Parallel Algorithm for Combination of SQP and PSO

En-Cheng Chang
Department of Industrial Engineering and engineering Management
National Tsing Hua University, Hsinchu, Taiwan
Tel: (+886) 3-5717654, Email: p2921185@gmail.com

Yu-Ching Lee
Department of Industrial Engineering and engineering Management
National Tsing Hua University, Hsinchu, Taiwan
Tel: (+886) 3-5717654, Email:yclee@ie.nthu.edu.tw

Abstract. The optimization technique has been developed in several ways, but as the amount of data increase, the optimization solvers' speed has become an important issue. Therefore, in this paper we first evaluate the role of probabilistic methods and SQP in the solution technique of optimization problems, then we apply SQP as the main solver to solve constraint optimization problems. To build the solver program, we take python as the major language and use the message passing interface (MPI) library to parallel the original sequential algorithm. In order to measure the speedup after parallelism, this research will implement both the sequential algorithm and the parallel algorithm in multiple clusters system and then conclude the efficiency of parallel computing in solving the large scale problems.

Keywords: Nonlinear programing, Parallel algorithm, Sequential quadratic programing, Particle swarm optimization, Message passing interface

1. INTRODUCTION

Solving optimization problem is one of the core technologies in the modern industries where the operations research methods are utilized and in artificial intelligence where the machine learning techniques, e.g., pattern recognition and prediction, are employed. As the mechanisms people wish to formulate become more sophisticated than ever, the required size of variables and constraints in the mathematical programming formulation for a real system have enlarged significantly. Generally speaking, there are two directions of the algorithmic development for solving the large-scale optimization problems. The first direction focuses on the improvement of the serial algorithms running on a single machine which ideally is a high-performance computer, whereas the second direction focuses on the development of the distributed (parallel) algorithms performing subroutines on a cluster of the computers. In this work, we study an algorithm from the perspective of the second path for solving the nonlinear constrained optimization problems.

We propose a parallel scheme for the popular nonlinear constrained optimization algorithm known as the Sequential Quadratic Programming (SQP henceforth). It is widely recognized that the improvement of runtime of a distributed optimization algorithm compared with its serial counterpart is restricted. It is because of the excessive demand of information passage arising from stationarity verification, feasibility confirmation, and improvement computation (in terms of a search direction) in any convergent optimization algorithms including SQP. On the other hand, performing-optimization algorithms include heuristics and meta-heuristics (without convergence result) can be greatly parallelized and the implementation of parallelism is relatively simple. Motivated by this, we adapted a meta-heuristics known as the Particle Swarm Optimization (PSO henceforth) to the distributed computing environment and embedded the parallel PSO in the SQP method.

The implementation of the proposed parallel algorithm is done in MPI. This algorithm is compared with (i) the serial PSO, (ii) serial SQP and PSO combination, and (iii) parallel PSO, based on three quantities (speedup, efficiency, and accuracy.) In the scope of this paper, the numerical result is primarily obtained from the small-size instances. The experiment results show that the parallel SQP and PSO combination performs well in both the speedup and

the efficiency. These results act as evidence showing that the proposed design has potential in solving large-scale optimization problems in parallel while the convergence to stationarity remains.

1.1 Sequential Quadratic Programming

Sequential quadratic programming is an optimization technique for solving nonlinear programming. Consider a minimization problem of the form $\min_x \{ f(x) \mid g_1(x) \ldots g_i(x) \leq 0 \}$. The SQP method translates the original problems into exact penalty problems as follows:

$$\min_{x} f(x) + cP(x)$$
$$P(x) = \max \{ g_i(x) \ldots g_j(x) \} \qquad (1)$$

Then, we switch to solve the necessary condition of (1). The condition is formed as a subproblem at an x as in below:

$$\min_{d} \nabla f(x)' d + \frac{1}{2} d' H d$$
$$\text{s.t} \quad g(x) + \nabla g(x)' d \leq 0 \qquad (2)$$

In this subproblem, we solve for the best improvement direction instead of solving for the variables x. After the improvement direction is found, we can judge whether the x vector is a stationary point or not. If the direction doesn't equal zero, it means that the solution of the variables x still needs to be updated. The next point will be updated by the current point and the computed direction d. Conversely, if the direction equals zero, it represents that the current point is a stationary point and may be the solution of the original problem (Bertsekas, 2016).

Every time before we solve program (2), we need to calculate the values of all the constraints at the certain solution, and then pick the constraint with maximum value as max constraint P(x) to form the subproblem (2). If the amount of constraints is huge, the computers will take a lot of time to calculate the maximum value of each constraint. Hence, we consider dividing the constraints into different processors and calculating them synchronously.

1.2 Parallel Computing Frameworks

To parallel the original algorithm, there are a lot of tools that can be utilized, for instance, OpenMP and MPI. OpenMP is usually employed in a shared memory system, and in such a system, most variables are open variables. Thus, different threads can share the variable immediately and can calculate data quickly (Kang et al, 2015).

MPI is usually used in multiple clusters system, but it can also be implemented in a single memory system. The delivery medium of MPI is the high speed internet. Even though MPI is fast in running speed, it is generally not faster than OpenMP.

Compared with OpenMP and MPI in the normal-size problems, which can be afforded in a single memory system, OpenMP is the best tool because its speed is three times faster than MPI. However, if the size of the problem is so large that one computer can't handle, MPI is the only option. In our other research, we face a problem which can't be afforded in a single memory system. Thus, we tend to apply MPI as our main tool.

1.3 Measures for Parallel Computing Performance

The most important aspects of parallel computing are the speedup and the efficiency. The benchmark for the runtime of a parallel computing procedure is that follows the Amdahl's law, which is shown below:

$$T_P = \frac{aT_S}{P} + (1 - a)T_S \qquad (3)$$

T_P represents the runtime of the parallel algorithm. T_S represents the runtime of the sequential algorithm and a here represents the proportion of parallelism. The formula (3) indicates that the runtime of parallel program should approach the parallelism proportion of sequential algorithm, which pulps the number of processors P and then adds non-parallelism proportion of sequential algorithm time.

The speedup of parallelism can be evaluated by the following formula:

$$S = \frac{T_S}{T_P} \qquad (4)$$

We hope that the speedup S can be almost equal to the total number of processors P, but it's very difficult to achieve for two main reasons.

First, parallel computing has a limit. According to formula (3) and formula (4), if we keep increasing processors as computing node into clusters system, we can get a formula as follows:

$$\lim_{P \to \infty} S = \frac{T_S}{\frac{aT_S}{P} + (1-a)T_S} \qquad (5)$$

Most parallel algorithms take ninety percent as a high level of parallelism. Based on formula (5), the speedup limit will be equal to the reciprocal of non-parallelism proportion. Therefore, the limit of the speedup is difficult to be over ten times. It means that attaining a higher proportion of parallelism is an important issue in parallel computing.

Second, in parallel computing, we sometimes need a master computer as a center node which is responsible for integrating the whole numerical result. In such a structure,

the limit of speedup S will be P-1, and thus it is impossible to have the speedup equal to the number of processors P. Therefore, we use formula (6), which represents the acceleration per processor, to evaluate the parallelism,

$$E = \frac{S}{P} \qquad (6)$$

Above all, we comprehended the factors affecting the parallel efficiency. The major factor is the parallelism proportion. In the light of our preliminary experiment results, the most convenient way to parallel over ninety percent is to partition the data. If the amount of data can be partitioned into P parts and be respectively assigned to different processors, then the load of each processor can be thus reduced to 1/P. Among the parallel nonlinear programing solvers, the probabilistic methods can especially perform such a partition (Dixon and Jha ,1993). For instance, multistart method can respectively start from several points by different processors and find local minimum asynchronously to reach its stop criteria. Particle swarm optimization can divide the swarms into several processors uniformly, each processor is in charge of its own part. Thus, the load of each processor decreases significantly. (Chang et al, 2005).

Multistart is generally used to find the global solution. However, PSO can be implemented to find both the local solution and the global solution based on the parameter it is set. Hence, we consider PSO to be more flexible and more suitable to be combined with SQP to develop a new algorithm (SQP_PSO henceforth), and parallel it to test the value of the new parallel algorithm.

2. METHOD: SQP_PSO Combination

The parallel algorithm of PSO was published in the last few years. According to previous research, we consider PSO to be valuable in parallel computing. However, when we tried to apply PSO as the main parallel algorithm, we discovered some disappointing issues. In constrained problems, the particles need to test whether they are in the feasible region or not. Hereby, when the constraints increase, the running times for checking the feasibility for every constraint also increase. It thus takes much time.

For this reason, we considered importing SQP method, which performs exact penalty function and can remove redundant constraints temporarily. After removing constraints, we assume PSO can be used as the main solver of subproblem in effect (Parsopoulos and Vrahatis, 2002).

After the model of SQP_PSO is built preliminarily, we divide the work of calculating the maximum values of constraints into several processors. Each processor will handle equal amounts of constraints. After all the processors have completed their work, the multiple clusters system will return the data we want by the function we use. MPI_Send is a point-to–point function, it may cause critical section when all the slave processors send the maximum value synchronously. However, MPI_Reduce is a collective communication function which can avoid critical section. Hence, we tend to apply function MPI_Reduce to obtain the constraint with max value instead of MPI_Send (P. Pacheco, 2011). The idea of this communication can be comprehended by Figure 1.

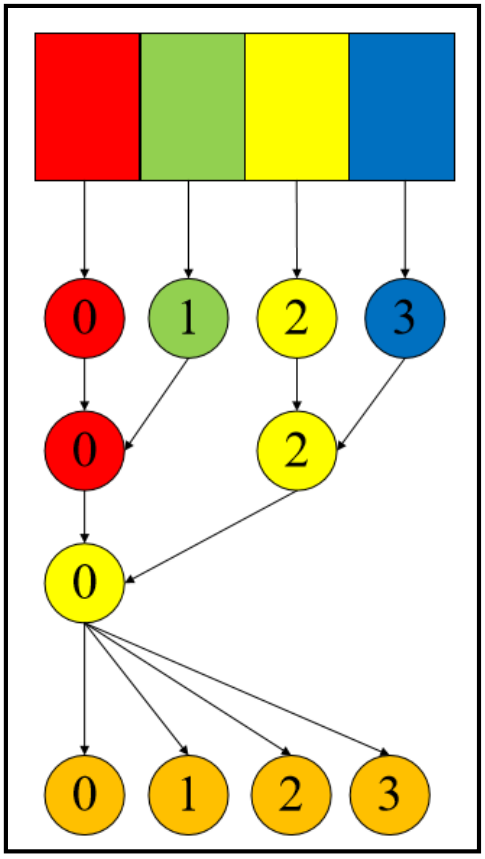

Figure 1. The model of parallel SQP

In Figure 1, different colors represent different constraints, and all the constraints are imported into each processor automatically. After calculating the values of the constraints, all the processors reduce the value we want into the processor zero. The processor zero then soon broadcast the constraint with maximum value into all the slave computers by function MPI_Bcast and the subproblem will be built in each processor.

The approximate runtime in PSO is influenced by the amount of particles and the iteration times. To maintain the accuracy of PSO, we can't decrease the amount of particles and the iteration times below a certain level. We thus decided to divide particles into P-1 parts, and assign them into P-1 processors. If the calculation data has been divided first, the workload in each processor can be reduced significantly.

Figure 2 represents the above opinion

In Figure 2, particles are assigned equally into P-1 slave processors. When all particles start to move, the direction and velocity are dependent upon its inertia, self-experiment and global experiment. Therefore, every time when the best solution in each processor is produced, the slave processors need to send the position and related information of the best solution to the master computer. The master computer then integrates the particles information it receives, and sends the best solution to all slave processors after arrangement.

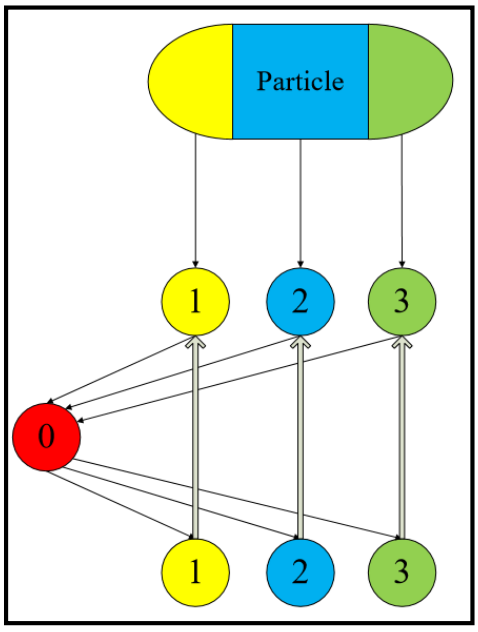

Figure 2. The model of parallel PSO

With Figure 1 and Figure 2, we can combine parallel SQP and parallel PSO. Figure 3 is the combination of it. In Figure 3, we set the processors zero as the master computer to monitor the process of PSO. After the subproblem is solved, if the variables of the original problem still need to be modified, the master computer will broadcast the solution information to all the processors including itself. Then, all the processors update the numerical data and recalculate the value of each constraint to build a new subproblem until the stationary point is obtained.

We switch the role of the processors zero because there is no need to monitor the SQP part. To change the processor zero's role here can increase the number of computation nodes, it thus becomes more valuable.

We write all the work in the same program, thus all the processors can run the same program in parallel. Figure 4 is the main algorithm of it. Before starting to run, all the processors will be assigned a rank. Rank zero represents the master computer, if the rank is greater than zero represents that it's a slave computer.

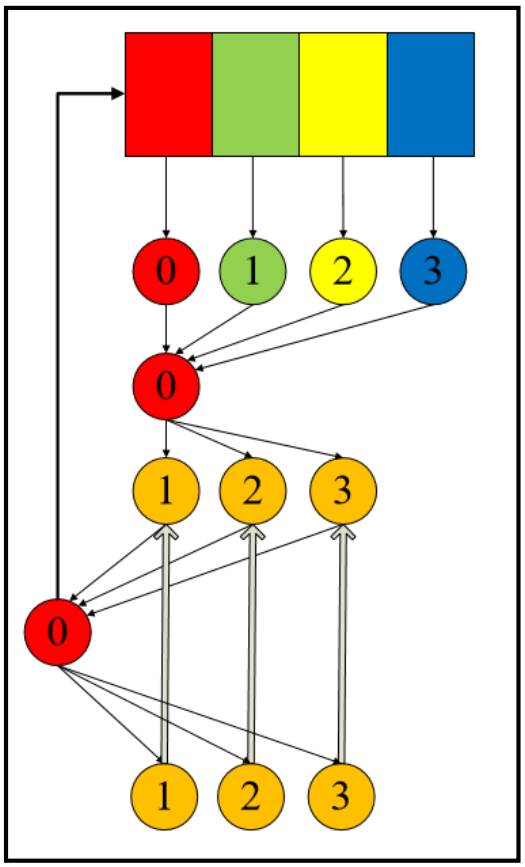

Figure 3. The model of parallel SQP and parallel PSO

```
Step 0.   Initialize
Step 1.   Import constraints
Step 2.   Do
              1. Calculate the value of each constraint independently
              2. Reduce max value to master processor
          If rank==0
              Broadcast subproblem
Step3.    If rank !=0
              1. Solve subproblem by parallel PSO
              2. Send best position to master processor
              3. Receive gbest position from master processor
          If rank ==0
              1. Receive local best position from master processor
              2. Arrange data
              3. Send gbest position to all slave processors
Step4.    If direction in subproblem all equal to zero
              Return value of local minimum
          Else
              Update new position and go back to step 2
```

Figure 4. The algorithm of parallel SQP and parallel PSO

3. IMPLEMENTATION

After finishing building the model of the new algorithm, we start to program. However, we encountered some problems which needed to be studied and researched for a while.

3.1 Computation Error

There exists a floating point calculation error in machine language, especially in different languages. We tried to program the sequential program in two languages, such as MATLAB and Python. In both languages, the direction we obtained from the subproblem can't equal zero. The reason for this is possibly a floating point error. Therefore, if we set stop criteria as direction should equal zero then break the loop, the iteration will not stop but keep calculating. This error (jumping problem thereafter) usually causes the solution of variables to be over improved and repeat calculating subproblem instead of falling into a stationary point.

In MATLAB we define the sum of the square of all directions as error. If the error is less than 10^{-4}, then the program terminates. After this revision, the jumping problem which was caused by a floating point error can be solved and the objective value can also be converged to local minimum smoothly.

In Python, we also define the sum of the square of all direction as error, and set the error as the stop criteria. However, the jumping problem still exists. We thus decide to focus on the direction. If the direction continues to decrease and the objective value is approaching a better value, we tend to judge that it's going to arrive at a stationary point, then we force it to stay on the same constraint and do not improve to another constraint.

In this way, although we avoid a jumping error, there is still another difficulty. Because we force it to stay on a specific constraint, essentially its improvement may sometimes be interrupted. Hence, in this design, the solutions we found are restricted and may conflict with other constraints which means the solution is infeasible.

Thereafter, we added a second condition to the program. When the value of the subproblem is consequently improving, the variables can fit all the constraints and the direction approaches zero, then the program terminates. By this setting we can find the feasible solution which is close to stationary point. Therefore, we can temporarily apply this algorithm to estimate the speed of SQP_PSO and parallel it. However, we don't completely approve of this modus operando, so in the future, we will keep ameliorating this algorithm.

4. EXPERIMENT

We divided our experiments into two parts. The first part is to test the sequential algorithm on a single memory system. The sequential algorithms include the serial PSO and the serial SQP_PSO. We generated two problems, one of the optimization problems is set to be a two dimensional problem, i.e. optimization problems with two decision variables. Another one is set to be a three dimensional problem, i.e. optimization problems with three decision variables. Each synthetic problem contains either one hundred constraints or three hundred constraints. The environment we prepared for testing the runtime and object value is as follows:

- ✧ O.S. Windows 10
- ✧ CPU Intel core i5-2450M 2.50GHz
- ✧ RAM 12GB

The second part is to test the parallel algorithm in multiple clusters system. The parallel algorithm contains parallel PSO and parallel SQP_PSO. The testing optimization problems and the working environment are the same with the setting for the sequential algorithm.

After these two experiments were done and repeated 10 times. We can use the average data to analyze four issues as follows:

1. Dimension affects solver
2. Constraints amount affects solver speed
3. Accuracy of new algorithm
4. Efficiency of parallelism

Finally, with all numerical data results, we can conclude the evaluation, applications and the improvement methods in the following section.

From the experiment, we obtain sets of data for discussion to consider whether the algorithm we developed is acceptable or not. In the following subsections, we analyze the performance of parallel PSO and SQP_PSO. This research will compare the accuracy of objective values in different solvers and possible improvements.

4.1 Results for Parallel PSO

As we mentioned in section 2, PSO is a powerful stochastic performing-optimization technique. However, as the amount of the constraints increase, the efforts of checking the feasibility of the solution at the current iteration also increases. Hence, the runtime of PSO gets significantly slower. In Figure 5 and Figure 6, we present the runtime of serial PSO and parallel PSO in each dimension of variables with different quantities of constraints. According to Figure 5 and Figure 6, we can observe that, the runtime of parallel PSO decreases along with the increase of the number of processors.

Table 1 contains the average value of all the data we obtained from the experiments. We use the speedup (see as

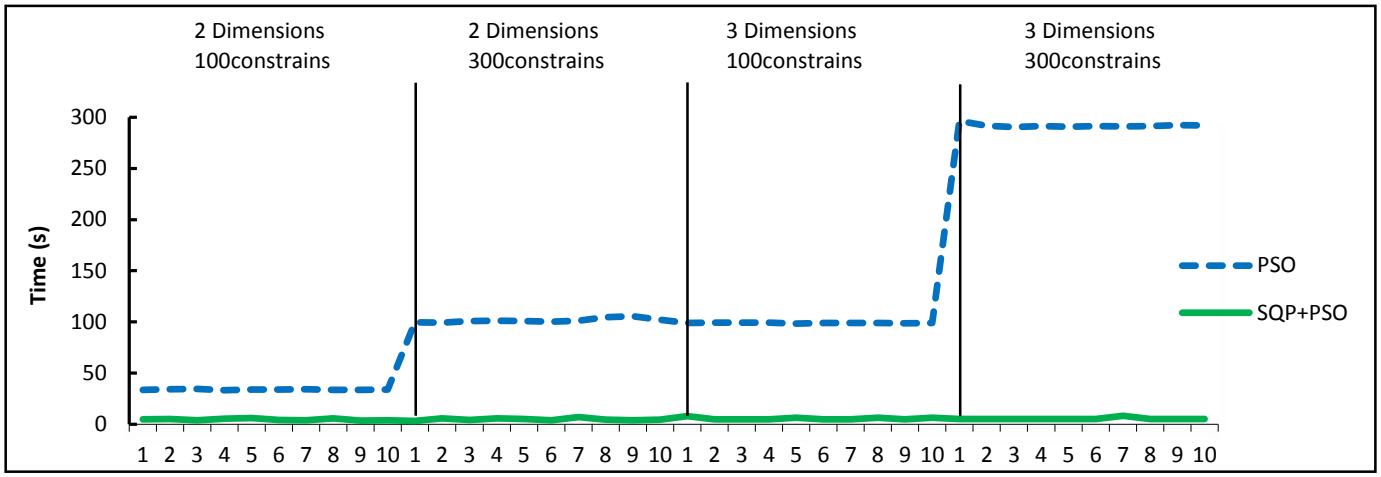

Figure 5: The runtime of PSO and SQP_PSO in different problem

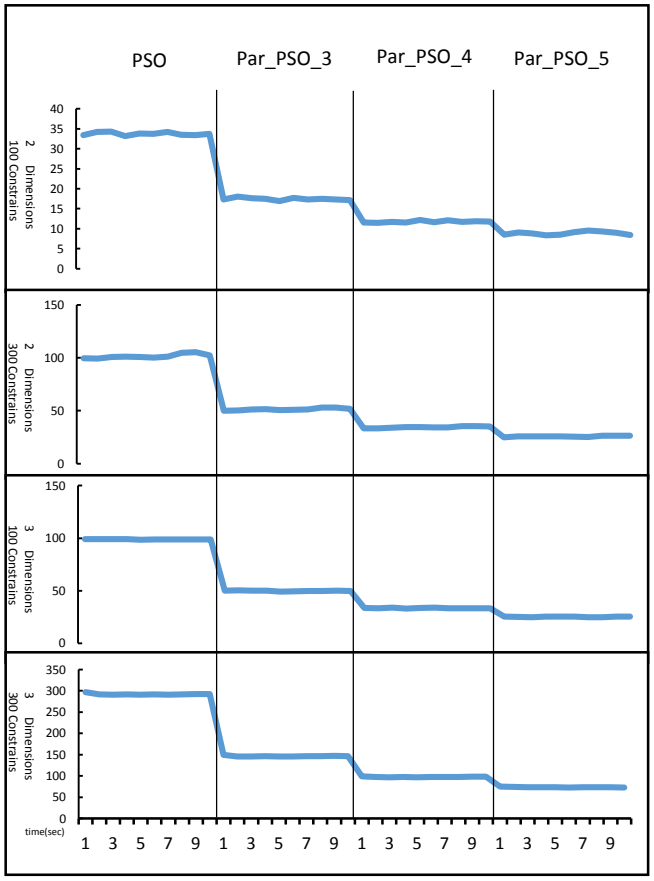

Figure 6, The runtime of different problems for PSO in multiple processors

formula 4) and the efficiency (see as formula 6) to evaluate the merit of parallelism. According to Table 1, when we increase the amount of processors, the efficiency also gets higher, and the speedup almost equals P-1. Therefore, the efficiency estimation of parallel PSO in multiple processors will be similar to the formula (7). In formula (7), if the number of processors increase infinitely, the efficiency will almost equal to one. By this result, we are aware that the parallelism proportion in parallel PSO is high, and can be one of the reasons to utilize parallel PSO as subproblem's solver.

$$E = \frac{P-1}{P} \quad (7)$$

The second reason for utilizing parallel PSO in this research as the main solver to solve subproblem in SQP is because according to the previous research, the runtime of PSO will be influenced by the amount of constraints. It has better performance in fewer constraints problems. Moreover, in our SQP subproblem, it only needs to solve problems with one constraint. So we consider parallel PSO to be a good choice.

4.2 Results for Parallel SQP_PSO

The main parallelism part for SQP_PSO in this research is to find the constraint with maximum value at a certain solution. Based on the high speed calculation, if the amount of constraints is not numerous, it won't influence the runtime of SQP too much, which can be observed from Figure 7 and Table 2. In Figure 7 and Table 2, the broken line of SQP_PSO and the runtime changes little. By this result, we can forecast that in the future work, parallel SQP_PSO has potential in solving large scale problems.

4.3 Speedup and Efficiency

We set the processor zero as the master processor to control the global information update in PSO. The swarms can't assign to master processor. Thus, only P-1 processors

participate particles division. However; when calculating efficiency, the speedup still needs to be divided by P processors. Therefore, even though the speedup is pretty close to P-1, as formula (7) represents, it still causes bad performance in the efficiency when the amount of processors isn't large enough.

According to Table 2, the efficiency becomes larger when the amount of processors increases. Depending on the growing trend, we assume the efficiency will still keep growing. Hereby, we can assume SQP_PSO algorithm and parallel SQP_PSO will be valuable in future work.

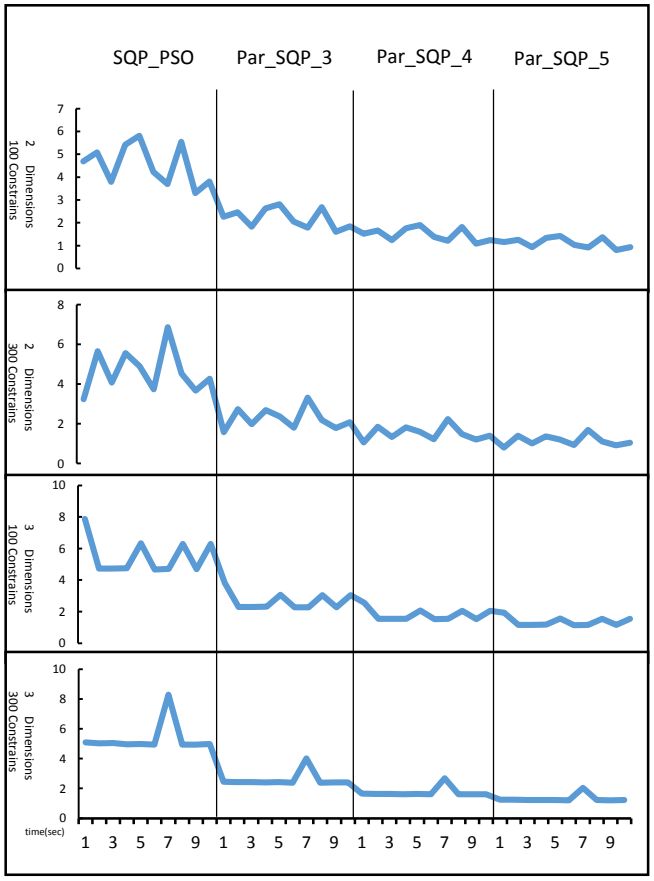

Figure 7, The runtime of different problems for SQP_PSO in multiple processors

4.4 Objective Value

The information of the objective values is tabulated in Table 1 and Table 2. By Table 1 and Table 2, we can discover that in two dimensions' problem, there's no difference between PSO and SQP_PSO in the objective values issue. However, in three dimensions' problem, PSO is more precise than the values found by SQP_PSO. It may be caused by two factors. First, as we mentioned in section 3, to avoid jumping problem, we lock the constraint when the objective value seems to converge. Second, we relax the stop criteria to make sure that

Table 1: Result of PSO and parallel PSO experiment

	PSO	par_PSO_3	par_PSO_4	par_PSO_5
2D-100(sec)	33.76	17.43	11.75	8.87
Speedup	NA	1.94	2.87	3.81
Efficiency	NA	0.65	0.72	0.76
Obj_value	0.20	0.20	0.20	0.20
2D-300(sec)	101.51	51.36	34.44	25.84
Speedup	NA	1.98	2.95	3.93
Efficiency	NA	0.66	0.74	0.79
Obj_value	0.20	0.20	0.20	0.20
3D-100(sec)	98.90	49.91	33.50	25.27
Speedup	NA	1.98	2.95	3.91
Efficiency	NA	0.66	0.74	0.78
Obj_value	0.12	0.12	0.12	0.12
3D-300(sec)	291.90	146.43	97.77	73.55
Speedup	NA	1.99	2.99	3.97
Efficiency	NA	0.66	0.75	0.79
Obj_value	0.11	0.11	0.11	0.11

Table 2: Result of SQP and parallel SQP experiment

	SQP	par_SQP_3	par_SQP_4	par_SQP_5
2D-100(sec)	4.54	2.20	1.48	1.12
Speedup	NA	2.06	3.07	4.05
Efficiency	NA	0.69	0.77	0.81
Obj_value	0.20	0.20	0.20	0.20
2D-300(sec)	4.65	2.25	1.52	1.14
Speedup	NA	2.07	3.06	4.08
Efficiency	NA	0.69	0.76	0.82
Obj_value	0.20	0.20	0.20	0.20
3D-100(sec)	5.50	2.67	1.79	1.35
Speedup	NA	2.06	3.07	4.07
Efficiency	NA	0.69	0.77	0.81
Obj_value	0.13	0.12	0.12	0.12
3D-300(sec)	5.32	2.58	1.73	1.31
Speedup	NA	2.06	3.08	4.06
Efficiency	NA	0.69	0.77	0.81
Obj_value	0.12	0.11	0.12	0.11

the program can output the value successfully. Above all, the objective value we obtain from SQP_PSO is not exact. It's an approximation value which is in the feasible region and is close to stationary point.

The implements we used in program to solve jumping error still need to be considered. We tend to improve it until it can converge on the exact local minimum or the direction should be equaled zero. Therefore, we are trying other methods to solve it. So far, we assume the possible reason of jumping error may be caused by step size. In the future research, we want to import Armijo rule or minimizing rule instead of constant step size and expect that the jumping error can be solved.

5. CONCLUSION

Based on the result we obtained from the numerous experiments; we can easily see that this algorithm is valuable for large scale problems. Therefore, we tend to apply this algorithm in three ways. The first way is to solve some big problems, which the amount of variables and constraints can't be afforded by one single memory system. In this kind of problem, we can't just use powerful commercial software such as Knitro to deal with it. The second way is to use this to solve global optimization problem. After all, apart from proving the property, the way to find out the global solution usually requires exhaustive searching. This type of work takes too much time, so we think the algorithm we propose can aid it. Third, the most important of all, the context of algorithm still includes some bugs, and the property needs to be stated clearly. Hereby, in the future we expect we can discover the solution of the system bugs, and prove the property, convergence or efficiency of it.

6. REFERENCE

Bertsekas, D. P. (2016). *Nonlinear Programming 3rd Edition*. Athena Scientific.

Chang, J. F., Chu, S. C., Roddick, J. F., Pan, J. S. (2005). Parallel Particle Swarm Optimization Algorithm with Communication Strategies. *Journal of Information Science and Engineering (21)*, pp. 809-818.

Dixon, L. C. W., Jha, M. (1993). Parallel algorithms for global optimization. *Journal of Optimization Theory and Applications, 79(2)*, pp. 385-395.

Kang, S. J., Lee, S. Y., Lee, K. M. (2015). Performance Comparison of OpenMP, MPI, and MapReduce in Practical Problems. *Advances in Multimedia*, pp. 1-9.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

Parsopoulos, K. E., Vrahatis, M. N. (2002). Particle Swarm Optimization Method for Constrained Optimization Problems. Greece: Department of Mathematics, University of Patras Articial Intelligence Research Center.