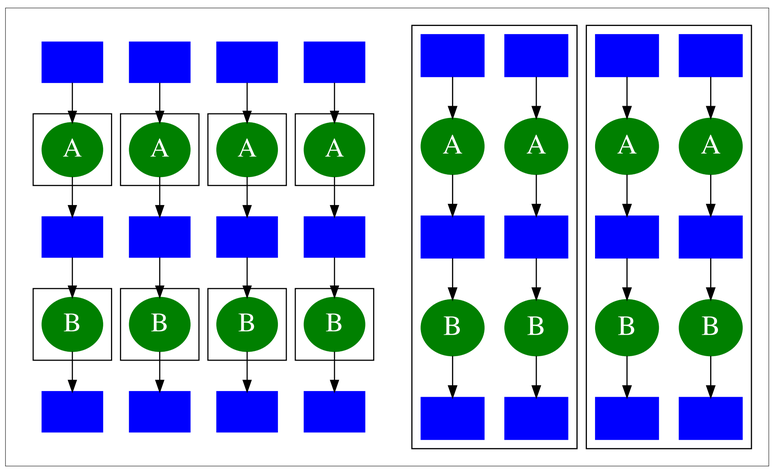
A First Look at the JX Workflow Language

Tim Shaffer, Kyle M.D. Sweeney, Nathaniel Kremer-Herman, and Douglas Thain

Workflow Partitioning

Workflow partitioning is the process of splitting a workflow graph into sub-graphs, such that each sub-graph will become a discrete batch job in the target execution system. The most appropriate partitioning depends on many properties of the workflow graph, such as the size of data objects and the execution time of tasks, as well as the performance properties of the execution system.



On the left, a fine-grained scheme assigns each task to its own partition. This is the most conservative approach, and usually the default for workflow managers. On the right, coarser partitions group multiple tasks together.

Evaluation

We explored schemes for partitioning Lifemapper, a distributed biodiversity modeling application. As a high-throughput application, Lifemapper offers significant freedom in organizing computation beyond simply following data dependency relationships. We observed that the granularity at which we distribute pieces of the workflow has a significant impact on its overall behavior.

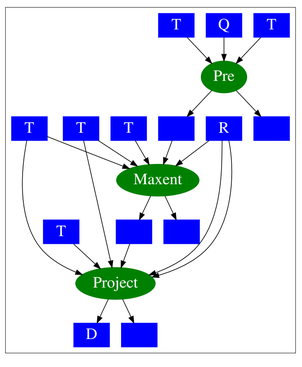
We measured the behavior of Lifemapper under two different partitioning schemes and ran the application on the IU-Jetstream, ND-Condor, and SDSC-Comet execution sites. We observed substantial differences in performance in terms of execution time, node-local storage, and data transfer between configurations when running on the same execution site. Across all three computing sites, there were similar trends in reduced data transfer and execution time with coarser workflow partitioning.

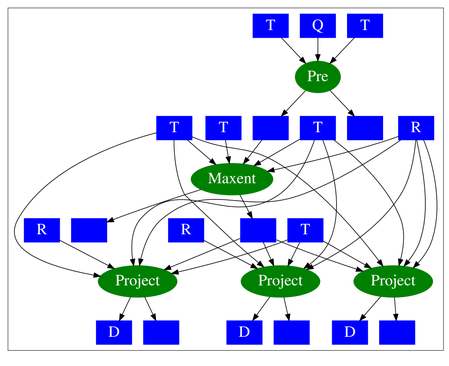
Summary of Execution Sites

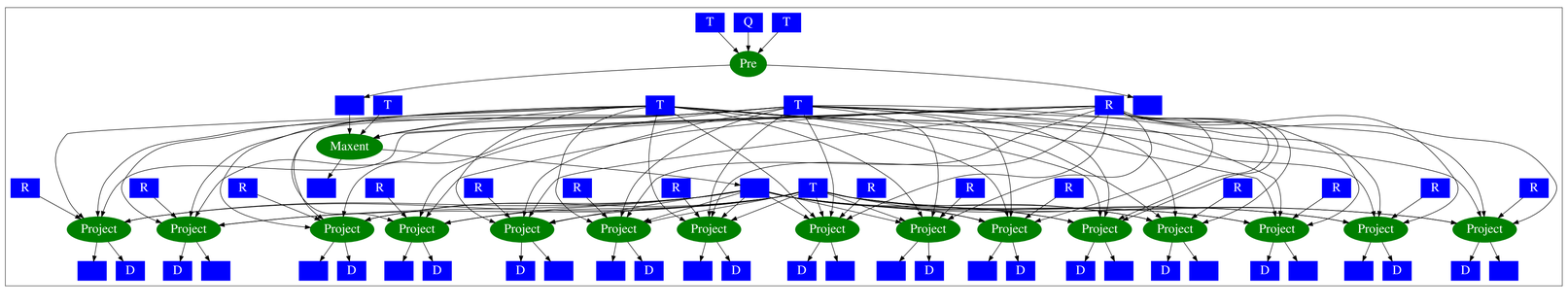
	Cores/ Worker	Total Cores	Memory	Peak Network
ND-Condor	12	∼1200	20 GB	1 GB/s
IU-Jetstream	10	110	30 GB	10 GB/s
SDSC-Comet	12	∼1200	20GB	InfiniBand

Lifemapper queries at three different scales

In each diagram, boxes represent files with arrows connecting to tasks, represented as ovals. Here **Q** denotes query data, **T** denotes tools such as Python scripts and Java JAR files, **R** denotes common reference data, and **D** denotes output data. The upper left workflow shows a small query against a single data layer for the *Heuchera* data set. The next workflow to the right shows a larger query against the *Saxifragales* data set with three data layers. Below, we duplicated layers to create a twelve-layered query against *Saxifragales*.



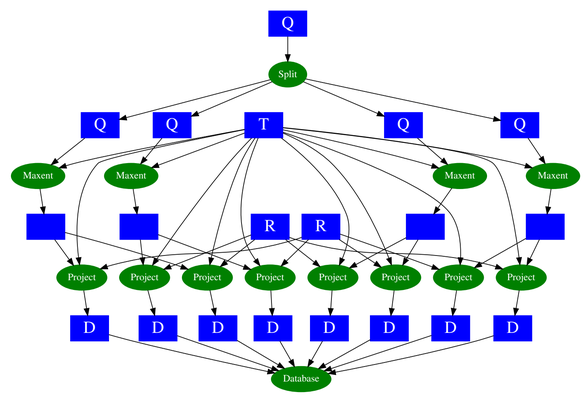




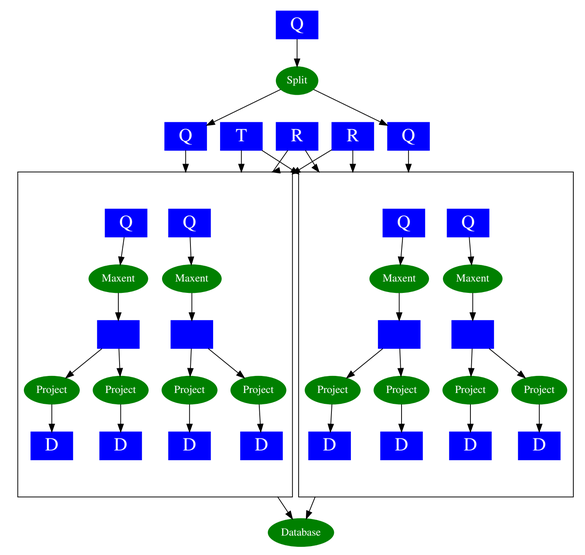
Partitioning Schemes for Lifemapper

These simplified workflows consist of four small taxa. The labels have the same meaning as above. Unlabeled boxes are intermediate files that are discarded after the workflow completes. With the Fine-Grained configuration, There is no additional structure within the workflow beyond data dependencies. With the Coarse-Grained configuration, The taxa are arranged into two partitions. Each partition becomes a task in the high level workflow.

Fine-Grained Model



Coarse-Grained Model



JX: JSON eXtended

We developed JX (JSON eXtended) as a language for expressing workflows that allows for easy manipulations to the structure and partitioning of a workflow. JX extends a JSON representation of the workflow by supporting a subset of Python expressions, allowing for a concise intermediate representation that expands to a normal JSON document. Templates in JX can expand to complicated nested workflow structures based on parameters, allowing flexible changes to a workflow's partitioning scheme.

The following workflow "template", when expanded, expresses an entire map-reduce type workflow. The expanded workflow includes a rule for each of the *N* input files, and a reduce step that takes all *N* intermediate files and produces the final output. This template exposes the structure of the workflow as a parameter. These parametric templates are the primary way JX allows for flexible changes in the organization and partitioning scheme of a scientific workflow.

Complete JX Workflow

```
{
  "rules": [{
    "inputs":
      ["split." + str(i)],
    "outputs":
      ["out." + str(i)],
    "command":
      "./process.sh split." + str(i),
  } for i in range(N)] + [{
    "inputs":
      ["out." + str(i) for i in range(N)],
    "outputs":
      ["result.dat"],
    "command":
      "./reduce.sh out.*",
  }]
}
```

Make Rule

```
proj/I_japonica.asc: I_japonica.csv
    ./project.sh I_japonica.csv proj
```

The GNU Make format is well-known, compact, and easy for novices to write, but it has some syntactic limitations. It is also difficult to add additional data or fields to a rule in a programmatic way, which is needed to handle partitioning and other workflow transformations. A JSON representation is easier to generate and parse via script, but still very verbose.

Single Task Expressed in JSON

```
{
  "inputs": ["I_japonica.csv"],
  "outputs": ["proj/I_japonica.asc"],
  "command":
    "./project.sh I_japonica.csv proj"
}
```

An entire workflow could be expressed as a sequence of plain JSON records like the above. JX expands upon JSON to allow compact patterns to expand into JSON rules. JX supports variable substitutions when expanding a document, multiple data types such as numbers and lists, and common operators such as arithmetic and comparison. These operators function the same as in Python.

JX Rule with Variable Substitutions

```
{
  "inputs": [s + ".csv"],
  "outputs": ["proj/" + s + ".asc"],
  "command":
    "./project.sh " + s + ".csv proj",
}
```

To quickly generate a list of items (e.g. a range of outputs like `file.1 ...file.n`), JX supports Python-style list comprehensions. For `SAMPLES = ["I_japonica", "A_arboreum"]`, we can produce a pair of rules, one for each sample.

JX List Comprehension for Rules

```
[{
  "inputs": [s + ".csv"],
  "outputs": ["proj/" + s + ".asc"],
  "command":
    "./project.sh " + s + ".csv proj",
} for s in SAMPLES]
```

Since the connections between tasks in a workflow (inputs and outputs) are specified as lists, this gives substantial freedom in programmatically defining the structure of a workflow. In addition, the workflow itself is primarily a list of rules, making it possible to expand a large number of rules from a single template. As an example, consider a workflow that takes an input file (`INPUT`), splits it into `N` pieces, and runs a processing script on each.

JX Workflow Template

```
{
  "rules": [{
    "inputs": [INPUT],
    "outputs":
      ["split." + str(i) for i in range(N)],
    "command": "./split.sh " + INPUT,
  }] + [{
    "inputs": ["split." + str(i)],
    "outputs": ["out." + str(i)],
    "command":
      "./process.sh split." + str(i),
  } for i in range(N)]
}
```

Combining these features, JX allows us to treat a sub-workflow as a job. A JX template serves to define the structure of the sub-workflow subject to some parameters. We can reuse the same definition to programmatically produce a large number of rules that fit the concrete arguments to each invocation. We can pass the chosen partitioning into the high-level workflow, which can then repeatedly expand the partition template and produce each sub-workflow as it is executed. The following shell command could be a rule in the Makeflow workflow management system which expands the previous example template and runs the resulting workflow as a task.

Invoking Makeflow Using a JX Template

```
makeflow --jx-define 'INPUT="heuchera"' \
         --jx-define N=100 \
         template.jx
```

Acknowledgments

Visit `http://ccl.cse.nd.edu/` to learn more!
This work was supported in part by National Science Foundation grant OCI-1148330.