Work In Progress: FLUX Runtime Environment

FLUX Meeting

Sep 17 2013

Dong H. Ahn

Lawrence Livermore National Laboratory

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC.
LLNL-PRES-643938

Work in progress on two initial phases on run-time areas

- Phase I – conceptualized next-generation resource management challenges and design space
- Phase II – gaining experiences and insights by producing prototype software
- Today, an interim report on the runtime system
  - Highlight key concepts in runtime (Phase I)
  - Experiences with early prototyping efforts (Phase II)

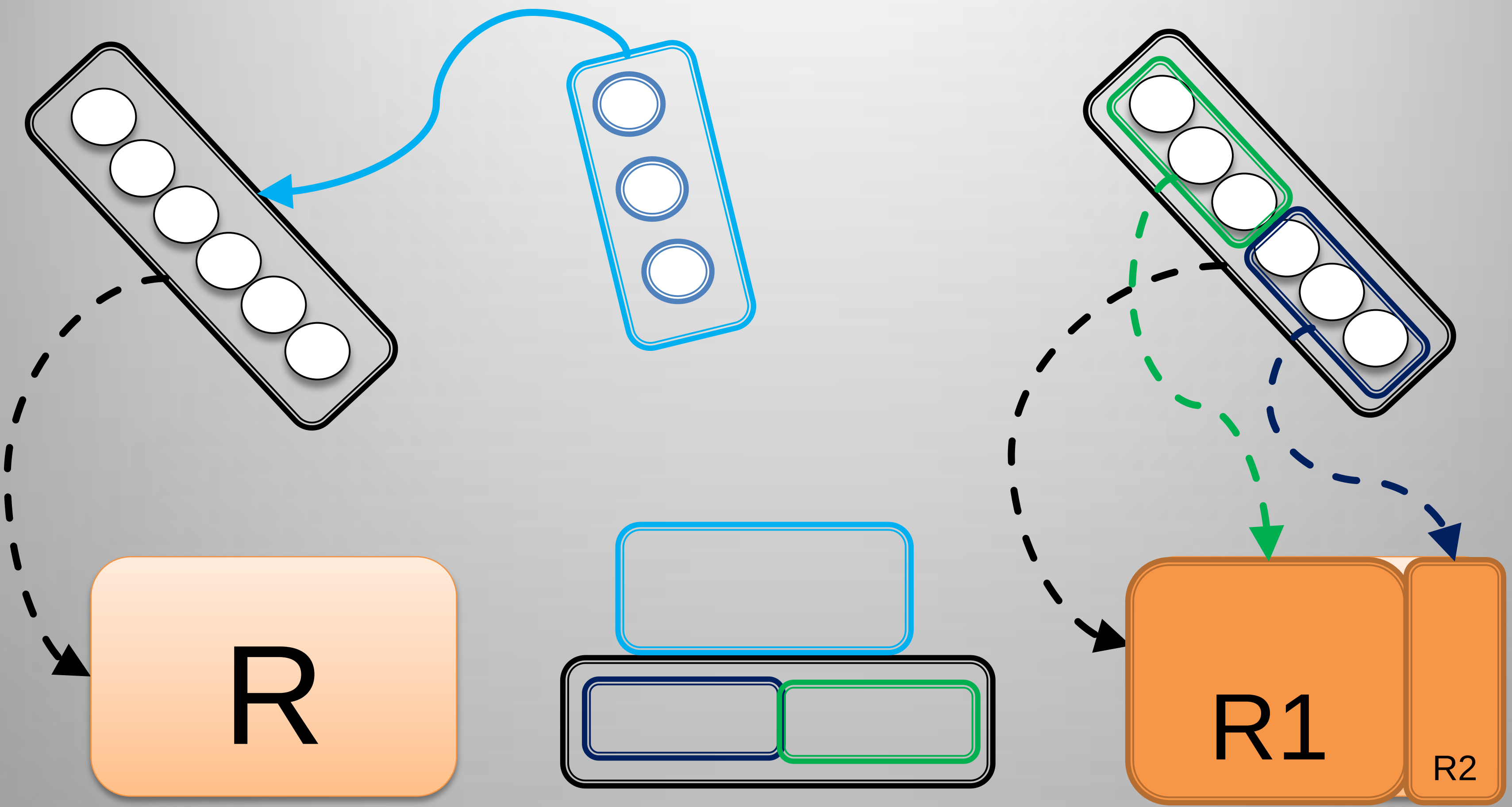
A scalable run-time to execute various transactions in a job in the new paradigm

- A paradigm shift in resource management
  - Capable of imposing complex resource bound
  - Highest operational efficiency at any level across the computing facility
- Scheduler sets the overall bound for resources and the duration for a job—now what?
- Workload Runtime And Placement (WRAP) thrust area
  - A job consists of various transactions
  - Need a powerful run-time system to execute a wide range of transactions of a job efficiently while under the overall bound

The new paradigm needs new ways to organize and group processes of a job

- The traditional approach models transactions of a job as a set of compute steps (e.g., job steps)
- Limits next-generation computing in many ways
- Designing WRAP after this model would be an under-design

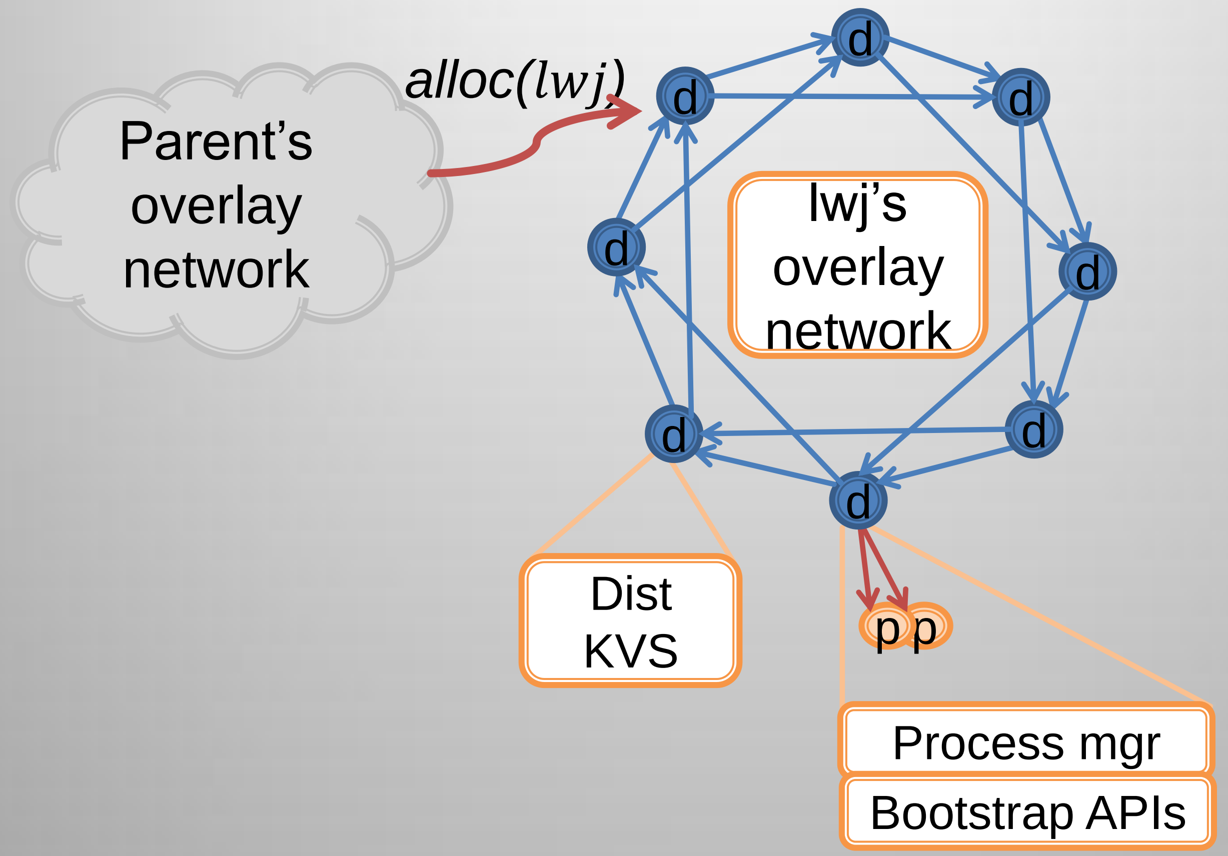
Lightweight job (LWJ) as our model to capture a transaction—i.e., grouping processes



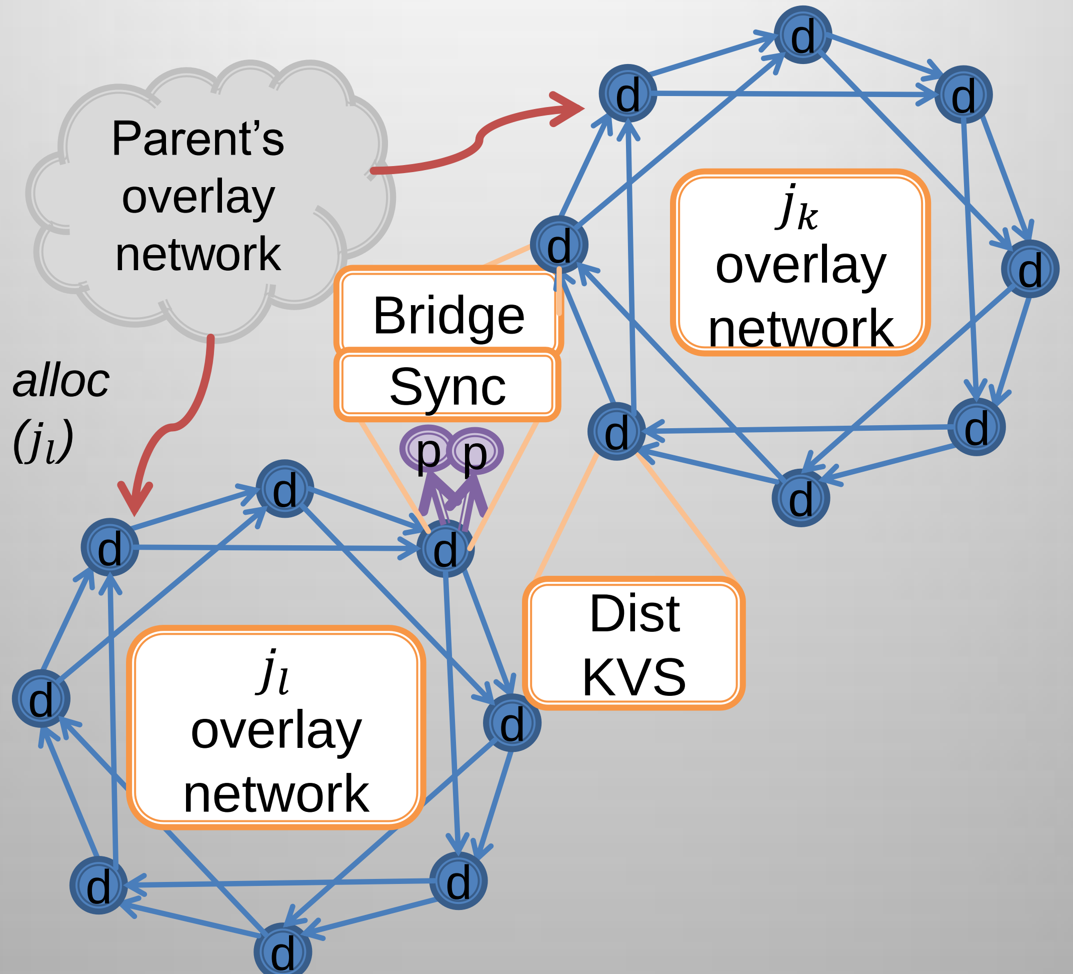
LWJ enables us to express various run-time services concisely under the new paradigm

- Serves as group identifiers to relate a group of processes to resources as well as to other groups of processes
- Resource allocation and elasticity: ***alloc(lwj, c)***, ***realloc(lwj,c)***, and ***release(lwj, c)***
- Process management/confinement: ***launch(lwj)***, ***destroy(ljw)***
- Synchronization: ***sync(lwj(i), lwj(k))***
- Resource discovery and provenance: ***query(lwj)***, ***record(lwj)***

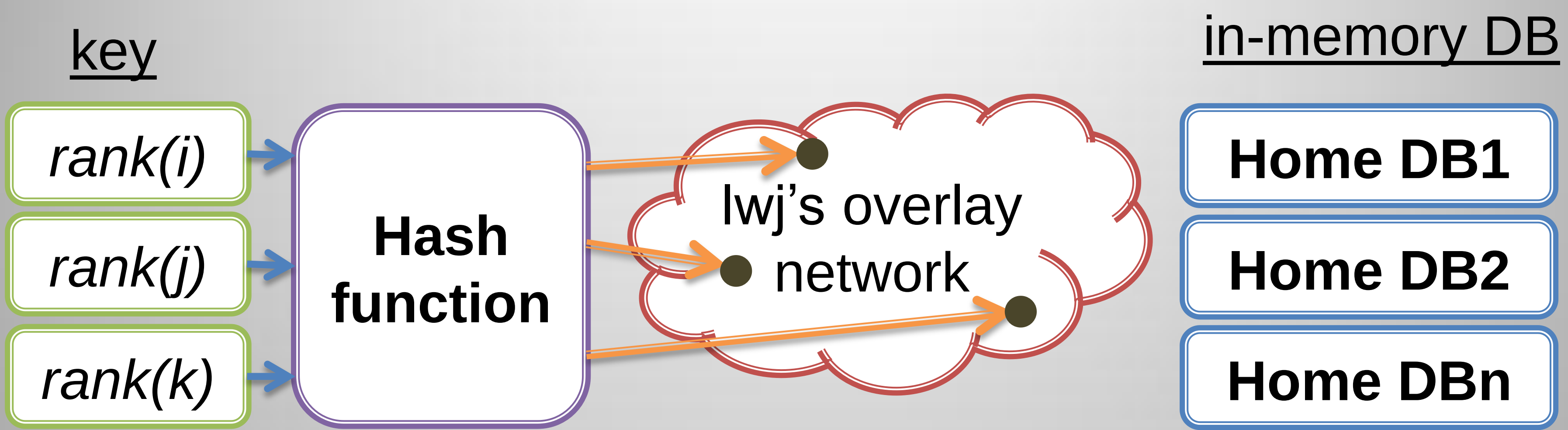
The base WRAP architecture builds on comms. framework and distributed key-value store



We can easily extend the base architecture to implement *sync*



DKVS is scalable distributed shared memory for an LWJ and its descendants



- Get/put for data access
- Collective Fence for memory consistency

lwj(1)::resource	cores (128)	power(10KW)	lic (10 tokens)	…
lwj(1)::rank(10)	host(1)	pid(345)	port(445)	
lwj(1)::record	info1	info2	…	…
lwj(1)::lwj(2)::resource	cores(64)	power(4KW)	lic (2 tokens)	

Scalable KVS allows ease integration with various types of LWJs beyond MPI

- PMI 1, 2 will be a very thin layer on top of KVS
- PMGR, PMGR Collective, COBO, LaunchMON, and LIBI use essentially the same bootstrapping technique that KVS can easily enable
- Our plug-ins for these well-known bootstrappers will serve as the reference implementation
- Other types of LWJs can write their own plug-ins for ease integration into WRAP

Phase II aims to strategic prototyping to gain insight into the detailed design

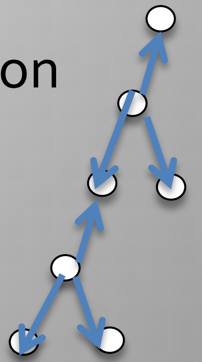
- Goal
  - Gain experiences for the final design
  - Prove that FLUX's rich run-time will significantly boost user productivity
- Plan
  - Bring up COBO on top of KVS
  - Native LaunchMON API Support
  - Bring up MRNet by porting the LIBI interface
  - Bring up STAT on top of LaunchMON and MRNet
  - Bring up and enhance SPINDLE
- Show FLUX can bring in rich sets of scalable productivity tools
- Show FLUX can start up massive applications
  - By enabling seamless integration with a specialize software system

KVS service via CMB solved a notorious chicken-and-egg problem for bootstrapping

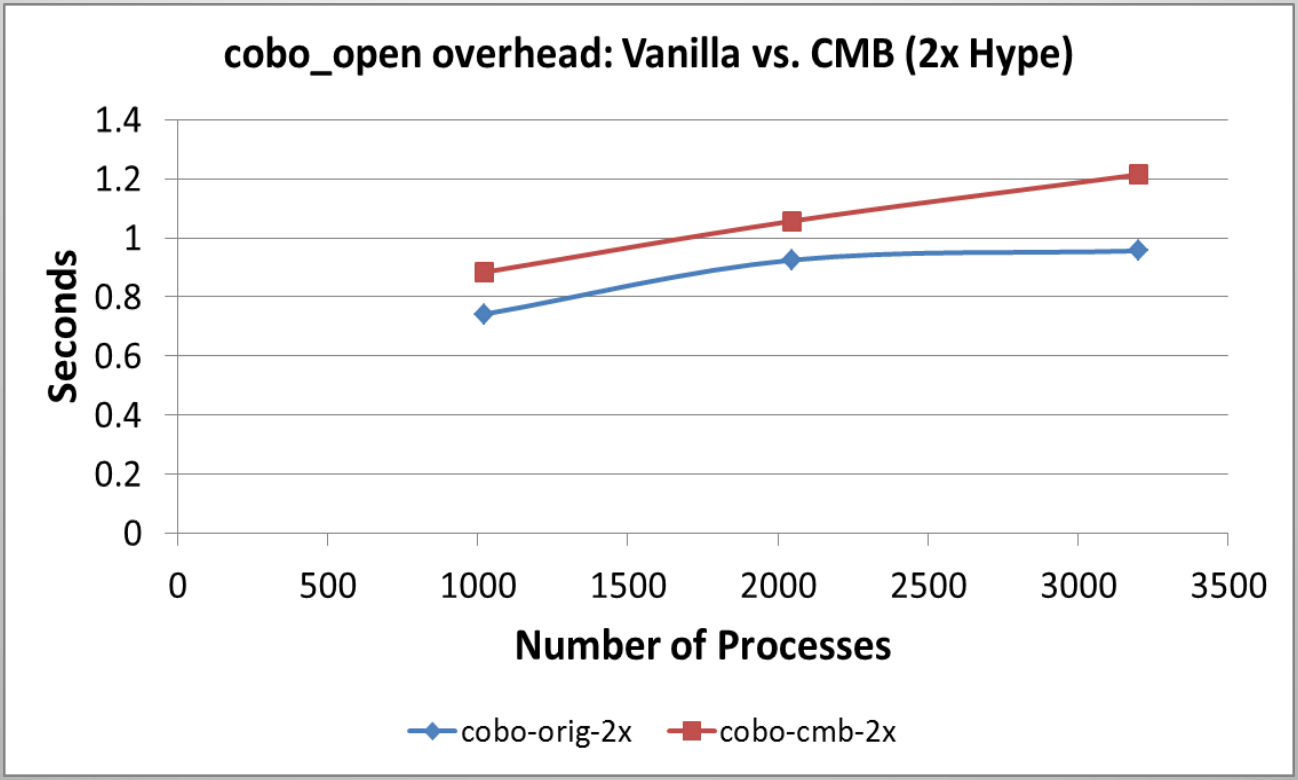
- COBO: a bootsrapper used in scalable tools infrastructure
- A simple TCP-based tree-based overlay network
- Chicken-and-egg problem: no common mechanism exists to bootstrap this bootstrapper!
  - Initial version used all-send-to-one algorithm—not scalable
  - Current version uses ad hoc port-range scheme—scalable but not ideal
- Use CMB's KVS service to address this problem

CMB/KVS-based COBO Connection algorithm and implementation

- Extended its tree open call in pmgr_collective_client_tree.c
  - #include "cmb.h"
  - int pmgr_tree_open_cmb (pmgr_tree_t *t, …
- Each spawned process creates a key-value tuple (key=its rank, value= ip:port) and push it to KVS
  - cmb_kvs_put (cmb_cxt, keystr, valstr) /* keystr=rank, valstr=ip:port */
  - cmb_kvs_commit (cmb_cxt, &error_cnt, &put_cnt)
  - cmb_barrier (cmb_cxt, "topen-cmb", ranks) /* named barrier */
- Each process computes its position in the binary tree based on it rank and size and fetch ip/port of its parent and children:
  - res_val = cmb_kvs_get (cmb_cxt, (const char *) keystr)
- Then, a simple two-step connection algorithm



Initial performance under a single KVS server (no KVS optimization)

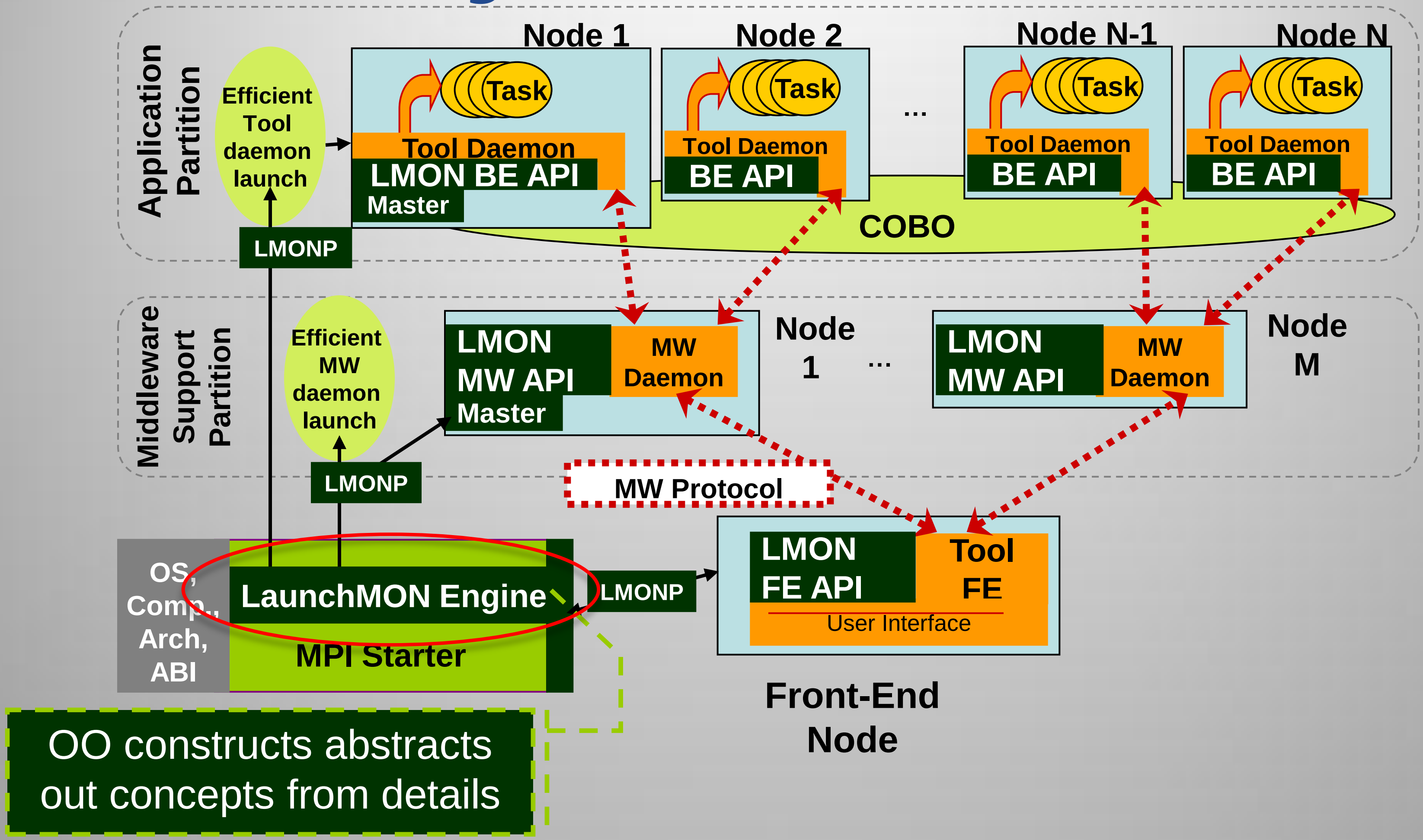


Code at the cobo-tester directory in git@github.com:chaos/ngrm.git

Native LaunchMON API support under FLUX

- LaunchMON: tool daemon launching infrastructure
- Used by many scalable tools
- For high portability, it uses the MPIR process acquisition interface (a de factor standard RM interface for debuggers)
- Requires tracing a MPI starter process and this makes it difficult to compose multiple tools
- Many problems can be addressed when LaunchMON is more deeply into the resource manager

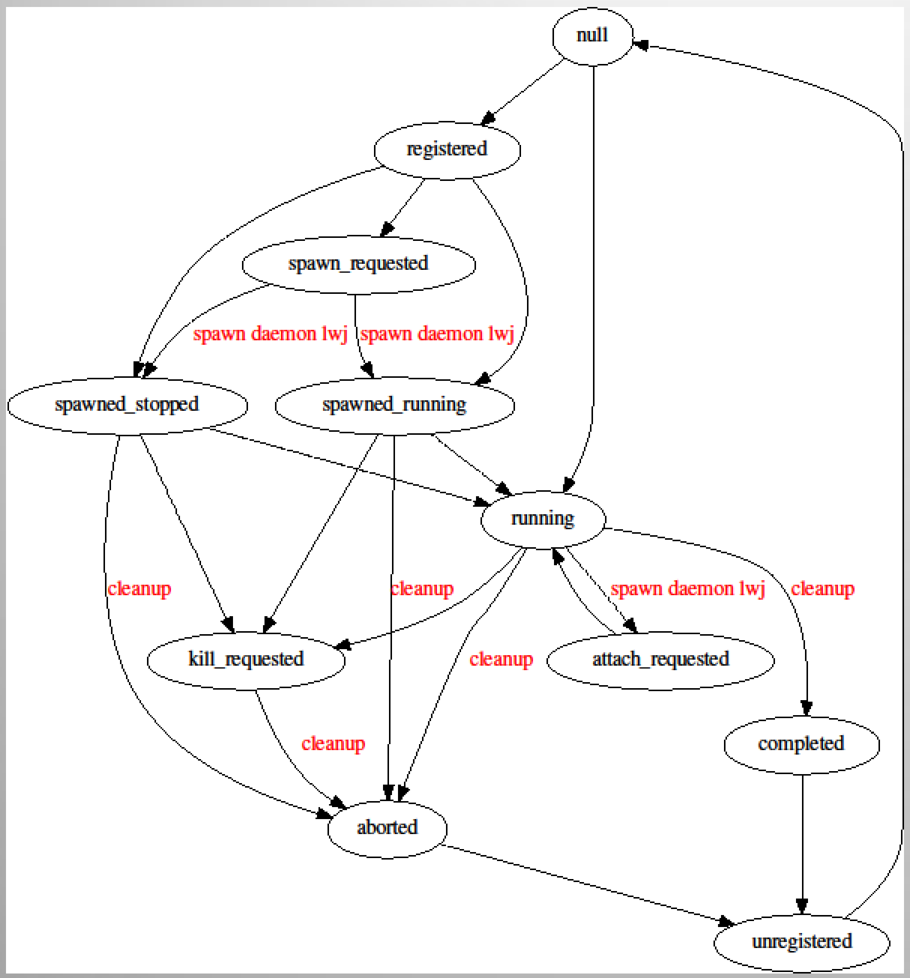
Adding a new LaunchMON engine that interacts with RM through its API



FLUX API mockup committed

enum	_flux_rc_e { FLUX_OK, FLUX_ERROR, FLUX_OK, FLUX_ERROR }
enum	_flux_lwj_event_e { status_null = 0, status_registered, status_spawn_requested, status_spawned_stopped, status_spawned_running, status_running, status_attach_requested, status_kill_requested, status_aborted, status_completed, status_unregistered, status_reserved, status_null = 0, status_registered, status_spawn_requested, status_spawned_stopped, status_spawned_running, status_running, status_attach_requested, status_kill_requested, status_aborted, status_completed, status_unregistered, status_reserved }
flux_rc_e	FLUX_init ()
flux_rc_e	FLUX_update_createLWJCxt (flux_lwj_id_t *lwj)
flux_rc_e	FLUX_update_destoryLWJCxt (flux_lwj_id_t *lwj)
flux_rc_e	FLUX_query_pid2LWJId (const char *hn, pid_t pid, flux_lwj_id_t *lwj)
flux_rc_e	FLUX_query_LWJId2JobInfo (const flux_lwj_id_t *lwj, flux_lwj_info_t *info)
flux_rc_e	FLUX_query_globalProcTableSize (const flux_lwj_id_t *lwj, size_t *count)
flux_rc_e	FLUX_query_globalProcTable (const flux_lwj_id_t *lwj, MPIR_PROCDESC_EXT *pt, size_t count)
flux_rc_e	FLUX_query_localProcTableSize (flux_lwj_id_t *lwj, const char *hn, size_t *count)
flux_rc_e	FLUX_query_localProcTable (const flux_lwj_id_t *lwj, const char *hn, MPIR_PROCDESC_EXT *pt, size_t count)
flux_rc_e	FLUX_query_LWJStatus (flux_lwj_id_t *lwj, int *status)
flux_rc_e	FLUX_monitor_registerStatusCb (const flux_lwj_id_t *lwj, int(*cb)(int *status))
flux_rc_e	FLUX_launch_spawn (const flux_lwj_id_t *me, int sync, const flux_lwj_id_t *target, const char *lwjpath, char *const lwjargv[], int coloc, int nn, int np)
flux_rc_e	FLUX_control_killLWJs (const flux_lwj_id_t target[], int size)
flux_rc_e	error_log (const char *format,...)

Good progress made in new LaunchMON engine implementation



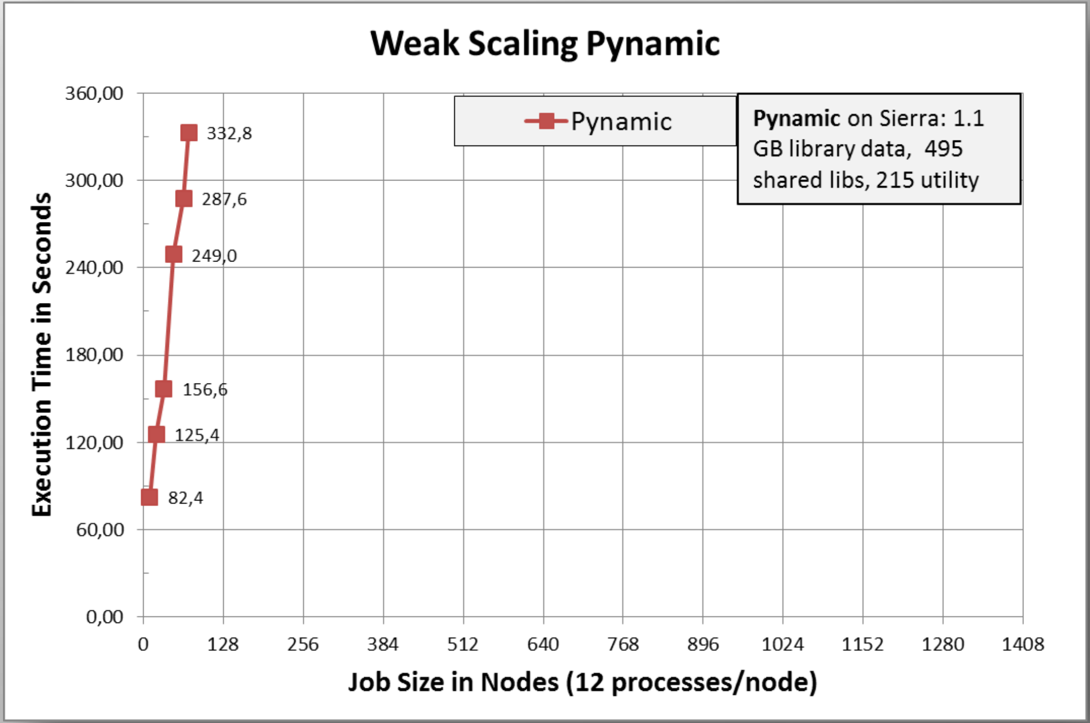
- Event-based system
- Many of the actions on LWJ state changes implemented
- Need to complete all of the actions and wire-up with FEN API through LMONP
- Mostly importantly, bring this up when the real FLUX API is in place

Phase II aims to strategic prototyping to gain insight into the detailed design

- Goal
  - Gain experiences for the final design
  - Prove that FLUX's rich run-time will significantly boost user productivity
- Plan
  - Bring up COBO on top of KVS
  - Native LaunchMON API Support
  - Bring up MRNet by porting the LIBI interface
  - Bring up STAT on top of LaunchMON and MRNet
  - Bring up and enhance SPINDLE
- Show that FLUX can bring in rich sets of scalable productivity tools like STAT
- Show that FLUX can start up massive applications
  - By enabling seamless integration with middleware software like SPINDLE

Dynamic linking and loading causes major disruption at large scale

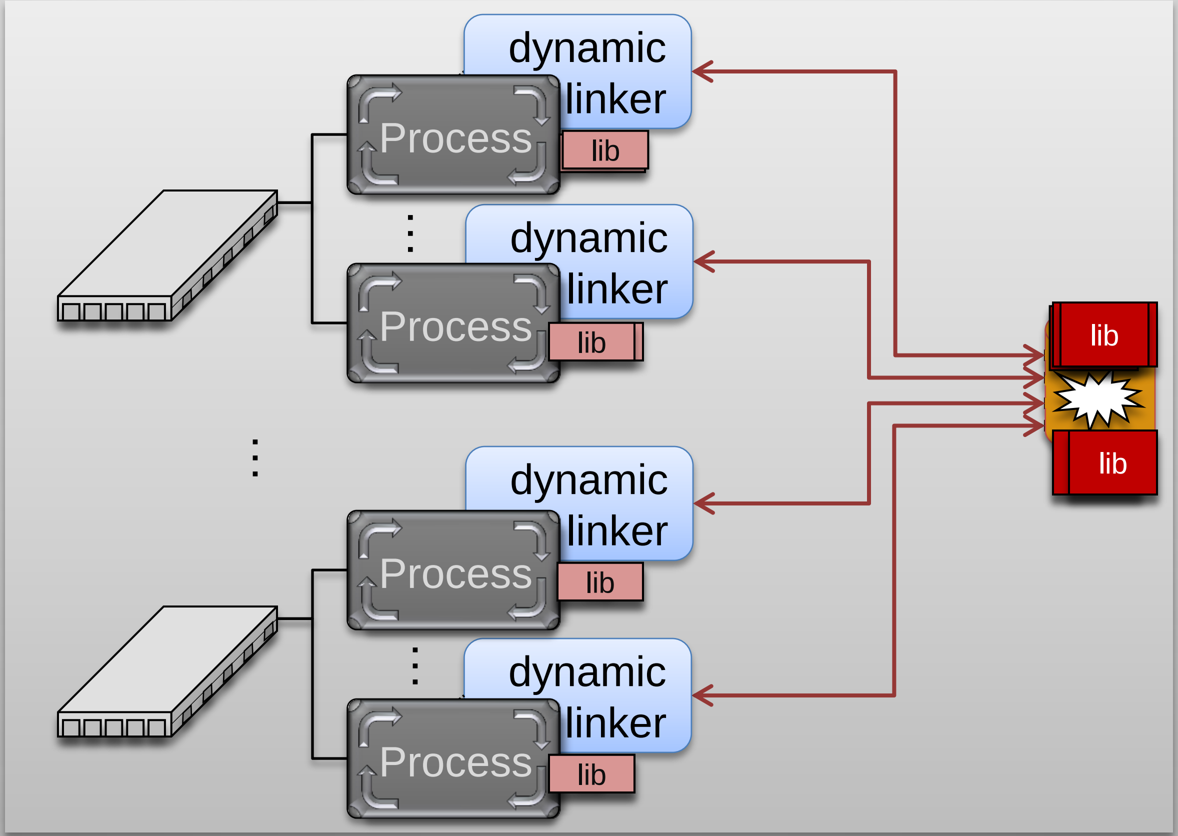
- **Multi-physics applications at LLNL**
  - 848 shared library files
  - Load time on BG/P:
    - 2k tasks → 1 hour
    - 16k tasks → 10 hours
- **Pynamic**
  - LLNL Benchmark
  - Loads shared libraries and python files
  - 495 shared objects → 1.1 GB



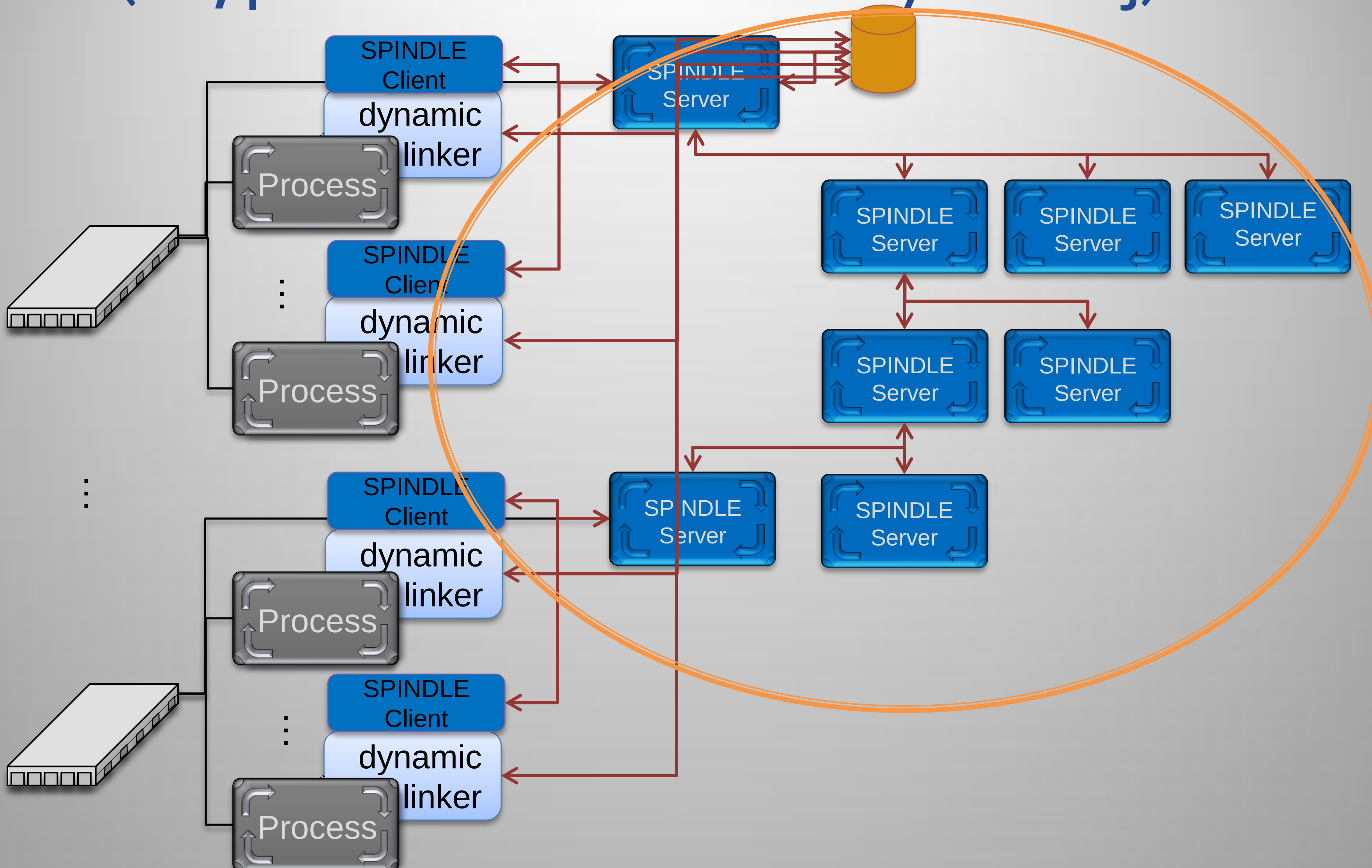
Pynamic running on LLNL Sierra Cluster
1944 nodes, 12 tasks/node,
NFS and Lustre file system

File Access is uncoordinated!

- Loading is nearly unchanged since 1964 (MULTICS)
- ld-linux.so uses serial POSIX file operations that are not coordinated among process.



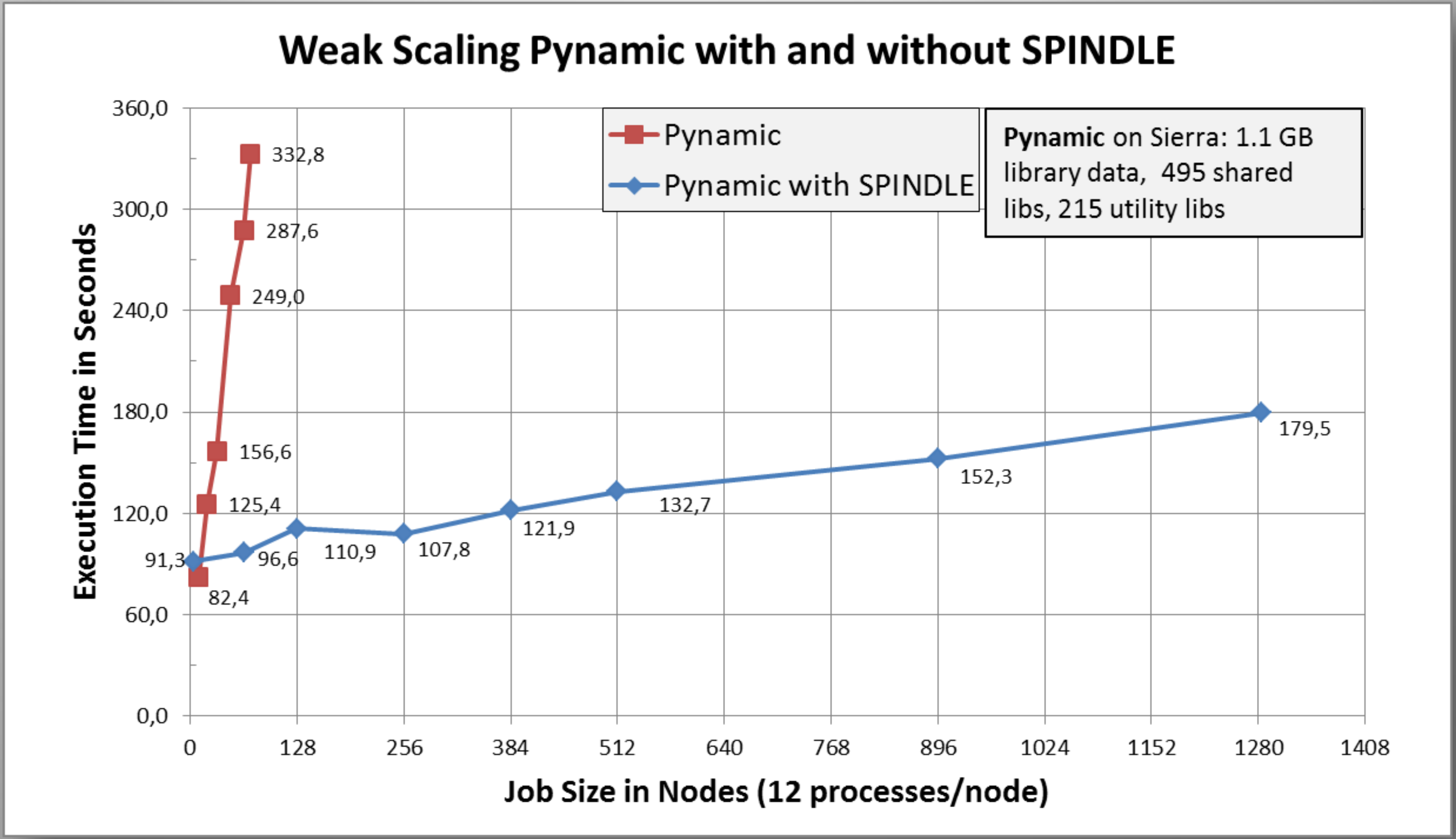
SPINDLE server components will be supported as a LWJ (i.e., performance booster subsystem lwj)



Concluding remarks

- Interim report on the first two phases in our effort to provide a rich and scalable run-time system for FLUX
- Phase I: conceptualized our run-time system around the notion of LWJ
- Phase II: made good process with strategic prototyping
- To prove rich FLUX run-time can solve many next-generation computing challenges by leveraging other technologies through easy integration

Back-up: SPINDLE's Performance



Back-up: Constant Overhead of SPINDLE's Data Distribution