

# CPR

**Where we've come from**  
**Why we're not further today**  
**How to plan for the future**





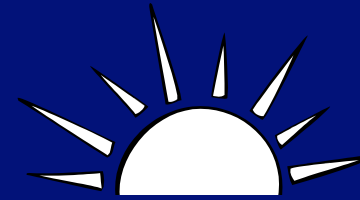
# Why bother with CPR?

- If your application is not fault-tolerant:
  - You won't be able to get anything done
  - You won't be allowed to try on PCS-1





# In the beginning...



B.C. = Before Clusters

...there was Cray. (First among MPPs...)

## □ Kernel-level CPR

- Worked for *almost* everything
  - And the *exemption* list got shorter every year!
- Users loved it
  - Because they didn't know about it → *transparent*
- Administrators loved it
  - Because users didn't know about it (no flame email !)
  - Complete freedom in resource (re)allocation



...and then there were none.

A.D. = Anno Distributo

The demise of the “MPP”

➤ the rise of *distributed machines*

- ❑ Users mourned the consequent loss of CPR
  - “Darn! That’s the end!” – SchoolHouse Rock
- ❑ Administrators struggled against the vendors
  - “You don’t have CPR, and I can’t even see your source code!”
  - “If everyone ran Linux I could do this myself... only the kernels keep changing too fast.”
  - But... There’s no way to synchronize coherent multi-system snapshots!





# Your future Petaflops Machine

Consider what this will look like:



- Highly parallel
- Many processors
  - Not just faster – can't bank on Moore's law to give you back a PCS SMP (not for a while, at least)
- Many “nodes”
  - Blades (SMP), CPU “modules” (MPP), P.I.M.
- Many file systems (or at least file streams)
- Many... breakable parts → need CPR!!!



# Who is using TCS-1?

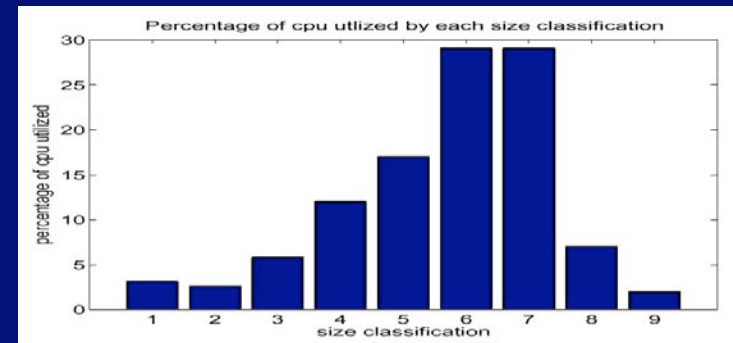
## □ TCS-1 utilization

- “4” is 64-127
- “6” is 256-511
- “7” is 512-1023
- Majority at 1/3 to 1/10

□ Q: What happened to all the users w/ PEs < 64 ?

□ A: They aren't allowed to run here

- Scale or Starve!
- Ignore scaling < 64 PEs...





# Who will will be using PCS-1?

- Those with highly scalable applications
  - Everyone else will run elsewhere (or nowhere)
    - Not approved for computing time...
- Those with Fault-Tolerant applications
  - Either CPR or FT algorithm
    - Which is easier to write?

*“We are lucky that we had to suffer through that [communication] transition.” – S.G.  
[because we re-wrote it and now it’s better]*

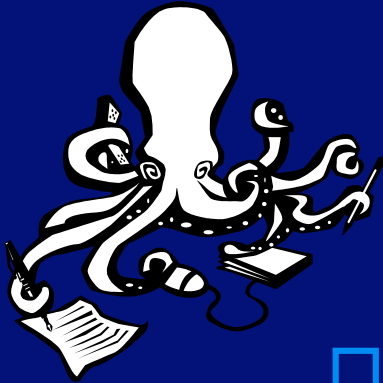




# CPR Challenges

## □ Scope & Expertise

- True CPR touches many aspects of both the system and the application
  - Memory, network(s), file systems, exec. stack, app. libraries



## □ Effort & Money

- It would take a long time to do it right
  - Attention to detail → missed use cases
  - vs. Performance → know “when to quit”



# Who will solve this problem?

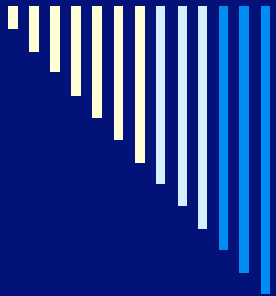
"If there must be trouble let it be in our day,  
so that our children may have peace."

– Thomas Paine



- *"You're the one."*  
– Paul Simon
- *"You are the man!"*  
– Nathan, the prophet
- **You** are the architects  
of the apps of tomorrow!





# No, really... Who will solve the problem?

- Someone else? (Please!?)
    - Vendors: *“They know the most about the HW.”*
      - But there are never C.O.T.S. PetaFlops systems
      - Kernel-level (distributed) CPR isn’t coming back
        - While U wait, your competitors win!
        - Not portable (even if it *does* come back...)
    - Government: *“They should demand it”*
    - Software community: *“They should create it.”* (compilers, S.C. developers, etc.)
      - Wait for universities and S.C. centers to *standardize*?
- \*\*\*

*Ask not what your computer can do in spite of you,  
ask what you can do with your computer!*





# Motivation: Why Checkpoint?

## □ Fault-tolerance

- Capability of restarting “where you left off” following a hardware failure (*MTTF of 1/yr \* 750 = 2/day*)

## □ Job Scale

- Typical TCS running time for a “complete” project is already of order CPU-months
  - (and you rarely get that all in one shot) -- queueing

## □ Steering

- Capability for stopping/turning jobs that are going “the wrong direction”





# CPR Features

- ❑ An application-level CPR library
  - C/C++/Fortran API & libs
- ❑ Infrastructural services
  - I/O daemons, replication, C.P. state mgmt, tracking
- ❑ Cluster Management integration
- ❑ A promise:
  - If there is a system failure while you're using our CPR system...
    - ❑ Automated (successful) restart
    - ❑ Credit for lost time





# Strategy: the TCS CPR API

## ***System functions:***

```
open(char *file, int flags, mode_t mode)
read(int fd, void *ptr, size_t len)
write(int fd, void *ptr, size_t len)
close(int fd)
```

## Bonus:

This file-oriented API is widely applicable to other systems & applications

## ***CPR functions:***

```
tcs_open(char *prefix)
tcs_read(int lun, void *ptr, size_t len)
tcs_write(int lun, void *ptr, size_t len)
tcs_close(int lun)
```

and...

```
tcs_init(void *not_used)
tcs_finalize()
tcs_jobrestarted()
```

and that's not all...

```
tcs_drainoperation()
tcs_preservestate()
tcs_postmessage()
```





# Strategy: How to Checkpoint

- Choose your checkpoint interval (algorithm-dep.)
  1. Loop-based (e.g. every <N> iterations)
  4. Feature-based (e.g. at *stable* points, adaptive)
  5. Triggered (e.g. external input)
- Write all of the “essential” arrays/globals
  - Only those that cannot be regenerated
  - *Re-use* them if at all possible
- Write the loop counters
  - Incremented by one (!)
    - start from values in C.P. file
- Use a large blocksize, if possible...





# Strategy: Function vs. Flag

- CPR Function(s) – concentrated
  - `checkpoint_me()` / `recover_me()`
    - Works well with global-scoped or few arrays
  - Concentrates all of the CPR-related I/O in one place
    - Easiest to debug (or upgrade) CPR I/O
- CPR Flag(s) – dispersed
  - `doCheckpoint` / `doRecover` (global/common)
    - Keeps CPR I/O “close” to the engine...
- Hybrid: a CPR I/O **region** in the code...
  - e.g. all C.P. I/O at *end of loop*, recover at beginning





# CP File Issues

- Naming conventions
  - Make it predictable (fixed prescription)
  - Avoid collisions (for multi-step, multi-stream)
- Number of files
  - Wildcards can't match more than <256-2048> files
  - Use subdirectories wisely (data→directory struct's)
  - Write fewer (global?) files
- File paths
  - Use ENV variables, not PWD (this is problematic)
- Consider file replication (?)



# Specific Recommendations



The “Microsoft Keyboard”

your output data

ome here

”

you write





# Trends to watch

- Diskless compute nodes
  - How will this affect your *I/O patterns*?
    - Stay configurable!
- I/O directly to HSM archives
  - Free redundancy, higher latency(?), many ioctls
- Heavy-weight data management (organization/transfer) software
  - Might be worth the investment esp. with large numbers of files





# The Call to Responsibility

“Let me add that only a virtuous people are capable of freedom. As nations become corrupt and vicious, they have more need of masters.”  
– Thomas Jefferson

I keep giving this talk...

(too often) to the sound of echoes.

- Remember: either add CPR or rewrite your algorithm!
- Like security: *“until you lose something of value...”*
  - midnight the night before SC’09?





# Questions or Comments?

- Nathan Stone

- <http://www.psc.edu/~nstone/>
- <mailto:stone@psc.edu>
  - See white-paper for more details

- PSC Advanced Systems Group

- [http://www.psc.edu/advanced\\_systems/](http://www.psc.edu/advanced_systems/)
- <mailto:advsys@psc.edu>

- PSC Terascale Computing System Status

- <http://www.psc.edu/machines/tcs/status/>

