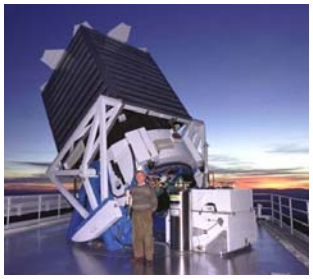
Automatic Parallelization for Scripting Languages: Toward Transparent Desktop Parallel Computing

Xiaosong Ma[1,2]
Jiangtian Li[1]
Nagiza Samatova[2]

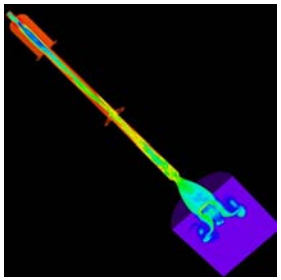
1. North Carolina State University
2. Oak Ridge National Laboratory

NSF-NGS Workshop, March 2007

Problem Statement


- More scientific data generated!

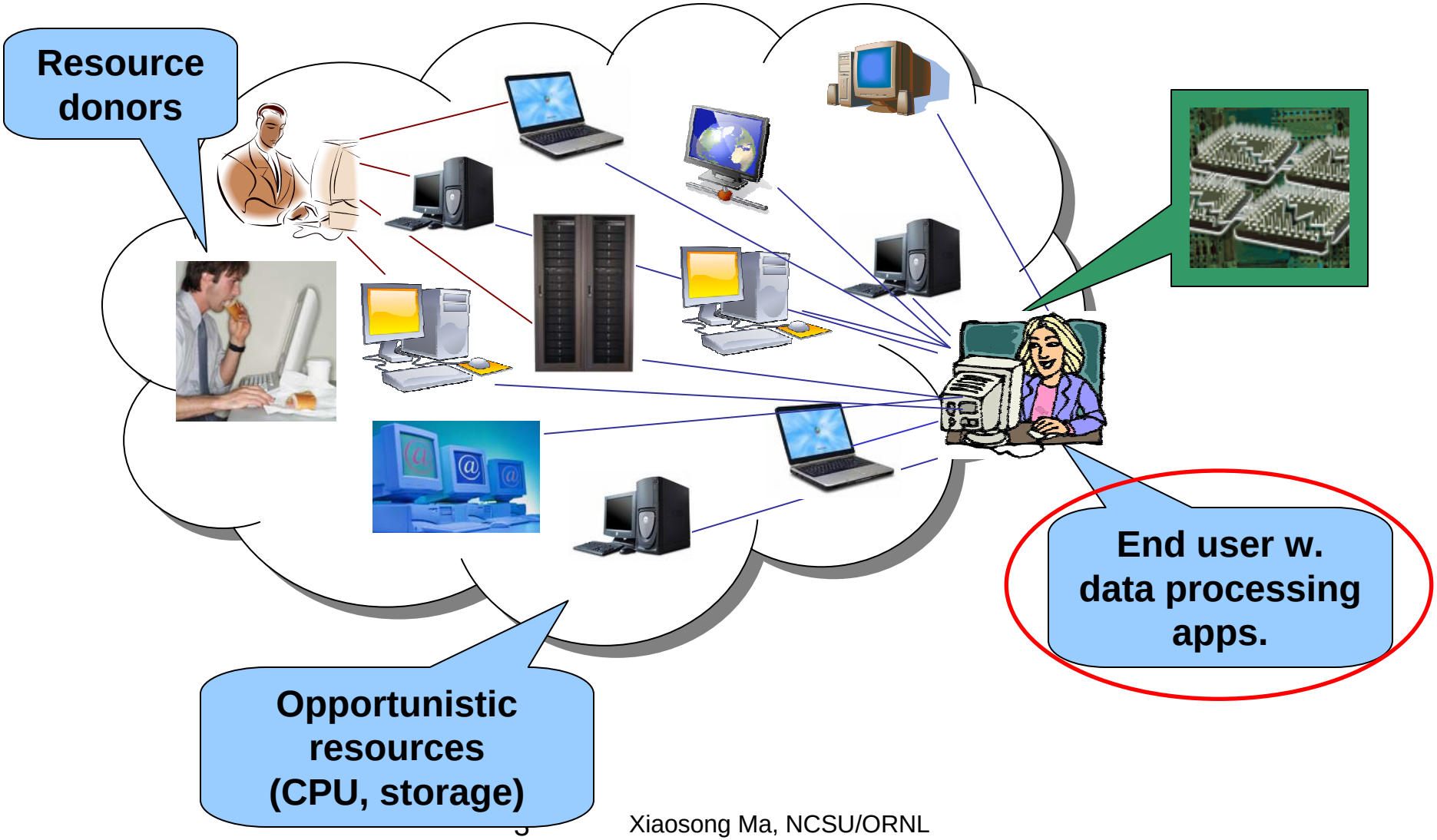


- R&D focused on **production** side
  - High-performance I/O, data repository, high-speed transfer

- Data **consumption** bottleneck-prone
  - Requires interactiveness and high performance
  - Abundant resources but parallel processing remains challenging
    - Data end users are domain scientists
    - Distributed, non-dedicated resources

Mission Statement

- **Transparent, interactive parallel data processing**



Automatic Parallelization

- Scientists use ready-made tools
  - Data analysis/mining, visualization, feature extraction
- Heavy use of scripting languages (e.g., Matlab)
  - Powerful functions
  - Interactive data processing
  - Computation- and data-intensive
- Can we automatically parallelize these scripts?

- Initial step: ***automatic and transparent parallel R***
- R: scripting language and environment for data processing
  - Open-source, portable
  - Powerful statistics functions
  - Widely used in many science domains
- **Goal:** transparent parallel execution of **sequential** R code

Parallelizing Scripting Languages

- People have been trying hard
  - 27+ projects in parallelizing Matlab [Choy05]
  - 5 categories in approach
    - Embarrassingly parallel — Snow package
    - Message passing — RMPI
    - Shared memory
    - Back-end support — RScaLAPACK
    - Compilers
  - **Problems:** code modification required, portability, limited types of parallelism

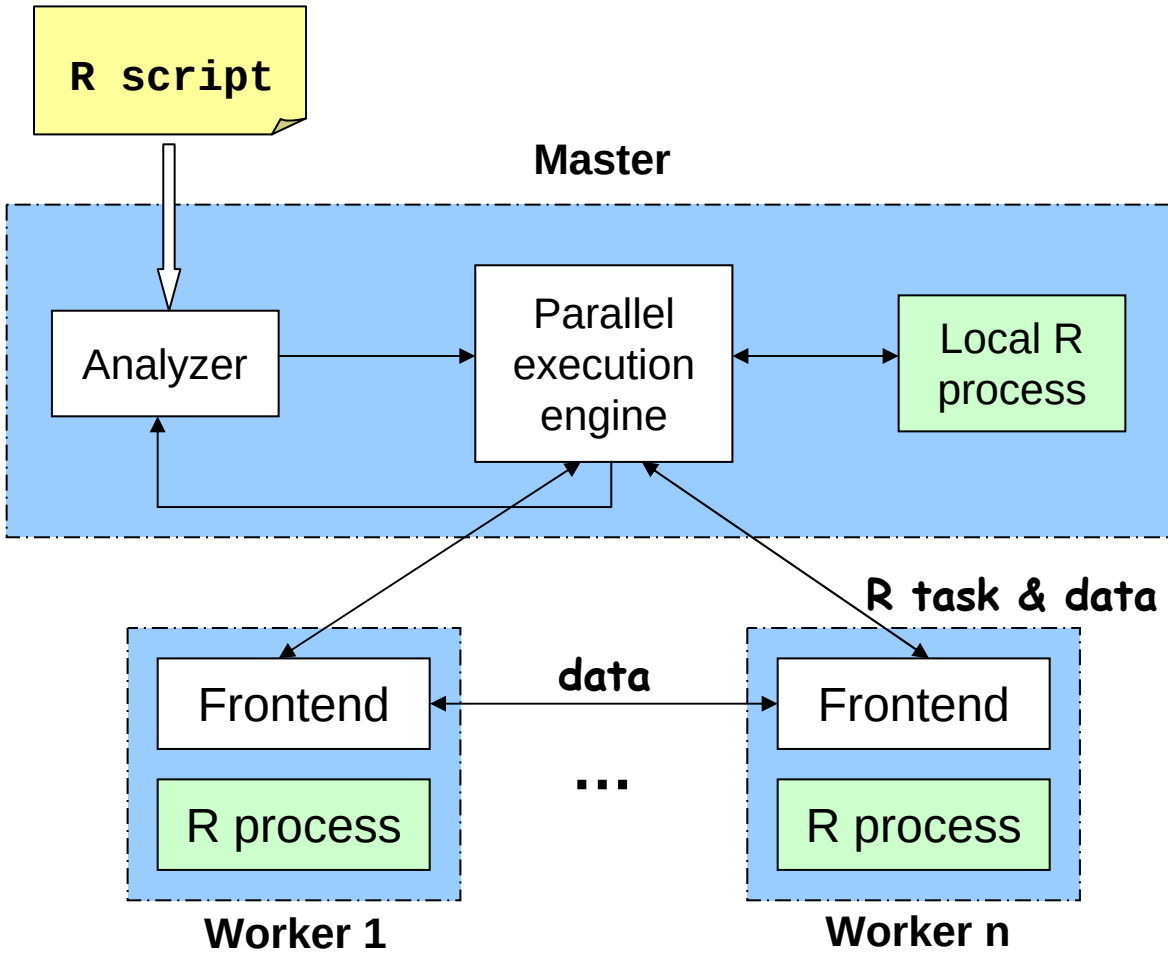
- **Our contribution**: ***pR*** framework
  - Automatically parallelizes sequential R scripts
    - Runtime, full-program code analysis
  - Exploits both **task** and **data** parallelism
  - Portable parallel R environment
  - Techniques applicable to other languages

pR Design Rationale

- Key observations
  - R codes consist of high-level pre-built functions
    - *svd*, *eigen*, *hist*
    - Supported by mature numerical packages
  - Loops tend to be independent, w. higher per-iteration execution cost
  - Task parallelism important

- Leveraging parallelizing compiler technology
  - Easier job: no pointers, functional language, tricky index unlikely
  - Steps beyond
    - Not limited to loops,
    - Dynamic analysis

pR Architecture



Dependence analysis

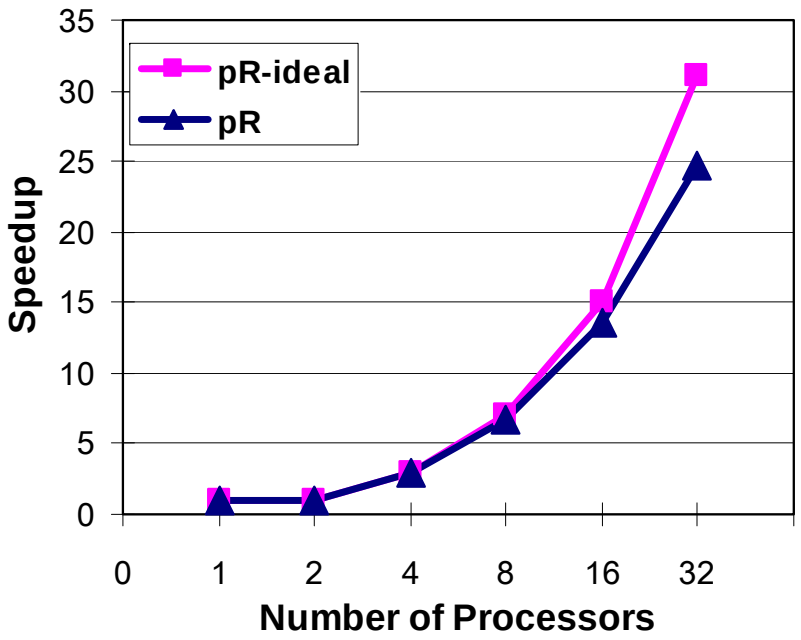
- Statement dependence analysis
- Loop dependence analysis
  - GCD test [Banerjee93]
  - Partition loop if no dependence discovered
  - Adjust task precedence graph
- I/O operation dependence analysis
  - Coarse-granule, file based
  - Obtain file information w. system calls
- Incremental analysis
  - Pause points inserted when evaluation results required

Performance

- Testbed
  - Opt cluster: 16 nodes, 2 core, dual Opteron 265, 1 Gbps Ether
  - Fedora Core 5 Linux x86_64(Linux Kernel 2.6.16)

- Benchmarks
  - Boost (real-world application)
  - Bootstrap (computation-intensive)
  - SVD (computation- and data-intensive)
  - Task parallel benchmark

Results from Boost



	2	4	8	16	32
Initialization	**0.05%**	**0.13%**	**0.31%**	**0.65%**	**1.28%**
Analysis	**0.00%**	**0.00%**	**0.00%**	**0.01%**	**0.04%**
Master MPI	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.01%**
Max wkr serial.	**0.42%**	**0.69%**	**1.15%**	**2.05%**	**3.19%**
Max wkr MPI	**0.00%**	**0.03%**	**0.07%**	**0.15%**	**0.26%**
Max wkr socket	**0.01%**	**0.01%**	**0.02%**	**0.04%**	**0.05%**

- Analysis/scheduling overhead very small
- Close-to-ideal speedup

Automatic Parallelization: Summary and Next Step

- First step towards transparent parallel data processing
  - No code modification required
  - Exploit both task and data parallelism

- Next step
  - Port to desktop environment
    - Interactive execution
  - Combine with backend parallelization [JPhysics06]

References

- [JPhysics06] N. Samatova et al., High performance statistical computing with parallel R: applications to biology and climate modelling, *Journal of Physics: Conference Series*, Volume 46, 2006.
- [Banerjee03] U. Banerjee, R. Eigenmann, A. Nicolau and D. Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2), 1993.
- [Choy05] R. Choy and A. Edelman. Parallel Matlab: Doing It Right. *Proceedings of the IEEE*, 93(2), 2005.

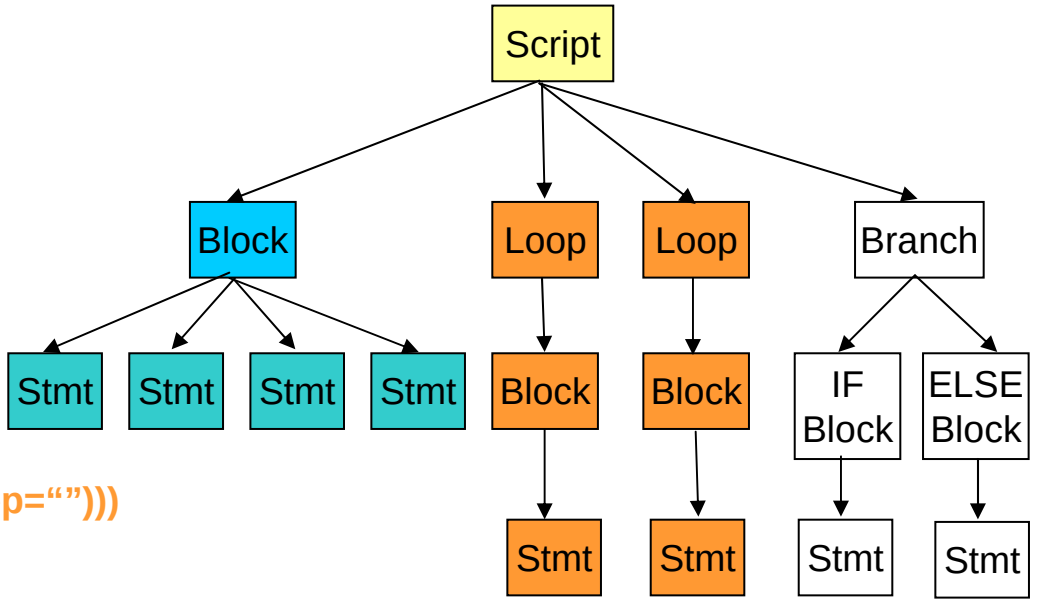
Thank you!

Parallel Execution Engine

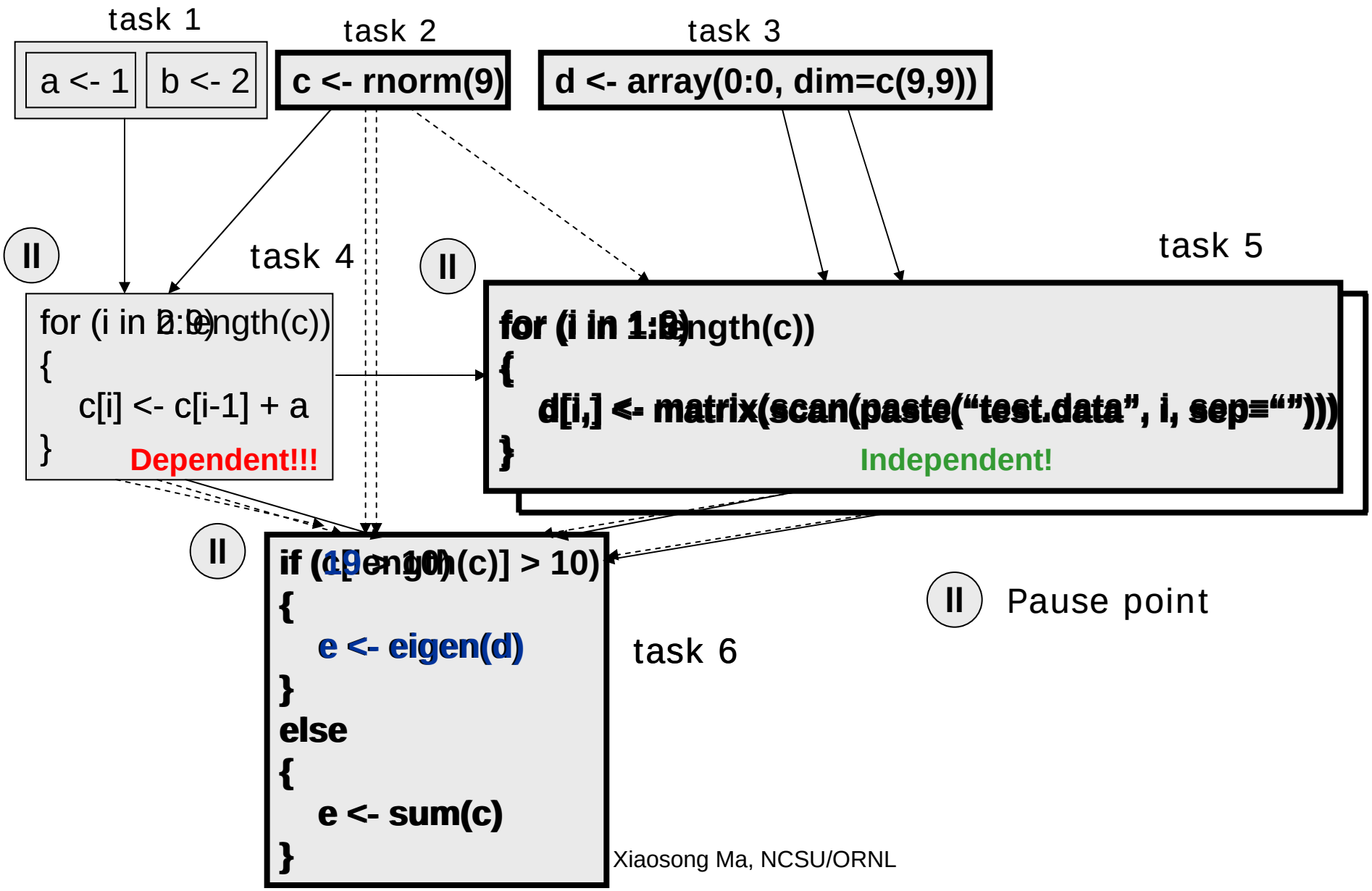
- Dispatches ready tasks
- Updates analyzer with runtime results
- Coordinates peer-to-peer data communication and monitor execution status
  - Worker front-end manages communication
    - Intermediate results shipped to other nodes without interrupting R computation

pR Sample Script

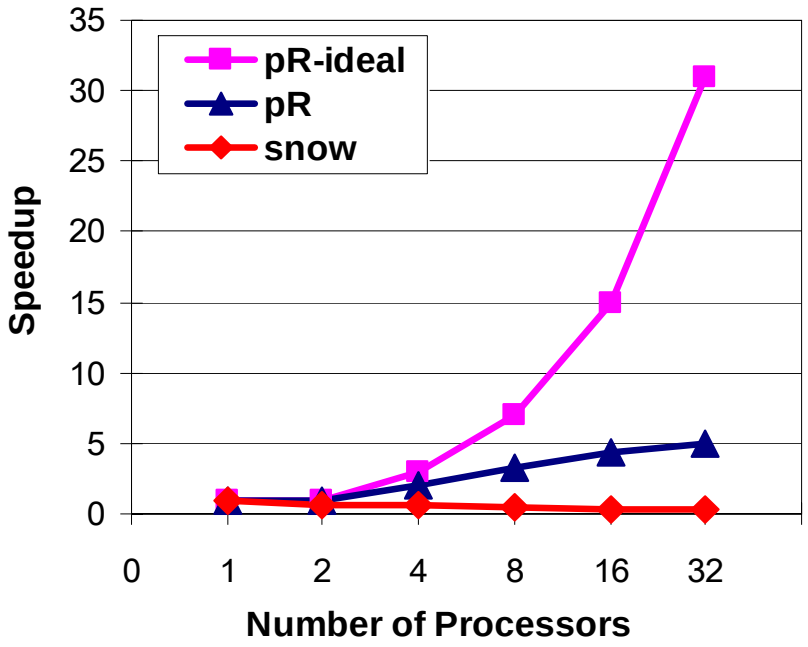
```
a <- 1
b <- 2
c <- rnorm(9)
d <- array(0:0, dim=c(9,9))

for (i in b:length(c))
{
   c[i] <- c[i-1] + a
}

for (i in 1:length(c))
{
   d[i,] <- matrix(scan(paste("test.data", i, sep="")))
}

if (c[length(c)] > 10)
{
   e <- eigen(d)
}
else
{
   e <- sum(c)
}
```



Example: Runtime Analysis



SVD

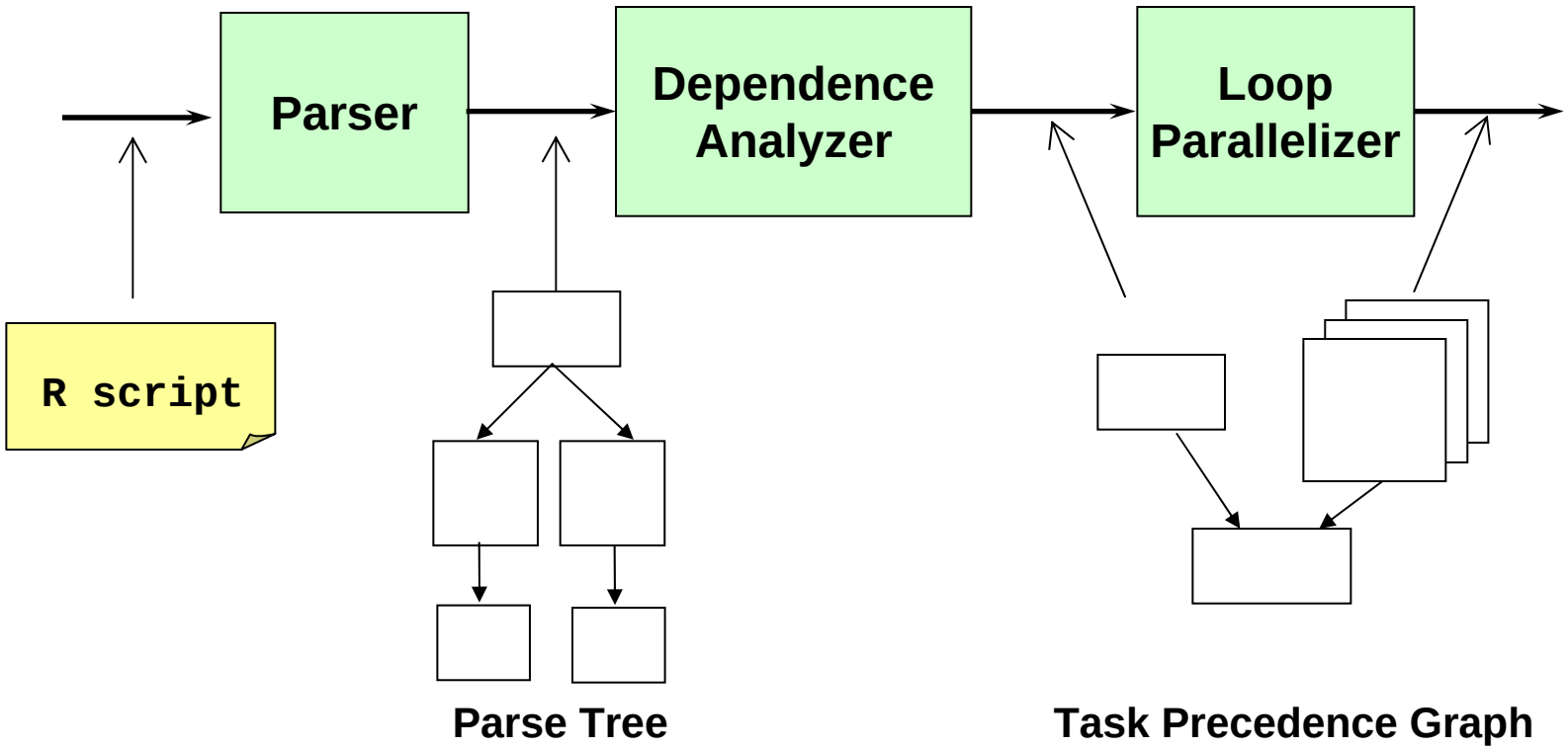


	2	4	8	16	32
Initialization	0.23%	0.49%	0.78%	1.12%	1.27%
Analysis	0.00%	0.00%	0.00%	0.01%	0.02%
Master MPI	0.00%	0.00%	0.00%	0.00%	0.01%
Max wkr serial.	11.7%	26.5%	41.7%	53.0%	58.0%
Max wkr MPI	0.00%	2.10%	4.32%	6.44%	7.83%
Max wkr socket	1.45%	1.56%	1.99%	2.40%	2.51%

- Serialization large dataset in R causes major overhead
  - 1.9 MB/s
- Order of magnitude better than **snow package**

pR Analyzer

- Input
  - R script
- Output – Task Precedence Graph
  - Broken down to R tasks



Ease of use demonstration

- Comparison of pR and snow (an R add-on package)
- pR – no user interference of source code
- snow – user plugs in APIs

```
a <- matrix(1:1000, 100, 10)
b <- list()
c <- mean(a)
d <- sum(a)
for (i in 1:dim(a)[1])
{
  b[i] <- sum(a[i,])
}
```

```
library(Rmpi)
library(snow)

cl <- makeCluster(2, type = "MPI")
a <- matrix(1:1000, 100, 10)
b <-list()
c <- mean(a)
d <- sum(a)
b <- parApply(cl, a, 1, sum)
stopCluster(cl)
```

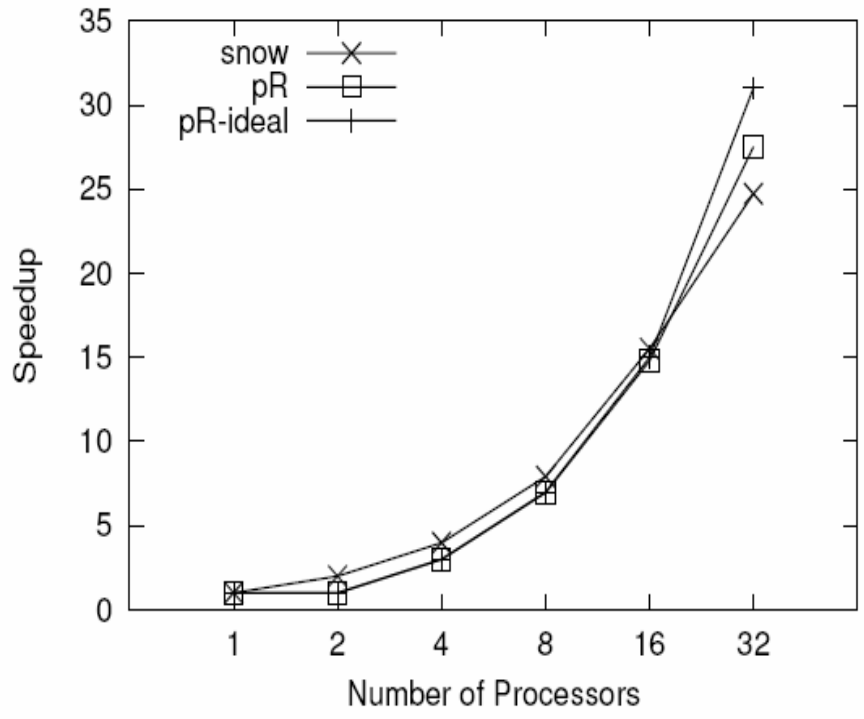
Related Work

- Embarrassingly parallel
  - snow package - *Rossini et al.*
- Message passing
  - MultiMATLAB - *Trefethen et al.*
  - pyMPI - *Miller*
- Back-end support
  - RScaLAPACK - *Yoginath et al.*
  - Star-P - *Choy et al.*
- Compilers
  - Otter - *Quinn et al.*
- Shared memory
  - MATmarks – *Almasi et al.*

Related Work

- Parallelizing compilers
  - SUIF – *Hall et al.*
  - Polaris - *Blume et al.*
- Runtime parallelization
  - Jprm - *Chen et al.*
- Dynamic compilation
  - DyC - *Grant et al.*

Boostrap



	2	4	8	16	32
Initialization	0.02	0.09	0.17	0.39	0.77
Analysis	0.00	0.00	0.00	0.00	0.01
Master MPI	0.00	0.02	0.00	0.00	0.00
Max wkr serial.	0.00	0.00	0.00	0.00	0.01
Max wkr MPI	0.00	0.00	0.01	0.01	0.00
Max wkr socket	0.00	0.00	0.00	0.00	0.00

Table 2. Itemized overhead with the bootstrap code, in percentage of the total execution time. The sequential execution time of bootstrap is 2918.2 seconds.

Task Parallelism Test

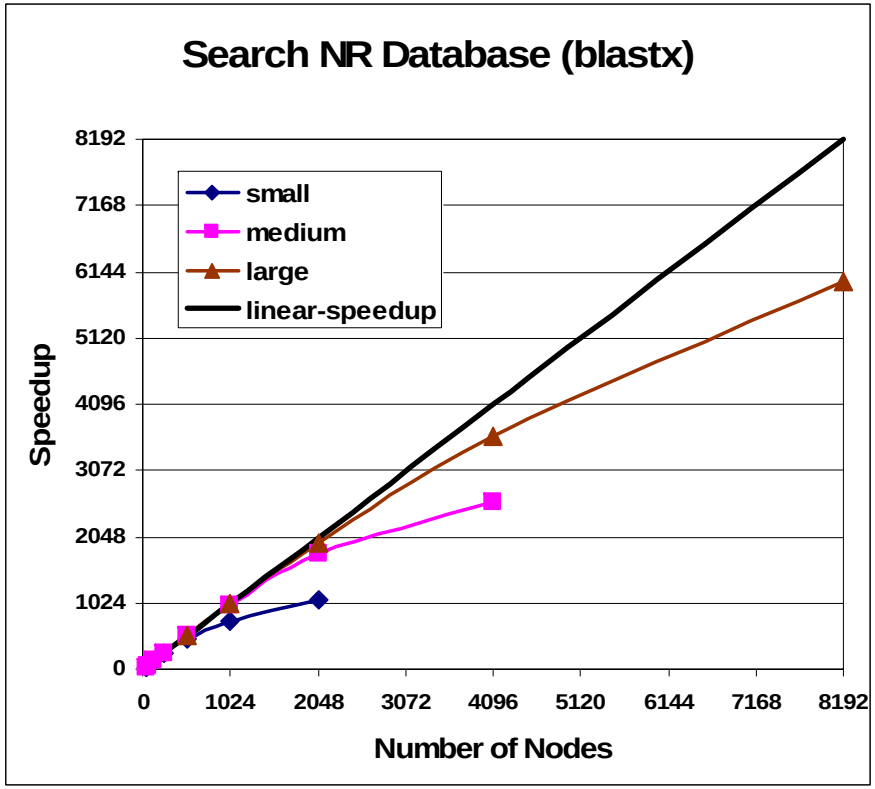
- Statistical functions
  - **prcomp** – principal component analysis
  - **svd** – singular value decomposition
  - **lm.fit** – linear model fitting
  - **cor** – variance computation
  - **fft** – Fast Fourier Transform
  - **qr** – QR decomposition
- Execution time of task
  - 3-27 seconds

```
a <- array(rnorm(1000000), dim=c(1000,1000))
b <- matrix(scan("test.data"), 1000, 1000)
c <- rnorm (1000)

s <- prcomp(b)
sd <- svd(a)
l <- lm.fit(b,c)
st <- sort(a)
f <- fft(b)
sv <- solve (a,c)
sp <- cor(b, method = "spearman")
q <- qr(a)
```

	1	2	4	8
Exec time	87.69	114.86	45.8	37.9
Speedup	1	0.76	1.91	2.31

NR Database Results

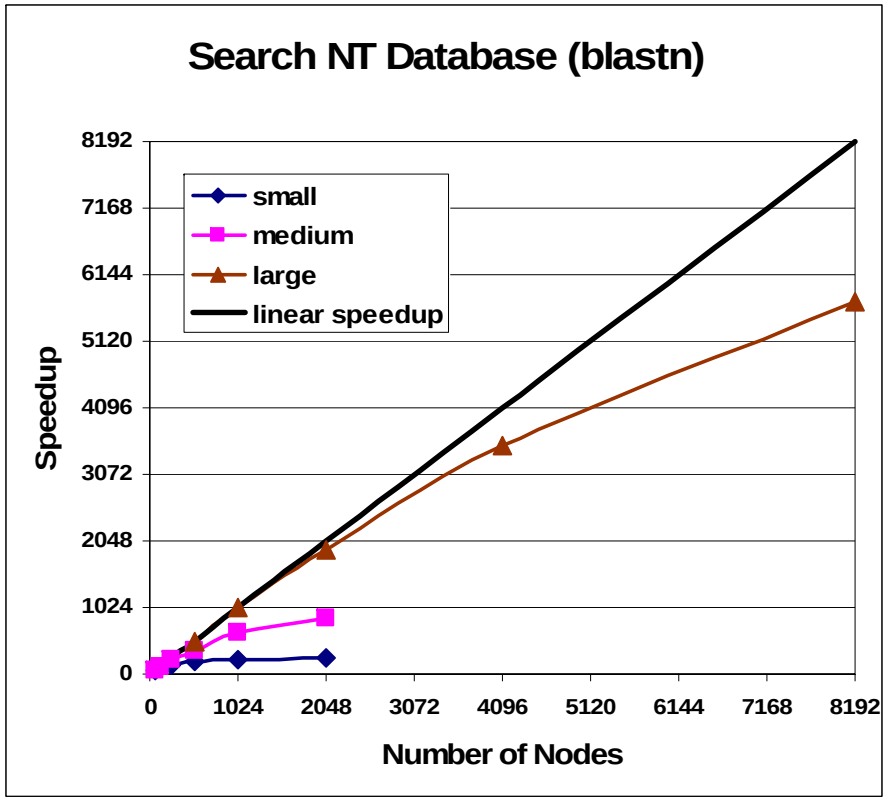
- Large query set scale up to 8192 processors (74% efficiency)



#Nodes	Small	Medium	Large
32	721.8	4073.9	
64	334.1	1963.9	
128	171.7	993.7	
256	96.2	504.0	
512	50.0	251.4	5620.6
1024	31.5	131.2	2863.0
2048	21.3	73.4	1484.6
4096		50.3	796.7
8192			479.2

NT Database Results

- large query set scale almost linearly to 2048 (93% efficiency), continue to 8192 (70% efficiency)



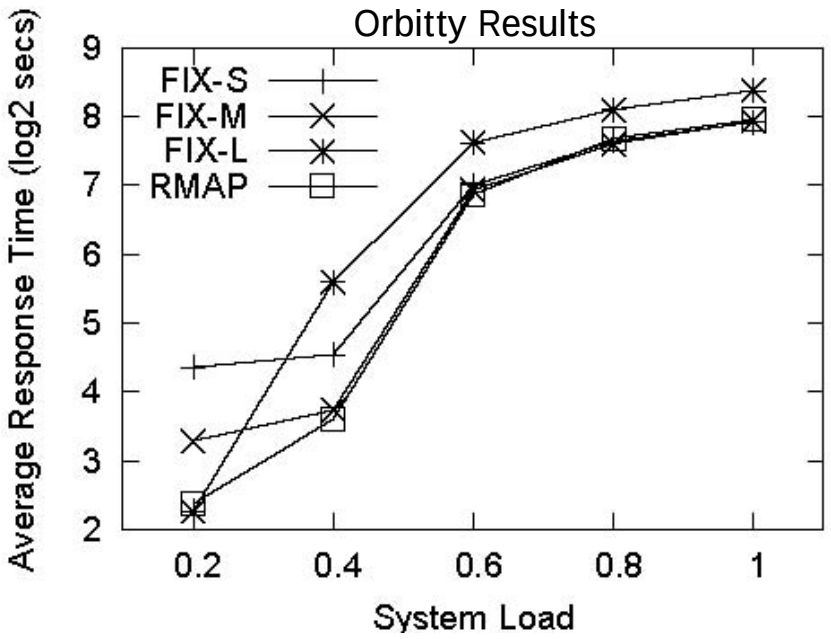
#Nodes	Small	Medium	Large
32	435.8	2395.2	
64	250.6	1374.9	
128	193.7	690.2	
256	142.85	416.1	
512	125.3	244.2	7108
1024	111.6	178.4	3589.6
2048			1895.7
4096			1037.1
8192			636.3

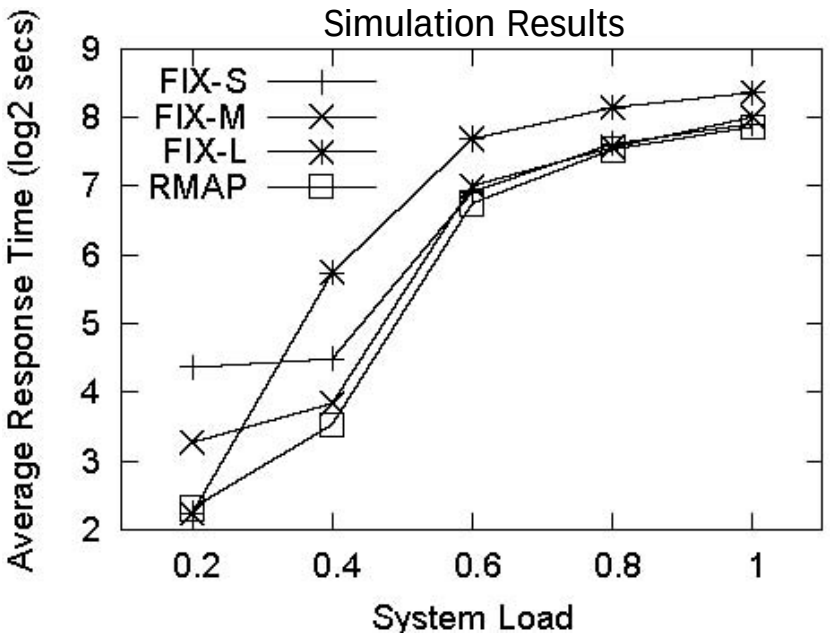
Processor Scheduling Related Work

- Space sharing parallel job scheduling
  - Static partitioning
    - Fixed partition size
  - Adaptive partitioning
    - Partition size determined at scheduling, remain unchanged until finish
  - Dynamic partitioning
    - Partition size adjusted during execution
    - High overhead in distributed shared memory machine

Buffer Cache Simulation Verification

- Compare processor scheduling with results from real cluster
- NCSU Orbitty
  - 20 computation nodes
  - each node with dual Intel Xeon 2.40 GHZ CPUs
  - NFS shared file system, Linux operating system





Compare Strategies Under Different Local Storage Limit, I/O Bandwidth