Chapter 5
Parallelization of Intelligent Optimization Algorithm

Today, different kinds of hardware for computing are more and more powerful, in accordance with large scaled complex computing tasks. From multi-core computer to clusters, various parallel architectures are developed for computing acceleration. In terms of the long time iteration and population based mechanism of intelligent optimization algorithm, parallelization is attainable and imperative in many complex optimization. Among the existing parallel methods developed for intelligent optimization algorithm, almost all of them are established upon population division with periodical communication. In several cases, the performances of different topologies and different communication mechanisms are varied. Thus in acceleration of intelligent optimization algorithm, the selection and design of topology and communication mechanism are two crucial parts and can also be configured flexibly.

That is to say, the implementation of different topology and communication mechanism can be encapsulated into modules according to different hardware architectures. These modules are independent with the operators applied in different sub-populations, thus can be reused like operators.

According to such idea, in this chapter, we firstly introduce the parallel implementation ways of intelligent optimization algorithm on different hardware architectures. Then we elaborate the typical parallel topologies based on general population division. After that, two configurable parallel ways are presented in different hardware both with module based configuration idea.

5.1 Introduction

As parallel technology continues to evolve, peta-flops parallel computers, large-scaled distributed clusters are emerging in several areas. From the perspective of computing hardware, the pure computing capabilities are largely improved.

© Springer International Publishing Switzerland 2015

F. Tao et al., *Configurable Intelligent Optimization Algorithm*,
Springer Series in Advanced Manufacturing,
DOI: 10.1007/978-3-319-08840-2_5

However, the development is gradually out of Moore's law. That means, the computing speed is no longer grown with the increased computing cores. No matter in manufacturing or industrial engineering, high performance hardware not only did not bring about enough acceleration, but also induced several problems, such as load imbalance, communication blocking, etc., with large energy wasting. Therefore, the performance of parallel technology depends not only on its hardware, but also on the computing and communication assignments and the algorithm design.

In manufacturing, facing with mass productive resources and large-scaled tasks, the optimal problems such as resource scheduling, workflow arrangement and part design in diverse networked manufacturing modes are becoming more and more complex. Its evaluation indexes are generally non-linear functions, and the production steps are increased. For accelerating the whole process, the attention is gradually switching from accuracy improvement to parallelization digging. In such heterogeneous manufacturing system, the parallelization of optimization algorithms is the most important part. That is because both coarse grained and fine grained manufacturing tasks are directly scheduled and parallelized in distributed resources through different kinds of algorithms. The efficiency of task execution is decided by optimization algorithms. If we have an efficient optimization algorithm with high decision accuracy and high speed, then the whole system can be effectively accelerated.

Hence, the parallelization of intelligent optimization algorithms for diverse manufacturing optimization problems is of the essence. Back to its basic operators, according to the theory of "no free lunch" pointed out by [1], any performance improvement in algorithm needs some sacrifice from other sides. For large scaled problems, high quality solutions are obtained either by increasing iteration number and population number, or by adding other improved and hybrid operators. All of these modifications increase the time consumption in decision. Population-based parallelization can largely reduce the time consumed by the operators with high computing complexity. Its mainframe can be shown in Fig. 5.1. In such framework, more operators can be applied with lesser individuals in each sub-population. But in each sub-population, if without communication, the independent iteration will be much less efficient. That is to say, the accuracy preservation is realized by periodical individual exchange among sub-populations.

Based on the framework, parallel intelligent optimization algorithm can be designed and implemented in any multi-processor hardware with uniform communication topology and individual exchange mechanism. The whole design architecture can be represented in Fig. 5.2. With the characteristic of natural concurrency of intelligent algorithms, since the early 90th, a series of parallel intelligent optimization algorithms are proposed. Most of these algorithms are under three typical modes [2]: *master-slave mode*, *coarse-grained mode* and *fine-grained mode*.

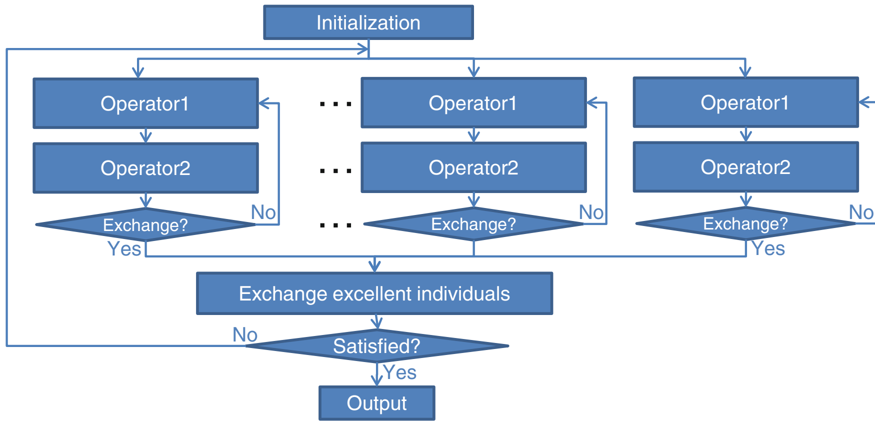

Fig. 5.1 The general framework of parallel intelligent optimization algorithm

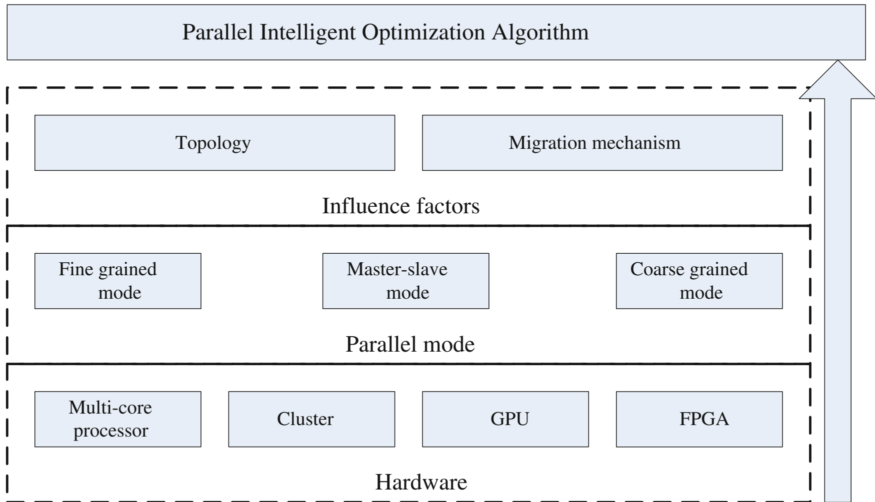

Fig. 5.2 The design architecture of parallel intelligent optimization algorithm

(1) *Fine-grained mode* [3].
In fine grained mode, each sub-population contains only one or two individuals, and operators are executed between different parallel nodes. There are no periodic exchanges but huge communications. In this mode, neighbor structure which decide the scope of learning, cross and exchange between individuals are the main consideration. It defines the information propagation path in the whole population. Generally, for small sized population, large scope structure is suitable, while for large sized population, box structure with four to eight groups are more adaptable. Shapiro et al. [4] have taken

experiments and discussions based on different neighbor structure and concluded that neighbor structure with four groups is quite adaptable.

However, fine-grained mode can only be implemented in shared memory architecture because of its huge communication load during iteration. Nowadays, it is less used.

(2) *Master-slave mode* [5].

Master-slave mode refers to use one master node to manage other slave nodes with sub-populations. Operators are executed in slave nodes and individual exchange is realized through individual reduce and broadcast by master node. Generally, in each period, slave nodes send their best individuals to master node. Master node then screen the global best one and send it again back to slave nodes as a member for next evolution. Communication mainly happens in the collection and broadcasting process and is much less than which in fine-grained mode.

This mode is still widely used for different problems, it is easy to implement and diverse population number and operators can be executed in different slave node. Uneven loads among master and slaves is the main shortcoming. In each period, the simultaneously individual sending from slave nodes can lead to data surging, and all of the slave nodes have to wait for master to calculating the best one and broadcasting it.

(3) *Coarse-grained mode* [6].

It's the most adaptable parallelization mode. Sub-populations are evenly divided and evolve independently and exchange in specific frequency. Without supervision of master node, sub-populations exchange excellent individuals in a specific topology. Communication and computation in each period are more even than master-slave mode.

In this mode, communication topology and individual migration mechanism are two main influence factors for the performance. Communication topology represents the information propagation path of each sub-group. There are already many typical topologies are presented, such as ring) [7], grid [8] and full connection and so on. In the case of sparse connection, the information transforming speed is low, and the whole population has high diversity and low collaboration. On the contrary, with dense connection, the individuals can be largely shared with low diversity and high collaboration. Premature is easily caused in such case. Therefore, the selection of topology is vital.

Besides, migration mechanism can also influence the optimal searching process. It includes the design of individual number to be shared, the period and the replacing scheme. The individual number to be shared refers to the number of individuals sending by each sub-population. The period means the generation number between two exchanges. The replacing scheme decides which local individuals are to be replaced by the newly introduced ones from other sub-groups. This three migration factors, together with topology determine the whole parallel searching performance. Matsumura et al. [9] had compared different topologies in some cases, but different migration mechanisms are still to be discussed.

Clearly, different topologies and migration mechanisms can be reused in different problems. With the idea of dynamic configuration, they can be dynamically configured in different generation or different node with different hardware. Therefore, in the sections below, we will follow Fig. 5.2 and briefly talk about different implementation ways of intelligent optimization algorithm in different hardware, and then give some typical parallel topologies commonly used in industrial optimizations. Based on that, the configuration design of parallel intelligent optimization algorithm is elaborated in this chapter.

5.2 Parallel Implementation Ways for Intelligent Optimization Algorithm

5.2.1 Parallel Implementation Based on Multi-core Processor

Multi-core processor is a processor that integrates two or more complete cores. To traditional single-core processor, the way to increase the processing speed is to improve its work frequency. However, the frequency improvement means the improvement of manufacturing process, which is not unlimited. Quantum effect largely restricts the work frequency and the size of transistor. Thus, the method to boost CPU performance by increasing processor frequency encounters an unprecedented predicament.

The emergence of multi-core processors brings new hope to the improvement of processing speed. Large companies such as INTEL and AMD in turn modify the architectures of CPU, integrate multiple cores in one chip and hence launch dual-core, three-core and four-core CPU products. Figure 5.3 shows the architecture of a multi-core processor.

Currently, based on multi-core processor, many researches [10] focus on the design of parallel programs and parallel technology (e.g. OpenMP). Among the many parallel optimization algorithms based on multi-core processor, some are implemented with the help of OpenMP techniques. In fact, as long as the programmers encode the optimization algorithms in parallel programs and run them on parallel multi-core processor computers, these parallel algorithms can be well implemented. Accordingly, Mahinthakumar and Saied [11] successfully completed the parallel Genetic Algorithm (GA) on multi-core processor while Wang et al. [12] made the parallel Particle Swarm Optimization Algorithm possible and took advantage of it to solving facility location problems. Rajendran and Ziegler [13] well introduced parallel Ant Colony Algorithm based on multi-core processor and solved permutation flowshop scheduling problem.

To sum up, many research and studies have already been conducted on the topic of multi-core processor parallel optimization algorithms, and their successful research achievements involve Genetic Algorithm, Particle Swarm Algorithm and Ant Colony Algorithm, etc.

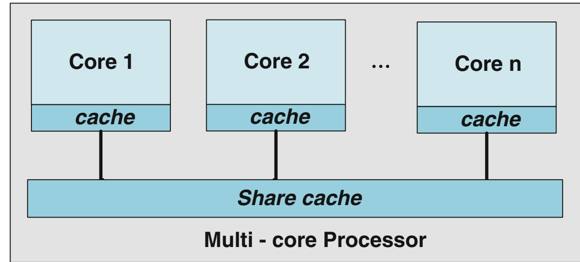

Fig. 5.3 Multi-core Processor

5.2.2 Parallel Implementation Based on Computer Cluster

Computer cluster is a special computer system with a set of loosely integrated computer software and hardware, which closely collaborates to complete the computational works efficiently. To some extent, a computer cluster can be regarded as a single host computer. Moreover, a single computer in the cluster system is often referred to as a node, usually connected via a LAN, but there are other possible connections. Computer cluster is usually used to improve the computing speed and reliability of single computer.

In a computer cluster, the multiple processors usually work in parallel, and every processor has more than one computational core. Therefore, the level of parallelism of a computer cluster is far higher than that of a multi-core processor. Usually, the design of parallel algorithms on computer clusters depends on MPI and OpenMP programming [14, 15].

In recent years, people have carried on some research about the design of parallel optimization algorithms based on computer clusters. E.g., Kalivarapu [16] thoroughly analyzed and discussed the parallel implementation method of Particle Swarm Algorithm, including the implementation on computer clusters. Also, Borovska [17] and Sena et al. [18] programmed optimization Ant Colony Algorithm on computer clusters and tested it through solving the problem of TSP. However, other types of cluster optimization algorithms are relatively less.

In short, through designing parallel optimization algorithms on computer cluster, we can acquire higher level of parallelism than on multi-core processor. Currently, the relatively mature parallel algorithms are computer cluster-based Particle Swarm Algorithm and Ant Colony Algorithm.

5.2.3 Parallel Implementation Based on GPU

Computer graphics processor (Graphics Processing Unit, GPU) is defined as 'a single-chip processor, integrated geometric transformation, illumination, triangular configuration, clipping and drawing engine and other functions, and having

per second at least 10 million polygons handling capacity.' GPU greatly enhanced the processing speed of the computer graphics and the graphics quality, and meanwhile promoted the rapid development of computer graphics applications. Unlike the serial design pattern of the central processor (Central Processing Unit, CPU), GPU is initially designed for the graphics processing, thus, has a natural parallel character. However, the parallelizable instructions in computation are less, and increasing instruction-level parallelism through superscalar, deep water, long instruction word cannot achieve good results.

Because the graphics processor is equipped with parallel hardware structure, thus the calculation performed in the graphics processor has a natural parallelism. These years many research about the design of GPU-based optimization algorithms emerged. The research in [19, 20] studied how to implement optimization algorithms on GPU, while Kalivarapu [16] not only achieved the implementation of Particle Swarm Algorithm on computer clusters, but also on GPU. In addition, relied on GPU, Zhu and Curry [21] gave a detailed study of Ant Colony Algorithm and its parallel application, and Chitty [22], Li et al. [23] carried on a specifically concrete work on the implementation of Genetic Algorithm. There are many other literatures focused on this field of study, which have done a lot of concrete works.

In summary, nowadays, parallel GPU-based optimization algorithm design has been extensively studied and has accomplished a variety of intelligent optimization algorithms and comparatively good results.

5.2.4 Parallel Implementation Based on FPGA

Field-programmable gate array (FPGA) is a further developed product on the basis of many programmable devices such as the PAL, GAL, and CPLD. It is a semi-custom circuits,which is different from the Application Specific Integrated Circuit (ASIC).

In general, both multi-core processor and graphics processor have fixed hardware circuit structures, on which the design of algorithms fails to have a high level of parallelism. Moreover, although general-purpose microprocessors are flexible to design and easy to upgrade, their processing speed and efficiency are relatively low. On the other hand, ASIC can complete the computation tasks by a specific operation and processing unit, thus the execution of instructions is in parallel. To specific integrated circuits, the processing speed and efficiency are higher, but the development cycle is longer and the design flexibility is less. Therefore, in some occasions with higher requirements of real-time performance and flexibility, general-purpose microprocessor or ASIC is not able to solve the problem very well. FPGA, with a natural parallel hardware structure, is not only flexible to design and easy to upgrade as general-purpose microprocessor, but also faster and more efficient as ASIC. Therefore, it provides a new way for the parallel design of optimization algorithms.

So far, FPGA-based optimization algorithm parallel design is not that much, and existing research mainly concentrate on the FPGA implementation of Genetic Algorithm [24, 25] and Ant Colony Algorithm [26]. The studies on other FPGA-based optimization algorithms, such as Particle Swarm Algorithm, are far less and need to be further investigated.

Currently, people have carried out series of research about the design of parallel optimization algorithms, including Genetic Algorithm, Particle Swarm Algorithm and Ant Colony Algorithm. The common method of design is to program with the use of OpenMP. Through programming parallel optimization algorithms on computer clusters, we can obtain a higher level of parallelism than multi-core processor-based design. What's more, after the successful implementation of parallel Genetic algorithm and Particle Swarm Algorithm on computer clusters, they reap a quite extensive application, and the general pattern is to design programs by OpenMP and MPI. Until now, the most widely used parallel technology is the parallel program design based on GPU. Many intelligent optimization algorithms have been well implemented on GPU, and the results seem bright and promising. This method mainly takes advantages of the special hardware structure of GPU. Finally, studies about the algorithm design based on FPGA are chiefly focused on Genetic Algorithm. As to others types of algorithms, there is no common parallel method.

5.3 Implementation of Typical Parallel Topologies for Intelligent Optimization Algorithm

As introduced before, parallel topology represents the connection way among sub-populations. It controls the transform ways and speeds of excellent individuals, so as to make the parallel searching process exerting different influences in varied cases. During existing methods, master-slave topology and ring topology, mesh topology and full-mesh topology in coarse grained mode are the most typical ones. In this section, based on MPI architecture, we will introduce them and give brief MPI implementation of them respectively, from sparse connection to dense connection. Each of the implementation can be encapsulated as modules and reused with configuration methods.

5.3.1 Master-Slave Topology

As shown in Fig. 5.4a, the operation in slave node contains intelligent optimization operators, general evolutionary update steps and individual sending actions,

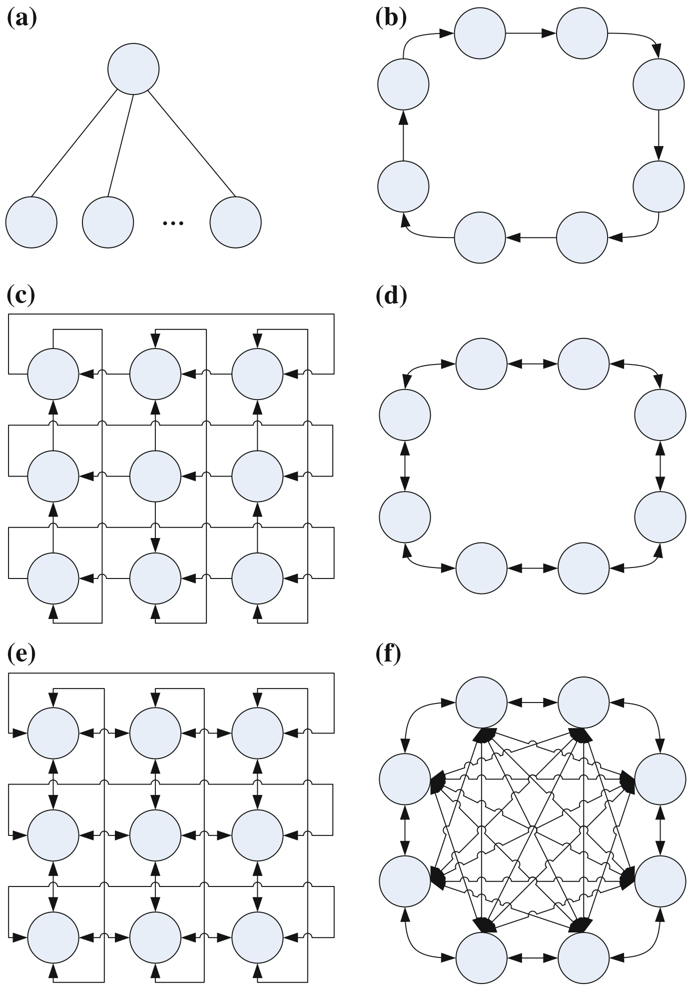

Fig. 5.4 Typical parallel topologies for intelligent optimization algorithm. **a** Master Slave; **b** Single Ring; **c** Single Mesh; **d** Double Ring; **e** Double Mesh; **f** Full Mesh.

while the operation in master node contains only receiving the individuals, calculating the best ones and broadcasting them to slave nodes.

Specifically, the implementation pseudo-code can be represented as follows.

```
For (each sub-population I)
    Initialize subpopulation;
    generation = 0;
    While (generation <= MAX_generation or convergence criterion satisfied)
        generation ++;
        MPI_Gather(best_individual, 1, gene_struct, root_population, 1, gene_struct,
        ROOT, MPI_COMM_WORLD);
        If (processor_id == ROOT)
            choose n individuals from root_population to obest[n];
        End if
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Bcast(obest, n, gene_struct, ROOT, MPI_COMM_WORLD);
        Insert obest[n] to each sub-population;
        If (processor_id != ROOT)
            Apply algorithm's operators;
            Evaluate solutions in the sub-population;
        End if
    End while
End for
```

In the implementation, *best_individual* represents the best individual array in each sub-population to be sent in each period. *gene_struct* represent the class type of each individual, it is generated by *MPI_Type_struct* and contains gene-bits, individual states and its fitness values. *root_population* represent the received individuals in master node (i.e. root node), so that the size of this array are decided by the number of slave nodes and the size of *best_individual*. **obest**[n] then represents the screened best individuals to be broadcasted in master node. If $n = 1$, then only one individual will be selected and broadcasted to other slave nodes. So the communication load is in part decided by n.

5.3.2 Ring Topology

Ring topology consists of two kinds, single-ring and double-ring topology, as shown in Fig. 5.4b and c. Among them, single-ring topology can also be named as the least communication topology. Ring topology is easy to implement and occupies less bandwidth during communication. Information in this topology are spread slowly. In MPI programming, only point to point communication mode can

efficiently implement it. Here takes non-blocking communication as example, the specific pseudo-codes of single-ring and double-ring topology can be shown as follows.

(a) Single-Ring Topology

```
Forward propagation
For (each sub-population I)
    Initialize subpopulation;
    generation = 0;
    While (generation <= MAX_generation or convergence criterion satisfied)
        generation ++;
        If (generation % MT == 0)
            MPI_Irecv(obest, scnt, gene_struct, processor_id+1, 123,
            MPI_COMM_WORLD, &req);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id-1, 123,
            MPI_COMM_WORLD, &req2);
            Insert obest to each sub-population;
        End if
        Apply algorithm's operators;
        Evaluate solutions in the sub-population;
    End while
End for

Back propagation
For (each sub-population I)
    Initialize subpopulation;
    generation = 0;
    While (generation <= MAX_generation or convergence criterion satisfied)
        generation ++;
        If (generation % MT == 0)
            MPI_Irecv(obest, scnt, gene_struct, processor_id-1, 123,
            MPI_COMM_WORLD, &req);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id-1, 123,
            MPI_COMM_WORLD, &req2);
            Insert obest to each sub-population;
        End if
        Apply algorithm's operators;
        Evaluate solutions in the sub-population;
    End while
End for
```

(b) Double-Ring Topology

```
For each sub-population I
    Initialize subpopulation;
    generation = 0;
    While (generation <= MAX_generation or convergence criterion satisfied)
        generation ++;
        If (generation % MT == 0)
            MPI_Irecv(obest_1, scnt, gene_struct, processor_id-1, 123,
            MPI_COMM_WORLD, &req1_1);
            MPI_Irecv(obest_2, scnt, gene_struct, processor_id+1, 321,
            MPI_COMM_WORLD, &req2_1);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id+1, 123,
            MPI_COMM_WORLD, &req1_2);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id-1, 321,
            MPI_COMM_WORLD, &req2_2);
             Insert obest_1 and obest_2 to each subpopulation;
        End if
        Apply algorithm's operators;
        Evaluate solutions in the subpopulation;
    End while
End for
```

In the above codes, *obest*, *obest_1*, *obest_2* represents temporary arrays for receiving neighbor individuals, *scnt* represents the number of individuals to be migrated and *req1_1*, *req1_2*, *req2_1* and *req2_2* are *MPI_Request* parameters. It can be seen that forward propagation and back propagation ring communication can be separated as two modules and implemented as the above pseudo-code (a). Then (b) is the fusion of the two single-track communications as double-ring topology.

5.3.3 Mesh Topology

Mesh topology is similar with ring topology. It contains both left-right and up-down neighbor communication, while ring topology contains only left-right neighbor communication. If the process node is less than 9, the communication will be uneven and cause large exchange load. Also, mesh topology can be implemented as single-side topology and double-side mesh topology. The key MPI implementation can be shown as follows.

(a) Single-side mesh topology

```
For (each sub-population I)
    Initialize sub-population
    generation = 0;
    While (generation <= MAX_generation or convergence criterion satisfied)
        generation ++;
        If (generation % MT == 0)
            //left-right
            MPI_Irecv(obest_1, scnt, gene_struct, processor_id-1, 123,
            MPI_COMM_WORLD, &req1_1);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id+1, 123,
            MPI_COMM_WORLD, &req1_2);
            //up-down
            MPI_Irecv(obest_2, scnt, gene_struct, processor_id-m, 321,
            MPI_COMM_WORLD, &req1_1);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id+m, 321,
            MPI_COMM_WORLD, &req1_2);
            Insert obest_1 and obest_2 to each subpopulation;
        End if
        Apply algorithm's operators;
        Evaluate solutions in the subpopulation;
    End while
End for
```

(b) Double-side mesh topology

```
For (each sub-population I)
    Initialize subpopulation;
    generation = 0;
    While (generation <= MAX_generation or convergence criterion satisfied)
        generation ++;
        If (generation % MT == 0)
            //left-right
            MPI_Irecv(obest_1, scnt, gene_struct, processor_id-1, 123,
            MPI_COMM_WORLD, &req1_1);
            MPI_Irecv(obest_2, scnt, gene_struct, processor_id+1, 321,
            MPI_COMM_WORLD, &req2_1);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id+1, 123,
            MPI_COMM_WORLD, &req1_2);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id-1, 321,
            MPI_COMM_WORLD, &req2_2);
            //up-down
            MPI_Irecv(obest_3, scnt, gene_struct, processor_id-m, 123,
            MPI_COMM_WORLD, &req1_1);
            MPI_Irecv(obest_4, scnt, gene_struct, processor_id+m, 321,
            MPI_COMM_WORLD, &req2_1);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id+m, 123,
            MPI_COMM_WORLD, &req1_2);
            MPI_Isend(best_individual, scnt, gene_struct, processor_id-m, 321,
            MPI_COMM_WORLD, &req2_2);
            Insert obest_1, obest_2, obest_3, obest_4 to each sub-population;
        End if
        Apply algorithm's operators;
        Evaluate solutions in the sub-population;
    End while
End for
```

In such topology, the serial numbers of the neighbors at the up-down side can be calculated as the integral upper bound of the square root of the whole processor number.

$$m = \sqrt{processor_number} \tag{5.1}$$

5.3.4 Full Mesh Topology

In this topology, each sub-population broadcast its best individuals to be migrated. After that, each sub-population receives multiple individuals and accepts part or all of them to replace some bad local ones. It is the most communication topology and have high communication load. Compared with master-slave topology, it is more likely to cause data surging. In large-scaled distributed parallel architecture, it is not suitable. In the MPI programming, full mesh topology is easy to implement with MPI_Allgather. The pseudo-code can be represented as follows.

```
For (each sub-population I)
    Initialize subpopulation;
    generation = 0;
    While (generation <= MAX_generation OR convergence criterion satisfied)
        generation ++;
        If (generation % MT == 0)
            MPI_Allgather(best_individual, scnt, gene_struct, obest, scnt,
            MPI_COMM_WORLD);
            Insert obest[n] to each subpopulation;
        End if
        Apply algorithm's operators;
        Evaluate solutions in the sub-population;
    End while
End for
```

In the implementation, the size of **obest** n is decided by the migration number and the whole processor number.

$$n = scnt \times processor_number \tag{5.2}$$

5.3.5 Random Topology

During the above topologies, no matter with dense or sparse connections, have advantages and disadvantages. For balance the two kinds, Defersha and Chen [27, 28] presented a new random topology for parallelization of intelligent optimization algorithm in solving manufacturing optimization. In such topology, the exchanges among sub-populations are decided by a binary matrix. It is generated by a single node and then broadcast to others to make sure the correct exchange. The dimension of the matrix is equal to the number of processor nodes. Let **A** represents the matrix. If the element $a_{ij} = 1$, then node i will send some excellent individuals to node j. The value of a_{ij} can be calculated as follow [27].

$$a_{ij} = \begin{cases} 1, & rand() < \rho \text{ and } i \neq j \\ 0, & otherwise \end{cases} \quad (5.3)$$

where 'rand()' represents random generalized number and $\rho \in [0, 1]$ refers to the control parameter of communication density. If ρ is low, the communication is becoming sparse, vice versa. At the same time, Defersha also concludes that $\rho = 0.5$ is generally suitable for individual exchanges.

In MPI implementation, only point to point mode can be applied. With such method, we found that the matrix generation and broadcasting still takes extra time consuming which can not be ignored. Therefore, its performance in different cases is still to be discussed. The pseudo-code of random topology can be shown as follows.

```
For (each sub-population I)
    Initialize sub-population;
    generation = 0;
    obest[n] = 0;
    While (generation <= MAX_generation or convergence criterion satisfied)
        generation ++;
        If (generation % MT == 0)
            Generate random_matrix[processor_number][processor_number]; //1 or 0
            For (processor_id_i = 1 to processor_number)
                For (processor_id_j = 1 to processor_number)
                    If (i != j && random_matrix[i][j] == 1)
                        MPI_Irecv(obest[i], scnt, gene_struct, j, 123,
                        MPI_COMM_WORLD, &req);
                        MPI_Isend(best_individual, scnt, gene_struct, i, 123,
                        MPI_COMM_WORLD, &req2);
                    End if
                End for
            End for
            For (k = 1 to n)
                If (obest[k] != 0)
                    Insert obest[k] to each sub-population;
                End if
            End for
        End if
        Apply algorithm's operators;
        Evaluate solutions in the sub-population;
    End while
End for
```

In the above code, **random_matrix** represents the random matrix **A**. In each period, a new **random_matrix** is generated by root node and broadcast to others. **obest** stores the individuals received from other nodes decided by the random matrix. If **obest**$[i] \neq 0$, then insert it into the local population and replace a bad one.

Besides, more information about MPI programming can be found in [29].

5.4 New Configuration in Parallel Intelligent Optimization Algorithm

Generally speaking, in parallel searching, communication is independent with operators. As with configuration in algorithm improvement and hybridation, parallelized algorithms can also be dynamically configured in different hardware architecture. But with different communication prototypes, parallel configurations in different hardware are totally different. Therefore, this section mainly focuses on the parallelization and algorithm configuration on general multi-processors and FPGA respectively.

In large sized multi-processors, take general cluster with MPI as an example, configuration can be classified into two kinds, (1) topology configuration, and (2) operation configuration. Topology configuration refers to invoke different communication topology in different period, while operation configuration here consists of algorithm-based, operator-based and parameter-based configuration introduced in Chaps. 3 and 4. In small sized hardware, i.e. FPGA, topology configuration cannot be implemented in most time. Parallelization based on FPGA is totally different with which in other hardware. Without population division, it parallelizes operators, encapsulated them as modules and tries to flexibly connect different parts together. That is to say, the inner part of the module cannot be changed but only reloaded. Therefore, in FPGA, we could only connect different kinds of algorithm modules or operator modules in divided generations to realize flexible configuration, here we call it module-based configuration. The configuration types on the above two hardware architecture can be summarized as shown in Fig. 5.5.

Regardless of which kinds of hardware we are based, the general design process of parallel intelligent optimization algorithm can be shown in Fig. 5.6. The steps contain (1) algorithm design, (2) scale of sub-populations, (3) topology selection, (4) migration mechanism decision, and (5) algorithm implementation. If we want to design and implement a parallel intelligent optimization algorithm, we need first to design a serial algorithm with improved or hybrid operators which can solve the specific problem with high accuracy. Next, according to the existing environment, the scale of sub-populations needs to be specified before topology design, for the reason that the exchange performance of topology depends on the number of subgroups. Based on particular topology and algorithm, we could then set the migration mechanism, i.e. how many individuals to be migrated and which of the local ones to be replaced. Based on these decisive factors, the parallel algorithm can finally be implemented. Likely, general design process is quite cumbersome.

If we encapsulate these topologies with various mechanisms, combined with the above-mentioned serial algorithm parts, the design of parallel intelligent optimization algorithm can be easier. Figure 5.7 shows the new configuration process for it. In this mode, the topology module can be invoked as a function which only implementing data transform and individual replacement. Take ring topology as an example, the corresponding module mainly contains the following part. That is to

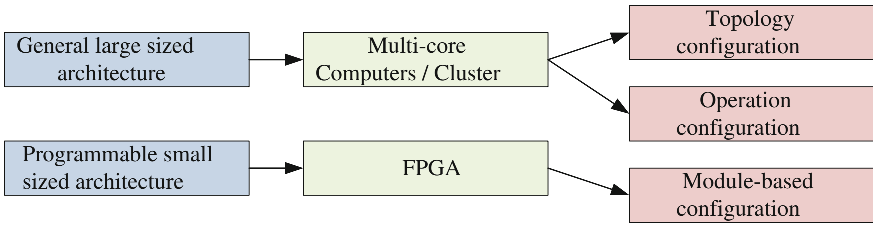

Fig. 5.5 parallel configuration types in different hardware

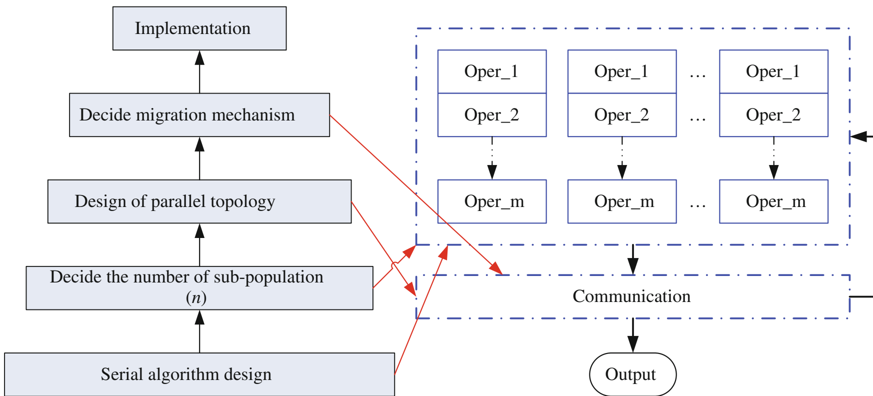

Fig. 5.6 The general design process of parallel intelligent optimization algorithm

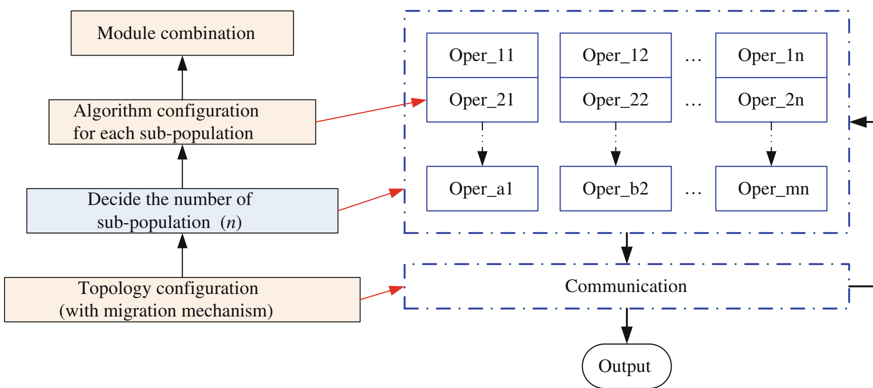

Fig. 5.7 Configuration process of parallel intelligent optimization algorithm

say, we just put data sending, receiving and the individual replacement sentences into the topology module. The parameters of it include sending array, receiving array and number of individuals to be migrated. In each sub-population, basic operators are invoked in every generation, while topology module is called at a certain period.

```
//Topology module
Basic_ring_topology(best_individual, obest, scnt)
{
    MPI_Irecv(obest, scnt, gene_struct, processor_id+1, 123, MPI_COMM_WORLD,
    &req);
    MPI_Isend(best_individual, scnt, gene_struct, processor_id-1, 123,
    MPI_COMM_WORLD, &req2);
    Insert obest to each sub-population;
}

//Module Invoking
If (generation % MT == 0)
    Basic_ring_topology(best_individual, obest, scnt);
End if
```

It is clear that in traditional design ways, the selection of topology is dependent with both the design of serial operators in each sub-group and the hardware environment. If any of them performs not well, then we need to redesign it again. Different with the traditional process, we could select the topology firstly only according to the specific hardware environment. Then in each sub-group, different operators with uniform population input and output can be tested and applied respectively. In such case, topology is independent with operators. The only thing we need to do is module combination. With different operators, the sub-group who performs better could help other bad performed ones to break out from local optimal through individual exchange. Then better optimal searching capability can be preserved with low time consumption for wider complex problems.

In the following sections, we will elaborate different configuration types both in multi-processor computers and FPGA.

5.4.1 Topology Configuration in Parallelization Based on MPI

As mentioned previously, topology configuration means to apply multiple topologies in a parallel algorithm. It also contains two styles, (1) single-domain topology configuration and (2) multi-domain topology configuration.

Firstly, single-domain topology configuration is to allocate multiple topologies into different generations with one operation domain, as shown in Fig. 5.8. In such scheme, although operators in different sub-population are different, they will not change along with iteration. All sub-populations belong to one domain. After dividing the generations into several parts, we could change topology module to

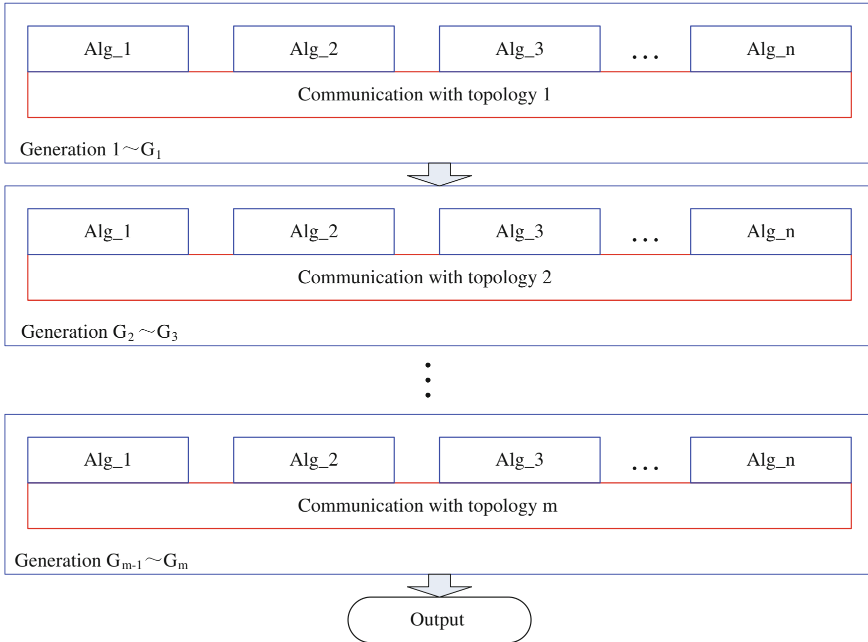

Fig. 5.8 Single-domain topology configuration in parallel intelligent optimization algorithm

make sub-populations communicating with different ones, as well as in random topology. Moreover, extra communication is needless. The information propagation can be easily controlled according to the whole population state. If the population has high diversity, then the topology with dense connection can be applied. On the contrary, if the population has low diversity, then the topology with sparse connection is more suitable. Operators in different sub-groups are responsible for digging solution with lesser members and less time, topology then tries to balance the searching state and preserve high quality. Following the general searching rules, topology with sparse connection should be adopted at the beginning for dynamic exploration. Then topology with dense connection can be used in the end for population convergence accordingly.

Multi-domain topology configuration, as shown in Fig. 5.9, refers to divide sub-populations into several domains and apply different topology to each domain. In each period, sub-population with different algorithms only communicates with the ones in the same local area through corresponding connection topology. Groups in different domains will not do exchange any more. For wider information exchange, we could also divide generations into several parts and regroup the sub-populations to different topology domains. It is clear that the information propagation is narrower and slower than which in single-domain scheme. It can keep better diversity state and is more suitable for heterogeneous clusters, in which we could allocate sparse topology to the nodes with low communication bandwidth and dense

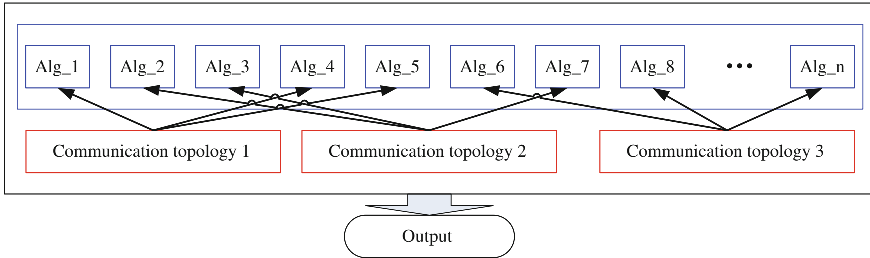

Fig. 5.9 Multi-domain topology configuration in parallel intelligent optimization algorithm

topology to the nodes with high communication speed. In this scheme, one of the most important steps is regrouping. We could generate a group of random numbers which refer to the topology numbers in a root node and then broadcast them for allocating sub-populations uniformly. So, the main drawback of this method turns out to be the restructuring step which may take many extra times so as to slow down the whole process. For simplifying the process, people can also configure the same algorithm for each sub-population with only topology hybridization.

It can be seen that topology configuration is suitable especially for large scale parallelization with a large number of computing nodes. When sub-populations are less, multi-topologies are then becoming useless. For example, in a parallel intelligent optimization algorithm, if there are only four nodes (processors), then mesh topology has no much difference with full-mesh topology. The control of changing topology in single-domain scheme and the regrouping step in multi-domain scheme are both time consuming with low-efficiency.

5.4.2 *Operation Configuration in Parallelization Based on MPI*

Correspondingly, in small scale parallelization, operation configuration is more suitable. As mentioned before, operation configuration means to do three-layer configurations in each sub-population without topology changing. It is the same with the parallel configuration in module-based improvement and hybridization mentioned in Chap. 4. The only difference is that different operators are simultaneously executed in multiple processors. It is much easier to design than the above topology configuration ways. With fixed individual exchanging scheme, the evolutionary process is more stable.

With limited computing resources, operation configuration in parallel intelligent optimization algorithm can be very adaptable especially for dynamic complex problems, such as parameter adjustment in part design and dynamic job-shop scheduling. For instance, for a continuous parameter setting problem in part design, we could divide the whole population into four groups and applied

continuous genetic algorithm (GA), particle swarm optimization (PSO), differential evolution (DE) and cuckoo search (CS) in sub-populations respectively. According to 'no free lunch theory', these four algorithms are suitable for different cases. In simultaneously execution with exchanges, the most suitable one in a specific case will offer its current best solution to others and guide them to better positions. It the constraints are changed or a new part is needed to be designed, another algorithm might be a new leader to preserve the whole searching quality. We need not to design new algorithms, only one scheme with several configured algorithms can applied to different kinds of problems with good quality. For achieving such performance, we should note that the algorithms configured in different sub-populations need to be very different with diverse emphasis on exploration and exploitation, as well as in the design of serial intelligent optimization algorithm, for balance searching.

Moreover, it should be note that even in large-scaled parallelization design, we need not to configure topology and operators both. That is because too much dynamics will totally break the searching paces and obtain a chaos situation as a result. Therefore, although configuration is easy to implement, the collaboration between operators and topologies need to be considered seriously.

5.4.3 Module Configuration in Parallelization Based on FPGA

In this section, we presented a new parallelization way of intelligent optimization algorithm on FPGA. With several blank logical resources in FPGA, we could implement the original operators as multiple arithmetic units. For connecting them, some state machine is also designed for connecting these units to form a specific intelligent optimization algorithm. Based on these design structure, we will implement some typical intelligent optimization algorithm on FPGA and do some configuration design further in the following chapters.

All of the designs are established based on VHDL (Very-high-speed Hardware Description Language). It is used to describe a digital system, including its structure, behaviors, functions and interfaces. The style and grammar of VHDL is much similar to advanced computer programming languages, except that it stands for hardware describing. In VHDL, a digital system is called an entity which can be defined as inner processors and external interfaces separately. After external interfaces are set up, inner processes can be developed in detail. And then, the developed entity can be called as a subsystem in order entities. In other to help readers to learn about the design and validation of intelligent optimization algorithm on FPGA, we will introduce the Virtex-5 family FPGA chips and floating-point format of IEEE 754 in following.

Firstly, the algorithms designed in this book are implemented and validated withVirtex-5 family FPGA chips. Figure 5.10 shows their structure. It can be seen

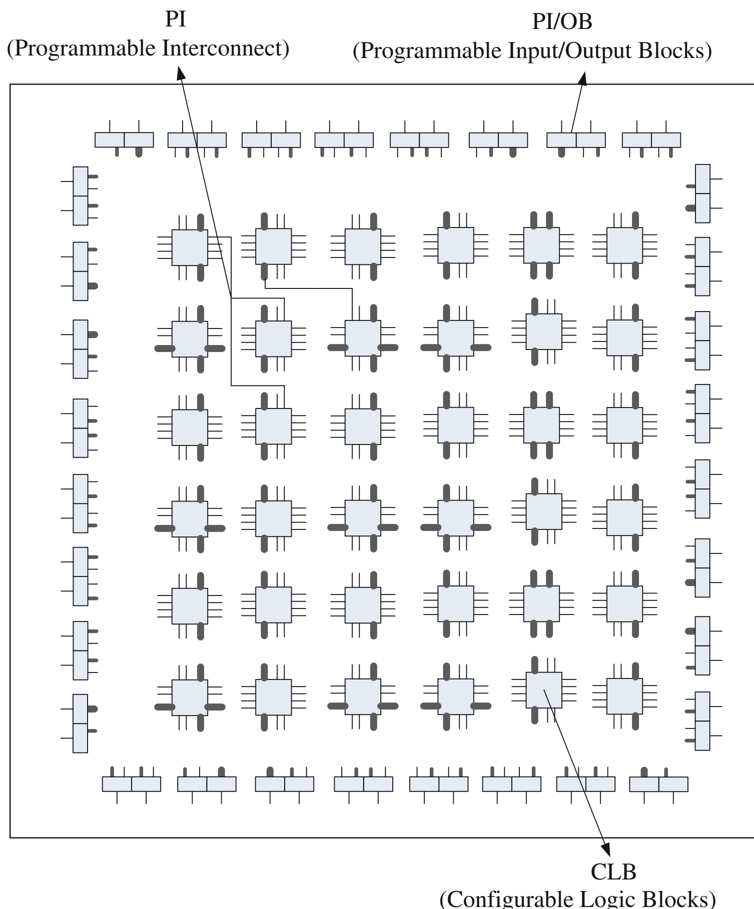

Fig. 5.10 The inner structure of FPGA

that there are mainly CLB (Configurable Logic Blocks), PI (Programmable Interconnection) and PI/OB (Programmable I/O Blocks) inside an FPGA chip. Except these three components, there are also some other abundant resources, such as DSP48E for computing, Block RAM for data storage and CMT (Clock Management Tiles) for clock managing and so on.

A CLB contains several logical resources inside, which are used to implement combinational circuit and sequential circuit. Each CLB in Virtex-5 includes 2 slices, 8 LUT (Look Up Table), 8 triggers, 2 arithmetic and carry chains, 256-bits distributed RAM and 128-bits shifting register. DSP48E Slice module in it can handle 25×18 complement multiplication and can also configured as multiplier, subtracter or accumulator. In the design process of parallel intelligent optimization algorithm, large amount of computing tasks can be assigned to this module to execute.

For different types of chips, the inner resources are different. In Virtex-5 family, there are five platforms, i.e. LX, LXT, SXT, FXT and TXT. LX and LXT are mainly used for high-speed logical design, while SXT is primarily applied for complex digital signal processing. The embedded PowerPC processor of FPGA in FXT is chiefly designed for the development of embedded system. And the FPGA of TXT is especially for customized and complete high-performance system. On account of the abundant logistical resources in FPGA, we mainly considered to use the FPGA of LX/LXT to design parallel intelligent optimization algorithms especially for the complex problems with high requirements on real-time decision efficiency. Here we list the properties of the FPGA chip of Xilinx Virtex-5 LX platform [30].

From Table 5.1 we can see that XC5VLX50T type FPGA includes $120 \times 30 = 3600$ CLBs. In the device, there are 48 DSP48E slice modules which can realize high speed floating point arithmetic together with abundant logistical resources. The block RAM which is 120×18 Kb can store plenty of intermediate data during algorithm execution. And the CMT which contains 6 time management modules is fully enough for counting the optimization time. Therefore, XC5VLX50T can fully satisfy the design requirements of intelligent optimization algorithms.

Secondly, the data format adopted by the research is floating-point data format specified by IEEE 754 standard. The standard divides floating-point data into three types: single float, double and extended. It includes three sections in the memory: sign, exponent and mantissa. For different types of floating-point data, the word lengths of the three sections are different, as shown in Table 5.2.

According to variable symbols shown in Table 5.2, a floating-point data can be calculated by the following equation.

$$x = (-1)^S \times 1.M \times 2^{E-B} \tag{5.4}$$

In the following section, for simplicity, only single-precision floating-point data format is adopted in our design of FPGA-based intelligent optimization algorithm. One must notice that the design is not limited in single float precision.

(1) **Traditional design process of parallel intelligent optimization algorithms**
Multi-core processors, as well as GPU, have their own computing architecture. The processing element has its specific arithmetic unit and controller. With these units, general design process of parallel intelligent optimization algorithm can be abstracted and summarized as follows.

Step 1 **Analysis of algorithm parallelization:** In this step, we need to extract the parts which can be parallel implemented. Unlike the parallelization in coarse-grained hardware architecture, in such a fine grained chip, the cyclic parts in operators of intelligent optimization algorithm in which the data is processed independently can always be parallelized directly.

Table 5.1 Some properties of the FPGA in Virtex-5 LX platform

Device	CLB			DSP48E Slice	Block RAM			CMT
	Array (R × C)	Virtex-5 Slice	Distributed RAM(Kb)		18 Kb	36 Kb	Max (Kb)	
XC5VLX30	80 × 30	4,800	320	32	64	32	1,522	2
XC5VLX50	120 × 30	7,200	480	48	96	48	1,728	6
XC5VLX85	120 × 54	12,960	840	48	192	96	3,456	6
XC5VLX110	160 × 54	17,280	1,120	64	256	128	4,608	6
XC5VLX155	160 × 76	24,320	1,640	128	384	192	6,912	6
XC5VLX220	160 × 108	34,560	2,280	128	384	192	6,912	6
XC5VLX20T	60 × 26	3,120	210	24	52	26	936	1
XC5VLX30T	80 × 30	4,800	320	32	72	36	1,296	2
XC5VLX50T	**120 × 30**	**7,200**	**480**	**48**	**120**	**60**	**2,160**	**6**
XC5VLX85T	120 × 54	12,960	840	48	216	108	3,888	6
XC5VLX110T	160 × 54	17,280	1,120	64	296	148	5,328	6

Table 5.2 IEEE 754 standard floating-point format

Data type	Memory bits			Word length (bits)	Offset (B)
	Sign (S)	Exponent (E)	Mantissa (M)		
Single float	1(Highest)	8(23–30)	23(0–22)	32	127
Double	1(Highest)	11(52–62)	52(0–51)	64	1023
Extended	1(Highest)	15(65–79)	65(0–64)	80	16383

Step 2 **Parallel programming:** It refers to rewrite the algorithms into the corresponding parallel programming language. This step is quite related to specific processor type and the whole execution environment.

Step 3 **Debugging and improvement:** The performance of the same parallel program in different multi-core processors is varied. So the parallel codes need to be modified and improved and generate different version for users to apply.

(2) **New design process of parallel intelligent optimization algorithms on FPGA**

As we introduced before, FPGA is a kind of blank processor. It has no fixed computing architecture, no specific arithmetic unit and controller. Only a group of programmable logistical resources and other assistant resources are provided. However, general algorithm is composed by some basic calculations and process control statements. Therefore, the design of parallel intelligent optimization algorithm in FPGA is different from the above process. Extra design of relative operational unit and state machine for specific algorithm is quite essential. Then, the FPGA-based design of parallel intelligent optimization algorithm can be drawn as the following four steps.

Step 1 **Analysis of algorithm parallelization:** Firstly, all key basic calculation parts need to be listed and the parts which can be parallelized should also be extracted.

Step 2 **Design of arithmetic units:** According to the key calculation parts extracted from step 1, we need to design arithmetic units with different functions. Such arithmetic units can either be float-pointing calculation, or be binary-sequence processor. In this step, the parts which can be parallelized are implemented in the chip with multiple logistical resources. The design of calculation units combined with these resources will execute the corresponding operation in parallel. So the parallel design of arithmetic units can largely decide the whole execution performance of the algorithm.

Step 3 **Design of state machine:** In accordance with the execution process of the algorithm, we could introduce a state machine to connect and control these units for iterative searching. Each state can be seen as an executor of different calculation process. The states can be triggered in parallel.

Step 4 **Debugging and improvement:** No matter the arithmetic units or the state machine designed in FPGA for a specific intelligent optimization algorithm require some modifications in different implementation environments. And for further configuration in solving different problems, several improved versions for both the arithmetic units and the state machine are also required.

5.5 Summary

This chapter systematically introduces almost all kinds of parallel ways of intelligent optimization algorithms. Firstly, the implementation of several kinds of topologies can be applied in many areas for solving complex problem by multi-core computing resources. With these topologies, a lot of parallel intelligent optimization algorithms can also be generated quickly. And through generation division and population division, configuration can also be flexibly implemented in parallel intelligent optimization algorithm. This chapter presented two kinds of configuration ways for fully using existing algorithms and solving wider problems with complex properties. Moreover, the parallelization way of intelligent optimization algorithms based on FPGA is also introduced.

Of cause all of these configurations are not only based on the establishment of a group of typical operators and algorithms but also rely on the implementation of typical topologies in coarse-grained hardware architecture and the design of arithmetic units and state machines in fined-grained FPGA platform. But in diverse parallel platform we must know that not the design of operator but the design of parallel structure and the flexible configuration in different structure worth more in solving large-scaled optimization algorithms. As a group of new design schemes, the parallelization of intelligent optimization algorithm based on the concept of DC-IOA can be applied to reconnect existing operators or algorithms to solve wider problem more conveniently and more easily.

References

1. Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Trans Evol Comput 1(1):67–82
2. Crainic TG, Toulouse M (2010) Parallel meta-heuristics. Gendreau M, Potvin J-Y (eds) Handbook of metaheuristics, vol 146, pp 497–541
3. Collins RJ, Jefferson DR (1991) Selection in massively parallel genetic algorithms. In: The international conference on genetic algorithms
4. Shapiro BA, Wu JC, Bengali D, Potts MJ (2001) The massively parallel genetic algorithm for RNA folding: MIMD implementation and population variation. Oxford University Press, Oxford
5. Sun D, Sung WP, Chen R (2011) Master-slave parallel genetic algorithm based on MapReduce using cloud computing. Appl Mech Mater 121–126:4023–4027

6. Lin SC (1994) Coarse-grain parallel genetic algorithms: categorization and new approach. In: The 6th IEEE symposium on parallel and distributed processing, pp 28–37
7. Beckers MLM, Derks EPPA, Melssen WJ, Buydens LMC (1996) Using genetic algorithms for conformational analysis of biomacromolecules. Comput Chem 20(4):449–457
8. Fukuyama Y, Chiang HD (1996) A parallel genetic algorithm for generation expansion planning. IEEE Trans Power Syst 11(2):955–961
9. Matsumura T, Nakamura M, Okech J, Onaga K (1998) A parallel and distributed genetic algorithm on loosely-coupled multiprocessor system. IEICE Trans Fundam Electron Commun Comput Sci 81(4):540–546
10. Akhter S, Roberts J (2006) Multi-core programming. Intel Press, Hillsboro
11. Mahinthakumar G, Saied F (2002) A hybrid MPI-OpenMP implementation of an implicit finite-element code on parallel architectures. Int J High Perform Comput Appl 16(4):371–393
12. Wang D, Wu CH, Ip A, Wang Q, Yan Y (2008) Parallel multi-population particle swarm optimization algorithm for the uncapacitated facility location problem using openMP. IEEE Congress Evol Comput 1214–1218
13. Rajendran C, Ziegler H (2004) Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs. Eur J Oper Res 155(2):426–438
14. Dolbeau R, Bihan S, Bodin F (2007) HMPP: a hybrid multi-core parallel programming environment. In: Workshop on General purpose processing on graphics processing units (GPGPU)
15. Rabenseifner R, Hager G, Jost G (2009) Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: The 17th Euromicro International conference on parallel, distributed and network-based processing, pp 427–436
16. Kalivarapu VK (2008) Improving solution characteristics of particle swarm optimization through the use of digital pheromones, parallelization, and graphical processing units (GPUs). Iowa State University, Iowa
17. Borovska P (2006) Solving the travelling salesman problem in parallel by genetic algorithm on multicomputer cluster. In: International Conference on computer systems and technologies, pp 1–6
18. Sena GA, Megherbi D, Isern G (2001) Implementation of a parallel genetic algorithm on a cluster of workstations: traveling salesman problem, a case study. Future Gener Comput Syst 17(4):477–488
19. Zhou Y, Tan Y (2009) GPU-based parallel particle swarm optimization. In: IEEE Congress on Evolutionary computation, pp 1493–1500
20. Mussi L, Nashed YSG, Cagnoni S (2011) GPU-based asynchronous particle swarm optimization. In: Proceedings of the 13th ACM annual conference on Genetic and evolutionary computation, pp 1555–1562
21. Zhu W, Curry J (2009) Parallel ant colony for nonlinear function optimization with graphics hardware acceleration. In: The IEEE International conference on systems, man and cybernetics, SMC, pp 1803–1808
22. Chitty DM (2007) A data parallel approach to genetic programming using programmable graphics hardware. In: Proceedings of the 9th ACM annual conference on Genetic and evolutionary computation, pp 1566–1573
23. Li JM, Wang XJ, He RS, Chi ZX (2007) An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In: 2007 NPC workshops IFIP International conference on network and parallel computing, IEEE, pp 855–862
24. Graham P, Nelson B (1996) Genetic algorithms in software and in hardware-a performance analysis of workstation and custom computing machine implementations. In: IEEE symposium on FPGAs for Custom computing machines, pp 216–225
25. Shackleford B, Snider G, Carter RJ, Okushi E, Yasuda M, Seo K, Yasuura H (2001) A high-performance, pipelined, FPGA-based genetic algorithm machine. Genet Program Evolvable Mach 2(1):33–60
26. Juang CF, Lu CM, Lo C, Wang CY (2008) Ant colony optimization algorithm for fuzzy controller design and its FPGA implementation. IEEE Trans Industr Electron 55(3):1453–1462

27. Defersha FM, Chen M (2008) A parallel genetic algorithm for dynamic cell formation in cellular manufacturing systems. Int J Prod Res 46(22):6389–6413
28. Defersha FM, Chen M (2009) A parallel genetic algorithm for a flexible job-shop scheduling problem with sequence dependent setups. Int J Adv Manuf Technol 49(1–4):263–279
29. Pacheco PS (1997) Parallel programming with MPI. Morgan Kaufmann, San Francisco
30. Jose S (2006) http://www.xilinx.com/prs_rls/2006/silicon-vir/0658lxship.htm