EXPLORATION OF PARALLEL OPTIMIZATION TECHNIQUES FOR ACCELERATOR DESIGN[*]

Y. Wang[†], M. Borland, V. Sajaev, ANL, Argonne, IL 60439, USA

Abstract

Optimization through simulation is one of the most time-consuming tasks in accelerator design, especially for high-dimensional problems. We explored several parallel optimization techniques, including parallel genetic algorithm (PGA), parallel hybrid simplex (PHS), and parallel particle swarm optimization (PPSO), to solve some real-world optimization problems. The serial version of the simplex method in `elegant` [1] is used as a benchmark to compare with the newly developed parallel optimization algorithms in `Pelegant` [2]. Our experimental results show almost all of the tested parallel optimization methods converge to an equal or better optimization result compared with the serial simplex method, which indicates the serial version of the simplex method could be easily trapped to a local optimum for some applications. The PPSO method is reliable for global optimization and is well suited for parallel computing. The optimization time for a coupling minimization problem was reduced significantly from more than six hours with serial simplex to about twenty minutes on a BlueGene/P supercomputer at the Argonne Leadership Computer Facility (ALCF). The HPS method requires much less time than other optimization algorithms for an experiment where the optimal solution is very close to the initial point for the optimization. These parallel optimization methods not only make the optimization result more reliable but also provide a feasible approach to on-demand accelerator optimization.

INTRODUCTION

Computational complexity is a prohibitive factor in the optimization of some real-world accelerator design applications. We explored several optimization methods found in the literature and integrated the parallel implementation of the algorithms into `Pelegant` to find the optimal solution for some practical problems. The function evaluation for these optimization problems is achieved through simulations, and the algorithms studied do not require the problem to be differentiable. In this section, we give a brief description of the PHS and PPSO algorithms. An existing parallel genetic algorithm library (PGAPack) was used for the genetic optimization [3]. One strong motivation for this work is the desire to have accelerator optimization tools that take advantage not only of clusters, but also future desktop computers that are expected to have a large number of cores.

[*] Work supported by the U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences, under Contract No. DE-AC02-06CH11357.

[†] ywang25@aps.anl.gov

Parallel Hybrid Simplex Algorithm

While the simplex method is widely used, it can take a very long time to find an optimal solution and may be trapped in a local optimum for some complex problems. We developed the PHS method to achieve a better optimization result in a shorter time, combining ideas from the traditional simplex algorithm and genetic algorithms. The simplex algorithm is very efficient for achieving an optimum by exploring in a limited region, while the genetic algorithm tends to be good at finding a good global optimum. To best take advantage of the existing simplex implementation in serial `elegant`, we chose to vary the step size scaled by a uniform random number for each of the directions to create a new starting point on each of the processors.

The algorithm in pseudo code follows:

Algorithm 1 Parallel hybrid simplex algorithm

```
Start with initial solution on all processors
while $i < max\_iteration$ do
    // For each processor, generate a mutated starting point
    for $d = 1$ to Dimension do
        $step_d^i = r \times original\_step_d \times scale\_factor$
        $x_d^i = xBest_d^{i-1} + step_d^i$
    end for
    Do serial simplex optimization on every processor
    Get the optimal value $yBest^i$ across all the processors
    Broadcast the optimal solution vector $\mathbf{xBest^i}$
    // to be used as the starting point of the next iteration
    if $yBest^i < target$ then
        break
    end if
end while
```

In Algorithm 1, r is a random number with uniform distribution in $(-0.5, 0.5)$. By multiplying r with the original step size ($original\_step$) and adding the result to the previous best solution $xBest$, we can generate a different simplex starting point on each processor within one step size ($step$) of the best solution. This essentially mutates the best solution, as in a genetic algorithm. The variable $scale\_factor$ can be adjusted by the user to control the range within which the simplexes can be generated. After the new point is created, a simplex optimization is conducted on each processor. At the end of every iteration, the optimum across all the processors $yBest$ is recorded and the corresponding solution vector $\mathbf{xBest}$ is broadcast, to be used as the starting point for the next iteration. The procedure continues until either the maximal iteration is reached or the optimal function value reaches the $target$.

Parallel Particle Swarm Optimization

The particle swarm optimizer (PSO) was introduced by Kennedy and Eberhart in 1995 [4]. It has been successfully applied to many optimization problems in different fields. The algorithm is based on a metaphor of social interaction. At the start, a group of agents, called particles, are randomly spread through a region. Each individual particle stochastically moves toward the best position it has experienced and the best position all the particles have reached.

Algorithm 2 Parallel swarm algorithm

```
// Initialize population
$population\_size = population\_total/n\_CPUs$
for $i = 1$ to $population\_size$ do
  $\mathbf{x^i} = \mathbf{xLow} + \mathbf{r} \times (\mathbf{xHigh} - \mathbf{xLow})$
  $\mathbf{p^i} = \mathbf{x^i}$
end for
while $k < max\_iteration$ do
  // Update the best result for current iteration
  for $i = 1$ to $population\_size$ do
    if $f(\mathbf{x^i_k}) < f(\mathbf{p^i_k})$ then
      $\mathbf{p^i_k} = \mathbf{x^i_k}$
    end if
    if $f(\mathbf{p^i_k}) < f(\mathbf{p^g_k})$ then
      $\mathbf{p^g_k} = \mathbf{p^i_k}$
    end if
  end for
  $\mathbf{p^g_k} \leftarrow min(\mathbf{p^g_k})$ across all the CPUs
  if $\mathbf{p^g_k} < target$ then
    break
  end if
  // Generate coordinate for next iteration
  for $i = 1$ to $population\_size$ do
    $\mathbf{v^i_{k+1}} = w_k \mathbf{v^i_k} + c_1 r_1 (\mathbf{p^i_k} - \mathbf{x^i_k}) + c_2 r_2 (\mathbf{p^g_k} - \mathbf{x^i_k})$
    $\mathbf{x^i_{k+1}} = \mathbf{x^i_k} + \mathbf{v^i_{k+1}}$
  end for
end while
```

In Algorithm 2, the entire population is distributed to each of the processors evenly at the beginning. Then they are initialized with uniform random numbers within the range for each of the directions. **xLow** and **xHigh** are the vectors to store the lower and upper limits of the optimization variables. The position of the ith particle is represented with $\mathbf{x^i}$. $\mathbf{p^i}$ is the best position that the particle has experienced and $\mathbf{p^g}$ is the best position for the entire population among all the processors. $\mathbf{v^i_{k+1}}$ is the velocity for the ith particle in the $(k+1)$th iteration, which consists of three parts: inertia $w_k \mathbf{v^i_k}$, personal influence $c_1 r_1 (\mathbf{p^i_k} - \mathbf{x^i_k})$, and social influence $c_2 r_2 (\mathbf{p^g_k} - \mathbf{x^i_k})$. A linear reduction of inertia weight [5] w_k is used to allow the exploration to evolve from a global search to a local search gradually, where w_k is $w_{max} - (w_{max} - w_{min}) \times k/max\_iteration$ for the kth iteration. The weight is reduced from $w_{max} = 0.9$ at the beginning to $w_{min} = 0.2$ at the end of optimization. This parameter has improved the convergence significantly compared with the original algorithm proposed in [4]. c_1 and c_2 are constants, called cognitive and social parameters, respectively. r_1 and r_2 are random numbers with uniform distribution between 0 and 1. The computed velocity is added to the particle's current position $\mathbf{x^i_k}$ to be evaluated in the next iteration. The procedure continues until the termination criteria are met.

APPLICATIONS AND RESULTS

One of the practical problems we studied is minimizing the coupling of the Advanced Photon Source (APS) by adjusting the strength of the 19 skew quadrupoles. This can be performed using a response matrix method and singular value decomposition to minimize the cross-plane response matrix and vertical dispersion. However, this method frequently performs poorly when assessed in terms of the vertical beam size at the source points, a result of the small number of skew quadrupoles. Instead, we minimized the sums of the squares of the vertical beam size at the source points. A special requirement of this problem is that the optimization should be completed within half an hour for online tuning. It took more than 6 hours for the serial simplex method to reach an optimum of 0.057 after 3600 function evaluations on a single AMD Opteron 2.4 GHz CPU. To reduce the optimization time, we first tried the hybrid simplex method, but it took more than an hour to finish one iteration of the simplex simulation, which requires several hundreds of function evaluations. The PSO method needs as little as one function evaluation per processor for one iteration, which makes the algorithm very suitable for parallel computing. The frequent information exchange between the agents makes the optimization result converge much faster for this problem.

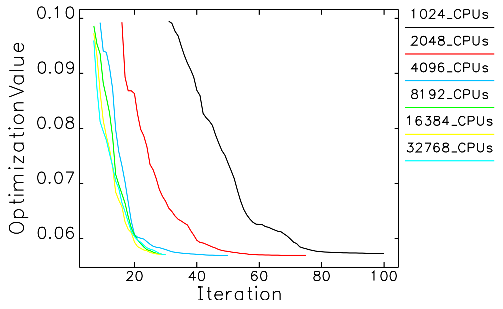

Figure 1: The results of parallel particle swarm optimization with different numbers of CPUs and individuals.

Figure 1 shows the coupling minimization result with different numbers of CPUs/individuals on the ALCF's Intrepid supercomputer. To better compare the convergence speed for different configurations, the graph displays only the part where the minimization value is less than 0.1. The initial particles were distributed within the given range randomly, assuming we don't have any previous knowledge for the location of the optimal solution. As a large CPU pool is available on this computer, we can set the population size to be the same as the number of CPUs. Only one function evaluation per iteration on a CPU is needed for all these experiments. We find that the results with different numbers of particles converge to a result close to the optimum 0.057 found with the serial optimization program. With a larger number of individuals, the algorithm converges faster to the optimal value. The total time is roughly in portion to the number of iterations, or the number of function evalutions per CPU. It took about 28 minutes to finish 50 iterations with 4096 individuals on 1024 compute nodes (4096 cores), which is one rack of the total 40 racks of the supercomputer. With 4096 compute nodes (16384 cores), it took about 20 minutes to finish 30 iterations to reach the same result. More than 8k nodes doesn't give faster convergence for this problem. Using 4k individuals on 1k compute nodes appears to be an efficient choice.

Another application is to optimize the Twiss parameters for a complex configuration of the APS, using 38 independent quadrupoles. The challenge of this problem is not only the high dimension, but also that it requires very fine tuning around a very small neighborhood to reach the target value, which is set to be 1 for this optimization problem. All three parallel optimization implementations were tested with 1k compute nodes (4k cores) on Intrepid. The sizes of the populations are the same as the number of CPU cores, so only one function evaluation per iteration on each CPU is needed for the parallel particle swarm optimization and genetic algorithm. We can use the number of function evaluations to decide which algorithm is most efficient for this problem.

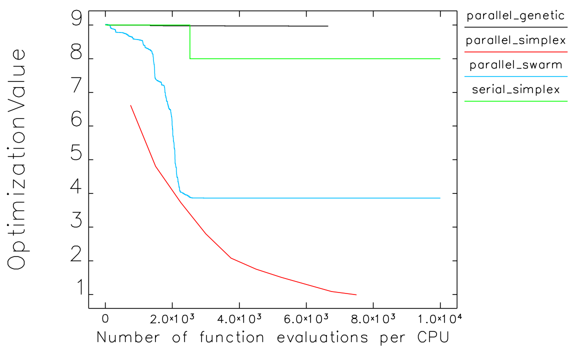

Figure 2: The Twiss parameter minimization problem with different algorithms.

Figure 2 shows the optimized function value versus the number of function evaluations per processor. It took more than 6 hours for the serial version to reach 4.27 after 264k function evaluations on a computer with an Intel Xeon 2.3 GHz CPU. Both the parallel genetic optimization and particle swarm optimization failed to reach the target value after thousands of evaluations. The parallel hybrid simplex method converged to the target optimum after 7.5k function evaluations, which took about 1.5 hours with 4k Intrepid cores. The optimal function value for the hybrid simplex method was recorded after every simplex call, which had about 750 function evaluations for this experiment. The optimal solution turns out to be very close to the given starting point, which indicates that an efficient search in a local neighborhood is the key to reaching the target value for this problem. (A successful application of the PGA algorithm is shown in [6].)

CONCLUSION

At present, computer manufacturers seek to increase computer performance by adding more cores instead of increasing processor speed. The traditional, serial optimization techniques used by accelerator design programs will be unable to take advantage of this. Hence, we implemented several parallel optimization algorithms and applied them to some practical accelerator design problems. It is not easy to find a single algorithm for all problems. The parallel particle swarm optimization and parallel genetic optimization have the advantage in global optimization where no previous information of the optimum is required. They also require a very few number of function evaluations before all the processors exchange their best result to get the optimal result faster. Applying adaptive step-size control for each of the dimensions would improve the performance of the parallel genetic optimization. The hybrid simplex algorithm takes advantage of both the local optimization ability of the simplex optimization and the mutation technique from the genetic optimization. It is good for the applications, which only need some fine tuning, but it cannot meet the on-demand adjustment requirements because of the large number of function evaluations. The parallelized simplex algorithm in [7] could be used to reduce the optimization time in the future.

ACKNOWLEDGMENTS

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

REFERENCES

[1] M. Borland, Advanced Photon Source Light Source note LS-287, Sept. 2000.

[2] Y. Wang et al., Proc. ICAP09, p. 355 (2009).

[3] D. Levine, ANL Report ANL-95/18 (1996).

[4] J. Kennedy and R. C. Eberhart, Proc. of IEEE International Conference on Neural Networks, p. 1942 (1995).

[5] Y. Shi and R. C. Eberhart, Proc. of IEEE International Conference on Evolutionary Computation, p. 69 (1998).

[6] C. Wang et al., Proc. IPAC10, p. 4605 (2010).

[7] H. Shang et al., Proc. PAC05, p. 4230 (2005).