Copernicus: a dataflow tool for parallel computation

Peter Kasson
Departments of Molecular Physiology and Biomedical Engineering
University of Virginia
Motivation: problems in simulation and sampling

The simulation problem:
How to do this in a manner that is robust, reproducible, and involves reusable components
Dataflow...”a series of pipes”?

- Data flow and control flow as two ways of thinking about workflow
- Dataflow explicitly represents the inputs and outputs; dependencies are implicit. Control flow does the reverse.
- Explicit representation of input and output for each task makes dependency relationships obvious: a given task “activates” when its inputs are present.
- Tasks “activate” when all inputs are ready

“Data-centric” view
- Used in many real-time processing systems (Google Millwheel, Twitter Storm)
Why this makes a difference

Control flow:
1. Do \textit{a}
2. Do \textit{b}
3. Do \textit{c}

Data flow:
\textit{a}(static inputs)
\textit{b}(\textit{a}.outputs)
\textit{c}(\textit{a}.outputs)

\textit{c} will automatically run in parallel with \textit{b} because it doesn’t depend on \textit{b}. 
Tensorflow as a recent example of a data-flow architecture

• Illustrates both pros and cons of dataflow

• Tensorflow is powerful, efficient, but not as “natural” to program for people who are used to thinking in a control-flow formalism

• More recently, a set of “simple” front-ends have been developed to allow basic access to Tensorflow functionality for people who don’t want to tackle the learning curve up front.

Inception training on tensorflow

Tensorflow Playground.

github.com/tensorflow

cloud.google.com
Adaptivity as a key concept in advanced workflows

Static workflows: The computations to be performed are pre-specified in advance. Run a, run b, run c.

Adaptive workflows: The number, sequence, and identity of computations to be performed depend on the results along the way.

Run a; if a.result > 5, run b else run c
Do run b while not a.result

Does everyone need adaptivity all the time? No.
What does adaptivity gain you? High-level algorithms can be expressed as workflows.

(Intermediate workflows exist that have for loops and/or while loops but not all adaptive features)
Copernicus design: general but designed in particular for molecular simulations & sampling

- Copernicus nodes (functions) can change the structure of the graph as well as operate on their inputs.
- This ability to modify the graph means that relatively complex logic can be encoded at the data-flow network level.

- Practically, statistical estimation problems for molecular simulation are high-level algorithms, and these can be either contained wholly within modules (encapsulation) or encoded at the data-flow network level.
Design goals, use cases, where we’re going

• Copernicus was designed to allow scalable execution of adaptive workflows, targeting molecular simulation but with a general plugin design.

• We have used it for simple “parallel pipeline” simulations, free energy calculations, “swarms” of simulations, and Markov State Model calculations.

• The goal is to provide composable, reusable “modules” that are easy to link together yet also enable users to write their own modules easily.

• Right now, module composition is relatively easy. Writing new modules and plugins is harder than we’d like.

• Planned development for Copernicus includes simplifying the process of writing new modules (via subclassing) and overlaying a simpler API to enable lightweight execution of parallel adaptive workflows.
Decentralized structure--not a “push” framework

- Copernicus encodes a **distributed execution engine**--a “server” coordinates tasks, while “workers” execute tasks. *Workers do not have to be local.*

- Workers request tasks rather than servers launching workers.

- Goal: run on local resources, run on one or more supercomputer clusters, run on cloud resources, run anywhere you have cycles.
So how does this work?

1. **server** maintains the dataflow network (manages workflow), translates this into tasks, stores results.

2. **command-line client** controls the server. lightweight, can run from anywhere (secure keys)

3. **workers** do the computational heavy lifting. These run on a target compute resource (or many), contact the server to request tasks, run tasks, return data.
How will we run Copernicus today? How might you run it at home?

- Now: server runs as job on HPC compute node. Workers also run as jobs on HPC compute node(s).

- Other possibilities: run server in home laboratory or on a different cluster. Run workers as jobs on (multiple) HPC platforms.

- Workers just need to be able to contact the server. Relay servers can bundle work and help bridge firewalled communication. The server can restart robustly but should be persistently available. Data reside on the server file system (can specify location).
Let’s get started...

Agenda:

1. Get files
2. Launch server
3. Tell it to do stuff (tutorials)
4. Connect workers
0. Log in to Blue Waters

• You should have received account details

• ssh to username@bwbay.ncsa.illinois.edu

• Enter your password
1. Get files

1. Clone Copernicus (a version for this demo)

   git clone https://github.com/kassonlab/copernicus.git

2. Clone scripts for setup and tutorials

   git clone https://github.com/kassonlab/copernicus-images

3. Run setup script:

   on Blue Waters:

   cd copernicus-images/bw

   ./cpc_setup.sh

   The setup script will ask you to choose a password for the cpc-admin account
1b. Key pieces (review) and where we run them

- **“client”** (cpcc)
  This is the command-line interface. We’re running it from the Blue Waters login node, but it can live anywhere. It doesn’t do significant computation.

- **“server”** (cpc-server)
  The client talks to a server. This is a long-running and restartable process that manages all the work to be done (and the dataflow graph, etc.) and stores the data. We’re running this on a Blue Waters compute node. It does some computation but primarily needs storage and network. More advanced architectures can involve relay servers on each cluster. The server does not “push” work; instead, workers “pull” work.

- **“workers”** (cpc-worker)
  These actually run computations. Workers can run anywhere that can reach a server. We submit worker jobs to the queue to run on Blue Waters compute nodes. Workers are similar to a “pilot” abstraction c.f. Radical cybertools.
2. Launch the server

On Blue Waters: (note other directories with script versions for other systems. It’s not hard to adapt your own.)

qsub cpc-server.sh

Once the server is running, we’re ready to go.

Key commands in the launch script:

cpc-server start
cpc-server bundle -o local_bundle.cnx

# this creates a “connection bundle” containing the server location as well as keys
Some Copernicus concepts

**modules and instances**

**Modules** provide definitions of reusable components. You import a module much as you do in Python. Modules will provide function definitions. We will use pre-written modules that ship with Copernicus, but you can write your own.

Modules only give function definition; you create an **instance** of a function to actually use it.

**Encapsulation**: As demonstrated in later Copernicus tutorials, modules can encapsulate more complicated networks of functions. That way a complex procedure can be presented and called as a simple high-level function.

**run-once**: Data-flow networks handle dependencies by running a function instance once all its inputs are satisfied. Copernicus uses a run-once guarantee: unless specifically reset, a function will not rerun if its input changes. (There are other ways to handle loops)
First tutorial. Let’s go through it.

```bash
# set SERVER variable
cpcc add-server $SERVER $SERVER_PORT # specify the server

# log in to the server
cpcc login cpc-admin

# let’s start a new project
cpcc start md_simulations_test

# import a module. Here Gromacs
cpcc import gromacs

# create an instance of “grompps”, the simulation input generator
cpcc instance gromacs::grompps grompp

# create an instance of “mdruns”, the simulation run engine
cpcc instance gromacs::mdruns mdrun

# activate the network. right now, nothing is connected yet.
cpcc activate

# compound transaction

cpcc transact

cpcc setf grompp:in.conf[+] ~/copernicus/examples/mdrun-test/conf.gro

# set an input file. the [+]
# specifies the next item in an
# array. We can make a parallel
# pipeline here if we want.

cpcc setf grompp:in.mdp[+] ~/copernicus/examples/mdrun-test/grompp.mdp

cpcc setf grompp:in.top[+] ~/copernicus/examples/mdrun-test/topol.top

# this is the key for dataflow.
# We connect an output to an input.
# This specifies sequence.
# finish the compound transaction

cpcc connect grompp:out.tpr mdrun:in.tpr

# print the command queue

cpcc commit

cpcc q

cpcc s
```

# print the status
Running the tutorial

1. Make sure the server is running

   `rm ~/.copernicus/clientconfig.cfg # if you need to wipe your config file`

2. Set up the project and network (your choice of line-by-line or running the script)

3. “Commands” i.e. tasks are now queued. `mdrun` needs a worker to execute it.

4. Connect a worker: submit a worker job script

   `(qsub ~/copernicus-images/bw/cpc-worker1.sh)
   cpc-worker -d -c local_bundle2.cnx -wd $CPC_TEMP -q 20 smp -n 32`
   # run the worker with a connection bundle to specify the server and authenticate
   # in this case, we are running on a single node, so “SMP” is simplest.
   # We could also run MPI on a single node or across nodes for a single worker.
   # -q 20 means to quit after 20 minutes of no work
   # -n 32 means to use 32 cores. This can also be detected automatically
   # (a worker will try to use all available cores)
Looking at dataflow status

1. `cpcc s` gives status information

2. `cpcc get` gets information about a given function instance and its associated data e.g.

   ```
   > cpcc get mdrun:out
   mdrun:out: {
     edr : [ mdrun/mdrun_0/_run_0000/ener.edr ],
     trr : [ None ],
     xtc : [ mdrun/mdrun_0/_run_0000/traj.xtc ],
     log : [ mdrun/mdrun_0/_run_0000/md.log ],
     conf : [
       mdrun/mdrun_0/_run_0000/confout.gro
     ]
   }
   ```

   `cpcc getf mdrun:out:log[0]` returns the contents of the log file output.

   `cpcc save md_simulations_test foo.tar` saves the entire project to a tar file
Questions regarding the first tutorial?

`cpcc help is useful!`

`shut down the server: cpcc stop-server`
Getting a little more complicated:

1. More complex modules (encapsulating a bit of complexity and adaptivity)

2. Parallelism (do the same operation on a larger number of inputs in parallel)
   Advantage of dataflow here is that dependencies are resolved in a straightforward manner

3. (not shown here: more complex logic)
A more complex and encapsulated workflow: Markov State Model building

Markov State Model theory: If we have an underlying Markovian process with unknown states $i$ and transition probabilities $P_{ij}$,

And if we have a set of trajectories sampling this process (in this case molecular simulations),

We can estimate the states and probabilities by clustering trajectory snapshots to estimate states and counting transitions to estimate probabilities.

But we can do this much more efficiently if we use information along the way (preliminary estimates of the states and transitions) to drive sampling (where we start our trajectories). This is the dataflow problem.
A look at the MSM module with subnetworks
**MSM tutorial walk-through**

```bash
cpcc login cpc-admin
cpcc start alanine-dipeptide-msm
cpcc import msm

cpcc instance msm::msm_gmx_adaptive msm   #as before, create an instance

cpcc activate   # as before, activate the network

cpcc transact

# Set parameters for the MSM instance
# Descriptions can be obtained with cpcc info msm::msm_gmx_adaptive

cpcc setf msm:in.grompp.top copernicus/examples/msm/alanine-dipeptide-msm/topol.top

cpcc setf msm:in.grompp.mdp copernicus/examples/msm/alanine-dipeptide-msm/grompp.mdp

# set four different starting conformations as inputs

cpcc setf msm:in.confs[+] copernicus/examples/msm/alanine-dipeptide-msm/equil0.gro

cpcc setf msm:in.confs[+] copernicus/examples/msm/alanine-dipeptide-msm/equil1.gro

cpcc setf msm:in.confs[+] copernicus/examples/msm/alanine-dipeptide-msm/equil2.gro

cpcc setf msm:in.confs[+] copernicus/examples/msm/alanine-dipeptide-msm/equil3.gro


cpcc set msm:in.recluster 1.0     # every time another 1 ns of aggregate data is returned, recluster

cpcc set msm:in.num_microstates 100  # clustering happens in two stages. the first yields “microstates”

cpcc set msm:in.num_macrostates 10   # the second stage of clustering groups microstates into macrostates

cpcc set msm:in.num_sim 20        # start 20 simulations in the next round from each macrostate

cpcc set msm:in.lag_time 2

cpcc set msm:in.grpname  Protein  # parameter to do clustering on protein atoms only


cpcc set msm:in.num_generations 5  # run 5 rounds or “generations” of sampling and clustering

# commit the changes

cpcc commit
```

# this runs Markov State Model construction on a simple system, so it should be comparatively fast.
# In the first Copernicus paper, we used ~1500 cores and ~100 workers active at any given time
# to automatically build a similar MSM for a larger system. Scaling goes >10x higher at current time.
Time permitting, tutorial 3 shows a free-energy-perturbation calculation

- Again, encapsulated adaptive logic on how to optimally calculate a statistical quantity. Terminates upon convergence.
Network:
  connected to 0 other servers
  connected to 0 local workers
  0 workers in total in network
  1 server in total in network

Project fetest:
  435 function instances, of which
    435 in state 'active'
  no commands in queue
  no commands running

Project md_simulations_test:
  4 function instances, of which
    4 in state 'active'
  no commands in queue
  no commands running

Project alanine-dipeptide-msm (current project):
  551 function instances, of which
    551 in state 'active'
Working with Copernicus: high- and mid-level

Many data-flow networks can be constructed entirely using pre-defined modules and functions, so a user does not have to build these from scratch.

```
cpcc instance gromacs::grompps preproc
cpcc instance gromacs::mdruns md_sim
cpcc connect preproc.out.tpr md_sim.in.tpr
```

However, functions and modules are written in simple Python with Copernicus extensions and an accompanying XML schema. So the set of functions and modules is readily extensible by power users.
Where Copernicus is going

• Simpler creation of new modules via subclassing

• Lightweight high-level API to ease flexible, adaptive workflow construction

• Swappable runtime backends with some of the other systems you’ll hear about

• Questions: kasson@virginia.edu
More information:

- Project home page:  http://copernicus-computing.org/

- Documentation:  http://copernicus.readthedocs.org

- Source on GitHub:  https://github.com/gromacs/copernicus

- Papers:  Pronk et al., SC11
  Pronk et al., JCTC 2015
Acknowledgements

other Copernicus project leaders
• Sander Pronk
• Iman Pouya (KTH)
• Erik Lindahl (SU, KTH)

Kasson group members (computational)
• Bert Cortina
• Jennifer Hays
• Cara Broshkevitch
• your name here!

Other developers
• Björn Wesen (KTH)
• Marnus Lundborg (KTH)
• Patrik Falkman (formerly KTH)
• Grant Rotskoff (formerly KTH)
• Oliver Soell

Resources:

Funding:
NIGMS R01 GM098304 (Kasson)
STINT (KTH-UVa)
ERC and VR awards to Lindahl
Notes for other workshop/HPC systems

• Bridges: running server on compute nodes not yet supported (PSC sysadmins working on networking issue). For the moment, can run lightweight server process on head node. Change client and server port numbers to default + your compute ID # to avoid collisions.

• In general, setting up Copernicus on a new HPC system is *much* simpler if the Copernicus project server resides elsewhere. Then the only things to navigate are making sure the MPI commands work, the nodes can talk to the server, and that the correct executables are available.