DYAD

This page is maintained by the <a href="https://github.com/flux-fr

Sep 07, 2023

USER DOCS:

DYAD is a synchronization and data movement tool for computational science workflows built on top of Flux. DYAD aims to provide the benefits of in situ and in transit tools (e.g., fine-grained synchronization between producer and consumer applications, fast data access due to spatial locality) while relying on a file-based data abstraction to maximize portability and minimize code change requirements for workflows. More specifically, DYAD aims to overcome the following challenges associated with traditional shared-storage and modern in situ and in transit data movement approaches:

- Lack of per-file object synchronization in shared-storage approaches
- Poor temporal and spatial locality in shared-storage approaches
- Poor performance for file metadata operations in shared-storage approaches (and possibly some in situ and in transit approaches)
- Poor portability and the introduction of required code changes for in situ and in transit approaches

In resolving these challenges, DYAD aims to provide the following to users:

- Good performance (similar to in situ and in transit) due to on- or near-node temporary storage of data
- Transparent per-file object synchronization between producer and consumer applications
- Little to no code change to existing workflows to achieve the previous benefits

CHAPTER

ONE

GETTING STARTED

1.1 Prerequisites

DYAD has the following minimum requirements to build and install:

- A C99-compliant C compiler
- A C++11-compliant C++ compiler
- Autoconf 2.63
- Automake
- Libtool
- Make
- pkg-config
- Jansson 2.10
- flux-core

1.2 Installation

1.2.1 Manual Installation

Attention: Currently, DYAD can only be installed manually. This page will be updated as additional methods of installation are added.

Note: Recommended for developers and contributors

You can get DYAD from its GitHub repository using these commands:

```
$ git clone https://github.com/flux-framework/dyad.git
$ cd dyad
```

DYAD uses the Autotools for building and installation. To start the build process, run the following command to generate the necessary configuration scripts using Autoconf:

```
$ ./auotgen.sh
```

Next, configure DYAD using the following command:

```
$ ./configure --prefix=<INSTALL_PATH>
```

Besides the normal configure script flags, DYAD's configure script also has the following flags:

Flag	Type (default)	Description
–enable-dyad-debug	Bool (true if provided)	if enabled, include debugging prints and logs for DYAD at runtime
–enable-perfflow	Bool (true if provided)	if enabled, build PerfFlow Aspect-based performance measurement annotations for DYAD

Note: The installation prefix (i.e., `--prefix`) is also used to try to locate flux-core. First, `configure` will look for flux-core in the installation prefix. If it is not found there, `configure` will then use `pkg-config` to locate flux-core.

Finally, build and install DYAD using the following commands:

```
$ make [-j]
$ make install
```

1.2.2 Building with PerfFlow Aspect Support (Optional)

DYAD has optional support for collecting cross-cutting performance data using PerfFlow Aspect. To enable this support, first build PerfFlow Aspect for C/C++ using their instructions. Then, modify your method of choice for building DYAD as follows:

- **Manual Installation**: add `--enable-perfflow` to your invocation of *./configure*

1.3 Using DYAD's APIs

Currently, DYAD provides APIs for the following programming languages:

- C
- C++

This section describes the basics of integrating them into an application.

1.3.1 C API

DYAD's C API leverages the LD_PRELOAD trick to integrate into user applications. As a result, users can utilize DYAD's C API by simply adding the following before the shell command that launches their application:

```
$ LD_PRELOAD=path/to/dyad_wrapper.so
```

Once preloaded, DYAD's C API will intercept the `open` and `fopen` functions when consuming files and the `close` and `fclose` functions when producing files. As a result, if their code already uses thse functions, users do not need to change their code.

1.3.2 C++ API

DYAD's C++ API is implemented as a small library that wraps C++'s Standard Library file streams. To use DYAD's C++ API, first, add the following to your code:

```
#include <dyad_stream_api.hpp>
```

This header defines thin wrappers around the file streams provided by the C++ Standard Library. More specifically, it provides the following classes:

- `dyad::basic_ifstream_dyad`
- `dyad::ifstream_dyad`
- `dyad::wifstream_dyad`
- `dyad::basic_ofstream_dyad`
- `dyad::ofstream_dyad`
- `dyad::wofstream_dyad`
- `dyad::basic_fstream_dyad`
- `dyad::fstream_dyad`
- `dyad::wfstream_dyad`

When using DYAD, these file streams should be used in place of the file streams from the C++ Standard Library. The DYAD file streams should be directly used to do the following:

- Open files (with the file stream's `open` method)
- Close files (with the file stream's `close` method or destructor)
- Access the underlying C++ Standard Library file stream using the DYAD stream's `get_stream` method

All reading from and writing to files should be done using the underlying C++ Standard Library file stream. A simple example of using DYAD's C++ API in a producer application is shown below:

```
#include <dyad_stream_api.hpp>

void produce_file(std::string& full_path, int32_t* data, std::size_t data_size)
{
    dyad::ofstream_dyad ofs_dyad;
    ofs_dyad.open(full_path, std::ofstream::out | std::ios::binary);
    std::ofstream& ofs = ofs_dyad.get_stream();
    ofs.write((char*) data, data_size);
    ofs_dyad.close();
}
```

After replacing C++ Standard Library file streams with their DYAD equivalents, there is one final requirement to using the C++ API. When compiling your code, you must link the associated library (i.e., `libdyad_stream.so` or `libdyad_stream.a`). This library can be found in the `lib` subdirectory of the install prefix.

1.4 Running DYAD

There are three steps to running DYAD-enabled applications:

1. *Create a Flux key-value store (KVS) namespace*
2. *Determine the managed directories for each application*
3. *Load DYAD's Flux module*
4. *Configure and run the DYAD-enabled applications*

1.4.1 Create a Flux KVS Namespace

DYAD uses its own namespace in Flux's hierarchical key-value store (KVS) to isolate itself from the KVS entries from other Flux services. Thus, the first step in running DYAD is to create a KVS namespace. This namespace is used by DYAD to exchange file information (e.g., the Flux broker that "owns" a file) needed to synchronize the consumer application and transfer the file from producer to consumer. To create this namespace, run the following:

```
$ flux kvs namespace create <DYAD_KVS_NAMESPACE>
```

The namespace can be whatever string value you want.

1.4.2 Determine the Managed Directories for Each Application

To determine when to perform synchronization and data transfer, DYAD tracks two directories for each application: a **producer-managed directory** and a **consumer-managed directory**. At least one of these directories must be specified for DYAD to do anything. If neither are provided, the application will still run, but DYAD will not do anything.

When a producer-managed directory is provided, DYAD will store information about any file stored in that directory (or its subdirectories) into a namespace within the Flux key-value store (KVS). This information is later used by DYAD to transfer files from producer to consumer.

When a consumer-managed directory is provided, DYAD will block the application whenever a file inside that directory (or subdirectory) is opened. This blocking will last until DYAD sees information about the file in the Flux KVS namespace. If the information retrieved from the KVS indicates that the file is actually located elsewhere, DYAD will use Flux's remote procedure call (RPC) system to ask DYAD's Flux module to transfer the file. If a transfer occurs, the file's contents will be stored at the file path passed to the original file opening function (e.g., `open`, `fopen`).

Before running the following steps, determine the producer- and/or consumer-managed directories for each application. These directories will need to be provided to the commands in the next steps.

Note: When opening or closing a file not in the producer- or consumer-managed directories, DYAD will simply open or close the file. DYAD changes the behavior of opening or closing only the files in the managed directories.

1.4.3 Load DYAD's Flux Module

The next step in running DYAD is to load DYAD's Flux module. The module is the component of DYAD responsible for sending files from producer to consumer. Once loaded, this module will run whenever its associated Flux broker receives a relevant remote procedure call from a DYAD-enabled consumer. To load the module, first, determine where `dyad.so` is located. This should normally be `<PREFIX>/lib/dyad.so`. Once you have found the path to `dyad.so`, you can load the module on the current Flux broker using:

```
$ flux module load path/to/dyad.so <DYAD_PATH_PRODUCER>
```

The `dyad.so` module takes a single command-line argument: the producer-managed directory. The producer uses this directory as the root from which the module will look for files to transfer.

Note that the command above will only load the module on the Flux broker on which the command is run. This can be an issue if you are submitting jobs because you will not know on which broker your jobs will be run. As a result, it is **highly** recommended that you launch the DYAD module on all brokers in your Flux instance. You can do this by running:

```
$ flux exec -r all flux module load path/to/dyad.so <DYAD_PATH_PRODUCER>
```

1.4.4 Configure and Run the DYAD-Enabled Applications

Once the KVS namespace and DYAD module are set up, the DYAD-enabled applications can be run. To run a DYAD-enabled application, simply run your application as normal with certain environment variables set. A table containing the current environment variables recognized by DYAD is shown below.

Name	Type	Required?	Default	Description
`DYAD_KVS_NAMESPACE`	String	Yes	N/A	The Flux KVS namespace that DYAD will use to record or look for file information
`DYAD_PATH_PRODUCER`	Directory Path	Yes[1]	N/A	The producer-managed path of the application
`DYAD_PATH_CONSUMER`				The consumer-managed path of the application
`DYAD_SHARED_STORAGE`	Integer	No	0	If 1 (i.e., true), only provide per-file synchronization of the consumer (i.e., no transfer)
`DYAD_KEY_DEPTH`[2]	Integer	No	3	The number of levels in Flux's hierarchical KVS to use within DYAD's namespace
`DYAD_KEY_BINS`[2]	Integer	No	1024	The maximum number of unique values for the keys associated with any given level of Flux's hierarchical KVS within DYAD's namespace

[1] For DYAD to do anything, at least one of `DYAD_PATH_PRODUCER` or `DYAD_PATH_CONSUMER` must be provided. Applications will still work if neither are provided, but DYAD will not do anything.

[2] Since the Flux KVS is hierarchical, the number of KVS levels (controlled by `DYAD_KEY_DEPTH`) and the size of each KVS level (controlled by `DYAD_KEY_BINS`) will affect the performance of DYAD. To obtain optimal performance, tune these values for your use case.

CHAPTER

TWO

ECP TUTORIAL: FEBRUARY 10, 2023

Material for the DYAD demo at Flux's ECP tutorial can be found here. Additional information about the demo will be added soon.

CHAPTER

THREE

PUBLICATIONS AND PRESENTATIONS

3.1 Posters

- J. Yeom, D. H. Ahn, I. Lumsden, J. Luettgau, S. Caino-Lores, and M. Taufer, “Ubique: A new model for untangling inter-task data dependence in complex HPC workflows,” in *2022 IEEE 18th international conference on e-science (e-science)*, 2022, pp. 421–422. doi: 10.1109/eScience55777.2022.00068. `Paper PDF` | `Poster PDF`

CHAPTER

FOUR

DEVELOPER GUIDE

Since DYAD is part of the Flux Framework, developers are expect to follow all rules and contribution guidelines specified in the Collective Code Construction Contract (C4.1).

Below are some additional links regarding contributing, code styling, and commit etiquette:

- The 'Contributing' Page from the Flux Framework's ReadTheDocs
- The Flux Coding Style Guide (used for C code)
- The black Coding Style Guide (used for Python code)

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search