Inspection of I/O Operations from System Call Traces using Directly-Follows-Graph

Aravind Sankaran*, Ilya Zhukov†, and Wolfgang Frings‡
Jülich Supercomputing Center
Forschungszentrum Jülich, Germany
{*a.sankaran, †i.zhukov, ‡w.frings}@fz-juelich.de

Paolo Bientinesi
Department of Computer Science
Umeå Universitet, Sweden
pauldj@cs.umu.se

Abstract—We aim to identify the differences in Input/Output (I/O) behavior between multiple user programs through the inspection of system calls (i.e., requests made to the operating system). A typical program issues a large number of I/O requests to the operating system, thereby making the process of inspection challenging. In this paper, we address this challenge by presenting a methodology to synthesize I/O system call traces into a specific type of directed graph, known as the Directly-Follows-Graph (DFG). Based on the DFG, we present a technique to compare the traces from multiple programs or different configurations of the same program, such that it is possible to identify the differences in the I/O behavior. We apply our methodology to the IOR benchmark, and compare the contentions for file accesses when the benchmark is run with different options for file output and software interface.

Index Terms—High-Performance Computing, Performance Analysis, Input/Output, strace, Directly-Follows Graph, Process Mining

I. INTRODUCTION

The efficiency of a computer program is often inhibited by contention for system resources. This issue is particularly evident in programs that perform significant Input/Output (I/O) operations on storage systems. In programs executed by users, requests to access system resources happen through the operating system. In this paper, we aim to analyze arbitrary user programs, without modifying them, in terms of I/O requests made to the operating system.

A user program communicates with the operating system by issuing *system calls*. The record of the sequence of system calls made by the user program during its execution is referred to as the *system trace* of that program. We analyze the contentions for system resources in programs using the system call traces. We consider the I/O-related system calls from the traces of user programs, particularly the system calls on LINUX-based operating systems that are implemented based on the interfaces defined in the C standard library (*libc*) under the headers *unistd.h* and *sys/uio.h*. Performing I/O accesses directly by using the *libc* calls is time-consuming and one requires an understanding of the system-specific programming requirements; for example, when porting the code to a different architecture or a Linux variant, one may have to correctly invoke the *libc* calls available for that system configuration to make the I/O accesses efficient. Therefore, users typically rely on standard I/O interfaces such as STDIO that manages the *libc* calls under the hood[1]. Moreover, when it comes to parallelization of I/O accesses, users rely on more high-level interfaces (such as MPI-IO) and libraries (such as HDF5 [1] and Parallel NetCDF [2]). The high-level interfaces are optimized for ease of use, but when to comes to achieving optimal efficiency, it has been noted that these interfaces should be used with a configuration that is tuned to a setting that is optimal for the concerned application [3], [4]. In the process of tuning the I/O performance of a program, users must decide not only on the choice of the interface and its configuration, but also on several other parameters that should be set according to requirements of the application. For example, it is important to determine the pattern of file output: whether each process should access its own file or if all processes should access a single shared file. To understand the performance impacts of the interface and configuration choices, users typically rely on I/O profiling and tracing tools to analyze their programs.

Several tools exist that intercept the I/O calls from *libc* to extract information and use it for analyzing and improving the I/O behavior of programs [5]–[12] However, it is still challenging to perform analyses that spots *differences* in I/O behavior between different configurations of a program in terms of contentions for system resources. This is mainly because each program makes a vast number of system calls through *libc*, and translating a large volume of information from the system calls into a representation that facilitates precise identification of differences between the programs is not straightforward. In this paper, we consider the problem of *synthesis* of the data from system traces, i.e., combining the information in the data to extract precise insights for understanding the I/O contentions caused by a program for system resources. We do not introduce yet another tool; instead, we present a methodology to synthesize the trace data into a Directly-Follows-Graph [13] that depicts patterns of I/O system calls, which then facilitates the comparative analysis of programs in terms of requests made to the operating system.

---

[1]In most user programs, such as those written in FORTRAN or Python, the *libc* calls are encapsulated within the software stack.

The contributions of this work are the following:

- We present the theory behind the synthesis of the Directly-Follows Graph from large amounts of information in the I/O system call traces.
- We present a methodology to color the Directly-Follows-Graph, which facilitates the comparison of patterns of I/O system call accesses between several programs or multiple process running simultaneously in a program.
- We apply our methodology to the IOR benchmark, and infer the differences in file access contention when (1) several processes access a single shared file versus all the processes access their own individual files, and (2) default read and write calls in IOR are replaced with the MPI-IO counter-parts.

Organization: In Sec. II, we review the related works. In Sec. III, we describe the format of the trace data used as input. In Sec IV., we present the methodology to translate the trace data into Directly-Follows Graph and explain the technique for comparative analysis. In Sec V, we conduct experiments and analyse the overheads, and finally in Sec. VI, we summarize the findings and draw conclusions.

II. RELATED WORKS

In the existing tools for analyzing the I/O behavior of programs [5]–[12], we identify the following two common steps: (1) instrumentation of the program, which involves intercepting or interrupting calls made by the program from one or more layers of software interfaces to record relevant information, and (2) synthesis of the recorded information after the execution of the program, which involves a calculation of statistical metrics and putting them together in visualizations such as histograms, timeline plots, Gantt charts, heat maps, and more.

Instrumentation. Most of the existing tools record information by intercepting I/O calls of the standard C library, and considerable work has been done to optimize the collection and storage of this information in formats suitable for HPC workloads. For example, Darshan's instrumentation is lightweight, non-intrusive and designed for 24x7 monitoring of HPC applications [14], while the Score-P measurement system [8], which collects traces in Open Trace Format Version 2 (OTF2) [15] and profiles in CUBE4 [16] formats, is tailored for scalable yet detailed program analysis. The traces and profiles generated by Score-P can be used and processed by various performance analysis tools such as Scalasca [12], TAU [7], Vampir [9], and CubeGUI [16]. In this work, we do not rely on the instrumentation process of any specific tool, but rather focus on introducing a method to synthesize the instrumented data. To this end, we utilize the raw system call traces recorded by *strace* [17]—a Linux utility that uses the *ptrace* system call under the hood to instrument arbitrary commands or programs in the user environment without requiring code modification. The methodology by itself does not depend on strace and can be applied over data instrumented by one of the other existing tools.

Synthesis. Extensive efforts have been made to develop different methods for synthesizing and visualizing measurement data. For example, the Vampir visualization environment transforms measurements into a variety of graphical views, such as timeline plots and heat maps, with interactive elements [9]. The result of synthesis could also be a performance report that brings together information from various visualizations and statistical metrics to provide a comprehensive overview of the program. For instance, Darshan provides interfaces to synthesize their log files as static PDF reports providing an overview of the I/O performance of the program [5]. PyDarshan is a wrapper around Darshan that facilitates the generation of interactive HTML reports [18]. Drishti is a tool that synthesizes traces from DXT-Tracer to generate a variety of interactive plots [6]. To the best of our knowledge, we observe a lack of detailed exploration of *dependency-graph-based* modeling in the synthesis of I/O related instrumented data.

Dependency-graph-based modeling has previously been used to reconstruct call-graphs, detect potential parallel regions in sequential programs (e.g., in DiscoPoP [19], Parwiz [20]), develop models to enable smart scheduling of HPC jobs [21], etc. In this work, present a technique to synthesize the traces into a specific form of dependency graph known as the Directly-Follows Graph, as defined in Definition 4 in [13]. We discuss the computational complexities in synthesizing the graph, and then use it for comparative analysis of I/O operations across one or more programs.

III. THE INPUT DATA

We consider setups where every process involved in the execution of a program independently records the I/O system calls, and each system call captures at least the path of the accessed file, the start timestamp and the duration between start and return of the call. In this context, processes refer to tasks executing an arbitrary command, where those tasks can be co-located on a single core, distributed across cores within a host machine, or even span multiple hosts in a parallel execution. In this section, we describe the trace records generated by strace, which are then used as inputs for our methodology.

Tracing with strace

One could generate the traces of system calls of a command by prefixing strace to the command that needs to be traced. In Figure 1, we show an example of running strace with `ls` and `ls -l`. Each command is run simultaneously by three MPI processes (specified as `srun -n 3`), and each MPI process records its traces in a separate file[2] specified by the option `- o`. In order to uniquely identify each trace file, we follow a naming convention which is a combination

---

[2]For applications with a large number of processes, having a large number of files could lead to meta-data performance issues [22]. Therefore, after recording the traces, it is recommended that the relevant data (described in the remainder of this section) from individual trace files are parsed and combined efficiently into a suitable data format (such as a single HDF5 file).
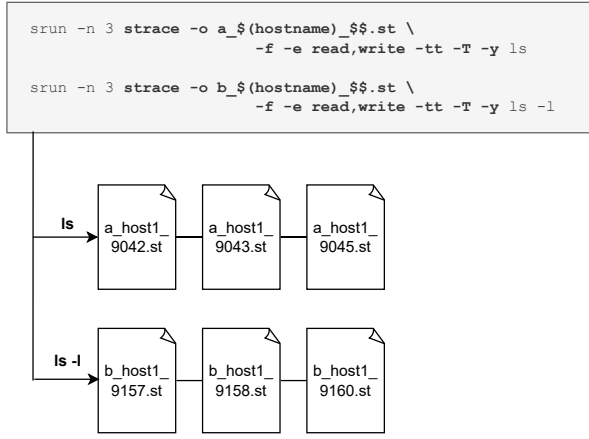
```
srun -n 3 strace -o a_$(hostname)_$$.st \
                       -f -e read,write -tt -T -y ls

srun -n 3 strace -o b_$(hostname)_$$.st \
                       -f -e read,write -tt -T -y ls -l
```

Fig. 1: The commands for tracing `ls` and `ls -l` with strace, each executed on three MPI processes, generating one trace file for each process.

of name of the host machine, identifier of the MPI process (*rid*) and identifier of the command (*cid*). In Linux, one could obtain the name of the host machine from the shell variable `hostname`. Each MPI process is represented by the identifier of the launching process, which is obtained from the variable `$$`. In our example, let the commands `ls` and `ls -l` be identified with *cid*='a' and *cid*='b' respectively. According to our example, the identifiers for each of the six trace files are shown in Figure 1. The ASCII contents of the trace files *a_host1_9042.st* (*rid=9042* for the command `ls`) and *b_host1_9157.st* (*rid=9157* for the command `ls -l`) are shown in Figure 2a and Figure 2b respectively.

We parse the following information from each line of each trace file:

1) **The process identifier** (*pid*). The identifier of the process executing the system call is recorded by specifying the option `-f`. Note that *pid* would be different from *rid* if the MPI process forks a child process to execute the command; in our example, *rid* and *pid* are different. In general, for the case of Simultaneous Multi-Threading or shared memory multi-threaded applications (such as OpenMP), each MPI process could spawn more than one child processes. In the considered example, however, each MPI process is associated with only one child process.

2) **The system call name** (*call*). The list of system calls to be traced is specified using the `-e` option. In our example, for simplicity, we trace only the `read` and `write` system calls. However, strace can trace additional I/O system calls, and they all can be included as input to our methodology.

3) **The timestamp** (*start*). The wall clock time at the start of each system call, including microseconds, is recorded by specifying the option `-tt`. For MPI processes executing in different host machines, we do *not* require the system clocks to be synchronized.

4) **The duration** (*dur*). The duration of a system call, which is the time between its start and return of the call, is recorded using the `-T` option.

5) **The file path** (*fp*). The path of the accessed file is indicated as the first argument in the system call's signature. This is recorded using the `-y` option.

6) **The transfer size** (*size*). The number of bytes transferred from the page table is indicated as the return value of the call (i.e., the number after the = sign). This information is parsed only for the variants of read and write system calls (and not for other I/O system calls such as `lseek`, `openat`, etc.). Note that the number of bytes requested, which is indicated as the last argument in the system call's signature may differ from the actual number of bytes transferred.

Note that for read operations on regular files, if the requested data is not in the page table, the read system call internally issues a request to obtain the data from the storage system. Read access from the storage system can be traced by recording accesses to block-device files, which have a different directory path from regular files. For write operations, the system call returns as soon as the page table is updated without the guarantee of the data update being completed in the storage system; for the guarantee, one should trace the `fsync` system calls.

If a system call is interrupted, it contains the keyword `ERESTARTSYS`, and we ignore these calls. If a call is being executed and meanwhile another one is being called from a different process, then strace will preserve the order of those events and mark the ongoing call with the keyword `<unfinished ...>`. When the call returns, it will be marked with the keyword `resumed>`. In such cases, the duration of the call and the transfer size are indicated only in the resumed record; an example of such a log in shown in Figure 2c. The unfinished and the resumed records are matched using the *pid*, and merged into a single record.

Thus, the trace files are processed according to the requirements mentioned in this section and used as input to our methodology.

IV. THE DFG SYNTHESIS

We now explain the methodology for synthesizing the information parsed from a set of trace files into a Directly-Follows Graph (DFG). Our input data can be likened to an *event log* described in the field of process mining [23]. Formalizing the input data as an event log enables us to symbolically describe the construction of the DFG and introduce methods for comparing the I/O operations. To this end, we define the following terminologies:

Event. Every record of system call is considered as a unique *event*. An event e consists of the attributes described in Sec. III:

$$e = [cid, host, rid, pid, call, start, dur, fp, size] \quad (1)$$

The attributes cid, $host$ and rid are inferred from the name of the trace file, and the other attributes are parsed from the records of the corresponding trace file. Let $\mathcal{E} = \{e_1, \ldots, e_m\}$ be the set of all possible events in all the trace files under consideration. Then, $\nexists e_1, e_2 \in \mathcal{E}$ such that $e_1 = e_2$; i.e., no

```
pid        Timestamp      Sys Call                    The file path                                    Bytes    Transfer   Duration
                                                                                                       request  size

9054    08:55:54.153994 read(3</usr/lib/x86_64-linux-gnu/libselinux.so.1>, ..., 832) = 832 <0.000203>
9054    08:55:54.156640 read(3</usr/lib/x86_64-linux-gnu/libc.so.6>, ..., 832) = 832 <0.000079>
9054    08:55:54.159294 read(3</usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.10.4>, ..., 832) = 832 <0.000087>
9054    08:55:54.162874 read(3</proc/filesystems>, ..., 1024) = 478 <0.000052>
9054    08:55:54.163049 read(3</proc/filesystems>, "", 1024) = 0 <0.000040>
9054    08:55:54.163560 read(3</etc/locale.alias>, ..., 4096) = 2996 <0.000041>
9054    08:55:54.163679 read(3</etc/locale.alias>, "", 4096) = 0 <0.000044>
9054    08:55:54.176260 write(1</dev/pts/7>, ..., 50) = 50 <0.000111>
```

(a) System calls recorded by the MPI process with the ID 9042 for the command `ls` (Trace file: *a_host1_9042.st*).

```
9173    08:56:04.731999 read(3</usr/lib/x86_64-linux-gnu/libselinux.so.1>, ..., 832) = 832 <0.000187>
9173    08:56:04.734569 read(3</usr/lib/x86_64-linux-gnu/libc.so.6>, ..., 832) = 832 <0.000075>
9173    08:56:04.737108 read(3</usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.10.4>,..., 832) = 832 <0.000063>
9173    08:56:04.740961 read(3</proc/filesystems>, ..., 1024) = 478 <0.000080>
9173    08:56:04.741210 read(3</proc/filesystems>, "", 1024) = 0 <0.000067>
9173    08:56:04.742237 read(3</etc/locale.alias>, ..., 4096) = 2996 <0.000097>
9173    08:56:04.742505 read(3</etc/locale.alias>, "", 4096) = 0 <0.000083>
9173    08:56:04.754208 read(4</etc/nsswitch.conf>, ..., 4096) = 542 <0.000140>
9173    08:56:04.754487 read(4</etc/nsswitch.conf>, "", 4096) = 0 <0.000027>
9173    08:56:04.755279 read(4</etc/passwd>, ..., 4096) = 1612 <0.000037>
9173    08:56:04.756740 read(4</etc/group>, ..., 4096) = 872 <0.000091>
9173    08:56:04.758661 write(1</dev/pts/7>, ..., 9) = 9 <0.000074>
9173    08:56:04.759173 read(3</usr/share/zoneinfo/Europe/Berlin>, ..., 4096) = 2298 <0.000074>
9173    08:56:04.759471 read(3</usr/share/zoneinfo/Europe/Berlin>, ..., 4096) = 1449 <0.000033>
9173    08:56:04.759816 write(1</dev/pts/7>, ..., 74) = 74 <0.000099>
9173    08:56:04.760043 write(1</dev/pts/7>, ..., 53) = 53 <0.000073>
9173    08:56:04.760233 write(1</dev/pts/7>, ..., 65) = 65 <0.000099>
```

(b) System calls recorded by the MPI process with the ID 9157 for the command `ls -l` (Trace file: *b_host1_9157.st*).

```
77423   16:56:40.452431 read(3</usr/lib/x86_64-linux-gnu/libselinux.so.1>, <unfinished ...>
...
...
77423   16:56:40.452660 <... read resumed> ..., 405) = 404 <0.000223>
```

(c) An example of strace records in case of simultaneous multi-processing.

Fig. 2: Examples of traces generated by `strace`.

two events are exactly the same. For example, if there are two events from the same command that indicate read access of same size to the same file at the same time for the same duration from the same MPI process from the same host, then these two events *must* have different pid. For instance, if the `-f` option is not added to `strace`, the pid is not recorded. This can result in two independent invocations of system calls being identical and pointing to the same event, which is not desired.

Case. A group of events that belong to a particular rid, $host$ and cid, arranged in increasing order of their timestamps, is referred to as a *case*. In other words, the group of events in each trace file is considered a unique case. A case **c** is indicated as an arrangement of events as follows,

$$\mathbf{c} = \langle e_1, e_2, \ldots e_n \rangle \tag{2}$$

where all $e_i \in \mathbf{c}$ are the events that occur in the case **c**, and $\forall e_i, e_{i+1} \in \mathbf{c}$, the $start$ timestamp of e_i is less than or equal to that of e_{i+1}. For example, the case corresponding to the execution of `ls` command on rid=9042, consisting of sequence of events parsed from the trace file $a\_host1\_9042.st$,

is shown in Figure 2a. Note that, according to this definition of case, we do not distinguish between different SMT or OpenMP processes within the same MPI process. However, one could do so by re-defining case as a group of events belonging to the same cid, $host$, and pid (instead of rid).

Event-log. A set of cases $\mathcal{C} = \{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$ is referred to as an *event-log*. For example, the following sets of cases can be considered from the experiment in Figure 1:

$$\begin{aligned} \mathcal{C}_a &= \{\mathbf{a9042}, \mathbf{a9043}, \mathbf{a9045}\} \\ \mathcal{C}_b &= \{\mathbf{b9157}, \mathbf{b9158}, \mathbf{b9160}\} \\ \mathcal{C}_x &= \mathcal{C}_a \cup \mathcal{C}_b \end{aligned} \tag{3}$$

where $\mathcal{C}_a$ is the set of cases executing the command `ls`, $\mathcal{C}_b$ is the set of cases executing the command `ls -l`, and $\mathcal{C}_x$ is the set of cases involved in the execution of both the commands.

Mapping and Activity. A *mapping* is a *partial* function f that maps an event $e \in \mathcal{E}$ to a named entity referred to as an *activity* $a \in \mathcal{A}_f$, and it is denoted as $f : \mathcal{E} \rightharpoonup \mathcal{A}_f$. The mapping f is a function because an event $e \in \mathcal{E}$ is mapped to at most one activity $a \in \mathcal{A}_f$, and it is partial because not all

$e \in \mathcal{E}$ are required to have a mapping. For example, consider the following mapping:

$\hat{f}$: for a given event, return a string concatenating

call with fp truncated to contain at most (4)

top two directory levels.

According to this mapping, the event parsed from the first line of the trace file in Figure 2b would map to "***read:/usr/lib***". Notice that f can be one-to-one or many-to-one. Therefore, the reverse mapping $f^{-1} : \mathcal{A}_f \to \mathcal{E}$ is a *multi-valued* function[3] that maps every $a \in \mathcal{A}_f$ to a subset of events in $\mathcal{E}$; i.e., $f^{-1}(a) \subseteq \mathcal{E}$. Following up on our example, $\hat{f}^{-1}($***read:/usr/lib***$)$ is a subset of events that correspond to those in the first three lines of the trace file shown in Figure 2b.

Trace. For a given mapping $f : \mathcal{E} \rightharpoonup \mathcal{A}_f$ and a case $\mathbf{c} \in \mathcal{C}$, the sequence of activities corresponding to the events observed in $\mathbf{c}$ is called an activity trace or simply *trace* $\sigma_f(\mathbf{c})$, i.e.,

$$
\begin{aligned}
\sigma_f(\mathbf{c}) &= f \circ \mathbf{c} \\
&= \langle f(e_1), f(e_2), \ldots f(e_n) \rangle \\
&= \langle a_1, a_2, \ldots a_n \rangle
\end{aligned}
\tag{5}
$$

For example, for the mapping $\hat{f}$ (defined in Equation 4), the trace for the case **a9042** $\in \mathcal{C}_a$ (corresponding to the trace file in Figure 2a) is:

$$
\begin{aligned}
\sigma_{\hat{f}}(\mathbf{a9042}) = \langle\ &\textbf{\textit{read:/usr/lib}}, \textbf{\textit{read:/usr/lib}}, \textbf{\textit{read:/usr/lib}}, \\
&\textbf{\textit{read:/proc/filesystems}}, \textbf{\textit{read:/proc/filesystems}}, \\
&\textbf{\textit{read:/etc/locale.alias}}, \textbf{\textit{read:/etc/locale.alias}}, \\
&\textbf{\textit{write:/dev/pts}} \rangle
\end{aligned}
$$

Note that for all $e_i, e_j \in \mathbf{c}$, e_i precedes e_j implies a_i precedes a_j for all $a_i, a_j \in \sigma_f(\mathbf{c})$.

Activity-log. For a given event-log $\mathcal{C}$ and a mapping $f : \mathcal{E} \rightharpoonup \mathcal{A}_f$, an *activity-log* $L_f(\mathcal{C})$ is a multi-set (i.e., a set with multiple instances of the same element) of traces over $\mathcal{A}_f$ for $\mathcal{C}$, and it is represented as $L_f(\mathcal{C}) \in \mathbb{B}({\mathcal{A}_f}^*)$, where ${\mathcal{A}_f}^*$ is the set of all possible sequences of activities in $\mathcal{A}_f$. For example, consider a fictitious event-log $\mathcal{C} = \{0, 1, 2\}$. If $\mathcal{A}_f = \{a, b, c\}$, and the traces are $\sigma_f(0) = \langle a, a, b \rangle$, $\sigma_f(1) = \langle a, a, b \rangle$, $\sigma_f(2) = \langle a, c \rangle$, then the activity-log $L_f(\mathcal{C}) = \{\langle a, a, b \rangle^2, \langle a, c \rangle\}$; the trace $\langle a, a, b \rangle$ from $\sigma_f(0)$ and $\sigma_f(1)$ is indicated with multiplicity 2. Now, consider the event-logs shown in Equation 3 and the mapping $\hat{f}$. The activity-log $L_{\hat{f}}(\mathcal{C}_a)$ after appending every trace with a start ($\bullet$) and an end ($\blacksquare$) activity would be:

$$
\begin{aligned}
L_{\hat{f}}(\mathcal{C}_a) = \{\langle \bullet,\ &\textbf{\textit{read:/usr/lib}}, \textbf{\textit{read:/usr/lib}}, \textbf{\textit{read:/usr/lib}}, \\
&\textbf{\textit{read:/proc/filesystems}}, \textbf{\textit{read:/proc/filesystems}}, \\
&\textbf{\textit{read:/etc/locale.alias}}, \textbf{\textit{read:/etc/locale.alias}}, \\
&\textbf{\textit{write:/dev/pts}}, \blacksquare \rangle^3 \}
\end{aligned}
$$

From $\mathcal{C}_a$, all the three cases **a9042**, **a9043** and **a9045** map to the same trace, and hence $L_{\hat{f}}(\mathcal{C}_a)$ consist of a single trace with multiplicity of 3. Similarly, the activity logs $L_{\hat{f}}(\mathcal{C}_b)$ and $L_{\hat{f}}(\mathcal{C}_x)$ are:

$$
\begin{aligned}
L_{\hat{f}}(\mathcal{C}_b) = \{\langle \bullet, &\textbf{\textit{read:/usr/lib}}, \textbf{\textit{read:/usr/lib}}, \textbf{\textit{read:/usr/lib}}, \\
&\textbf{\textit{read:/proc/filesystems}}, \textbf{\textit{read:/proc/filesystems}}, \\
&\textbf{\textit{read:/etc/locale.alias}}, \textbf{\textit{read:/etc/locale.alias}}, \\
&\textbf{\textit{read:/etc/nsswitch.conf}}, \textbf{\textit{read:/etc/nsswitch.conf}}, \\
&\textbf{\textit{read:/etc/passwd}}, \textbf{\textit{read:/etc/group}}, \\
&\textbf{\textit{write:/dev/pts}}, \textbf{\textit{read:/usr/lib}}, \textbf{\textit{read:/usr/lib}}, \\
&\textbf{\textit{write:/dev/pts}}, \textbf{\textit{write:/dev/pts}}, \textbf{\textit{write:/dev/pts}}, \blacksquare \rangle^3 \}
\end{aligned}
$$

$$
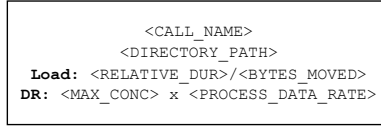L_{\hat{f}}(\mathcal{C}_f) = L(\mathcal{C}_a) \cup L(\mathcal{C}_b)
$$

Thus, an activity-log can be seen as a query and an abstraction applied to an event-log through the mapping f; this mapping provides a mechanism to shift the focus of information in the event-log. The activity-log is used as an input to construct the DFG.

A. Construction of the Directly-Follows-Graph

Given an event-log $\mathcal{C}$ and a mapping $f : \mathcal{E} \rightharpoonup \mathcal{A}_f$, the activity-log $L_f(\mathcal{C}) \in \mathbb{B}({\mathcal{A}_f}^*)$ is determined, and the DFG $G[L_f(\mathcal{C})]$ is constructed such that $a \in \mathcal{A}_f$ are the nodes and an edge (a_1, a_2) where $a_1, a_2 \in \mathcal{A}_f$ exists if and only if there exists a trace in the activity-log, i.e., $\exists \sigma_f \in L_f$ such that a_1 immediately precedes (or directly follows) a_2. If $a_1 = a_2$, then the node has an edge pointing to itself.

For example, for the activity-logs $L_{\hat{f}}(\mathcal{C}_a)$, $L_{\hat{f}}(\mathcal{C}_b)$ and $L_{\hat{f}}(\mathcal{C}_x)$, the corresponding the DFGs $G[L_{\hat{f}}(\mathcal{C}_a)]$, $G[L_{\hat{f}}(\mathcal{C}_b)]$ and $G[L_{\hat{f}}(\mathcal{C}_x)]$ are shown in Figure 3b, Figure 3c and Figure 3d respectively. The number on the edges indicates how many times the corresponding directly-follows relation was observed in the activity-log. The statistics indicated in the nodes (i.e., Load and DR) and the coloring schemes will be explained in the following sub-sections. The implementation for scalable construction of G from activity-log $L_f(\mathcal{C})$ is discussed in [24], [25]. Thus, $G[L_{\hat{f}}(\mathcal{C}_a)]$ is the DFG synthesis according to $\hat{f}$ for the processes executing the `ls` command, and similarly $G[L_{\hat{f}}(\mathcal{C}_b)]$ for the processes executing the `ls -l` command. $G[L_{\hat{f}}(\mathcal{C}_x)]$ is the synthesis of events from all the processes executing both commands.

The DFG is a response to a query applied through f on the event-log. One could modify the query to restrict the synthesis to a particular section of the event-log. For example, to restrict the synthesis to the directory `/usr/lib`, define a mapping f_1 such that it maps an event to an activity only if the file path contains the sub-string */usr/lib*. Then, for the corresponding activity-log $L_{f_1}(\mathcal{C}_x) \in \mathbb{B}({\mathcal{A}_{f_1}}^*)$ over $\mathcal{C}_x$ and the mapping $f_1 : \mathcal{E} \rightharpoonup \mathcal{A}_{f_1}$, the DFG file access footprint $G[L_{f_1}(\mathcal{C}_x)]$ is shown in Figure 4. Thus, the DFG provides a way for the users to interactively visualize the I/O accesses made by their application.

(a) Semantics inside a node of the DFG.

(b) $G[L_{\hat{f}}(\mathcal{C}_a)]$

(c) $G[L_{\hat{f}}(\mathcal{C}_b)]$

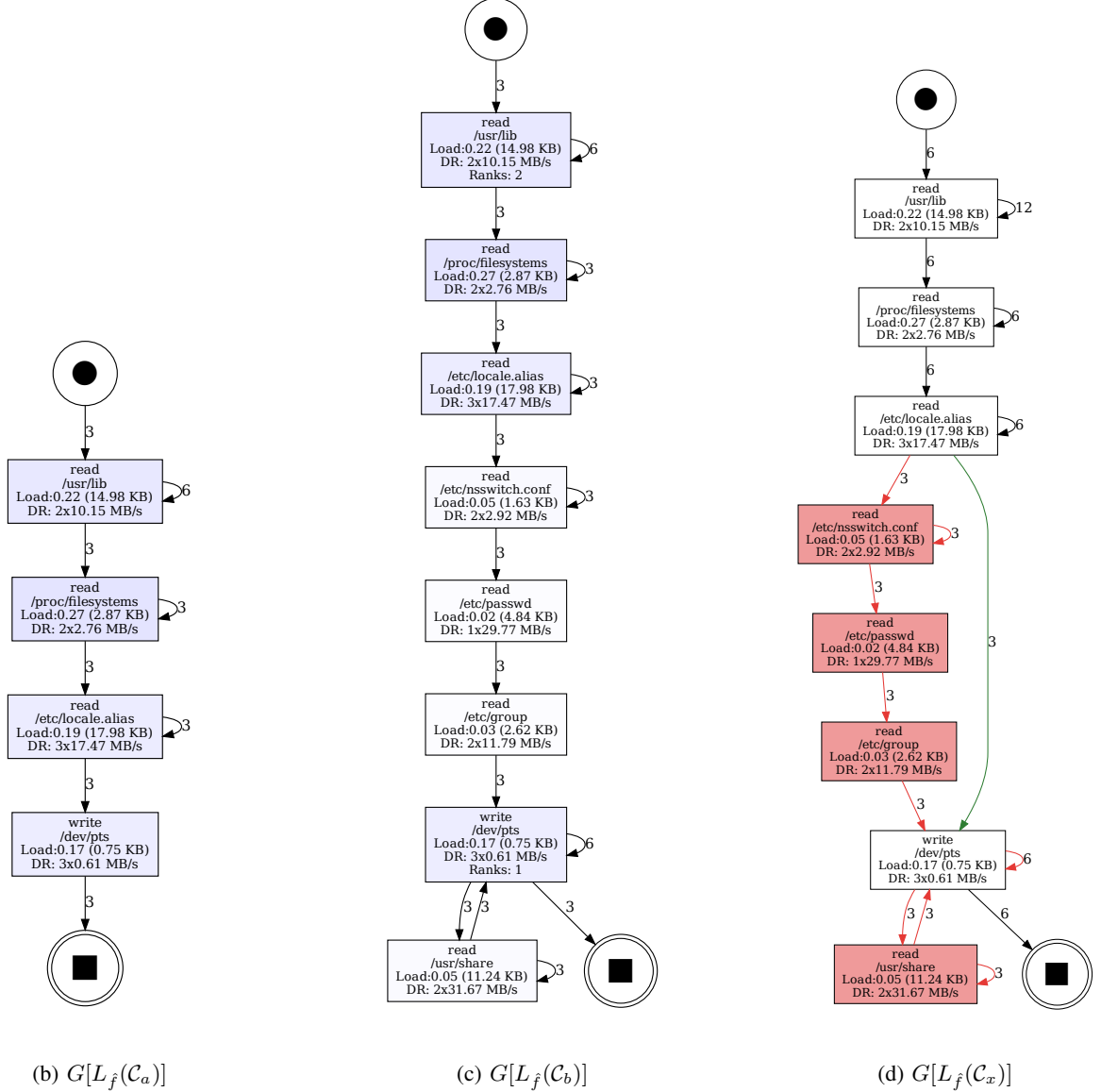(d) $G[L_{\hat{f}}(\mathcal{C}_x)]$

Fig. 3: The DFG synthesis for the event-logs in Equation 3. The nodes indicate the file access activities and the number on the edges indicate the number of times the directly-follows relation between two activities was observed.

B. Activity Statistics

Given an event-log $\mathcal{C}$ and a mapping $f : \mathcal{E} \rightharpoonup \mathcal{A}_f$, we compute statistics for each $a \in \mathcal{A}_f$, which add performance perspectives to the nodes of the DFG. Particularly, for each node, we aim to determine the proportion of system time spent relative to activities represented by the other nodes, the number of bytes transferred, and the rate of data movement. To this end, we compute the following statistics:

- **Relative duration**: The *relative duration* of an activity $a \in \mathcal{A}_f$ encountered in $\mathcal{C}$ is defined as the proportion of the time spent by events on activity a relative to the total time spent across all activities $\forall a \in \mathcal{A}_f$. Let $e[dur]$ be the duration of system call for event $e \in \mathcal{E}$. In order to compute the relative duration $rd_f(a, \mathcal{C})$, we first
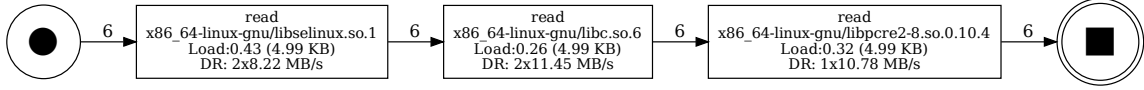
Fig. 4: The DFG synthesis for the event-logs in Equation 3. The nodes indicate the file access activities and the number on the edges indicate the number of times the directly-follows relation between two activities was observed.
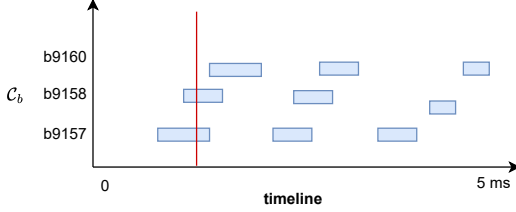

Fig. 5: The timeline plot: $\mathbf{t}_{\hat{f}}(\text{“}\textbf{read:/usr/lib}\text{”}, \mathcal{C}_b)$.

aggregate the duration of all the events related to $a \in \mathcal{A}_f$ occurring in $\mathcal{C}$; i.e.,

$$\mathbf{d}_f(a, \mathcal{C}) = \{e[dur] \mid \forall \mathbf{c} \in \mathcal{C}, \forall e \in \mathbf{c} \cap f^{-1}(a)\} \quad (6)$$

In words, for a given event-log $\mathcal{C}$ and an activity $a \in \mathcal{A}_f$, we check for each case in $\mathbf{c} \in \mathcal{C}$ and for each event $e \in \mathbf{c}$, if this event e is in the set of events that maps to activity a, i.e., $f^{-1}(a)$. If it does, then the value corresponding to the duration attribute of this event, i.e., $e[dur]$ is added to the set $\mathbf{d}_f(a, \mathcal{C})$. For example, $\mathbf{d}_{\hat{f}}(\text{“}\textbf{read:/usr/lib}\text{”}, \mathcal{C}_a)$ is a set that constitutes the duration values parsed from the three trace files represented by $\mathcal{C}_a$, and consists of only those events with file-path containing the sub-string /usr/lib. After determining $\mathbf{d}_f(a, \mathcal{C})$, we compute the sum of all the duration values in the set,

$$\bar{\mathbf{d}}_f(a, \mathcal{C}) = \sum \mathbf{d}_f(a, \mathcal{C}) \quad (7)$$

and the relative duration $rd_f(a, \mathcal{C})$) is computed as follows:

$$rd_f(a, \mathcal{C}) = \frac{\bar{\mathbf{d}}_f(a, \mathcal{C})}{\sum_{a \in \mathcal{A}_f} \bar{\mathbf{d}}_f(a, \mathcal{C})} \quad (8)$$

This metric allows us to gauge the relative importance of each activity in terms of the system time spent on I/O.

- **Total bytes moved**: Let $e[size]$ be the number of bytes moved for event $e \in \mathcal{E}$. The total number of bytes moved for activity $a \in \mathcal{A}_f$ occurring in $\mathcal{C}$ is:

$$b_f(a, \mathcal{C}) = \sum \{e[size] \mid \forall \mathbf{c} \in \mathcal{C}, \forall e \in \mathbf{c} \cap f^{-1}(a)\} \quad (9)$$

In the nodes of the DFGs in Figure 3, the relative duration and total bytes moves are combined and indicated as:

$$\text{“}\textbf{Load: } rd_{\hat{f}} \ (b_{\hat{f}})\text{”} \quad (10)$$

Next, we approximate the rate of data movement for each activity by computing the following:

- **Process Data Rate**: The average number of bytes transferred per second per process for activity $a \in \mathcal{A}_f$ encountered in $\mathcal{C}$ is considered as the *process data rate* for activity a, and it is denoted as $\bar{dr}_f(a, \mathcal{C})$. In order to compute the rate at which each process performing an activity moves data, we first define the data rate for an event $e \in \mathcal{E}$:

$$dr(e) = \frac{e[size]}{e[dur]} \quad (11)$$

We then aggregate the event data rates related to activity a occurring in $\mathcal{C}$:

$$\mathbf{dr}_f(a, \mathcal{C}) = \{dr(e) \mid \forall \mathbf{c} \in \mathcal{C}, \forall e \in \mathbf{c} \cap f^{-1}(a)\} \quad (12)$$

The process data rate dr_f is then the arithmetic mean (μ) of all the values in $\mathbf{dr}_f$:

$$\bar{dr}_f(a, \mathcal{C}) = \mu(\mathbf{dr}_f(a, \mathcal{C})) \quad (13)$$

- **Max-Concurrency**: The *maximum concurrency* attained for activity $a \in \mathcal{A}_f$ encountered in $\mathcal{C}$ is the highest number of concurrent events corresponding to a that occurred in $\mathcal{C}$, and it is denoted as $mc_f(a, \mathcal{C})$. In order to compute mc_f, we first define the start and end timestamp for each event $e \in \mathcal{E}$ as a tuple:

$$t(e) = (e[start], e[start] + e[dur]) \quad (14)$$

and aggregate the time stamps of all the events for each a occurring in $\mathcal{C}$ into a list:

$$\mathbf{t}_f(a, \mathcal{C}) = [t(e) \mid \forall \mathbf{c} \in \mathcal{C}, \forall e \in \mathbf{c} \cap f^{-1}(a)] \quad (15)$$

Each $t(e) \in \mathbf{t}_f(a, \mathcal{C})$ can be visualized as a range from start to end timestamp in a timeline plot. For example, the timeline plot of $\mathbf{t}_{\hat{f}}(\text{“}\textbf{read:/usr/lib}\text{”}, \mathcal{C}_b)$ is shown in Figure 5. The max-concurrency mc_f is computed as:

$$mc_f(a, \mathcal{C}) = \texttt{get\_max\_concurrency}(\mathbf{t}_f(a, \mathcal{C})) \quad (16)$$

The algorithm $\texttt{get\_max\_concurrency}$ first sorts $\mathbf{t}_f$ according to increasing start timestamps, iterates through the sorted $\mathbf{t}_f$, and determines the maximum number of consecutive events that could be identified such that the end time of the first event is greater than the start time of the last event. For example, in $\mathbf{t}_{\hat{f}}(\text{“}\textbf{read:/usr/lib}\text{”}, \mathcal{C}_b)$, max-concurrency is 2. Notice that, for precise estimation

of this metric in a program with processes distributed across multiple nodes, the system clocks have to be synchronized. If they are not, then the mc_f values may not be exact. However, *not* having the clocks synchronized does *not* affect the DFG construction or the other metrics.

In the nodes of the DFGs in Figure 3, the process data rate and the max-concurrency statistics are combined and indicated as:

$$\text{``}\mathbf{DR:}\ mc_{\hat{f}} \times \bar{dr}_{\hat{f}}\text{''} \tag{17}$$

This metric shows an estimation of the rate at which a file access activity induces I/O load on the system.

Thus, appending the DFG with statistics related to file access activities enhances the visualization by providing additional information, with which one could analyse not only the file accesses but also how the activities differ from each other in terms of system load and data movements.

C. Performance Comparisons via Graph Coloring

For a given event-log $\mathcal{C}$ and a mapping $f : \mathcal{E} \rightharpoonup \mathcal{A}_f$, we color the nodes and edges of the DFG $G[L_f(\mathcal{C})]$ according to one of the following strategies:

1) **Statistics-based coloring**: A straightforward method to visually compare the I/O operations represented by the activities in $\mathcal{A}_f$ is by coloring the nodes of the DFG $G[L_f(\mathcal{C})]$ based on the statistics described in the previous sub-section. For instance, in Figure 3b and Figure 3c, the activities are colored based on the relative duration; i.e., higher the value of rd_f, the darker the shade of blue. Alternatively, one could color the nodes based on the number of bytes b_f moved. However, with this method, one could not identify the similarities and differences in the I/O operations *among the different processes*, i.e., among the different cases $\mathbf{c}_i \in \mathcal{C}$.

2) **Partition-based coloring**: In order to make comparisons among the cases in an event-log $\mathcal{C}$, we perform the following steps:

   a) From the event-log $\mathcal{C}$, identify two mutually exclusive subsets $\mathcal{G}$ and $\mathcal{R}$.

   b) Construct the DFG $G[L_f(\mathcal{C})]$ from the full event-log $\mathcal{C}$, and the DFGs $G[L_f(\mathcal{G})]$ and $G[L_f(\mathcal{R})]$ from the event-log subsets.

   c) Color the nodes and edges of $G[L_f(\mathcal{C})]$ as follows:
   - The nodes and edges that occur *exclusively* in $G[L_f(\mathcal{G})]$ are given the color *green*.
   - The nodes and edges that occur *exclusively* in $G[L_f(\mathcal{R})]$ are given the color *red*.
   - The nodes and edges that occur in both $G[L_f(\mathcal{G})]$ and $G[L_f(\mathcal{R})]$ are not colored.

   For example, let us compare and contrast I/O operations between the processes executing the commands `ls` and `ls -l`. To this end, we consider the following partition of $\mathcal{C}_x$ (based on Equation 3):

   $$\begin{aligned} \mathcal{G}_x &= \mathcal{C}_a & \text{i.e., the processes executing } \texttt{ls} \\ \mathcal{R}_x &= \mathcal{C}_a & \text{i.e., the processes executing } \texttt{ls -l} \end{aligned} \tag{18}$$

The coloring of the DFG $G[L_{\hat{f}}(\mathcal{C}_x)]$ based on the DFGs constructed from the subsets $\mathcal{G}_x$ and $\mathcal{R}_x$ is shown in Figure 3d. The nodes and edges in red are those that occur exclusively in the processes executing the command `ls -l`. There are no activities that occur exclusively in `ls`, except a single directly-follows relation indicated as an edge from "***read****:/etc/locale.alias*" to "***write****:/dev/pts*". The remaining activities and relations are observed in the processes executing both commands.

V. EXPERIMENTS

In this section, we apply DFG synthesis to I/O traces from runs of IOR benchmark. First, we describe our implementation and the HPC environment on which the experiments are conducted. Then, we run the IOR benchmark with different options for file output and software interface, and compare the file access contentions between the runs.

Implementation: We use strace version 6.4 to trace user programs. After the program execution, the individual trace files are processed as described in Sec. III and stored in a single HDF5 file. Each processed trace file (i.e., each case) is stored in a separate group within the HDF5 file as a table. The columns of these tables correspond to the event attributes *pid, call, start, dur, fp, size* as defined in Sec. III. Each table contains the events for a particular case, sorted by increasing start timestamps (*start*). This format of the HDF5 file naturally represents an event-log according to our definition in Sec. IV. To synthesize the DFG, we perform the following steps in Python (also shown in Figure 6):

1) From each table in the HDF5 file, retrieve the events containing a given sub-string (e.g., "/usr/lib") in the file path value (*fp*). The resulting tables are concatenated and stored as a DataFrame object implemented by the Pandas library. The DataFrame additionally has a column named "case" that points each event to the corresponding trace file name.

2) Implement the mapping function f and apply it to the DataFrame to add a column named "activity". In step 2 of the code in Figure 6, we show the Python implementation of the mapping function defined in Equation 4. This operation is $\mathcal{O}(n)$, where n is the number of rows in the DataFrame, and it is scalable as it is applied independently to each row.

3) Notice that the DataFrame with only the "case" and "activity" columns represents the activity-log according to our definition in Sec. IV. The construction of the DFG requires a single iteration through the activity-log. Therefore, this operation is also $\mathcal{O}(n)$ and can be scaled [25].

4) The computation of I/O statistics requires a single pass over the DataFrame followed by a grouping and aggregation based on activity values. Therefore, the complexity is $\mathcal{O}(mn)$, where m is the number of unique activities $|\mathcal{A}_f|$ (i.e., the number of nodes in the DFG). For all intents and purposes, the mapping function should be

```python
import pandas as pd
from st_inspector import *

#0) Pointer to the H5 event-log file
event_log = EventLogH5(H5_FILE_PATH)

#1) Filter the event log. Stores the data as a pd.DataFrame
event_log.apply_fp_filter('/usr/lib')

#2a) Implement the mapping of events to activity values
def f(event:pd.Series) -> str:
        # Get the file_path attribute of the event
        fp = event['fp']

        # Truncate file path to top-two directories
        dirs = fp.split('/')
        if len(dirs) > 2:
                fp = f'/{dir[1]}/{dir[2]}'

        # return concatenatenation of call and fp
        return f'{event['call']}\n{fp}'


#2b) Apply the mappping fn to determine the activity values
event_log.apply_mapping_fn(f)

#3) Construct the DFG
dfg = DFG(event_log)

#4) Compute I/O statistics
stats = IOStatistics()
stats.compute_statistics(event_log)

#5a) Apply statistics-based coloring on the dfg
colored_dfg = DFGViewer(dfg, styler=StatisticsColoring(stats))
colored_dfg.render()

#5b) (or) Apply partition-based coloring on the dfg
green_event_log, red_event_log = PartitionEL(event_log)
green_dfg = DFG(green_event_log)
red_dfg = DFG(red_event_log)
partition_coloring = PartitionColoring(green_dfg,red_dfg,stats)
colored_dfg = DFGViewer(dfg, styler=partition_coloring)
colored_dfg.render()
```

Fig. 6: DFG synthesis using the Python library: *st_inspector*.

defined such that m is small; otherwise, the visual analysis of the DFG would be tedious.

5) The DFG along with the statistics are rendered. The rendered DFG is styled by applying one of the coloring methods described in Sec. IV-C. The complexity of the rendering is $\mathcal{O}(m^2)$ in the worst case; i.e., when every node has an edge to every other node. Note that partition-based coloring (Step 5b in Figure 6) requires at most one additional pass over the activity-log to construct both the DFGs, i.e., $\mathcal{O}(n)$, to compute both *green_dfg* and *red_dfg*.

Our implementation is available through the Python library *st_inspector*. The source code is available in Zenodo [26]. This implementation is used for the experiments.

The HPC Environment: The experiments are conducted on the compute nodes of the JUWELS cluster [27] at the Jülich Supercomputing Centre. We use the nodes of JUWELS that have 2x24 cores Intel Xeon Platinum 8168 CPUs with 96 GB DDR4 memory. JUWELS is connected via Connext-X4 Infiniband/Ethernet adaptor to the 6th generation JUST storage infrastructure [28], which is a GPFS based file system.

A. Single Shared File vs File Per Process

In applications where multiple processes are simultaneously involved in I/O operations, we compare the following two scenarios: (1) Single Shared File (**SSF**): all processes read from or write to a single shared file, and (2) File Per Process
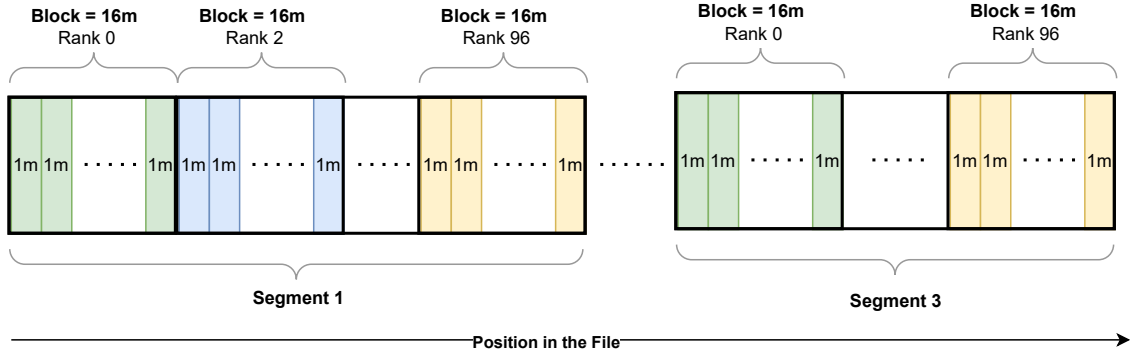
(**FPP**): each process accesses its own individual file. Generally, allowing each process to work on its own file eliminates the contention issues arising from file locking, which is otherwise common in the shared file scenario. However, creating a file for each process at scale leads to metadata overhead, which hits performance especially when data needs to be frequently gathered across processes. Therefore, users need to gauge the trade-offs between the two approaches and quantify, for a given application, whether contention leads to significantly increased execution times compared to the case where each process operates on its own file. In this experiment, we apply our methodology to verify the possibility of identifying these contention issues and visualizing the differences in file access activities between the two scenarios.

We use the IOR benchmark suite to write and read in parallel from 96 MPI processes or rank spanning across 2 nodes of the JUWELS cluster (i.e., 96 cores in total and we execute one MPI rank per core). Each rank first writes 3 segments of data, with each segment consisting of a 16 MB block, and each block divided into 16 write operations of 1 MB each. Subsequently, each rank reads the data written by a process from the neighboring node (this is done to avoid reading the data stored in the DRAM). The format of the file is shown in Figure 7a, and the IOR options for the SSF and FPP experiments are shown in Figure 7b. The number of segments, block size, and size of each operation are specified with the options -s, -b, and -t, respectively. The option -C forces the MPI ranks to read the data written by the neighboring node, and the option -e ensures that the write to the file system is complete before starting the subsequent read operation. By default, IOR runs in the SSF mode, and switches to FPP when the option -F is specified. We run IOR in both modes; for SSF, the files are accessed from the folder $SCRTACH/ssf, and for FPP, the files are accessed from the folder $SCRTACH/fpp. The access path is specified using the -o option. We aim to identify the differences in file contentions occurring in these two folders.

We record the events related to variants of read, write and openat system calls and prepare the HDF5 event-log. The event-log $\mathcal{C}_X$ consists of 192 cases (each stored as a table in the HDF5 file); 96 from the SSF run and 96 from the FPP run. We retrieve all the events without applying any file path filters. The map of events to activity values ($\bar{f}$) is similar to the one defined in Equation 4, but abstracts the file paths based on site-specific variable.

After applying the mapping $\bar{f}$, the activity-log $L_{\bar{f}}(\mathcal{C}_X)$ is determined, and the DFG $G[L_{\bar{f}}(\mathcal{C}_X)]$ is constructed, as shown in Figure 8a. The I/O statistics described in Section IV-B are computed and the nodes of the DFG are colored based on the corresponding relative duration statistic ($rd_{\bar{f}}$) computed according to Equation 8; higher the value of $rd_{\bar{f}}$, darker the shade of blue. It can be seen that openat and write operations under the $SCRATCH directory have a relatively high load.

Now, knowing that there is a relatively high load under the $SCRATCH directory, we filter the event-log to retrieve
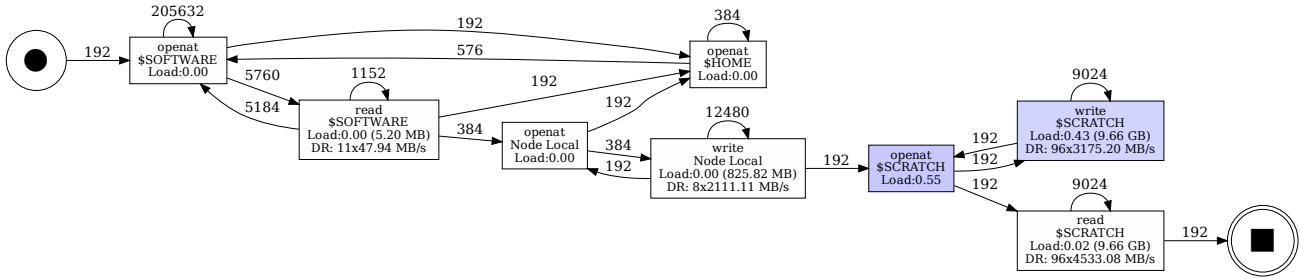
(a) The format of the IOR file.

```
#Single Shared File
srun -n 96 ./strace.sh ./ior -t 1m -b 16m -s 3 -w -r -C -e -o $SCRATCH/ssf/test

#One File per Process
srun -n 96 ./strace.sh ./ior -t 1m -b 16m -s 3 -w -r -F -C -e -o $SCRATCH/fpp/test
```
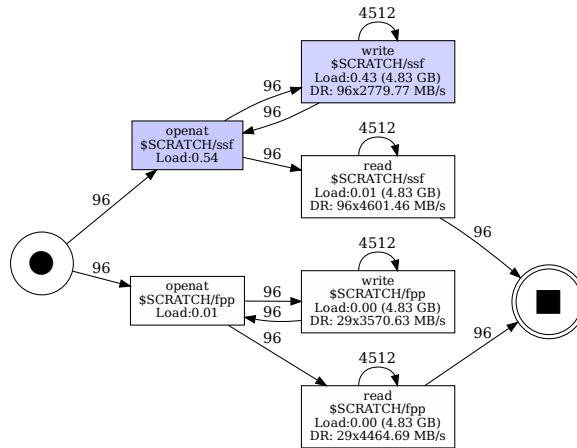
(b) IOR commands to simulate SSF and FPP scenarios.

Fig. 7: The IOR Experiment.

(a) DFG synthesis applied to all the events.

(b) DFG synthesis applied only to events that access the $SCRATCH directory.

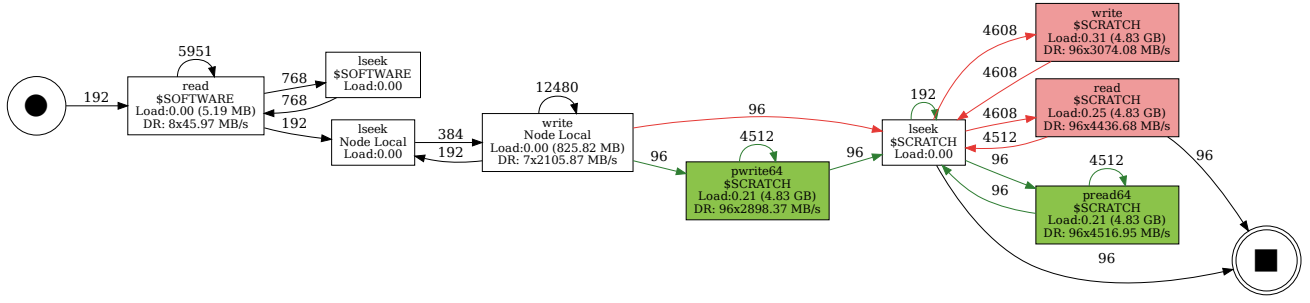Fig. 8: DFG synthesis of the events from both SSF and FFP modes of IOR runs.

Fig. 9: DFG synthesis of events from the MPI-IO experiment.

only those events that access the $SCRATCH directory. We re-apply the mapping and construct the DFG, which is shown in Figure 8b. It can be seen that the openat and write activities on files under $SCRATCH/ssf (which is the access path for the IOR run in SSF mode) have a significantly higher relative duration than those corresponding to the files under $SCRTACH/fpp (access path for the IOR run in FPP mode). This quantifies the contention issue due to file locking in the SSF scenario in terms of execution times.

B. With vs Without MPI-IO Interface

The MPI-IO provides standard interfaces for parallel I/O access [29]. It has been noted that performance gains through this interface are not guaranteed unless it is correctly configured by identifying the MPI-IO defined file access patterns within the program [4]. Therefore, tools that assist users in comparing the performance impacts of different interface configurations can be beneficial. In this experiment, we run the IOR benchmark both with and without the MPI-IO interface. By default, IOR does not use the MPI-IO interface. Adding the option -a mpiio would do a naive replacement of standard file operations with the MPI-IO counterpart, without the use of advanced MPI-IO configurations. We run IOR in SSF mode as before, this time with and without the -a mpiio option, and compare the file access contention between the runs.

In addition to variants of read, write, and openat, we also record the events related to lseek, and prepare the event-log $\mathcal{C}_Y$. We retrieve all the events, apply the mapping $\bar{f}$, construct the DFG $G[L_{\bar{f}}(\mathcal{C}_Y)]$, and compute the I/O statistics. Unlike the previous run, this time the two runs do not use distinct file access paths. Therefore, to compare the two runs, we apply the partition-based coloring described in Sec. IV-C. To this end, the event-log $\mathcal{C}_Y$ is partitioned into two mutually exclusive event logs, $\mathcal{G}_Y$ and $\mathcal{R}_Y$. The event-log $\mathcal{G}_Y$ constitutes cases that were run *with* the MPI-IO interface, and $\mathcal{R}_Y$ constitutes cases that were run *without* the MPI-IO interface. We color the nodes of $G[L_{\bar{f}}(\mathcal{C}_Y)]$ according to the steps defined in partition-based coloring (Sec. IV-C). The resulting DFG is shown in Figure 9; the green nodes and edges are those that occur exclusively in the run with MPI-IO interface, and red nodes and edges are those that occur exclusively in the

run without the MPI-IO interface. All other nodes and edges occur in both the runs (we skip the rendering of openat calls in Figure 9 as it does not highlight useful differences).

It can be seen that MPI-IO utilizes the system calls pread64 and pwrite64 instead of the standard read and write. Standard read or write calls start from the offset left by the previous access in an opened file, which means other processes must call lseek to reset the offset to the correct position before performing their own file operations. The pread64 and pwrite64 system calls combine file access and seek operations into a single command. This eliminates the need for an explicit lseek call before reading or writing, thereby reducing the number of system calls issued from the user environment. Therefore, one could observe that the number of lseek calls preceding file accesses is significantly lower in the run that use MPI-IO interface compared to the run without MPI-IO. We observe that the reduction in the number of system calls resulted in a relatively reduced load in terms of overall duration.

Availability of Data and Materials: The experimental data and the results that support the findings of this study are available in Zenodo with the identifier https://doi.org/10.5281/zenodo.13325645.

VI. CONCLUSION

In this paper, we presented a dependency-graph-based methodology for comparative analyses of arbitrary user programs in terms of I/O requests made to the operating system. To this end, we considered the I/O operations from the traces of system calls of one or more programs as input. We presented the methodology to transform the input data into a Directly-Follows-Graph consisting of nodes that represent I/O operations occurring at specific file paths, and the edges representing the directly-follows relation between those I/O operations. Based on the Directly-Follows-Graph, we described the technique to compare and contrast the patterns of I/O operations between multiple programs. We tested our methodology by applying it to the IOR benchmark and validating the similarities and differences in the patterns of I/O requests made to the operating system when IOR was run with different options for file output and software interface.

Our methodology can be used in tools that aim to diagnose and compare programs based on system resource contentions to identify I/O performance bottlenecks. Such tools are particularly valuable for supporting users of HPC systems and ensuring optimal resource usage. In future work, we plan to apply our technique to typical HPC workloads and document the findings.

REFERENCES

[1] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali, "Parallel computational steering for HPC applications using HDF5 files in distributed shared memory," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 6, pp. 852–864. [Online]. Available: http://ieeexplore.ieee.org/document/6152102/

[2] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific i/o interface," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. ACM, p. 39. [Online]. Available: https://dl.acm.org/doi/10.1145/1048935.1050189

[3] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman, "Tuning the performance of i/o-intensive parallel applications," in *Proceedings of the fourth workshop on I/O in parallel and distributed systems part of the federated computing research conference - IOPADS '96*. ACM Press, pp. 15–27. [Online]. Available: http://portal.acm.org/citation.cfm?doid=236017.236027

[4] R. Thakur, W. Gropp, and E. Lusk, "A case for using MPI's derived datatypes to improve i/o performance," in *Proceedings of the IEEE/ACM SC98 Conference*. IEEE, pp. 1–1. [Online]. Available: http://ieeexplore.ieee.org/document/1437288/

[5] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/5289150/

[6] H. Ather, J. L. Bez, B. Norris, and S. Byna, "Illuminating the i/o optimization path of scientific applications," in *High Performance Computing*, A. Bhatele, J. Hammond, M. Baboulin, and C. Kruse, Eds. Springer Nature Switzerland, vol. 13948, pp. 22–41, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-031-32041-5

[7] S. Shende, A. D. Malony, W. Spear, and K. Schuchardt, "Characterizing i/o performance using the TAU performance system," *IOS Press*, pp. 647–655.

[8] A. Knüpfer, C. Rössel, D. A. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, TAU, and vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Springer Berlin Heidelberg, pp. 79–91. [Online]. Available: http://link.springer.com/10.1007/978-3-642-31476-6

[9] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer Berlin Heidelberg, pp. 139–155. [Online]. Available: http://link.springer.com/10.1007/978-3-540-68564-7

[10] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel i/o tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/9150354/

[11] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Oliker, "Parallel i/o performance: From events to ensembles," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, pp. 1–11. [Online]. Available: http://ieeexplore.ieee.org/document/5470424/

[12] I. Zhukov, C. Feld, M. Geimer, M. Knobloch, B. Mohr, and P. Saviankou, "Scalasca v2: Back to the future," in *Tools for High Performance Computing 2014*, C. Niethammer, J. Gracia, A. Knüpfer, M. M. Resch, and W. E. Nagel, Eds. Springer International Publishing, pp. 1–24. [Online]. Available: https://link.springer.com/10.1007/978-3-319-16012-2

[13] W. M. P. Van Der Aalst, "Foundations of process discovery," in *Process Mining Handbook*, W. M. P. Van Der Aalst and J. Carmona, Eds. Springer International Publishing, vol. 448, pp. 37–75, series Title: Lecture Notes in Business Information Processing. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-08848-3_2

[14] P. Carns, "Darshan," in *High Performance Parallel I/O*, 2014th ed. Chapman and Hall/CRC, pp. 351–358.

[15] OTF2 Developer Community, "Open trace format version 2 (OTF2)." [Online]. Available: https://zenodo.org/record/7817732

[16] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, "Cube v4: From performance report explorer to performance analysis tool," *Procedia Computer Science*, vol. 51, pp. 1343–1352. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S187705091501128X

[17] "strace - the linux syscall tracer." [Online]. Available: https://github.com/strace/strace/releases/tag/v6.9

[18] J. Luettgau, S. Snyder, T. Reddy, N. Awtrey, K. Harms, J. L. Bez, R. Wang, R. Latham, and P. Carns, "Enabling agile analysis of i/o performance data with PyDarshan," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. ACM, pp. 1380–1391. [Online]. Available: https://dl.acm.org/doi/10.1145/3624062.3624207

[19] Z. Li, R. Atre, Z. Ul-Huda, A. Jannesari, and F. Wolf, "DiscoPoP: A profiling tool to identify parallelization opportunities," in *Tools for High Performance Computing 2014*, C. Niethammer, J. Gracia, A. Knüpfer, M. M. Resch, and W. E. Nagel, Eds. Springer International Publishing, pp. 37–54. [Online]. Available: https://link.springer.com/10.1007/978-3-319-16012-2

[20] A. Ketterlin and P. Clauss, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, pp. 437–448. [Online]. Available: http://ieeexplore.ieee.org/document/6493640/

[21] J.-B. Besnard, A. Tarraf, C. Barthélemy, A. Cascajo, E. Jeannot, S. Shende, and F. Wolf, "Towards smarter schedulers: Molding jobs into the right shape via monitoring and modeling," in *High Performance Computing*, A. Bienz, M. Weiland, M. Baboulin, and C. Kruse, Eds. Springer Nature Switzerland, vol. 13999, pp. 68–81, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-031-40843-4

[22] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni, "Parallel i/o and the metadata wall," in *Proceedings of the sixth workshop on Parallel Data Storage*. ACM, pp. 13–18. [Online]. Available: https://dl.acm.org/doi/10.1145/2159352.2159356

[23] W. Van Der Aalst, *Getting the Data*. Springer Berlin Heidelberg, pp. 125–162. [Online]. Available: http://link.springer.com/10.1007/978-3-662-49851-4

[24] S. J. J. Leemans, D. Fahland, and W. M. P. Van Der Aalst, "Scalable process discovery with guarantees," in *Enterprise, Business-Process and Information Systems Modeling*, K. Gaaloul, R. Schmidt, S. Nurcan, S. Guerreiro, and Q. Ma, Eds. Springer International Publishing, vol. 214, pp. 85–101, series Title: Lecture Notes in Business Information Processing. [Online]. Available: https://link.springer.com/10.1007/978-3-319-19237-6

[25] J. Evermann, "Scalable process discovery using map-reduce," *IEEE Transactions on Services Computing*, vol. 9, no. 3, pp. 469–481. [Online]. Available: http://ieeexplore.ieee.org/document/6948229/

[26] A. Sankaran, "STrace inspector (v1.0.0-beta)." [Online]. Available: https://doi.org/10.5281/zenodo.13325645

[27] D. Alvarez, "JUWELS cluster and booster: Exascale pathfinder with modular supercomputing architecture at juelich supercomputing centre," *Journal of large-scale research facilities- JLSRF*, vol. 7, p. A183. [Online]. Available: https://jlsrf.org/index.php/lsf/article/view/183

[28] S. Graf and O. Mextorf, "JUST: Large-scale multi-tier storage infrastructure at the jülich supercomputing centre," *Journal of large-scale research facilities JLSRF*, vol. 7, p. A180. [Online]. Available: https://jlsrf.org/index.php/lsf/article/view/180

[29] M. P. I. Forum, "MPI-2: Extensions to the message-passing interface." [Online]. Available: https://www.mpi-forum.org/docs/mpi-2.1/mpi21-report.pdf