LLNL-PRES-405584

Lawrence Livermore National Laboratory

LLNL Tool Components: LaunchMON, P^{N}MPI, GraphLib

CScADS Workshop, July 2008

Martin Schulz

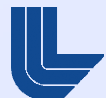
Larger Team: Bronis de Supinski, Dong Ahn, Greg Lee

Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94551

This work performed under the auspices of the U.S. Department of Energy by
Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

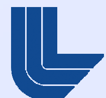
LLNL Tool Components

- LaunchMON: Portable and Scalable Daemon Control
  - Launch tool daemons together with any application
  - Support for intermediate communication daemons
- P^NMPI: Dynamic MPI Tool Assembly
  - Dynamically assemble PMPI tool stacks
  - Separate and reuse common tool functionality
  - Collection of existing tool modules
- GraphLib: Graph Representation and Analysis Library
  - Define and store arbitrary (un)directed graphs
  - Basic graph manipulation and analysis routine



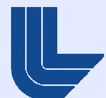
Covered Topics

- For each tool component …
  - Functionality
  - Usage scenarios
  - APIs / Integration
  - Status, Next Steps, Open Questions
- Questions:
  - Is this there general interest in this functionality?
  - What should be added?
  - What is too much/not useful/otherwise covered?
  - How can this be integrated into other tools?
  - How can we integrate other tool components?

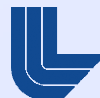


LaunchMON

- Targeted problem:
  - Many tools work with daemons on application nodes
  - Requires system & RM knowledge
- LaunchMON = portable daemon launch and control
  - Identify application tasks and nodes
  - Launch, connect, and initialize daemons
  - Support for hierarchical daemon infrastructures
- Implementation
  - Builds on top of Totalview / MPIR interface
  - Porting requires adjustments for resource managers

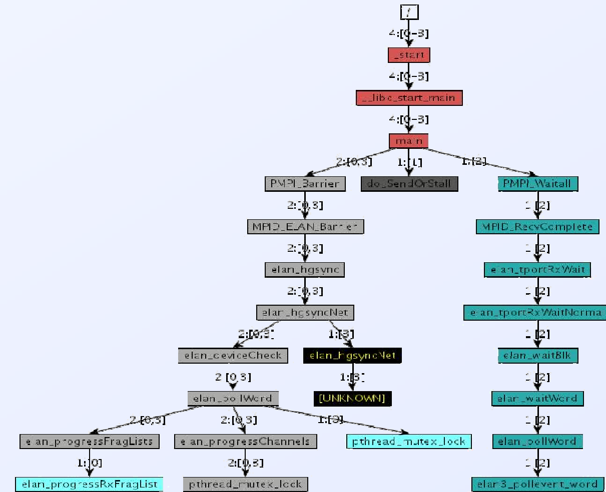


LaunchMON Architecture



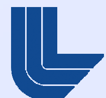
Usage Examples

- STAT=Stack Trace Analysis Tools
  - Gather and merge stack traces
  - Attaches to all tasks
  - LaunchMON controls STAT



- Jobsnap
  - Collect job environments
  - Gather through MRNet
  - Quick prototype possible with LaunchMON

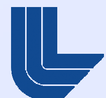
- Open|SpeedShop
  - Prototype to replace direct MPIR usage
  - Reduced overhead due to limited RM binary parsing



APIs & Worksflow

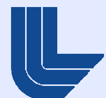
- Frontend API
  - Create Sessions
  - Launch Daemons
  - Get participating tasks
- Backend API
  - Initialize & Connect
  - Get Identity and Context
  - Basic Communication
- Middleware API (under development)
  - Launch Comm. Daemons
  - Connection & Control up to MW software (like MRNet)

- Workflow
  - FE: create Session
  - FE: create and spawn - OR – attach and spawn
  - BE: Init
  - BE: get rank
  - BE: signal ready
  - FE: continue
  - (BE: work & communicate)
  - (FE: receive user data)
  - BE: done
  - FE: terminate



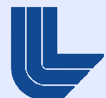
Status / Next Steps / Open Questions

- Status: Version 1.0 available by request
  - Tested on Opteron/SLURM Clusters, BG/L in progress
  - Scalability tests and modeling successful
- Next steps:
  - Porting to more platforms
  - Complete middleware API
  - Automatic topology creation and launch
- Open questions:
  - Session concept: per job session or per tool session?
  - How to allocate additional resources for extra daemons?
  - Application process interface (MPIR functionality)?



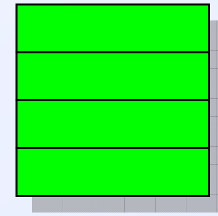
P^N MPI

- Dynamically assemble PMPI tools
  - Chain/Stack any (binary) PMPI tool
  - Configure during application launch
- Usage:
  - Create ".pnmpi-conf" file
    - Lists modules in order of invocation
    - Optional arguments / multiple stacks
  - Run application
    - Detect any function intercepted at least once
    - Build dynamic tool stacks
  - Intercept and redirect all MPI & PMPI calls

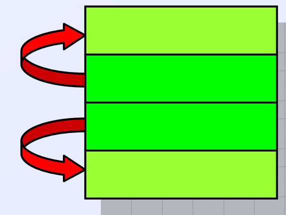


General Usage Scenarios

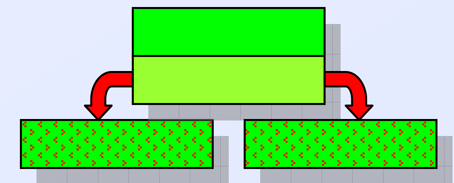
- Concurrent execution of transparent tools
  - Tracing and profiling
  - Message perturbation and MPI Checker



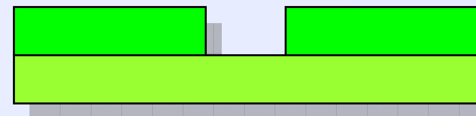
- Tool cooperation
  - Encapsulate common tool operations
  - Access through service interface
  - E.g.: datatype walking, request tracking

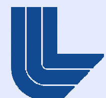


- Tool multiplexing
  - Apply tools to subsets of applications
  - Run concurrent copies of the same tool



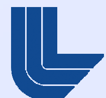
- MPI job virtualization





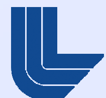
Module Types & Internal APIs

- Tool Modules
  - Standard PMPI API
  - Transform using “patcher”
  - Install in central library path
- Transparent Modules
  - Pure PMPI modules
  - Can reuse binaries
  - Shared libraries required
- P^NMPI Modules
  - PMPI + Internal interface
  - Registration required
  - Offer and use services
- Registration Callback
  - Set module name
  - Define offered services
    - Global variables
    - Functions
    - Type signatures
- Initialization API
  - Query for other modules
  - Query for other services
- Execution API
  - Directly call services
  - Alternative MPI interface to select target stacks



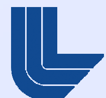
Existing Tool Modules

- Experimented with:
  - mpiP
  - TAU & Vampir tracer
- Status extension
  - Request additional space
- Request tracking
  - Read operation information at Test or Wait operations
  - Request additional storage
- Datatype tracking
  - Query datatype sizes
  - Walk messages
- Communication tracking
  - Abstract any communication
  - Individual callbacks
  - Quick prototyping
- Piggybacking
  - Multiple protocols
  - Request piggyback size
  - Get and Set PB data
- Upcoming modules
  - StackWalker integration (walk through *dlopen()* nec.)
  - ScalaTrace
  - NUMA pinning



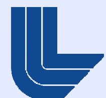
Quick Tool Prototyping Scenarios

- Piggybacking implementation
  - Requires status extension and request tracking
  - Can be built on top of communication module
- Checksum tool
  - Datatype module to create checksum
  - Piggyback checksum for control
- Critical path analysis uses piggyback to associate send and receive operations
- ScalaTrace integration requires stack walker module
- Quick prototypes of new tools
  - Communicator aware tool applications
  - NOPE tracer (using communication module)



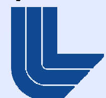
Status / Next Steps / Open Questions

- Status: P^NMPI v1.2 available by request
  - Being thoroughly tested
  - General release 1.3 in the next weeks
- Next steps
  - Complete service architecture for static stacking
  - Platform independent patcher (-> SymtabAPI)
  - Automatic loading with dependency control
- Open questions
  - Service naming for publish/subscribe
  - Tool interoperability and prevention of side effects

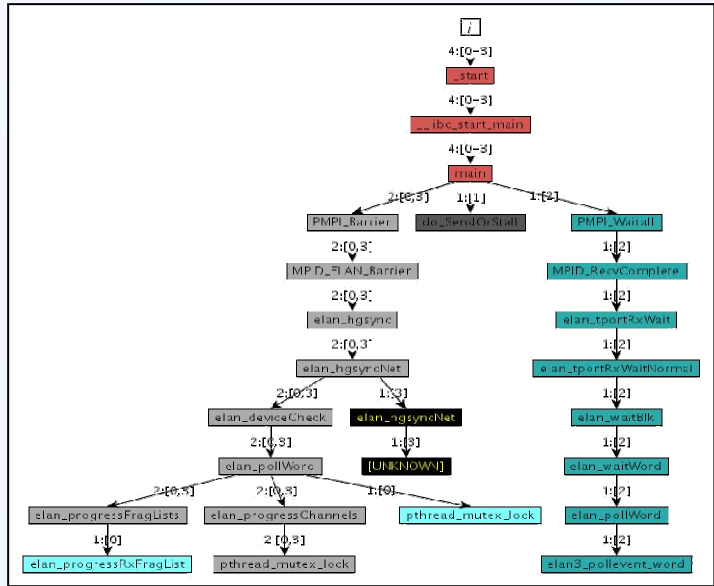


GraphLib

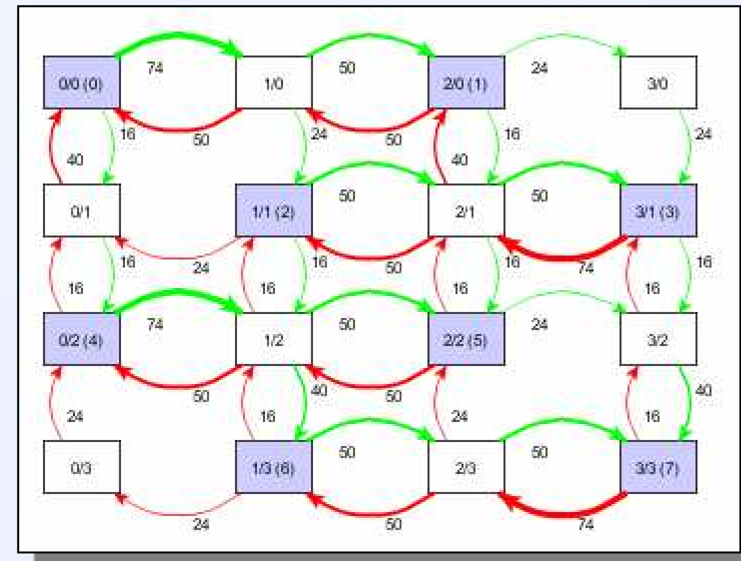
- Reoccurring requirement for tools:
  - Represent, store and display graphs
  - Manipulate and analyze graphs
- GraphLib: C/C++ library for graph representation
  - Add nodes and edge, merge and truncate graphs
  - Node and edge attributes
  - Load, store, and export to GML or DOT format
  - Backtrack analysis, path detection
  - Graph coloring and arbitrary annotations
  - Handles multiple trees or forests
  - Scalable node ID representation (own component?)



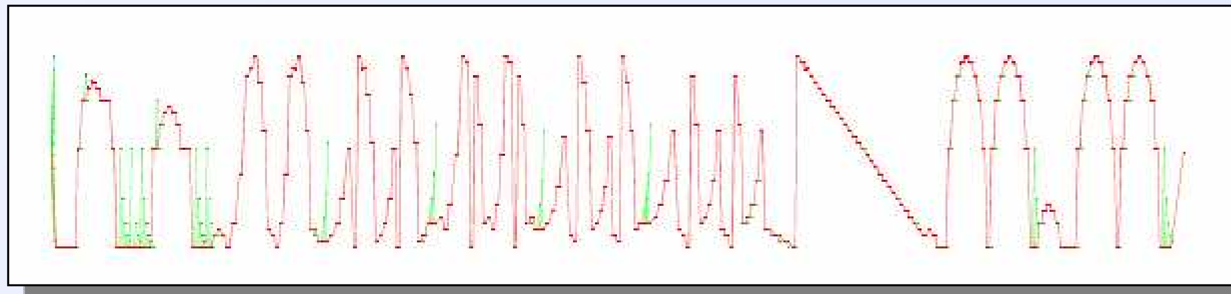
Usage Scenarios



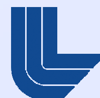
Stack Traces in STAT



BG/L Topologies

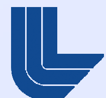


Critical Path Information



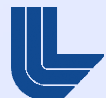
GraphLib API

- Creation & Deletion
  - new(Annotated)Graph
  - delGraph, delAll
- Basic graphs
  - addNode, addNodeNoCheck
  - edgeCount, nodeCount
  - add(Un)directed)Edge
- Attributes
  - setDefNode/EdgeAttr
  - annotationKey/Set/Get
- Storage
  - load/store/exportGraph
  - (de)serializeGraph
- Basic manipulation
  - scaleNodeWidth
  - mergeGraphs
  - deleteTree
  - CollapseHor
- Critical path analysis
  - Backtrack and color through directed graph
- STAT
  - Node IDs as edge labels
  - Ranged merge
  - Edge label based coloring



Status / Next Steps / Open Questions

- Status
  - Available in version 1.0 as part of STAT
  - Useful functionality, but ad-hoc implementation
- ToDo list:
  - More efficient/higher density storage
  - Generalize coloring scheme
  - Generalize and classify analysis algorithms
- Open questions:
  - How to vary node/edge attributes (C++ classes)?
  - Adjust graphs to requested analysis
  - Separate scalable node representation



Summary / Discussion

- Three LLNL Tool Components
  - LaunchMON: Scalable Tool Daemon Control
  - P^NMPI: Flexible MPI Experiments
  - GraphLib: Reusable Graph Representation
- Questions:
  - Is this there general interest in this functionality?
  - What should be added?
  - What is too much/not useful/otherwise covered?
  - How can this be integrated into other tools?
  - How can we integrate other tool components?