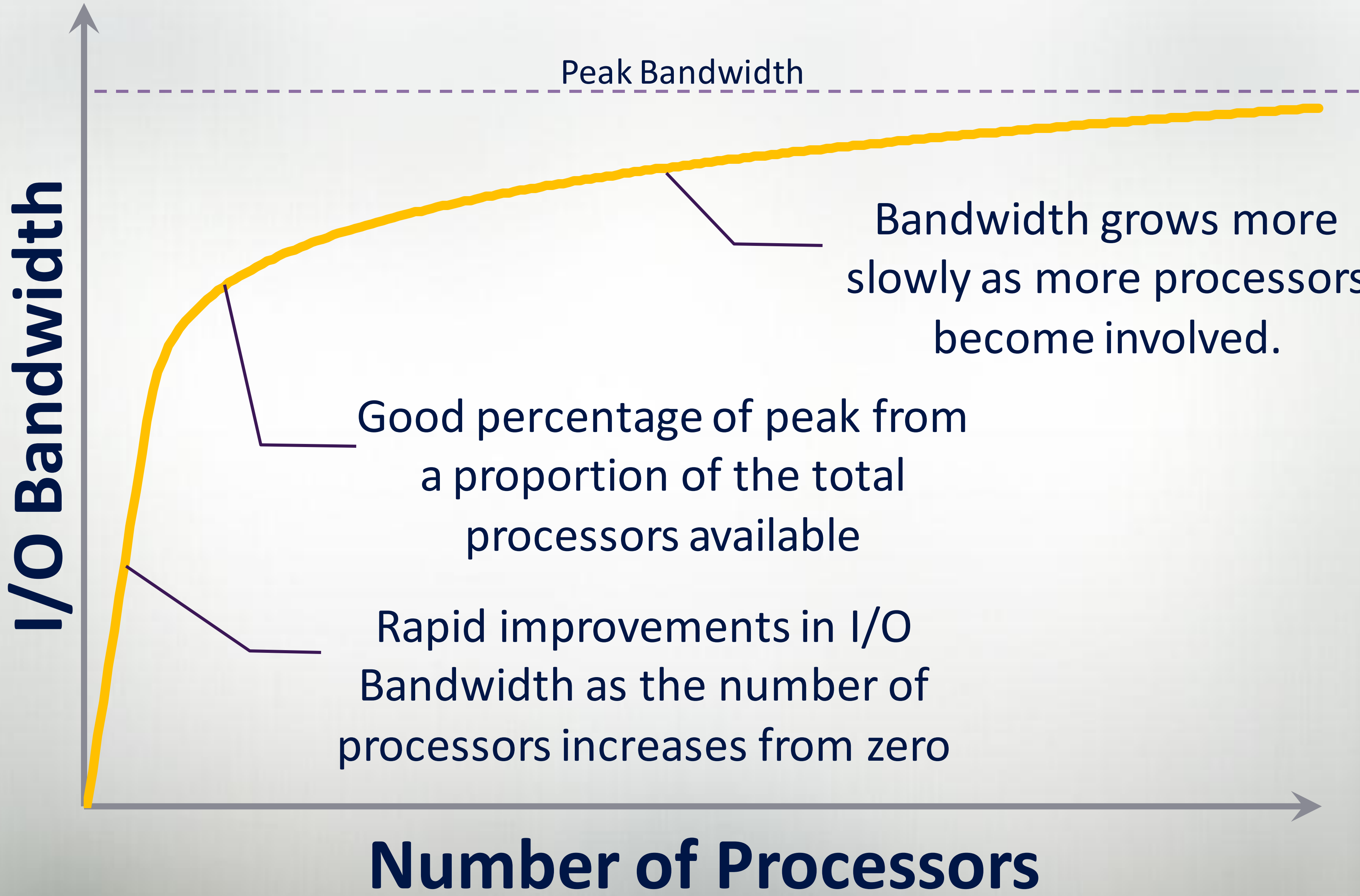
Using I/O Servers to Improve Application Performance on Cray XT Technology

Thomas Edwards, Kevin Roy
Cray Centre of Excellence for HECToR

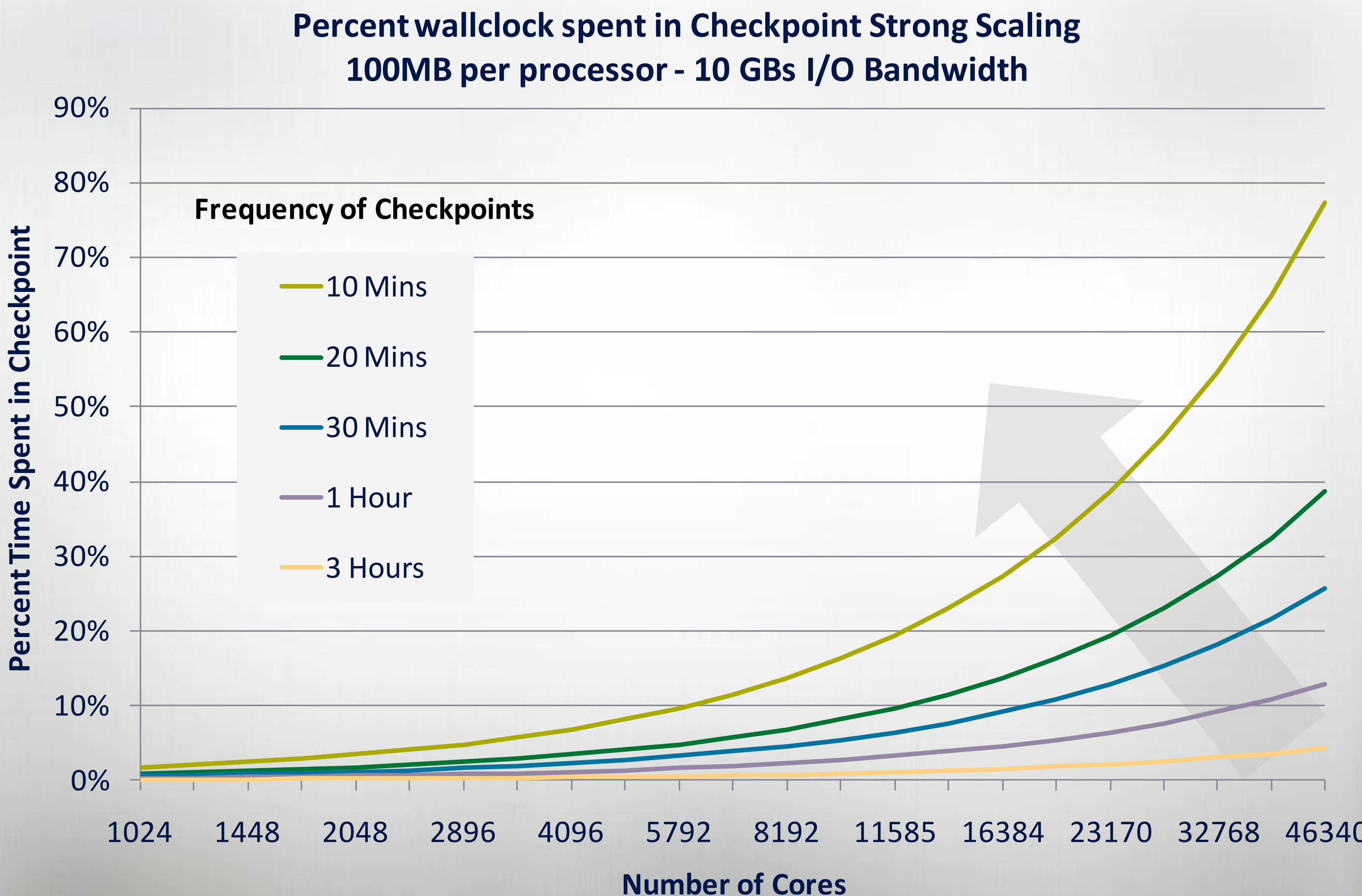
Overview

- This talk is not about how to get maximum performance from a Lustre file system.
  - Plenty of information about tuning Lustre Performance
    - Previous CUGs
    - Lustre User Groups
- This talk is about a way to design applications to be independent of I/O performance
  - All about Output, but Input technically possible with explicit pre-posting

Lustre Performance Profile Model



Weak Scaling Checkpoint Cost Model



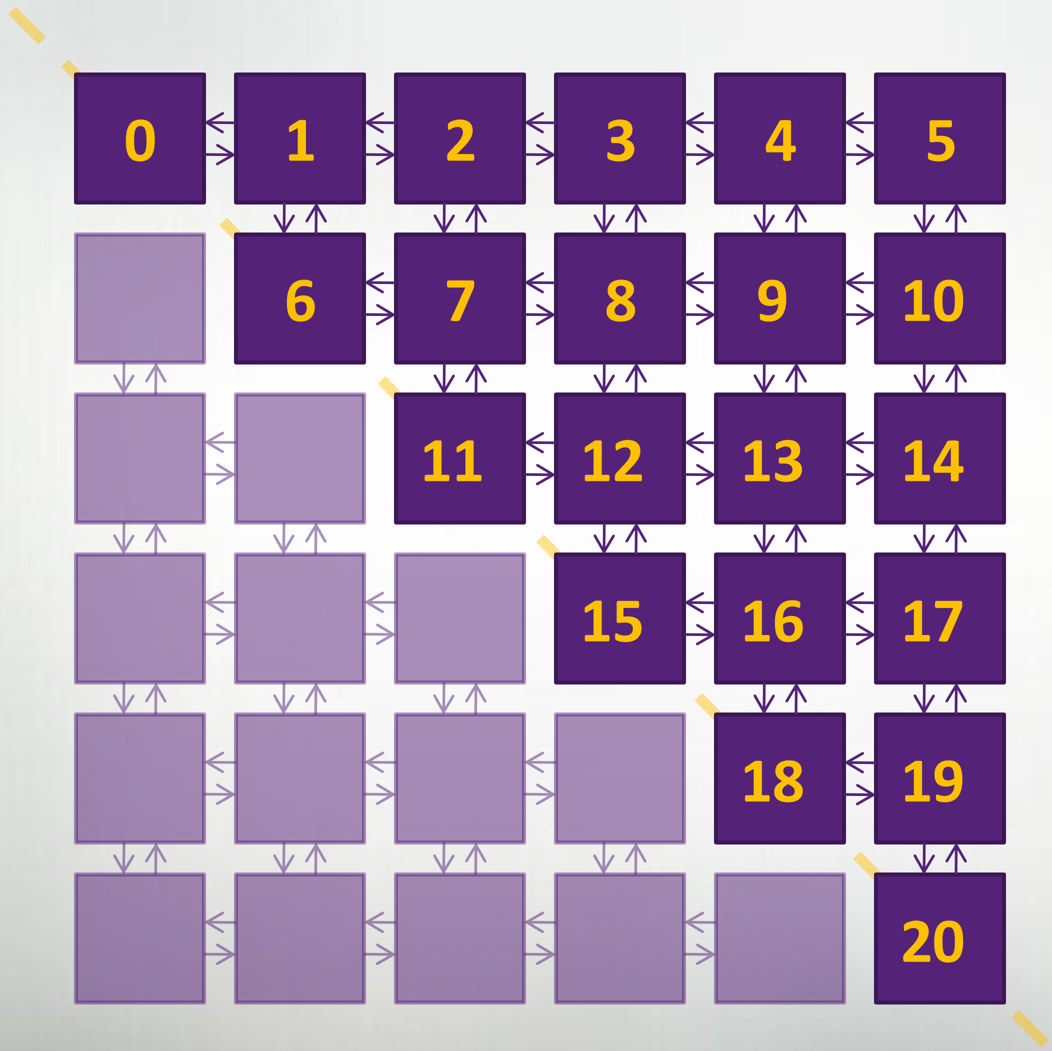
Hypothesis

- As apps show good weak scaling to ever larger numbers of processors the proportion of time spent writing results will increase.
- It's not always necessary for applications to complete writing before continuing computation if the data is cached in memory
  - Therefore I/O can be overlapped with computation
- This I/O could be performed by only a fraction of the processors used for computation and still achieve good I/O bandwidth.

HELIUM

- Developed by Prof K. Taylor and team at Queen’s University, Belfast
- Solves the Time Dependent Schrödinger Equation for two electrons in a Helium atom interacting with a laser pulse.
- Parallelised using domain decomposition and MPI
- Very computationally intensive, uses high order methods to integrate PDEs
- Larger problems result in larger checkpoints
- I/O component is being optimised as part of a Cray Centre of Excellence for HECToR project.
  - Preparing the code for the next generation machine

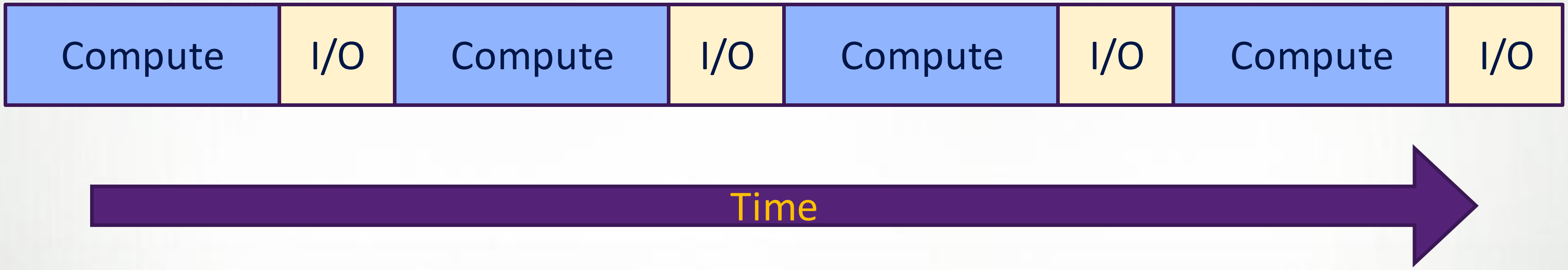
HELIUM Decomposition



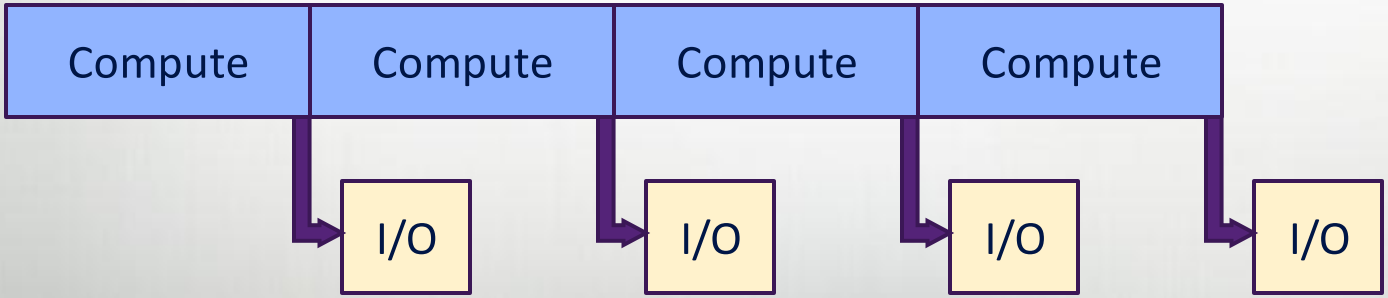
- Upper-triangular domain decomposition
- Does not fit HDF5 or MPI-IO models cleanly
- Regular Checkpoints
  - File per process I/O
  - 50 MB per file
  - Scientific data extracted from checkpoint data

Asynchronous I/O

Standard Sequential I/O



Asynchronous I/O



Naive MPI Pseudo Code

Compute Node

```
do i=1,time_steps
   compute(j)
   checkpoint(data)
end do

subroutine checkpoint(data)
   MPI_Wait(send_req)
   buffer = data
   MPI_Isend(IO_SERVER, buffer)
end subroutine
```

I/O Server

```
do i=1,time_steps
   do j=1,compute_nodes
      MPI_Recv(j, buffer)
      write(buffer)
   end do
end do
```

Enforces the order of processing ... sequential

Less Naive MPI Pseudo Code

Compute Node

```
do i=1,time_steps
   compute(j)
   checkpoint(data)
end do

subroutine checkpoint(data)
   MPI_Wait(send_req)
   buffer = data
   MPI_Isend(IO_SERVER, buffer)
end subroutine
```

I/O Server

```
do i=1,time_steps
   do j=1,compute_nodes
    MPI_Irecv(j,buffer(j),req(j))
   end do
   do j=1,compute_nodes
     MPI_Waitany(req, j, buffer)
     write(buffer(j))
   end do
end do
```

Requires a lot more buffer space... Receives in any order

Everyone sends at once

- Many compute nodes per I/O Server
- All compute nodes transmitting (almost) simultaneously
- Potentially too many incoming messages or pre-posted receive messages
- Overloads the I/O server

MPI Pseudo Code

Compute Node

```
do i=1,time_steps
   compute()
   send_io_data()
   checkpoint()
end do

subroutine send_io_data()
   if(data_to_send) then
      MPI_Test(pinged)
      if(pinged) then
         MPI_Isend(buffer, req)
         data_to_send = .false.
      end if
   end if
end subroutine

subroutine checkpoint(data)
   send_io_data()
   MPI_Wait(req)
   buffer = data ! Cache data
   data_to_send = .true.
end subroutine
```

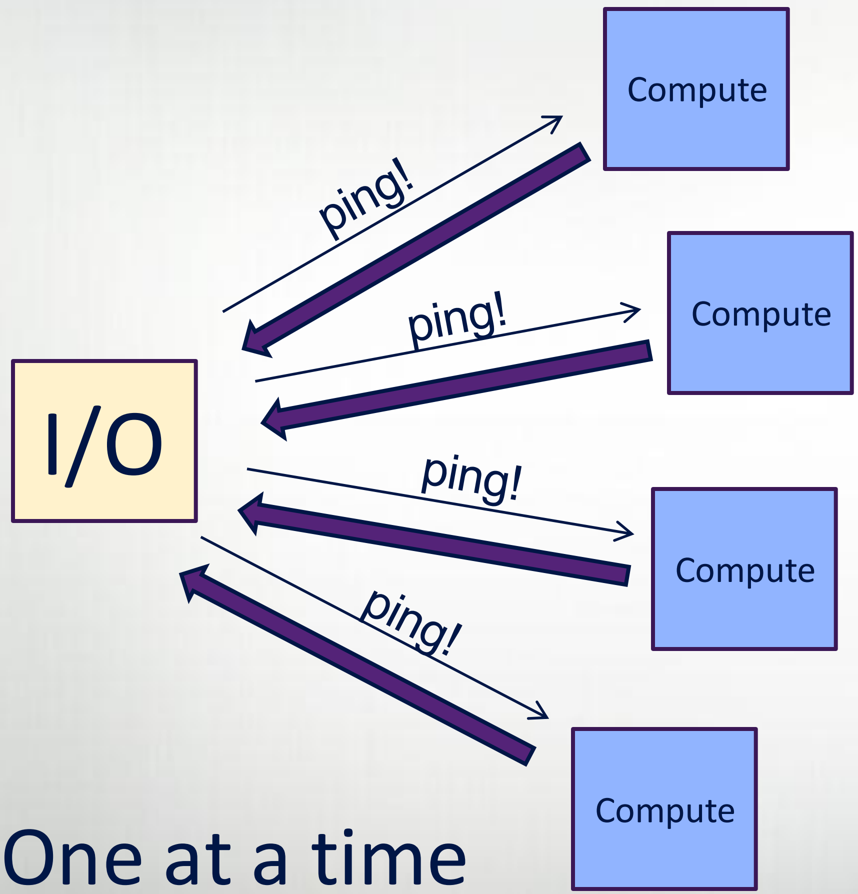
I/O Server

```
do i=1,time_steps
   do j=1,compute_nodes
      MPI_Send(j)         ! Ping
      MPI_Recv(j, buffer)
      write(buffer)
   end do
end do
```

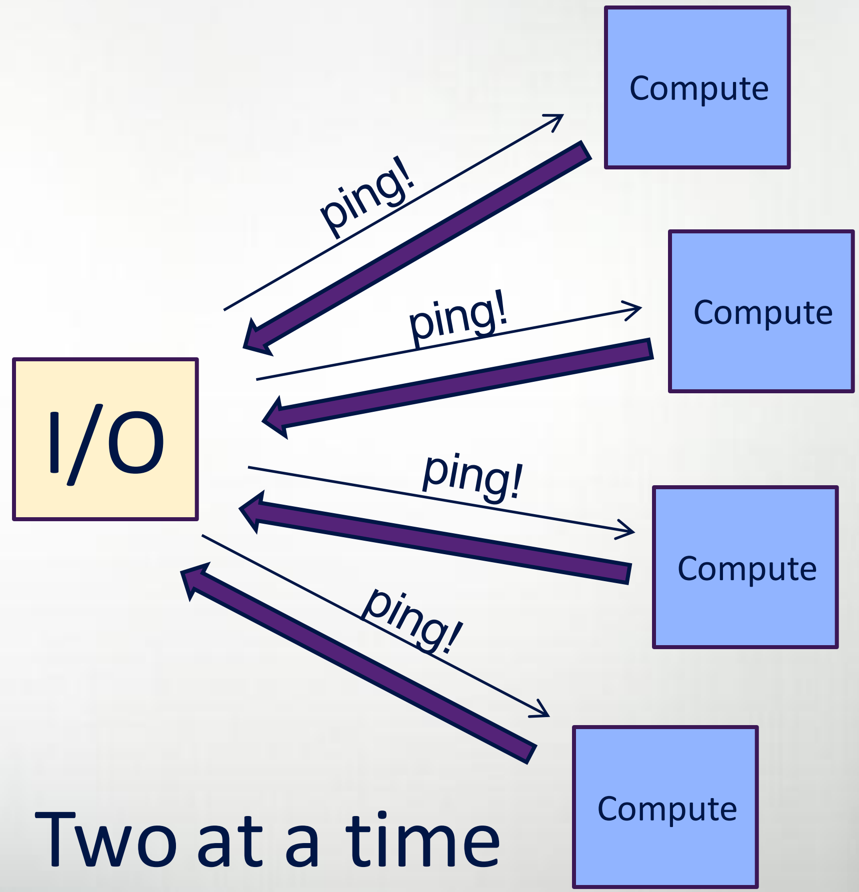
Enforces the order of processing ... Sequential but only one message to the server at a time

Subroutine called so infrequently that data rarely sent

Stemming the flood



One at a time



Two at a time

Interrupt Driven MPI Pseudo Code

Compute Node

```
do i=1,time_steps
  do j=1,sections
    compute_section(j)
    send_io_data()
  end do
  checkpoint()
end do

subroutine send_io_data()
  if(data_to_send) then
    MPI_Test(pinged)
    if(pinged) then
      MPI_Isend(buffer, req)
      data_to_send = .false.
    end if
  end if
end subroutine

subroutine checkpoint(data)
  send_io_data()
  MPI_Wait(req)
  buffer = data ! Cache data
  data_to_send = .true.
end subroutine
```

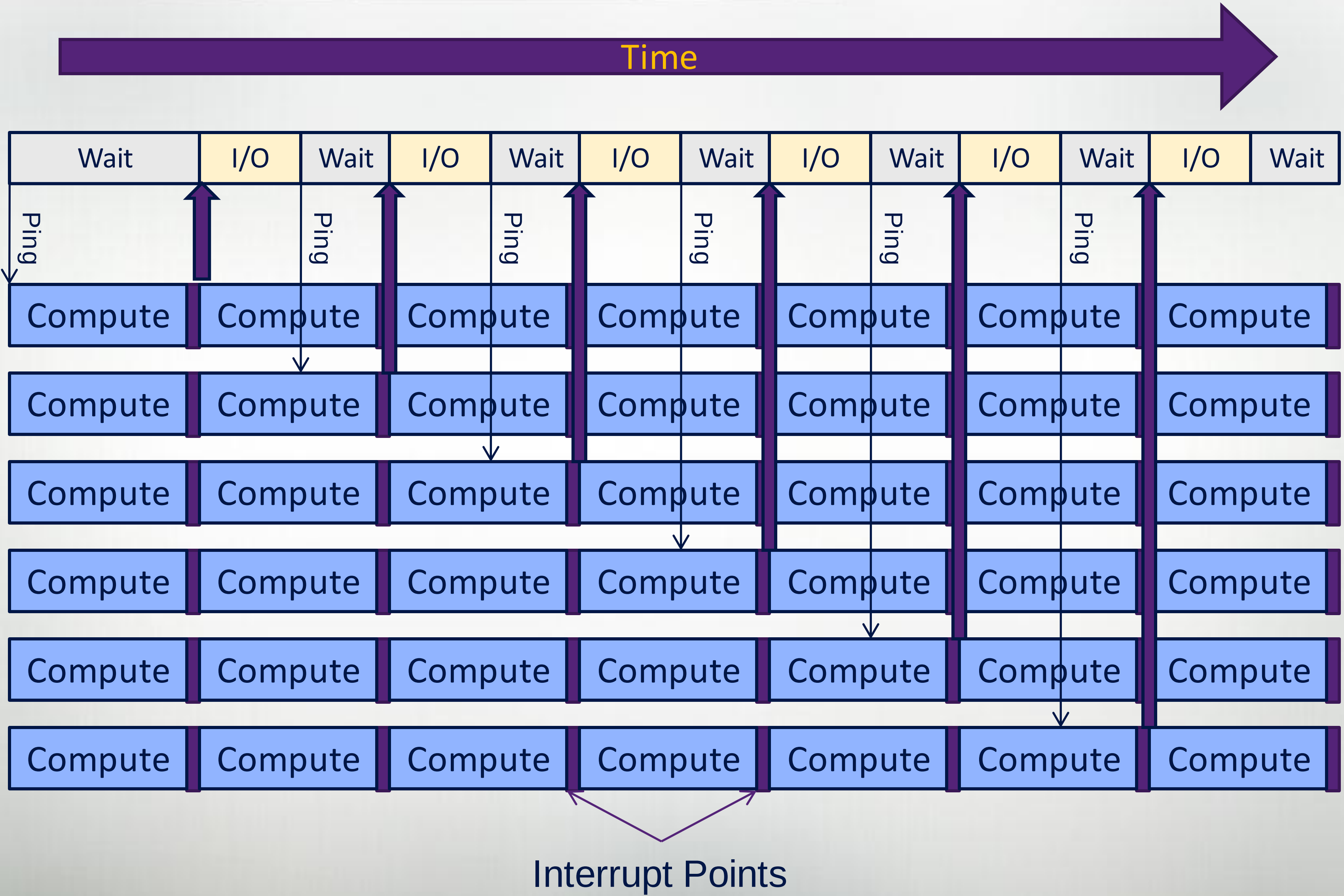
I/O Server

```
do i=1,time_steps
  do j=1,compute_nodes
    MPI_Send(j)        ! Ping
    MPI_Recv(j, buffer)
    write(buffer)
  end do
end do
```

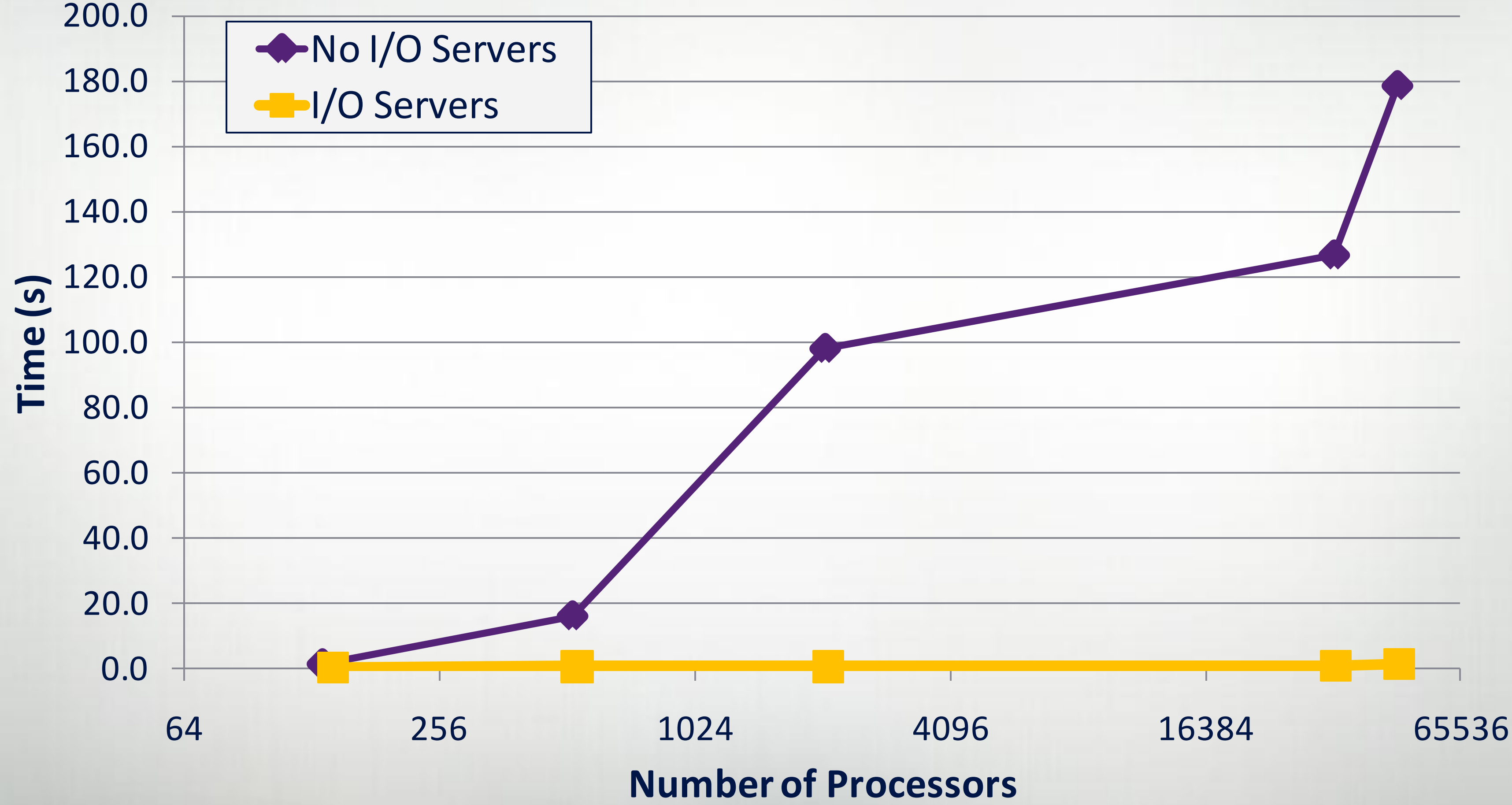
Now called more frequently so greater chance of success

The greater the frequency of calls the more efficient the transfer, but the higher the load on the system

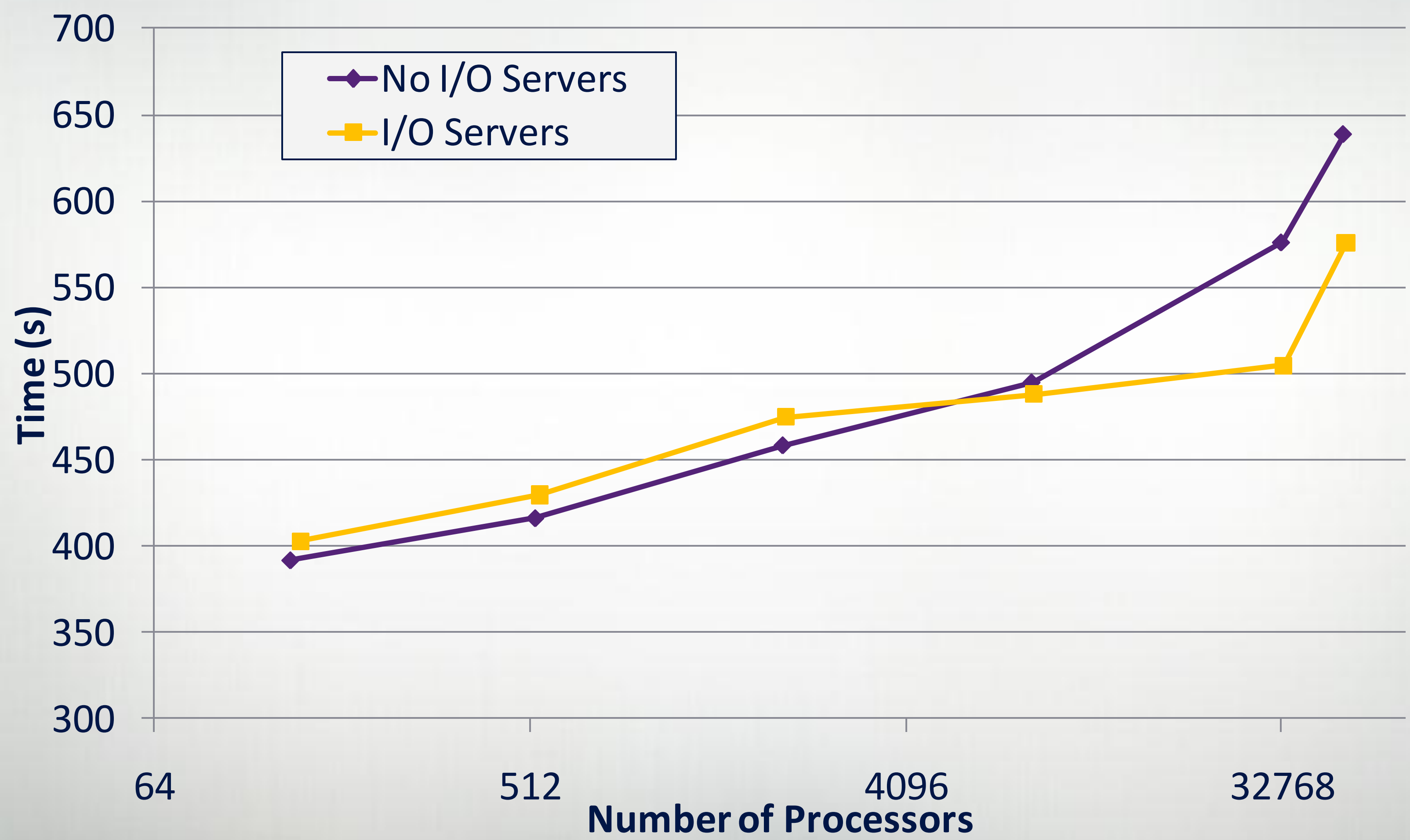
Interrupted Computation



Wall Clock Time For Checkpoint



Total Wall clock Time Between Checkpoints



Single Sided Communication

- Using MPI , messages have to be sent from the compute nodes to the I/O Server
  - To prevent overloading the I/O Server the compute nodes have to actively check for permission to send messages.
- It is simpler to have the I/O Server pull the data from the compute nodes when it is ready
  - SHMEM is a single sided communications API supported on Cray systems
  - SHMEM supports remote push and remote pull of distributed data over the network
  - It Can be directly integrated with MPI on Cray Architectures

SHMEM Pseudo Code

Compute Node

```
do i=1,time_steps
   compute()
   checkpoint()
end do

subroutine checkpoint(data)
   if(.not. CP_DONE) then
      wait_until(flag, CP_DONE)
   end if
   buffer = data ! Cache data
   flag = DATA_READY
end subroutine
```

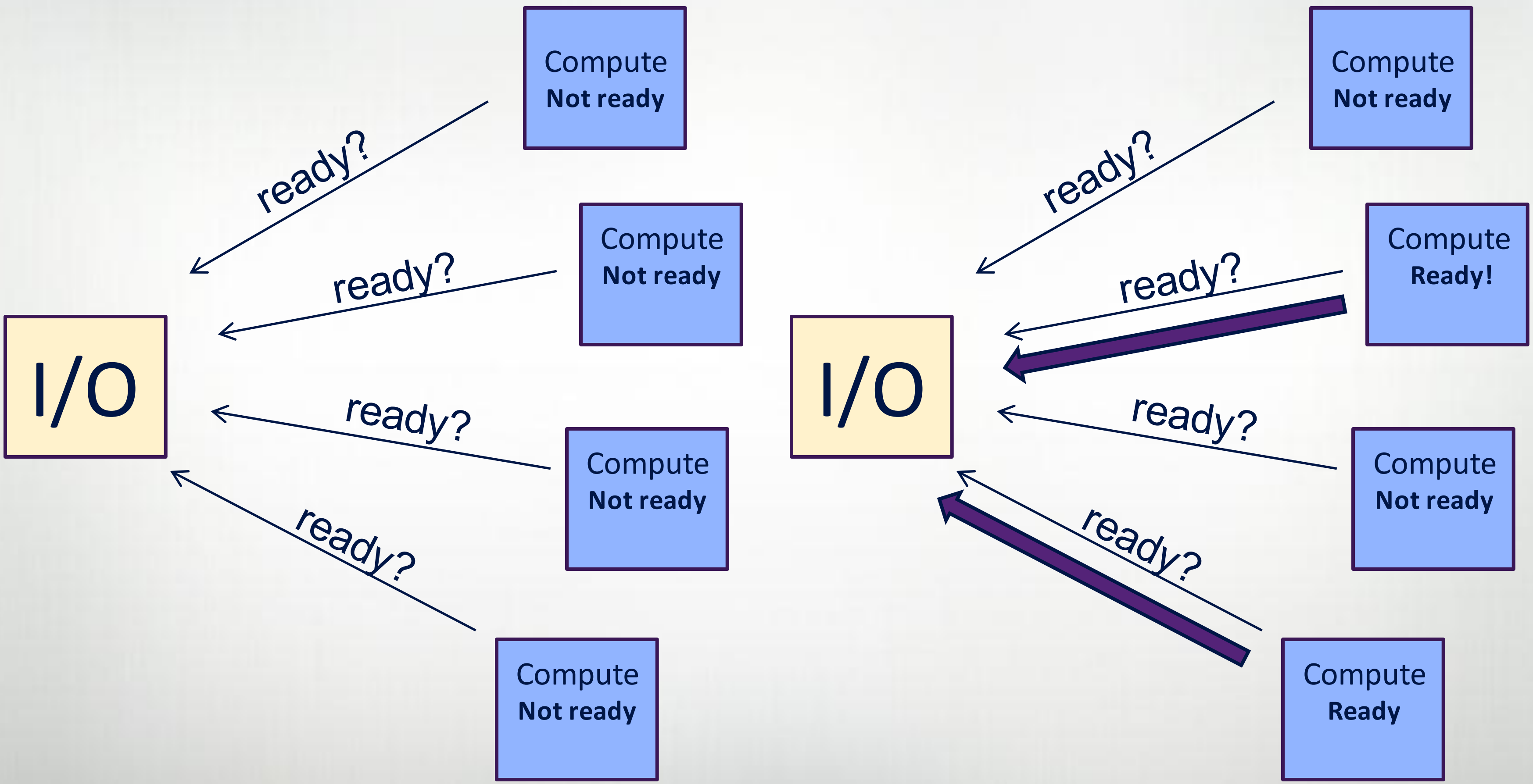
I/O Server

```
do
   do j=1,compute_nodes
      get(j, local_flag)
      if(local_flag = DATA_READY)
         get(j, buffer)
         write(buffer)
         put(j, flag, CP_DONE)
      end if
   end do
end
```

- Compute node code becomes much simpler...
- No requirement to explicitly send data
- Polling interrupt done by the system libraries

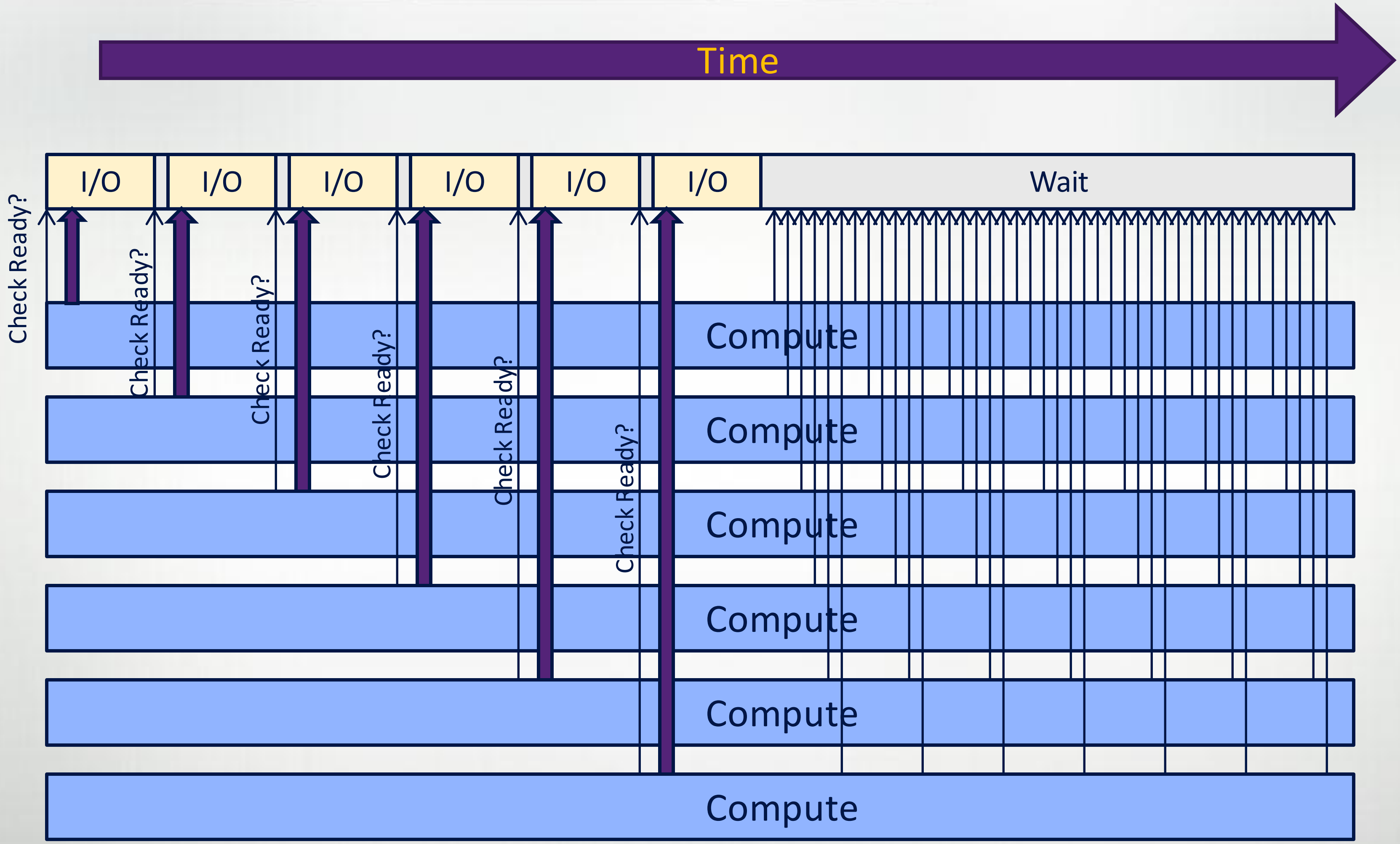
- I/O Server slightly more complicated.
- Constantly polling the compute nodes.
- Only one message at a time

Pulling data from the computes



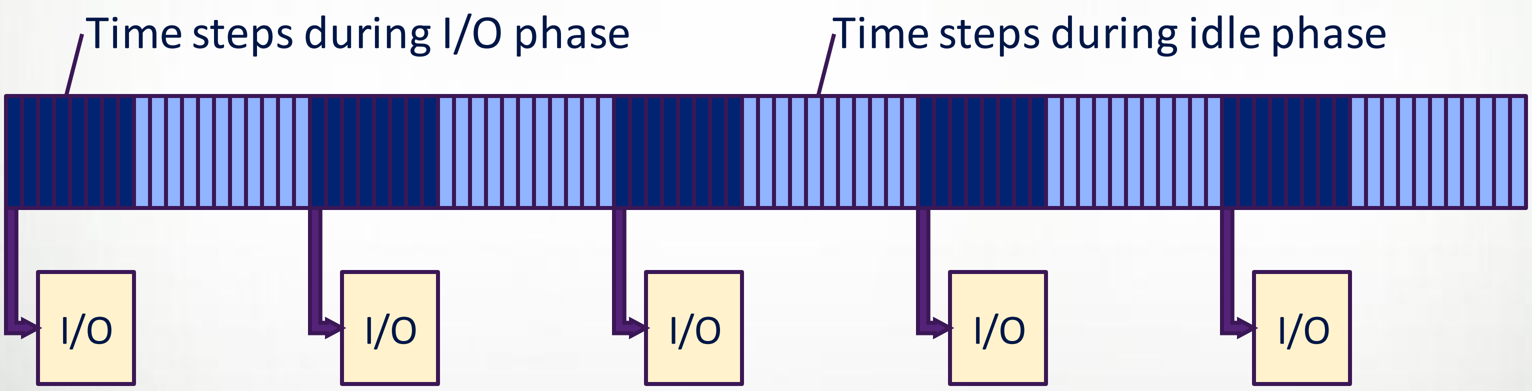
Polling computes

SHMEM Implementation



I/O Overhead

- I/O Servers introduce additional communication to the application.
  - Does this additional load affect the application's overall performance ?

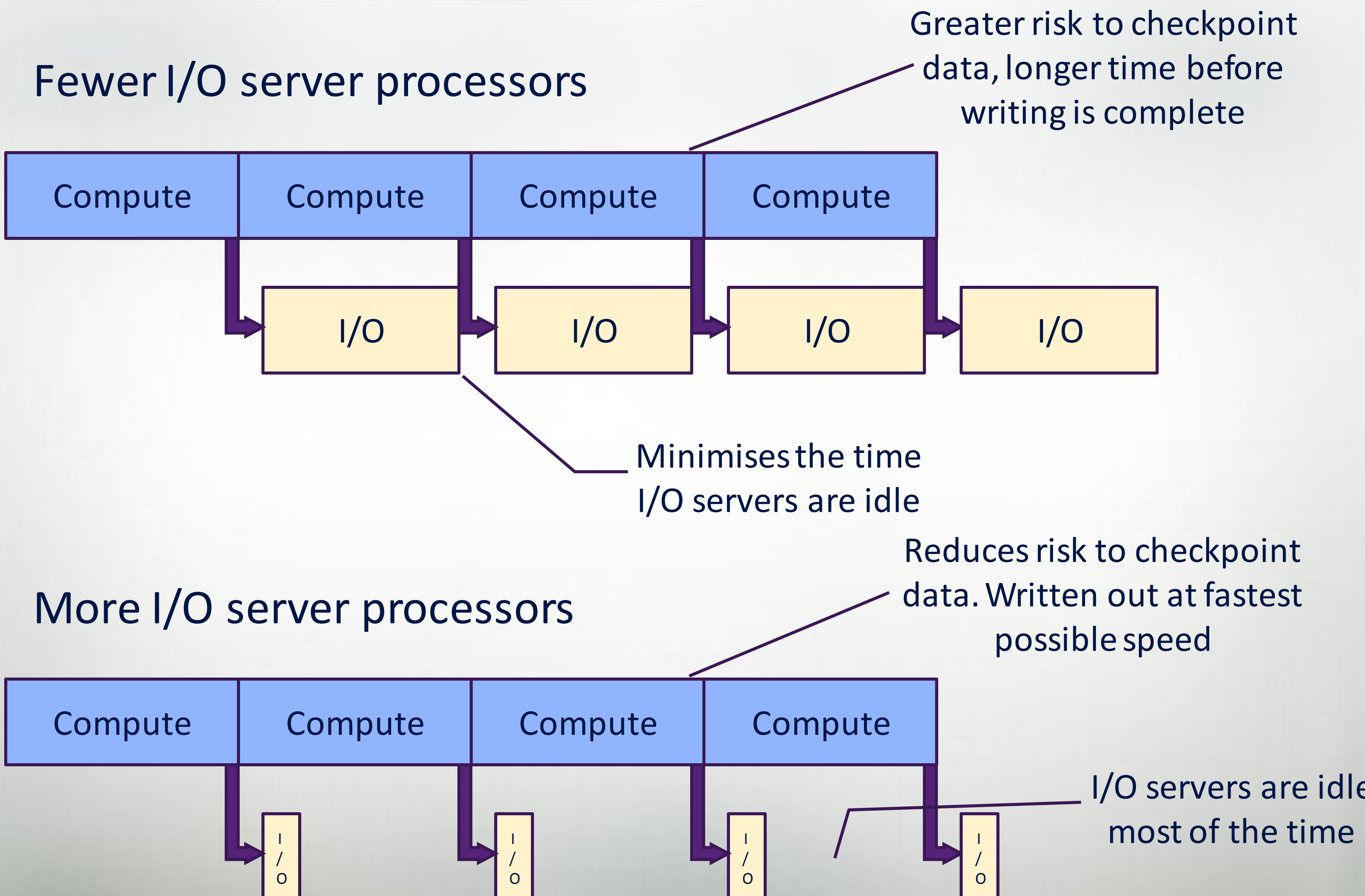


Tests measured the wall clock time to complete standard model time steps during I/O communications and during I/O idle time

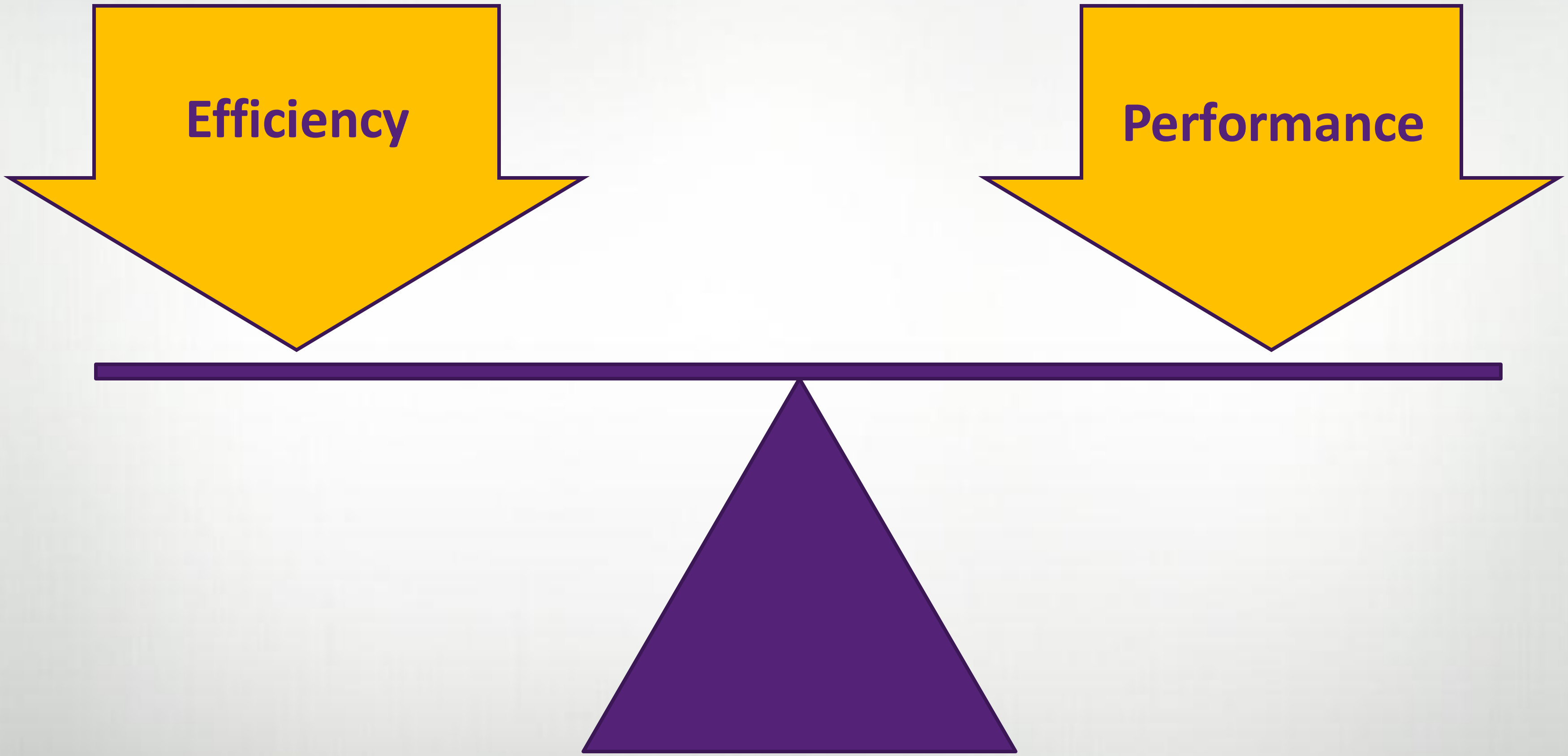
SHMEM vs MPI

- An average Time step took 9.31s with MPI, 9.72s with SHMEM
- 86% of Time steps were during idle time using MPI, 75% with SHMEM.
- Using MPI, time steps during the I/O phase cost 2.33% more, with SHMEM 0.19%.

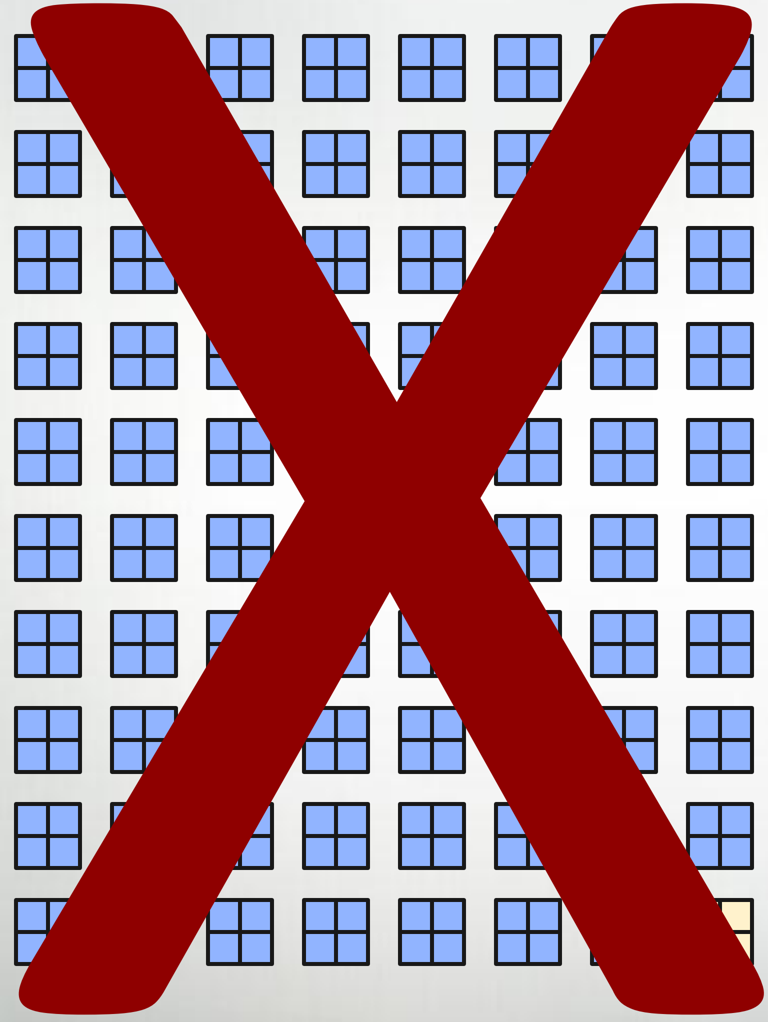
Selecting number of processors

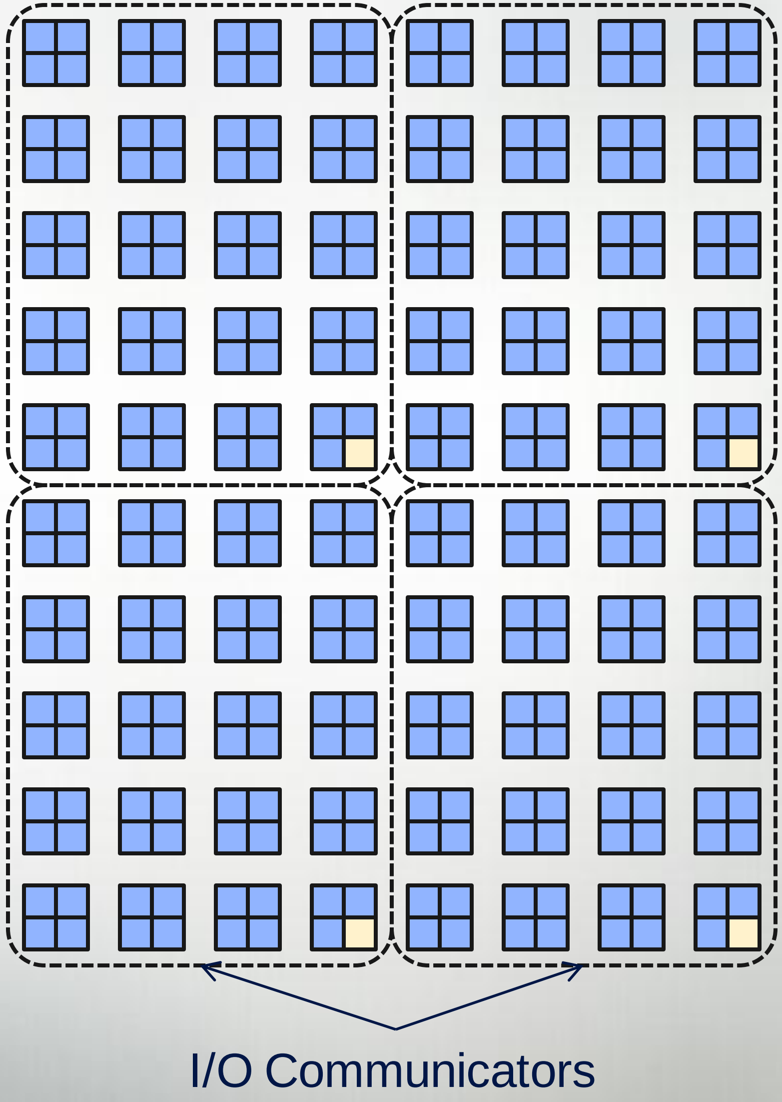


Finding the Balance

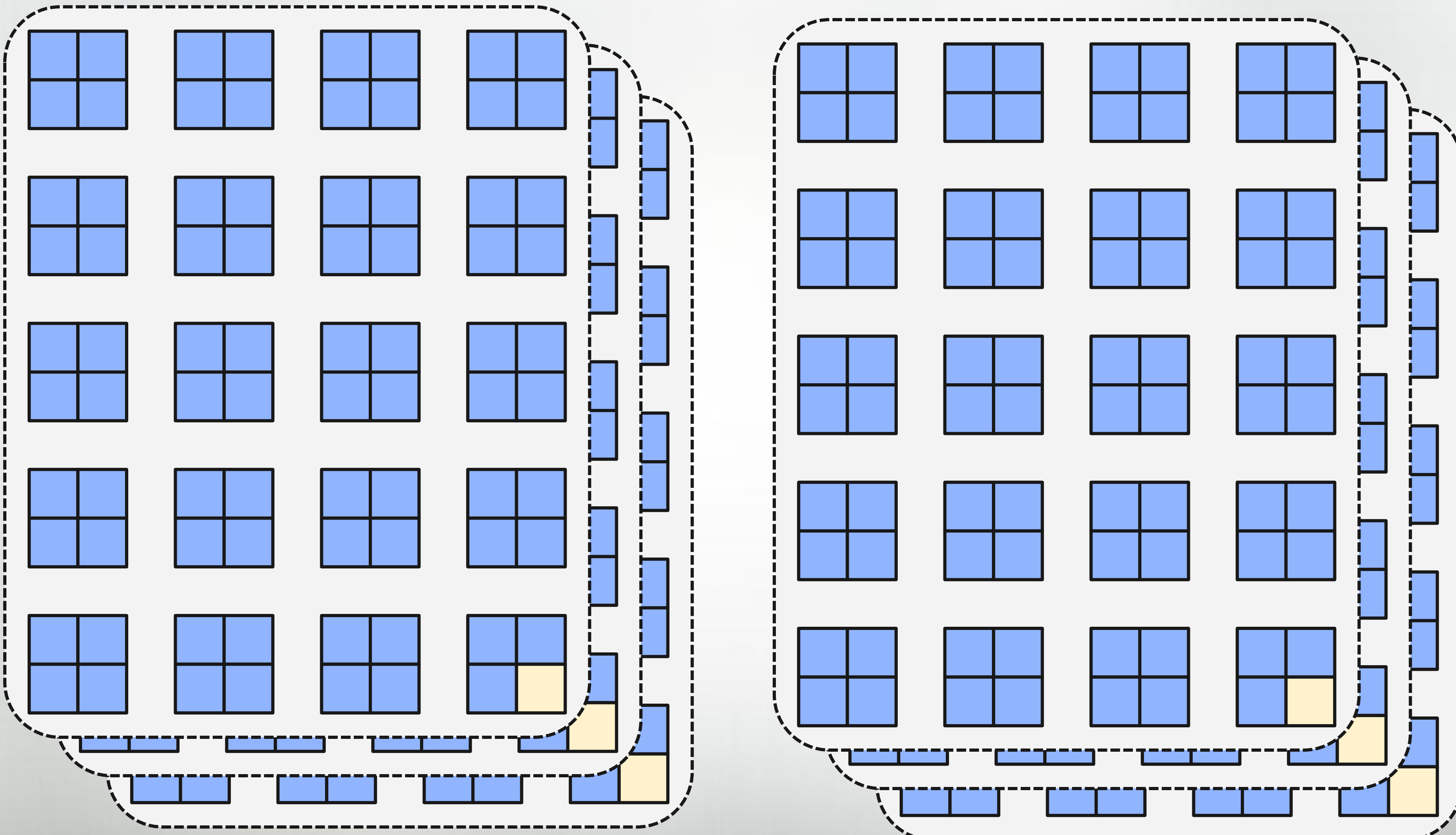


Placement of I/O Servers





I/O Communicators

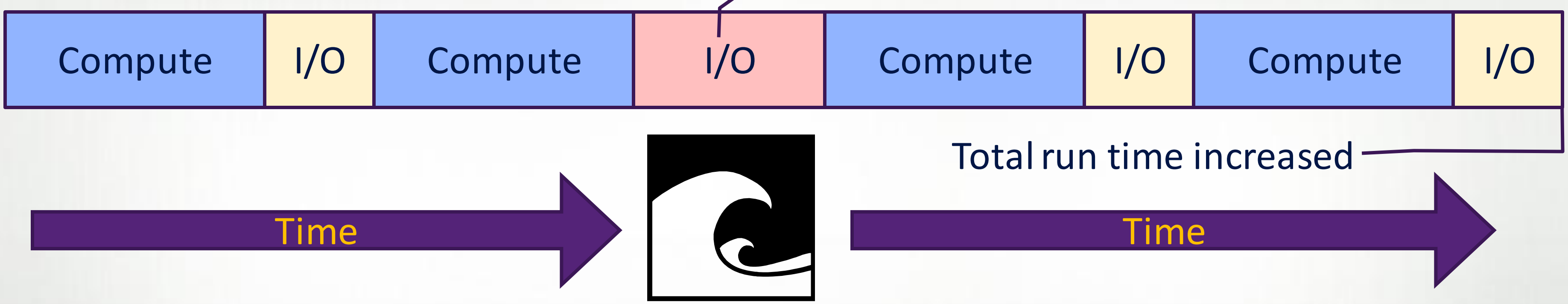


Smoothing out I/O Performance

Bandwidth is shared between jobs on the system

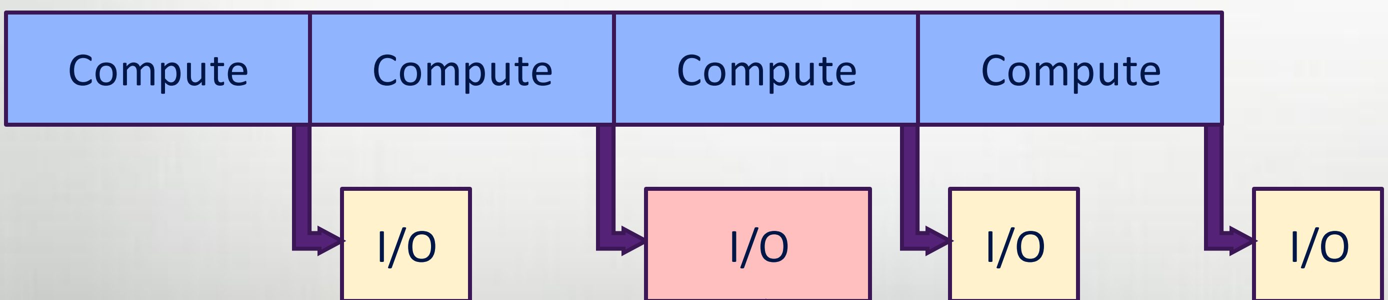
Another application on the system writes out a checkpoint at the same time. Effective application I/O bandwidth halved. Write time doubles

Standard Sequential I/O



Total run time increased

Asynchronous I/O



Same event, constant total run time

Post Processing

- I/O Server idle time could be put to good use
  - Performing post-processing on data structures
    - Averages, sums.
    - Restructuring data (transposes etc)
    - Repacking data (to HDF5, NetCDF etc)
    - Compression (RLE, Block sort)
  - Aggregating information between multiple jobs
    - Collecting information from multiple jobs and performing calculations
  - Ideally large numbers of small tasks
    - Short jobs that can be scheduled between I/O operations
    - Serial processes, or parallel tasks over the I/O servers
  - I/O Servers could become multi-threaded to increase responsiveness

Conclusions

- Writing data to disk can become a significant proportion of runtime with weak scaling applications
- Asynchronous I/O offers a way for a set of applications to hide I/O time.
- It also makes application runtime less dependent upon the available I/O bandwidth
- I/O Servers are a way of implementing asynchronous I/O using MPI or SHMEM constructs. They also provide additional opportunities for post processing.
- SHMEM offers a nicer programming model for implementation but requires further work. Should perform well on Gemini.

Acknowledgements

- Kevin Roy, Cray Centre of Excellence for HECToR
- Prof K. Taylor and the HELIUM development team at Queen's University Belfast
- Some results obtained on Jaguar-PF with approval from Oak Ridge National Laboratory Leadership Computing Division

CRAY

THE SUPERCOMPUTER COMPANY