Automatic Parallelization for Scripting Languages: Toward Transparent Desktop Parallel Computing

Xiaosong Ma\textsuperscript{1,2}
Jiangtian Li\textsuperscript{1}
Nagiza Samatova\textsuperscript{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 interactivity 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

End user w. data processing apps.

Resource donors

Opportunistic resources (CPU, storage)

Xiaosong Ma, NCSU/ORNL
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
    - Message passing
    - Shared memory
    - Back-end support
    - 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
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

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

### Table: Speedup Comparison

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>0.05%</td>
<td>0.13%</td>
<td>0.31%</td>
<td>0.65%</td>
<td>1.28%</td>
</tr>
<tr>
<td>Analysis</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.01%</td>
<td>0.04%</td>
</tr>
<tr>
<td>Master MPI</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.01%</td>
</tr>
<tr>
<td>Max wkr serial.</td>
<td>0.42%</td>
<td>0.69%</td>
<td>1.15%</td>
<td>2.05%</td>
<td>3.19%</td>
</tr>
<tr>
<td>Max wkr MPI</td>
<td>0.00%</td>
<td>0.03%</td>
<td>0.07%</td>
<td>0.15%</td>
<td>0.26%</td>
</tr>
<tr>
<td>Max wkr socket</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.02%</td>
<td>0.04%</td>
<td>0.05%</td>
</tr>
</tbody>
</table>

### Graph: Speedup vs. Number of Processors
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


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
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

\begin{verbatim}
# task 1
a <- 1 b <- 2
c <- rnorm(9) d <- array(0:0, dim=c(9,9))

# task 2
for (i in 1:length(c))
  c[i] <- c[i-1] + a

# task 3
for (i in 1:length(c))
  d[i,] <- matrix(scan(paste("test.data", i, sep="")))

# task 4
if (c[length(c)] > 10)
  e <- eigen(d)
else
  e <- sum(c)

# task 5
for (i in 2:9)
  c[i] <- c[i-1] + a

# task 6
for (i in 1:5)
  d[i,] <- matrix(scan(paste("test.data", i, sep="")))

if (19 > 10)
  e <- eigen(d)
else
  e <- sum(c)

for (i in 1:9)
  d[i,] <- matrix(scan(paste("test.data", i, sep="")))

Example: Runtime Analysis

if (c[length(c)] > 10)
  e <- eigen(d)
else
  e <- sum(c)
\end{verbatim}
SVD

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

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>0.23%</td>
<td>0.49%</td>
<td>0.78%</td>
<td>1.12%</td>
<td>1.27%</td>
</tr>
<tr>
<td>Analysis</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.01%</td>
<td>0.02%</td>
</tr>
<tr>
<td>Master MPI</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.01%</td>
</tr>
<tr>
<td>Max wkr serial</td>
<td>11.7%</td>
<td>26.5%</td>
<td>41.7%</td>
<td>53.0%</td>
<td>58.0%</td>
</tr>
<tr>
<td>Max wkr MPI</td>
<td>0.00%</td>
<td>2.10%</td>
<td>4.32%</td>
<td>6.44%</td>
<td>7.83%</td>
</tr>
<tr>
<td>Max wkr socket</td>
<td>1.45%</td>
<td>1.56%</td>
<td>1.99%</td>
<td>2.40%</td>
<td>2.51%</td>
</tr>
</tbody>
</table>
pR Analyzer

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

- **Parser**
- **Dependence Analyzer**
- **Loop Parallelizer**

Input:
- R script

Output:
- Task Precedence Graph

Diagram:
- R script
- Parse Tree
- Task Precedence Graph
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.
Bootstrap

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>0.02</td>
<td>0.09</td>
<td>0.17</td>
<td>0.39</td>
<td>0.77</td>
</tr>
<tr>
<td>Analysis</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Master MPI</td>
<td>0.00</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Max wkr serial.</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>Max wkr MPI</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>Max wkr socket</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

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)
```
NR Database Results

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

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>721.8</td>
<td></td>
<td>4073.9</td>
</tr>
<tr>
<td>64</td>
<td>334.1</td>
<td></td>
<td>1963.9</td>
</tr>
<tr>
<td>128</td>
<td>171.7</td>
<td></td>
<td>993.7</td>
</tr>
<tr>
<td>256</td>
<td>96.2</td>
<td></td>
<td>504.0</td>
</tr>
<tr>
<td>512</td>
<td>50.0</td>
<td></td>
<td>251.4</td>
</tr>
<tr>
<td>1024</td>
<td>31.5</td>
<td>131.2</td>
<td>2863.0</td>
</tr>
<tr>
<td>2048</td>
<td>21.3</td>
<td>73.4</td>
<td>1484.6</td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td>50.3</td>
<td>796.7</td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td></td>
<td>479.2</td>
</tr>
</tbody>
</table>
NT Database Results

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

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>435.8</td>
<td>2395.2</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>250.6</td>
<td>1374.9</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>193.7</td>
<td>690.2</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>142.85</td>
<td>416.1</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>125.3</td>
<td>244.2</td>
<td>7108</td>
</tr>
<tr>
<td>1024</td>
<td>111.6</td>
<td>178.4</td>
<td>3589.6</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td></td>
<td>1895.7</td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
<td>1037.1</td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td></td>
<td>636.3</td>
</tr>
</tbody>
</table>
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

Data Scheduling - 5G/50MB

Data Scheduling - 10G/50MB

Data Scheduling - 5G/200MB

Data Scheduling - 10G/200MB