Communication issues in HPC

Jiarui XIE
UCA-ISIMA
Clermont-Ferrand, France
jrxhit@qq.com

***Abstract*-In high performance calculating, communications between different threads seem very important. By researching the advantages and disadvantages of different methods in communication will bring us big benefits. In this survey, we will discuss something about the type of communication 、synchronization、aggregation and the communication models.**

Keywords: communication; synchronization; aggregation

I. INTRODUCTION

In high-performance computing, a task is usually assigned to multiple processes to calculate and then aggregate the distribute results to produce the final result. In the process of assigning tasks, calculate, and aggregate results will involve with the communication between different process. The computational efficiency of tasks depends on how we handled with the communication. If it is not handled properly, it will waste time、resources even can not get the correct results. Based on the knowledge and related literature, this paper sorts out the communication knowledge in the field of parallel computing summarizes its basic connotation and analyzes its advantages and disadvantages.

II. SYNCHRONIZATION

A. *classification of synchronization*

We summarized the classification of synchronization, and draw the illustration as follows we will descript some of those：

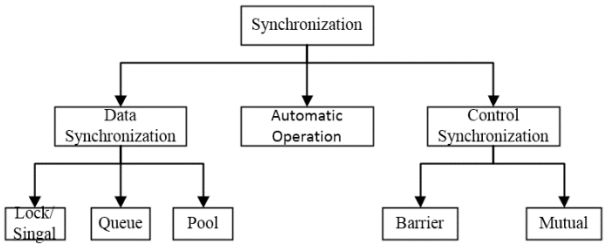

Figure 1. classification of synchronization

B. *Atomicity*

Atomicity usually means that a process needs to perform a series of operations in a single atomic operation. The example program is as follows：

```
Parfor(i:=1; i<n; i++)
{
  Atomic {X=X+1；y=y-1；}
}
```

Figure 2. Example of Atomicity

The most common example of atomicity is a bank transfer. At the same time, atomicity is also an important goal to be achieved by synchronous control.

C. *Synchronization Primitives*

The parallel programming environment of today's multiprocessor systems provides three types of synchronization primitives:

- Lock/Signal: used to achieve mutual exclusion form atomicity
- Critical Region: used to achieve mutual exclusion form atomicity
- Roadblock: used in fence synchronization

1) *Signal lights and Locks*

The common use of locks is to convert critical sections into atomic operations through mutual exclusion.

The semaphore S is a non-negative 0 or 1 Integer that can perform two atomic operations P(S) and V(S):

- If S is not 0, the P(S) operation decrements the S value by 1;
- If S is not 1, the V(S) operation increments the S value by 1;

S is a signal which is true or false, also known as a lock. Correspondingly, for lock S its P(S) and V(S) are often written as lock(S) and unlock(S).

The main advantage of the lock is that it has been supported by most multiprocessors and has been studied quite deeply. Locks are a very flexible mechanism that enables almost any synchronization. However, when the mutex technique is used to implement the atomicity, it has some serious drawbacks, resulting in the following eight problems:

- Non-structural: A lock is not a structured structure. It is easy to use it. If the lock/unlock statement is missing or redundant, the compiler cannot detect it.
- Damages portability: locks are not what users really want, it is just a way to achieve atomicity. Locks impair the portability of the program and make the code difficult to understand;
- State-related: the lock introduces the signal light S and uses the conditional atomic operation lock(S), whether a process can pass through the lock(S), depending on the signal light variable S. In general, state-related data like this is difficult to understand.;
- Sequential execution: For some transaction operations, even if they can be accessed in parallel, they can only execute one at a time because of the use of lock mutex, and this sequential execution is not what the user wants;
- Lock overhead: sequential execution of lock (S) and unlock (S) also have additional overhead, and when *N* processes each perform lock(S) operation, at most one of them can succeed, the rest must be repeated Visit S and try again;
- Priority inversion: When a lock required by a high priority process is preempted by a low priority process, the high priority process cannot advance because it is locked by the lock;
- Escort blocking: When a process that keeps the lock is interrupted due to page faults or timeouts, other processes cannot advance due to waiting for the lock;
- Deadlock: Assume two processes P and Q, want to perform X and Y operations: when process P has held a lock for X and wants to apply for a lock for Y; and process Q has been maintained for Y When a lock is applied and a lock is requested for X, no process releases a lock before it gets the lock, and as a result, no one can get the lock requested.

2) Critical Region

Refers to the critical Region used by the OS. Its syntax is as follows:

```
Critical _ region resource
{
S1; S2;...; Sn;
}
```

Figure 3. Example code of critical_region

Where resource represents a set of shared variables. All critical region sharing the same resource must be mutually exclusive.

3) Roadblocks

Force all processes to wait at the roadblocks, and when all the processes arrive, remove the roadblocks and release all the processes to form a synchronization.

To achieve roadblock synchronization, two rotary locks are generally used: one to record the number of processes arriving at the roadblock; the other is used to block the process until the last process arrives. In the implementation of roadblocks, the specified variables must be continuously detected until the conditions are met.

Example: The implementation of a roadblock, and the "total" is the total number of processes.

```
lock(counterlock); /*lock*/
if(count==0) release=0; /*set the first process as release */
count=count+1; /*count the number of process*/
unclock(counterlock); /*unlock*/
If (count==total){
  Count=0; /*reset the counter*/
  Release=1; /*release the process*/
}
Else{
  spin(release=1); /*wait for processes which have not arrived*/
}
```

Figure 4. Example of roadblock

Lock the counter lock to ensure the atomicity of the incremental operation. The variable count records the number of processes that have arrived. The variable release is used to block the process until the last process reaches the roadblock. The function spin (release=1) causes the process to wait until All processes reach the roadblock.

D. Low-level synchronization primitive

Most multiprocessor hardware provides some atomic instructions that perform a single read-modify-write operation on the primary variables.

There are many kinds of atomic operations, and the most commonly used are three low-level synchronous structures.

1) Test-and-Set

Test-and-Set(S，temp) is an atomic operation instruction that reads the shared variable S into the local variable temp and then sets S to 1。

The following is the implenment of Test-and-Set

```
Definition：Test-and-Set(address，bit-position)
Begin
  Temp:=Memory[address].bit-position；
  Memory[address].bit-position：=1；
  Condition-code:=Temp.bit-position；
End
```

Figure 5. Implement of Test-and-Set

Example：

```
while(S)；
Test-and-Set(S，temp)；
while(temp)Test-and-Set(S，temp)；
/ *Critical Region* /
S=False；  / * unlock(S)* /
```

Figure 6. Example of Test-and-Set

In the above example, the Test-and-Set operation is used to execute lock(S), where the first while loop checks if lock S has been released by another process. Since the shared variable S is written each time Test-and-Set is executed, it may result in frequent memory access.

2) Compare-and-Swap

Compare-and-Swap(S， old， new， flag) It is also an atomic operation instruction that compares the shared variable S with the local variable old: if S is consistent with old, then S=new, and flag=True, to indicate that S is modified; if S is inconsistent with old, then old=S And flag=False, its main purpose is to perform the lock function.

Example:

```
0ld=balance[x]；  / *read the value of share variable* /
do{
  new=old-100  ；  / *modify* /
  Compare-and-Swap(balance[x] ， 0ld ， new，flag); / *write* /
}while(flag==False)；
```

Figure 7. Example of Compare-and-Swap-1

The above operation can be implemented with a lock as follows：

```
lock(S);
balance[x]=balance[x]  -100 ；  / *read-modify-write* /
unlock(S);
```

Figure 8. Example of Compare-and-Swap-2

The above lock function makes the whole process of reading and writing mutually exclusive. The advantage of using Compare-and-Swap is that the length of the critical section is reduced to only one instruction.。

3) Fetch-and-Add

Fetch-and-Add(S, V) is also an atomic operation instruction that returns the shared variable S to the local variable Result and then adds the local value V to S. The syntax is as follows:

```
Fetch-and-Add (S, V)
Result=S；
S=S+V；
Return Result；
```

Figure 9. Example of Fetch-and-Add-1

This instruction is not only simple but also fast. For example, the code segment of the previous example can be implemented with only one Fetch-and-Add instruction:

```
Fetch-and-Add(balance[x]，-100)
```

Figure 10. Example of Fetch-and-Add-2

III. AGGREGATION

The partial results calculated by each of the sub-processes in the parallel program need to be combined with the aggregation operation to obtain a complete result.

The main features of aggregation are:

- It consists of a series of super-steps
- In each super step, each process completes a small granularity calculation
- Communicating a short message

Example 1: Find Partial and (Recursive Folding Reduction Operations)

Suppose there are n processes: P(0), P(1), ..., P(n-1). The array element a[i] is assigned to the process P(i) upon initialization. Then the task is to calculate the result of a[0]+ a[1]+ .... +a[n]

It can be represented by the following single code program for P(i), where i=0 to n-1;

```
Sum=a[i]；//every process will have a local Sum
For（j=1；j<n；j=j*2){ 步
if(i％j=0) {
Get Sum Of process P(i+j) into a local variable tmp ;
Sum=Sum+tmp；  } }
```

Figure 11. Code to calculate the Sum

This illustration also descript how the code work:

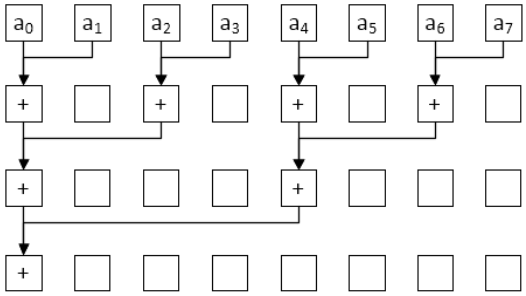

Figure 12. Illustration of aggregation

IV. COMMUNICATION MODE

Communication mode refers to the impact of some processes on other processes. There are some models for peer to peer:

- One to one. P_1 send value 1 to P_3.

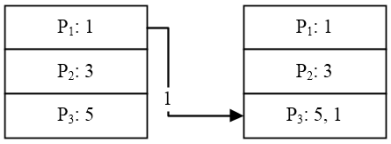

Figure 13-a. One to One

- One to *N*. In Figure 13-b. P_1 send value 1 to P_1 、P_2 、P_3 Respectively; In Figure 13-c. P_1 send value 1、3、5 to P_1 、P_2 、P_3 Respectively

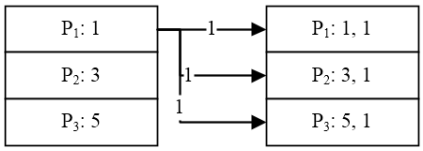

Figure 13-b. One to *N*

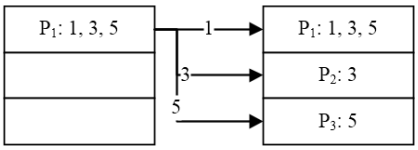

Figure 13-c. One to *N*

- *N* to One . In Figure 13-d. P_1 aggregate values from P_1 、P_2 、P_3;

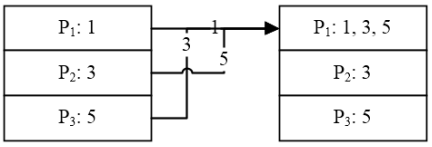

Figure 13-d. *N* to One

- *N* to *N* .. Each node sends a different message to each node; In Figure 13-f, P_1 get 1、P_2 get 1 + 3 = 4、P_3 get 1 + 3 + 5 = 9;

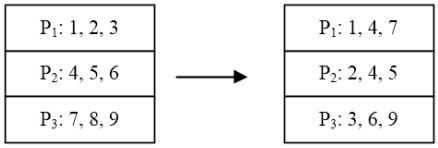

Figure 13-e. *N* to *N*

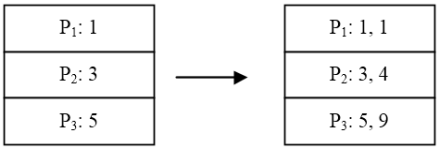

Figure 13-f. *N* to *N*

V. COMPETITIVE & COOPERATIVE COMMUNICATION

Competitive Communication: One major difference is that processes communicate in different ways. In concurrent operating systems, the interaction between processes is primarily to compete for shared resources.

Cooperative Communication: They are to communicate with each other, synchronize with each other, or generate combined values from some calculated values.

CONCLUSION

This paper expounded synchronization、aggregation and models of communication between different threads in high performance computing. As well as the advantages and disadvantages of those contents. An important way to achieve high-performance computing is to accomplish a task by collaborating with multiple machines. Communication is very important, and the quality of communication control affects the performance of computing.

REFERENCES

[1] 黄铠. 可扩展并行计算技术、结构与编程[M]. 北京:机械工业出版社, 2000. 50-54.

[2] 都志辉. 高性能计算之并行编程技术—MPI 并行程序设计[M]. 北京. 122-155.

[3] A. Karp: Programming for Parallelism. Computer . 1987

[4] Xue-Jun Yang,Yong Dou,Qing-Feng Hu. Progress and Challenges in High Performance Computer Technology[J]. Journal of Computer Science and Technology . 2006 (5).