Simulating distributed applications with SIMGRID

Henri Casanova[1,2]

[1]Associate Professor
Department of Information and Computer Science
University of Hawai'i at Manoa, U.S.A.
[2]Visiting Associate Professor
National Institute of Informatics, Japan

NII Seminar Series, October 2013

Acknowledgments

- Joint work with a LOT of people:

  http://simgrid.gforge.inria.fr

Introduction

- As seen in the previous seminar, many results in parallel/distributed computing research are obtained in simulation
- Simulation has been used for decades in various areas of computer science
  - Network protocol design, microprocessor design
- By comparison, current practice in parallel/distributed computing is in its infancy
- Part of the problem is the lack of a standard tool, likely due to research being partitioned in different sub-areas
  - cluster computing, grid computing, volunteer computing, peer-to-peer computing, cloud computing, ...
- Part of the problem is the lack of open methodology...

Lack of open methodology

- The key to open science is that results can be reproduced
- Unfortunately, open science is not yet possible in parallel/distributed computing simulations
- Example: "Towards Yet Another Peer-to-Peer Simulator", Naicken et al., *Proc. of HET-NETs*, 2006
  - Surveyed 141 papers that use simulation
  - 30% use a custom simulator
  - 50% don't say which simulator is used (!!)
  - Only few authors provide downloadable software artifacts / datasets
- Researchers don't trust simulators developed by others?
- They feel they can develop the best simulator themselves?
- Ironic consequence: published simulation results are often not reproducible

And yet....

- There are parallel/distributed computing simulators that are intended to be used by others!
- These simulators all attempt to do the same 3 things:
  - Simulate CPUs
  - Simulate networks
  - Simulate storage
- The two main concerns are:
  - Simulation accuracy
  - Simulation speed / scalability
    - Ability to simulate large/long-running applications/platforms fast and without too much RAM
- Simulators make choices to trade off one for the other
- Let's look at typical such choices...

Simulating computation (I)

- There are two main approaches for simulating computations: microscopic or macroscopic
- Microscopic approach: cycle-accurate simulation
  - Simulate micro-architecture components at the clock cycle resolution based on instructions from real (compiled) code or for synthetic instruction mixes
  - Typically used by computer architecture researchers
  - ☺ Arguably very accurate (bus contention, cache effects, instruction-level parallelism, multi-core, GPUs, ...)
  - ☹ Very slow (ratio of simulation/simulated time > 100)
  - ☹ Arbitrary unscalable as the volume of computation increases, which is a problem for simulating long-running applications (e.g., grid computing, cluster computing)

Simulating computation (II)

- Macroscopic approach: (scaled) delays
  - Defined for each compute resource a *computation speed*
  - Define a computation as a *computation volume*
  - The simulated time is computed as the *ratio* of the two, plus an optional random component
  - ☺ Reasonably accurate for compute-bound applications
    - Compute times can also be sampled from real application or benchmark executions on a reference computer architecture, and then scaled
  - ☺ Scalable: simulation of a computation in $O(1)$ space/time
  - ☹ Can be wildly inaccurate for memory-bound applications

Simulating communication (see previous seminar)

- Each link: *latency* and *bandwidth*
- Protocol-accurate packet-level simulation
  - Used by network protocol researchers
  - ☺ Accurate
  - ☹ slow and not scalable
- Non-protocol-accurate store-and-forward of packets
  - ☹ An attempt to be more scalable than the above, but can be made arbitrarily inaccurate (packet size?)
- Ad-hoc fluid models
  - ☺ Scalable
  - ☹ Not protocol-realistic
- TCP fluid models
  - ☺ Scalable and accurate within limits (see previous seminar)

Simulating storage

- Microscopic approach: detailed discrete-even simulation (e.g., DiskSim)
  - ☺ Accurate
  - ☹ Just like cycle-accurate, and packet-level: unscalable
    - Arbitrarily high simulated/simulation ratio for large data
- Most simulators provide very little
  - Notion of storage capacity and stored data
  - Each transfer has a latency and bandwidth, with optional random components
  - ☹ Fails to capture caching effects, file system effects, ...

Specifying the simulated application (I)

- There are three main approaches: Finite automata, Event traces, Concurrent Sequential Processes (CSP)

- Finite automata:
  - Each simulated process is described as an automaton or a Markov chain
  - Each state is an action that lasts for some number of (simulated) seconds
    - Example: compute for 10s, then with probability 0.5 communicate for 1s, then with probability 1.0 do nothing for 30s, ...
  - ☺ Very scalable since each process is described with only a few bytes and fast algorithms can do state transitions
  - ☹ Limited expressive power, no/little application logic

Specifying the simulated application (II)

- Event traces:
  - Each simulated process is described as a sequence of compute, communicate, store events
    - At $t = 0$ compute for 10s, at time $t = 10$ send a message for 2s, at $t = 12$ receive a message for 20s, ...
  - Events obtained from real-world executions and "replayed", while scaling delays
  - ☺ Fast since event replay is algorithmically simple
  - ☹ Not always scalable (traces can be large, obtaining large traces on dedicated platforms can be hard)
  - ☹ Difficult to extrapolate traces to simulate more/fewer processes

Specifying the simulated application (III)

- Concurrent Sequential Processes (CSP):
  - Application implemented as fragments of arbitrary code that call simulation API functions
    - `sim_compute`, `sim_send`, `sim_recv`, ...
  - ☺ Maximum expressive power, arguably
  - ☹ Not scalable
    - Number of threads is limited (e.g., "only" a few thousands Java threads)
    - Context switching and synchronization overhead can be high
    - The user can implement expensive logic/computation, which may be useful but also unscalable

State-of-the-art simulators

Simulator	Community of Origin					Application			Network						CPU		Disk		
	high perf. comp.	grid comp.	cloud comp.	volunteer comp.	peer-to-peer comp.	execution trace	abstract spec.	programmatic spec.	latency	bandwidth	store-and-forward	ad-hoc flow	TCP fluid model	packet-level	scaled real delay	scaled delay	capacity	seek + transfer	block-level
PSINS	✓					✓			✓	✓		✓			✓	✓			
LogGOPSim	✓					✓			✓	✓						✓			
BigSim	✓					✓			✓	✓				✓	✓	✓			
MPI-SIM	✓							✓	✓	✓					✓	✓			
OptorSim		✓						✓	✓	✓		✓				✓	✓		
GridSim		✓						✓	✓	✓	✓	✓				✓	✓	✓	
GroudSim		✓	✓					✓	✓	✓		✓				✓			
CloudSim			✓					✓	✓	✓	✓					✓	✓	✓	
iCanCloud			✓					✓	✓	✓				✓		✓	✓	✓	✓
SimBA				✓			✓	✓	✓							✓			
EmBOINC				✓			✓	✓	✓							✓			
SimBOINC				✓				✓	✓	✓			✓			✓			
PeerSim					✓		✓		✓										
OverSim					✓			✓	✓	✓				✓					
SIMGRID		✓				✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	

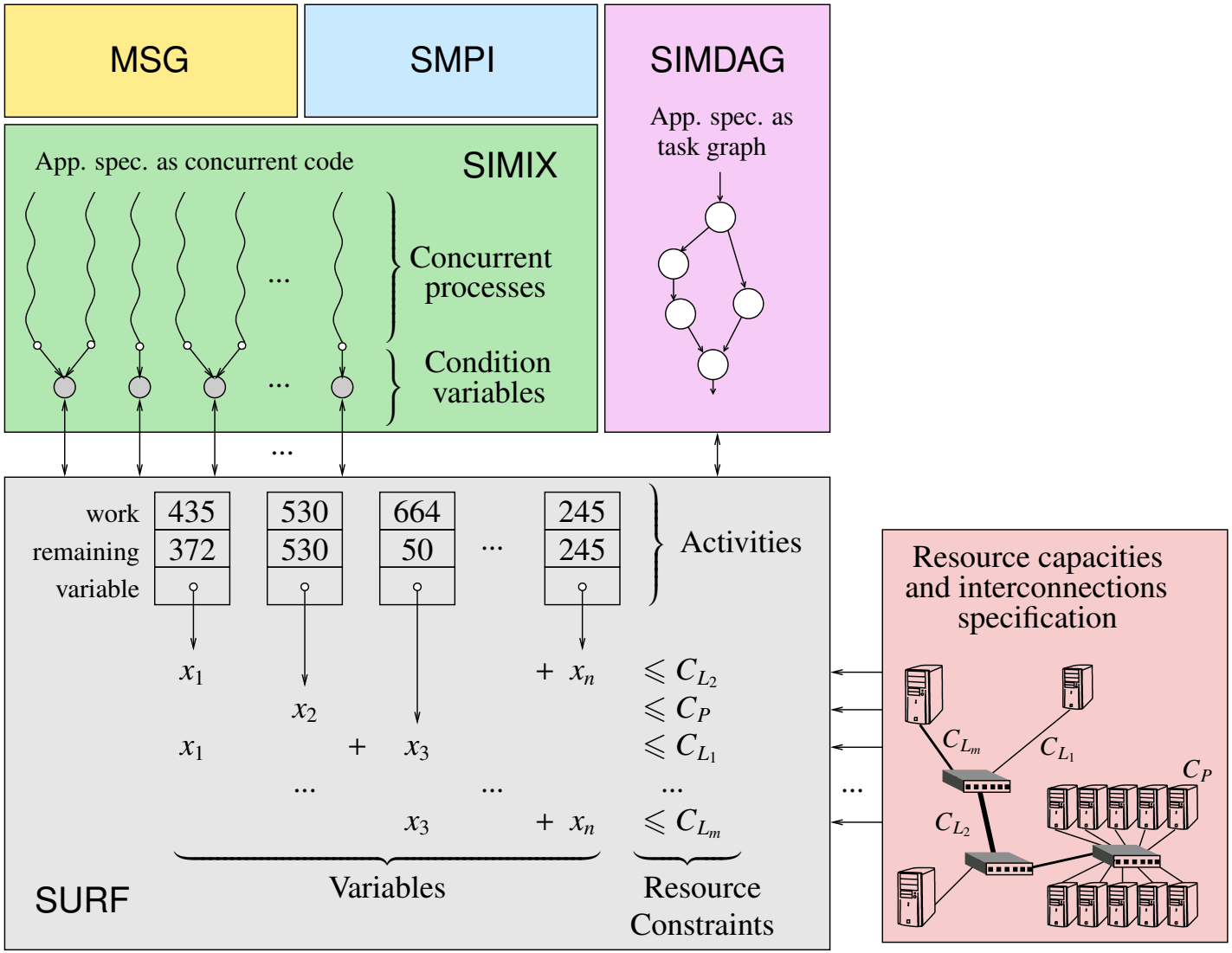
SIMGRID: a counter-intuitive design approach

- Accepted wisdom: to be both accurate and fast/scalable, a simulator must be highly *specialized* to a target domain
- Typical rationale: to achieve scalability one must "cut corners" and reduce accuracy in ways that are hopefully ok for the target domain
  - Example: when simulating a p2p application, no need to simulate network contention, or compute times
  - Example: when simulating a cluster computing application, no need to simulate external load on the system
- Instead, with SIMGRID we target multiple domains:
  - Grid, cloud, cluster/HPC, volunteer, peer-to-peer
- We claim/demonstrate that we can be accurate/scalable across domains!

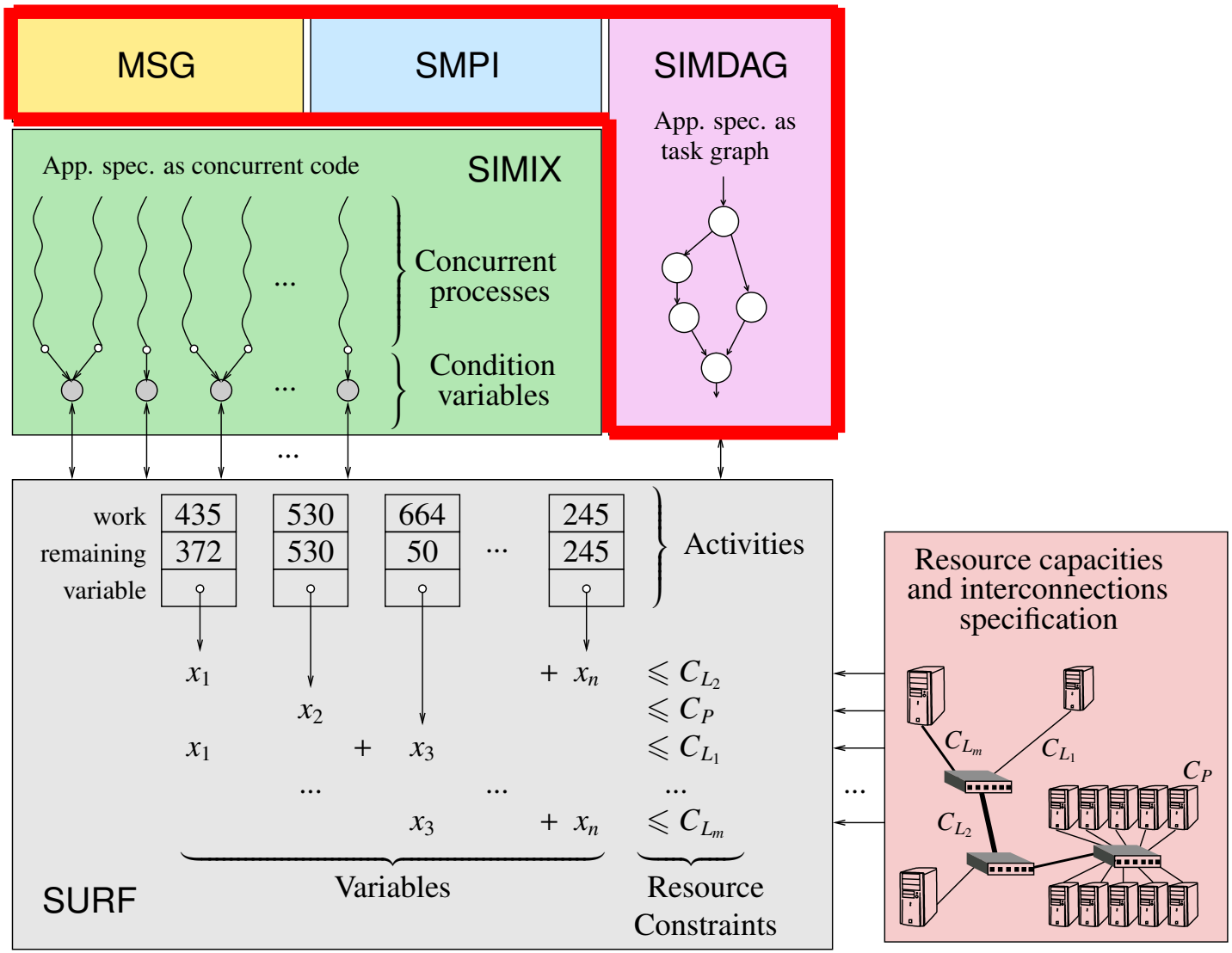
SimGrid: history

- A 14-year old open source project
- First release: 1999
  - A tool to prototype/evaluate scheduling heuristics, with naïve resource models
  - Essentially: a way to not have to draw Gantt charts by hand
- Second release: 2000
  - Addition of TCP fluid network models
  - Addition of an API to describe simulated app as CSPs
- Third release: 2005
  - Stability, documentation, packaging, portability, ...
- Release v.3.3: 2009
  - Complete rewrite of the simulation core for better scalability
  - Possible to describe transient resource behavior via traces
  - Addition of an "Operating System"-like layer
  - Two new APIs

Software stack



APIs


MSG
SMPI
SIMDAG
App. spec. as task graph
App. spec. as concurrent code
SIMIX
Concurrent processes
Condition variables
work
remaining
variable
435
372
530
530
664
50
245
245
Activities
x_1
x_2
x_1 + x_3
x_3 + x_n
$\leqslant C_{L_2}$
$\leqslant C_P$
$\leqslant C_{L_1}$
$\leqslant C_{L_m}$
Variables
Resource Constraints
SURF
Resource capacities and interconnections specification
C_{L_m}
C_{L_1}
C_P
C_{L_2}

The SIMDAG API

- Application is described as a task graph
  - SD_TASK_CREATE(), SD_TASK_DEPENDENCY_ADD(), ...
  - SD_TASK_SCHEDULE() (on a host)
  - All types of API functions to get/set task properties/parameters
- One call to SD_SIMULATE() launches the simulation:
  - While there are ready tasks, run them
    - Computation + communication
  - Resolve dependencies
  - Repeat
- A very simple API designed for users who don't need the full power of the CSP abstraction
  - Looks a lot like SIMGRID v1.0

The SMPI API

- Designed to simulate Message Passing Interface (MPI) applications
  - Standard way to implement communication in parallel applications
- The (almost unmodified) application is compiled so that MPI processes run as threads
- MPI calls are intercepted and passed to SimGrid's simulation core
- "Tricks" are used to allow simulation on a single computer
  - CPU burst durations are samples a few times and "replayed" for free
  - Arrays are shared among threads (wrong data-dependent application behavior but small memory footprint)

The MSG API

- This is the most commonly used API: basic CSP abstraction
- It has bindings in C, Java, Lua, Ruby
- Let's go through a full (but simple) *master-worker* example
  - The master process has tasks to send to workers
  - Each worker "processes" the tasks until it receive a termination signal
- Let's look at:
  - Master code
  - Worker code
  - Main function
  - XML platform description file
  - XML application deployment description file

Master

int master(int argc, char *argv[ ]) {

```
  int number_of_tasks = atoi(argv[1]); double task_comp_size = atof(argv[2]);
  double task_comm_size = atof(argv[3]); int workers_count = atoi(argv[4]);
  char mailbox[80]; int i;
  char buff[64];
  msg_task_t task;

  /* Dispatching tasks (dumb round-robin algorithm) */
  for (i = 0; i < number_of_tasks; i++) {
    sprintf(buff, "Task_%d", i);
    task = MSG_task_create(buff, task_comp_size, task_comm_size, NULL);
    sprintf(mailbox,"worker-%d",i % workers_count);
    print("Sending task %s to mailbox %s\n",buff,mailbox);
    MSG_task_send(task, mailbox);
  }
  /* Send finalization message to workers */
  for (i = 0; i < workers_count; i++) {
    sprintf(mailbox,"worker-%ld",i % workers_count);
    MSG_task_send(MSG_task_create("finalize", 0, 0, 0), mailbox);
  }
  return 0;
}
```

Worker

int worker(int argc, char *argv[ ]) {

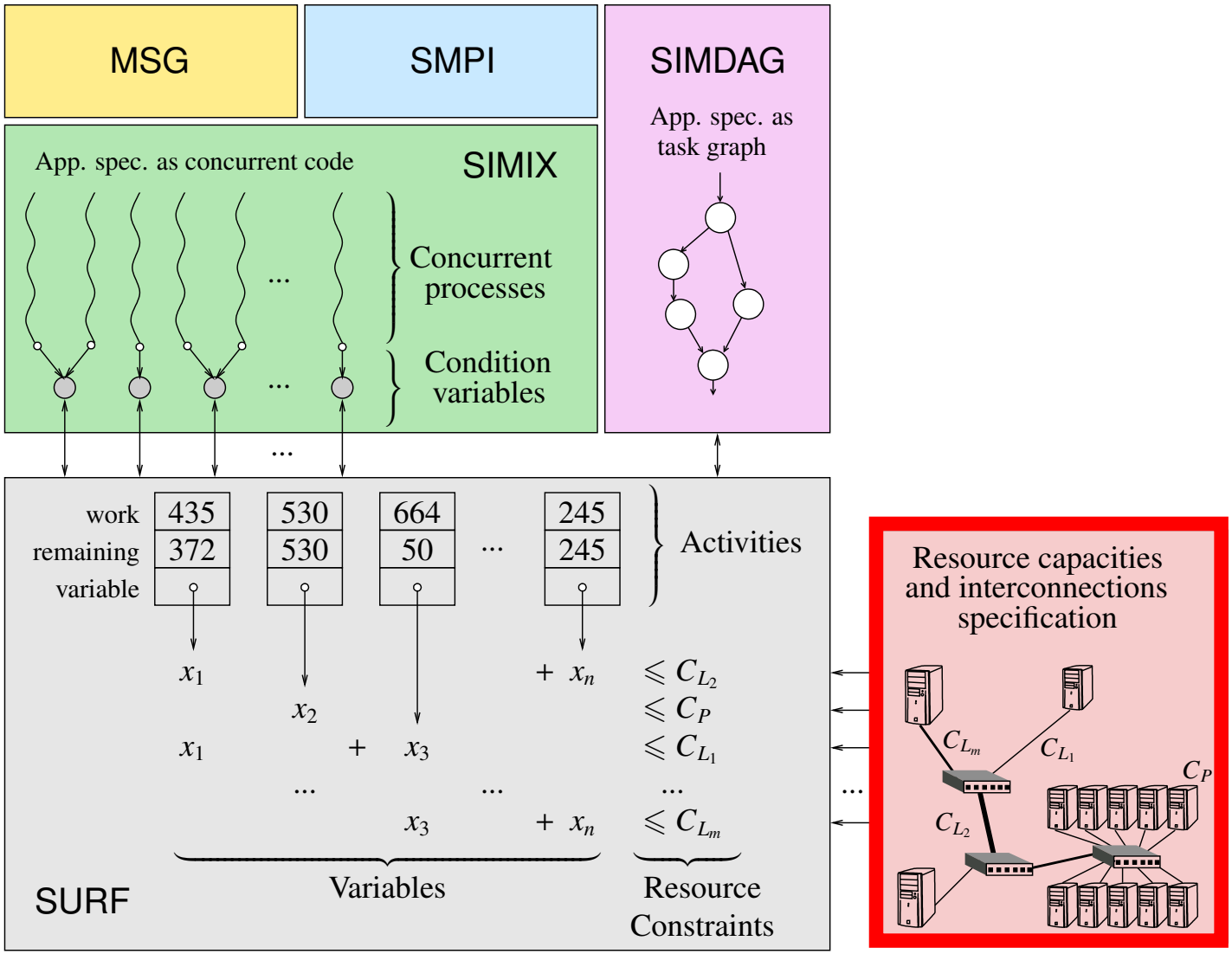
```
  msg_task_t task; int errcode; int id = atoi(argv[1]);
  char mailbox[80];
  sprintf(mailbox,"worker-%d",id);
  while(1) {
    /* Receive a task */
    errcode = MSG_task_receive(&task, mailbox);
    if (errcode != MSG_OK) {
      print("Error");  return errcode;
    }
    if (!strcmp(MSG_task_get_name(task),"finalize")) {
      MSG_task_destroy(task);
      break;
    }
    print("Processing %s", MSG_task_get_name(task));
    MSG_task_execute(task);
    print("Task %s done", MSG_task_get_name(task));
    MSG_task_destroy(task);
  }
  print("Worker done!");
  return 0;
}
```

Main program

int main(int argc, char *argv[ ]) {

```
  char *platform_file = "my_platform.xml";
  char *deployment_file = "my_deployment.xml";

  MSG_init(&argc,argv);

  /* Declare all existing processes, binding names to functions */
  MSG_function_register("master", &master);
  MSG_function_register("worker", &worker);

  /* Load a platform description */
  MSG_create_environment(platform_file);

  /* Load an application deployment description */
  MSG_launch_application(deployment_file);

  /* Launch the simulation (until its terminates) */
  MSG_main();

  print("Simulated execution time %g seconds",MSG_get_clock());
}
```

Platform and app deployment description



Platform description file (XML)

my_platform.xml

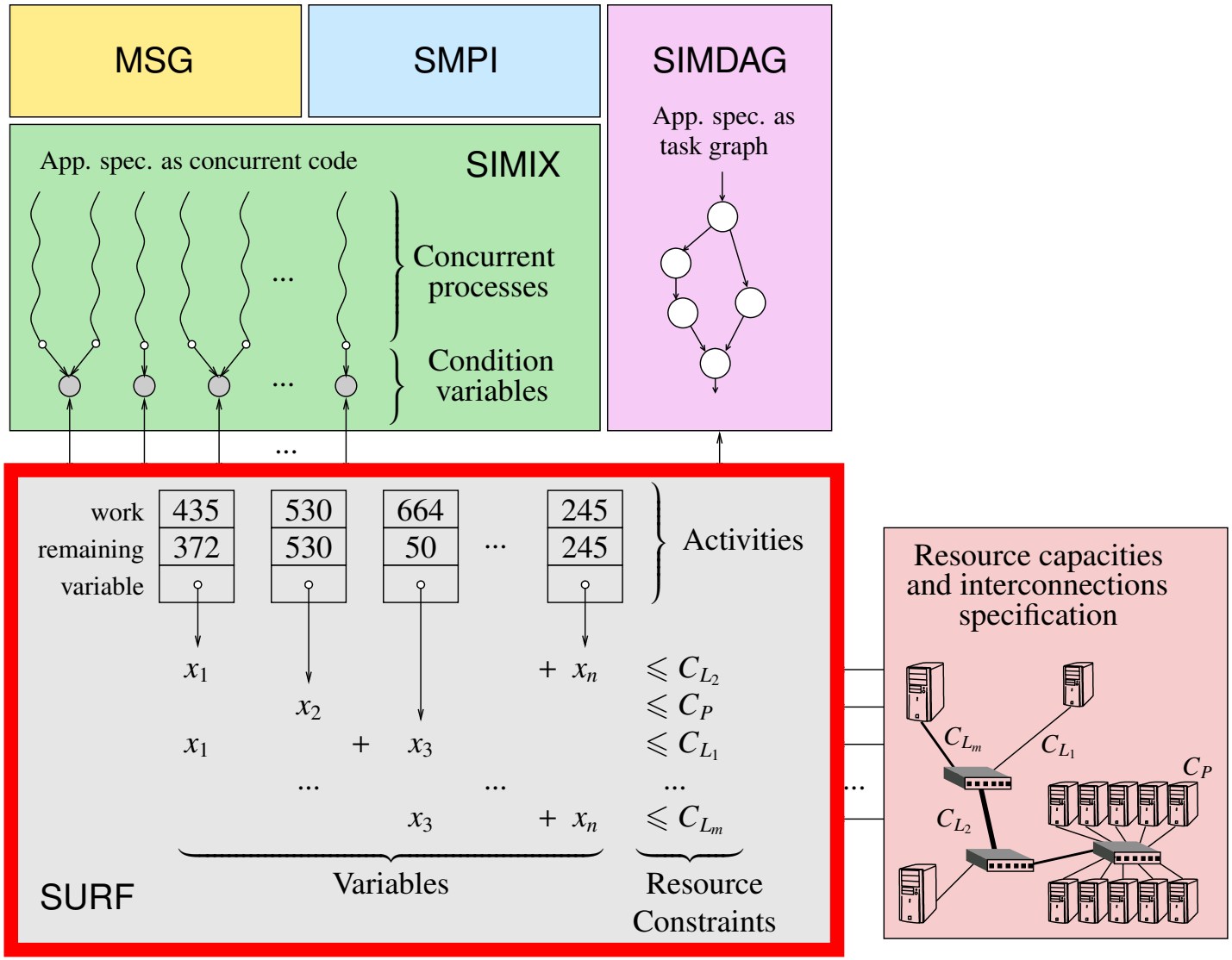
```
<?xml version="1.0"?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
  <platform version="3">
    <AS id="mynetwork" routing="Full">
      <host id="host1" power="1E6" />
      <host id="host2" power="1E8" />
      <host id="host3" power="1E6" />
      <host id="host4" power="1E9" />

      <link id="link1" bandwidth="1E4" latency="1E-3" />
      <link id="link2" bandwidth="1E5" latency="1E-2" />
      <link id="link3" bandwidth="1E6" latency="1E-2" />
      <link id="link4" bandwidth="1E6" latency="1E-1" />

      <route src="host1" dst="host2"> <link id="link1"/> <link id="link2"/> </route>
      <route src="host1" dst="host3">  <link id="link1"/> <link id="link3"/> </route>
      <route src="host1" dst="host4"> <link id="link1"/> <link id="link4"/> </route>
    </AS>
  </platform>
```

Application deployment description file (XML)

my_deployment.xml

```
<?xml version="1.0"?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
  <!-- The master (with some arguments) -->
  <process host="host1" function="master">
    <argument value="6"/> <!-- Number of tasks -->
    <argument value="50000000"/> <!-- Computation size of tasks -->
    <argument value="1000000"/> <!-- Communication size of tasks -->
    <argument value="3"/> <!-- Number of workers -->
  </process>

  <!-- The workers (argument: mailbox number to use) -->
  <process host="host2" function="worker"><argument value="0"/></process>
  <process host="host3" function="worker"><argument value="1"/></process>
  <process host="host4" function="worker"><argument value="2"/></process>
</platform>
```

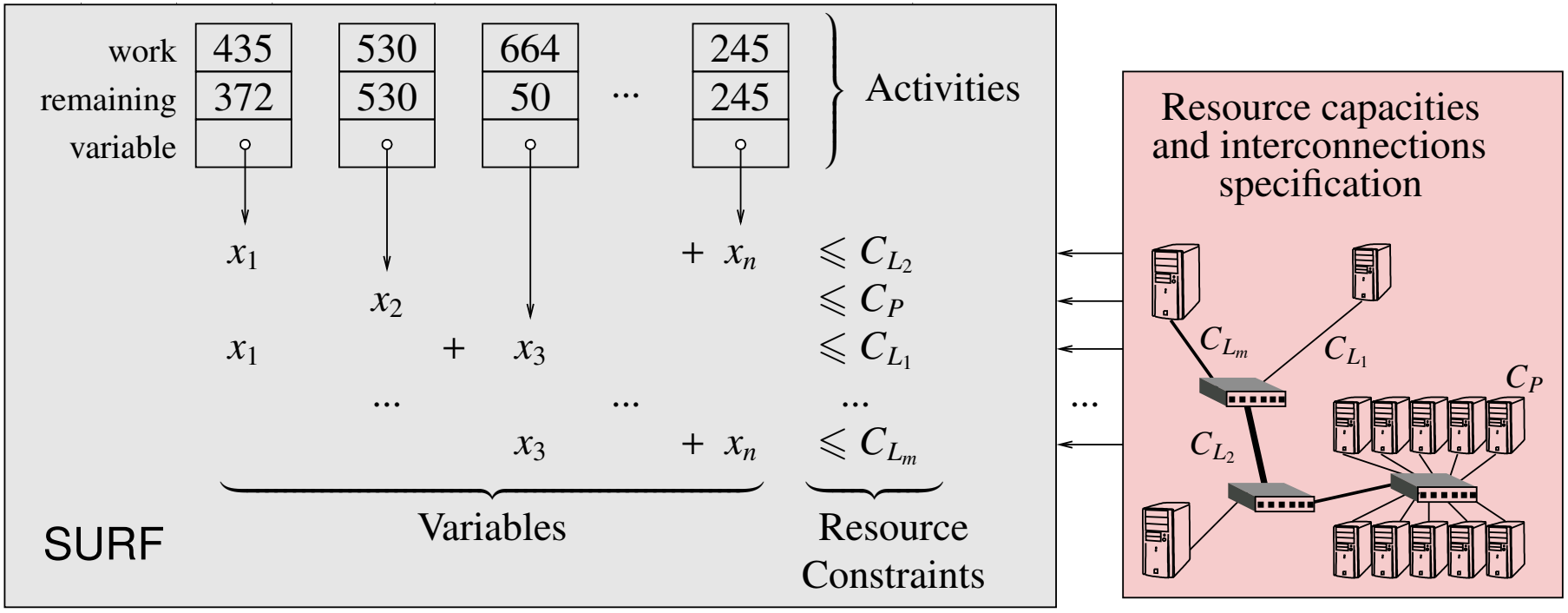
Simulation core



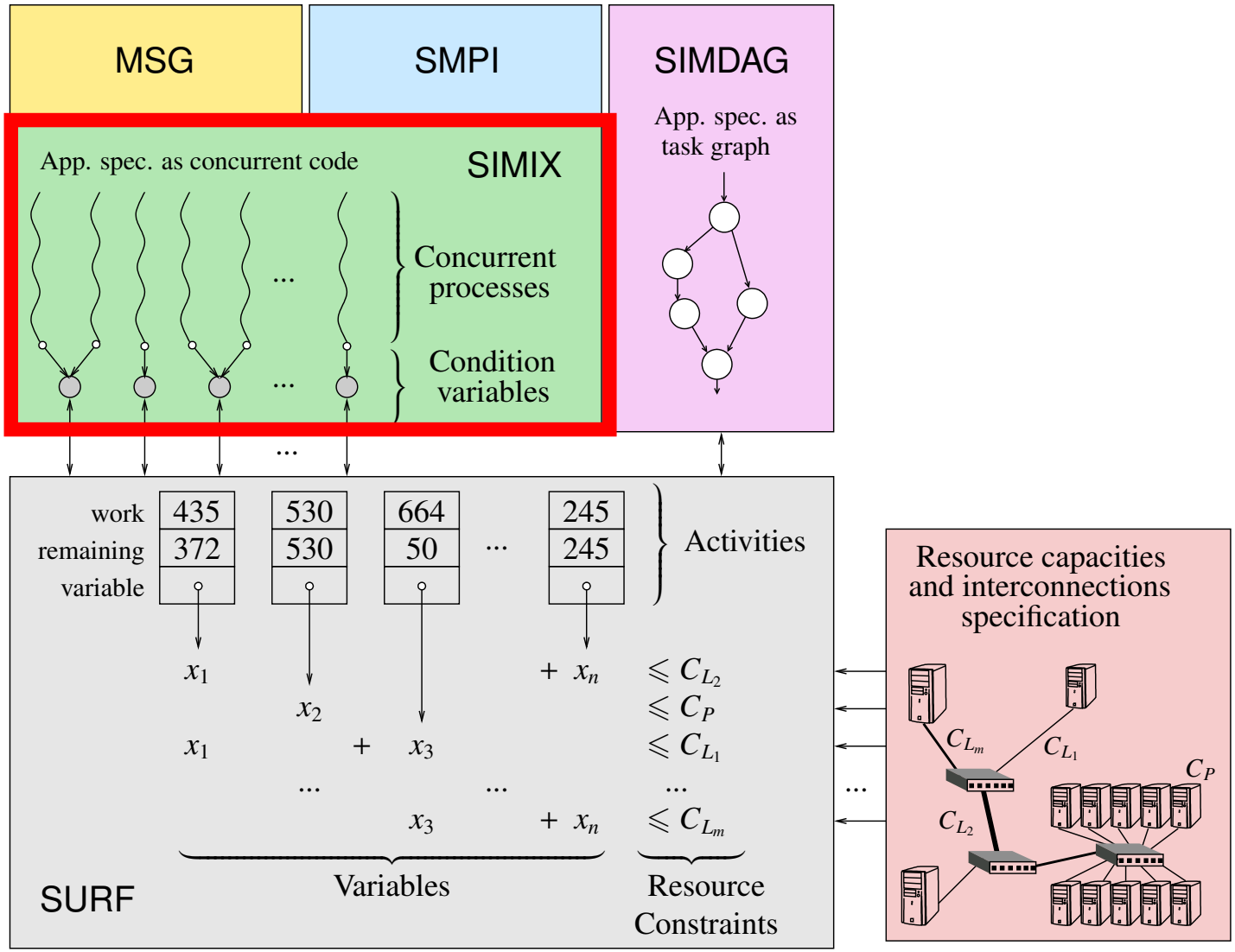
Simulation core

- The SURF component implements all simulation models
- All application activities are called actions
- SURF keeps track of all actions:
  - Work to do
  - Work that remains to be done
  - Link to a set of variables
- All action variables occur in constraints
  - Capture the fact that actions use one of more resources
- SURF solves the a (modified) Max-min constrained optimization problem betwen each simulation event
  - See previous seminar for more details
- Let's explain the example in the figure...

Simulation core



The SIMIX simcall interface



The SIMIX simcall interface

- SIMIX: an "OS kernel" on top of the simulation core
- Each simulated processes is a "thread" (more on this later)
- These threads run in mutual exclusion, round-robin, as controlled by SIMIX
- Each time a thread places an API call, translated to a simcall (a simulated syscall), it blocks on a condition variable in SIMIX
- When all threads are blocked in this way, SIMIX tells the simulation core computes the simulation models
- Threads are then unblocked and proceed until they all enter the SIMIX "kernel" again
- Total separation: application / synchronization / models

Scalability

- We have spent many years trying to increase scalability
- The first step was fast analytical resource modeling
  - Solving a weighted Max-min problem as opposed to packet-level, cycle-accurate simulation
  - Implementing the solver with cache-efficient data structures
- Scalability issues in SimGrid don't come from the models!

- Four limits to scalability:
  - X Running the simulation models too often
  - X Too large platform descriptions
  - X Too many simulated processes
  - X Simulation limited to a single core

Running models too often

- SimGrid was originally intended for simulating tightly-coupled parallel apps on hierarchical networks
  - e.g., a grid platforms with 3 clusters on a fast wide-area network for running parallel scientific applications
- In this setting, every simulated action can have an impact on every other simulated action
  - A data transfer completion frees up some bandwidth usable by many other transfers
  - A computation completion can lead to a message that will unblock many other computations
- As a result, SimGrid was implemented in the typical loop:
  - Run all simulation models
  - Determining the next event (e.g., action completions)
  - Update all actions remaining work amounts
  - Advance the simulated time and repeat

Running models too often

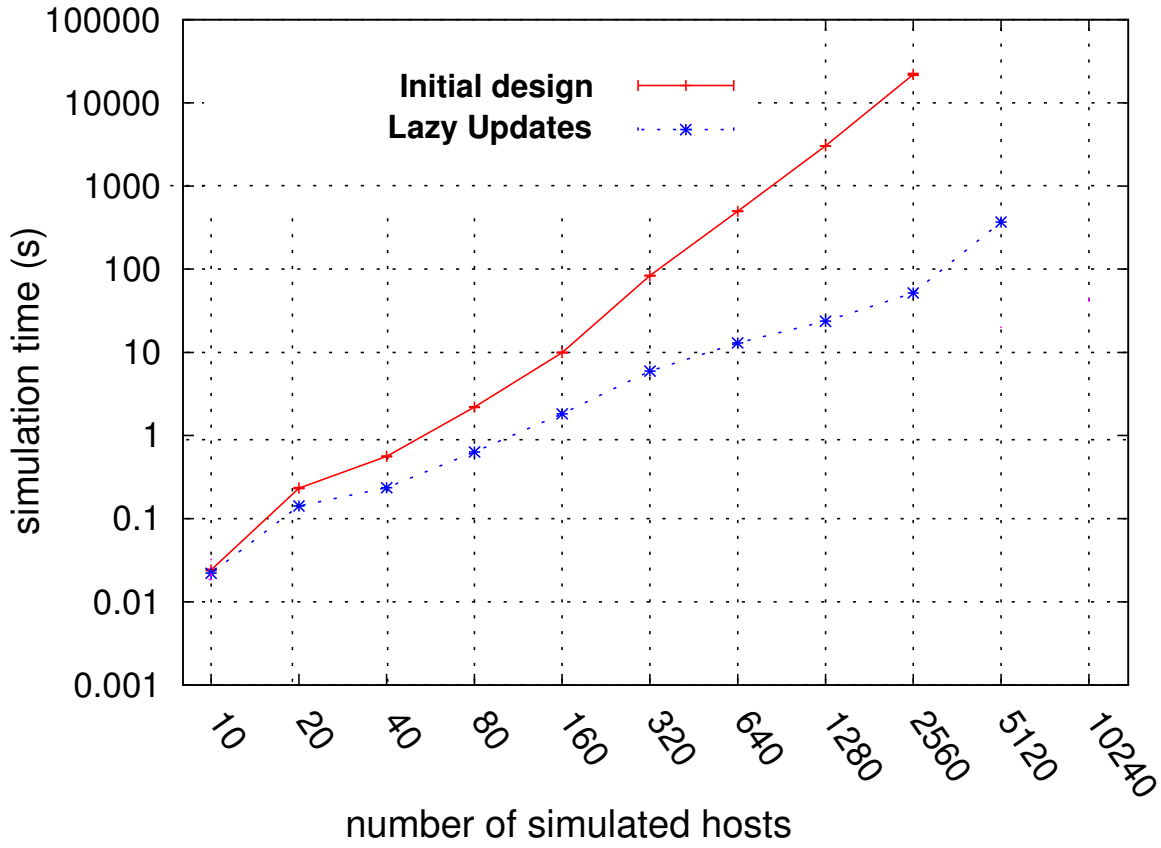
- As SimGrid gained popularity, we and users tried to apply it to different domains
- One such domain: volunteer computing
  - Donated compute cycles and disk space at the edge of the network to contribute to public-interest projects
  - e.g., SETI@Home, AIDS@Home, BOINC, etc.
- In this setting, many actions are independent
  - There is little resource contention among participating hosts
  - Computations are independent and long-running
- Yet there are many events (thousands of simulated hosts)
- Essentially, SimGrid keeps decreasing the remaining work amounts of all actions by ε over and over
- The result: slooooooow simulations at large-scale

Lazy action updates

- Modified "Lazy Updates" simulation loop:
  - All actions are stored in a heap, sorted by their current completion dates
  - When a resource state is modified, we remove relevant actions (those that use the resource) from the heap, we update their remaining work amounts and completion dates, and we re-insert them into the heap
  - Removing/Inserting from/to a heap: $O(\log n)$
  - Finding the next action that completes: $O(1)$
- Not a revolutionary idea of course
  - Large simulation literature on efficient *future event sets*
  - But not seen in parallel/distributed computing simulators
- If the application is tightly coupled, then lazy updates are slower because all actions are removed/inserted
- Lazy updates enabled by default but optional

Lazy updates in action

- Lazy updates for the motivating volunteer computing scenario [Heien et al., 2008]



SimGrid better than specialized simulator?

- Lazy updates are really effective
  - Example: from 3h to 1min for a simulation with 2,560 hosts on a 2.2GHz processor
- And in fact, SimGrid, even though it implements more sophisticated (network) models is $\sim$25 times faster than the SimBA volunteer computing simulator
- SimBA was optimized for scalability in a different way
  - It uses finite automata to describe simulated processes
- And yet, a more versatile simulator can "out-scale" it thanks to careful design

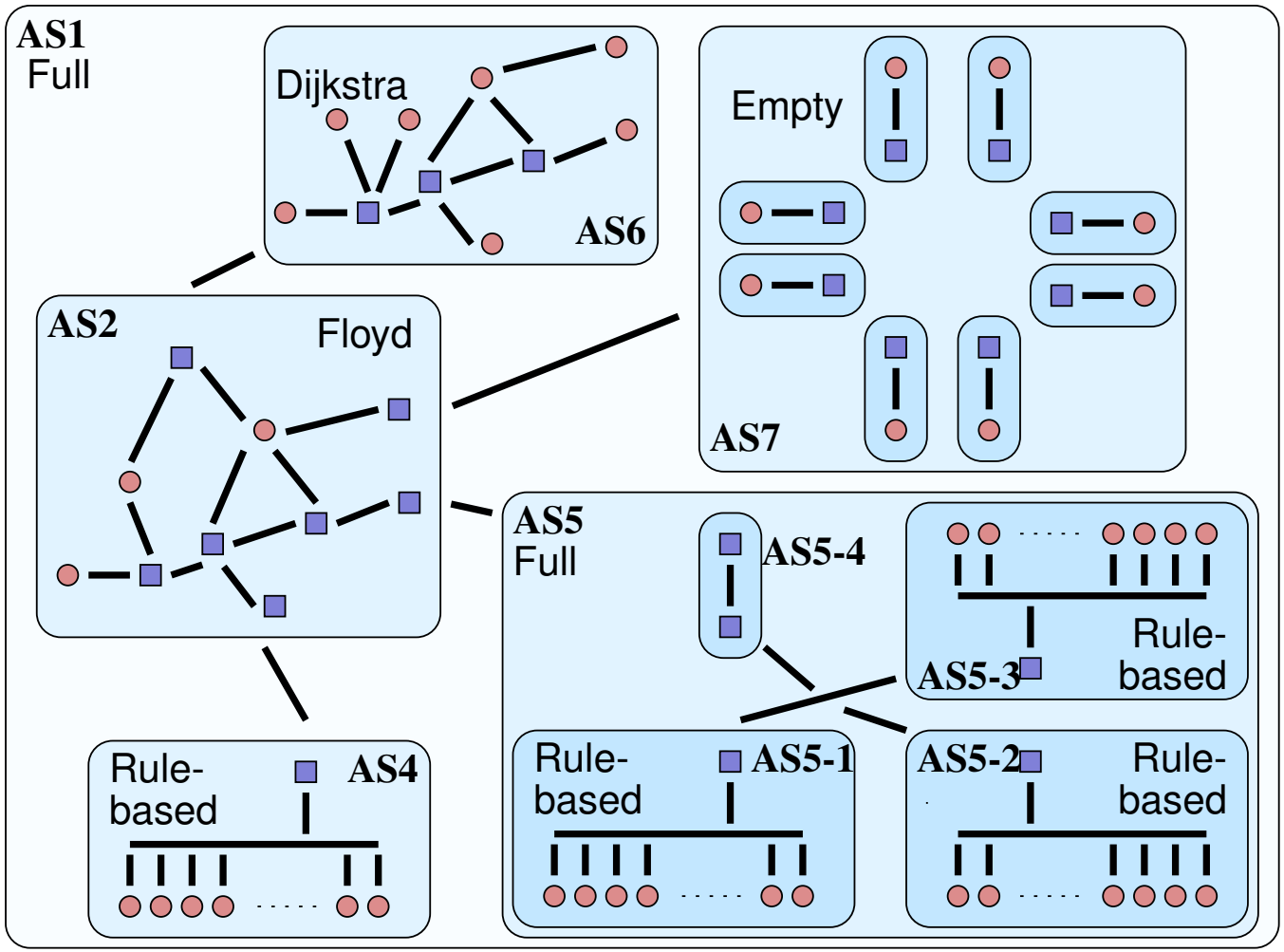
Four limits to SimGrid's scalability

- ✓ Running the simulation models too often
- X Too large platform descriptions
- X Too many simulated processes
- X Simulation limited to a single core

Large platform descriptions

- Users who used SimGrid for truly large-scale platform simulations often found themselves stuck
  - Long XML parse time
  - Out-of-memory errors
  - Long time to compute network routes, especially because we need $N \times (N-1)$ routes for N hosts!
- To enable large-scale simulation we must have hierarchical platform descriptions
- A platform is an Autonomous System (AS), that can contain interconnected ASes
- Each AS has its own routing scheme
  - Full routing tables, Dijkstra, Floyd, no routing, routing based on rules encoded as regular expressions

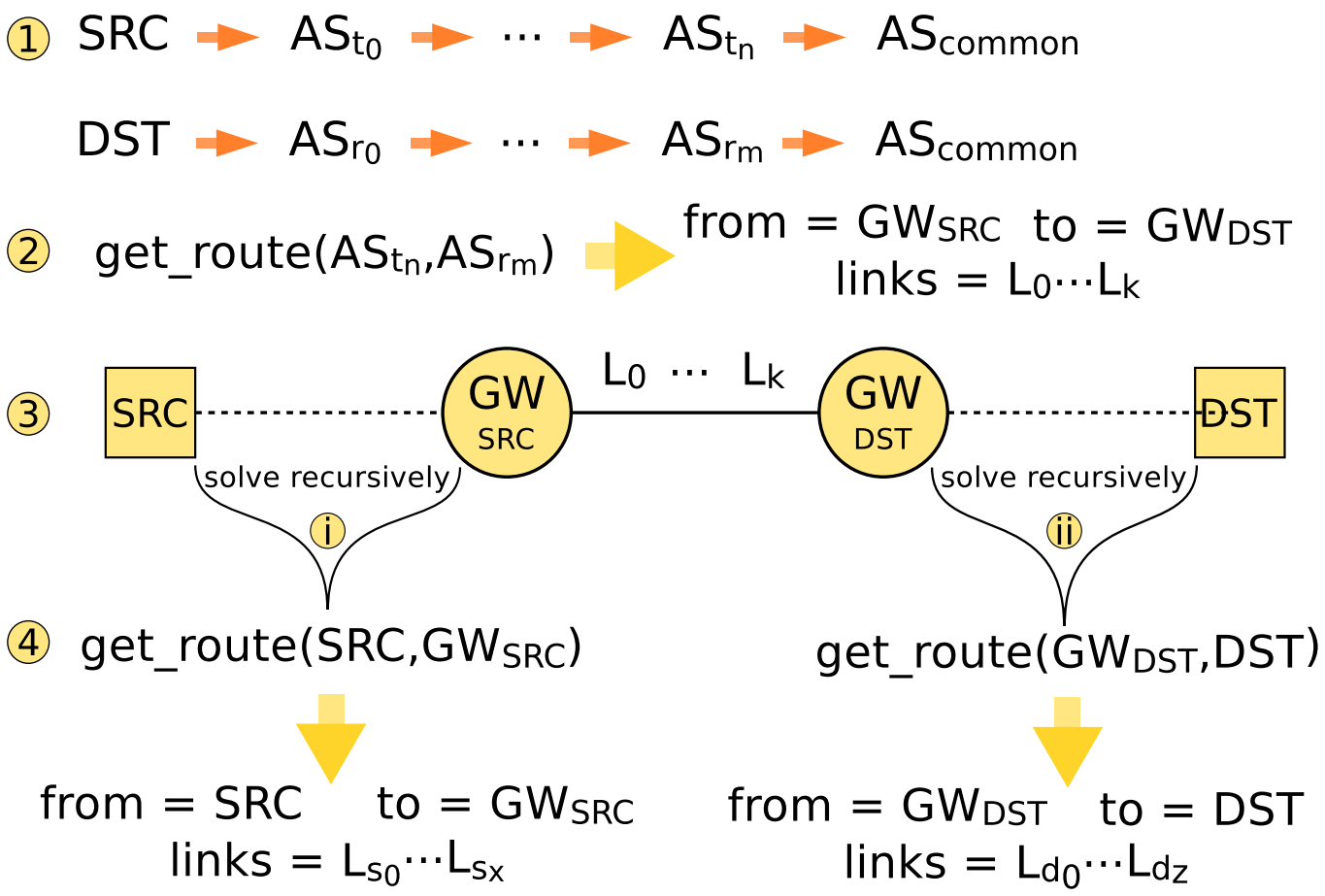
Platform description example



Recursive route computation

- Each AS declared *gateways* to other ASes, and that are used to compute routes
- To determine a route between two hosts:
  1. Search for the common ancestor AS
  2. Search for the route between the two relevant AS children
  3. Repeat until the full route is determined
- Let's see this on a figure...

Recursive route computation



Generating platform descriptions

- The SimGrid user can either generate an XML file, or used SimGrid's platform generation API
- The overhead of (recursively) computing the route is negligible in our implementation
- The memory footprint of the platform description is small
- XML parsing is fast
- Example:
  - The Grid'5000 testbed (10 sites, 40 clusters, 1,500 nodes)
  - Described with 22KiB of XML, parsed in < 1s
  - Previous SimGrid versions: 520MiB, parsed in 20min

Four limits to SimGrid's scalability

- ✓ Running the simulation models too often
- ✓ Too large platform descriptions
- **X** Too many simulated processes
- **X** Simulation limited to a single core

Too many threads

- SimGrid allows users to described simulated apps as sets of CSPs
- Great for flexibility and expressivity
- Not scalable if implemented as processes/threads:
  - Thread creation/management overhead in the kernel
  - Thread memory footprint in the kernel
  - Thread synchronization overhead (locks + condvar)
- **But** in a SimGrid simulation threads run in mutual exclusion and in a round-robin fashion
- Therefore, we don't need the full power/flexibility of kernel threads since we do our own scheduling and our own synchronization

Scalable CSPs (I)

- As opposed to having threads each with a bunch of locks and condition variables we take a different approach
- A single "core context":
  - Since simulation models are fast, a single thread does all model computations (i.e., it run the SURF code)
  - All simulated processes place SIMIX simcalls, and all these simcalls are resolved by the core context: no shared state among threads
    - Two simulated processes waiting on each other don't really wait on each other
    - i.e., no multi-step process-to-process interactions
    - Instead, they place wait/notify-like simcalls to the core context

Scalable CSPs (I)

- Lightweight "continuations":
  - Since we don't need full threads we can use cooperative, light-weight, non-preemptive threads
    - Known as continuations
    - No actual context-switching by the kernel
  - Windows: *fibers*
  - Linux, Mac OSX: *ucontexts*
  - We actually re-implemented them in assembly to avoid a costly system call that is not needed for our purpose

How scalable is it?

- With all three scalability improvements so far, we can now compare SimGrid to "competitors"
- Case study #1: Grid computing
  - Master-worker scenario
  - Comparison to GridSim (implemented in Java)
- Case study #2: Peer-to-peer computing
  - The Chord protocol [Stoica et al., 2003]
  - Comparison to PeerSim and OverSim

Scalability case study #1

- One master, N workers, P tasks, round-robin scheduling
- Simulation on a 2.4GHz core and 8GiB of RAM

- GridSim:
  - No network topology simulated (simply latency+bandwidth communication costs)
- SimGrid:
  - Grid'5000 topology simulated (with TCP flow-level modeling, etc.)

Scalability case study #1

- Polynomial fits based on measured values

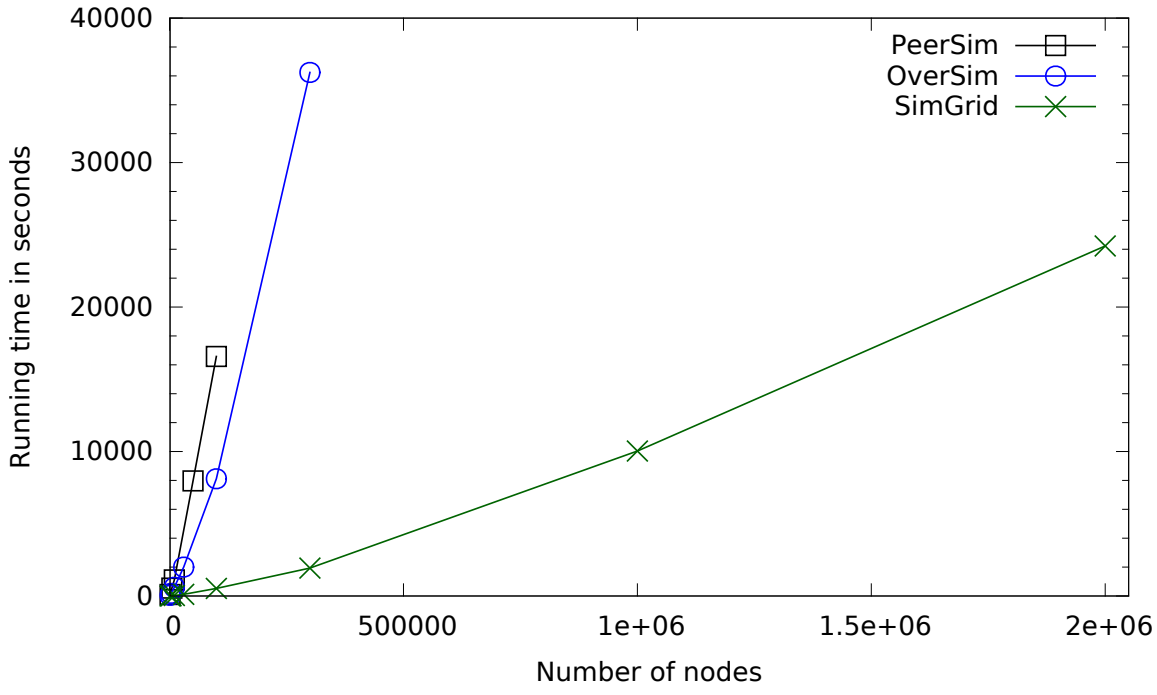
	Simulation Time (s)	Peak Memory Footprint (byte)
GridSim	$5.599 \times 10^{-2}P + 1.405 \times 10^{-8}N^2$	$2.457 \times 10^{6} + 226.6P + 3.11N$
SimGrid	$1.021 \times 10^{-4}P + 2.588 \times 10^{-5}N$	$5188 + 79.9P$

- Example: $N = 2,000$, $P = 500,000$
  - GridSim: 4.4GiB of RAM, > 1hour
  - SimGrid: 165MiB of RAM, < 14s
- And SimGrid uses more sophisticated models!

Scalability case study #2

- Implementation of the Chord protocol for N hosts
- Simulations on a 1.7Ghz core with 48GiB of RAM
- SimGrid
  - TCP flow-level modeling on a full topology
- OverSim [Baumgart et al., 2007]
  - Communication delays based on Euclidian distance between peers
  - Implemented in C++
- PeerSim [Montresor et al., 2009]
  - Constant communication delays
  - Implemented in Java

Scalability case study #2



- PeerSim: 100,000 peers in 4h36min
- OverSim: 200,000 peers in 10h
- SimGrid: 2,000,000 peers in 32min

Scalability results

- We have shown SimGrid to be faster than specialized simulators, even when it uses more sophisticated network simulation models!
- Some of these simulators were designed specifically for scalability (especially p2p simulators)
  - Of course, they may suffer from implementation inefficiencies, while we have spent hours trying to optimize our implementation
- Nevertheless, we claim that it is not necessary to be specialized to be scalable, at least for parallel/distributed computing simulations
- Can we go further?

Four limits to SimGrid's scalability

- ✓ Running the simulation models too often
- ✓ Too large platform descriptions
- ✓ Too many simulated processes
- **X** Simulation limited to a single core

Parallelizing SimGrid

- Because of all the optimizations we've talked about, often most of the compute time is spent in user code!
  - What simulated processes do outside of SimGrid
- There is thus no need to parallelize SimGrid's internals
  - Which would be very difficult anyway since Parallel Discrete Event Simulation is difficult
- We are thus able to run concurrent user processes easily on multiple cores
- Experiments for "difficult cases" (e.g., peer-to-peer Chord) show that achieved speedup is minimal (13%) but non-zero
- Experiments for "easy cases" (e.g., simulated processes that do complex logic in between calls to SimGrid) show that achieved speedup up is large

The SIMGRID community

- SIMGRID is both a *usable simulator* and a *research vehicle*
  - Research papers with results obtained with SIMGRID
  - Research papers about SIMGRID itself
- SIMGRID has a large user community and a large development team
- SIMGRID is well-funded for the upcoming years
- SIMGRID welcomes collaborators, patches, comments, typo fixes in the documentation ☺

Where to find out more information

http://simgrid.gforge.inria.fr

The End

This concludes this 6-seminar series

Thanks again to NII for the invitation

I am always available for questions
henric@hawaii.edu