. We use data from the runtime system to reconstruct the task graph and dependences. This allows us to show the programmer how the runtime interpreted the information about tasks and dependences passed by the programmer. Ideally, this should be the same as the programmer's intent. Unfortunately, the OCR programming model, which is very different from those that the programmers are usually familiar with (like MPI or OpenMP), makes it easy for the programmer to assume certain ordering of tasks but he or she may fail to specify the appropriate dependences. This way, the tasks could be executed in a different order than the programmer intended, producing incorrect results or even making the application crash.

The rest of the paper is organized as follows. First, we provide a brief description of the relevant aspects of OCR in Section 2. Then, Section 3 describes the way we collect and visualize the task data. Section 4 provides examples of visualized task graph of three different OCR applications. Related work is covered in Section 5. The last section concludes the paper and discusses future work.

## 2 Open Community Runtime

In OCR, a task is a C function. Any data block accessible by a task is passed as an argument to the function. The function contains computation, but it may also contain calls to the OCR API, which is used to create new OCR objects (like tasks or data blocks) and manage these objects (e.g., set up dependences). The dependences in OCR are not specified directly among tasks. Instead, events are used. Event is an OCR object used purely for synchronization of tasks. Dependences may be set up between an event (as the source) and event or task (as the destination). For each task, there is an automatically created event (completion event), which represents the end of that task. So, if a

task $B$ needs to start after task $A$ finishes, a dependence needs to be set up from $A$'s completion event to task $B$.

The OCR specification provides a memory model for OCR programs. The memory model is based on three relations. First, the order among operations within a single task is defined by the *sequenced-before* relationship. This is provided by the C language used to implement the tasks and it is the natural ordering of operations performed by a C program, as one would expect. Second relation is *synchronized-with*, which is defined by dependences among OCR events and tasks. The simplest example is a task, whose completion event is used as a dependence (precondition, pre-slot) for another task. In this case, it is natural to expect that the second task comes after the first task. There are more complex examples of *synchronized-with*, which will be discussed in the next paragraph. The third relation is *happens-before*, which is a transitive closure of combined *sequenced-before* and *synchronized-with*.

Task dependences are not the only factor that affects the synchronization of tasks. For example, if task $A$ creates task $B$, it is clear that task $B$ cannot start before task $A$. Also, it is possible for a task to explicitly satisfy a dependence of another task. Again, the affected task cannot start before the dependence is satisfied, also causing them to be synchronized. All of these "implicit dependences" are also reflected in the *synchronized-with* relation (and therefore also *happens-before*). So, for example, the OCR API call that creates a task has a *happens-before* relation with the start of that task.

## 3 Task graph visualization

The OCR-Vx collection of OCR runtimes [3] also contains OCR-V1 runtime, which is a single threaded implementation of the OCR specification. It was created to simplify debugging of OCR applications, by providing a simple (single-threaded) environment, but also by trying to check as many error conditions as possible. At the same time, it also generates a list of all OCR operations invoked by the application and their context (e.g., task). It encodes subsets of the *sequenced-before* and *synchronized-with* relations, such that the transitive closure of their union is the *happens-before* relation. With some work, this information could also be reconstructed from the logs of the other OCR-Vx runtimes, but also the XSOCR reference implementation of the OCR specification. We use OCR-V1 because it requires minimal data pre-processing.

It may be tempting to display the complete information available: all tasks, events, and their synchronization. However, even for very simple OCR programs, such graph would be too large and complex. We have discovered that it is better to show a much smaller (and simpler) graph.

The vertices in the graph are the tasks executed by the application. These are labeled by the task ID and the name of the C function which serves as the body of the task. In OCR, the first task always runs the `mainEdt` function. The task always gets ID 10 in OCR-V1. Therefore, the first task in the graph is labeled `10: mainEdt` and it is the ultimate source of all dependences (a source in the usual graph sense), since no other task may start before `mainEdt`. Many OCR programs also contain a sink – a final task that is executed last and shuts down the runtime, but this is not a requirement.

The edges are constructed in two steps. First, we create a set of edges $E$. $E$ contains an edge from task $T1$ to task $T2$ iff $ready(T1)$ *happens-before* $ready(T2)$, where $ready(T)$ is the OCR operation that transfers a task to the ready state. A task becomes ready, once all dependences have been satisfied. This is usually an indirect result of an OCR API call made by a task or the result of another task finishing, if $T$ depends on the completion event of that other task. In these cases, the API call or task completion *happens-before* $ready(T1)$.

It is important to note that an edge from $T1$ to $T2$ does not mean that $T2$ starts after $T1$ has finished. It means $T2$ becomes ready after $T1$. Based on this, $T2$ could theoretically start before $T1$. However, this is never the case, because the only way the *happens-before* relationship can be formed between $ready(T1)$ and $ready(T2)$ is that some operation performed by $T1$ after it has started has a *happens-before* relationship with $ready(T2)$. So, we know that $T2$ may only start after $T1$ has started, but it is possible that $T2$ starts while $T1$ is still running.

The second step in the construction of the set of edges to visualize is performing a transitive reduction on $E$. This means removing all edges that can be obtained using transitivity. Because our graph is a DAG (due to the way it is constructed and because *happens-before* is acyclic), the reduction is unique. Furthermore, no information is lost, since the original edge set $E$ is a transitive closure of itself. This is due to the fact that *happens-before* itself is a transitive closure.

The vertices, their labels, and the reduced edges are then drawn using the dot layout program from the Graphviz suite [4]. Many OCR programs work in iterations, using a set of parallel tasks to perform each iteration. This makes the layered layout used by dot a good fit for the graph. Assuming that the layers are drawn horizontally, this gives us a good distribution of the tasks along the top-down direction.

The vertex placement in the left-right direction can be more problematic. As we have already mentioned earlier, the OCR programs often use long "chains" of tasks, where a chain corresponds to a single thread in a multi-threaded program or a single rank in an MPI program. The work of the thread/rank needs to be split into smaller pieces in

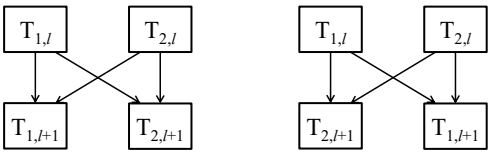


Figure 1: Considering only the graph layout, these two drawings are equivalent. However, the figure on the right may easily confuse the programmer, who naturally assumes that task from the same chain (e.g., $T_{1,l}$ to $T_{1,l+1}$) are aligned, so that the next task in the chain is right below the previous task.

places where synchronization or data exchange is necessary. This is required by the OCR programming model. In most cases, tasks in a chain are serialized – the programmer sets up the dependences in a way that forces the next task to wait until the previous task finishes. In our graph, it means that there is an edge between the tasks. So the task chain forms a path in the graph.

However, there are exceptions. The work of the chain may be further parallelized, corresponding to nested parallelism in OpenMP or to the MPI+X programming model. In OCR, the parallelization is achieved by creating the tasks in a way that allows them to run concurrently, i.e., by not having a dependence on one another. Such tasks are only synchronized with tasks coming earlier and later within the chain. This is not a problem for visualization. The path that corresponds to the chain splits and later joins again.

Another situation where the path is not simple is synchronization (usually data exchange) between the chains. This has the form of edges going between the chains, usually from one layer to the next layer. Imagine the situation where two chains synchronize in both ways. That is, task $T_{1,l}$ in chain 1 at layer $l$ sends data to the other chain's next-layer task $T_{2,l+1}$ and also $T_{2,l}$ sends data to $T_{1,l+1}$. At the same time, the chains are also synchronized, so there are edges connecting $T_{i,l}$ to $T_{i,l+1}$. From the layout algorithm's point of view, $T_{1,l+1}$ and $T_{2,l+1}$ are equivalent, since they both have incoming edges from $T_{1,l}$ and $T_{2,l}$. The two drawings are shown in Figure 1. In some cases, the edge from $T_{i,l}$ to $T_{i,l+1}$ may be missing, making $T_{2,l+1}$ the ideal candidate for a vertex to be drawn directly below $T_{1,l}$. From the programmer's point of view, this is undesirable. He or she probably expects to see the chain as one path, similar to the way a thread or an MPI rank would be drawn on a sequence diagram. Such drawing may be worse from a purely graph-drawing perspective (e.g., it may have a larger crossing number), but it is more useful for the user.

Normally, a task's membership in a chain is not reflected in the OCR model. It is only the programmer's in-

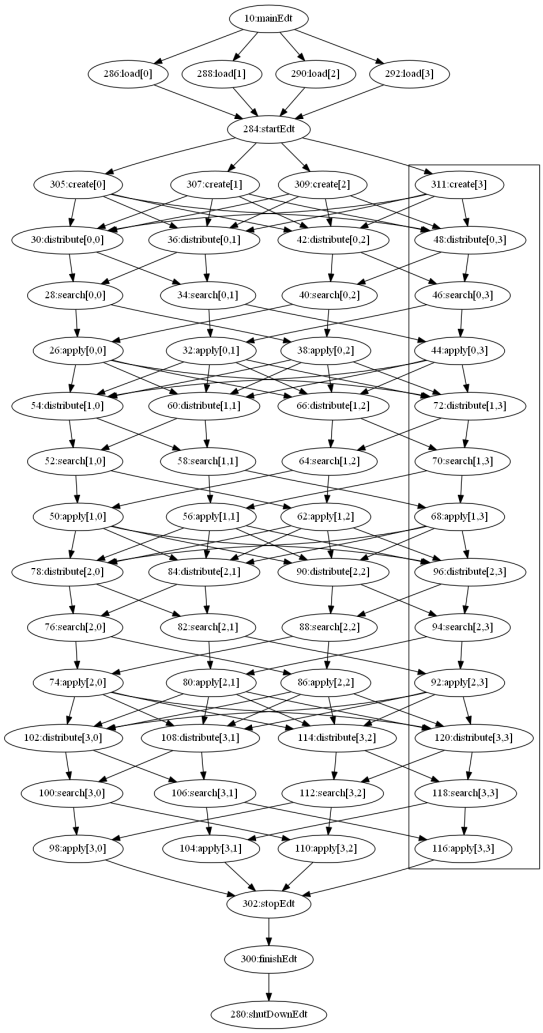


Figure 2: The visualization of tasks in the Graph500 application. Four tasks of each type (distribute, search, and apply) are used per iteration and three iterations were executed. The numbers after task's name are the iteration number and chain number. The cluster for the last chain is drawn with a rectangle around it as a demonstration.

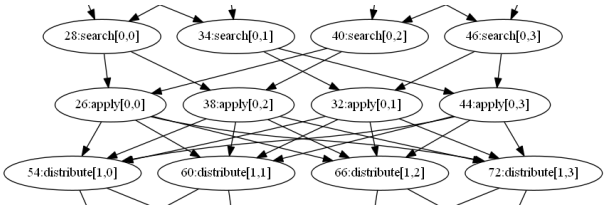


Figure 3: When the chains are not grouped into subgraphs, the apply tasks from chains 1 and 2 are swapped.

terpretation of the computation. However, there are two ways in which this could be expressed. One feature of OCR allows the creation of "labeled tasks". Among other things, this assigns an index to the task. In some applications, members of the chain may be labeled and use the same index. Most of the existing applications don't use this feature. For those, an alternative is a debugging extension available in OCR-Vx, which lets the application programmer assign debugging tags to tasks. These can be numbers or short texts. It is easy to use the numbers to identify the chain and often also the iteration number, as these two are usually the key identifiers in any form of debugging. If the chain number is known, it can be used to put all tasks in a chain into a subgraph, forcing dot to group them also in the left-right direction.

## 4 Examples

In this section, we will present several examples of visualizations produced by our approach. We use three different applications, all of which follow the task chain pattern.

**Graph500** is an implementation of the Graph500 benchmark – a Breadth-First Search (BFS) of a graph, represented by a sparse matrix. The implementation processes the sparse adjacency matrix using a fixed number of chains that communicate by sending data blocks. It corresponds closely to an MPI implementation of the same algorithm. An example visualization is shown in Figure 2. It is an example of a code where dot tends to not align chains horizontally. An example of such misalignment is shown in Figure 3.

**Seismic** is a 2D seismic simulation code, originally distributed as an example with the Intel Threading Building Blocks library. Like Graph500, it consists of communicating chains. However, it also includes parallelism inside a chain, where two (or more) tasks are used to perform computation on the chain's data. Unlike Graph500, where synchronization and communication is realized by sending data blocks, in Seismic the chains only synchronize using events. Data is not explicitly sent from sender to recipient, but the recipient task requests read access to the data. From the OCR point of view, this is a very different approach, but the synchronization pattern is the same and it is visualized the same way – there is an edge connecting the synchronized tasks. Figure 4 shows the task graph with chains grouped into subgraphs. Without the grouping, the chains do not mix like in the Graph500 example, but they are still not as clearly separated (see Figure 5).

**Stencil2D** is a 2D stencil code similar to Seismic, but the communication pattern is different. Seismic iterations are split into multiple phases and the phases only synchronize in certain direction (e.g., only to the next chain, but not the previous). In Stencil2D, each iteration uses just one task per chain for computation. However, the synchronization is performed not directly by the application, but by

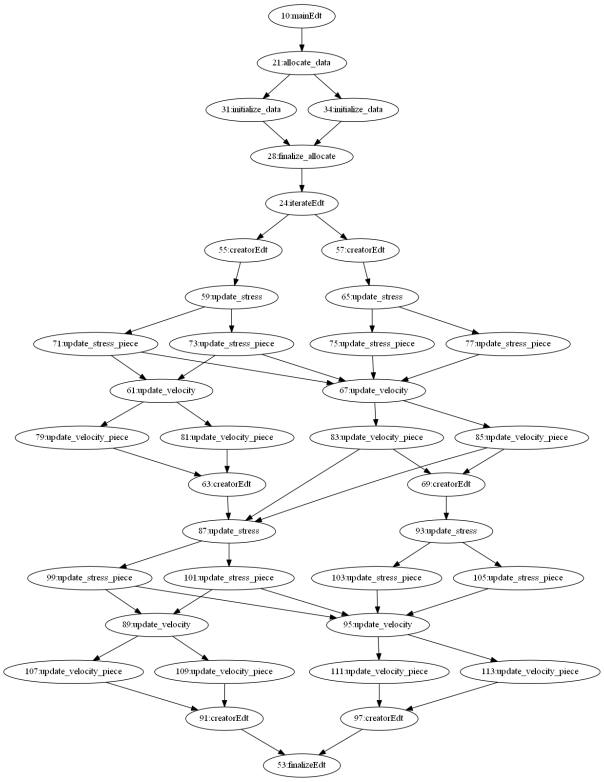


Figure 4: The visualization of tasks in the Seismic application. Note the two parallel `update_stress_piece` tasks per chain. These synchronize not only with the next task in the chain, but also with the next task in the second chain, since that task also uses values they compute. The synchronization goes only one way (left chain to right chain). The other direction happens later, with `update_velocity_piece`.

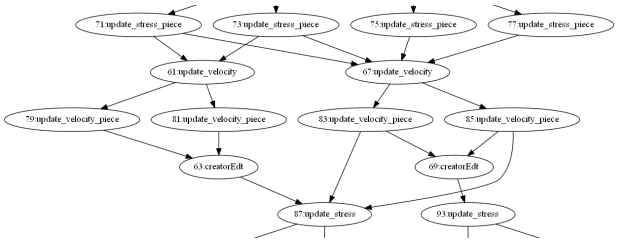


Figure 5: When the chains are not grouped into subgraphs, they are still drawn together, but the distinction between chains is less clear.


Figure 6: The visualization of tasks in the Stencil2D application in top-down layout.

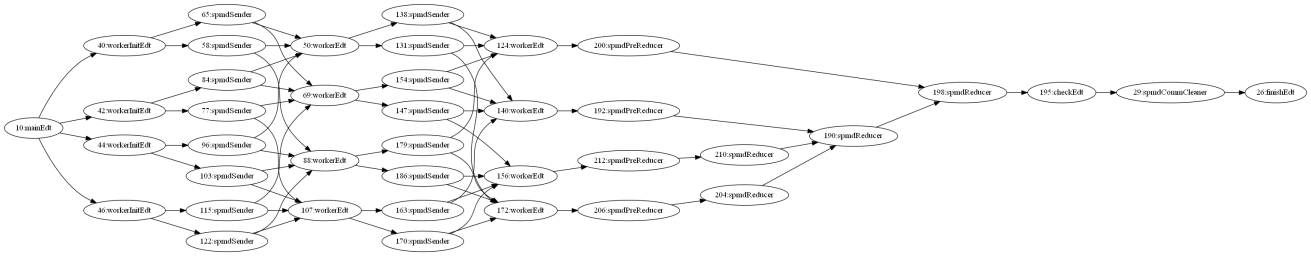

Figure 7: The visualization of tasks in the Stencil2D application in left-right layout.

an SPMD-style communication library, which creates separate `spmdSender` tasks to handle the communication. There are two such tasks per chain after each iteration. Once again, this causes chain grouping to have significant effect. An example visualization (with grouping) is shown in Figure 6. For this example, we have also generated the visualization using left-right layered layout instead of the top-down. The result is more compact, as you can see in Figure 7.

## 5 Related Work

Existing visualization approaches mostly focus on analyzing performance and communication of MPI programs [7]. MPI applications consist of a fixed number of communicating processes. In OCR, there may be a similar structure present (the chains), but some parts of the application could use a more interesting organization of tasks and synchronization, for example a tree-shaped graph of tasks used for a reduction operation. Some applications completely lack the chains, for example a divide-and-conquer algorithm may be implemented by recursive task splitting.

Our goal of our task visualization is to help programmer

find problems with task synchronization. The main goal of visualization tools for MPI codes is to explore and optimize communication patterns and to identify bottlenecks and load imbalance. They often try to not just present the data, but further analyze it and identify the interesting areas [6, 10]. Our goal is more similar to software visualization [5], focusing on what the program does, rather than monitoring and improving performance. In [2], the authors view MPI programs as synchronized tasks and find bugs in the way they are synchronized.

The Application Flowgraph Visualization (AFV) tool created as part of the OCR work performed at Intel within the X-Stack project [1] also uses OCR trace data to visualize the task graph. Unlike our approach, where only tasks are shown and their synchronization is displayed based on the *happens-before* relation between the tasks, the AFV also displays events and directly visualizes the explicit dependences as they are set up by the programmer. We use events and explicit dependences combined with implicit dependences (e.g., task creation) to provide a higher-level view on the task graph. When working on an OCR application, a developer would probably first use our visualization to check the synchronization patterns and, if some of them are wrong, use a tool like the AFV to look at a specific task in greater detail.

## 6 Conclusion and future work

Using data from the OCR runtime, we were able to visualize task graphs to give the application developer a better understanding of how the intender synchronization of the tasks works out in reality. This may aid the difficult process of developing an OCR application. There was already one case where the visualization led to a discovery of a bug in an application.

The main task for the future is tackling larger scale graphs. An OCR application may consist of thousands and even millions of tasks. We plan to explore two different paths. First, an interactive visualization which allows the developer to navigate a larger graph and filter the tasks could allow the ideas described in this paper to be used on a much larger scale. Second, by discovering patterns in the graph, it may be possible to only show the developer the interesting parts of the graph. For example, all iterations apart from the first and the last look exactly the same in all of our three applications. In Seismic, all chains apart from the first and the last look the same. Therefore, only three types of chains and three iterations are sufficient to show all patterns present in an execution of Seismic with any number of chains and iterations.

## Acknowledgment

This work was supported in part by VEGA 1/0684/16.

## References

[1] EDT visualization, X-Stack wiki. `https://xstack.exascale-tech.com/wiki/index.php/EDT_Visualization`.

[2] Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 44:1–44:11, New York, NY, USA, 2009. ACM.

[3] Jiri Dokulil, Martin Sandrieser, and Siegfried Benkner. OCR-Vx - an alternative implementation of the Open Community Runtime. In *International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15. Austin, Texas, November 2015*, November 2015.

[4] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. *Graphviz—Open Source Graph Drawing Tools*, pages 483–484. Springer, Berlin, Heidelberg, 2002.

[5] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, 2005.

[6] Kevin A. Huck and Allen D. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 41–, Washington, DC, USA, 2005. IEEE.

[7] Katherine E Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. State of the art of performance visualization. *EuroVis 2014*, 2014.

[8] Alexey Kukanov and Michael J. Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(04):309–322, November 2007.

[9] T. G. Mattson et al. The Open Community Runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2016.

[10] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with jumpshot. *The International Journal of High Performance Computing Applications*, 13(3):277–288, 1999.