TECHNICAL REPORT 1

BSC-INB

COMPUTATIONAL BIOLOGY PROGRAM

Javier Vera (BSC-INB)

A parallelization of a bioinformatic program: PHASE

Javier Vera. Barcelona Supercomputing Center (Spain) javier.vera@bsc.es

Summary of the project

This paper presents a way to parallel PHASE, a program that implements a Bayesian statistical method for reconstructing haplotypes from population genotype data. The software can deal with SNP, microsatellite, and other multi-allelic loci and missing data are allowed.

This version of PHASE can perform a permutation test for assessment of the significance of differences in haplotype frequencies in case and control groups. This feature test the null hyphotesis that the case and control haplotypes are a random sample from a single set of haplotype frequencies, versus the alternative that cases are more similar to other cases than to controls.

The work presented here is focussed in the parallelization of PHASE to improve its performance in computers with parallel architecture. The effort has been focussed in the parallelization of the permutation test for assessing significance of differences in haplotype frequencies. Accordingly, the speedup obtained by executing the parallel version in a multiprocessor only will be noticed if it is turned on (-c parameter).

1. Introduction to PHASE

PHASE[1] is a popular bioinformatics program designed to process and to derive Haplotype information on population data, something crucial in current genomic studies [2, 3, 4, 5]. The software is free for non-commercial use, and may be licensed for commercial use.

PHASE creates statistical models that can be used to infer phase at linked loci from genotype, defining then haplotypes. The two most statistical methods used in haplotype studies are maximum likelihood, implemented via the expectation-maximization algorithm [6, 7, 8] and a parsimony method [9]. PHASE is a new statistical method that improves on these by exploiting ideas from population genetics and coalescent theory that make predictions about the patterns of haplotypes to be expected in natural populations. For more details on PHASE see [10, 11].

This report shows an external parametrization work of parallelization of PHASE. All the work was done without knowing about PHASE code, which implies that this methodology can be applied to programs. The work done considers PHASE as a simple black-box dealing only with program structures, algorithms, etc. without detail knowledge of the mathematical or physical principles guiding the algorithms. Using this procedure, no is possible a great improvement in the algorithms, but I will show that despite of that a tremendous improvement in the performance of the application is possible.

2. Motivation

PHASE execution is very compute-intensive; as can be seen beyond, a single execution may take almost three days. Normally, the user of PHASE collects data and runs a single execution of the program. Since there are not other executions to perform in parallel, the only way to get the results in less time is making faster the execution of PHASE. This can be done by parallelizing the code.

3. PHASE

To run PHASE [16], a proper environment needs to be created. The user is the responsible of creating this environment, extracting the files from the tar.gz file, preparing the input files and read the manual to know what parameters are needed.

Before the "production" runs of PHASE tests and benchmarking was performed to: i) verify the goodness of results and ii) to obtain a set of reference values before parallelization. After this, we can focus in the most interesting part for this study: the feature case-control permutation test of PHASE.

3.1 Performing a case-control permutation test

Version 2 of PHASE can perform a permutation test to validate the significance of differences in haplotype frequencies in case and control groups. More precisely, PHASE tests the null hypothesis that the case and control haplotypes are a random sample from a single set of haplotypes frequencies, versus the alternative possibility, i.e: that some cases are more similar to other cases than to controls. To perform the permutation test two things must be done:

1. specify the case-control status of each individual in the input file
2. use -c flag when running the program to tell PHASE that the input file contains case-control status data

The number of permutations performed in PHASE can be specified after the -c flag. Eg. -c100 will perform 100 permutations. If no number is specified, the default is 100 permutations. The permutation test is quite computer-intensive, and in fact the authors of PHASE recommend to start with a low value and to decide then whether or not it is worth doing a larger number.

3.2 Execution time. Motivation for parallelizing

The execution time of PHASE performing case-control permutation test is quite long, depending on the input file it could take from some minutes -very small files- to several days. As an example, running an input file with 557 SNPs takes more than 69 hours in an Itanium2 processor. Increasing the number of SNPs will make unaffordable the calculation. This underlines the need to parallelize the program. The use way of PHASE is doing sporadic executions, only one run for each study. Therefore the main goal is to try to accelerate the single execution of PHASE. To be efficient, in this process we should locate which part of the code of PHASE is the most used and try to parallelize it.

3.3 Execution profiling. Find out the zone to parallelize

The main problem of using profiling with tools like gprof [13] is the unpredictability of the dynamic behavior of the programs. That is, profile depends on input definition, which implies that the location of hot points requires of different executions to define consensus or systematic hot-spots. For this purpose, PHASE calculations were performed with inputs sizes of 557 SNPs to 20 SNPs, 10 SNPs and 5 SNPs.

After the profiling was done for these inputs and all results were filtered the hottest region of program was found. It is located in the HapListMCMCResolvePhaseRemove function, where the code spends more than 90% of the time. It is then clear that the target zone is inside this function.

3.4 Execution tracing. Examine the zone

Paraver[15] and Ompitrace[14] were used to trace the target function HapListMCMCResolvePhaseRemove. The instrumentation of this function has been done manually, it has consist in identify the different zones inside this function, flagging the main loops. After analyzing these loops, we found that PHASE has five important zones inside the target function, which we labelled as B1, B2, B3, B4 and B5. Trace analysis reveals B1 and B4 zone implies a very large consumption of CPU time, while regions B2 or B3 have a smaller impact in total execution time. More detailed analysis show that in fact two subzones (B4.1 and B4.2) can be located within the B4 zone. Taking together B1 and B4 zones represent more than 94% of execution time.

The contribution of a zone to the execution time can be due to different reasons: i) it is huge in code (ie nested loops or loops with great number of iterations), ii) the execution is slower than the other zones (ie calling hard functions) or iii) this zone is entered many times. For practical purposes it is useful to group these 3 zones in a different way: i) those which contribute with long time intervals but which are entered just a few times and ii) zones that contributes to the execution time along little intervals, but are visited very often.

It is worth noting that these two zones must usually be parallelized in a different way and synchronization must be preserved at all times. Caution is necessary, since if there are many threads performing a very fast work in parallel, maybe the benefits from parallelization are overcomed by the overhead of synchronizing these threads. A priori the big chunks are more likely to obtain a good Speedup parallelizing them that the little chunks because will require -in global terms- less points of synchronization.

4 Parallelization. OpenMP

I will summarize the process of parallelization, without entering in tedious technical details on the code structures that must be changed in order to parallelize with OpenMP[17] or with MPI[18], which are the two protocols used for parallelization of the program. There are two versions of the parallel code, they are quite similar but one is made using only OpenMP and the other is made using only MPI.

In this report only will be take into account the parallelized version with OpenMP, but the MPI version is quite similar.

After identifying the zones that can be parallelized to obtain a great performance the parallelization using OpenMP was not very complicated. It has been done using, basically, "parallel for" directives and some reduction operation after these.

The process of parallelizing an application sometimes is very tedious and must be feed with many executions and see what happens across these executions, to evaluate if the achievement is worth, etc. The speedup shown is good, but further improvement is possible, especially in reference to the scalability of the process for higher number of threads (see Figure 1 and Table 1).

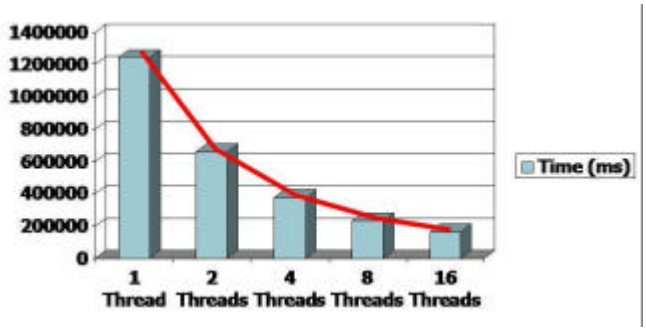

Figure 1: Reduction time achieved with the first parallelization

The poor scalability of the parallelization work indicates the presence of bottlenecks in the parallel zone. Using Paraver, the 2D analysis tool, one can spot that the Speedup achieved in the hot zones are not the same for each one. This suggests that there are parallel zones that potentially could be parallelized but in fact should be not parallelized because the overhead of synchronizing is huge and hinders the entire calculation. Thus we need identify these zones to avoid the parallelization or to change the way in what these zones are parallelized to improve the throughput.

With tracing tools we can look inside the hot zones and to identify the subzones that are causing the bottlenecks. The speedup obtained in the zone B1 or B4.2 is not very good (speedup 5 and 1.85), mainly because the lack of scalability. On the contrary, zone B4.1 achieves 15.23 Speedup (from an optimum 16).

Furthermore, we can look inside this zones trying find out where is the bottleneck. One way is to split these zones in more zones to isolate these that are not good to parallelize. We can do these with the help of tracing tools, manual instrumentation. Thus, Paraver demonstrate that inside B1 there are one zone that is not a good idea to parallelize because the overhead of parallelizing overcomes the benefits of the execution in parallel. This also happens in B4.2, then we need to avoid the parallelization of these problematics subzones (new labeled): B1.2, B4.2.2 and B4.2.3.

The parallel fraction of PHASE after avoiding zones that are not optimum for parallelization is about 92.79%. The maximum Speedup achievable is then given by Amdahl's Law[12]: 1 / (0.0721 + (0.9279/P)). Expanding P we have the Table 1 where are shown the maximum Speedup achievable and the current Speedup got with the last parallelization. Clearly, we are very close to the maximum theoretical limit given by Amdahl's law. Note that there is always a synchronization overhead that limits the Speedup and as more threads you put, more synchronization has to be done.

	2 threads	4 threads	8 threads	16 threads
Maximum	1.8655	3.2887	5.3167	7.6868
Achieved	1.8659	3.1482	4.7628	6.0241

Table 1: Limits of the Speedup and Speedup achieved by the parallelization

5 Conclusions and recommendations

PHASE is a good example of a very complex program that it would be very hard to parallelize without using profiling or tracing tools. The best thing to optimize a code execution comes, for sure, of knowing the code perfectly and be able to run many profiling recompilations with many different inputs. However, the time to do this with a very complex program as PHASE is sometimes prohibitive and require some expertise that is not common in software engineers.

I show here a way to implement parallelism in a complex program like PHASE. After working with PHASE several months, I know that a very big improvement could be taken from changing the code structures. In fact, the numbers in Table 1 are good but if we compare these numbers with the numbers obtained from the original version of PHASE they are even better. This means that PHASE original sequential code can be improved without parallelize it and with this parallel version I have inserted some of these improvements. Thus if you want to run PHASE in a single-processor machine, try to run this parallel version because it has some code tricks that may accelerate this execution even with only one processor. These code tricks are based on reusing memory –don't call several times to *malloc* and *free* subroutines if it is not necessary- and some minor changes in the way of using the STL of C++.

References

[1] PHASE v2 http://www.uwopendoor.org/ViewSoftware.asp?softwareid=10
parallel PHASEv2 http://www.uwopendoor.org/ViewSoftware.asp?softwareid=13

[2] N. Risch, K. Merikangas The future of genetics studies of complex human diseases. Science 273:15161517, 1996

[3] SE. Hodge, M. Boehnke, MA. Spence Loss of information due to ambiguous haplotyping of SNPs. Nat Genet 21:360361, 1999

[4] MJ. Rieder, SL. Taylor, AG. Clark, DA. Nickerson Sequence variation in the human angiotensin converting enzyme. Nat Genet 22:5962,1999

[5] RM. Harding, SM. Fullerton, RC. Griffiths, J. Bond, MJ. Cox, JA. Schneider, DS. Moulin, JB. Clegg Archaic African and Asian lineages in the genetic ancestry of modern humands. Am J Hum Genet 60:772789, 1997

[6] L. Excoffier, M. Slatkin Maximum-Likelihood Estimation of Molecular Haplotype Frequencies in a Diploid Population Molecular Biology and Evolution 12(5):921-927,1995

[7] M. Hawley, K. Kidd HAPLO: a program using the EM algorithm to estimate the frequencies of multi-site haplotypes. J Hered 86:409411, 1995

[8] JC. Long, RC. Williams, M. Urbanek An E-M algorithm and testing strategy for multiple locus haplotypes. Am J Hum Genet 56:799810, 1995

[9] AG Clark Inference of haplotypes from PCR-amplified samples of diploid populations. Molecular Biology and Evolution 7(2):111-122,1990

[10] M. Stephens, N. Smith, and P. Donnelly A new statistical method for haplotype reconstruction from population data. American Journal of Human Genetics, 68, 978–989.,2003

[11] M. Stephens, and P. Donnelly A comparison of Bayesian methods for haplotype reconstruction from population genotype data. American Journal of Human Genetics, 73:1162-1169., 2003

[12] G. Amdahl Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities AFIPS Conference Proceedings, (30), pp. 483-485, 1967

[13] S. L. Graham, P. B. Kessler, M. K. McKusick gprof: a Call Graph Execution Profiler SIGPLAN Symposium on Compiler Construction, 1982

[14] OMPITRACE: Instrumentation of combined OpenMP and MPI applications http://www.cepba.upc.es/paraver/docs/OMPItrace.pdf

[15] PARAVER: Parallel Program Visualization and Analysis tool http://www.cepba.upc.es/paraver/

[16] M. Stephens, N. Smith, and P. Donnelly Documentation for PHASE, version 2.1 http://www.stat.washington.edu/stephens

[17] OpenMP: Standard and specifications http://www.openmp.org

[18] MPI: The message Passing Interface standard http://www-unix.mcs.anl.gov/mpi/index.html

Acknowledgements

The author acknowledges the valuable help and advice of Prof. Xavier Messeguer and Prof. J.Dopazo. The author wishes to express his deepest gratitude to BSC-INB colleagues for many helpful discussions. I want to give a special acknowledgement to Prof. Modesto Orozco and Prof. Jesus Labarta by their aid and advice in writing and reviewing this technical report. This work was supported by Genoma España through the Instituto Nacional de Bioinformática.