A Review of the Parallelization Strategies for Iterative Algorithms

Xingxing Zhou
Southeast University
Ming Ling (trio@seu.edu.cn)
Southeast University
Shidi Tang
Southeast University
Yanxiang Zhu
VeriMake Innovation Lab

Research Article

Keywords: iterative algorithms, parallel computing, convergence, parallelization strategy

Posted Date: November 20th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3573900/v1

Additional Declarations: No competing interests reported.

A Review of the Parallelization Strategies for Iterative Algorithms

Xingxing Zhou[1], Ming Ling[1*], Shidi Tang[1], YanXiang Zhu[2]

[1*]National ASIC System Engineering Technology Research Center, Southeast University, Nanjing, 210096, China.
[2]VeriMake Innovation Lab, Nanjing, 210088, China.

*Corresponding author(s). E-mail(s): trio@seu.edu.cn;
Contributing authors: 220216193@seu.edu.cn; 230239320@seu.edu.cn; zhuyanxiang@verimake.com;

Abstract

Iteration-based algorithms have been widely used and achieved excellent results in many fields. However, in the big data era, data that needs to be processed is enormous in terms of both depth (the dimensionality of data) and breadth (the volume of data). Due to the slowdown of Moore's Law, the computing power of single-core CPUs is becoming saturated. The increase in the computational complexity and the bottleneck of the single-core processors speed exacerbate the time-consuming problem of iterative algorithms. With the rise of multi-core computers and distributed computing systems, parallelizing and deploying iterative algorithms on such systems can make full use of computing resources and accelerate iterative computation, providing a new idea for solving the aforementioned problems. However, due to the logical dependency between two consecutive iterations in an iterative algorithm, it is difficult to directly implement the concurrent computation of such algorithms. To this end, many studies have been conducted on the parallelization of iterative algorithms in both academia and industry. This paper aims to conduct an in-depth research and analysis of these parallelization strategies. Firstly, the abstract description and classification of iterative algorithms are given. Then four concurrency strategies for iterative algorithms are summarized, including logical units that can be intrinsically concurrently computed, multi-initial state parallel search strategy, data parallelism, and task parallelism. Finally, the paper detailed the convergence of parallel iterative algorithms, focusing on building the mathematical model of asynchronous iterative algorithms, and summarizing the convergence conditions of asynchronous iterative algorithms.

Keywords: iterative algorithms, parallel computing, convergence, parallelization strategy

1 Introduction

With the proliferation of multi-core computers, distributed computing systems, and cloud services, parallel computing has become a burgeoning area of research. The fundamental premise of parallel computing is to maximize the utilization of computational resources across all nodes, while minimizing node idle time. Nevertheless, iteration-based algorithms (hereinafter referred to as "iterative algorithms") are extensively utilized in the fields of machine learning, data analysis, and bioinformatics, including the backpropagation algorithm for neural network training, support vector machines in machine learning [1], the Quasi-Newton algorithm for numerical optimization [2], the simulated annealing algorithm for optimization problems [3], and cluster analysis in data statistics [4], among others. By implementing specific iterative strategies, intermediate solutions for these iterative algorithms may be progressively updated until the convergence termination criteria are met, yielding the optimal approximate solution that is both accurate and readily searchable. Although direct algorithms can yield an exact solution to a problem in a finite number of computations, iterative algorithms, on the other hand, offer only an approximate value that approaches the exact solution with each successive iteration [5]. When dealing with large or high-dimensional data sets, direct algorithms may be unable to produce an exact solution, whereas iterative algorithms can generate a close approximation with a finite number of iterations.

Many areas that use iterative algorithms today are witnessing two notable trends. First, in the era of big data, iterative algorithms must process increasingly larger and more complex data sets [6]. The breadth of data refers to its dimensionality, while the depth of data pertains to its quantity. For example, in bioinformatics virtual screening for potential drug molecules, the structural characterization of candidate drugs can span tens of dimensions or more, with numbers reaching up to 106-107 or higher [7]. The second trend is that algorithms used in many fields, such as face detection in edge computing [8], CT (Computed Tomography) in medical and healthcare [9], are increasingly demanding real-time performance. Both of these trends place greater emphasis on the speed of iterative solutions. In the past, the speed of iterative algorithms could be increased through performance improvements in single-core processors. However, as the dominant frequency improvement rate of single-core processors slows down, the speed enhancement of iterative algorithms has also reached a bottleneck.

The parallelization of iterative algorithms constitutes an efficient approach to expedite their execution. However, given the logical interdependence among consecutive iterations of such algorithms, deploying them directly on devices such as FPGA (Field Programmable Gate Array) and GPU (Graphics Processing Unit) that can be easily accelerated in parallel presents a challenge. If implemented on FPGA or GPU without modification, the algorithms typically fail to leverage a significant amount of the device's computing resources simultaneously, resulting in suboptimal acceleration benefits. Consequently, various parallelization strategies for iterative algorithms have been

proposed. In this paper, we categorize these strategies into four types. The first strategy entails employing arithmetic units that can be intrinsically computed in parallel within the iterative algorithm. These computing units (operators) encompass operations involving vectors and matrices. For example, Fang Yu-ling [10] implemented the parallel acceleration of the K-means algorithm by leveraging the idea of matrix multiplication. This approach involved computing matrix transposes, calculating the distances between each data point and each cluster center, and updating the sample points in parallel. The second strategy is the multi-initial state parallel search strategy. It involves generating multiple initial solutions randomly in the search space and reducing the maximum number of iteration loops (search depth) of each thread. Then, iterative computations are performed independently on different threads until all threads meet the convergence termination criteria. Finally, the optimal solution obtained by each thread is sorted based on the size of its corresponding objective function. As an illustration, Tang Shi-di et al. [11] initialized multiple molecular conformations, employed AutoDock Vina on the GPU, and achieved a speed-up ratio of 21-fold. Strategy 3 pertains to data partitioning. This approach involves dividing the search space (input data), with each thread executing iterative calculations based on the assigned subspace. Upon convergence and termination of all threads, the final solutions are computed by combining the outputs of all threads. For instance, the MapReduce framework [12] proposed by Google for processing and analyzing large datasets adopts this concurrent computing model. Strategy 4 entails task parallelism, where the iterative algorithm is partitioned into several tasks via a judicious partitioning strategy, and each thread is responsible for executing a task. Due to the possibility of data dependencies among different tasks, communication mechanisms such as shared storage are utilized to synchronize or asynchronously update local data. For instance, Dan Alistarh et al. [13] proposed a flexible task scheduler to parallelize iterative algorithms.

The convergence of iterative algorithms and the detection of convergence after parallelization are significant issues. Depending on whether the communication of iterative algorithms deployed on distributed computing devices is synchronized, iterative algorithms can be further categorized into synchronous iterations and asynchronous iterations [14]. Synchronous iteration accomplishes simultaneous updates of each computing node after each iteration with the help of global communication, while each computing node in asynchronous iteration performs its own iterative calculation independently, using possibly outdated data to update itself [6]. This significantly reduces the time overhead brought by synchronous communication. Therefore, the convergence of the synchronous iterative algorithms is consistent with that of the original iterative algorithms, while the convergence of the asynchronous iterative algorithms is uncertain. To avoid algorithm divergence, BertSekas established a general convergence theorem for the asynchronous iterative formats by constraining the iterative function and the data staleness during the iterative process [15]. Additionally, in a distributed environment, it is essential to avoid blocking communication requests to efficiently run asynchronous iterative calculations, which increases the difficulty of isolating and processing global vectors. As a result, as the number of communication loads increases, it becomes challenging to determine the convergence of the asynchronous iterative

algorithms from the residuals of the global solution components. Therefore, it is necessary to explore the convergence detection mechanism of the asynchronous iterative algorithms [16].

The paper is organized as follows: Section 2 introduces the definition of iterative algorithms and the challenges they face, along with some typical iterative algorithms; Section 3 elaborates on the four parallelization strategies for iterative algorithms; Section 4 discusses the convergence of parallel iterative algorithms; Section 5 summarizes the entire paper and provides insights into future research.

2 Iterative Algorithm

2.1 Overview of Iterative Algorithms

For an input vector $\boldsymbol{S}$, the loop iterative calculation is performed according to the iterative strategy of the specific problem. Assume that the iterative strategy is denoted as f, which can be a function or some specific operations. After one iterative computation, a new iteration vector can be obtained, as shown in formula 1:

$$\boldsymbol{S_{n+1}} = f(\boldsymbol{S_n}) \tag{1}$$

The vector $\boldsymbol{S_n}$ corresponds to the vector obtained in the n^{th} iteration, whereas $\boldsymbol{S_{n+1}}$ represents the vector obtained in the $(n+1)^{th}$ iteration. The iterative process continues in accordance with the iterative formula 1 until the convergence termination criterion is met.The commonly employed criteria for convergence termination encompass two types: the first being the attainment of a predetermined value for the number of iterations, and the second being the diminishment of the error between vectors generated in two successive iterations (which may be quantified by either the Euclidean or Manhattan distance) to a specified small quantity, as depicted in formula 2 and formula 3:

$$n \geq preset_{loop} \tag{2}$$

$$|\boldsymbol{S_n} - \boldsymbol{S_{n-1}}| \leq \varepsilon \tag{3}$$

The constant $preset_{loop}$ in formula (2) represents the maximum number of iterations that have been pre-determined, and ε in formula (3) is the minimum allowed value that has been set. Additionally, iterative algorithms usually have convergence, which is to say that there exists a value $\boldsymbol{S^*}$ that satisfies:

$$\boldsymbol{S^*} = f(\boldsymbol{S^*}) \tag{4}$$

$\boldsymbol{S^*}$ is known as the fixed point of the function f, and this is why formula 3 serves as the convergence criterion. In other words, as the iterative algorithm approaches the convergence point, the difference between two consecutive iterative solutions should become increasingly negligible.

2.2 Common Iterative Algorithms

Iterative algorithms are ubiquitous and utilized in various fields, such as deep learning, big data analysis, numerical analysis, and medical and healthcare. While direct algorithms can provide an exact solution to a problem in a finite number of operations[5], for instance, producing an expression for a partial differential equation solution, iterative algorithms usually only furnish an approximate value that converges towards the exact solution. For instance, finding the exact solution to a nonlinear partial differential equation [17] can be challenging, and it is often more effective to use iterative methods to derive its numerical solution. Specifically, iterative algorithms are advantageous when processing high-dimensional data since they generally necessitate fewer iterations to obtain a satisfactory approximation compared to the considerable amount of computing time needed to obtain the analytical solution. Furthermore, the development of iterative algorithms is inextricable from the progression of contemporary semiconductor technology and computer architecture. As the number of pipeline stages in processor architecture increases and the dominant frequency of the processor rises [18], the execution efficiency of the processor (indicated as Instructions Per-Cycle, IPC) becomes even more efficient under the same frequency, due to the continuous innovation of processor micro-architecture. Given space constraints, this paper only covers a few of the most common iterative algorithms.

2.2.1 Back Propagation Algorithm

The study of neural networks has gained significant traction in recent years, finding its application in various fields such as facial recognition [19], smart cities[20], economic forecasting [21], and biomedicine[22]. A neural network comprises an input layer, multiple hidden layers, and an output layer. Modern neural networks can have numerous hidden layers, with millions of parameters. The accuracy of the neural network model for data classification or regression hinges on the quality of the weights and bias parameters of each layer in the neural network, which makes model training a crucial step in constructing the neural network model. The Back Propagation algorithm (BP algorithm) is generally used to iteratively adjust and optimize the parameters of each layer of the neural network based on the training set, making it an iterative optimization algorithm.

The BP algorithm utilizes gradient descent to iteratively adjust the weights and biases of each layer in the neural network to minimize the loss function. The loss function, denoted as C, and the activation function, denoted as σ, are used in the algorithm with L layers. BP algorithm is based on the concept of recursion and problem partitioning, and it is derived in batches. In the formula 5, δ_j^l represents the partial derivative of the loss function with respect to the weighted input of the j^{th} neuron in the l^{th} layer.

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{5}$$

z_j^l in formula 5 denotes the weighted input of the j^{th} neuron in the l^{th} layer. z_j^l can

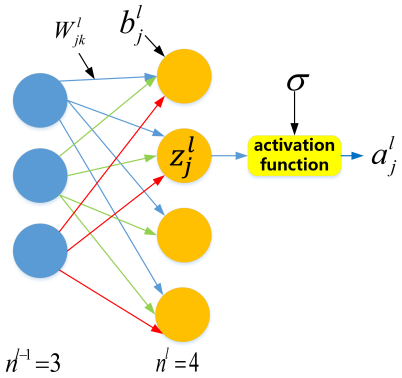

Fig. 1 Symbol diagram of back-propagation algorithm for neural network

be expressed as:

$$z_j^l = \sum_{k=1}^{n^{l-1}} w_{jk}^l a_k^{l-1} + b_j^l \tag{6}$$

The symbols relevant to formula 6 are displayed in Fig.1. In this figure, w_{jk}^l represents the weight coefficient that links the j^{th} neuron in the l_{th} layer to the k^{th} neuron in the $(l-1)^{th}$ layer. Furthermore, a_k^{l-1} denotes the activation function output of the k^{th} neuron in the $(l-1)^{th}$ layer, n^l signifies the count of neurons in the l^{th} layer, and b_j^l stands for the bias of the j^{th} neuron in the l^{th} layer. The δ_j^L for the L^{th} layer (i.e., the final layer) can be derived as:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \times \frac{\partial \sigma(z_j^L)}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \sigma^{'}(z_j^L) \tag{7}$$

Then the σ_j^L for the l^{th} layer is:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_{k=1}^{n^{l+1}} \delta_k^{l+1} w_{kj}^{l+1} \sigma^{'}(z_j^l) \tag{8}$$

Henceforth, the partial derivative of the loss function with respect to the bias of each neuron is:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \tag{9}$$

The partial derivative of the loss function with respect to each weight is:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \tag{10}$$

In summary, as presented in Algorithm 1, the backpropagation algorithm for the neural network comprises two layers of nested iterations. The outer iteration selects

Algorithm 1 Back Propagation

Require: $C(Loss\,Function), \sigma(Active\,Function), Z_j^l(Weighted\,Input)$, $\alpha(Iteration\,Step\,Size), batch_{num}(Number\,of\,Batch)$

Ensure: $w_{jk}^l, a_k l-1$

for $i=1, i \leq batch_{num}, i=i+1$ **do**
  $\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma^{'}(z_j^L)$
  for $l=L-1, l \geq 1, l=l-1$ **do**
    $\delta_j^l = \sum_{k=1}^{n^{l+1}} \delta_k^{l+1} w_{kj}^{l+1} \sigma^{'}(z_j^l)$
    $\frac{\partial C}{\partial b_j^l} = \delta_j^l$
    $\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$
    $b_j^l = b_j^l - \alpha\delta_j^l$
    $w_{jk}^l = w_{jk}^l - \alpha\delta_j^l a_k^{l-1}$
  end for
end for

distinct batches (subsets of the training set) for batch gradient descent, while the inner iteration updates all parameters of the neural network layers based on the partial derivative of the loss function with respect to each parameter, as per the gradient descent method.

2.2.2 BFGS Algorithm

The BFGS algorithm, as described in [2, 23], is a derivative of the conventional Newton iterative algorithm. Assuming that the objective function is denoted as $f(X)$, and the aim of optimization is to minimize $f(X)$ in a given solution space, the iteration format of the Newton iterative algorithm can be expressed as follows:

$$X_{k+1} = X_k - M_k^{-1} G_k \tag{11}$$

In formula 11, X_k represents the solution derived after the k^{th} iteration, M_k denotes the Hessian matrix of $f(X)$, and G_k represents $\nabla f(X_k)$, which is the gradient vector of $f(X)$ at X_k. However, when the dimension of X is considerably large, computing the second-order partial derivatives of the objective function and the Hessian matrix can become immensely complex. To circumvent this, an approximate Hessian matrix is introduced in the BFGS algorithm, thus avoiding the need to calculate the second-order partial derivatives and the inverse of the matrix. The iteration format is presented in formula 12:

$$X_{k+1} = X_k - \alpha H_k G_k \tag{12}$$

In formula 12, H_k represents the approximate Hessian matrix, while α denotes the update step. Moreover, $S_k = X_{k+1} - X_k$, and $Y_k = G_{k+1} - G_k$. The updated formula for the approximate Hessian matrix[24] is displayed in formula 13:

$$H_{k+1} = (I - \frac{S_k Y_k^T}{Y_k^T S_k}) H_k (I - \frac{Y_k S_k^T}{Y_k^T S_k}) + \frac{S_k S_k^T}{Y_k^T Y_k} \tag{13}$$

Algorithm 2 BFGS

Require: $X(Generated\,Randomly), H_0 = I(Unit\,Matrix), \varepsilon(Iteration\,Termination\,Criterion)$
Ensure: $X^*(Minimize\,f(X))$
1: $X_0 = X$
2: **for** $k = 0, k \leq pre\_set\_param, k++$ **do**
3: $g_k = \nabla f(X_k)$
4: $D_k = -H_k G_k$
5: $X_{k+1} = X_k + \alpha \times D_k$
6: $G_{k+1} = \nabla f(X_{k+1})$
7: $S_k = \alpha \times D_k$
8: $Y_k = G_{k+1} - G_k$
9: $H_{k+1} = H\_update(H_k)$
10: **if** $|y_k| < \varepsilon$ **then**
11: Break
12: **end if**
13: **end for**
14: $X^* = X_k$

Algorithm 2 illustrates the flow of the BFGS algorithm.

2.2.3 NIPALS PCA Algorithm

Principal Component Analysis (PCA) is a statistical method that transforms high-dimensional data sets into low-dimensional representations for the purposes of data analysis, visualization, and feature extraction [25]. Typically, the eigenvalues and eigenvectors of the data matrix in the PCA algorithm are computed using the method of Singular Value Decomposition (SVD). However, for large datasets, computing all the eigenvectors of $X^T X$, can be unnecessary and time-consuming [26]. Therefore, it is sufficient to calculate only the eigenvectors that correspond to the eigenvalues with the largest absolute values. To address this, the NIPALS-PCA algorithm was proposed [27], where NIPALS stands for Non-linear Iterative Partial Least-Squares. The NIPALS-PCA algorithm is a variant of the power method that involves computing a principal component, performing matrix contraction, and then computing the subsequent principal component. Assuming that the data matrix is denoted as X, with dimensions $m \times n$, and the element at row i and column j is x_i^j, with each column corresponding to a feature. Suppose A is an $N \times N$ matrix, and γ_1 is the eigenvalue with the largest modulus among its Neigenvalues, satisfying formula 14:

$$|\gamma_1| > |\gamma_i| \quad i = 2, 3, \ldots \tag{14}$$

The eigenvector corresponding to γ_1 is referred to as the principal eigenvector of A. Let it be assumed that the N eigenvectors of A, denoted as $\boldsymbol{\varepsilon_i}$ for $i = 1, 2, \ldots, N$, are all linearly independent and satisfy $A\boldsymbol{\varepsilon_i} = \gamma_i \boldsymbol{\varepsilon_i}$, where γ_i is the ith eigenvalue. Then, for any non-zero vector $\boldsymbol{v_0} \in R^N$, formula 15 holds:

$$\lim_{k \to \infty} A^k \boldsymbol{v_0} = \alpha_1 \gamma_1^k \boldsymbol{\varepsilon}_1 \tag{15}$$

Algorithm 3 NIPALS-PCA

Require: $X(Data\,Matrix), \varepsilon(Iteration\,Termination\,Criterion)$, $num(Number\,of\,Eigenvectors)$

Ensure: $\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_{num}}(Feature\,Vectors)$

1: **for** $k = 1, k \leq num, k = k + 1$ **do**
2: $\quad \boldsymbol{t_k'} = generate\_random\_not\_zero\_vector()$
3: $\quad$ **while** $|\boldsymbol{t_k} - \boldsymbol{t_k'}| \geq \varepsilon$ **do**
4: $\quad\quad \boldsymbol{t_k} = generate\_random\_not\_zero\_vector()$
5: $\quad\quad \boldsymbol{p} = \frac{X^T \boldsymbol{t_k}}{\boldsymbol{t_k^T t_k}}$
6: $\quad\quad \boldsymbol{p} = \frac{\boldsymbol{p}}{(\boldsymbol{p^T p})}(\frac{1}{2})$
7: $\quad\quad \boldsymbol{t_k'} = \frac{X\boldsymbol{p}}{\boldsymbol{p^T p}}$
8: $\quad$ **end while**
9: $\quad \boldsymbol{t_k} = \boldsymbol{t_k'}$
10: $\quad X = X - \boldsymbol{t p^T}$
11: **end for**

From formula 15, we can conclude that the vector $A^k \boldsymbol{v_0}$ corresponds to the eigenvector of the largest eigenvalue γ_1. To obtain the corresponding eigenvector, it is necessary to normalize the vector $A^k \boldsymbol{v_0}$ to prevent overflow. The eigenvectors can be obtained as follows:

$$\boldsymbol{v_k} = \frac{A^k \boldsymbol{v_0}}{|A^k \boldsymbol{v_0}|} \tag{16}$$

The main steps of the NIPALS-PCA algorithm are shown in Algorithm 3. First, a non-zero vector $\boldsymbol{t_k}$ is generated, and the vector $\boldsymbol{t_k'}$ is obtained through one power iteration. Then, whether the vector $\boldsymbol{t_k}$ is stable is judged by step 9 in Algorithm 3. If so, $\boldsymbol{t_k}$ is considered to be the principal eigenvector (the eigenvector corresponding to the largest absolute value), and then the secondary eigenvector is calculated through the matrix deflation in step 10. Otherwise, the power iteration continues until the vector $\boldsymbol{t_k}$ converges.

2.2.4 Simulated Annealing Algorithm

Simulated Annealing Algorithm is a global optimization technique that is widely employed in diverse fields, such as biomedicine and numerical computation, due to its ease of implementation [3]. Its inception can be traced to the natural cooling process of matter, where the loss function is treated as the energy of matter. As depicted in Algorithm 4, in each iteration, a new state with fresh energy (E_{new}) is generated from the current state with old energy (E_{old}). If E_{new} is less than E_{old}, the new state is accepted. However, if E_{new} is greater than E_{old}, the new state is accepted with a probability of $\exp(-(E_{old} - E_{new})/T)$ [28]. The state sequence at a constant temperature is solely dependent on the prior state sequence at a higher temperature, hence the numerical iterative solution sequence of the simulated annealing algorithm can be viewed as a Markov chain. The Metropolis criterion in the simulated annealing algorithm makes it possible to circumvent local optimal solutions and efficiently search

Algorithm 4 Simulated Annealing

Require: $T = T_0(Initial\,Temperature), x = x_0(Initial\,Solution), E_{best} = f(x_0), f(x) = f(x_0), d(Search\,Depth), \rho(Cooling\,Rate)$
Ensure: $E_{best}, x_{best}(miniminze\,f(x))$

```
while T > T_final do
    for search_depth = 1 : d do
        x' = random_neighbour(x)
        δ = f(x') − f(x)
        Random r uniformly in the range (0, 1)
        if (δ < 0) or (r < exp(−δ/T)) then
            x = x'
        end if
    end for
    T = ρT (0 < ρ ≤ 1)
end while
```

for global optimal solutions. However, a large number of iterative calculations are frequently required to ensure the quality of the optimal solution, which leads to a lengthy computation time. As a result, the time-consuming calculation has become the bottleneck for implementing the simulated annealing algorithm to large-scale optimization problems.

2.2.5 K-means Algorithm

In the era of big data, data clustering analysis has emerged as an indispensable component of data statistical analysis. The K-means clustering algorithm [4] is widely favored by the academic community due to its simplicity and ease of design. Given a dataset $S = \{x_i | i = 1, \ldots, n, x_i \in R^d\}$, the objective is to divide the n elements in the dataset into k clusters in such a way that the sum of squared distances of all elements to their respective centers is minimized. The centers of K clusters are represented by $C = \{C_i | i = 1, \ldots, K, C_i \in R^d\}$, the number of elements included in the i^{th} class is denoted by n^i, and the Euclidean distance is employed. The minimized objective function f can be mathematically expressed as formula 17:

$$f = min(\sum_{j=1}^{k} \sum_{i=1}^{n^j} ||x_i - C_j||^2) \tag{17}$$

The algorithm's flow is presented in Algorithm 5. As can be observed, each iterative update of the cluster center is dependent on the data of the center in the preceding calculation, which includes the data clusters of each data point and the specific locations of each center.

Algorithm 5 K-means

Require: $S = x_i | i = 1, \ldots, n, x_i \in R^d (Dataset)$, $pre\_set\_param(Maximum\,Number\,of\,Iterations), \varepsilon(Iteration\,Termination\,Criterion)$
Ensure: $C = C_i | i = 1, \ldots, K, C_i \in R^d$

C^0 (random generated or based other strategies)
for $i = 0, ipre\_set\_param, i = i + 1$ **do**
  for $j = 1, j \leq n, j = j + 1$ **do**
    for $k = 1, k \leq K - 1, k = k + 1$ **do**
      if $dist(x_j, C_k^i) \leq dist(x_j, C_{k+1}^i)$ **then**
        $x_j \in Class_k^i$
      else
        $x_j \in Class_{k+1}^i$
      end if
    end for
  end for
  for $k = 1, k <\leq K, k = k + 1$ **do**
    $C_k^{i+1} = (\sum_{x_q \in Class_k^i} x_1)/size(Class_k^i)$
  end for
  $obj_{func1} = f(S, C^i)$
  $obj_{func2} = f(S, C^{i+1})$
  if $abs(obj_{func1} - obj_{func2}) < \varepsilon$ **then**
    Break
  end if
end for

2.3 Problems with Iterative Algorithms

Presently, the development of iterative algorithms faces two major challenges. The first challenge is that it is difficult to further enhance the principal frequency of the CPU, and as a result, the calculation speed of the iterative algorithms has become a bottleneck. This is primarily due to two factors: firstly, the semiconductor process has reached its physical limits, and secondly, the heat density of the processor chip is too high. As the clock frequency increases, the number of transistors within the processor, numbering in the billions, undergoes a greater number of flips within a unit of time. However, the elevation of the processor's clock frequency gives rise to a substantial amount of heat (referred to as the power consumption barrier), exerting a profound influence on the performance of the processor. The constraint on the increase in the main frequency of the processor implies that there is a bottleneck in the computing power of the single-core processor. This limits the advantage of iterative algorithms relying on the high principal frequency of the CPU for fast iterative calculations. Currently, many commercial processors, such as AMD's Ryzen CPU, have entered the multi-core era, increasing the number of cores to enhance the computing power of computers. However, due to the inherent logical dependence of iterative algorithms, it is difficult to deploy them directly on a multi-core processor, making it challenging to fully utilize the parallel computing capability of a multi-core processor.

The magnitude of scientific computing problems is progressively expanding, and the iterative algorithms are becoming more intricate. "The augmentation of computational scale encompasses not only the proliferation of data quantity but also the augmentation of data dimensions. Ultimately, both will expand the solution space of the problem. The complexity of iterative algorithms lies in the abundance of operators within the algorithm, as well as the computational intensity, thereby manifesting an augmented temporal expenditure for each individual iteration calculation. For iterative algorithms, this implies the necessity for a greater number of iterative computations to ensure the ultimate convergence and, consequently, the expenditure of a substantial amount of computational time. This kind of delay is insufferable for real-time applications, such as edge computing-based face detection[8], which commonly employs convolutional neural networks as its detection algorithm. During forward inference and backpropagation, two adjacent layers have data dependency. Furthermore, as the number of layers and parameters of the neural network increases, the time required for forward inference and backpropagation also intensifies, prompting the launch of the Neural Processing Unit (NPU). Another instance is the field of biology's Molecular Docking, where the aim is to ascertain the best docking posture between a receptor (a large molecule, typically a protein) and a ligand (a small molecule, usually a drug candidate). Nonetheless, discovering a drug candidate often necessitates docking millions of small molecules, and the computational scale is quite extensive. AutoDock Vina [29] is a commonly employed molecular docking tool, whose search algorithm is a global optimizer with quasi-Newton method (BFGS algorithm in Vina) as the local search algorithm. Its fundamental essence is an irregular long iterative algorithm. Due to the irregular and long iteration characteristics of the Vina searching algorithm, the calculation time is quite prolonged, making it challenging to apply to the large-scale virtual screening of drug molecules [7].

2.4 Synchronous Iteration and Asynchronous Iteration

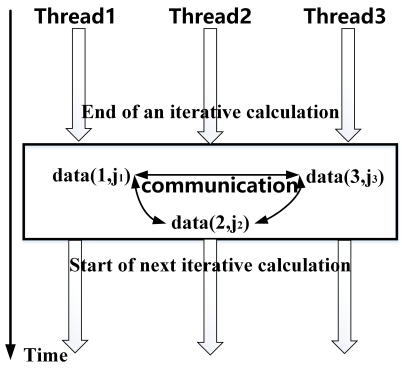

Fig. 2 Synchronous Iteration and Asynchronous Iteration. Synchronous iteration requires $j_1 = j_2 = j_3$, while in asynchronous iteration, there is no strict equality relationship between j_1, j_2, and j_3. Only certain constraints exist, such as $max67\{|j_1 - j_2|, |j_1 - j_3|, |j_2 - j_3|\} \leq bound$[30]

The iterative algorithm deployed on distributed devices can be classified into synchronous iteration and asynchronous iteration based on the need for synchronization[14]. For example, deploying an iterative algorithm across three threads, denoted as Thread1, Thread2, and Thread3, with distinct inter-thread data dependencies, necessitates the establishment of communication mechanisms between the threads. The data generated during the j^{th} iteration in the i^{th} thread is denoted as $data(i, j)$. A synchronous iterative algorithm implies that all threads commence iterative calculations concurrently. Without altering the execution sequence of the algorithm, each thread must update local variables via global communication following each iteration. As illustrated in Fig.2., in the communication phase, different threads perform the data update of local threads based on mechanisms like shared storage. $j_1 = j_2 = j_3$ ensures that all threads $data(i, j)$ are updated before starting a new round of iteration. The primary factor constraining the performance of synchronous iterative algorithms is the communication between processors. As a result of varying computing capabilities of different processors, the processor with higher performance might have to wait for the processors with lower performance to complete calculations, resulting in some processors being idle. Furthermore, as communication load increases, communication bandwidth becomes a critical bottleneck [31]. A viable solution to this problem is implementing load balancing [32] to ensure that the execution time of each iteration for every computing node is comparable, thereby eradicating the penalty resulting from synchronization between nodes in each iteration.

The asynchronous iteration algorithm [33] entails the independent execution of iterations by all processors. Prior to the initiation of each iteration, if the required data has not been updated from other threads, and assuming Thread 1 requires data from Thread 2, but if Thread 2, due to a high computational load or weak computational capability, results in $j_2 < j_1$, then the thread will resort to using outdated ($data(2, j_2)$) data to proceed with the computation.Asynchronous algorithms are generally utilized for parallel computing in distributed systems, encompassing the optimization of large-scale linear algebra to the distributed coordination of small embedded devices [34]. The asynchronous algorithm allows computing nodes to operate asynchronously, facilitating the implementation of distributed algorithms and eliminating the communication waiting overhead associated with synchronization. Hence, the time taken for the asynchronous version of the iterative algorithm to complete calculation is 1/2 to 2/3 of that taken by the synchronous version [35]. If a node in the asynchronous iterative algorithm fails, it is not necessary to reinitialize all nodes; instead, resetting the node is sufficient[35]. The convergence of asynchronous iterative algorithms is difficult to guarantee, but theoretical analysis shows that many asynchronous iterative algorithms are convergent as long as the communication between asynchronous iterations is reasonably constrained [36]. Tritsiklis J N [37] has proven that the convergence rate is optimized when all components are updated in each iteration. Nevertheless, since asynchronous iterative algorithms may use outdated data, the convergence rate may decelerate, necessitating more iterations than synchronous versions, thereby offsetting the advantages of asynchronous iteration to a certain extent. Mikael Johansson's team [30] has concluded that the convergence rate will deteriorate with an increase in delay by modeling the communication delay and update rate.

3 Parallelization Strategies for Iterative Algorithms

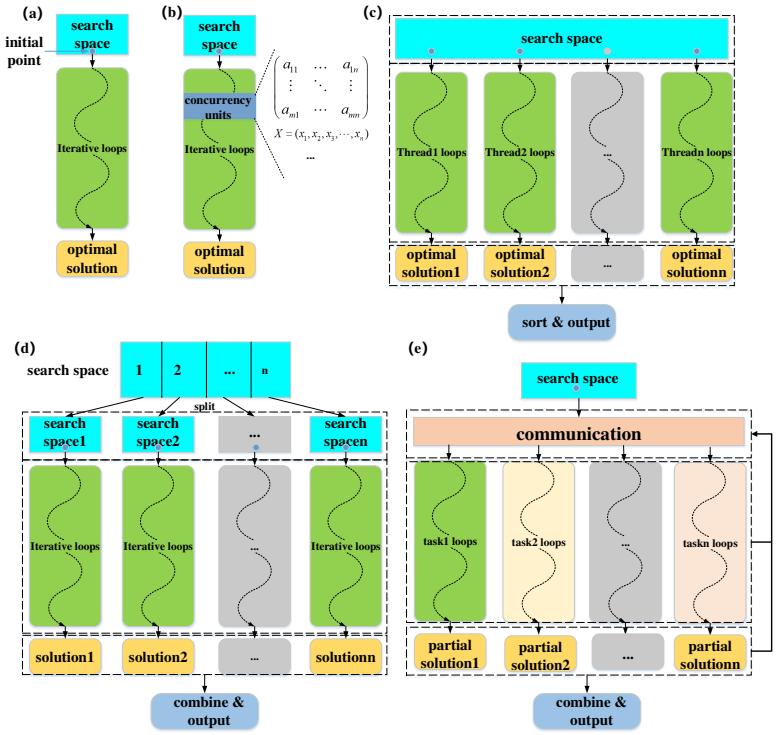

Fig. 3 Schematic diagram of parallelization strategies for iterative algorithms. a) Iterative algorithm. b) Iterative algorithm with logical units that can be intrinsically concurrently calculated. c) Multi-initial state parallel search strategy which generates multiple initial states in the solution space, and is driven by different threads for parallel independent computation. d) Data partitioning which divides the solution space (input data), with different threads executing in different sub-solution spaces. e) Task partitioning which divides the whole iterative algorithm into several tasks, with different threads performing iterative calculations.

A visual representation of a generic iterative algorithm is depicted in Fig.3(a), where an initial point is generated in the search space, followed by a series of iterative loops that eventually satisfy the convergence termination criterion and deliver the optimal solution. Based on our literature review, parallelization strategies for iterative algorithms can be classified into four categories. The first category employs logical units that can be inherently computed concurrently within the iterative algorithm. The second approach is the multi-initial state parallel search strategy. The third category is data parallelism, while the fourth is task parallelism.

Table 3 illustrates the iterative algorithms and implementation methods that are suitable for various iterative algorithm parallelization strategies. The first parallelization strategy involves logical units that can be intrinsically computed concurrently.

Table 1 Iterative algorithms and implementation methods (synchronous iteration, asynchronous iteration) applicable to different parallelization strategies for iterative algorithms

Parallelization Strategy	Applicable Iterative Algorithms	References	Implementation methods
logical unit that can be intrinsically concurrently computed	Algorithms with logical units that can be concurrently computed	[37,38,39]	Synchronous iteration asynchronous iteration
Multi-initial state parallel search strategy	Optimization iterative algorithms (such as BFGS, SA, etc. for finding the optimal solution)	[11,40,41,42,43,44,45, 46,47,48,49,50,51,52]	Mostly adopt asynchronous iteration
Data parallelism	Big data analysis and exploration of the whole solution space	[53,12,54,55,56, 57,58,59,60,61,62,63]	Asynchronous iteration
Task parallelism	All, the algorithm needs to be properly divided into tasks	[13,64,66,6, 67,68,69,70]	Mostly adopt asynchronous iteration

Fig.3(b) demonstrates that specific logical units of the iterative algorithm can be computed concurrently, such as operations involving matrices and vectors. It is crucial to comprehend the computational details of the iterative algorithm and identify the logical part that can be computed concurrently. While both synchronous iteration and asynchronous iterations are feasible, synchronous iteration is predominantly used to guarantee the convergence of the parallelized iterative algorithm.The second parallelization strategy is the multi-initial state parallel search strategy. As depicted in Fig.3(c), this approach first randomly generates multiple initial solutions in the search space, reduces the maximum number of iteration loops (search depth) of each thread, independently performs iterative calculations in different threads until all threads meet the convergence termination criteria, sorts the optimal solutions obtained by each thread based on the size of their corresponding objective functions, and then outputs the solutions. This strategy is primarily applicable to optimization iterative algorithms that involve finding the optimal solution for a specific problem, such as the simulated annealing algorithm, quasi-Newton algorithm, genetic algorithm, and others. To decrease the communication overhead between threads, Strategy 2 primarily employs the implementation method of asynchronous iteration.The third parallelization strategy is data parallelism. As displayed in Fig.3(d), this approach first divides the search space (input data), and each thread performs iterative calculation based on the assigned sub-search space. Once all threads reach the convergence termination criteria, the final solutions of each thread are combined and calculated to produce the output. Strategy 3 is appropriate for big data analysis or algorithms that need to explore the entire solution space (input data), such as updating the parameters of training neural network models or counting the number of occurrences of a particular word in an article. As data parallelism uses the same algorithm operation for different input data, it mostly adopts the implementation method of asynchronous iteration.The fourth parallelization strategy is task parallelism. As demonstrated in Fig.3(e), this approach first divides the iterative algorithm into several tasks, and each thread is responsible for one task. Since there may be data dependencies between different tasks, it is necessary to implement synchronous or asynchronous communication

through communication mechanisms such as shared storage. Strategy 4 is appropriate for general iterative algorithms. The crucial factor is how to divide tasks to minimize the logical dependencies between tasks and how to design an efficient task scheduler. Strategy 4 mainly employs asynchronous iteration to break the data dependencies between different tasks.

3.1 Intrinsically Concurrently Computable Logical Unit

Amdahl's Law, which specifically describes the impact of enhancing the performance of a portion of a system on the overall operating rate of the entire system [38], can be applied to iterative algorithms. In the context of these algorithms, logical units that can be computed concurrently are suitable for performance improvement. These logical units refer to algorithmic details that are easily parallelizable, such as matrix and vector operations. Let T_{origin} be the total execution time of an iterative algorithm, and let t be the time for the logical units. If concurrent computing is performed on these logical units and the acceleration ratio is n, the computation time for these units becomes $\frac{t}{n}$. The time after parallel acceleration is then given by Formula 18:

$$T_{inter\_parallel} = T_{origin} + \frac{t}{n} - t \tag{18}$$

Let the proportion of the computation time of the concurrent logical units to the total time be denoted by α. That is:

$$\frac{t}{T_{origin}} = \alpha \tag{19}$$

Then the overall speed-up ratio of the iterative algorithm can be calculated as:

$$\frac{T_{origin}}{T_{inter\_parallel}} = \frac{T_{origin}}{T_{origin} + \frac{t}{n} - t} = \frac{1}{(1-\alpha) + \frac{\alpha}{n}} \tag{20}$$

From formula 20, the maximum speed-up ratio can be obtained as:

$$\lim_{n\to\infty} \frac{1}{(1-\alpha) + \frac{\alpha}{n}} = \frac{1}{1-\alpha} \tag{21}$$

The upper bound of the speed-up ratio for an iterative algorithm utilizing its inherent concurrently computable logical units is contingent on the percentage of these units relative to the entire computational time. The built-in concurrent computing units referred to herein are not confined to matrix and vector operations. As long as a particular portion of the arithmetic logic in the iterative algorithm can be segmented into multiple autonomous computing modules, it can be deemed as inherent and concurrent.

For example, formula 22 is a system of linear equations:

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \tag{22}$$

If matrix A and vector X are known, and vector $\boldsymbol{B}$ is unknown, matrix A can be divided into n row vectors $[\boldsymbol{A_1}, \boldsymbol{A_2}, \ldots, \boldsymbol{A_n}]$, and n parallel threads are used to calculate $\boldsymbol{A_1X}$, $\boldsymbol{A_2X}, \ldots, \boldsymbol{A_3X}$ respectively to obtain the result vector $\boldsymbol{B}$. In this way, each thread undertakes $1/n$ of the original calculation amount, and the theoretical speed-up ratio reaches n.

When presented with a system of linear equations where the matrix A and vector $\boldsymbol{B}$ are known and the vector $\boldsymbol{X}$ is unknown, the goal is to solve for $\boldsymbol{X}$. If the size of matrix A is relatively small and there is a unique solution, the inverse matrix A^{-1} can be derived, which would allow for the vector X to be obtained using formula 23.

$$\boldsymbol{X} = A^{-1}\boldsymbol{B} \tag{23}$$

Nevertheless, when the dimension of matrix A is very large, it requires a significant amount of memory storage to compute the inverse matrix operation of matrix A. In such cases, a common approach is to obtain an approximate value of $\boldsymbol{X}$ by using the Jacobi iteration method or other numerical iterative techniques. The Jacobi iterative method is expressed in the format of formula 24:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{i=1, j\neq i}^{n} a_{ij}x_j^{(k)}) \tag{24}$$

where $\boldsymbol{X^{(0)}} = (x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, \ldots, x_n^{(0)})$ is an initialized non-zero Vector, $\boldsymbol{X^{(k)}}$ represents the solution vector after the k^{th} iteration.

The Jacobi algorithm is known for its feature of low computational requirements, and its ability to solve multiple components of the solution independently. Kai Song et al. [39] have applied the Jacobi iterative algorithm to a ternary optical computer with reconfigurable processor bits and many data bits, and developed a new parallel design scheme to address the issue of low efficiency for large linear equations. M.ANDRECUT [40] proposed a GPU-parallel implementation of NIPALS PCA, which is convenient to deploy and accelerate on GPU due to the large-scale matrix that needs to be processed in the NIPALS PCA algorithm. However, the error accumulated by the NIPALS PCA algorithm in each iteration results in a loss of orthogonality, and thus it is only utilized for estimating the first few components in practice [40]. To address this problem, the GS PCA (Gram-Schmidt PCA) algorithm has been proposed, which can be conveniently represented by matrix-vector operations, and orthogonality correction is performed for both the scores and loads in each iterative step. According to numerical results, the GPU parallel optimized version based on CUBLAS (NVIDIA) is approximately 12 times faster than the CPU version based on CBLAS (GNU Scientific Library). The K-means algorithm is an iterative computation process without inherent parallelism. Fang Yu-ling et al. [10] employed the concept of matrix multiplication to parallelize the primary steps of the K-means algorithm, including matrix transposition, computation of the distance from each data point to multiple cluster centers, clustering of sample points to the nearest cluster, and updating the cluster center. Molecular docking is a crucial step in drug design, which is basically an optimization problem based on a scoring function. Although AutoDock Vina has good

docking accuracy and speed, the long iterations and irregular nature of the algorithm make it difficult to deploy on hardware acceleration devices. However, Ming L et al. [41] have proactively explored the available logical units for concurrent computation in the AutoDock Vina algorithm, such as parallel computation of intramolecular and intermolecular energy, updating of Hessian matrix involved in the BFGS algorithm, and a novel BRAM access strategy based on data rearrangement. When ensuring relative docking accuracy, the Vina-FPGA [41] version implemented by Ming L et al. only consumes 2.5% of the energy consumed by the CPU version, and achieves an average speed-up ratio of 3.7 times.

3.2 Multi-Initial State Parallel Search Strategy

3.2.1 Multi-Initial State Parallel Search Strategy

The traditional iterative algorithm continuously performs iterative calculations in a specific format on the input initial solution, and updates the solution in each iteration until the iteration termination criterion is met to exit the iteration and to give the searched optimal solution. Therefore, the traditional iterative algorithms can be regarded as iterative sequences with a single initial state. The choice of the initial point plays a decisive role in the speed of convergence. A good initial point often means less computation time and higher computational efficiency. Therefore, the iterative calculation can be accelerated by increasing the number of initial states, which is to increase the number of iteration threads, and reducing the maximum number of iterations in the iteration termination criterion. This parallelization strategy for iterative algorithms is called multi-initial states parallel search, and its algorithm steps are shown in the pseudocode Algorithm 6. First, multiple initial solutions are generated, and then each iterative thread performs mutually independent iterative calculations. When all threads exit the iterative calculation, the best result among all the threads is selected as the final result. Fig.4 shows the multi-initial states parallel search strategy for the iterative algorithms. Tang Shi-di et al. [11] adopted the multi-initial state parallel search strategy, used OpenCL to transplant AutoDock Vina on the GPU for acceleration. The Vina-GPU architecture diagram is shown in Fig.5. The Host end is mainly responsible for the preparation of docking molecular data, and the sorting and output of final docking result. The Device end randomly initializes multiple molecular conformations and executes the main steps of the AutoVina docking algorithm. The number of threads in Vina-GPU is 8000, and the search depth of each thread is determined by the heuristic formula 25:

$$\begin{aligned} Vina\_GPU\_search\_depth = max\,\{1, \\ floor\,(0.24 \times N_{atom} + 0.29 \times N_{rot} - 3.41)\} \end{aligned} \tag{25}$$

The default search depth of AutoDock Vina is determined by formula 26:

$$\begin{aligned} Vina_{search_{depth}} = 105 \times N_{atom} + \\ 1050 \times (N_{torsion} + 6) + 5250 \end{aligned} \tag{26}$$

Algorithm 6 Multi-initial state parallel search strategy in iterative algorithms

Require: n Initial Solutions $\{x_1, x_2, \ldots, x_n\}$
Ensure: $f_{best} = \mathbf{min}\{f(x_1), f(x_2), \ldots, f(x_n)\}, x_{best} = x_b(f(x_b) = f_{best})$

```
x_1^1 = x_1, x_2^1 = x_2, x_3^1 = x_3, ..., x_n^1 = x_n
for each thread_i = 1 : n concurrently do
    for k = 1 : d do
        x_i^{k+1} = iterative_func(x_i^k)
        if abs(f(x_i^{k+1}) − f(x_i^k)) < ε then
            break
        end if
    end for
end for
f_best = min{f(x_1^{k+1}), f(x_2^{k+1}), ..., f(x_n^{k+1})}
x_best = x_b(f(x_b) = f_best)
```

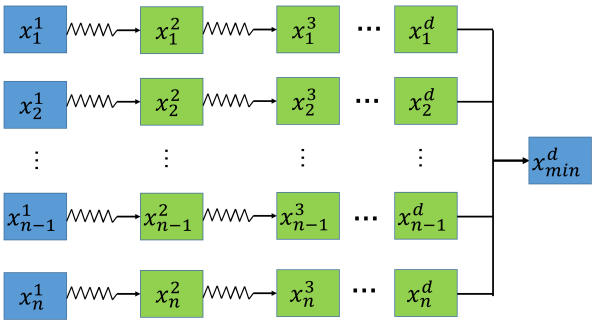

Fig. 4 multi-initial state parallel search strategy in iterative algorithms

In formula 25, N_{atom} and N_{rot} denote the number of atoms and rotatable bonds in the ligand molecule, respectively. The default number of search threads in AutoDock Vina running on the CPU is 8, but the number of physical processors available on the computing platform determines whether all 8 threads can execute concurrently. Table 3.2.1 displays the search depths of different compounds on AutoDock Vina and Vina-GPU, illustrating that the default search depth of each thread in AutoDock Vina is much higher than that in Vina-GPU. Vina-GPU achieves a substantial speed-up ratio of up to 50 times with an average speed-up ratio of 21 times, while ensuring relative docking accuracy. However, Tang Shi-di et al.'s Vina-GPU lacks theoretical justification. To address this, Zhou X et al. [42] provided a rigorous mathematical proof for the convergence of the multi-initial state parallel search strategy and the practicality of reducing the number of iterations of each thread. By increasing the number of iterative search threads and reducing the number of iterations of a single thread, the accuracy of the search solution remains the same or even improves while the total workload remains constant. This proves the feasibility of the multi-initial state parallel search strategy for iterative algorithms both theoretically and practically.

Table 2 Search Depth of Different Compounds on Vina and Vina-GPU

Complex	N_{atom}/N_{rot}	Search Depth of AutoDock Vina	Search Depth of Vina-GPU
1jd0	15/2	15225	1
1bm2	33/7	22365	6
1jyq	60/20	38850	16

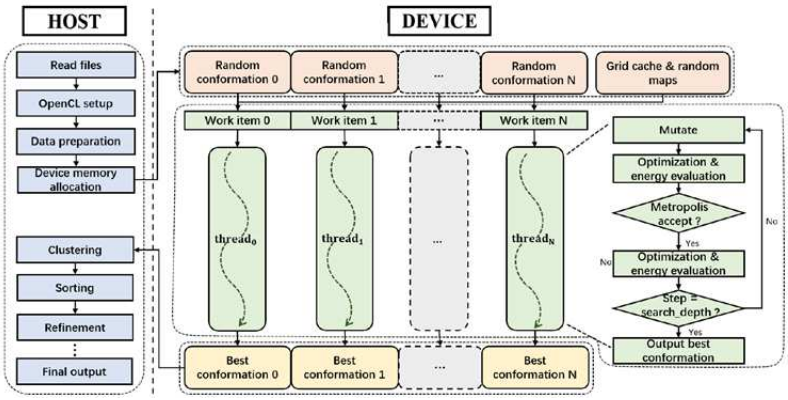

Fig. 5 Architecture Diagram of Vina-GPU [11]

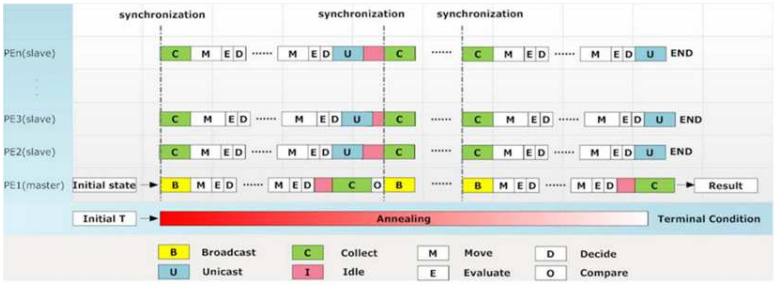

Fig. 6 Timeline of MMC PSA [43]

Furthermore, it is worth noting that the different threads involved in parallel computing may not be entirely independent, and therefore, they can interact with each other. For instance, after a certain number of loop iterations, threads can initiate an interaction, whereby the starting state of the subsequent iteration for all threads is set as the current best state. This technique is commonly known as the Multi-Start algorithm [44, 45] and represents a special type of multi-initial state parallel strategy. Multi-Start combines the local information of individual thread ends with the global information of all other thread ends to expedite convergence. However, it entails a cost in terms of inter-thread communication time.

Computational and modeling complexities represent major challenges in 3D engineering design. In order to address these complexities, Li N et al. [43] utilized MMC

PSA (Multiple Markov Chains Parallel Simulated Annealing). As depicted in Fig.6, MMC PSA employs the main thread for data broadcasting and synchronization, while the auxiliary thread performs the specific calculation process. By deploying only four PEs (Processing Elements, roughly equivalent to four threads), MMC PSA can achieve a time savings of about 70% compared to the sequential version.

The Weapon-Target Assignment (WTA) problem is a combinatorial optimization problem of NP-style, with the aim of optimally assigning weapons to targets in order to minimize the expected value of surviving targets. In order to reduce computation time and improve solution quality, Emrullah SONUC et al. [46] have implemented a parallel Simulated Annealing (pSA) algorithm based on the multi-initial point search strategy and deployed it on the GPU platform, achieving a 250-fold speed-up ratio with 1024 blocks on the GPU, compared to a single-core CPU. Diffdock[47] is the first molecular docking algorithm based on diffusion generative models recently introduced by Corso G and others. It initializes multiple molecular conformations deployed in different inference threads. Each molecular conformation undergoes several steps of generation, resulting in multiple final conformations. These conformations are then sorted and outputted based on a confidence model score for each molecular conformation. While ensuring fast docking speed, Diffdock achieves approximately twice the docking accuracy compared to traditional molecular docking algorithms. MICHAEL T. FELDMANN et al. [48] have proposed a manager-worker-based parallelization algorithm for Quantum Monte Carlo (QMC). The fundamental idea is to execute independent QMC computations and combine statistical information of results from all processors to obtain the global result. The paper dynamically adjusts the workload of each processor based on the Manager-Worker model, so the entire parallel QMC computation is load-balanced.

3.2.2 Concurrent Strategy for Iterative Algorithm with Stochastic Properties

If an iterative algorithm incorporates a stochastic process, a concurrent strategy can be established based on this random process, which refers to the algorithmic module with stochastic properties, such as the random perturbation in the simulated annealing algorithm used to generate a new neighbor. In Fig.7, an iterative calculation consists of three parts: Module 1, the random module, and Module 2. At this point, n parallel random modules can be created after Module 1. Once all the random modules have finished their computation, a randomized result is selected using a predetermined evaluation strategy in the selection module and inputted into Module 2. Selecting a good random result can help accelerate the convergence of the algorithm.

Chazan D et al. [49] have presented a highly parallelized approach for solving unconstrained optimization problems, as illustrated in Fig.8. The proposed method employs the direction of the steepest descent (represented by the purple arrow in the negative gradient direction) as the primary search direction and generates several directions by applying random rotations as auxiliary search directions (indicated by the bright blue arrows). Next, a one-dimensional search is carried out in both the primary and auxiliary search directions concurrently, and the minimum value point is estimated by comparing the size of the objective function. This approach overcomes

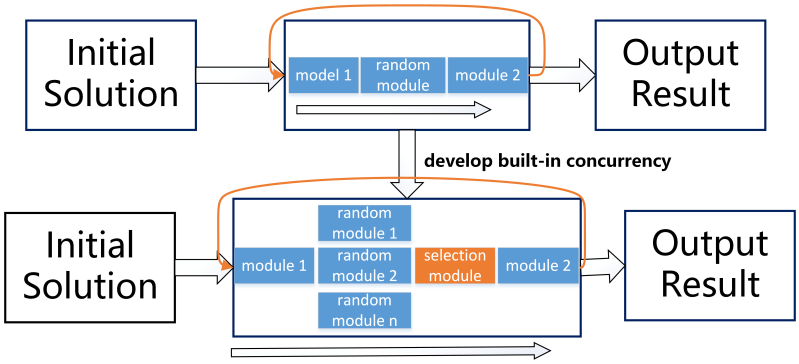

Fig. 7 Concurrency Strategy for Iterative Algorithms with Stochastic Properties

the issue of the lace phenomenon that arises near the minimum value point of the objective function, which is characterized by ridges on the contour lines, and avoids the undesired zigzag search routes.

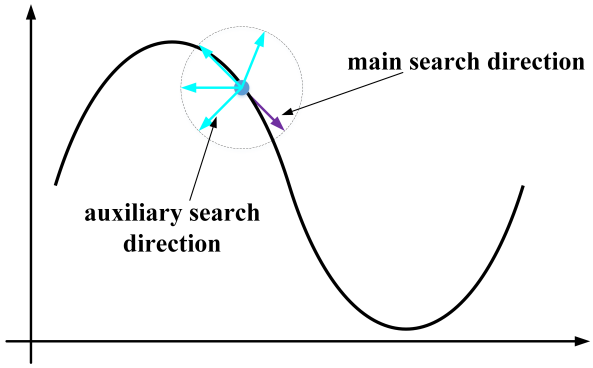

Fig. 8 (One-Dimensional) Main and Auxiliary Collaborative Search

Ana M. Ferreiro-Ferreiro and her colleagues (Ferreiro-Ferreiro et al., [50]) implemented the basin-hopping algorithm, which employs the global simulated annealing technique along with the L-BFGS local optimizer. Fig.9 depicts the specific iterations numbered accordingly. The fundamental concept of the basin-hopping algorithm involves initiating from an initial estimate and then generating several random neighboring solutions at each step of the Metropolis process. Subsequently, a local minimum optimizer, such as L-BFGS, is executed from the latter, and the optimal value obtained is employed as the starting point for the next random perturbation. By intensifying the number of local searches and exploring the neighboring solutions surrounding the current state, the iterative algorithm's convergence can be improved.

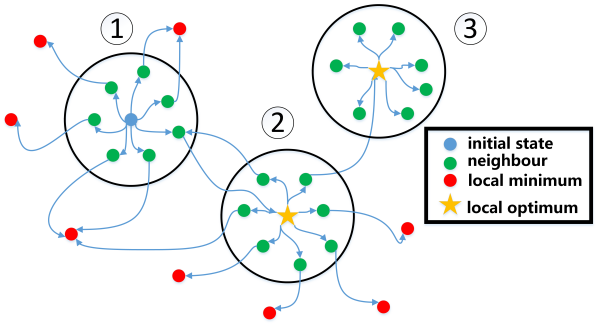

Fig. 9 Schematic Diagram of Basin-Hopping Algorithm (Two-Dimensional) [50]

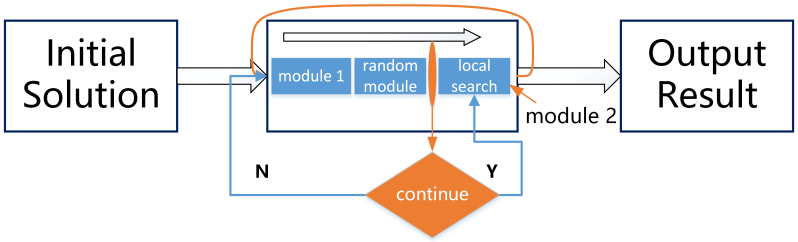

Fig. 10 Reducing the Number of Local Searches in Iterative Algorithms

3.2.3 Reducing the Number of Local Searches in Iterative Algorithms with Randomness

The computational time of the iterative algorithm is primarily influenced by the choice of the initial point and the artificially predetermined number of iterations. Sections 3.2.1 and 3.2.2 center predominantly on selecting a favorable initial point. Furthermore, the convergence of the iterative algorithm can be expedited by avoiding some needless iterative searches based on certain information gleaned from the iterative process. The strategy to curtail the number of local searches in random iterative algorithms lies in diminishing the time of a single iteration. As depicted in Fig.10., assuming that Module 2 is a time-consuming computational module, like a local search algorithm, a selection logic may be added before Module 2 to ascertain whether it is worthwhile to proceed with Module 2's calculation based on a specific strategy. Otherwise, the iteration program returns to the outset, Module 1. As the number of local iterative searches directly determines the computation time of the iterative algorithm, abridging any unnecessary iterative searches during the local search process could quicken the iterative algorithm's convergence.

In 2012, Stephanus Daniel Handoko and his colleagues introduced QuickVina as a means to accelerate AutoDock Vina (Handoko et al., [51]). Given that AutoDock Vina employs a simulated annealing algorithm based on BFGS local optimization, and the BFGS local search represents the most time-consuming part of the AutoDock Vina algorithm, they adopted a novel mathematical strategy to assess the importance of the initial value of each local iteration and avoided superfluous local searches using

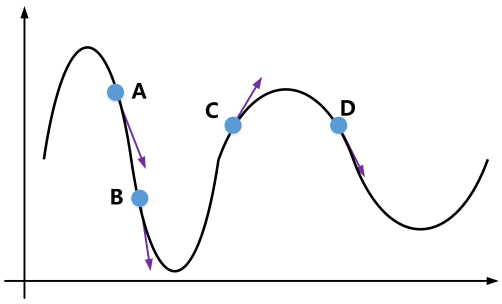

Fig. 11 Schematic Diagram of Reducing the Number of iterations in QuickVina

heuristic methods. The underlying concept is based on Fermat's theorem, which posits that the local minimum constitutes a stationary point in the continuous function. The locally searched point is denoted as the set S, and for the midpoint P of the currently executed searched points, if there exists a neighboring point Q that satisfies formula 27 in two dimensions:

$$f^{'}(P)f^{'}(Q) \leq 0, Q \in S \tag{27}$$

If point P satisfies the test, local search can be conducted; otherwise, it will be skipped. For instance, in Fig.11., point A and point C satisfy formula 27 and therefore pass the test, whereas point A and point B do not. QuickVina, which employs a heuristic algorithm, achieves a maximum speedup of approximately 25 times and an average speedup of 8.34 times compared to AutoDock Vina. Nonetheless, QuickVina's accuracy is compromised when the initial molecular conformations (exhaustiveness) in the AutoDock Vina algorithm are limited. To address this issue, Stephanus Daniel Handoko et al. [52] developed QuickVina2 with the same idea of avoiding unnecessary local searches using heuristic methods but introduced a new consistency check heuristic method to improve significance testing. Assuming that the solution space's dimension is N and that P is the intermediate point of the current local searches, if there exists a point Q among the $2N$neighbors of point P that satisfies:

$$\begin{aligned} sign\left\{\tfrac{\partial f(x)}{\partial x_i}|_{x=P}\right\}.sign\left\{[f(P)-f(Q)]\right. \\ \left.[P_i-Q_i]\right\} \leq 0 \end{aligned} \tag{28}$$

If point P passes the test, local search may proceed; otherwise, it will be skipped. Essentially, formula 28 can be converted to formula 27. According to Lagrange's Mean Value Theorem, for $sign\left\{[f\left(P\right)-f\left(Q\right)]\left[P_i-Q_i\right]\right\}$ in formula 27, we can infer that there exists a point M that satisfies:

$$\frac{\partial f(x)}{\partial x_i}|_{x=M} = sign\left\{[f(P)-f(Q)]\left[P_i-Q_i\right]\right\} \tag{29}$$

Then:

$$sign\left\{\frac{\partial f(x)}{\partial x_i}|_{x=P}\right\}.sign\left\{\frac{\partial f(x)}{\partial x_i}|_{x=M}\right\} \leq 0 \tag{30}$$

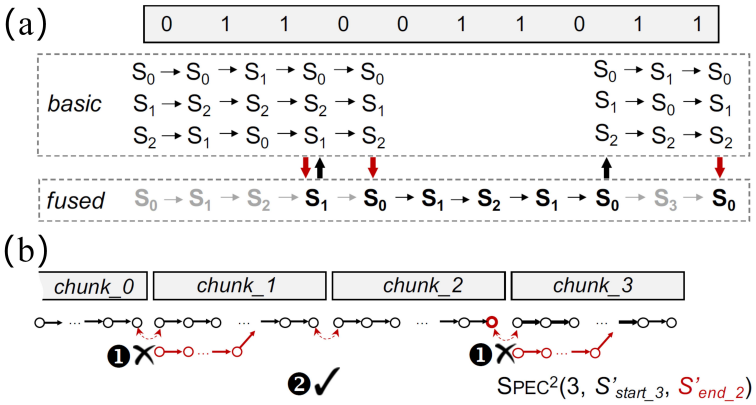

Fig. 12 (a) Path Fusion (b)High-Order Prediction [53]

Stephanus Daniel Handoko et al. [52] achieved a maximum speedup of approximately 20.49x with an exhaustiveness of 8, and an average speedup of about 2.3x. In 2017, Yuguang Mu et al. [54] introduced QuickVina-W, which enables researchers to efficiently and accurately screen large ligand libraries without prior definition of target pockets. Given the vast sampling space of molecular blind docking, QuickVina2 only extends historical point information from local to global threads, whereas QuickVina-W offers high efficiency and precision within a short period of time.

Qiu Jun-qiao et al. [53] have accomplished the parallelization of an extensible Finite States Machine using path fusion and high-order prediction. The concepts of path fusion and higher-order prediction are based on state enumeration and vector fusion, and speculative execution, respectively. In Fig12(a),S_i represents various states of the Finite State Machine, and S_i represents state vectors in the fusion state. The state of "basic" represents the enumeration state, and the state of "fused" represents the fusion state. When the fusion state does not have transition information for the current condition, it jumps to the "basic" state and records the transition information at that time. It then returns to the fusion state to achieve parallelization of the Finite State Machine. To speed up the process of reaching the final state, the entire state sequence is divided into four different chunks, and these chunks are calculated in parallel as shown in Fig.12(b). Finally, starting from the second chunk, the starting state of the chunk and the ending state of the previous chunk are sequentially verified to ensure they are the same. If not, the chunk is re-executed. Path fusion merges multiple initial-state search paths into one vector path, thereby conserving running memory space. High-order prediction optimizes space for time to forecast the subsequent iterations and break internal dependencies. This method can be considered as reducing the number of iterations in the dimension of time.

A Finite State Machine that determines the transition condition can be viewed as an iterative sequence with interdependent adjacent states. Suppose the search space is **S** and there are 3 search threads, and the space is divided into N subspaces based on certain division strategy, represented as $sub_1, sub_2, \ldots, sub_N$. Let $S_1, S_2, \ldots, S_N$ denote N different states. As depicted in Fig.13, Thread 1 acts as the main thread.

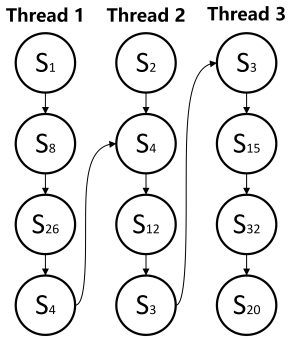

Fig. 13 State Partition and Speculative Execution Based on the Finite State Machine

Initially, perform a parallel search with multiple initial states, such as S_2 as the initial state of Thread 2, andS_3 as the initial state of Thread 3. After several iterations, utilizing speculative execution mechanism, it is discovered that Thread 2 has also reached S_4, and Thread 3 has reached S_3, and the final state achieved is S_{20}. Eliminate the explored state from the solution space to reduce the solution space. Then, Thread 1 takes S_{20} as the initial state, and the initial state of the other two threads is generated by specific strategies, such as randomly selecting the unexplored state or selecting the neighbor of the initial state of the main thread.

3.3 Data Parallelism

The concept of data parallelism is illustrated in Fig.14. The execution procedure of data parallelization for iterative algorithms [55] is as follows: the large input data is partitioned into smaller data inputs and distributed to multiple computing nodes to perform the same task, or the original task is split into several smaller tasks that can be executed independently and assigned to different nodes for parallel processing. Once all processors have completed their calculations, the results are collected and combined to obtain the final solution of the problem.

In 2014, Google introduced MapReduce technology in [12], which is a parallel computing model for big data analysis and processing. MapReduce is dedicated to parallel processing of large-scale data using large clusters of low-cost servers, taking into full account the scalability and system availability [56]. MapReduce adopts the "divide and conquer" programming paradigm, breaking down large-scale tasks into smaller tasks, each of which is executed separately by sub-nodes in the cluster, and finally, the intermediate results of each node are integrated to obtain the final result [57]. The flowchart of MapReduce execution is illustrated in Fig.15. Firstly, the MapReduce function library divides the input file or data into several data segments. In the Map phase, the input data is processed in parallel through the user-defined Map() function, generating corresponding intermediate key-value pairs. In the Shuffle phase, the output of each node's Map is divided and reorganized to reduce bandwidth consumption. In the Reduce phase, the custom Reduce() function is called to categorize and merge the results after Shuffle. Finally, the result is output. Amin Mohebi [58] highlights that

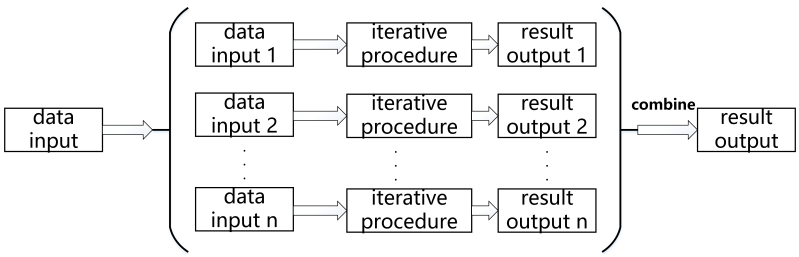

Fig. 14 Schematic Diagram of Data Division

the flexibility, programming simplicity, and fault tolerance of the MapReduce framework have made it a renowned parallel programming model that has gained a lot of attention.

Zinkevich M [59] has executed the parallel implementation of gradient descent algorithms utilizing the MapReduce framework. Algorithm 7 exhibits the single-machine stochastic gradient descent which involves selecting a random data sample for gradient descent calculation within each iteration, given the array samples, the number of iterations, learning factors, and the initial gradient. The parallelized adaptation of LocalSGD is SimuParallelSGD, presented in algorithm 8. This technique randomizes the training set samples, dispatches subsets of data to each machine, executes a random gradient descent in parallel, and subsequently aggregates and averages the corresponding weights on each machine to derive the final outcome.

Algorithm 7 LocalSGD

Require: $S = \{c_i\}_{i=1}^{m}\ (DataSample), T(number\ of\ loops), \eta(learning\ rate), w_0(initial\ gradient)$
Ensure: $w(gradient)$
1: **for** $t = 1, t \leq T, t++$ **do**
2: $\quad j = random\_generate(1, m)$
3: $\quad w_t = w_{t-1} - \eta \nabla f_j(w_{t-1})$
4: **end for**
5: $w = w_T$

Algorithm 8 SimuParallelSGD

Require: $P(Number\ of\ Machines), \eta(Learning\ Rate), w_0(Initial\ Gradient), T = ceil(m/P), S = c_i{}_{i=1}^{m}(Data\ Sample)$
Ensure: $w(gradient)$
1: **for** $i = 1, i \leq p, i++,\ \ concurrently$ **do**
2: $\quad w_{i,0} = w_0$
3: $\quad localSGD(S = \{c_i\}_{i=1}^{m}, T, \eta, w_{i,0})$
4: **end for**
5: $w = \frac{1}{P}\sum_{i=1}^{P} w_{i,T}$

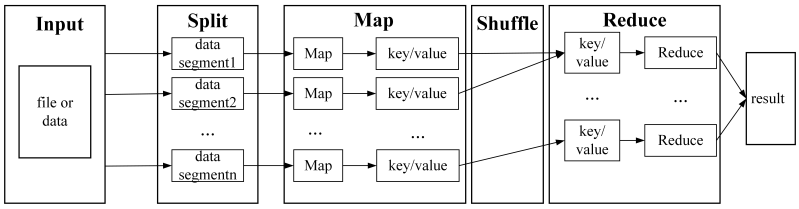

Fig. 15 MapReduce Execution Flow Chart

Leonid proposed the BSF (Bulk Synchronous Farm)-skeleton [60] to facilitate the parallelization of iterative numerical algorithms on cluster systems based on data partitioning. The BSF-skeleton template bears resemblance to the MapReduce framework, but with a master-slave paradigm, where the master thread governs the overall control, and the slave threads are chiefly responsible for specific calculations. Martnez J A et al. [61] delved into the load balancing problem by implementing the iterative algorithm on heterogeneous multiprocessors through data partitioning. They proposed the ADITH algorithm to achieve dynamic load balancing by redistributing the workload in the remaining iterations based on the computing speed of each node in the first iteration. Pelle Jakovis et al. [62] evaluated the MapReduce framework for iterative scientific computing applications, focusing mainly on analyzing the performance and adaptability of three iterative algorithms: Spark, Twister, and HaLoop, while executing iterative algorithms based on the MapReduce framework. The iterative algorithms are categorized into four groups: 1) requiring a single execution of MapReduce, 2) requiring a constant number of sequential MapReduce executions, 3) requiring a separate MapReduce execution in each iteration, and 4) requiring multiple MapReduce executions in each iteration.

Calvin A. Johnson [63] proposed a Tomographic Image Reconstruction technique that involves reconstructing images from a set of measured projections. Building on the computationally intensive iterative reconstruction algorithm by Fourier, the author suggests a parallelization scheme for data division in the projection space and then assigns the data to different processors for computation. Meanwhile, Boglaev I P et al. [64] apply a combination of time discretization and domain decomposition to solve singular perturbed semilinear parabolic problems, decompose the solution space region, and construct a parallelized iterative algorithm.

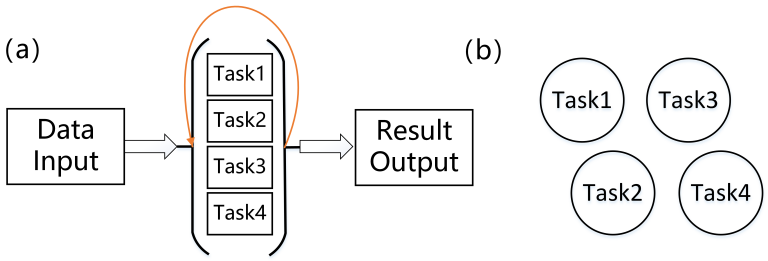

Fig. 16 (a) Task Partitioning Graph (b) Directed Loop Graph in an Ideal Situation

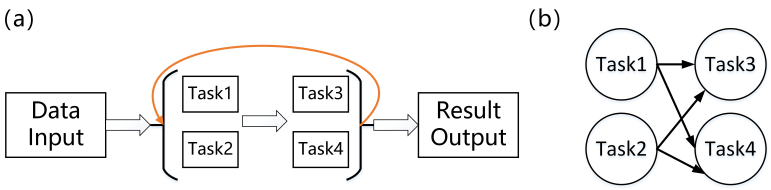

Fig. 17 (a) Task Partitioning Graph and (b) Directed Loop Graph when Task 3 and Task 4 depend on Task 1 and Task 2 respectively

The Iterative Closest Point (ICP) is a distance data processing method commonly used in temperature inspection and range data processing fields. In each ICP iteration [65], the correspondence between two datasets is determined, and the transformation that minimizes the mean square error (MSE) of the correspondence is calculated. The iteration terminates when the MSE is less than a certain threshold or the maximum number of iterations is exceeded. Christian Langis et al. have successfully parallelized ICP for image processing [65]. To compute the distance between the input image and the reference image, the input image is divided into N parts and sent to N sub-threads. Each thread calculates the correspondence between the current image subset and the reference image, sends it to the main thread after each iteration, and collects the increment calculated by the main thread for a new round of iteration.

3.4 Task Parallelism

The concept of task parallelism involves breaking down an iterative algorithm into a directed sequence of multiple tasks, where task interdependence is determined by both the algorithm itself and the task partitioning strategy [13]. Ideally, these tasks are independent of one another and can be executed concurrently. However, in the worst case scenario, any two consecutive tasks in the task sequence may have interdependencies. For instance, an iterative algorithm may be divided into four tasks, namely Task1, Task2, Task3, and Task4, with the ideal scenario illustrated in Fig.16. On the other hand, if there are task interdependencies, such as Task3 being dependent on Task1 and Task2, and Task4 being dependent on Task1 and Task2, this can be represented by Fig.17.

Dan Alistarh et al. [13] put forward a methodology for effectively parallelizing iterative algorithms via a flexible scheduler. The scheduler can relax the rigid order stipulated by sequential algorithms and allow speculative processing of tasks before their dependencies, although this may lead to some loss of correctness. The crux of the matter is the balance between the performance gain brought by the scheduler and the trade-off between the resulting deterministic loss and the wasted workload. The traditional algorithm for structural optimization consists of alternating phases of analysis and optimization. In each iteration, the structural analysis is completed by solving the finite element global matrix, while simultaneously minimizing the objective function. Wang Xi-cheng et al. [66] proposed a parallel method of structural optimization, which breaks down the entire structural optimization task into obtaining potential energy by summing the energy of each element , thereby avoiding the time-consuming work associated with the stiffness matrix. Jacques M. Bahi et al. [67] proposed a methodology

Algorithm 9 asynchronous parallel iterative algorithm of Jacobi iterative scheme

Require: $X^{(0)} = \left(x_1^{(0)}, x_2^{(0)}, x_3^{(0)} \ldots\ldots, x_n^{(0)}\right)$ $(initial\ solution)$
Ensure: $X^* = (x_1, x_2, x_3 \ldots\ldots, x_n)$ $(optimal\ solution)$
1: **while** $k \leq pre\_set\_value$ **do**
2: **for** $each\ thread_i = 1 : p\ \ concurrently$ **do**
3: $Y_i^k = Jacobi\left(Y_1^{k_1}, Y_2^{k_2}, \ldots, Y_p^{k_p}\right)$
4: **if** $||Y_i^k - Y_i^{k-1}|| \leq \varepsilon$ **then**
5: break
6: **end if**
7: **end for**
8: **end while**
9: $X^* = (Y_1, Y_2, \ldots, Y_p)$

for decomposing the solution vectors, and the framework of the iterative algorithm is outlined as follows:

$$\boldsymbol{X^{k+1}} = g\left(\boldsymbol{X^k}\right) \tag{31}$$

where $\boldsymbol{X^k}$ is an n-dimensional vector and g is a function of R^n space. Divide $\boldsymbol{X^k}$ into m blocks, i.e. $\boldsymbol{X_i^k}, i \in 1, \ldots, m$. The function g can also be consistently divided into m constituent blocks G_i. Then formula 31 can be rewritten as:

$$X_i^{k+1} = G_i\left(X_1^k, \ldots\ldots, X_m^K\right), i = 1, \ldots, m \tag{32}$$

This approach enables the concurrent updating of these m constituent blocks using m processors in parallel.

Li Wen-jing et al. [68] proposed a strategy to reduce the communication time of Jacobi iterative computation by introducing a relaxed asynchronous parallel algorithm. The fundamental idea is that in iterative parallel processing, multiple processes can continue iterating for a solution directly using the old iteration values of the previous one or few iterations, without waiting for the results of this iteration carried out by other processors. The Jacobi asynchronous parallel iteration algorithm is presented as follows: Assuming p threads and vector $\boldsymbol{X}$ has n components, Y_i^k represents the solution component produced after the k^{th} iteration in the i^{th} thread. The algorithmic steps are demonstrated in algorithm 9:

Henggang Cui and colleagues [6] actively investigated the staleness of asynchronous updates and established three different models: the Bulk Synchronous Parallel (BSP) model, the Arbitrarily-sized BSP (A-BSP) model, and the Stale Synchronous Parallel (SSP) model. They assumed that each iteration can be divided into six tasks, and denoted the j^{th} task in the i^{th} iteration as $task(i, j)$, with three threads allocated. In the BSP model, updates are not made until the end of a clock cycle (barrier), where one clock cycle is equivalent to one iteration. For example, for $task(4, 6)$ circled in Fig.18, BSP can only guarantee updates for tasks of previous clocks (clock 3 and lower, with no shadows). The A-BSP model introduces $work - per - clock(WPC)$, which represents the amount of work (the number of iterations) completed in each cycle. As shown in

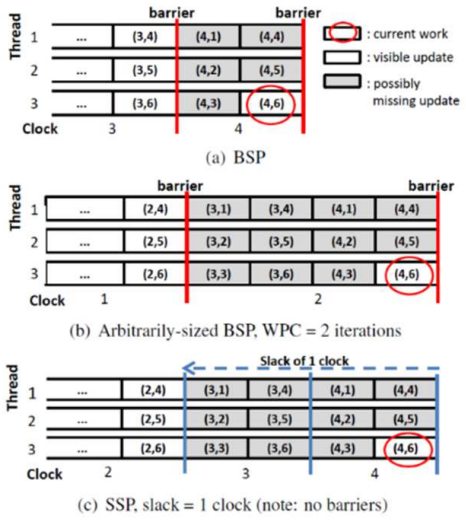

Fig. 18 BSPA-BSP and SSP models [13]

Fig.18(b), $WPC = 2$. Although the A-BSP model reduces communication work per clock through more computation, it still suffers from the main problem of the BSP model: a single slow thread will cause all threads to wait, a problem that deteriorates with increased parallelization. Consequently, the SSP model [69] was proposed, which defines *slack* as an explicitly introduced relaxation parameter that specifies how many obsolete clocks the current thread has for the view of shared state. Assuming that a thread is at clock t and slack is s, the thread can view all updates from clock 1 to $t - s - 1$, as shown in the subgraph (c) in Fig.18 with $slack = 1$.

When solving parabolic equations numerically, the explicit differencing scheme can be deployed on parallel computers, albeit with stringent constraints on the time step. Conversely, the fully implicit differencing scheme, lacking conditional stability and strict constraints, necessitates the solution of a global linear system at each time layer, rendering it unsuitable for direct implementation on parallel computers. To overcome this obstacle, Wenrui Hao et al. [70] proposed a parallel iterative method in the fully implicit schemes for the one-dimensional and two-dimensional problem of parabolic equations. This method is based on the idea of domain decomposition, which divides the linear equation system into numerous non-overlapping subsystems. Additionally, the authors demonstrated that this method enables the difference scheme solution to converge to the implicit differencing scheme.

Beam tracking is a computationally intensive endeavor, comprising primarily of three algorithms: 1) beam tracing, 2) the generation of beam octrees, and 3) grid generation. The first and third stages are iterative algorithms, while the second stage is a recursive algorithm. Beam tracking is a method that initiates twenty beams using a dodecahedron (centered at the source position). These beams are then partitioned based on the geometric shape of the surfaces they impinge upon, giving rise to reflected and refracted beams from the separated incident light. In the second stage, all beams

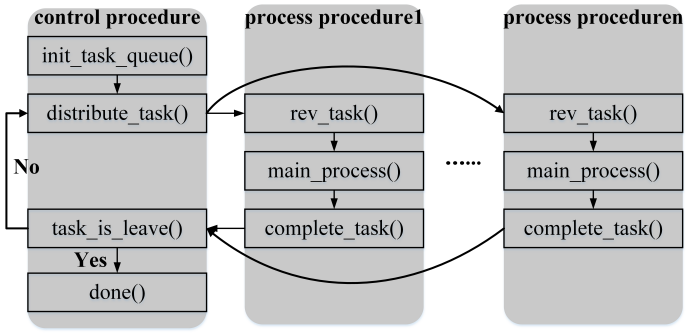

Fig. 19 Schematic Diagram of Master Thread - Worker Threads

generated by beam tracing are placed in an octree, which accelerates the spatial search of the beam. In the third stage, the spatial distribution of sound intensity levels is delineated in the form of a point grating. For each point in the grating, a process involving the filtration of an octree is undertaken to obtain the bundle encompassing said point. Subsequently, the summation of each beam originating from the point's position is performed to ascertain the level of sound intensity at that point. Sikora M et al. [71] implemented the parallelization of the first and third stages using the master-worker strategy. In Fig.19, the master thread serves as the control thread responsible for task distribution, while the worker thread is responsible for specific processing. The control process commences with the creation of a task queue. Then, the master thread simultaneously launches all worker threads by assigning each of them a task from the shared task queue. The calculation is deemed complete when all tasks are assigned, and all worker threads have finished processing.

The iterative reconstruction algorithm for computer tomography can yield highly precise sectional images. However, its substantial computational expense renders it impractical for clinical application. To address this issue, Daniel B. Keesing et al. [72] presented a parallel implementation of the iterative algorithm for full 3D image reconstruction. The approach involves updating image slices concurrently without the need for intercommunication or public projection passing. Specifically, for a volume of 16 slices and four available processors, with a minimum slice interval of 4, the first update will be for 0, 4, 8, 12, followed by the second update for 1, 5, 9, 13, enabling independent parallel computations by the processors.

4 Convergence of Parallel Iterative Algorithms

4.1 Symbols And Preliminary Knowledge

Before delving into the convergence of iterative algorithms, let us first define and introduce relevant symbols. Let R denote the set of real numbers, $\mathbb{N}^+$ denote the set of positive integers, andR^n denote the n-dimensional real space. Suppose that the n-dimensional vector $\boldsymbol{X} = (\boldsymbol{X_1}, \ldots, \boldsymbol{X_m}) \in R^n$, where $\boldsymbol{X_i} \in R^{n_i}, m \in \mathbb{N}^+$, and

$n_i \in \mathbb{N}^+$ satisfy:

$$\sum_{i=1}^{m} n_i = n \tag{33}$$

Define the block infinity norm as follows:

$$||\boldsymbol{X}||_b^w = \max_{1 \le i \le m} \frac{||\boldsymbol{X_i}||_i}{w_i} \tag{34}$$

Where each w_i is a positive scaling factor, and $||.||_i$ represents the norm defined on R^{n_i}. Notably, if $i = 1, \ldots, m$, then $n_i = 1$, and the above formula simplifies to the commonly known infinity norm **??**:

$$||\boldsymbol{X}||_\infty^w = \max_{1 \le i \le n} \frac{||\boldsymbol{X_i}||}{w_i} \tag{35}$$

For an iterative sequence $\boldsymbol{X(k)} \in R^n$, if there are $\boldsymbol{X^*}$ and $\mu \in (0, 1)$ satisfy the following formula, the iterative sequence is considered to be linearly convergent to $\boldsymbol{X^*}$.

$$\lim_{k \to \infty} \frac{||\boldsymbol{X}(\boldsymbol{k+1}) - \boldsymbol{X^*}||}{||\boldsymbol{X(k)} - \boldsymbol{X^*}||} = \mu \tag{36}$$

$||.||$ is the norm defined on R^n.

4.2 Convergence of Iterative Algorithms

Considering the iterative algorithm in the following format:

$$\boldsymbol{X_i}(\boldsymbol{k+1}) = f_i(\boldsymbol{X_1(k)}, \ldots, \boldsymbol{X_m(k)}) \tag{37}$$

where $i = 1, 2, \ldots, m$, and $\boldsymbol{X_i} \in R^{n_i}$, f_i is a mapping function of $R^n \to R^{n_i}$, and $n_1 + n_2 + \ldots + n_m = n$. If $\boldsymbol{X^*} = (\boldsymbol{X_1^*}, \boldsymbol{X_2^*}, \ldots, \boldsymbol{X_m^*})$ exists and satisfies the formula 36:

$$\boldsymbol{X_i^*} = f_i(\boldsymbol{X_1^*}, \boldsymbol{X_2^*}, \ldots, \boldsymbol{X_m^*}) \qquad \forall i = 1, \ldots, m \tag{38}$$

then $\boldsymbol{X^*}$ is considered as the fixed point of function $f(\boldsymbol{X}) = (f_1(\boldsymbol{X}), \ldots, f_m(\boldsymbol{X}))$. If function $f_i(\boldsymbol{X})$ is continuous at $\boldsymbol{X^*}$, and the iterative sequence $\boldsymbol{X_i(t)}$ excited by formula 37 converges to $\boldsymbol{X_i^*}$, then $\boldsymbol{X^*}$ is considered as the fixed point of function f. Formula 38 can be regarded as a network of m nodes, with each node responsible for calculating a sub-vector of vector $\boldsymbol{X}$ and finally finding a global stable point.

4.3 Convergence of the Synchronous Iterative Algorithm

As the synchronous iterative algorithm retains the original iterative algorithm's execution order and distributes the single-iteration algorithm's computational load among various computing nodes, it updates each node's data via communication after all nodes complete their calculations in each iteration. This process repeats until the

termination condition is met. Consequently, the synchronous iterative algorithm's convergence is in line with the original iterative algorithm's convergence[73].

The multi-initial state search strategy is a specific instance of the synchronous iterative algorithm. It has two distinctive features: firstly, it duplicates multiple complete iterative algorithm computation threads, and secondly, the initial state of each iteration thread is randomly generated from the solution space. Without diminishing the search depth of each thread, the convergence of the multi-initial state search strategy can be readily inferred from the convergence of the original iterative algorithm. Let $p(d)$ denote the probability that the iteration sequence $X(k)$ converges to the fixed point from the initial point after d iterations. It can be inferred that convergent iterative algorithms converge at the fixed point with probability 1, as follows:

$$\lim_{d\to\infty} p(d) = 1 \tag{39}$$

Assuming that the number of initial states is denoted by N, and let $P(d)$ represent the probability of convergence to the fixed point of the multi-initial state iterative sequence after d iterations, then:

$$P(d) = 1 - (1 - p(d))^N \tag{40}$$

From the above formula, it is known that:

$$\lim_{d\to\infty} P(d) = 1 \tag{41}$$

Therefore, the multi-initial state parallel search strategy guarantees convergence to the global fixed points with probability 1. However, in order to achieve acceleration benefits, it is necessary to reduce the search depth of each thread. Zhou X et al. [42] analyzed the effectiveness of the multi-initial state simulated annealing algorithm from two perspectives: the convergence of the algorithm and the probability of finding the optimal solution. They proved that, given a fixed total workload (the number of search steps), increasing the search width (the number of initial states) and reducing the search depth (the number of iterations) of each worker thread would not compromise the quality of the searched solution.

4.4 convergence of Asynchronous Iterative Algorithm

The asynchronous implementation of formula 37 represents an iterative algorithm where each node updates its own state at its own pace, utilizing data that may be outdated and obtained from other nodes. This type of iteration is mathematically represented as follows:

$$x_i(k+1) = \begin{cases} f_i\left(x_1\left(\pi_1^i(k)\right), \ldots, x_m\left(\pi_m^i(k)\right)\right), k \in K^i \\ x_i(k), k \notin K^i \end{cases} \tag{42}$$

where K^i represents the set of times that node i needs to perform updates, and $\pi_j^i(k)$ represents the latest effective data that node j can provide to node i at the

k^{th} iteration, so $(k-\pi_j^i(k))$ can be regarded as the delay from node j to node i. For $\forall k \in \mathbb{N}^+, 0 \leq \pi_j^i(k) \leq k$. If for all i and j, $\pi_j^i(k) = k, and\ K^i \in \mathbb{N}^+$, then the asynchronous iterative algorithm at this point is the synchronous version of the iterative algorithm.

Based on the assumption of communication delay and update rate, asynchronous algorithms can be divided into fully asynchronous and partially asynchronous [35].

Definition 1 (Fully Asynchronous)**.**

a) $\forall\ i$, set K^i is an infinite subset of $\mathbb{N}^+$.
b) If the set $\{k_q\}$ is a sequence of elements in the set K^i which tends to be infinite, and for $\forall\ i, j$, $\lim_{q\to\infty} \pi_j^i(k_q) = \infty$.

Assuming 1a) Ensuring that all nodes will continue to update without interruption, assuming 1b) signifies that outdated information will eventually be eradicated. In the case of complete asynchrony, $(k-\pi_j^i(k))$ may become infinite, which is also a key point that differs from partial asynchrony.

Definition 2 (Partially Asynchronous. There is a positive integer B)**.**

a) For each i and$k \in \mathbb{N}^+$, at least one element in the set $k, k+1, \ldots, k+B-1$ belongs to K^i.
b) For $\forall\ i, j\ \forall\ k \in K^i$, and the following inequality is satisfied:

$$0 \leq k - \pi_j^i(k) \leq B-1 \tag{43}$$

c) For all i, if $k \in K^i, \pi_i^i(k) = k$.

Assuming 2a) imposes a requirement that there must be an update within B steps, assuming 2b) constrains the staleness of information to not exceed 3535 steps, and assuming 2c) ensures that each node iterates its calculations using its most current data. If $B = 1$, then the model degenerates into a synchronous iterative algorithm.

The excessive utilization of outdated information can lead to a deceleration or even divergence in the convergence rate of asynchronous iterative algorithms. Bert Sekas [15] established a general convergence theorem for asynchronous iterative scheme 42. Note $E_i = R^{n_i}$, and the whole solution space can be expressed as $E = E_1 \times \ldots \times E_m$. The mathematical model established by BertSekas is as follows:

Theorem 1. *Assume the set $E_k \sqsubseteq E$, and it satisfies:*

1) $E^k = E_1^k \times E_2^k \times \ldots \times E_m^k, k \in \mathbb{N}^+$
2) $f\left(E^k\right) \subseteq E^{k+1} \subseteq E^k$
3) x^* *satisfies:*

$$y^k \in E^k, k \in \mathbb{N}^+ \to \lim_{k\to\infty} y^k = x^* \tag{44}$$

Based on theorem 1, the more widely used inference is formula 45; if $\mu \in [0,1]$, and it satisfies [72]:

$$||f(x) - x^*||_b^w \leq \mu\, ||x - x^*||_b^w, \quad \forall x \in R^n \tag{45}$$

the function f is regarded as the pseudo-compression under the maximum block norm [74]. It is shown that [75] if the function f satisfies the pseudo-compression under the maximum block norm, the asynchronous algorithm can converge to the fixed point and can tolerate communication and calculation delay at any size. In particular, for partial asynchrony, as long as the asynchronous factor B is reasonably set, the iterative algorithm is convergent in most cases [34].

4.5 Convergence Detection of Asynchronous Iterative Algorithm

At present, asynchronous parallel algorithms are studied in order to take full advantage of large-scale parallel structures and distributed platforms. In a distributed environment, much of the efficiency of parallel algorithms relies on inter-process communication. The distributed environment also raises concerns regarding the efficiency and precision of the convergence process during asynchronous parallel iteration in such distributed settings. Furthermore, with the escalation of communication load, there exists no straightforward and efficient method to compute consistent residuals from the potential global solution components. [16].

In this paper, a distributed convergence detection method for parallel iterative algorithms is proposed for two main reasons [5]. Firstly, the most common parallel iterative algorithm fundamentally corresponds to a decentralized asynchronous iterative algorithm. Secondly, centralization is not feasible when using these algorithms, as asynchronous iteration does not obstruct communication between threads, and the residuals of global vectors are challenging to use as a convergence criterion for global convergence detection. In most cases, when the global solution approaches a global fixed point, the solution may exhibit oscillations in the vicinity of the prescribed threshold, potentially leading to premature detection of local convergence. Therefore, devising a robust asynchronous iterative algorithm for convergence detection is of paramount importance in cases of locally convergent detection.

There are two primary techniques for detecting convergence in asynchronous iterative algorithms. The first approach is a leader election protocol that employs a tree topology and includes message-cancellation mechanisms to handle false convergence. Jacques M. Bahi addresses convergence detection of asynchronous iterative algorithms from two perspectives in [5]: local convergence detection and global convergence detection. To avoid premature detection of local convergence, a common heuristic method assumes that local convergence is achieved when the current node executes a set number of consecutive iterations below a specified accuracy threshold. The concept of global convergence detection relies on a leader election protocol that begins on a single leader node and is then propagated to all its neighboring nodes. Other worker nodes send information regarding the completion of local convergence to the leader node. If all nodes are determined to have reached local convergence, the asynchronous iterative algorithm is deemed to have achieved global convergence.

The second approach involves modifying the asynchronous iterative algorithm to ensure that it terminates in a finite time, and then using traditional termination detection protocols for distributed systems (such as [76, 77]). These protocols are designed for parallel applications that can be executed in a finite number of steps. In this

method, if any node reaches a local condition (i.e., local convergence), it will stop sending new data to its neighbors in the communication graph. The termination condition can then be expressed as all nodes being in a local condition and no messages being in transit. Evans D J [78] has proposed restoring asynchronous iterations to synchronous iterations at a certain node during computation execution, thereby ensuring that local convergence persists throughout the process.

5 conclution

In summary, this paper provides an overview of the parallelization strategies for iterative algorithms. It begins by introducing the mathematical description of the iterative algorithm and categorizes it into synchronous and asynchronous iterations. The paper then presents five classical iterative algorithms and their respective application fields, including the backpropagation algorithm in neural networks, the BFGS algorithm, the NIPALS-PCA algorithm, the simulated annealing algorithm, and the K-means clustering algorithm.

Four parallelization strategies for iterative algorithms are then discussed. The first strategy is to use intrinsically concurrently computable logical units in iterative algorithms. The second is the multi-initial state parallel search strategy, which includes detailed discussions on the parallelization method of iterative algorithms with random modules and how to reduce the number of iterations in the iterative algorithm. The third is the data parallelism strategy, elaborated upon using the MapReduce framework. The fourth is the task parallelism strategy, which discusses how to divide the iterative algorithm into different tasks from the perspective of task scheduling.

Furthermore, the paper addresses the convergence problem of parallel iteration algorithms. It not only analyzes the convergence of the multi-initial state parallel search strategy but also demonstrates the convergence of synchronous iteration and asynchronous iteration through mathematical models. The paper also discusses the research on convergence detection of asynchronous iterative algorithms.

Currently, there is a lack of literature reviews on the parallelization strategies for iterative algorithms in Chinese. It is hoped that this paper can serve as a strong reference for future research in this area.

References

[1] Suthaharan, S.: Machine learning models and algorithms for big data classification. Integr. Ser. Inf. Syst **36**, 1–12 (2016)

[2] Fletcher, R.: Practical Methods of Optimization. John Wiley & Sons, inc. (2000)

[3] Kirkpatrick, S., Gelatt Jr, C.D., Vecchi, M.P.: Optimization by simulated annealing. science **220**(4598), 671–680 (1983)

[4] Lloyd, S.: Least squares quantization in pcm. IEEE transactions on information theory **28**(2), 129–137 (1982)

[5] Bahi, J.M., Contassot-Vivier, S., Couturier, R., Vernier, F.: A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. IEEE Transactions on Parallel and Distributed Systems **16**(1), 4–13 (2005)

[6] Cui, H., Cipar, J., Ho, Q., Kim, J.K., Lee, S., Kumar, A., Wei, J., Dai, W., Ganger, G.R., Gibbons, P.B., *et al.*: Exploiting bounded staleness to speed up big data analytics. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14), pp. 37–48 (2014)

[7] Gorgulla, C., Boeszoermenyi, A., Wang, Z.-F., Fischer, P.D., Coote, P.W., Padmanabha Das, K.M., Malets, Y.S., Radchenko, D.S., Moroz, Y.S., Scott, D.A., *et al.*: An open-source drug discovery platform enables ultra-large virtual screens. Nature **580**(7805), 663–668 (2020)

[8] Xie, Y., Ding, L., Zhou, A., Chen, G.: An optimized face recognition for edge computing. In: 2019 IEEE 13th International Conference on ASIC (ASICON), pp. 1–4 (2019). IEEE

[9] Keesing, D.B., O'Sullivan, J.A., Politte, D.G., Whiting, B.R.: Parallelization of a fully 3d ct iterative reconstruction. In: 3rd IEEE International Symposium on Biomedical Imaging: Nano to Macro, 2006., pp. 1240–1243 (2006). IEEE

[10] Farivar, R., Rebolledo, D., Chan, E., Campbell, R.H.: A parallel implementation of k-means clustering on gpus. In: Pdpta, vol. 13, pp. 212–312 (2008)

[11] Tang, S., Chen, R., Lin, M., Lin, Q., Zhu, Y., Ding, J., Hu, H., Ling, M., Wu, J.: Accelerating autodock vina with gpus. Molecules **27**(9), 3041 (2022)

[12] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)

[13] Alistarh, D., Brown, T., Kopinsky, J., Nadiradze, G.: Relaxed schedulers can efficiently parallelize iterative algorithms. In: Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, pp. 377–386 (2018)

[14] Blathras, K., Szyld, D.B., Shi, Y.: Timing models and local stopping criteria for asynchronous iterative algorithms. Journal of Parallel and Distributed Computing **58**(3), 446–465 (1999)

[15] Bertsekas, D.P.: Distributed asynchronous computation of fixed points. Mathematical Programming **27**(1), 107–120 (1983)

[16] Magoules, F., Gbikpi-Benissan, G.: Distributed convergence detection based on global residual error under asynchronous iterations. IEEE transactions on parallel and distributed systems **29**(4), 819–829 (2017)

[17] Xu, Q., Wang, W.: A new parallel iterative algorithm for solving 2d poisson

equation. Numerical Methods for Partial Differential Equations **27**(4), 829–853 (2011)

[18] Danowitz, A., Kelley, K., Mao, J., Stevenson, J., Horowitz, M., DB, C.: Recording microprocessor history. ACM Queue Magazine **10**(4) (2002)

[19] Coşkun, M., Uçar, A., Yildirim, Ö., Demir, Y.: Face recognition based on convolutional neural network. In: 2017 International Conference on Modern Electrical and Energy Systems (MEES), pp. 376–379 (2017). IEEE

[20] Batool, T., Abbas, S., Alhwaiti, Y., Saleem, M., Ahmad, M., Asif, M., Elmitwal, N.S.: Intelligent model of ecosystem for smart cities using artificial neural networks. Intelligent Automation & Soft Computing **30**(2) (2021)

[21] Dixon, M., Klabjan, D., Bang, J.H.: Classification-based financial markets prediction using deep neural networks. Algorithmic Finance **6**(3-4), 67–77 (2017)

[22] McNutt, A.T., Francoeur, P., Aggarwal, R., Masuda, T., Meli, R., Ragoza, M., Sunseri, J., Koes, D.R.: Gnina 1.0: molecular docking with deep learning. Journal of cheminformatics **13**(1), 1–20 (2021)

[23] Andrei, N.: An adaptive scaled bfgs method for unconstrained optimization. Numerical Algorithms **77**(2), 413–432 (2018)

[24] Dai, Y.-H.: A perfect example for the bfgs method. Mathematical Programming **138**, 501–530 (2013)

[25] Bro, R., Smilde, A.K.: Principal component analysis. Analytical methods **6**(9), 2812–2831 (2014)

[26] Bro, R., Smilde, A.K.: Principal component analysis. Analytical methods **6**(9), 2812–2831 (2014)

[27] Lazcano, R., Madroñal, D., Fabelo, H., Ortega, S., Salvador, R., Callicó, G.M., Juárez, E., Sanz, C.: Parallel implementation of an iterative pca algorithm for hyperspectral images on a manycore platform. In: 2017 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 1–6 (2017). IEEE

[28] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. The journal of chemical physics **21**(6), 1087–1092 (1953)

[29] Trott, O., Olson, A.J.: Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. Journal of computational chemistry **31**(2), 455–461 (2010)

[30] Feyzmahdavian, H.R., Johansson, M.: On the convergence rates of asynchronous iterations. In: 53rd IEEE Conference on Decision and Control, pp. 153–159 (2014).

IEEE

[31] Magoules, F., Gbikpi-Benissan, G.: Distributed convergence detection based on global residual error under asynchronous iterations. IEEE transactions on parallel and distributed systems **29**(4), 819–829 (2017)

[32] Martínez, J.A., Garzón, E.M., Plaza, A., García, I.: Automatic tuning of iterative computation on heterogeneous multiprocessors with adithe. The Journal of Supercomputing **58**, 151–159 (2011)

[33] Bertsekas, D.P., Tsitsiklis, J.N.: Convergence rate and termination of asynchronous iterative algorithms. In: Proceedings of the 3rd International Conference on Supercomputing, pp. 461–470 (1989)

[34] Guyeux, C.: Convergence versus divergence behaviors of asynchronous iterations, and their applications in concrete situations. Mathematical and Computational Applications **25**(4), 69 (2020)

[35] Bertsekas, D.P., Tsitsiklis, J.N.: Some aspects of parallel and distributed iterative algorithmsa survey. Automatica **27**(1), 3–21 (1991)

[36] Langford, J., Smola, A., Zinkevich, M.: Slow learners are fast. arXiv preprint arXiv:0911.0491 (2009)

[37] Tritsiklis, J.N.: A comparison of jacobi and gauss-seidel parallel iterations. Applied Mathematics Letters **2**(2), 167–170 (1989)

[38] Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**(7), 33–38 (2008)

[39] Song, K., Li, W., Zhang, B., Yan, L., Wang, X.: Parallel design and implementation of jacobi iterative algorithm based on ternary optical computer. The Journal of Supercomputing **78**(13), 14965–14990 (2022)

[40] Andrecut, M.: Parallel gpu implementation of iterative pca algorithms. Journal of Computational Biology **16**(11), 1593–1599 (2009)

[41] Ling, M., Lin, Q., Chen, R., Qi, H., Lin, M., Zhu, Y., Wu, J.: Vina-fpga: A hardware-accelerated molecular docking tool with fixed-point quantization and low-level parallelism. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **31**(4), 484–497 (2022)

[42] Zhou, X., Ling, M., Lin, Q., Tang, S., Wu, J., Hu, H.: Effectiveness analysis of multiple initial states simulated annealing algorithm, a case study on the molecular docking tool autodock vina. IEEE/ACM Transactions on Computational Biology and Bioinformatics (2023)

[43] Li, N., Cha, J., Lu, Y.: A parallel simulated annealing algorithm based on

functional feature tree modeling for 3d engineering layout design. Applied Soft Computing **10**(2), 592–601 (2010)

[44] Vincent, F.Y., Lin, S.-W.: Multi-start simulated annealing heuristic for the location routing problem with simultaneous pickup and delivery. Applied soft computing **24**, 284–290 (2014)

[45] Lin, S.-W.: Solving the team orienteering problem using effective multi-start simulated annealing. Applied Soft Computing **13**(2), 1064–1073 (2013)

[46] Sonuc, E., Baha, S., Bayir, S.: A parallel simulated annealing algorithm for weapon-target assignment problem. International Journal of Advanced Computer Science and Applications **8**(4) (2017)

[47] Corso, G., Jing, B., Barzilay, R., Jaakkola, T., *et al.*: Diffdock: Diffusion steps, twists, and turns for molecular docking. In: International Conference on Learning Representations (ICLR 2023) (2023)

[48] Feldmann, M.T., Cummings, J.C., Kent IV, D.R., Muller, R.P., Goddard III, W.A.: Manager–worker-based model for the parallelization of quantum monte carlo on heterogeneous and homogeneous networks. Journal of computational chemistry **29**(1), 8–16 (2008)

[49] Chazan, D., Miranker, W.: A nongradient and parallel algorithm for unconstrained minimization. SIAM Journal on control **8**(2), 207–217 (1970)

[50] Ferreiro-Ferreiro, A.M., García-Rodríguez, J.A., Souto, L., Vázquez, C.: Basin hopping with synched multi l-bfgs local searches. parallel implementation in multicpu and gpus. Applied Mathematics and Computation **356**, 282–298 (2019)

[51] Handoko, S.D., Ouyang, X., Su, C.T.T., Kwoh, C.K., Ong, Y.S.: Quickvina: accelerating autodock vina using gradient-based heuristics for global optimization. IEEE/ACM transactions on computational biology and bioinformatics **9**(5), 1266–1272 (2012)

[52] Alhossary, A., Handoko, S.D., Mu, Y., Kwoh, C.-K.: Fast, accurate, and reliable molecular docking with quickvina 2. Bioinformatics **31**(13), 2214–2216 (2015)

[53] Qiu, J., Sun, X., Sabet, A.H.N., Zhao, Z.: Scalable fsm parallelization via path fusion and higher-order speculation. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 887–901 (2021)

[54] Hassan, N.M., Alhossary, A.A., Mu, Y., Kwoh, C.-K.: Protein-ligand blind docking using quickvina-w with inter-process spatio-temporal integration. Scientific reports **7**(1), 15451 (2017)

[55] Ekanayake, J., Pallickara, S., Fox, G.: Mapreduce for data intensive scientific analyses. In: 2008 IEEE Fourth International Conference on eScience, pp. 277–284 (2008). IEEE

[56] Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for mapreduce. In: Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 938–948 (2010). SIAM

[57] Liao, Q., Yang, F., Zhao, J.: An improved parallel k-means clustering algorithm with mapreduce. In: 2013 15th IEEE International Conference on Communication Technology, pp. 764–768 (2013). IEEE

[58] Mohebi, A., Aghabozorgi, S., Ying Wah, T., Herawan, T., Yahyapour, R.: Iterative big data clustering algorithms: a review. Software: Practice and Experience **46**(1), 107–129 (2016)

[59] Zinkevich, M., Weimer, M., Li, L., Smola, A.: Parallelized stochastic gradient descent. Advances in neural information processing systems **23** (2010)

[60] Sokolinsky, L.B.: Bsf-skeleton: A template for parallelization of iterative numerical algorithms on cluster computing systems. MethodsX **8**, 101437 (2021)

[61] Martínez, J.A., Garzón, E.M., Plaza, A., García, I.: Automatic tuning of iterative computation on heterogeneous multiprocessors with adithe. The Journal of Supercomputing **58**, 151–159 (2011)

[62] Jakovits, P., Srirama, S.N.: Evaluating mapreduce frameworks for iterative scientific computing applications. In: 2014 International Conference on High Performance Computing & Simulation (HPCS), pp. 226–233 (2014). IEEE

[63] Johnson, C.A., Sofer, A.: A data-parallel algorithm for iterative tomographic image reconstruction. In: Proceedings. Frontiers' 99. Seventh Symposium on the Frontiers of Massively Parallel Computation, pp. 126–137 (1999). IEEE

[64] Boglaev, I., Sirotkin, V.: Iterative domain decomposition algorithms for the solution of singularly perturbed parabolic problems. Computers & Mathematics with Applications **31**(10), 83–100 (1996)

[65] Langis, C., Greenspan, M., Godin, G.: The parallel iterative closest point algorithm. In: Proceedings Third International Conference on 3-D Digital Imaging and Modeling, pp. 195–202 (2001). IEEE

[66] Xicheng, W., Guixu, M.: A parallel iterative algorithm for structural optimization. Computer Methods in Applied Mechanics and Engineering **96**(1), 25–32 (1992)

[67] Bahi, J.M., Contassot-Vivier, S., Couturier, R.: Coupling dynamic load balancing

with asynchronism in iterative algorithms on the computational grid. In: Proceedings International Parallel and Distributed Processing Symposium, p. 9 (2003). IEEE

[68] Li, W., Liu, Z., Lan, Z.: Bfgs relaxation asynchronous parallel algorithm of unconstrained optimization problems. Jisuanji Gongcheng yu Yingyong(Computer Engineering and Applications) **48**(17), 44–47 (2012)

[69] Cipar, J., Ho, Q., Kim, J.K., Lee, S., Ganger, G.R., Gibson, G., Keeton, K., Xing, E.: Solving the straggler problem with bounded staleness. In: 14th Workshop on Hot Topics in Operating Systems (HotOS XIV) (2013)

[70] Hao, W., Zhu, S.: Parallel iterative methods for parabolic equations. International Journal of Computer Mathematics **86**(3), 431–440 (2009)

[71] Sikora, M., Mateljan, I.: A method for speeding up beam-tracing simulation using thread-level parallelization. Engineering with Computers **30**, 679–688 (2014)

[72] Keesing, D.B., O'Sullivan, J.A., Politte, D.G., Whiting, B.R.: Parallelization of a fully 3d ct iterative reconstruction. In: 3rd IEEE International Symposium on Biomedical Imaging: Nano to Macro, 2006., pp. 1240–1243 (2006). IEEE

[73] Reisizadeh, A., Taheri, H., Mokhtari, A., Hassani, H., Pedarsani, R.: Robust and communication-efficient collaborative learning. Advances in Neural Information Processing Systems **32** (2019)

[74] Bertsekas, D., Tsitsiklis, J.: Parallel and Distributed Computation: Numerical Methods. Athena Scientific, inc. (2015)

[75] Frommer, A., Szyld, D.B.: On asynchronous iterations. Journal of computational and applied mathematics **123**(1-2), 201–216 (2000)

[76] Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Information Processing Letters **11**(1), 1–4 (1980)

[77] Francez, N., Rodeh, M.: Achieving distributed termination without freezing. IEEE Transactions on Software Engineering (3), 287–292 (1982)

[78] Evans, D.J., Chikohora, S.: Convergence testing on a distributed network of processors. International journal of computer mathematics **70**(2), 357–378 (1998)