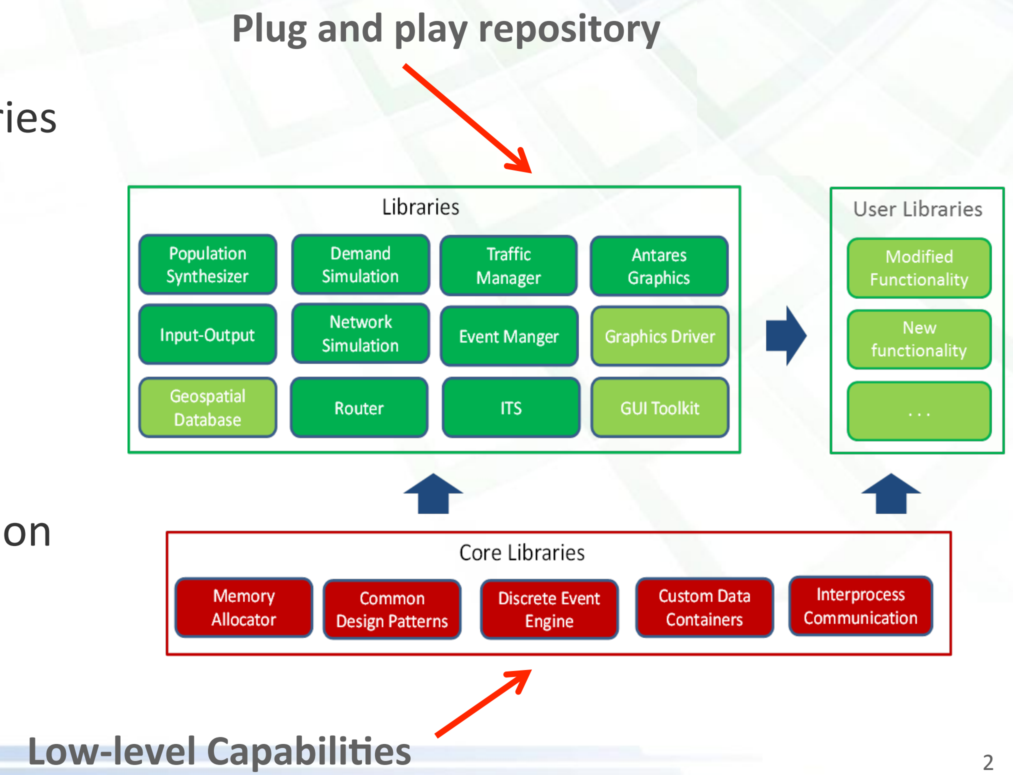
POLARIS: General Purpose Agent-based Modeling Framework Specialized for High-Performance Transportation Simulations

Joshua Auld, Michael Hope, Hubert Ley, Vadim Sokolov

Transportation Research and Analysis Computing Center
Argonne National Laboratory
January 11, 2015

What is POLARIS?

- Middleware for Developing Agent-based Models
  - Data Interchange
  - Visualization
  - Case Study Generation and Analysis
  - Discrete Event Simulation
  - Interprocess Communication
- A Repository of Transportation Libraries
  - Common Algorithms
  - Extended by Researchers
  - Standardized Style and Structure
- Fully Developed Applications
  - Transportation Network Simulation
  - Integrated Activity Based Travel Demand Simulation



Why Develop a Framework for Building Transportation Models?

- Pattern of extremely common objects being re-written, simply to provide slightly different views. Many models differ primarily in level of aggregation.

- Certain areas (namely Intelligent Transportation Systems) have entities which are rapidly changing and cannot be represented adequately in a black-box model.

- Groups who want to add features or change behavior tend to write new models rather than salvage material from existing models due to the difficulty of re-adpating them, this incurs a re-invention of the wheel.

- Many performance and modularity-enhancing capabilities in the realm of advanced computing are being under-utilized.

The POLARIS Core Library Implements the Building Blocks for High-Performance Simulations

- Memory Management Library
  - Optimized for the type of allocation needed in transportation modeling applications
- Discrete Event Engine
  - Enables writing from an agent-based perspective
- Interprocess Engine (de-emphasized in current version of POLARIS)
  - Enables parallel cluster execution
- High Performance Data Structures
  - Non-standard structures relevant for use in transportation modeling applications

Memory Manager Optimized for use in Discrete Event Simulations

- Motivation
  - Transportation code can be performance critical
  - Simulation code tends to follow distinct memory allocation / deallocation patterns
  - Discrete Event Engine execution requires a global tracking of allocated objects
  - Solution: Create a memory allocator which is designed for simulation systems

- Technique
  - Hierarchical memory layout: divide by type, then block, then object
  - Memory blocks owned by threads for allocation (control may be traded between threads on deallocation)
  - Effective structure for each type is an unrolled linked list of memory pools
  - Elements within memory pool (blocks) are cache line aligned
  - Block size is optimized by user input, object count, and object size
  - Replace default memory allocator with tc_malloc (for highly threaded applications)
  - Allocated memory is prepended with variables necessary for performing execution

Assembling an Object Using Polaris Memory Manager

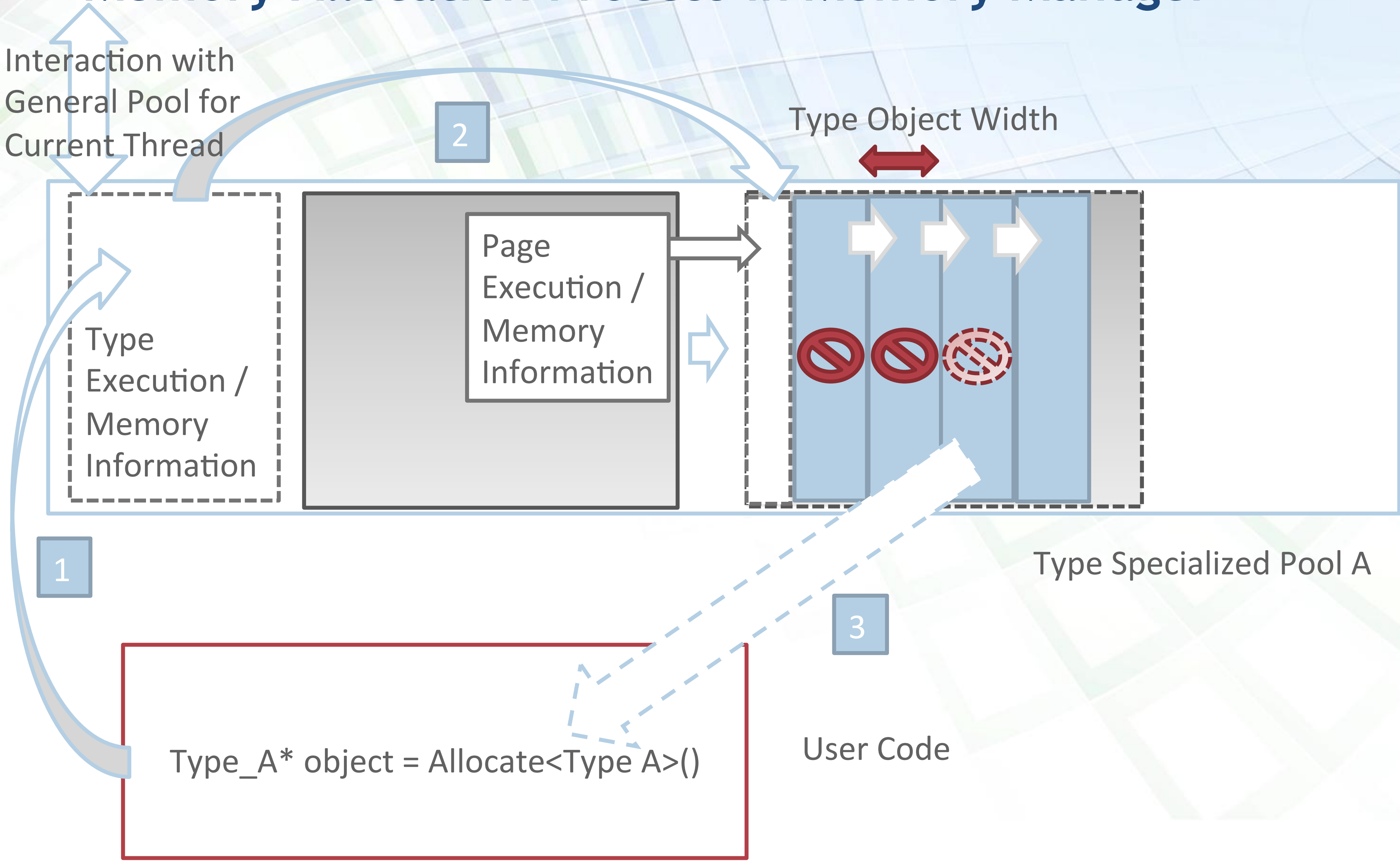
```
prototype struct Agent
{
    accessor(data,NONE,NONE)

    template<typename T> void Initialize(T data){ ... }
};

implementation struct Agent_Impl : public Polaris_Component<MasterType,INHERIT(Agent_Impl),Execution_Object>
{
    m_data(float, data, NONE,NONE);

    template<typename T> void Initialize(T data){ ... }
};

struct MasterType
{
    typedef Agent_Impl<MasterType> agent_type;
};

int main()
{
    typedef Agent<MasterType::agent_type> agent_itf;
    agent_itf* agent = (agent_itf*)Allocate<MasterType::agent_type>();

    agent->Initialize<float>(42.0);
}
```

Object interface

Object implementation, defines the object as an execution object

Allocation of an object returns a pointer to the object which can be cast to its prototype

Memory Allocation Process in Memory Manager



Custom Memory Manager offers Substantial Performance and Usability Benefits

- Performance Benefits
  - Allocations and deallocations are fully parallelized
  - Unrolled linked list structure provides an effective balance of stride optimization vs memory alteration
  - Can make use of user input (such as the expected number of objects) to further optimize the structure

- User Benefits
  - Multiple deallocation options: agent-directed, lazy deallocation, or immediate
  - Global tracking of memory allows global tracking of objects by ID
  - Provides additional protection when deallocating agents which may be currently executing on another thread
  - Tracking of memory usage without an external tool

Discrete Event Engine Designed for Transportation Simulations

- What does the discrete event engine do?
  - Allows the developer to create an agent of any kind (Traveler, Traffic manager,…)
  - Describe when it wants to act (what time under what conditions)
  - Define what agents do when when they do act
  - Define one or more actions which the agent can choose among
  - Then allow the agent to perform autonomously

- Why do we need it?
  - Transportation code can be performance critical
  - Agent-based design applies particularly well to travelers, signals, traffic management centers, and other "intelligent" objects in a simulation
  - A discrete event engine supports this paradigm.

- Solution: develop a Discrete Event Scheduling engine as the heart of the execution model

Key Features of the Discrete Event Engine

- The user does not explicitly control when time advances
- Rather than the user having to fire events at a given time, the user requests for an event to happen some time in the future and then defines the conditions under which it fires
- Making all event requests available at a global level before they occur allows an incredible opportunity to optimize their execution behind the scenes

Discrete Event Engine Design Benefits

- Structural Benefits
  - Eliminates the traditional execution loop over time and objects
  - Provides universal "time" in the simulation
  - Eases the task of coordinating the actions of agents
  - Allows user to write from the agents' point of view
  - Provides a space "under the hood" for advanced debugging

- Performance Benefits
  - Enables automated multi-threading which is highly scalable
  - System can self-optimize to balance workload among threads
  - Memory management allows fast creation of new agents
  - Polling all agents can be done quickly using optimized data containers (optimizing contiguous memory usage)

Scheduling an Event for an Agent

- Inherit from Polaris_Component

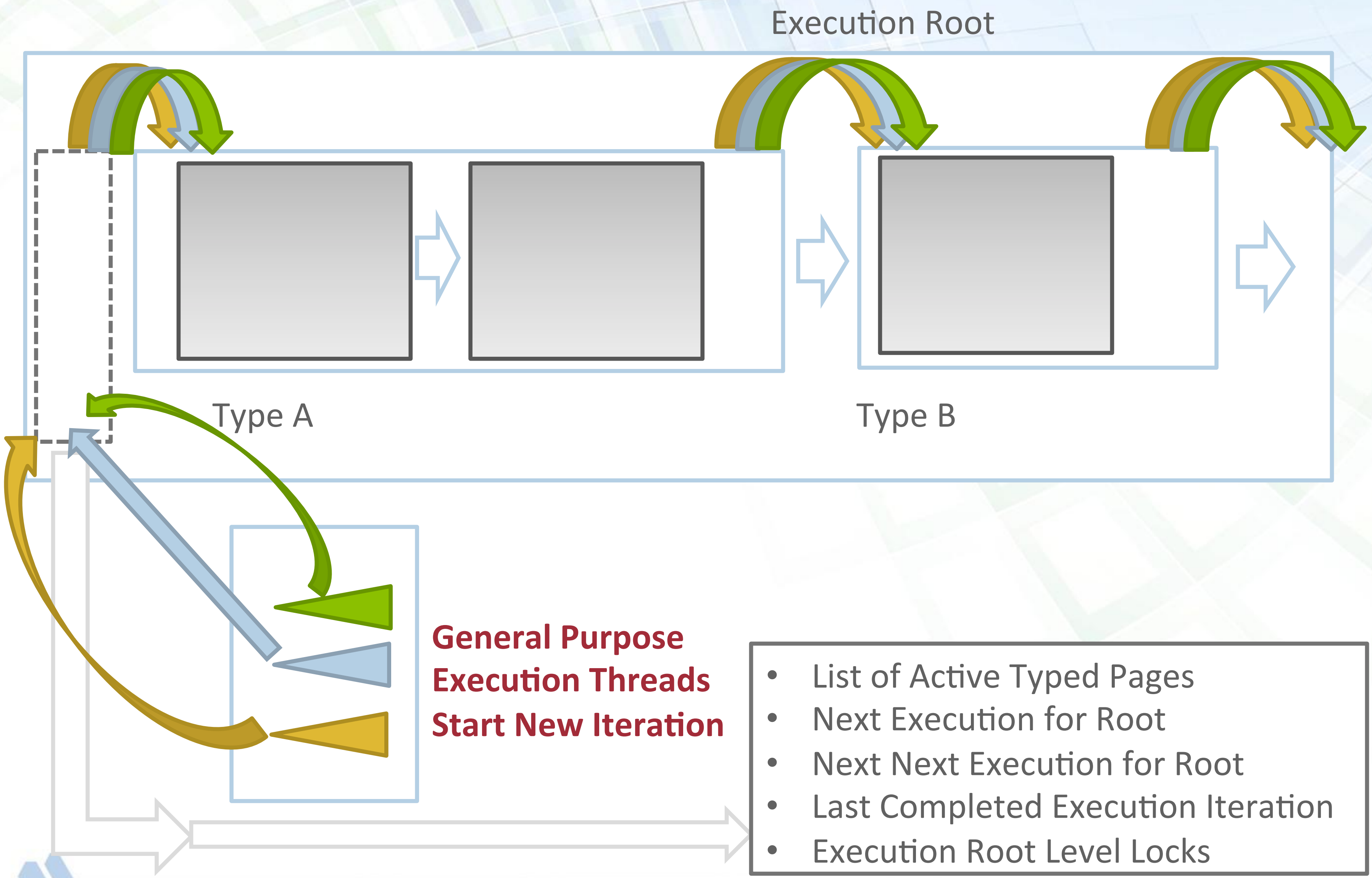
```
// POLARIS implementation of a moving agent
implementation struct Agent : public Polaris_Component<MasterType,INHERIT(Agent),Execution_Object>
{
    // Agent initializer - creates and draws agent at starting position, loads the starting event
    void Initialize(int start)
    {
        Load_Event<Agent>(&Do_Move, start, 0);
    }

    // Movement event, and associated event function
    static void Do_Move(Agent* _this,Event_Response& response)
    {
        // Process move
        bool done = _this->Move();
        if (done) Swap_Event((Event)&Do_Stop);

        // Set next iteration
        response.next._iteration = iteration() + 1;
        response.next._sub_iteration = 0;
    }
    void Move() { ... }

    // Stop event and associated stop function, this draws the stopped vehicle
    static void Do_Stop(Agent* _this,Event_Response& response) { ... }
    void Stop() { ... }
};
```

- Schedule the first event into register in the Initialize function

- Process the event
- Set next execution time

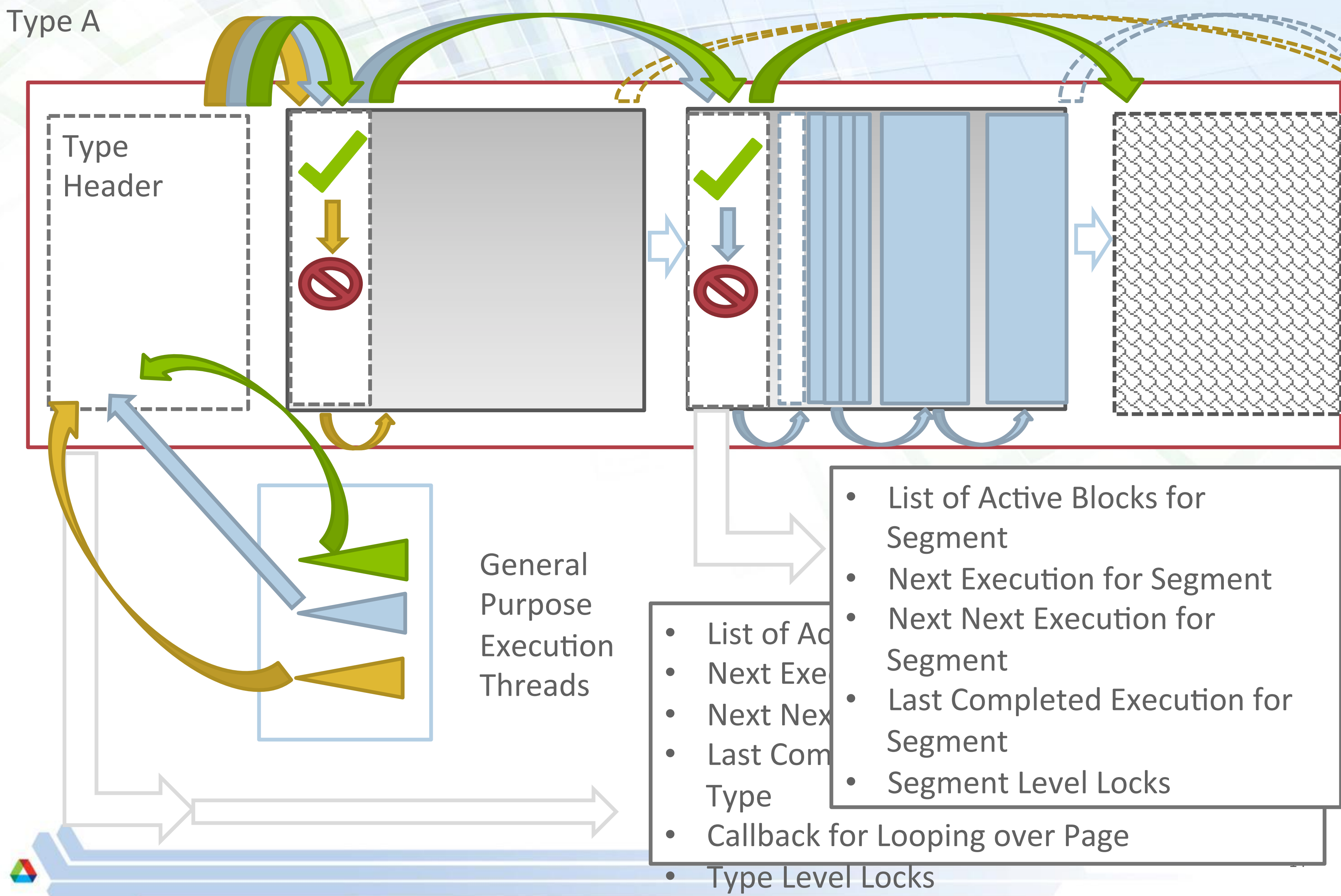
- Swap event when movement done
- On next iteration Do_Stop will be executed instead

Events are processed first by type

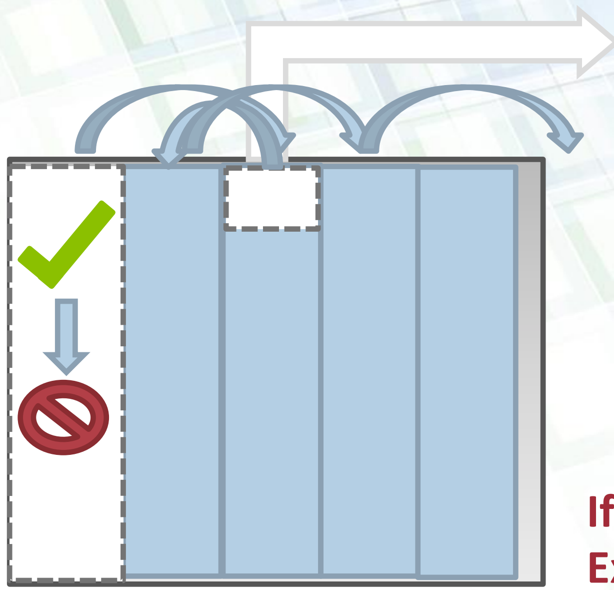


- List of Active Typed Pages
- Next Execution for Root
- Next Next Execution for Root
- Last Completed Execution Iteration
- Execution Root Level Locks

Threads then process types by segment and block



Finally, process objects within the execution block



- Next Iteration for Object
- Current Iteration for Object
- Register for Execution Callback
- Intrusive List Info (for Event_Object)

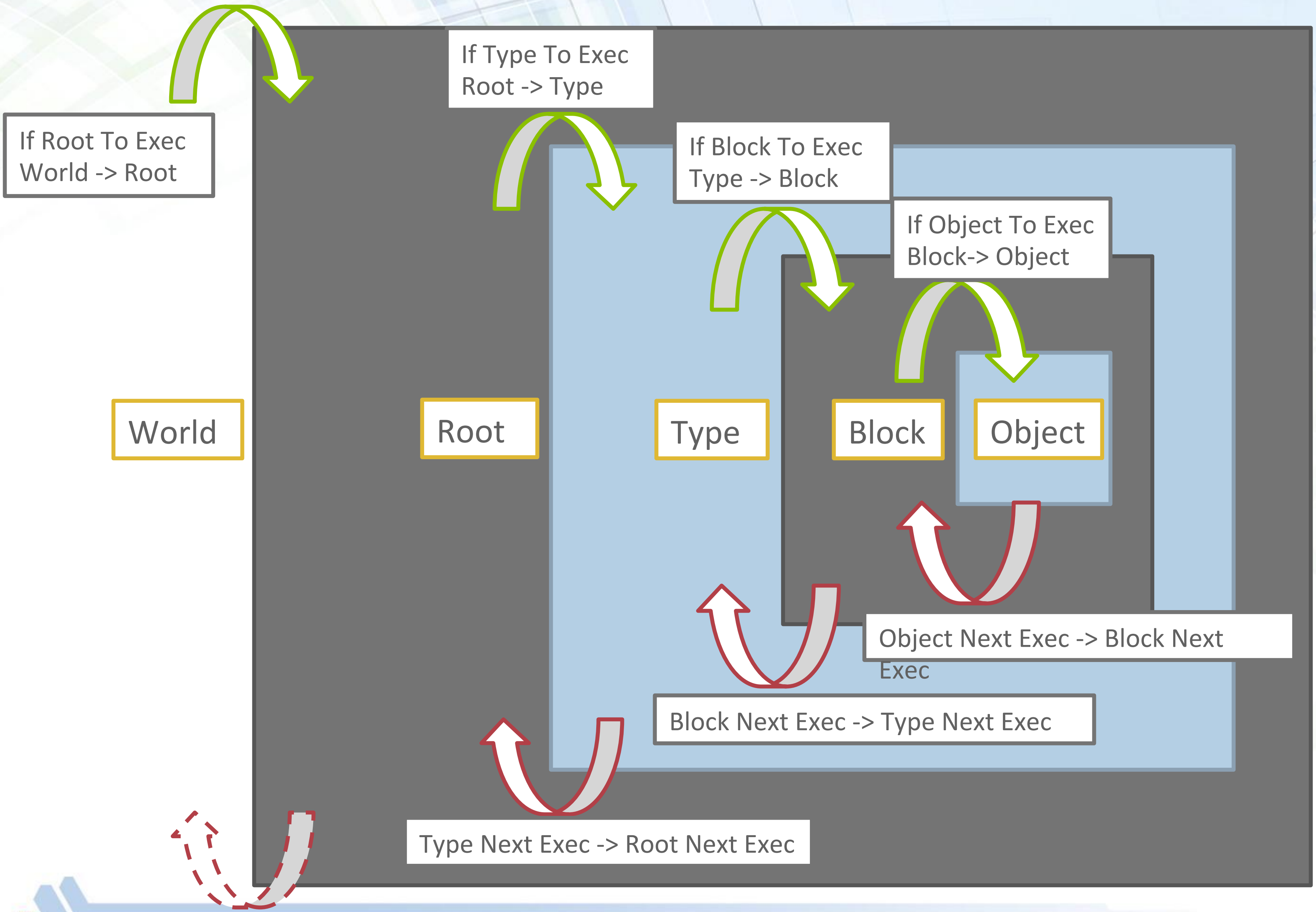
If Object is Scheduled - User Written Execution Code Fires

Engine Schedules Next Iteration (Next Iteration == This Iteration means “not ready try again later”)

Exit block if current object not scheduled for current iteration

- Next Execution for Block
- Next Next Execution for Block
- Last Completed Execution Step for Block
- Intrusive List of Active Objects (Event_Object types only)
- Block Level Lock

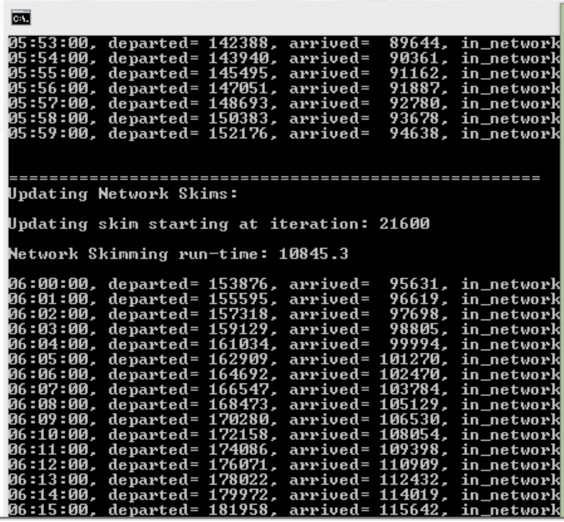
Scheduling Flow for Discrete Event Scheduling Engine

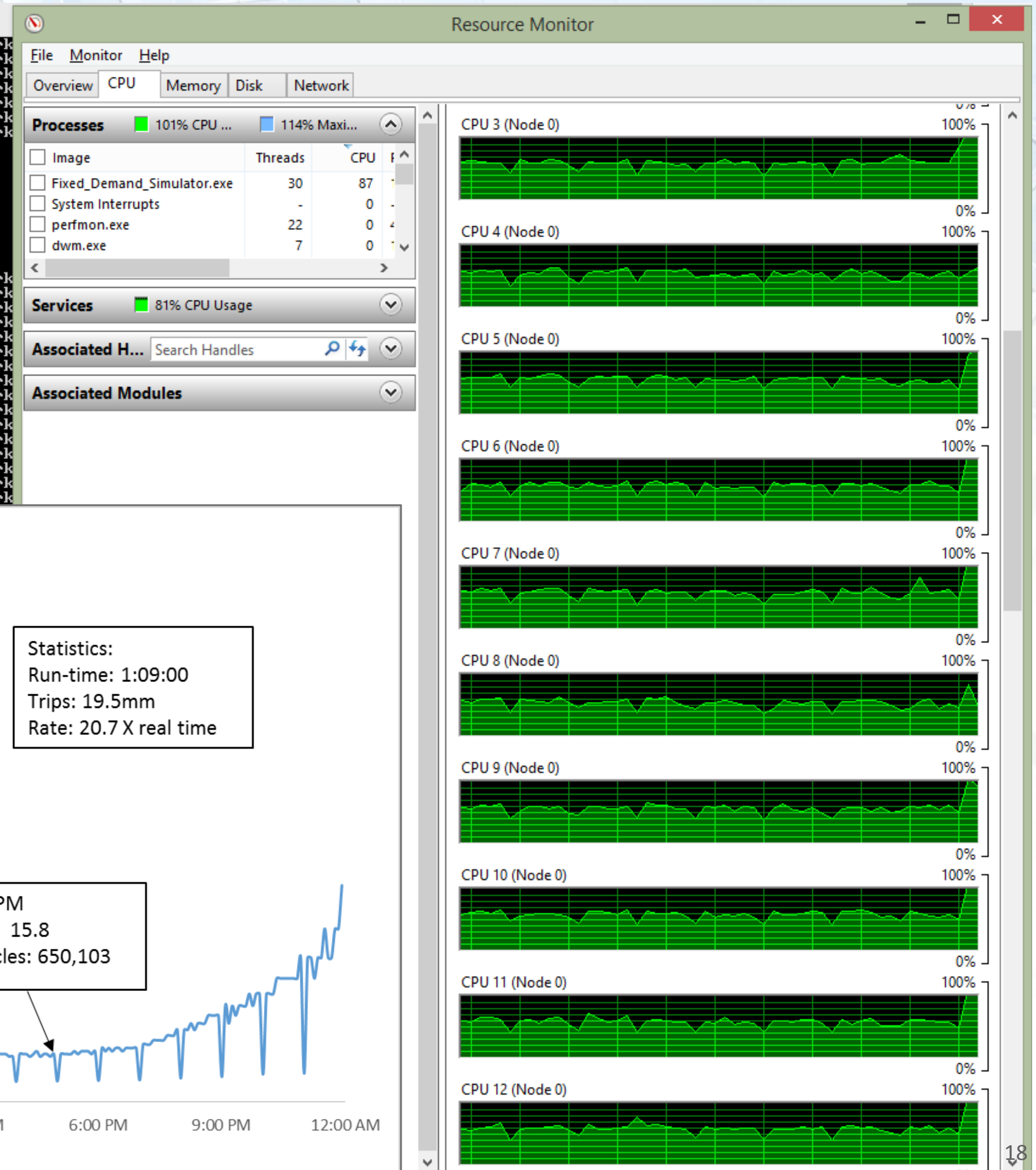


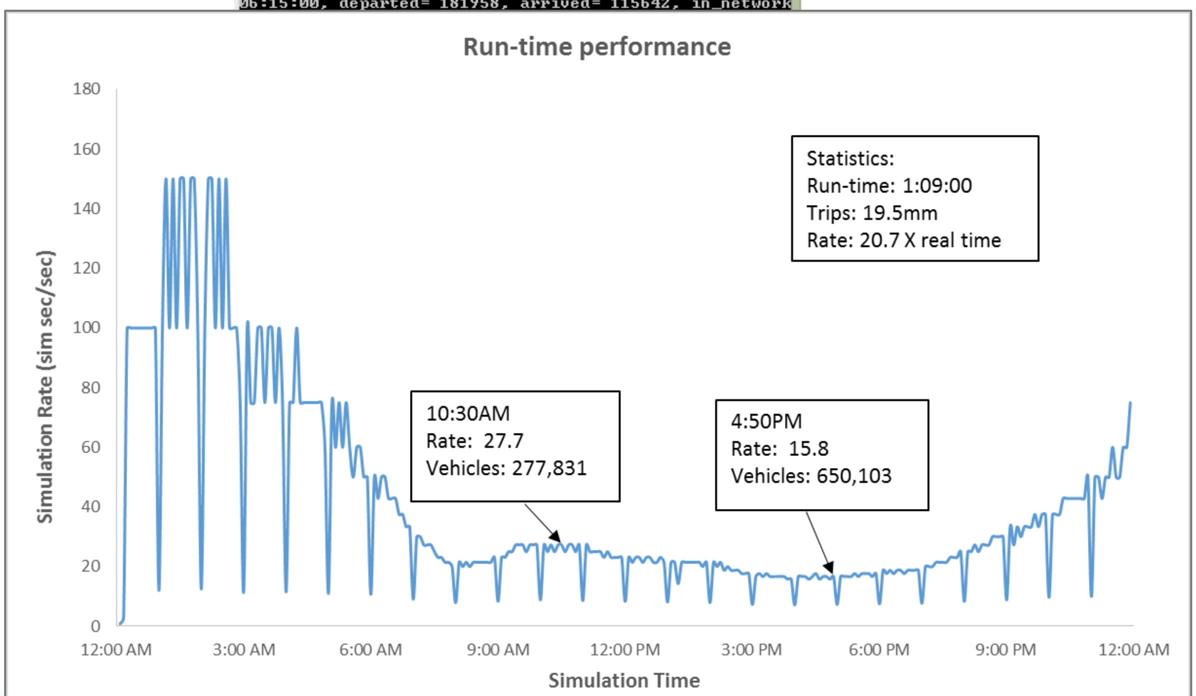
Prototype Interprocess Engine will Eventually Allow System to Extend to Multi-Processor Environments

- Allow Developers to Utilize Cluster Computers for Individual Cases
- Exchange Messages with other POLARIS Applications
- Automatically Aggregate, Coordinate, Parallelize, and Optimize all Exchanges
- Fully Integrate with the Discrete Event Engine
- Have Capability to Send Messages Addressed to Specific Objects
- Define Custom Parsing Functions For When a Message is Received

Key Performance Characteristics







Conclusion

- POLARIS Core libraries provide:
  - High performance memory management
  - Discrete event simulation optimized for multi-threaded execution of agents with sparse scheduling
  - Potential extension to multi-processor environment using inter-process communication library

- Takes advantage of modern processor architecture:
  - Automated threading
  - Transparent to model developer – but still controllable if needed
  - Highly optimized load-balancing
  - Memory manager with nested layout for efficient object execution

- Extensible framework can be used for variety of transportation simulation needs

Thank You!

For more information go to:

https://github.com/anl-tracc/polaris