A Parallel Approach to Microarray Preprocessing and Analysis

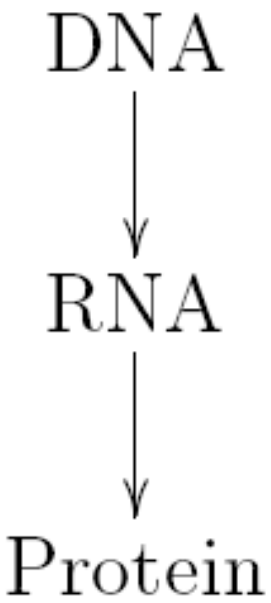
Patrick Breheny

2007

Outline

- Introduction
- Preprocessing
- Analysis

The Central Dogma

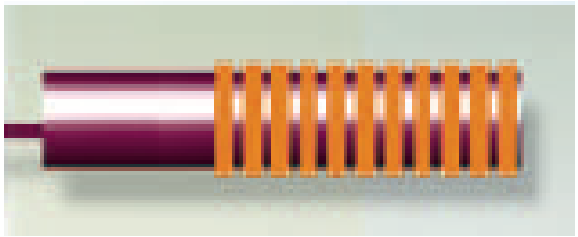


Purifying and labeling RNA

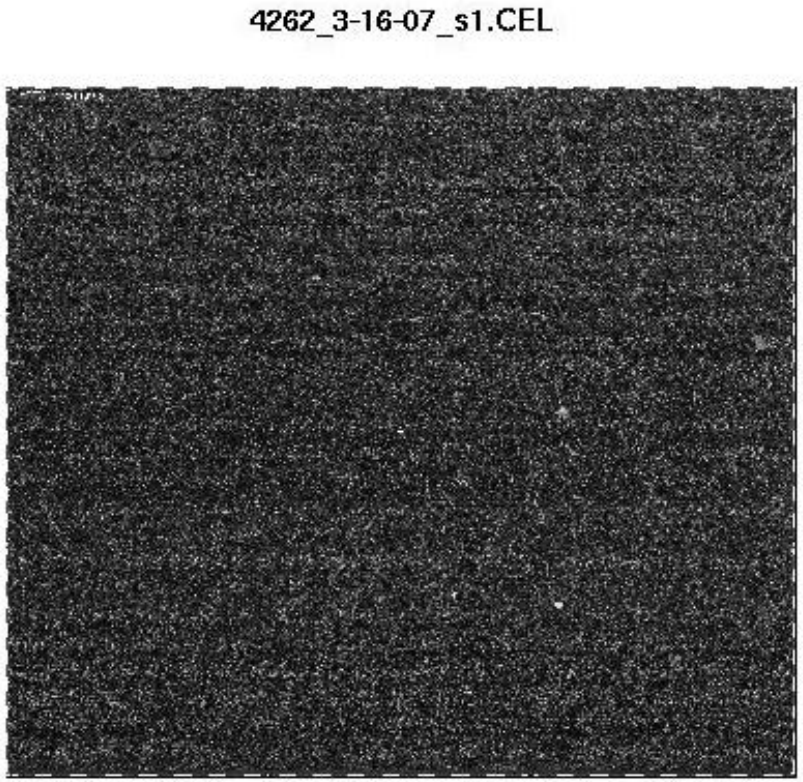
- Measuring of the amount of RNA corresponding to specific genes requires a number of steps, each of which introduces noise
- First, the RNA from a sample must be purified (a process taking several days of laboratory work)
- Next, the RNA must be labeled with a fluorescent dye so that we can quantify its concentration with a scanner

Probes

A common approach to identifying the RNA from specific genes is through the use of *oligonucleotide probes*



A Microarray



The microarray above contains $716 \times 716 = 512656$ probes (22 probes per gene)

Statistical issues

The prominent statistical issues are:

- Is there an experimental bias favoring some types of probes?
- Does a measurement of "100" on one microarray mean the same thing as a measurement of "100" on a different microarray?
- What are we going to do with the 22 measurements per gene?

Statistical issues (cont'd)

An ideal analysis would integrate measurement error into the eventual analysis. However, this is generally not done for two practical reasons:

- Can no longer run conventional statistical analyses
- A vector of probe-level intensities from a single microarray occupies 4 MB of system memory

Preprocessing

- Therefore, microarray data is usually *preprocessed* before it is analyzed
- There are dozens (thousands if you consider mixing and matching) of approaches to preprocessing, but they generally include the following steps:
  - Background adjustment
  - Normalization
  - Summarization
- I will focus on a parallel implementation of Wu and Irizarry's (*JASA*, 2004) GCRMA approach to microarray preprocessing

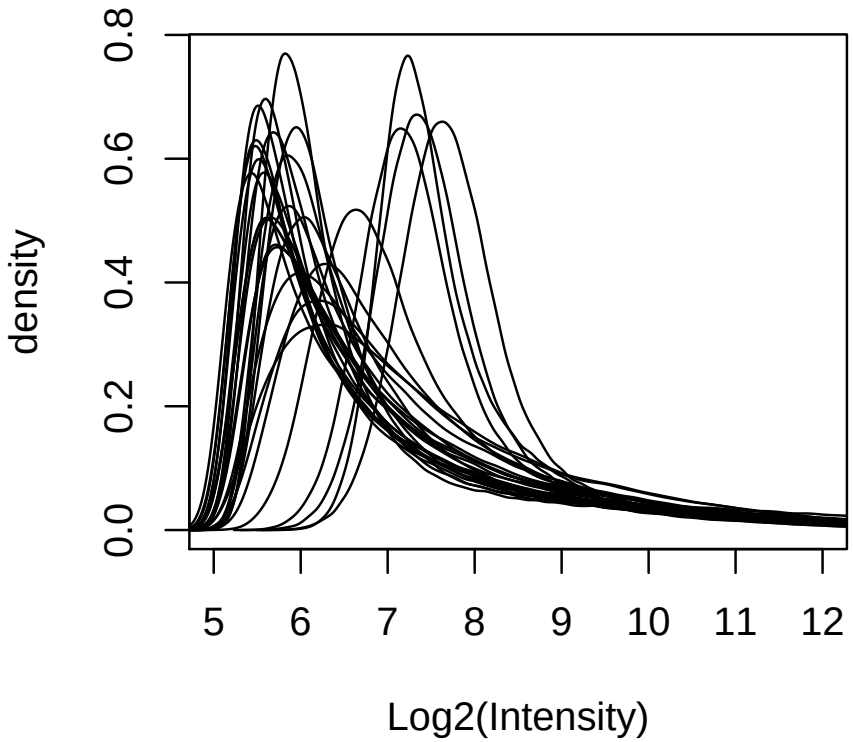
Background Adjustment

- Quality control microarray experiments using known concentrations of RNA have demonstrated that certain probes consistently give more signal than others even with the same sample (or no sample)
- The GCRMA approach relies on a stochastic model motivated by hybridization theory and fit to experiments of known standards to estimate probe affinities
- The probe affinities are then used as parameters in a model for observed intensity
- As background adjustment, an empirical Bayes estimate of the signal replaces the observed intensity

Background Adjustment: Parallelization Concerns

- Ideally, each processor could perform background adjustment on its own microarray(s)
- However, to fit the background adjustment model, certain parameters assumed to be constant over microarrays (e.g. variances) need to be estimated
- Nevertheless, there is more than enough data to estimate these parameters "locally"

The Need for Normalization



Normalization

- The most common approach to normalization is *quantile normalization*
- The algorithm is as follows, given a matrix $\mathbf{X}$:
  - Sort each column of $\mathbf{X}$
  - Find the mean of each row
  - Assign to each element of $\mathbf{X}$ the mean value of its within-column rank
- Easy to parallelize using `parRapply` and `parCapply`, although it potentially suffers from excess communication

Summarization

- At this point, we have, for each gene, a $p \times n$ matrix of adjusted intensities that we wish to summarize into an n-dimensional vector of expression levels
- This suggests a two-factor ANOVA model, where our outcome of interest would be the estimated column means
- Out of concern for potential outliers, the more robust *median polish* approach is used instead of an ANOVA model
- In principle, there is no problem parallelizing this operation, although I found the overhead costs of setting up the parallelization to trump the gains

Implementation: Data

- 27 microarrays
- 512656 probes per microarray
- $\sim$ 22 probes per gene
- 17361 genes

Implementation: Top-level Code

```
parProcess <- function(data.directory)
  {
    filenames <- list.files(data.directory,full.names=T)
    cl.split <- clusterSplit(cl,filenames)
    p <- length(filenames)

    sorted.list <- parLapply(cl,cl.split,parRAS,out.name="ind")
    sorted <- matrix(unlist(sorted.list),ncol=p)
    means <- parRapply(cl,sorted,mean)
    norm.list <- clusterCall(cl,parUnsort,means=means,ind.name="ind")
    norm <- matrix(unlist(norm.list),ncol=p)

    val <- calcRMA(norm)
    val
  }
```

Results

```
> system.time(E.std <- process(data.directory))
238.297  10.222 248.655

> system.time(E.fast <- process(data.directory,fast=T))
 91.922   9.867 101.811

> system.time(E.par3 <- parProcess(data.directory))
 13.309   2.474 137.526

> system.time(E.par9 <- parProcess(data.directory))
 10.948   2.692  54.103

> system.time(E.par27 <- parProcess(data.directory))
 16.084   2.947  36.522
```

Results (cont'd)

```
> mean(abs(E.par9-E.std)/E.std)
[1] 0.03134509
```

```
> mean(abs(E.fast-E.std)/E.std)
[1] 0.2389692
```

Analysis: Overview

- It is difficult to make broad generalizations about microarray analysis and whether operations can be made parallel
- But in general....
  - Expression as outcome: usually clear
  - Expression as predictors: less clear
  - Unsupervised learning: seems hard

My Analysis

- The microbiologists I worked with were interested in detecting changes in expression over a variety of experimental conditions
- An ANOVA model is generally appropriate
- In this case, then, our problem is to fit 17361 linear models

Fitting thousands of linear models

- A direct approach:

```
> system.time(fit <- apply(E.par9,1,do.lm,Design=Design))
   user  system elapsed
110.458   0.848 111.330
```

- A direct parallel approach

```
> system.time(fit <- parRapply(cl,E.par9,do.lm,Design=Design))
   user  system elapsed
 68.540   3.362  89.719
```

Fitting thousands of linear models (cont'd)

- However, this approach is enormously redundant (we invert the same matrix 17361 times)
- A function that fits the linear models efficiently would be useful
- Luckily, one already exists: `lm`

```
> system.time(fit <- lm(t(E.par9) ~ 0+Design))
   user  system elapsed
  0.513   0.223   0.736
```

- I thank the Statistical Computing & Graphics newsletter for this helpful tip

Conclusion

- Parallel computing using `snow` is fairly easy...
- ...in principle.
- However, functions that perform complex tasks require a good deal of work (and thought) to rewrite and optimize in a parallel manner