An Extensible Timing Infrastructure for Adaptive Large-scale Applications

Dylan Stark, Gabrielle Allen, Tom Goodale,
Thomas Radke (AEI), Erik Schnetter

Gdańsk, September 2007

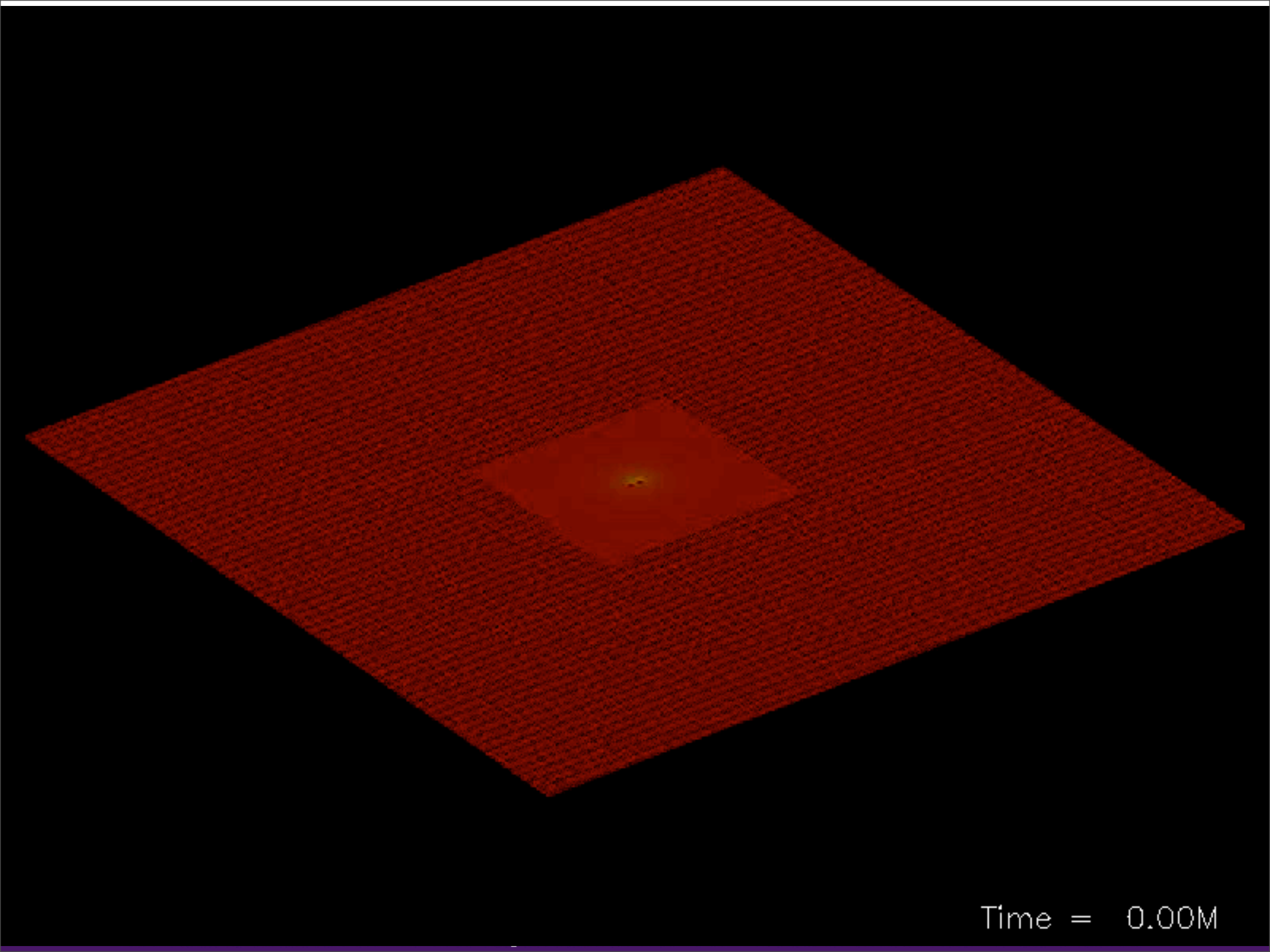
http://www.cct.lsu.edu/about/focus/numerical

Outline

- Black holes, motivating our research
- Cactus, a software framework
- Cactus timing infrastructure
- Checkpointing, an example scenario

Black Holes

- Black holes are out there (see e.g. http://www.mpe.mpg.de/ir/GC or http://en.wikipedia.org/wiki/Black_hole)
- Studying black holes is difficult (far away, dangerous)
- Therefore we use HPC simulations to predict their behaviour, then combine these with observational results

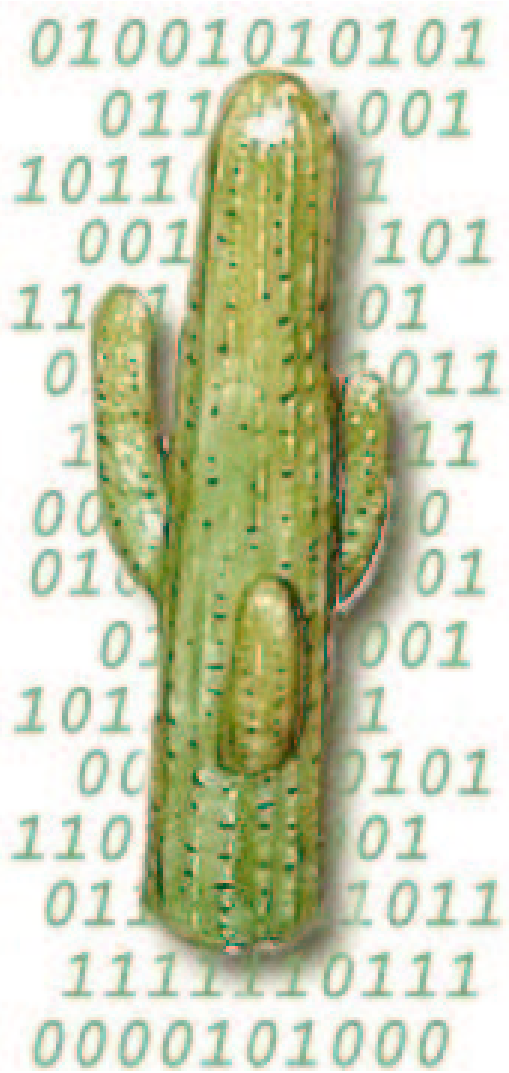

Time = 0.00M

Software Frameworks

- Our code: 10 years in the making, many authors, first authors have left the group
- Mixture of C/C++/Fortran 77/Fortran 90/ Perl, > 2 Mloc, many outdated parts
- Need to organise ourselves to manage the code
- We use a *Software Framework*, and we split the code into *components* (see Jarek Nieplocha's talk on Tuesday)

Cactus

- In Cactus speak, the framework is called *flesh*, and the components *thorns*
- There are many public thorns which use Cactus, especially for numerical relativity
- http://www.cactuscode.org



Saguaro

(Carnegiea gigantea)

Library vs. Framework

- A framework is like a library, except that it contains the main programme -- the user modules are libraries
- Crucial for easy interoperability -- otherwise, two modules may "fight" over who may be the main programme
- Cactus thorns are "connected" via their schedule
- Schedule is constructed at run time -- no code needs to know all compiled thorns
- Thorns can be developed completely independently

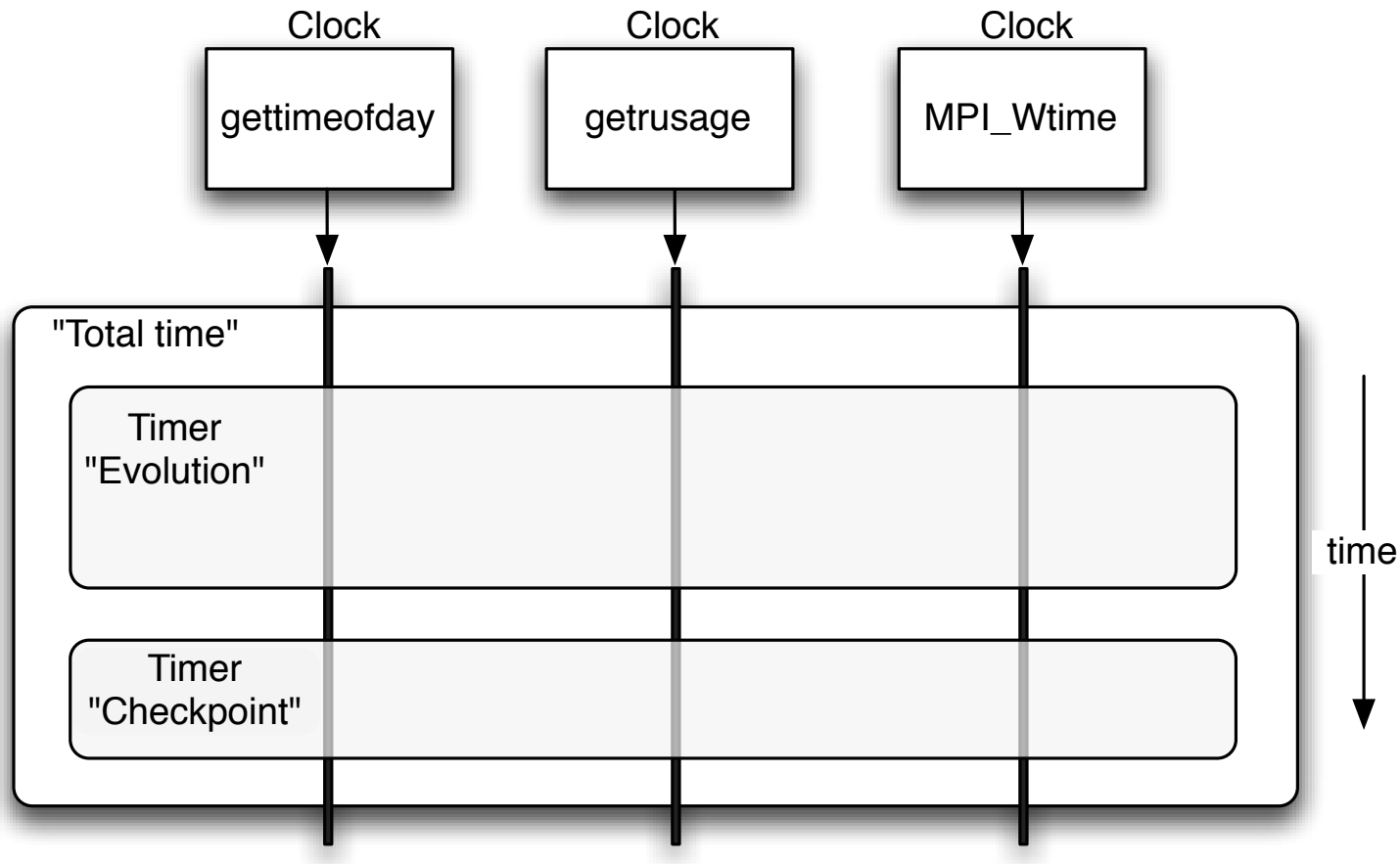
The Basic Idea

- The framework has much information about the overall programme (while the components are all independent)
- Thorns can query the framework to access this information to optimise their behaviour
- Cactus keeps timers for every thorn -- and thorns can look at Cactus's timers to reduce overhead

Clocks and Timers

- A *Clock* is a way to measure a certain kind of time, e.g. wall time or CPU time or cycles or ...
- A *Timer* contains a set of clocks; it can be started and stopped and read out and printed

Timers and Clocks



Three (partly nested) timers, using three different clocks

Other Possible Clocks

- clocks derived from PAPI or TAU
- rdtsc (Intel)
- omp_get_wtime
- clocks measuring discrete events, e.g. number of MPI messages sent/received
- clocks measuring bytes written to disk

Clock API

Function	Description
create	Create a new clock, returning a pointer to it
destroy	Destroy the clock
start	Start this clock
stop	Stop this clock
reset	Reset this clock, i.e., set the accumulated time to zero
get	Get the clock's values
set	Set the clock's values

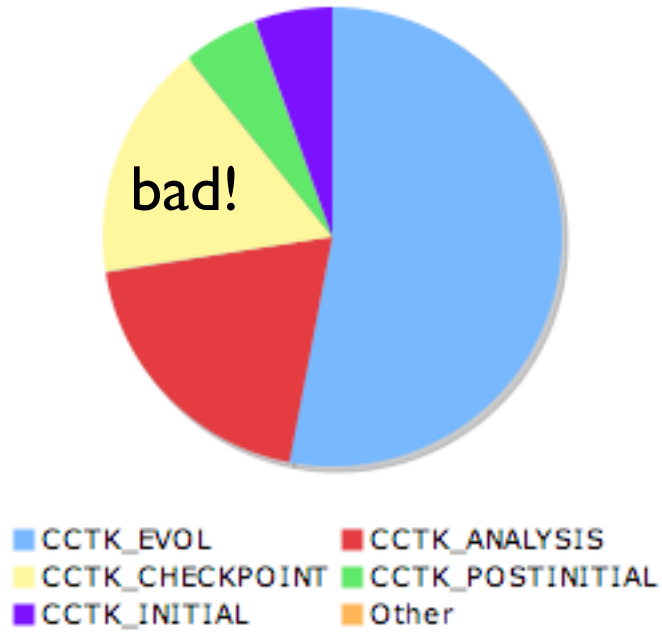
Timer example

```
/* Create timer */
static int handle = -1;
if (handle < 0) {
  handle = CCTK_TimerCreate ("Poisson: Evaluate residual");
  if (handle < 0) CCTK_WARN (CCTK_WARN_ABORT, "Could not create timer");
}
... other code ...
CCTK_TimerStartI (handle); /* Start timer */
... evaluate residual ...
CCTK_StopTimerI (handle); /* Stop timer */
... other code ...
CCTK_TimerPrintDataI (handle, -1); /* Output all clocks of this timer */
```

Improving Checkpointing Efficiency

- Checkpointing: From time to time the complete simulation state is saved to disk
- Necessary to recover from failures (fault tolerance!), or to overcome queue time limits ("local policies")
- Checkpointing too often is unnecessary overhead

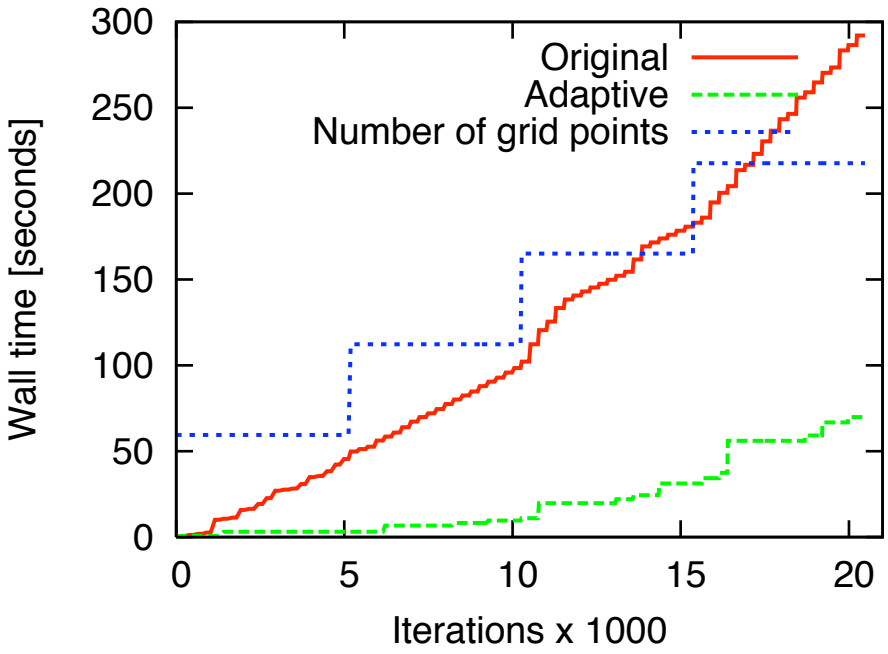
Timing chart
of a typical simulation
(measured automatically)

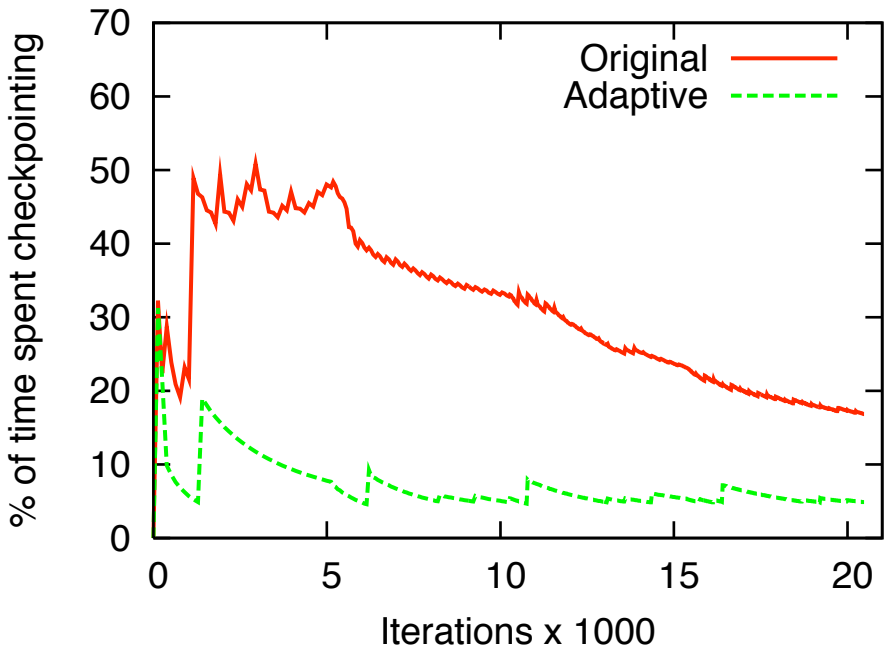


Experimental Setup

- Scenario: A simulation of a collapsing star, adding more refined grid as the star shrinks
- We adapt the checkpointing interval dynamically to satisfy certain criteria:
  - Goal: spend 5% of overall time checkpointing
  - Subject to some constraints: not too often, not too seldom ("at least every N iterations")

Experimental Results





Discussion

- Scenario is “fake”, but demonstrates that dynamic checkpointing intervals are important here (but we knew that)
- Information gathered by the framework and queried by the checkpointing components can be used to improve efficiency
- Current implementation is good enough for production use in black hole simulations

Future Scenarios

- Analyse simulation results at run-time if efficient, defer to post-processing otherwise
- Compare efficiency to other machines, migrate simulation if this saves time
- Monitor simulation, change number of processors at run time to improve efficiency

Conclusions

- The Cactus framework has a generic timing infrastructure (timers and clocks); user can add new clocks
- Timers are conveniently and automatically added to components by the framework
- Components can query timers and steer simulation parameters dynamically, improving efficiency *without prior system knowledge*

CCT

- Interdisciplinary research centre at LSU, about four years old
- Computer science, physics, mathematics, biology, music, ...
- http://www.cct.lsu.edu

T. Sterling