EPJ Web of Conferences **251**, 02065 (2021) https://doi.org/10.1051/epjconf/202125102065
CHEP 2021

Grid-based minimization at scale: Feldman-Cousins corrections for light sterile neutrino search

Marianette Wospakrik[1,*], *Holger* Schulz[2,3], *Jim* Kowalkowski[1], *Marc* Paterno[1], *Saba* Sehrish[1], *Wesley* Ketchum[1], *Guanqun* Ge[4], *Georgia* Karagiorgi[4], and *Mark* Ross-Lonergan[4]

[1]Fermi National Laboratory, Batavia, IL, USA
[2]Department of Physics, University of Cincinnati, Cincinnati, Ohio, USA
[3]Department of Computer Science, University of Durham, Durham, UK
[4]Department of Physics, Columbia University, New York, NY, USA

Abstract. High Energy Physics (HEP) experiments generally employ sophisticated statistical methods to present results in searches of new physics. In the problem of searching for sterile neutrinos, likelihood ratio tests are applied to short-baseline neutrino oscillation experiments to construct confidence intervals for the parameters of interest. The test statistics of the form $\Delta\chi^2$ is often used to form the confidence intervals, however, this approach can lead to statistical inaccuracies due to the small signal rate in the region-of-interest. In this paper, we present a computational model for the computationally expensive Feldman-Cousins corrections to construct a statistically accurate confidence interval for neutrino oscillation analysis. The program performs a grid-based minimization over oscillation parameters and is written in C++. Our algorithms make use of vectorization through Eigen3, yielding a single-core speed-up of 350 compared to the original implementation, and achieve MPI data parallelism by employing DIY. We demonstrate the strong scaling of the application at High-Performance Computing (HPC) sites. We utilize HDF5 along with HighFive to write the results of the calculation to file.

1 Introduction

Neutrino oscillation is a well-established phenomenon in particle physics, but continues to be a major frontier of study in particle physics. Not only do neutrino oscillation measurements shed light on the fundamental nature of neutrinos – like their mass ordering and CP-violating characteristics – but also can be used to probe the existence of additional neutrino species beyond the three known flavors, called sterile neutrinos [1].

To infer the properties of neutrino oscillations from detector measurements, it is generally necessary to consider statistical tests in a multi-dimensional space. For example, in the case of searches for sterile neutrinos, the oscillation can be considered as a minimal extension to the three neutrinos (3N) paradigm, referred to as the (3N+1) paradigm. It is generally summarized by a two-dimensional confidence region in the $\sin^2(2\theta)$ vs. Δm^2 space [2]. The confidence region is constructed by setting a certain confidence level to a set of parameter values that are compatible with the data. The construction of the confidence region starts from

*e-mail: wospakrk@fnal.gov

selecting a specific statistical test and its resulting test statistic. We can then define a grid over the parameter space and evaluate the test statistic at each point, and then perform parameter fitting at each point to infer the confidence region. This statistical approach, also known as Wilks Theorem [3], is usually sufficient when the "asymptotic distribution" is known and the parameters we are fitting to do not have physical bound. However, when these conditions are not met, it can lead to statistical inaccuracies and serious under-coverage for values of the oscillation parameters that are below the sensitivity of the experiment. In order to evaluate properly the allowed regions in the oscillation parameter space, we use the Feldman-Cousins approach [4], where we generate many toy Monte Carlo to evaluate the distribution of the difference$\Delta\chi^2$ between the χ^2 in each point of the parameter space and the global minimum of the χ^2. This is an expensive computational task that typically involves tens of thousands of points on a grid in the multi-dimensional oscillation parameter space. The computational task becomes even harder to accomplish when considering even higher-dimensional spaces, like in the case of long-baseline oscillation searches which may fit to multiple mixing angles, mass differences, and a possible CP-violating phase δ_{CP}, or in the case of more complicated sterile neutrino models [5].

In this paper, we explore the ability to perform these multi-dimensional grid searches for neutrino oscillation parameter fitting on High Performance Computing (HPC) facilities in order to significantly reduce the computation time. As a direct example, we adapt the SBNFit fitting framework [6], which was developed to perform multi-detector, multi-channel fits using grid search techniques for short-baseline sterile neutrino searches, to HPC facilities, and use its performance as a benchmark for running such algorithms in an HPC environment. The paper is organized as follows: in Section 2 we introduce the datasets and algorithm used for the computation of the Feldman-Cousins correction. We follow in Section 3 with detailed discussions of challenges and solutions to achieve computational performance and scalability. We asses the achieved performance and eventual limitations in Section 4. We make our concluding remarks in Section 5 and discuss prospects of future development.

2 Dataset and Algorithm

In order to perform the test statistics, the Monte Carlo expectation values from each detector are required. Specifically, these comprised of:

1. central value distribution: the expected number of events in the absence of oscillation in each analysis bin, as a function of the parameters to be fitted over.

2. oscillated distribution, **a**: the predicted number of events generated by scaling the central value distribution with oscillation probability associated with given Δm^2, **a**. The oscillation probability is dependent on the neutrino model.

3. pseudo-experiment, **b**: the fluctuated distribution generated by randomizing the oscillated distribution around its mean according to its statistical and systematics uncertainties. Each fluctuation of the distribution is also referred to as "universe".

4. A symmetric, positive definite covariance matrix, **M**, of size $N \times N$, which encodes the correlated bin-by-bin systematics uncertainties from each detector, as a function of the parameter to be fitted over, along with the systematics uncertainties correlation between the detectors.

These input distribution and matrix are stored as ROOT histogram objects [7], the most common data format in HEP. The χ^2 computation allows for a statistical interpretation of the

numerical similarity of **a** and **b**, and include the systematics uncertainties and correlations encoded in **M**:

$$\chi^2 = (\mathbf{a} - \mathbf{b})^\top \cdot \mathbf{M} \cdot (\mathbf{a} - \mathbf{b}) \tag{1}$$

To explore the full parameter space of interest, we perform a rectangular grid-based numerical scan: we determine physically motivated upper and lower bounds for each relevant parameter in the model, and divide the resulting interval in a number of grid cells that are equally distributed. This rectangular grid is then linearized to form a list (G) of the points in the D-dimensional grid. The length of G is referred to as $N_{\text{gridpoint}}$.

The test statistic is computed by generating an ensemble of pseudo-experiments with Monte Carlo simulation at a single point $p \in G$. Each pseudo-experiment is constructed by randomizing the expected number of events in the binned distribution based on their systematic and statistical uncertainties. The number of pseudo-experiments in the ensemble at a given $p \in G$ is referred to as N_{univ}. We compute the χ^2 test statistics for each of the pseudo-experiment in **p** and we compare them to the χ^2 in all other points in the grid to find the minimum $\chi 2$. The difference in the test statistics between the point which gives the minimum χ^2 and the χ^2 in **p** , is calculated to build the $\Delta\chi^2$ distribution for N_{univ}

Algorithm 1 represents a pseudo-code of the most relevant steps. Following the notation in 1, the procedure for a single pseudo-experiments or "universe" is as follows:

function BUILDDELTACHI2($\mathbf{p}, \mathbf{c}, \mathbf{M}, \mathcal{S}, N_{\text{univ}}$)
  $\mathbf{s} = \mathcal{S}(\mathbf{p})$ ▷ prediction at **p**
  for $i \in [0, N_{\text{univ}})$ **do**
    $\chi^2_{\text{min}} \leftarrow$ inf;
    $\mathbf{p}_{\text{min}} = \mathbf{p}$
    a = fluctuation of **s**
    for $\mathbf{q} \in \mathcal{G}$ **do** ▷ Iterate over all grid points
      $\mathbf{b} = \mathcal{S}(\mathbf{q})$
      chi2=CALCCHI2($\mathbf{a}, \mathbf{b}, \mathbf{M}$) ▷ using Algorithm 1
      if chi2 < χ^2_{min} **then**
        $\chi^2_{\text{min}} \leftarrow$ chi2;
        $\mathbf{p}_{\text{min}} = \mathbf{q}$
      end if
    end for
    deltaChi2 = CALCCHI2($\mathbf{a}, \mathbf{c}, \mathbf{M}$) - χ^2_{min}
  end for
  The deltaChi2 values are aggregated and written to disk.
end function

Algorithm 1 χ^2 calculation

function CALCCHI2($\mathbf{a}, \mathbf{b}, \mathbf{M}$)
  $\mathbf{d} \leftarrow \mathbf{a} - \mathbf{b}$
  $\chi^2 = \mathbf{d}^\top \cdot \mathbf{M} \cdot \mathbf{d}$
  return χ^2
end function

where **c** is the central prediction (experimentally measured data). This means that the computational complexity, and therefore the program run time, can be expected to scale $\propto N_{\text{univ}} \cdot N^2_{\text{gridpoint}}$. Measurements of the scaling are given in more detail in Section 4.

3 Computational Challenge and Solution

Listing 1: Implementation of (1) using Eigen3

```
double calcChi(VectorXd const & a,
               VectorXd const & b,
               MatrixXd const & M )
{
   auto const & diff = a-b;
   return diff.transpose() * M * diff;
}
```

Listing 2: Original implementation of (1)

```
double calcChi(vector<double> a,
               vector<double> b,
               vector<vector< double> > M,
               int N)
{
    double chi2 = 0;
    for(int i =0; i<N; i++){
        for(int j =0; j<N; j++){
            chi2 += (a[i]-b[i]) * M[i][j] * (a[j]-b[j] );
        }
    }
    return chi2;
}
```

The original implementation of the framework relies on the frequent access of the file system. ROOT objects are read from the file system every time they are created, and introduce burden on the file system. As an example, to calculate $S(p)$, we have to open two ROOT files object at the same time, perform mathematical procedures on them causing this to be very expensive. The solution to this is to read the input files only once and then use the collective MPI operation to copy the data to all the ranks through memory (see Table 1 for the details of the processors). Despite bringing significant improvement to the performance of the program, this approach however is still limited on the problem size. As the problem size grows with $N_{\text{gridpoint}}$, the required memory also increases and eventually exhausting the system's memory.

Table 1: Processors used in this work

Machine	Cori phase 1 (Haswell)	Cori phase 2 (KNL)
CPU	Intel Xeon E5-2698 v3	Intel Xeon Phi 7250
Clockspeed	2.3 GHz	1.4 GHz
Cores per node	32	68
Ranks per core	2	4

The opening and closing of these ROOT objects such as TH1D also creates a major performance bottleneck due to the many mathematical operations associated with them. Eigen3's VectorXd and MatrixXd [8] inherently employ the vectorization making them better suited

for these operations. Replacing ROOT objects with Eigen3's objects presents a significant performance benefit and eventually outweigh the original gain from caching the computations of the $S(p)$ in memory. The comparison of the single-core prediction rates are shown in Table 2. These changes effectively transform the memory limited program into CPU limited program.

Table 2: Comparison of single-core signal prediction rates. The achieved speed-up allows to avoid having to cache $\mathcal{S}(\mathbf{p})$ and thus eliminate the memory-limitation of the problem size.

	Frequency of $\mathcal{S}(\mathbf{p})$
Original, Haswell	30 Hz
New, Haswell	10 MHz
New, KNL	2 MHz

A further important change to boost the performance of the program is the use of HDF5 [9] as output file format, in conjunction with the header only HighFive library [10] that supports both serial and parallel HDF5 with C++. We utilize the MPI-capable file objects for writing to disk and minimize the necessary MPI communications. The output dataset can be organized into datasets (columns) and groups (tables) and the size is known upon start of the program, making it efficient for parallel I/O and in-memory processing on HPC machines.

Finally, we utilize DIY [11] for data parallelism across processes and nodes, and Eigen3 for linear algebra and vectorized calculations. The application is parameterized as a function of the number of universes N_{univ}, the number of grid points $N_{\text{gridpoint}}$, and the standard MPI option for number of ranks.

4 Performance and Scaling

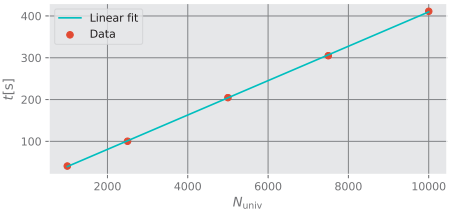
In this section we asses the overall program execution time as a function of the number grid points, $N_{\text{gridpoint}}$, to test our initial hypothesis that the run time should scale $\propto N_{\text{univ}} \cdot N^2_{\text{gridpoint}}$. We further demonstrate strong scaling of the application and predict the maximum size problem that can be solved when running on all of NERSC's Cori [12] for a full day.

4.1 Scaling with N_{univ} and $N_{\text{gridpoint}}$

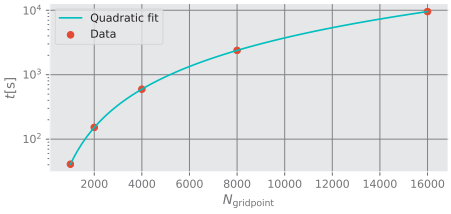
To collect the data, we used a fixed number of Haswell nodes (and therefore ranks) and varied N_{univ} and $N_{\text{gridpoint}}$ independently. Fig. 1a show the scaling of the program time for a fixed number $N_{\text{gridpoint}}$ as a function of N_{univ}. We fit the data linearly and show that there is no visible deviation from linear scaling. Correspondingly, in Fig. 1b we fix the number of universes and compute the program execution time as function of $N_{\text{gridpoint}}$. We fit the data to a quadratic function and shows that there is no visible deviation from the quadratic scaling.

4.2 Single node scaling

We measure the program execution time as a function of the number of ranks to assess benefits gained from using multiple MPI ranks (See table Table 1). To study this, we select a fixed size problem as function of $N_{\text{gridpoint}}$ and N_{univ}, distribute the work among the ranks as evenly as possible, and record the time spent in the main part of the program for each rank separately. The data is shown in Fig. 2a for Haswell nodes and in Fig. 2b for KNL. We benchmark the



(a) Scaling of the program run time with the number of universes, demonstrating a linear dependence on N_{univ}.



(b) Scaling of the program run time with the number of grid points, demonstrating a quadratic dependence on $N_{gridpoint}$.

Figure 1: Scaling of the program time for a fixed problem

scaling using the amount of time spent by the slowest rank (t_{max}) and found that Haswell attains the best single node scaling when using 32 ranks. We observe no significant improvement in the execution time when multi-threading is enabled by utilizing 2 ranks per core on 32 cores. When oversubscribing 1 and 2 cores in the 33 ranks and 34 ranks respectively, we observed an overall worse performance as shown in Fig. 2a. We repeat the measurements on a KNL node and observed similar performance when using non-integer multiples of the number of cores (68) due to ranks competing for resources in an unbalanced way. However, contrary to Haswell, we observe around 16% and 24% improvement in terms of t_{max} when using 2 and 4 ranks per core respectively (see Table 3).

Table 3: Total run time of a fixed problem on a single KNL node demonstrating that there is a mild benefit in using hardware threading.

Ranks	68	136	272
t_{max}[s]	87	75	70
Gain w.r.t. 68 ranks		16%	24%

4.3 Multi node scaling

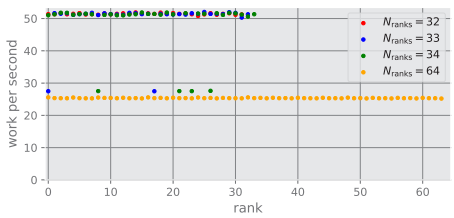
Understanding scalability of multi node systems is crucial to learn how much of a potential performance gain we can expect as we add more computing resources. To study this, we define a reasonable large problem of a fixed size and measure the time it takes to complete the main part of the program as a function of the number of used Haswell nodes (N_{nodes}). Following the finding in the previous section, we use 32 ranks per node in this study.

We define the program execution efficiency as

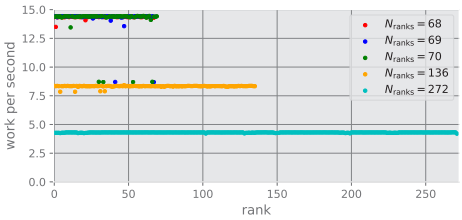
$$\varepsilon = N_{nodes} \cdot \frac{t(N_{nodes})}{t(1)}, \tag{2}$$

where $t(1)$ is the run time on a single core.

Fig. 3 demonstrates that the program scales reasonably well up to the point where the amount of work to be done per rank becomes relatively small. The 80% efficiency drop at 32 nodes (1024 ranks) suggests a non-uniform compute time of individual tasks.



(a) Measurement of the single Haswell node performance for a fixed problem size. The work is distributed among ranks and we measure the utilization of each rank in terms of units of work per second.



(b) Measurement of the single KNL node performance for a fixed problem size. The work is distributed among ranks and we measure the utilization of each rank in terms of units of work per second.

Figure 2: Single node scaling performance

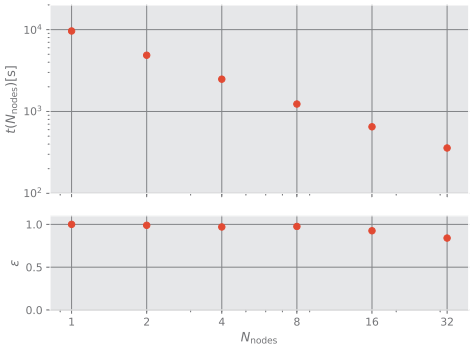


Figure 3: Strong scaling measurements on Haswell nodes. We observe generally good scaling up to the point where the amount of work per rank becomes relatively small.

4.4 Estimation of what is possible

We would like to to estimate the maximal problem size that can be solved on HPC machine like Cori. From 4, we found the scaling to be linear in the number of nodes for our measurements. This allows to estimate an upper limit of of *core computations* if the whole machine were available. For Cori phase 1 (Haswell) we estimate an upper boundary of 6^{10} $s-1$ from a linear extrapolation to the entirety of 2388 nodes. Similarly, we find an upper boundary of 1.8^{11} $s-1$ when using all 9688 nodes of Cori phase 2 (KNL). The data and linear fits are shown in 4. It should be noted that the linear scaling assumption must be considered optimistic and interpreted as an upper limit of the performance of the program.

5 Conclusion

We have adapted a general grid-search-based fitting application that calculates Feldman-Cousins corrections to confidence intervals to run efficiently on current state-of-the-art processor architectures and systems available at HPC facilities. In doing so, we transformed a

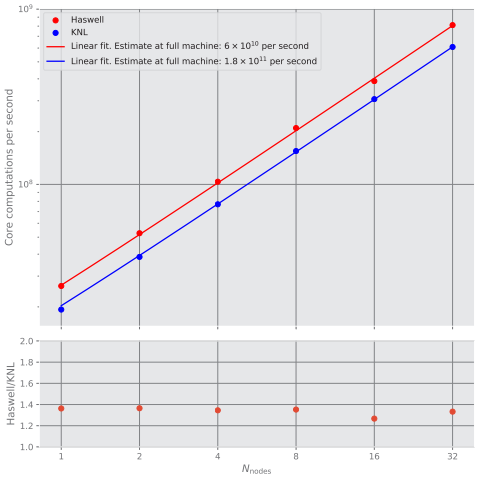


Figure 4: "Velocity" plot showing the "speed" of core computations on different architectures as function of the number of nodes.

High Energy Physics problem common within neutrino physics from a serial-execution program limited by machine memory into an MPI parallel application that scales up to available compute power of a facility. We achieve node-level and thread-level parallelism through DIY, and accomplish significant performance improvements by restructuring algorithms to use Eigen3 for matrix multiplications and array manipulations.

While this work was performed in the context of a specific application (SBNFit) designed for short-baseline neutrino oscillation experiments, the algorithms and procedures in this analysis application are similar to that used in other experiments, and so the techniques employed here are likely to provide similar benefits in terms of performance and design for the broader neutrino oscillation physics program.

5.1 Future Work

In spite of the achieved improvement, we observe that the computational complexity of the Feldman-Cousins approach, coupled with the current brute force grid scan over a discrete grid, severely limits the dimensionality of the problem that can be handled. We are exploring alternative techniques to this grid scan, such as utilizing optimizers to find the global minimum $\Delta\chi^2$, through our connection with the SciDAC FASTmath Institute [13] to move towards seven or more dimensions that the experiment would like to probe, along with approximating the discrete binned data into to a continuous function using the Multivariate Functional Approximation (MFA) model [14].

6 Acknowedgements

Support for this work was provided through U.S. Department of Energy, Office of Science Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, grant "HEP Data Analytics on HPC", No. 1013935.

References

[1] K.N. Abazajian, M.A. Acero, S.K. Agarwalla, A.A. Aguilar-Arevalo, C.H. Albright, S. Antusch, C.A. Arguelles, A.B. Balantekin, G. Barenboim, V. Barger et al. (2012), 1204.5379

[2] M. Antonello et al. (MicroBooNE, LAr1-ND, ICARUS-WA104) (2015), 1503.01520

[3] S.S. Wilks, The Annals of Mathematical Statistics **9**, 60 (1938)

[4] G.J. Feldman, R.D. Cousins, Phys. Rev. **D57**, 3873 (1998), physics/9711021

[5] D. Cianci, A. Furmanski, G. Karagiorgi, M. Ross-Lonergan, Phys. Rev. **D96**, 055001 (2017), 1702.01758

[6] *Sbnfit framework*, https://github.com/NevisUB/whipping_star (2019), accessed: 2020-03-06

[7] R. Brun, F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, in *AIHENP'96 Workshop, Lausane* (1996), Vol. 389, pp. 81–86

[8] *Eigen v3*, http://eigen.tuxfamily.org (2010), accessed: 2020-03-06

[9] *Hierarchical data format, version 5*, http://www.hdfgroup.org/HDF5/ (2020), accessed: 2020-03-06

[10] *Highfive*, https://github.com/BlueBrain/HighFive (2016-NNNN), accessed: 2020-03-06

[11] D. Morozov, T. Peterka (2016), https://github.com/diatomic/diy

[12] *The cori system at the nersc*, https://docs.nersc.gov/systems/cori/ (2016-NNNN), accessed: 2020-03-06

[13] *Fastmath is a program of the office of science within the department of energy*, https://fastmath-scidac.llnl.gov/ (2020), accessed: 2020-03-06

[14] I. Grindeanu, T. Peterka, V.S. Mahadevan, Y.S.G. Nashed, *Scalable, High-Order Continuity Across Block Boundaries of Functional Approximations Computed in Parallel*, in *2019 IEEE International Conference on Cluster Computing (CLUSTER)* (2019), pp. 1–9