Defining and Searching Communication Patterns in Event Graphs Using the g-Eclipse Trace Viewer Plugin

Thomas Köckerbauer and Dieter Kranzlmüller

Abstract The use of event graphs is a common approach to debug and analyze message passing parallel programs. Although event graphs are very useful for program understanding and debugging, they get confusing and hard to read for programs with complex communication behavior, long runtimes and a large numbers of processes. An approach to ease this problem is to simplify the event graph by marking occurrences of predefined well known communication structures. This allows to quickly identify different regions of activity in the event graph without further inspection. It also helps to identify parts, where certain communication patterns are expected but do not occur due to a bug in the parallel application, in this case the pattern might only match to a certain degree. In this paper we present a language for the description of such communication patterns, which allows to describe the patterns in a way that also covers variations in process numbers and process mappings. Furthermore it demonstrates a pattern matching plugin for the Trace Viewer of g-Eclipse which uses an specialized algorithm for detecting patterns in prerecorded event traces of parallel programs. Based on the presented approach a variety of improvements for the processing and presentation of event graphs are imaginable. The extracted pattern information could be used to optimize the analyzed program or to reduce the contents of the graph to areas of interest, by substituting non interesting parts by placeholders.

1 Introduction

Developing and debugging parallel applications running on HPC machines adds complexity in comparison to sequential programs that needs to be coped with by the application developers. Two additional potential problems that can occur in a parallel program are race conditions [4] and deadlocks [1].

T. Köckerbauer (✉) • D. Kranzlmüller
MNM-Team, Ludwig-Maximilans-Universität München (LMU), Oettingenstraße 67, 80538 Munich, Germany
e-mail: koecker@nm.ifi.lmu.de; Kranzlmueller@ifi.lmu.de

© Springer International Publishing AG 2017

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2016*,
DOI 10.1007/978-3-319-56702-0_2

Race conditions can occur if the result of a program execution is dependant on the timing of the involved (parallel) processes. In message passing systems this can mainly be caused by wrong message orderings. Deadlocks occur if two or more operations depend on each other before they can be finished.

Beside these potential problems the introduced communication between the nodes of the machine can cause, it also requires the application developer to pay attention on the performance impact it causes.

Debugging [7] and profiling [11] tools for message passing parallel programs provide an insight into the inner workings of the programs and aid the developer finding problems or bottlenecks. This is often done by intercepting communication calls of the programs and creating measurements during program runtime using tracing tools that store this information for further analysis and visualization. Using trace analysis tools that analyze the recorded communication steps and the timing of the program it is possible to provide a graphical representation of the recorded data. Event graphs are a common approach to visualize this data.

Event graphs are directed graphs that show the different processes on one axis and occurring events as well as the relations between them on a time scale on the other axis. Processes are represented using lines along the time axis, symbols on those lines mark the events that occur on the processes as vertices in the graph. If those events are related arcs are used to connect the involved events, showing the flow of data and possibly control.

Although these graphical representations make the dataflow and communication easier to understand, they suffer from getting overloaded and confusing with an increasing number of processes, increasing program runtime, and increasing complexity of the communication structure.

Information about patterns occurring in traces can be useful for program understanding, since the pattern information can ease the interpretation of trace data. Knowing if expected patterns occur during program execution can aid in the debugging process. Searching for known bad performing communication patterns can help to improve the program performance by giving hints where to optimize.

2 Pattern Definition

To search for patterns, we first have to define what a pattern is. In the context of this work, we define a set of constraints that a search pattern has to fulfill:

- Constraint 1: Patterns consist of send and receive events, that can blocking as well as non-blocking (the pattern match algorithm does not distinguish between these cases), that are forming a correct MPI communication structure.
  This ensures that there are no communication events between two processes that can be received in a different order than the one they were sent with. This is a hard requirement in the MPI standard (in which it is described as “non-overtaking” messages).

However, MPI has the possibility to create such communication structures by using different tags or communicators for the two messages. Tags and communicators are not covered in this work, but would be a possible extension of the proposed approach. Additional effort to track the used tags and communicators and to handle them separately would be needed in the search algorithm.
- Constraint 2: A pattern covers all processes of the trace, and it is necessary that there is no group of processes that is independent of the other processes. Independent means that the events on one process do not have any happened-before relationship to the events of another process and vice versa.
- Constraint 3: A group of event sequences can only be an occurrence of a pattern if they do not contain any additional events that are not part of the pattern.

3 Pattern Search

In order to search for a pattern it is necessary to provide some sort of reference data that describes the structure of the pattern. This can either be a reference instance of the pattern or some kind of description that captures the properties and structure of the pattern. Since many interesting patterns can have more than one possible manifestation, the use of a single reference pattern might not allow to search all instances of a pattern, whereas a description of the pattern might allow the use of parameters to cover different possible variations. Such variations could for example be a different dimension count, or a different distribution of processes along the dimensions.

In the proposed approach, pattern descriptions are used to generate a set of reference patterns that cover the range of parameters used in the descriptions.

The pattern search process consists of the following steps:

1. Parsing one or more pattern description files which creates an Abstract Syntax Tree (AST) of the description.
2. Executing the ASTs to create reference pattern instances, each containing a reference event sequence per process.
3. Searching of matching event sequences on the individual processes which finds the locations of the sequence matches per process.
4. Calculating a sequence dependency graph which describes the relationship of the sequences in the pattern.
5. Merging of found sequences using the dependency graph to a pattern which spans over all processes.

3.1 Pattern Description

The patterns are described using a new specialized language that allows to formulate the communication structure programmatically in a similar way as done with MPI. The language contains basic control (`for`, `if`, `else`, `do`, `while`) and arithmetic/logic statements that allow to model the communication of a program, statements that allow to describe the topology of the patterns (`description`, `pattern`, `sum`, `product`, `sweep`, `range`, `permutate`, `instanceid`), as well as some built-in functions (`send`, `recv`, `size`, `log2`, `pow`, `factorize`, `sqrt`, `cbrt`). The following example is used to illustrate the basic structure of the pattern description language.

Example: Description of a Torus pattern with a Von Neumann neighborhood

```
pattern "Torus (Von Neumann)"
  sweep(range(dimension, 1, size(factorize(numProcs)));
              product(dimLen[dimension]) == numProcs)
  instanceid("Topology: " dimLen) {
    dist = 1;
    nextDist = 1;
    for(j=0; j<size(dimLen); j=j+1) {
        nextDist = nextDist * dimLen[j];
        dimLowerBound = (myId / nextDist) * nextDist;
        upper = dimLowerBound + ((myId + dist) % nextDist);
        lower = dimLowerBound + ((myId - dist) % nextDist);
        send(upper);
        send(lower);
        recv(upper);
        recv(lower);
        dist = nextDist;
    }
}
```

The pattern description consists of two areas, the pattern instance properties and the pattern structure.

Pattern instance properties (lines 2–4)

Lines 2–3 of the example contain a `sweep` statement and arguments for it. It is used to describe the possible manifestations of the pattern. The `numProcs` variable used in the statement is set to the number of processes in the trace by the interpreter. The `sweep` statement specifies that the following description code (lines 5–17) is executed several times with different parameters which are depending on the arguments of the `sweep` statement:

- The first argument of the `sweep` statement in this example is a `range` statement. It specifies that the following arguments of the `sweep` statement are evaluated with the `dimension` variable set to the values `1` up to the result of `size(factorize(numProcs))` which is the amount of prime factors the

number of processes consists of. This value equals the maximum number of the dimensions along which the nodes in the Torus pattern can be distributed.

- The second argument is the `product` statement. The statement sets the output parameter `dimLen`, which is an array of the length `dimension`. It specifies that all following arguments of the `sweep` statement are evaluated for all products of the array `dimLen` that result in the value of `numProcs`. This means that the following statements are evaluated for all possible distributions of the processes along the specified amount of dimensions.

Line number 4 specifies an instance ID name which can be used to identify the instance generated using the previous statements.

Pattern structure (lines 5–17)

The rest of the pattern description consists of statements to calculate communication partner process IDs (lines 8–11) and `send` and `receive` statements (lines 12–15), that are executed procedurally, similar to an MPI program, but without any computational parts. A detail that is different to an implementation in C is that the `%` operator calculates the modulo instead of the remainder here, so an additional `+ dimLen[j]` is not needed for calculating the value of `lower`.

3.2 Execution of the Description

To search the pattern in a trace file the pattern description is executed in an interpreter to generate a reference trace (Step 2 in Fig. 1). The reference trace data of the individual processes is then used to find the occurrences in the trace data to analyze.

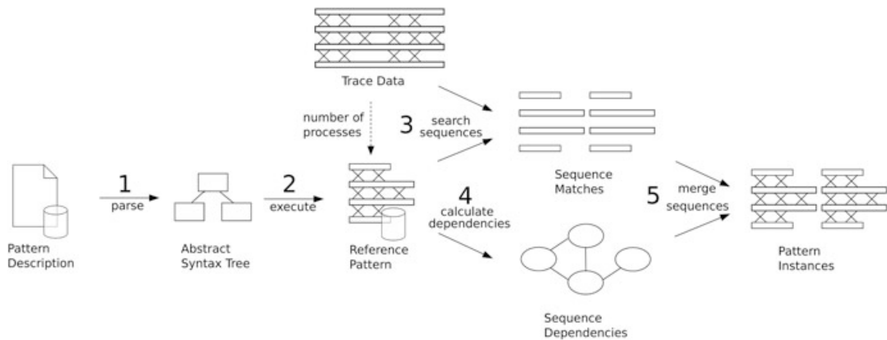

Fig. 1 Pattern description and search steps

3.3 Event Sequence Search

Since the processes of the trace to search in might have another order than in the reference pattern instances (for example due to another communication topology) it is necessary to compare all reference processes with each process in the trace. This prevents us from being able to do a simple comparison of the event types and partner IDs in the reference processes to those in the traced processes, since a possible permutation of the partner IDs has to be taken into account. In addition it is possible that events on reference processes are permuted (for example if there are wildcard receives). This means that it is necessary to make a comparison between the reference process and the trace process which fulfills following requirements:

- The partner numbers may be permuted, since it is possible that pattern instances in the trace have a different process numbering as in the reference pattern. This might for example be due to a different implementation or topology.
- The events of certain ranges in the reference pattern may be permuted. This might be the case since the order of some events might not influence the nature of a pattern. An example for this is the All-to-One pattern where the master process might have a series of wildcard receives. To specify the allowed sequences more precisely additional constraints on the event order are thinkable).

The check used does not compare the reference and the trace directly, instead it compares the amount of occurrences of send and receive events to and from the partners.

These counts can be stored in a triangular matrix A that contains "sequence length + 1" ($k + 1$) rows and columns which represent the send and receive count of the different partner processes in the reference trace or sliding window.

For an example reference trace process as in Eq. (1) the matrix A would look like in Eq. (2).

$$P_1 = \left[S1\ S3\ S4\ S4\ R3\ R1\ R4\ R4 \right] \tag{1}$$

$$A = \begin{bmatrix} 5\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 2\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 1\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0 \\ 0\ 0 \\ 0 \end{bmatrix} \tag{2}$$

$$S = \left[1\ 0\ 1\ 2\ 0\ 0\ 0\ 0 \right] \tag{3}$$

$$R = \left[1\ 0\ 1\ 2\ 0\ 0\ 0\ 0 \right] \tag{4}$$

The vectors S and R [Eqs. (3) and (4)] contain the amount of send or receive events per partner process ID, i.e. 1 process ID with 2 sends and 2 receives (process 3), 2 process IDs with 1 send and 1 receive (processes 0 and 2), and 5 process IDs with 0 sends and 0 receives (processes 1, 4, 5, 6 and 7). Matrix A contains the process counts—the row and column indices represent the corresponding send and receive counts.

These event counts in matrix A can be calculated using a sliding window over the trace. The changes in the event counts caused by shifting the window are used to update a hash.

A substring search with a hash that is updated using a sliding window can be done with the Karp-Rabin algorithm. Since the hash calculation for different permutations of characters in a window in the Karp-Rabin-Algorithm produces different values another hash calculation was used, which results in the same values for those permutations.

The values added and removed in the used hash calculation are triples which contain the amount of sends, the amount of receives and the count of how often this combination of send and receive amounts occurs in the window.

The following paragraphs describe the modifications made to the Karp-Rabin hash calculation.

Modified Karp-Rabin Algorithm

For finding the occurrences of the pattern in a trace we use a modified version of the Karp-Rabin algorithm [6]. The original Karp-Rabin algorithm uses a "rolling hash" to calculate a hash of a moving window in texts. It is mainly used for string-matching, where the reference string and the occurrence have to be identical.

The hash introduced for searching in event graph traces has different requirements:

- Since permutations of event ranges should be allowed the hash value has to be independent from the order of the characters (e.g. "S1S2S3" should produce the same hash value as "S3S2S1").
- Since the processes might be permuted it is not mandatory that the characters in the reference string are identical to those in an occurrence of the pattern. (i.e. "S1S2S3R1" and "S3S2S1R3" can be two different permutations of processes for the same pattern and therefore should result in the same hash value).

These requirements are fulfilled by introducing the following changes to the algorithm:

- Instead of encoding the character values c directly into the hash the number of sends to and receives from a partner process are encoded into a single value per partner process. These values and their number of occurrences in the window is used to update the hash.

- Since the values which are encoded into the hash do not represent positions in the sliding window anymore the factor a is not multiplied to the values anymore.

The hash value of a window can be calculated using Eqs. (5) and (6).

$$f(i,j,b) = b(m+1)^{(i+j(k+1)-\frac{j(j-1)}{2})} \tag{5}$$

$$h = \sum_{i=0}^{k} \sum_{j=0}^{k-i} f(i,j,A_{i,j}) \tag{6}$$

The hash of the window (h) consists of the sum of hashes for the counts (b) of the different send (i) and receive (j) event combinations. The hash for the example in Eq. (2), which has a process count (m) of 8, and a window length (k) of 8, would be:

$$h = f(0,0,5) + f(1,1,2) + f(2,2,1) \tag{7}$$

If the window is moved it is not necessary to recalculate the whole hash using Eq. (6) to get an updated value. Instead, it is possible to subtract the value that was added for the partner process of the new event in the window [see Eq. (9)] and to add an updated value [see Eq. (8)]. The same has to be done for the event that leaves the window.

For every triple (t_1, t_2, t_3) that gets added:

$$h_{new} = h_{old} + f(t_1, t_2, t_3) \tag{8}$$

For every triple (t_1, t_2, t_3) that gets removed:

$$h_{new} = h_{old} - f(t_1, t_2, t_3) \tag{9}$$

3.4 Sequence Dependency Graph

The Sequence Dependency Graph is needed to merge the sequences found in the Event Sequence Search in the final step of the pattern search. The graph is built from the reference traces that were created using the pattern descriptions (Fig. 2).

The graph contains nodes that represent the event sequences of the individual processes of the reference traces. The edges represent the relations to the other processes of the reference patterns. The information in the nodes describes the properties of the sequences they represent. The format of the information in the nodes is as follows:

- The first line contains the amount of send and receive events to partner processes and the amount of partner processes that have those individual combinations of send and receive events.

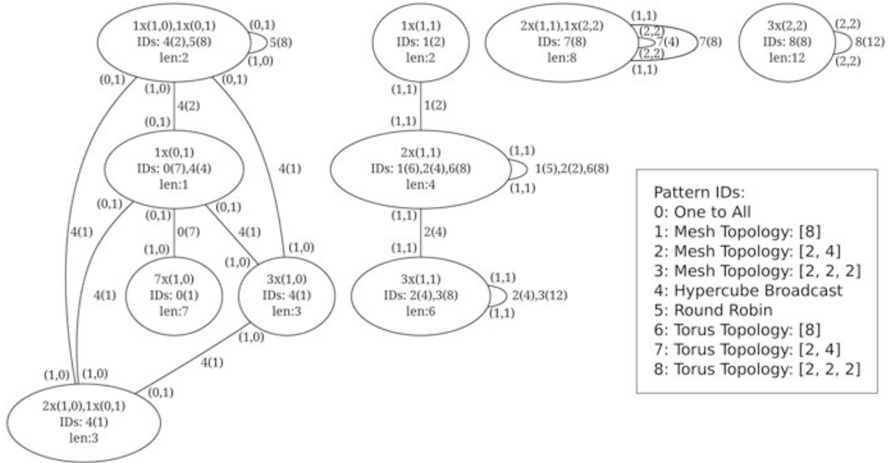

Fig. 2 Example of a sequence dependency graph

Format: <amount of partner processes>x(<amount of send events>, <amount of receive events>)
Example: The line "2x(1,1),1x(2,2)" means that there are two partner processes that this sequence is connected to with one send and one receive event each, and that there is one partner process that this sequence is connected with two sends and two receives.

- The second line lists the pattern instance IDs of the reference patterns that contain the sequence and also the count of occurrences in the individual reference patterns.
  Format: <pattern instance ID>(<number of processes in the pattern instance that have this sequence>)
  Example: The line "IDs: 7(8)" means that this event sequence is only part of reference pattern instance 7, which is a bidirectional communication with a Von Neumann neighborhood on a two-dimensional Torus topology. The sequence is contained in 8 processes, i.e. in all processes, of this pattern instance.
- The third line contains the length of the sequence. The sum of the events in the first line equals the length specified here. This information is redundant, but is included for better readability, and also for easier evaluation of the constraints.
  Example: The line "len: 8" means that this sequence has a length of 8 events, which are the 4 send and the 4 receive events described in the first line.

The labels on the edges of the graph contain information about the relationship of the sequences the nodes represent to the sequences on neighboring processes. Their format is as follows:

- The labels at the ends of an edge specify for which combination of amounts of send and receive events this edge connects to a sequence of a partner process.

The number of the send and receive events on one side of the edge are always the swapped numbers of the other side of the edge.
Example: "(2,2)" means that this edge describes a connection of a sequence to another sequence via two send and two receive events.

- The label at the middle of the edge specifies which reference patterns contain communication between the two connected reference sequences and the number of occurrences of this connection in the individual reference patterns.
Format: <pattern instance ID>(<number of connections between the two sequences in the pattern instance>)
Example: "7(4)" means that only reference pattern instance 7 contains connections of the sequences the edge connects with the send and receive counts specified at the ends of the edge, and that this pattern contains 4 of those connections.

3.5 *Merge of Potential Matches to Pattern Instances*

The event sequences found by the modified Karp-Rabin algorithm can potentially be part of pattern instances. Merging the sequences found on the different processes to instances of patterns is a Constraint Satisfaction Problem (CSP). In this instance for searching patterns the CSP $P = \langle X, D, C\rangle$ is defined as follows:

$X\ldots$ set of variables, one variable per process of the trace
$D\ldots$ domain of values, where every value is a tuple of a found matching sequence containing its sequence ID (every node in the sequence dependency graph gets an unique ID) and index
$C\ldots$ set of constraints defining the patterns via the relationships between the sequences

3.5.1 Constraints for Searching Patterns in Event Graphs

We already defined the set of variables X, which contains one variable per process, and the domain of values D, which contains the results of the sequence search on process level. This subsection contains the set of constraints C needed to find pattern instances on an event graph using a CSP.

The evaluation of the constraints has different runtime complexities. It might not be necessary to evaluate all of them if one of them fails. The following list of constraints is ordered so that checks with low complexity that might allow to skip checking the remaining constraints happen before the more complex ones, which improves runtime in non-worst case scenarios.

1. The variable to assign has to be in the same connected component of the sequence dependency graph as the already assigned variables.

2. The value of the variable to assign has to be within the upper and lower bounds given by already assigned variables.
3. The values of already assigned variables have to be within the ranges for partner processes required by the value to assign to the new variable.
4. The intersection of the pattern instance IDs of already assigned variables and the variable to assign must not be empty. This is only necessary when searching for more than one reference pattern instance in the same run to avoid a result that consists of a mixture of sequences from different patterns, but does not form a pattern instance itself.
5. The number of variables that already have the same value as the one that should be assigned to the new variable must be lower than the maximum number of occurrences for this value for at least one pattern instance ID that is member of the intersection in the previous constraint.
6. The edge count between the value to assign in the new variable and the already assigned values on other variables must be lower or equal the maximum number of edge occurrences for at least one pattern instance ID that is member of the intersection in constraint (4) and also fulfills the requirements of constraint (5).

3.5.2 Dynamic Backtracking

The approach to solve the CSP that was chosen in this work, and was also implemented for the experimental evaluation, is dynamic backtracking [3]. This algorithm was chosen due to several properties that can be beneficial when used for finding patterns in event graphs.

- It identifies which other variables conflict with the assignment of the next variable so that the amount of trashing is reduced, i.e. there is less exploration of search space that can not lead to a solution because an early assignment avoids it.
- When a conflict is found in dynamic backtracking it tries to replace the “culprits” that caused the conflict, trying to leave the potential subset of the solution that already was found intact. Such a behavior can be beneficial for our use since in typical patterns there can be groups of processes that are less connected to other groups of processes, or might also appear in different but similar pattern instances.
- The dynamic backtracking algorithm can be extended to allow the dynamic expansion of the search space. It can start with a small search space, and expand it whenever the assignment of a potential result value for a variable creates dependencies on the values of the other variables that can not be fulfilled with the current search space. This ensures that the search space of the unassigned variables always contains enough possible values to cover all potential assignments that may lead to a found pattern instance. The same applies for shrinking the search space. For our case the algorithm can be extended to shrink the search space to leave out parts that can not be reached anymore.

Since related events in a trace are usually close together on the time scale this can reduce the search space vastly.

Several modifications were made to the dynamic backtracking algorithm to optimize it to the problem of finding pattern instances in an event graph. There are two groups of modifications that were performed: Modifications that change the behavior of the algorithm and modifications that influence the runtime of the algorithm.

Among the changes, one change was made that modifies the behavior of the algorithm substantially:

- Termination Criteria: The search does not terminate after a pattern instance was found.

There are several changes to the dynamic backtracking algorithm that do not influence its function, but can drastically influence its runtime, depending on the structure of the patterns to search and the trace to search in.

- Immediate Backtracking if the already merged processes do not have any more neighbors.

  Since the pattern definition does only allow pattern descriptions in which all processes are connected (see constraint 2 in Sect. 2) it is possible to backtrack earlier than when one of the constraints of the CSP (see Sect. 3.5.1) is not fulfilled in some cases. If the set of already assigned variables and the set of neighbors of the already assigned variables is identical and does not contain all variables then the sequences that are represented by the assignments of those variables form a group that is not connected to any other process and therefore can not be part of an allowed pattern.
- Dynamic sizing of search space: Expand if necessary, reduce if possible.

  The original dynamic backtracking algorithm uses the whole search space from start to end, containing all possible values—the complete domain of every variable—in its checks for conflicts and possible assignments.

  The modified algorithm starts with a minimal search space. It only contains the first value of the domain of the variable that was chosen to start with, and expands the search space when necessary. This is possible due to the structure of the event graphs. The events on a specific process at the begin of a trace are very likely connected to events at the begin of other processes. Very likely there is a, in comparison to the size of the whole trace, small window of the trace which can contain potential candidates for partner events.

  If it is not possible to assign a variable anymore in the current search space, and usually a backtracking step would be performed, the modified algorithm tries to expand the search space so that values that are potential candidates for assignment, but were not in the search space before will be added to the search space.

  In a similar way the search space is shrinked again if parts of it are identified as not being reachable anymore, or if pattern instances are found.
- Variable assignment order: Assign neighbors of already assigned variables.

The assignment order of the variables in the original algorithm is not defined. Defining this order to prefer variables for processes that are neighbors of already assigned variables can improve the performance since it allows to check constraints already after the first assignment.
- Sequence match order: The order in which the sequence matches are verified vastly influences the amount of backtracks in some cases.

4 g-Eclipse Trace Viewer Pattern Search Plugin

This section shows the main features and the usage of the prototype implementation, which was used as a proof-of-concept.

The prototype implementation of the pattern description editor, pattern search and visualization is building on the trace viewer functionality [7] of the g-Eclipse [8] project, and extends its functionality by providing a plugin to these tools.

The plugin for the g-Eclipse trace viewer consists of several components for the different tasks in the process of finding the patterns. This section gives an overview of the major components provided by the plugin, as well as the ones provided by the g-Eclipse trace viewer that make up the core functionality.

- **Trace viewer** The g-Eclipse trace viewer plugin is the central component, the pattern search plugin is building on. It provides the basic functionality for accessing trace data and visualizing it. The trace viewer can be extended using the Eclipse extension point mechanism [5] and among others provides the following extension points that are used by the pattern search plugin:
  - **Actions on trace, process and event level** It is possible to add actions that can be triggered for the whole trace or selected events or processes to the trace visualization. These actions are provided using the context menu of the trace viewer. The pattern search can be triggered using such an action which starts an Eclipse job performing the search.
  - **Markers** The color and shape of the events displayed in the trace viewer can be altered using marker plugins. The pattern marker is used to change the background color of events that are inside the found pattern instances.
  - **Trace readers** The pattern search uses the trace reader functionality provided by the g-Eclipse trace viewer. The trace viewer offers a common interface for accessing the supported trace file formats. Currently these are the NOPE format and the OTF format.
- **Pattern description interpreter** The implementation of the pattern description language consists of a scanner and a parser generated from an attributed grammar using the COCO/R compiler generator for LL(k) grammars [10] and an interpreter basing on those.
- **Pattern description editor** The pattern description editor is shown in Fig. 3. The screenshot in the figure contains an occurrence of the error marker of the

Fig. 3 Pattern editor showing a pattern description with a syntax error

Fig. 4 The autocompletion feature of the editor can complete keywords, function names and variable names

editor, which in case of this example is for a missing semicolon in the pattern description. The errors displayed by this marker are gathered from the parser that is also used by the interpreter which evaluates the pattern descriptions.

The screenshot also shows the syntax highlighting feature of the editor. Syntax highlighting is done using a simpler rule based scanner provided by Eclipse. This scanner contains rules for detecting keywords, built-in function names, operators, strings and comments.

The editor also features autocompletion of keywords, built-in functions and variable names which are as well gathered using the COCO/R parser. The autocompletion feature is shown in Fig. 4.

- **Pattern selection view** The pattern selection view, shown on the left of Fig. 5, is an Eclipse view that allows to specify which patterns should be searched for by allowing to select from the available pattern descriptions.

  A pattern description can be used to generate and display the reference patterns that it describes. By using the context menu on an entry in the pattern selection view it is possible to generate reference patterns for the selected description. After entering the number of processes for the reference pattern

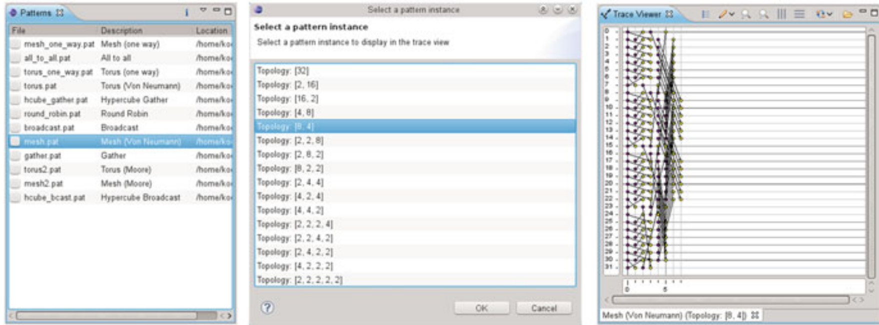

Fig. 5 Pattern selection view (*left*), dialog for selection of a reference pattern instance (*middle*), and the corresponding reference pattern instance (*right*)

the instance selection dialog is shown. The screenshot in the middle of Fig. 5 shows the instance selection dialog for a mesh pattern with a "Von Neumann" communication topology with 32 processes. This dialog allows to select a reference pattern instance to display. On the right side of the figure a reference pattern instance generated using this description is shown.

- **Pattern search** The plugin also implements the search algorithm described in Sect. 3 for finding patterns in the trace data provided by the trace viewer plugin of g-Eclipse.

5 Examples

An experimental evaluation using well known parallel benchmark programs was performed. In this section some observations in traces of well known benchmark codes are shown.

The wavefront propagation performed by the Sweep3D benchmark can easily be recognized in the trace visualization as seen in Fig. 6. There are two alternating directions in which the propagation takes place. The screenshot shows a change in direction of the wavefront propagation, which consists of several instances of an unidirectional mesh pattern with Von Neumann neighborhood. In this screenshot eight alternating colors were used in the pattern marker so that the displacement of the individual pattern instances can be seen better.

An interesting observation in the SMG2000 traces is that there are pattern instances, as described above for lower process counts, that are interrupted by communication that only takes place on a subset of the processes as shown in Fig. 7.

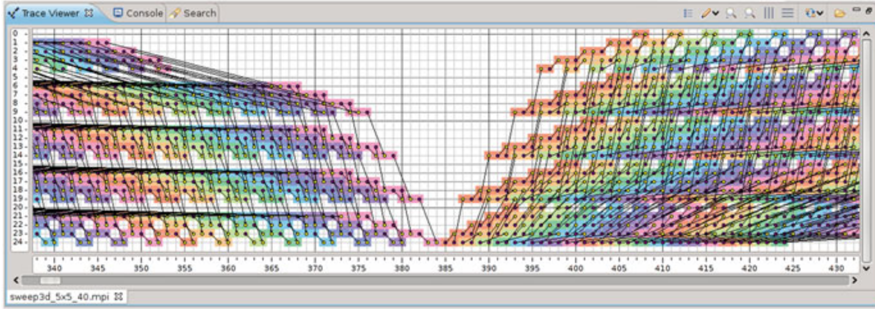

Fig. 6 Sweep 3D trace with wavefront propagation in two different directions

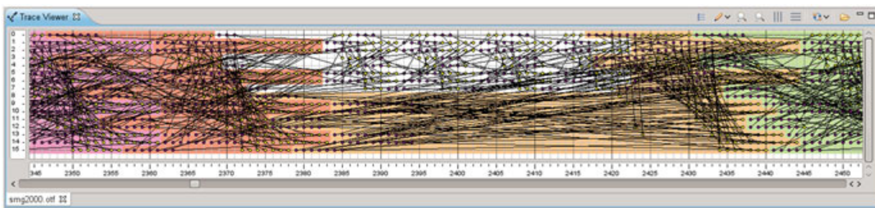

Fig. 7 Part of an SMG2000 trace which has mesh patterns interrupted by other communication

6 Future Work

To further improve the performance and applicability of the algorithm, several potential refinements can be investigated:

- **Additional constraints on event sequences**—The addition of constraints within the event sequences on the individual processes could further reduce the amount of detectable event sequences that are not part of a pattern, and therefore further reduce the search space that needs to be covered by the backtracking algorithm. A simple example for such an additional constraint for a bidirectional mesh pattern could be specified as follows: Every sequence that can be part a pattern instance needs to start with a send event.
- **Parallelization of the pattern search algorithm**— Another possibility to reduce the search duration could be the parallelization of the search process. While the parallelization of the pre-filtering step on process level is easy to implement, the parallelization of the dynamic backtracking algorithm is non-trivial. There has already been research on this topic [2, 12], but still the feasibility of applying these approaches onto this adaptation of the algorithm needs to be verified.
- **Improved visualization of the search result**—In the prototype implementation the results of the pattern search are visualized using a pattern marker that modifies the background color of the sections in the traces that belong to pattern

instances. More sophisticated ways to visualize this information are thinkable. As a simple improvement the pattern instances could be replaced by placeholders that make the trace visualization more compact similar to the approach presented in [9]. Since identical pattern instances are often recurring several times, they could be replaced by the display of an instance count only. This way the main part of the visualization is dedicated to non-repeating information and sections that do not match any pattern description.

- **Support for additional MPI features**—The approach as it is described in this work has some constraints on the use of MPI communicators and tags. This limitation is, however, not a mandatory restriction imposed by the basic principle of the approach itself. Adding support for these features could be archived by extending the search algorithm without changing its basic properties.

In conclusion, the problem of scalability still continues to increase with more and more supercomputers of more than a million CPU cores and corresponding applications. The solution presented in the work represents a possible solution, which needs to be improved further by additional abstraction and automatization.

7 Conclusion

After providing a definition of patterns in the context of this work and discussing the motivation for it the basic steps of the pattern description and search process are introduced.

A pattern description language that combines procedural description of the patterns with language constructs that allow to describe the topology of the patterns is proposed and discussed.

Different approaches on filtering the event sequences on a process level for reducing the search space, which is part of the proposed search process, are discussed with consideration of the requirements for this use case. A modified version of the Karp-Rabin algorithm is proposed for this task, which can efficiently detect candidates for being part of pattern instances.

Merging the found event sequences into pattern instances, which is also part of the search process, is modeled as a constraint satisfaction problem (CSP). A set of constraints for this CSP is defined and a "sequence dependency graph" that aids the evaluation of those constraints is introduced. Modifications to the dynamic backtracking algorithm are proposed to take advantage of properties of the event graph.

References

1. Barbosa, V.C.: Strategies for the prevention of communication deadlocks in distributed parallel programs. IEEE Trans. Softw. Eng. **16**(11), 1311–1316 (1990). doi:10.1109/32.60319
2. Bessière, C., Maestre, A., Meseguer, P.: Distributed dynamic backtracking. In: International Joint Conference on AI Workshop on Distributed Constraint Reasoning (2001)
3. Ginsberg, M.L.: Dynamic backtracking. J. Artif. Intell. Res. **1**, 25–46 (1993)
4. Helmbold, D.P., McDowell, C.E.: A taxonomy of race conditions. J. Parallel Distrib. Comput. **33**(2), 159–164 (1996)
5. Hennig, M., Seeberger, H.: Einführung in den "Extension Point"-Mechanismus von Eclipse. JavaSPEKTRUM **1**, 19–24 (2008)
6. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987). http://www.research.ibm.com/journal/rd/312/ibmrd3102P.pdf
7. Klausecker, C., Köckerbauer, T., Preissl, R., Kranzlmüller, D.: Debugging MPI Programs on the Grid using g-Eclipse. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, pp. 35–45. HLRS, Springer, Stuttgart (2008). doi:http://dx.doi.org/10.1007/978-3-540-68564-7_3
8. Kornmayer, H., Stümpert, M., Knauer, M., Wolniewicz, P.: g-Eclipse - an integrated workbench tool for grid application users, grid operators and grid application developers. In: Cracow Grid Workshop '06, Cracow (2006)
9. Kranzlmüller, D., Grabner, S., Volkert, J.: Event graph visualization for debugging large applications. In: SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, pp. 108–117. ACM, New York (1996). doi:http://doi.acm.org/10.1145/238020.238054
10. Mössenböck, H.: A generator for production quality compilers. In: CC '90: Proceedings of the Third International Workshop on Compiler Compilers, pp. 42–55. Springer, New York (1991). doi:http://dx.doi.org/10.1007/3-540-53669-8_73
11. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: visualization and analysis of MPI resources. Supercomputer **12**(1), 69–80 (1996). doi:http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.92.2371
12. Zivan, R., Meisels, A.: Concurrent dynamic backtracking for distributed CSPs. In: Proceedings Constraint Programming, pp. 782–787 (2004). http://jmvidal.cse.sc.edu/library/zivan04a.pdf