International Journal of Computer Applications (0975 – 8887)
Volume 118 - No. 11, May 2015

A Hybrid OpenMP-MPI Parallelization of Structure Software

Rafal Dobosz
Trent University
Peterborough, ON, Canada

Richard Hurley
Trent University
Peterborough, ON, Canada

Sabine McConnell
Trent University
Peterborough, ON, Canada

ABSTRACT

Big Data has an increasing impact on the use of bioinformatics software. One way to deal with this challenge is through parallel computing. Using the program Structure as a case study, this paper investigates ways in which to counteract the challenges created by the growing datasets. This paper proposes an OpenMP-MPI hybrid parallelization of the MCMC steps, which are an integral part of Structure, and analyses the performance under various scenarios. The results indicate that the parallelization produce significant speedups over the serial version in all scenarios tested. This allows for the use of the hardware in a more efficient manner, by adapting the program to the parallel architecture. This is important because not only does it reduce the time required to perform existing analyses, but also opens the door to the analysis of previously impractically large datasets.

Keywords:

MPI, OpenMP, parallelization, Structure, MCMC, SPRNG, SHARCNET, speedup, Big Data, High Performance Computing

1. INTRODUCTION

Since the development of Structure over a decade ago [1], the size of datasets analyzed have been steadily increasing. This has occurred for two reasons: the rapid increase in the genetic sequencing speeds [2] [3] [4] [5], and the increase in processing power, making larger problems solvable in a more reasonable length of time. Thus, parallelizing Structure may lead to a significant decrease the time needed to perform the analyses, and so in the same amount of time, more genetic data may be analysed (or the same amount of data may be analysed more thoroughly). It may even open the door to new techniques, which simply were impossible with present computational power.

2. STRUCTURE DESCRIPTION

Structure was created to explore how a given group of organisms is structured [1]. It can answer questions such as: *What are the distinct populations in a given group? Which individuals belong to which populations? What are the hybrid zones and their characteristics?* It can also provide insight into which individuals are the migrants or are admixed (containing mixed ancestral origin), as well as their *allele population frequencies* (the rates at which different forms of genes or genetic locus tend to manifest in different populations) [1]. Structure works by estimating the likely characteristics of the populations given the genetic data provided with the help of the Markov Chain Monte Carlo algorithm (MCMC). MCMC generates a large number of samples from a probability distribution forming a Markov chain, which has the desired distribution as its equilibrium distribution.

In Structure, the algorithm generates a specified number of samples from the likely population structures and summarizes the samples to infer the actual structure. Structure's computational side is written in C. The program is open source, and there is a dedicated Google Group which provides additional support for the users. Structure was originally developed in 2000, with a number of iterations and updates that followed. The version of Structure used in this article for the implementation of parallelization is the original version from 2000. It does not include some of the additional functionality, but the fundamental function of the MCMC simulation to explore the structure of the populations is the same. The 2000 version was used because it shortened the code base and did not include some unused options, making decoding of the code base and parallelization design more straightforward. The parallelisation techniques are not in conflict with the differences, and the parallelization potential is expected to be comparable if adapted to the latest version.

The program reads the genetic data of the individuals, initializes the membership of individuals based on prior information or randomly if no such information is available, and initializes other parameters. After initialization, the program starts the MCMC process, performing the main loop as many times as specified by the burnin plus the post burnin steps. For each of the repetition, the program updates the P, Q and Z parameters, discussed in more detail later. The program then updates the α parameter and frequency priors. If the given repetition is a burnin step, the program starts the next repetition. For the steps after the burnin period, the application also collects all the data and compiles it to produce summary statistics. The program does not collect the summary statistics for the burnin period, since these steps are unreliable. After finishing the MCMC phase, the software performs finalizing steps, stores the results and closes.

3. PARALLELIZATION OPPORTUNITIES FOR THE MCMC METHOD

3.1 Embarrassingly Parallel Approaches

There are various approaches for parallelizing MCMC. The easiest and most straightforward is the *embarrassingly parallel approach*, for which independent instances of Structure may be run at the same time with different parameters. For example, on CPUs 1 to 10, the same simulation can be performed with the exception for the parameter identifying the number of expected populations. The results can then be used to infer which result is most likely. This type of parallelization offers linear speedup but of course, it is limited in the types of analyses for which it applies. Conveniently, a limited version of such embarrassingly parallel approach is already available on sites such as www.bioportal.uio.no [6], and a threaded embarrassingly parallel version has been developed [7]. The embarrassingly parallel version does not speed up a single analysis but it simply speeds up certain experiments by not having to wait before the next experiment is performed. In order to speed up a single analysis, more complicated parallelization attempts are necessary as described in the following section.

3.2 Parallel Chain Approaches

Perhaps the most intuitive approach is to run a separate MCMC chain on each CPU, combining the results. The typical solution in this scenario would be to initialize each chain at a random point and run the burnin period until the chains group together. The maximum speedup in such a setup would be related to the relative size of burnin [8]:

$$Speedup(N) = \frac{b+n}{b+\frac{n}{N}} \; N \underset{\infty}{\longrightarrow} \; \frac{b+n}{b} \tag{1}$$

where b is the burnin, n is the post burnin chain and N is the number of processors. With burnin equal to post burnin period, which is a common ratio used in Structure [1] [9] [10] [11], the result is:

$$Speedup(N) = \frac{2}{1+\frac{1}{N}} \; N \underset{\infty}{\longrightarrow} \; 2 \tag{2}$$

which grows as the relative size of burnin is decreased and offers enough speedup to be worthy of pursuit on a small number of CPUs. Unfortunately, there is a complication in the way Structure infers the populations, which makes it difficult to combine the results of the simulations. The problem is rooted in symmetry of the inferred membership of individuals. For example, when dealing with two evenly divided populations, the optimal answer for Q of Population 1 can gravitate to (1,0) for the two possibilities or (0,1) with equal probability [1]. With a serial version of the program, this is not an issue, since results (or their inverted versions) are equally interpretable by the user. The problem arises when an attempt is made to combine separate chains; it is difficult to ensure that all chains follow the same symmetry.

3.3 Single Chain Parallelization

As an alternative to using multiple chains as the method of parallelization, one could attempt to parallelize a single chain. There are various possibilities in parallelizing a single chain. One is through using prefetching [12] in an MCMC algorithm. Since the next step in the chain in Metropolis-Hastings program is determined from a prior step and accepted with probability, which ensures detailed balance, it could be beneficial to prefetch the next proposals in the situation in which a number of initial propositions are rejected. This could give notable improvements in the run time. However, Structure does not follow a strict Metropolis-Hastings pattern, and only one of the possible updates of the Q parameter uses the Metropolis step, which is set to occur every user-specified number of MCMC steps. As a result, the speedup potential resulting from prefetching the proposals in this specific application would be quite limited.
Due to the unique nature of this MCMC application, Structure does offer some additional opportunities. A unique characteristic of Structure is the use of large arrays of parameters that are inferred at every step of the chain. These parameters are updated at each MCMC step, and have the potential of being divided and computed by separate processes. These key parameters include P - the estimated allele frequencies of the populations, Q - a fraction of estimated membership of an individual to each population, and Z - the unknown populations of the individuals. P is given by a list of real numbers with cardinality equal to the number of loci per individual, multiplied with the number of assumed populations and the maximum number of unique alleles possible. Q is a list of real numbers equal to the number of individuals analyzed multiplied by the number of possible populations. Z is a list of integers, where size is equal to the number of individuals multiplied by the ploidy of the data (number of sets of data for each individual for each chromosome) and the number of alleles.
It is not surprising, considering the structure of these parameter matrices, that as the size of the dataset used grows, the size of the arrays and therefore the amount of computation required grows rapidly. These parameter arrays depend on each other to be calculated, but they do allow for the division of the currently computed array to be computed in parallel.
In essence, the program run time consists of the initial setup, followed by a number of burnin and post burnin MCMC steps and then a final output and terminating processes. The work performed outside of the MCMC steps themselves is quite trivial, considering that an MCMC chain typically has no fewer than 10 000 iterations. In the article, these processes are referred to simply as *other computation*. Inside a single MCMC step, the following are the parameter updating methods performed in the given order: *P*, *Q*, *Z*, α and *Frequency Priors*. At the end, a procedure of collecting and generating summary statistics is initiated. In this article, this collection and generation of summary statistics is referred to as *summary statistics* computation.
For the Structure runs described in this article, the following options were used: α, the admixture parameter was inferred. *Allele frequencies* were correlated among populations. The *probability of data* under the model was computed. An *admixture model* was run in both burnin and post burnin phase. Prior population information was not used to improve the prediction of population membership. The program tested for immigrant ancestry back to grandparents. The *prior probability* that an individual is a migrant was set to 0.01. Uniform instead of gamma prior was used for α parameter. The maximum value for α parameter was set to 20. A randomized seed was used for each run. The frequency of using the Metropolis step to update Q under admixture model was set to 10.
As suggested in Figure 1, at each MCMC step, a set of parallel tasks can be performed and combined for each *P*, *Q* and *Z* parameter ma-

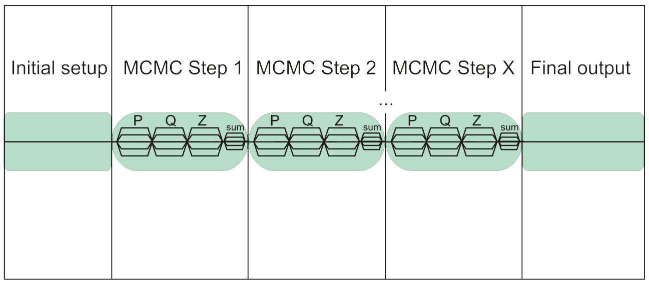

Fig. 1: Parallelizing P, Q, Z and Summary Statistics. Each green bubble represents a single MCMC step. Inside each step, it is possible to parallelize each of the parameter sets, as well as the Summary Statistics. With chain length of thousands of steps, the initial setup and the final output would take a trivial amount of time.

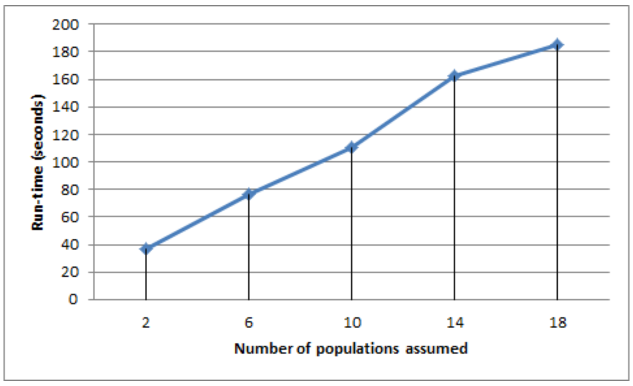

Fig. 2: The change in run time as the software assumes different number of populations. A desktop PC with Intel Core i7-3770 CPU was used. The dataset consisted of 19200 individuals with of 96 loci per individual. 10 000 MCMC steps were used.

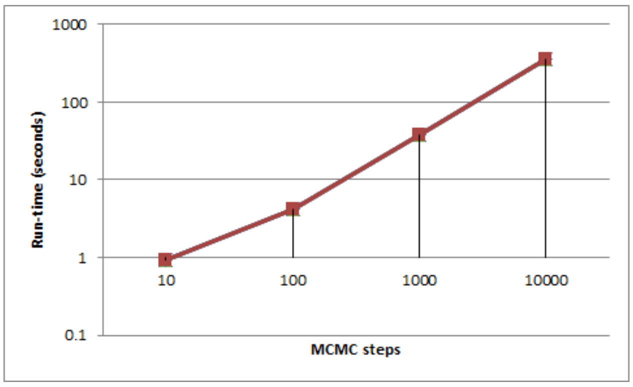

Fig. 3: The change in run time as the number of MCMC steps is increased. A desktop PC with Intel Core i7-3770 CPU was used. The dataset consisted of 19200 individuals with of 96 loci per individual and 2 populations.

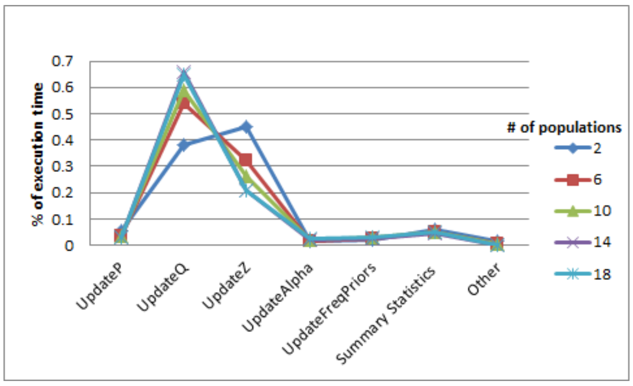

Fig. 4: The percentage of total execution time spent on different sections of the program as the number of assumed populations increases. Results generated from the runs featured in Figure 2.

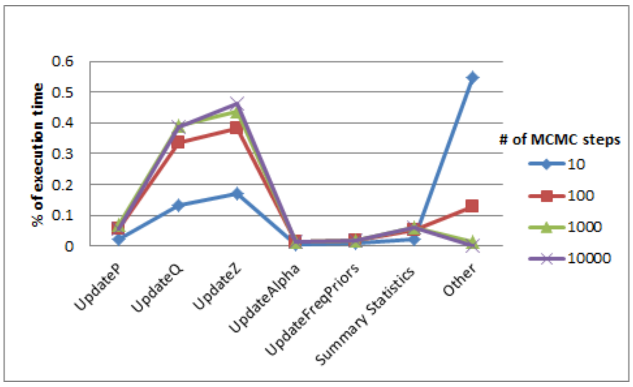

Fig. 5: The percentage of total execution time spent on different sections of the program as the number of MCMC steps increases. Results generated from the runs featured in Figure 3.

trices, followed by non-parallelizable instructions. Another section of the program that offers potential parallelization, is the portion that calculates the *summary statistics* at the end of each MCMC step. In relation to the total run time, the length of each parallelizable loop is quite trivial and the parallelization of these sections is very fine-grained.

It is important to explore precisely what portion of the total run time is spent on computing these parameter matrices and to calculate the parallelization potential. The timing methods from the OpenMP [13] package allows for profiling of the code. They are used to measure the total time spent on computing: P Updates, Q Updates, Z Updates, α Updates, *Frequency Priors* Updates, *Summary Statistics* and the remaining *Other* Computation. The data shown in Figures 2 and 3 depict the behaviour of run time as the number of assumed populations and the number of MCMC steps increases. The run time in relation to the growing number of loci and the number of individuals also grows consistently.

The graphs show an average of ten runs, with the run time growing linearly throughout the parameters that were explored. This includes the data shown in Figure 3, where both the number of MCMC steps and the run time are displayed on a logarithmic scale. The shape of the MCMC graph (Figure 3) is perhaps the least surprising, considering that with each additional MCMC step, the update calculations repeat. This suggests that any parallelization obtained is likely to be consistent across different parameters and datasets, as opposed to being specialized to a particular subset of analyses. Datasets do not usually exceed two thousand individuals [1] [9] [10] [11], but it is important to anticipate big datasets in the future. With the exponential rate at which Big Data is produced [14], obtaining a dataset with more than ten thousand individuals to study the population structure is possible. It is also interesting to note how increasing the assumed number of populations (as seen in Figure 2) also increases the run time linearly despite the fact that the data size remains the same. The calculations made necessary by populations accumulates with each additional population. This implies that it is generally a good idea to avoid overestimating the assumed number of populations. If an exploratory analysis is performed, where at the same time multiple runs with different assumed number of populations is performed, the analyses will finish at a corresponding range of times. Yet, in order to obtain the results, it is necessary to wait until the simulation with the largest number of assumed populations is complete. With a parallelized alternative it might be optimal to run the longest jobs with the help of multiple CPUs, while run the shortest ones on a single core.

The new set of graphs (Figures 4 and 5) explore how the proportions of the total computational time changes as the length of the MCMC chain grows. There are different tendencies that can be observed

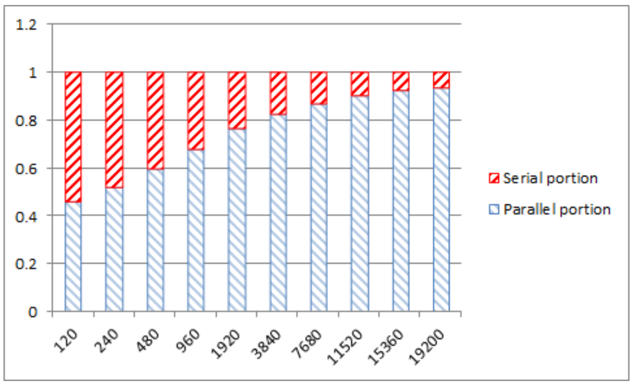

Fig. 6: The serial and parallel fractions of total computation time of a run over increasingly large problem size (number of individuals). As the problem size grows, the parallel portion of the program grows as well.

from the data shown in these graphs. With the growing number of individuals, UpdateP takes an increasing fraction of time up to 60 percent for 19200 individuals. A similar effect occurs with the growing number of loci. This suggests that UpdateP is the most sensitive portion of the program when dealing with different data sizes. Considering future Big Data, UpdateP could be the most important part of the program from the perspective of parallelization. When considering an increase in the number of assumed populations, the UpdateQ method tends to dominate the total run time. Lastly, as the number of MCMC steps increases, UpdateP, UpdateQ and UpdateZ take over 90% of total execution time.

It's not surprising that as the problem size grows, the sections of the code that are not parallelizable in nature (such as the Initial Setup Time and the Final Output) proportionally become smaller. A good example is the *Other* category from Figure 5, which indicates that a very small number of MCMC steps can take more than half of the total computational time. As the number of steps grows to and then beyond any likely values, the *Other* section drops to less than 1% of total computational time. This is because as the number of MCMC steps grows, the *Other* category (the initialization and termination of the program) remains the same. All the other sections are re-run at each additional MCMC step (except *Summary Statistics*, which are not calculated for the initial burnin period). This type of pattern tends to benefit most from parallelization. Since UpdateAlpha (α) and UpdateFreqPriors tend to take an increasingly very small fraction of time as the problem size grows, the OpenMP and MPI [15] overhead may be larger than the benefits obtained from parallelization.

Assuming *UpdateP*, *UpdateQ*, *UpdateZ* and *Summary Statistics* may be perfectly parallelized, then as the number of individuals grows, this portion reaches 93.4% by the largest test as seen in Figure 6. Applying Amdahl's law [16] for a run with 120 individuals using 100 CPUs, the optimal speedup that may be expected equals to:

$$Speedup_{100} = 1/(.535 + 0.465/100) = 1.85 \qquad (3)$$

For a very large run with 19200 individuals as opposed to only 120, the Amdahl's law suggests the following achievable speedup:

$$Speedup_{100} = 1/(0.066 + 0.934/100) = 13.27 \qquad (4)$$

This highlights the limit of parallelization defined by the serial portions of the code. Conveniently, a large portion of Structure execution time does warrant an attempt to parallelize, even if there is a relatively low limit on the number of CPUs that might be efficient to use. Since the parallel portions of the program do in fact grow much faster than the serial portions, then ignoring architectural concerns, as the size of the problem increases, the maximum speedup also increases with the same number of CPUs. It is reasonable then, to explore how the software will perform in a parallel environment in practice.

3.4 Data Generation

Before discussing the implementation and the results, it is important to discuss the dataset used in this article. The output generated by Structure does not produce definitive results. There are various ways in which to infer the accuracy of the answer, but there is no deductive certainty due to the stochastic aspect of the program. While certain values can hint at the correct number of MCMC steps, the minimum number of steps to guarantee the right answer is unknown. It is always possible for the simulation to become trapped in a local minima and as a result to be unable to explore the space sufficiently. One of the ways to ensure the chain is long enough is to run multiple chains until they all tend to give a similar solution. Nonetheless, the precise expected population structure of real-life genetic data is difficult to estimate.

An artificial dataset may solve these problems, by fully controlling the nature of the population structure, and differentiating how long the simulation should run before a correct answer is obtained. An accurate answer could be one where the simulation correctly estimates the population characteristics, such as the population membership. The size of the dataset can also be easily defined. A Python script was used to create such a dataset. The primary dataset generated has 19200 diploid individuals with 96 loci (the data file consists of 19200 individuals, each consisting of two lines of data). Each line of data consists of 96 bits of genetic information that varies at certain frequencies, depending on the population. The various tests throughout this article range from 120 to 19200 individuals, 8 to 256 loci, and 2 to 18 assumed populations. The number of MCMC steps range from 10 to 10 000. This range was chosen both to investigate the behaviour of the serial portion, and the minimum number of MCMC steps required to closely reflect the behaviour of the program as the chain is increased. These numbers were selected to produce a dataset divisible evenly among the processors with even load balancing, to avoid producing misleading results (due to affecting CPU utilization caused by distributing an uneven fraction of work).

Two unique populations are randomly generated and each possesses numerical representations of the genes that follow different normal distributions. The distribution overlap represents the overlap in genes similarity, where the distance of the two distributions is related to the difficulty in telling the two populations apart. The number of individuals, loci, and populations, as well as the similarity between the populations can be easily modified to accommodate any comparative tests that may be required by a given scenario. The run time of the simulation with a given number of MCMC steps does not vary greatly depending on the contents of the data, so the performance improvements remain consistent, regardless of whether the dataset is artificially generated or if it is based on real genetic data.

4. PARALLELIZATION RESULTS

4.1 OpenMP-MPI Hybrid Parallelization

An OpenMP-MPI hybrid parallelization allows the application to take advantage of shared-memory and distributed memory systems as well as various the hybrid architectures, which share both distributed and shared-memory characteristics. Such adaptability would create possibilities for taking advantage of a much wider

```
(...)
#pragma omp parallel for
for (v= 0; v<p_openmp; v++)
        UpdateP(P,Epsilon,Correls,NumAlleles,Geno,Z,
                (int)((NUMLOCI*my_rank)/(p) + NUMLOCI/p/p_openmp*v),
                (int)((NUMLOCI*my_rank)/(p) + NUMLOCI/p/p_openmp*v) +
                NUMLOCI/p/p_openmp);
#pragma omp barrier

        MPI_Allgather(&P[NUMLOCI*MAXPOPS*MAXALLELES/p*my_rank],
                NUMLOCI*MAXPOPS*MAXALLELES/p, MPI_DOUBLE, P,
                NUMLOCI*MAXPOPS*MAXALLELES/p, MPI_DOUBLE,
                MPI_COMM_WORLD);
(...)
```

Fig. 7: MPI-OpenMP Hybrid code modifications of UpdateP method.

```
(...)
#pragma omp parallel for
for (v= 0; v<p_openmp; v++)
        UpdateQMetro(Geno,Q,P,alpha,rep,Individual,
                (int)((NUMINDS*my_rank)/p + NUMINDS/p/p_openmp*v),
                (int)((NUMINDS*my_rank/p +NUMINDS/p/p_openmp*v) +
                NUMINDS/p/p_openmp) );
#pragma omp barrier

        MPI_Allgather(&Q[NUMINDS*MAXPOPS/p*my_rank],
                NUMINDS*MAXPOPS/p,MPI_DOUBLE, Q,
                NUMINDS*MAXPOPS/p,MPI_DOUBLE, MPI_COMM_WORLD);
(...)
#pragma omp parallel for
for (v= 0; v<p_openmp; v++)
        UpdateQAdmixture(Q,Z,alpha,Individual,
                (int)((NUMINDS*my_rank)/p + NUMINDS/p/p_openmp*v),
                (int)((NUMINDS*my_rank/p +NUMINDS/p/p_openmp*v) +
                NUMINDS/p/p_openmp) );
#pragma omp barrier

        MPI_Allgather(&Q[NUMINDS*MAXPOPS/p*my_rank],
                NUMINDS*MAXPOPS/p,MPI_DOUBLE, Q,
                NUMINDS*MAXPOPS/p,MPI_DOUBLE, MPI_COMM_WORLD);
(...)
```

Fig. 8: MPI-OpenMP Hybrid code modifications of UpdateQMetro and UpdateQAdmixture methods.

```
(...)
#pragma omp parallel for
for (v= 0; v<p_openmp; v++)
        UpdateZ(Z,Q,P,Geno,
                (int)((NUMINDS*my_rank)/p + NUMINDS/p/p_openmp*v),
                (int)((NUMINDS*my_rank/p +NUMINDS/p/p_openmp*v) +
                NUMINDS/p/p_openmp) );
#pragma omp barrier

        MPI_Allgather(&Z[NUMINDS*2*NUMLOCI/p*my_rank],
                NUMINDS*2*NUMLOCI/p,MPI_INT, Z,
                NUMINDS*2*NUMLOCI/p,MPI_INT, MPI_COMM_WORLD);
(...)
```

Fig. 9: MPI-OpenMP Hybrid code modifications of UpdateZ method.

```
(...)
*like = CalcLike(Geno,Q,P);
double like2 =*like;

MPI_Allreduce(&like2, &like2, 1,MPI_DOUBLE,MPI_SUM, MPI_COMM_WORLD);

*sumlikes += like2;
*sumsqlikes += (like2) * (like2);
(...)
(Inside the CalcLike method)
#pragma omp parallel for firstprivate (runningtotal, term,
                allele, line, loc, pop) reduction(+ : loglike)
for (ind=0; ind<NUMINDS; ind++)
{
        for (line=0; line<2; line++)
                for (loc=0; loc<NUMLOCI; loc++)
                {
                (...)
                }
                loglike += log(runningtotal); runningtotal = 1;
        }
return loglike;
(...)
```

Fig. 10: MPI-OpenMP Hybrid code modifications of CalcLike method.

range of high-performance computing systems. The following sections explore this option.

The hybrid parallelization using MPI and OpenMP is structured to use both methods similarly. The SPRNG [17] library is used for random number generation. The fundamental idea of the parallelization is to split the work by splitting parameter updates and the summary statistics at each MCMC step. The program divides the work by the product of the number of MPI nodes and the number of OpenMP threads held by each MPI node. The work is then distributed among the nodes and threads. When completed, each node shares its results with all other nodes, so that all nodes contain the solution. There is also additional communication that is made necessary by the distributed-memory architecture.

The parallelized UpdateP method call can be seen in Figure 7. *UpdateP* passes a percentage of values of the loci that need to be processed to each thread. Then *Allgather* method captures the appropriate fraction for each MPI node and passes the calculated work of each of its OpenMP threads to all other nodes. This update is stored in the appropriate location of the P array. *UpdateQMetro* and *UpdateQAdmixture*, the two versions of the method responsible for updating Q parameters are modified, as seen in Figure 8. This is a similar approach as used with *UpdateP*, except the work is divided by the number of individuals, which tends to be larger than the number of loci and thus is preferable. *UpdateZ* works in a similar way to *UpdateQ*, by dividing by the number of individuals but gathering sets of integers as opposed to doubles. The size of communicated fragments between the nodes is smaller for this method. The parallelized version of UpdateZ can be seen in Figure 9. Lastly, besides the *OpenMP* reduction, the *Summary Statistics* loop also requires a reduction between all the nodes to add up the likelihood values. The OpenMP reduction is performed inside *CalcLike*, and an *Allreduce* call is made after calling the method to sum up and then distribute the totals to all nodes. The already reduced likelihood is then used for calculations of summary statistics. An overview of this parallelization strategy is shown in Figure 10.

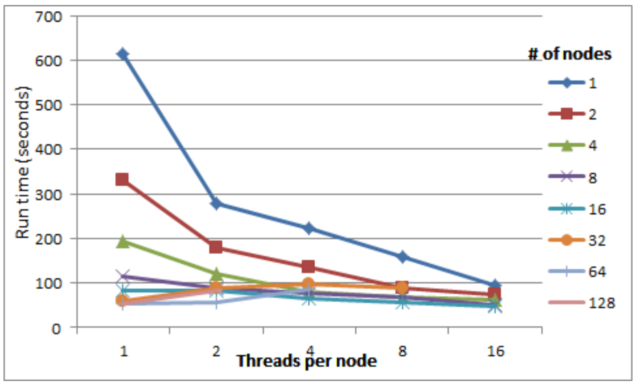

Fig. 11: Run time over different number of threads per node for a various number of nodes. Orca cluster was used. The dataset consisted of 2560 individuals with 256 loci per individual and 2 populations.

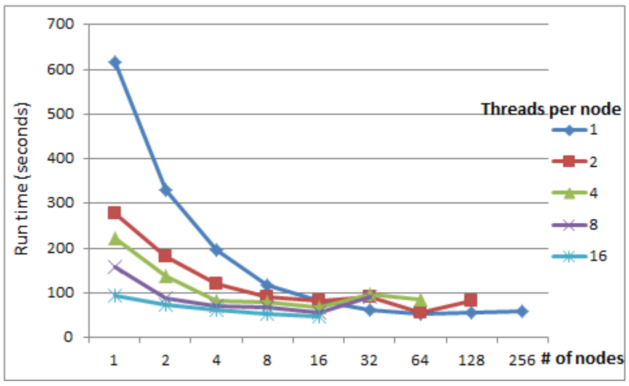

Fig. 12: Run time over growing number of nodes, each running with a different number of threads per node. Orca cluster was used. The dataset consisted of 2560 individuals with 256 loci per individual and 2 populations.

With these code modifications, the behaviour of the program should be the same as the serial version. The new dimension introduced is the possibility of using distributed-memory and shared-memory architectures. Because of this, it is important to explore how the behaviour of the program changes, as the balance between the two changes. A system with few nodes but many threads per node may behave very differently than a system with many nodes but with each running only a few threads.

4.2 Overview of OpenMP-MPI Hybrid Results

The OpenMP-MPI hybrid code was run on the SHARCNET cluster called Orca, which offers hundreds of nodes, each capable of running up to 24 threads. This allowed for a thorough exploration of the performance of the program under various ratios of MPI nodes to OpenMP threads. Since every node has to receive the completed results of all the other nodes, as the number of MPI nodes grows, the percentage of total time spent on communication will grow. The point at which this overhead will become prohibitive is unknown a priori. On a dataset of 2560 individuals and 256 indels, various combinations of runs ranging from 1 to 16 threads and with nodes ranging from 1 to 256 were tested.
The first two graphs (Figures 11 and 12) explore the run times of the program over different number of nodes and different number of

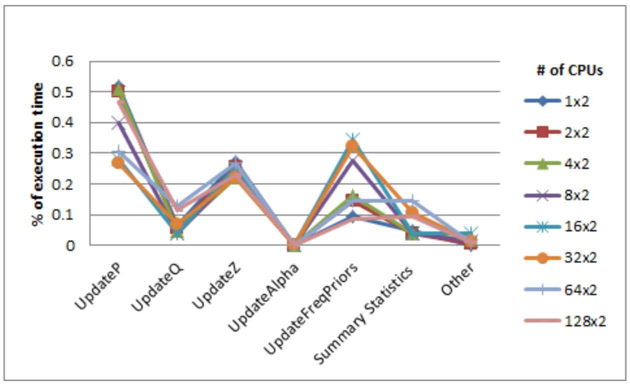

Fig. 13: The percentage of total execution time spent on different sections of the program, as the number of CPUs increases (example: 4x2 - two threads per each of the four nodes). 2560 individuals with of 256 loci used.

threads per node. As expected, the graphs indicate a tendency for the run time to decrease as more threads and more nodes are used. Some of the jobs also tend to lose efficiency particularly when using 34 to 256 nodes, to the point of taking slightly more time to complete with more nodes used. This may be an anomaly, considering that often the following points in the sequence return to lower run times. From the data shown in Figures 11 and 12, it seems that that the best run is obtained by using 16 nodes, each running 16 threads to the total of 256 threads. This is to be expected, considering that it maximizes the total number of CPUs used, while minimizing the distributed-memory communication (only 16 nodes need to communicate with each together).
While no more than 16 MPI nodes are used, the run times shown in the first two graphs seem interchangeable. That is, for up to 16 nodes, the communication between processors does not affect the results. However, beyond 16 nodes the efficiency is reduced due to the increasing serial portion of the program (as seen in Figures 13 and 14) and increased communication. Taking 128 nodes as the point at which communication becomes prohibitive (since the results do not reduce the run time beyond this number of CPUs), it would suggest that roughly splitting to 20 individuals per node (2560/128) and 2 indels per node (256/128) is the granularity at which communication becomes impractical. The run time decreases until 64 nodes, which is the maximum recommended number of nodes with no OpenMP threading. Not surprisingly, the best speedup occurs with the largest number of threads per node, but this performance is decreased with the addition of more nodes. These results suggest that while running on shared-memory architectures is preferred, distributed-memory architectures can produce significant speedups as well. With the performance differences, it is interesting to see the way the run times differ across the different number of nodes, as seen in Figure 13. The percentages of execution time for the parameter updates and summary statistics tend to decrease. *UpdateFreqPriors* method tends to grow, reaching up to 35%. Nonetheless, since the communication overhead between the updates is so high, the *UpdateFreqPriors* method parallelization at best would have a very small window of potential improvement. This suggests that OpenMP parallelization may be justified for this part of the program, if dealing with very large datasets.
Figure 14 illustrates another interesting difference when different distributions of 256 CPUs compute the problem. Since the run times become gradually worse when the 256 threads are spread over increasingly larger number of nodes, this architectural change also

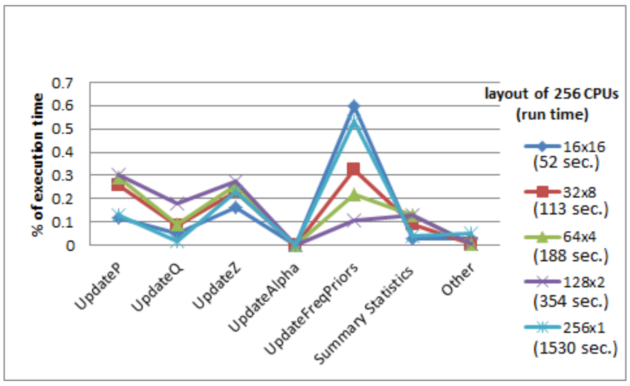

Fig. 14: The percentage of total execution time spent on different sections of the program, as the distribution of the 256 CPUs changes (example: 64x4 - four threads per each of the sixty-four nodes). 2560 individuals with of 256 loci used.

Table 1. : Complete speedup table from all 35 hybrid runs, each using up to 256 total threads.

CPUs/thrds	1	2	4	8	16
1	1	2.21	2.76	3.88	6.51
2	1.86	3.41	4.49	6.93	8.43
4	3.15	5.07	7.56	8.83	9.96
8	5.28	6.85	7.83	9.16	11.80
16	7.45	7.52	9.24	10.82	12.78
32	10.10	6.88	6.35	6.87	-
64	11.71	10.83	7.15	-	-
128	11.19	7.49	-	-	-
256	10.62	-	-	-	-

Table 2. : KarpFlatt values in relation to the speedup and the number of nodes using MPI.

n	2	4	8	16	32	64	128	256
speedup	1.70	2.76	4.34	6.15	7.06	8.78	5.90	0.35
e	0.18	0.15	0.12	0.11	0.11	0.10	0.16	2.90

changes the way in which computational time is spent on the program. The communication overhead is the primary source of difference between the times in the six last graphs. A graph showing data for 16 threads running on each of the 16 nodes suggests the program spends a proportionally longer time on the serial portions and less on parallel portions simply because it is more efficient.

Table 1 summarizes the full results. Particularly interesting is one diagonal (256x1, 128x2, 64x4, 32x8, 16x16), which depicts the case of using 256 threads with various numbers of separate nodes. The best speedup (11.80) is achieved with the use of 16 nodes, each with 16 threads. Interestingly, a pure MPI run of 256 nodes performs well, but the intermediate configurations (128 nodes with 2 threads each, 64 nodes with 4 threads each and 32 nodes with 8 threads each) have significantly lower speedups. A possibility for this may be the combined overhead of MPI and OpenMP, without sufficient number of threads to offset this overhead. Another interesting result is at 32 and 64 nodes, with a single thread per node. These speedups (10.10 and 11.71 respectively) are comparable to those obtained by 256 total threads, and in this scenario may be the best choice if the number of CPUs used is of concern.
At this point, the Karp-Flatt metric [18] (e) may be used to confirm the previous conclusion, looking at the MPI performance without OpenMP threading. As in the previous section, Table 2 shows that up until around 128 and 256 CPUs, e does not grow, indicating that the primary reason for the limited speedup is the sequential part of the program. The growth of e near the end indicates that the main reason for poor speedup is the parallel overhead, whether due to the amount of communication or architectural limits.

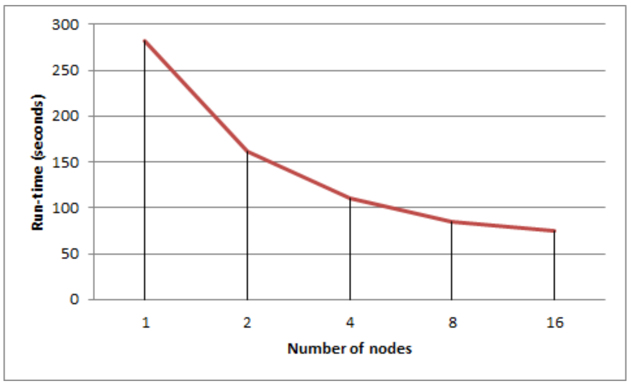

Fig. 15: Run time of a dataset of alpine Arabian burnet moth, over growing number of nodes.

4.3 Application using Real-World Data

An interesting test of the parallelization is to apply a real dataset, to test the performance of the parallelization with a practical example. For this, a dataset [19] of Arabian burnet moth *Reissita simonyi* was used, including 784 individuals from 35 different populations. Following the experimental design from the original article [19], 10 000 MCMC steps were used plus a burning of 1 000. Considering the very large assumed population number (twice as large as the largest tested in previous sections) and the fact that this parameter did not increase speedups when increased, while at the same time keeping the number of individuals relatively low in relation to the number of populations (on average only 23 individuals per population to differentiate between 35 populations), this experiment should test the lower bound of the speedups that can be expected from the parallelization. Figure 15 shows that the results are positive. The run time continues to decline as more MPI nodes were used. The speedup at 2 nodes equals to 1.75 and at 16 threads it equals to 3.78. While the speedup is not as high as with the artificially created dataset from earlier (resulting in the speedups of 1.86 at two nodes and 7.45 at 16 nodes), it is still significant, even with a dataset which does not take full advantage of the parallelization. Certainly, the more individuals and loci per individual in the dataset, the better the performance. If considering a dataset that is a matrix of vertically listed individuals and their horizontally listed loci, then both the number of individuals and the number of loci should not be exceeded by the intended number of CPUs.

5. DISCUSSION

The results show that the parallelization of Structure by dividing the parameter updates and summarizations into multiple processes, both through MPI and through OpenMP, is a sound approach. The speedups with a large number of threads available through SHARCNET supercomputers increase consistently. While the speedups eventually peak, datasets with over 64 loci per individual do occur, and most parallel environments that would likely be used to perform the analysis would be able to produce benefits using all of the CPUs. An important point to consider, is that since the time penalty of increasing the number of loci is reduced, especially if the number of potential threads available is larger than the number of loci (the bottleneck when computing the *P* parameter), this may motivate researchers to increase the number of loci used in their analysis. All other things being equal, increasing the number of loci can improve the accuracy of the results. This may be particularly true for closely related individuals or analyses where data from a large number of individuals is unavailable.

The performance results presented in this article should be of assistance to researchers when deciding what approach to take when using a parallelized version of Structure. If multiple analyses need to be performed, such as with different assumed number of populations, the results indicate that it may be most efficient to run separate jobs on each available node, with each node taking advantage of OpenMP threading to whatever extent it can. On the other hand, if the number of populations is known and only one large run needs to be performed or if a preliminary test needs to be performed before designing the experiment, then the results imply that using all computational resources available is likely to be optimal. In addition, the parallelization allows for adjustment of the experiment to maintain the run time, while increasing the data size or the number of MCMC steps, or to not modify the parameters and simply generate the results quicker. Currently a large dataset analyzed may have 1 000 individuals, and as seen in the profiling tests, the run time increases linearly with the increase in the number of individuals. This suggests that a test consisting of 1 000 individuals run in a serial environment may take a similar amount of time as an equivalent test with 10 000 individuals with as little as 16 threads, using OpenMP or MPI. With the deluge of genetic data, there is an opportunity for new experiments using this software that were previously impractical to explore.

6. FUTURE WORK

With the results indicating that this parallelization is a success, there are many opportunities to extend this work. Most beneficial would be the adaptation of this parallelization to the latest version of Structure, to allow the analyses that depend on the latest extensions to the program to benefit from the speedups. With this comes the potential parallelization of additional loops that can follow the model established in this article. While the latest version of the program fundamentally performs similarly, there are more alternative ways of updating parameters and more parameters to update. Structure allows for a wide range of options and alternative ways to solve the problem and to parallelize all possible paths may require significant amount of work. For the smaller loops such as frequency priors which might not lend themselves to MPI parallelization, OpenMP exclusively may be used to generate additional speedup.

An important addition would be an implementation of intelligent load balancing. The dataset may unevenly split among the CPUs and safe checks are required to ensure that the program optimizes the division of labour, by intelligently dividing the work among the processors. Another unexplored aspect is the question of having more CPUs than rows or columns of data. The easy answer might simply be use an equal or fewer number of CPUs than the number of individuals, due to the inevitable inefficiency. Conveniently, if the number of CPUs is larger than the number of individuals, then the analysis is unlikely to be particularly time consuming, while the smaller number of loci than the number of CPUs only limits the speedup achieved for the calculation of the *P* parameter.

7. REFERENCES

[1] Jonathan K. Pritchard, Matthew Stephens, and Peter Donnelly. Inference of Population Structure Using Multilocus Genotype Data. *Genetics*, 155(2):945–959, June 2000.

[2] Francis S. Collins, Michael Morgan, and Aristides Patrinos. The Human Genome Project: Lessons from Large-Scale Biology. *Science*, 300(5617):286–290, April 2003.

[3] Kristin L. Patrick. 454 life sciences: illuminating the future of genome sequencing and personalized medicine. *Yale J Biol Med*, 80(4):191–194, Dec 2007.

[4] Vicki Pandey, Robert C. Nutter, and Ellen Prediger. *Applied Biosystems SOLiD System: Ligation-Based Sequencing*, pages 29–42. Wiley-VCH Verlag GmbH and Co. KGaA, 2008. ISBN 9783527625130.

[5] Elaine R. Mardis. Next-generation sequencing platforms. *Annu Rev Anal Chem (Palo Alto Calif)*, 6(1):287–303, Jun 2013.

[6] Surendra Kumar, Asmund Skjaeveland, Russell Orr, Pal Enger, Torgeir Ruden, Bjorn H. Mevik, Fabien Burki, Andreas Botnen, and Kamran S. Tabrizi. AIR: A batch-oriented web program package for construction of supermatrices ready for phylogenomic analyses. *BMC Bioinformatics*, 10(1):357+, October 2009.

[7] Francois Besnier and Kevin A. Glover. ParallelStructure: a R package to distribute parallel runs of the population genetics program STRUCTURE on multi-core computers. *PLoS One*, 8(7):70651, July 2013.

[8] Darren Wilkinson. Parallel Bayesian computation. In E. J. Kontoghiorghes, editor, Handbook of Parallel Computing and Statistics, Statistics: Textbooks and Monographs. Marcel Dekker, New York, 2004.

[9] Daniel Falush, Matthew Stephens, and Jonathan K. Pritchard. Inference of Population Structure Using Multilocus Genotype Data: Linked Loci and Correlated Allele Frequencies. *Genetics*, 164(4):1567–1587, August 2003.

[10] Daniel Falush, Matthew Stephens, and Jonathan K. Pritchard. Inference of population structure using multilocus genotype data: dominant markers and null alleles. *Molecular Ecology Notes*, 7(4):574–578,

[11] Melissa J. Hubisz, Daniel Falush, Matthew Stephens, and Jonathan K. Pritchard. Inferring weak population structure with the assistance of sample group information. *Molecular Ecology Resources*, 9(5):1322–1332, September 2009.

[12] Anthony E. Brockwell. Parallel Markov Chain Monte Carlo Simulation by Pre-Fetching. *Journal of Computational and Graphical Statistics*, pages 246–261, March 2006.

[13] Leonardo Dagum and Ramesh Menon. OpenMP: an industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January–March 1998.

[14] Monya Baker. Next-generation sequencing: adjusting to data overload. *Nature Methods*, 7(7):495 - 499, July 2010.

[15] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), May 1994.

[16] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[17] Michael Mascagni and Ashok Srinivasan. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.*, 26(3):436–461, September 2000.

[18] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, May 1990.

[19] Cornelya F. C. Klütsch, Rodney J. Dyer, and Bernhard Misof. Combining multiple analytical approaches for the identification of population structure and genetic delineation of two subspecies of the endemic Arabian burnet moth Reissita simonyi (Zygaenidae; Lepidoptera). *Conservation Genetics*, 13(1):21–37, February 2012.