A parallel implementation of the Rank Product method for R

Lawrence Mitchell[*]

April 30, 2011

Abstract

We present the results of a project working on SPRINT to implement a parallel version of the rank product analysis technique for R. We have chosen a task parallel approach which, on typical data sets, shows good strong scaling on the HECToR supercomputing facility up to 4096 processes, with speedups over the original serial code of more than 4000. Previously time-consuming analyses that would run overnight, or even multiple days, may now be carried out in a few minutes. We also discuss a potential data parallel approach that would allow treatment of very large datasets.

Contents

[*]lawrence.mitchell@ed.ac.uk

1 Introduction

This dCSE project implements a parallel version of the rank product method (Breitling et al., 2004; Hong et al., 2006) for detecting differentially regulated genes in microarray experiments as part of the SPRINT library for R, enabling usage on the HECToR supercomputing facility and other parallel machines.

R (R Development Core Team, 2010) is an open source software package for statistical computing and graphics. The majority of the software is written in the the R programming language (which is scheme-like, interpreted and garbage-collected). The core of R consists of the interpreter for this language, written in C, along with the garbage collector. Additionally, a number of core functions are written in C for performance reasons. R provides a foreign function interface for both C and Fortran code. The C interface is richer and allows access to R objects directly as well as the ability to call back into R.

SPRINT (Hill et al., 2008; Petrou et al., 2010), the Simple Parallel R Interface is a project of the Division of Pathway Medicine and EPCC at The University of Edinburgh to provide parallelised workflows for microarray analysis within R. The aim is to provide functionality that is as close to existing code as possible so that end users, typically biostatisticians, can make use of HPC resources without altering their workflows excessively.

2 The rank product method

Rank products are a method of identifying differentially regulated genes in replicated microarray experiments. For example, we may be interested in the gene expression levels under two different experimental conditions (for example treated or untreated samples). Early approaches used a fold-change criterion (DeRisi et al., 1997), the ratio of the expression level in sample A to that in sample B. However, this method does not allow calculation of significance levels nor is it obvious what the cutoff value for the fold change should be. The rank product method builds on the fold-change criterion but applies it to replicated experiments. That is, rather than just a single sample representing each experimental condition (class) we have many samples in each class.

The rank product method is applicable to experiments comparing two different experimental conditions (class A and class B, say). For each gene we compute a rank product by ranking the fold-change value of that gene in all pairwise comparison of class A against class B, we then take the product of these ranks across all samples. The second step is to compute a null distribution for the rank products. This is the expected distribution if there is differentiation between neither genes nor samples. The experimentally observed rank product for each gene can be compared to the null distribution which allows accurate measures of the significance level and estimation of cutoff values (Breitling et al., 2004).

2.1 A detailed look at the algorithm

Consider some microarray data, which we may represent by an $N_g \times N_s$ matrix measuring the expression levels of N_g genes in N_s different samples. There are N_a samples from class A and N_b from class B ($N_s = N_a + N_b$). We construct a matrix containing the fold-changes for all pairwise comparisons of class A samples against class B samples (there are N_aN_b of these). Giving us a matrix of fold-changes

$$F = \begin{pmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,N_aN_b} \\ f_{2,1} & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ f_{N_g,1} & \cdots & \cdots & f_{N_g,N_aN_b} \end{pmatrix}, \tag{1}$$

where $f_{i,j}$ is the fold-change of gene i in the pairwise comparison j. We note that this step is necessary only for single-channel microarrays. Two-colour arrays measure fold-changes directly (rather than expression levels). These latter data are described as *one-class* as opposed to *two-class* and we analyse them in a similar way. The only difference is that we use the input data as the fold-change matrix directly.

Next we rank the fold changes in each sample from largest to smallest (for up-regulation) or smallest to largest (for down-regulation). Finally we compute the rank product of gene i by taking a suitably normalised product of the rank of that gene across all samples

$$r_i = \prod_{k=1}^{N_aN_b} r_{i,k}^{1/(N_aN_b)}. \tag{2}$$

where $r_{i,k}$ is the rank of gene i in comparison k.

To obtain siginificance levels for the experimentally observed rank products (say to determine which genes are statistically strongly up-regulated) we now need to to compare this experimentally observed value with the expected distribution of rank products under the null hypothesis. Recall that our null hypothesis is no differential expression of genes – expression levels of individual genes are independent of one another and drawn from the same distribution – and that all samples are independent (the expression level of gene i in sample A is uncorrelated with the level in sample B). Unfortunately, it is not possible to construct an analytic form for the null distribution, we therefore construct it numerically using a bootstrap procedure. We create a random experiment by independently permuting each sample's gene expression vector and calculate the rank product of all the genes in this random data. We repeat this many times to build a distribution of rank products for the null hypothesis.

Serial rank product analyses may be slow for a two reasons, which would necessitate different parallelisation approaches. The first is that the data to be analysed are very large: many genes and/or many samples. To decrease time to solution in this case, we must make the calculation of the rank product statistic for a single dataset faster. The second reason is that many bootstrap samples are requested, to decrease time to solution in this latter case, we must make calculation of bootstrap statistics faster. These two aspects naturally lead to different parallelisation options which we discuss in section 2.2.

2.2 Parallelisation options

There are two obvious parallelisation options available to us if we wish to speed up rank product analysis. The first is to implement a data parallel calculation of the rank product for a single dataset. The second is to calculate rank products in serial, but to parallelise across the bootstrap samples (a task parallel approach).

2.2.1 Data parallel calculation of the rank product

To decide which direction we would parallelise across, we need to consider the shape of the data. Microarray experiments typically have a small number of samples $1 < N_s < 10^3$ but a large number of expression measurements $10^4 < N_g < 10^7$.

2.2.2 Parallelising across samples

If we distribute the input data matrix, our implementation will necessarily require a lot of communication of the data as the following analysis shows. Typical experimental setups will have approximately equal numbers of samples in the two classes. Calculation of fold-changes will therefore requirement movement of $N_s N_g/2$ data elements from class B to each process carrying class A samples. There will be approximately $P/2$ processes with class A samples, requiring the movement of $\mathcal{O}(PN_s N_g)$ data in total to calculate fold changes. So this method would save absolute memory requirements at the cost of communicating the total data P times, which does not seem like a net win.

An alternate approach requiring less movement of data would be to distribute the fold-change matrix. However, we would need to generate this in serial on the master process (which has access to the input data) introducing a bottleneck in the code. Additionally, to generate the bootstrap samples we need the input data, rather than the fold-change data. So every bootstrap sample would also require a serial step.

2.2.3 Parallelising across genes

The alternative approach is to distribute the rows of the data matrix across processes. Fold changes can now be computed locally, but we must perform a parallel sort of the fold change to rank the elements correctly. There are many different options for parallel sorting (see Akl (1985) for an early overview). However, recursive methods like parallel quicksort suffer because it is difficult to load balance the work: choosing a good pivot point without communicating excessively is not possible. Later efforts focus on heuristics for choosing better pivots. Modern methods rely on some kind of sampling technique to choose the pivot (Shi and Schaeffer, 1992; Helman et al., 1998). These algorithms have good guarantees on the asymptotic worst-case behaviour of the required data movement and show good speedup in experimental studies. When the number of genes to be sorted, $N_g \geq P^3$, a sort requires $\mathcal{O}(N_g/\sqrt{P})$ data movements. Since we must sort N_s samples, the total required data movement is $\mathcal{O}(N_g N_s/\sqrt{P})$. This is a factor of $P^{3/2}$ better than the previously described approach. Bootstrap resampling would require a parallel permutation of each gene vector, prior to the calculation of the rank product using parallel sorting.

Given this analysis, although the code complexity of parallelising across genes would be higher, it would seem sensible to implement data parallel distribution across genes rather than samples.

2.2.4 Task parallel bootstrapping

A simple parallelisation approach is to generate the bootstrap samples, and the associated null distribution of rank products in parallel. We can then combine these partial distributions into a full distribution to measure statistical properties of the observed rank product.

2.2.5 Memory concerns

One reason to choose a data parallel approach over a task parallel one is that the data do not fit into memory on a single process. At first sight it appears that the memory usage of our algorithm is dominated by the the fold-change matrix ($\mathcal{O}(N_g N_a N_b)$ bytes), and the bootstrapped distribution ($\mathcal{O}(N_g N_{\text{boot}})$ bytes). However, a little thought allows us to eliminate both of these memory requirements. Rather than precomputing fold-changes and ranking them all, we can just compute the rank of one fold-change and keep a running product of the rank product for each gene. In addition to the storage for the input data, this requires $\mathcal{O}(N_g)$ temporary data.

For the bootstrap samples, we note that the statistics we are interested in are essentially $\int_{-\infty}^{r_i} P(x)dx$. Where r_i is the observed rank product of gene i and $P(x)$ is the bootstrapped null distribution. Rather than gathering the whole distribution and calculating this integral at the end of the computation, we can instead just keep a running count for each gene which we increment after every bootstrap sample. This reduces the memory requirement of the bootstrap phase to $\mathcal{O}(N_g)$ bytes. An additional advantage to this approach is a reduction in the runtime complexity of the bootstrapping phase. If we generate B bootstrap samples the gather and count approach has complexity $\mathcal{O}(N_g B \log N_g B)$ (sorting an array with $N_g B$ elements and then searching for the position of N_g elements in it) whereas our approach has complexity $\mathcal{O}(N_g B \log N_g)$ (sorting an array with N_g elements and searching for N_g elements in it, carried out B times).

Very large microarray data might have 10^6 expression measurements per sample and one hundred samples. As such, we have not found it necessary to implement a data parallel approach in this project. We have instead just implemented parallel bootstrapping. Should future data sizes require a data parallel approach, the method outlined in section 2.2.1 could be used.

3 Implementation

As with the other parallel functions in SPRINT, our aim is to reproduce the serial code both in results and the interface exposed to the end user. Our parallel implementation mimics the calling convention of the serial code, for which a typical call might look like

```
library(RankProd)
# data contains N_g rows each with N_s samples; classes
# contains N_s entries indicating the class of each sample
rp <- RP(data, cl=classes, num.perm=100, logged=FALSE)
# do further analysis of data
```

The equivalent parallel analysis with SPRINT is reproduced below, where parallelisation is carried out across the `num.perm` bootstrap samples. The change to the analysis script is minimal, we must load the parallel library, change `RP` to `pRP` and terminate MPI at the end of the script.

```
library(RankProd)
# load parallel library
library(sprint)
# replace RP with pRP
rp <- pRP(data, cl=classes, num.perm=100, logged=FALSE)
# do further analysis of data then exit MPI.
pterminate()
```

The R interface sets up various data structures and sanitises the input before calling out to a C function to carry out the actual computation. Results are computed on the various slave processes before being gathered on the master process and returned to R.

4 Benchmark results

We report results on the XT4 configuration of the HECToR supercomputing service. Previous work (Mitchell, 2011) has shown little difference in performance between the XT4 and XE6 configurations for the communications patterns required in this SPRINT project. Our test data are a typical microarray study, measuring the expression levels of 23292 genes across 62 samples in two classes (35 in class A, 27 in class B). Calculation of the two-class rank product therefore requires 945 pairwise comparisons to be ranked. To study the behaviour of the one-class rank product, we only use the data from class A samples. Calculation of the rank product in this case ranks 35 samples. Since the algorithmic complexity is linear in the number of samples to be ranked, we expect the one-class rank product to be calculated approximately 27 times faster than the two-class case.

4.1 Expected performance

Parallelisation of the bootstrap starts with a broadcast the input data to slave processes, calculation of the partial distribution of rank products requires no communication, finally further communication is required to gather the partial counts to the master process. We therefore expect that the scaling performance should be good: we should only see a drop-off in scaling when the number of bootstrap samples per process becomes very low.

A note on the timing data. A measure of the wallclock time for the execution of our parallel implementations includes two serial sections. We must calculate the experimental rank product values for up- and down-regulated genes in serial. We then bootstrap distributions for up- and down-regulated genes in parallel. The maximum possible parallel efficiency for the wallclock time of the analysis on P processes relative to a serial calculation is

$$\eta_p = \frac{b+1}{b+P} \tag{3}$$

where b is the number of bootstrap samples requested. To give an indication of the performance of our code, we plot this theoretical maximum along with the experimentally obtained values.

4.2 Actual performance

We first look at the performance of the code when analysing one-class data. The number of samples in this case is just the number of classes, which for our analysis is 35. Figure 1 shows the speedup and parallel efficiency of the whole analysis when bootstrapping 1024 and 16384 samples.

In comparison, the analysis of a similar sized two-class dataset takes significantly longer since the number of pairwise comparisons increases. We show results analysing data with 35 experimental samples in one class and 27 in another. This leads to 945 pairwise comparisons in the rank product analysis. Increasing expected runtime over the previous one-class example by a factor of around 27. Figure 2 shows the speedup and parallel efficiency of the whole analysis when bootstrapping 1024 and 16384 samples. We have not attempted to analyse 16384 samples in serial, although we would expect the runtime to be around 120 hours. Note in both the one-class and two-class analysis the initial decrease in parallel efficiency moving from one to four parallel processes. This is due to increasing contention for memory on a single HECToR node, note how in figure 2 the same effect is not seen for the case of 16384 samples when moving from 16 to 32 processes.

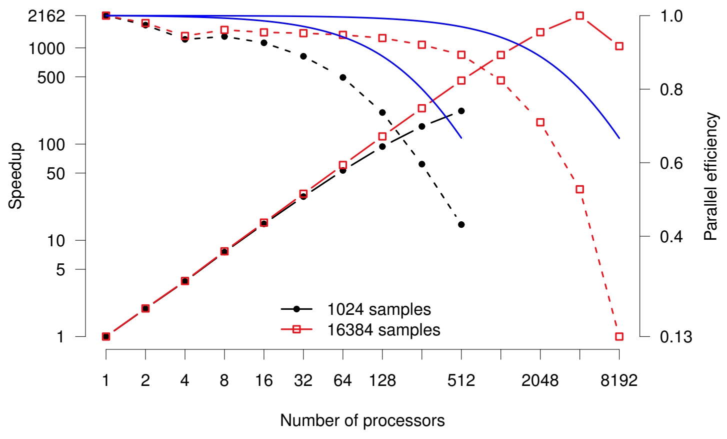

Figure 1 *Speedup (solid lines) and parallel efficiency (dashed lines) for one-class rank product analysis, blue lines show theoretical peak efficiency as given in equation 3. Dataset has 23292 genes and 35 samples. Execution time on a single core is 1100 seconds for 1024 samples and 17300 seconds for 16384 samples. In comparison, the original* `RankProd` *code requires 2100 seconds for 1024 samples.*

The choice made by bioinformaticians for the number of bootstrap samples is partly motivated by time constraints (Dunbar, 2011). However, if we wish to report p-values to a high degree of accuracy, we need to choose a large number of bootstrap samples. Typically, the size of the error in the measured p-value with N bootstrap samples is $1/\sqrt{N}$. Thus, if we want our p-values to be accurate to three decimal digits, we need around 10^6 bootstrap sample comparisons. Note that this is not the same as 10^6 bootstrap samples. Since each bootstrap sample generates N_g rank products which we can compare with our experimental value, if we have 10^4 genes, we only need 100 bootstrap samples to get 10^6 comparisons. If we want four decimal digits, we need 10^8 comparisons and would require 10^4 bootstrap samples.

4.3 Performance for larger datasets

Future microarray data will measure upwards of 10^5 expression levels on each sample, either exon markers, or single nucleotide polymorphisms (SNPs). If we double the number of expression levels per sample we expect, all other things being equal, that the runtime will increase by a factor of $2\frac{\log 2N_g}{\log N_g}$. This is because the algorithmic complexity is $\mathcal{O}(N \log N)$ in the number of genes. Table 1 shows the expected and measured runtimes when we increase the

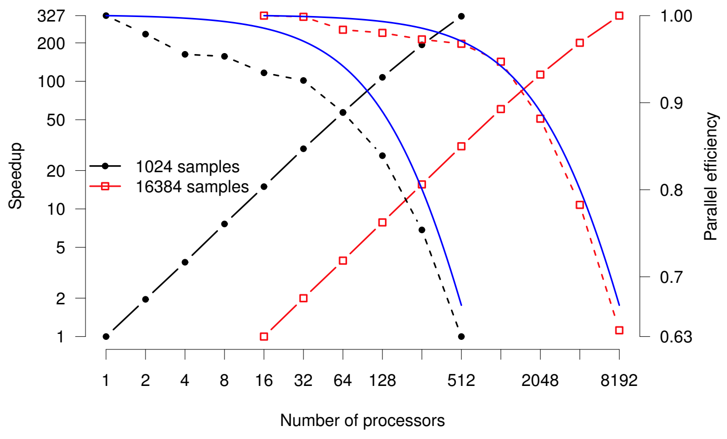

Figure 2 *Speedup (solid lines) and parallel efficiency (dashed lines) for two-class rank product analysis, blue lines show theoretical peak efficiency as given in equation 3. Dataset has 23292 genes and 35 + 27 samples, leading to 945 pairwise comparisons. Execution time on a single core is 26000 seconds for 1024 samples and estimated at 434000 seconds for 16384 samples. In comparison, the original* `RankProd` *code requires 42300 seconds for 1024 samples.*

number of gene expression values in our analysis. Our code shows correct algorithmic scaling with increasing numbers of genes, and we can keep the wallclock time to a reasonable level by increasing the number of parallel processes.

5 Conclusions

We have successfully implemented a task parallel version of the rank product method for analysis of microarray expression data. Since use cases suggested that the time-consuming part of the calculation was the bootstrap of the null distribution, we only parallelised this phase. Our implementation is approximately twice as fast in serial as the existing `RankProd` package and shows excellent parallel scaling. For large numbers of bootstrap samples our code will run thousands of times faster in parallel on HECToR than in serial, taking minutes, not days to deliver the results of an analysis. Development of the code has been carried out in the main SPRINT repository and will be incorporated into the next SPRINT release.

Table 1 *Runtime (in seconds) for a one-class analysis with 35 samples as the number of genes increases. Bracketed figures show the expected runtime using timings for 23292 genes as a base if the algorithmic complexity of $\mathcal{O}(N_g \log N_g)$ has the same prefactor for all data sizes*

MPI processes	23292 genes	46584 genes	93168 genes	186336 genes
1	1077	2344 (2302)	5128 (4902)	10624 (10398)
2	553	1202 (1182)	2574 (2517)	5487 (5339)
4	287	622 (614)	1348 (1306)	2905 (2771)
8	143	314 (306)	679 (651)	1468 (1381)
16	73	168 (156)	351 (332)	754 (705)
32	38	91 (81)	179 (173)	385 (367)
64	20	53 (43)	96 (91)	204 (193)
128	11	34 (24)	53 (50)	114 (106)
256	7	24 (15)	33 (32)	69 (68)
512	5	19 (11)	25 (23)	46 (48)

5.1 Future work

Although our implementation is reasonably memory efficient, future microarray data may necessitate a data parallel approach. We outlined one such possible scheme in section 2.2.1. However, we do not expect this to be an issue until data set sizes reach 1GB (around 1 million expression levels and 150 samples).

6 Acknowledgements

The SPRINT project is funded by a Wellcome Trust grant [086696/Z/08/Z]. Data were kindly provided by Thorsten Forster of the Division of Pathway Medicine, University of Edinburgh.

This project was funded under the HECToR distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR – A Research Councils UK High End Computing Service – is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE support Service is provided by NAG Ltd. http://www.hector.ac.uk.

References

S. G. Akl. *Parallel Sorting Algorithms.* Academic Press, Orlando, Florida, 1985.

R. Breitling, P. Armengaud, A. Amtmann, and P. Herzyk. Rank products: a simple, yet powerful, new method to detect differentially regulated genes in replicated microarray experiments. *FEBS Letters*, 573:83–92, 2004. doi: 10.1016/j.febslet.2004.07.055.

J. L. DeRisi, V. R. Iyer, and P. O. Brown. Exploring the metabolic and genetic control of gene expression on a genomic scale. *Science*, 278(5338): 680–686, 1997. doi: 10.1126/science.278.5338.680.

D. Dunbar. Personal communication, 2011.

D. R. Helman, J. Jájá, and D. A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *Journal of Experimental Algorithsm*, 3, 1998.

J. Hill, M. Hambley, T. Forster, M. Mewissen, T. Sloan, F. Scharinger, A. Trew, and P. Ghazal. SPRINT: A new parallel framework for R. *BMC Bioinformatics*, 9(1):558, 2008.

F. Hong, R. Breitling, C. W. McEntee, B. S. Wittner, J. L. Nemhauser, and J. Chory. Rankprod: a bioconductor package for detecting differentially expressed genes in meta-analysis. *Bioinformatics*, 22(22):2825–2827, 2006. doi: 10.1093/bioinformatics/btl476.

L. Mitchell. A parallel random forest implementation for R. Technical report, EPCC, 2011.

S. Petrou, T. M. Sloan, M. Mewissen, T. Forster, M. Piotrowski, and B. Dobrzelecki. Optimization of a parallel permutation testing function for the SPRINT R package. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 516–521, 2010. doi: 10.1145/1851476.1851551.

R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2010. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.