

Parallel software applications in high-energy physics

Marek Biskup

Warsaw University, Warsaw, Poland

Abstract

Parallel programming allows the speed of computations to be increased by using multiple processors or computers working jointly on the same task. In parallel programming difficulties that are not present in sequential programming can be encountered, for instance communication between processors. The way of writing a parallel program depends strictly on the architecture of a parallel system. An efficient program of this kind not only performs its computations faster than its sequential version, but also effectively uses the CPU time. Parallel programming has been present in high-energy physics for years. The lecture is an introduction to parallel computing in general. It discusses the motivation for parallel computations, hardware architectures of parallel systems and the key concepts of a parallel programming. It also relates parallel computing to high-energy physics and presents a parallel programming application in the field, namely PROOF.

1 Parallel computing

1.1 Motivation

The speed of modern general purpose processors is of the order of 10 billion floating point operations per second (10 GFLOPS). This number seems to be unreachably high, but there are applications which have much larger requirements — weather forecasting, climate changes prediction, finance analysis, earthquake simulation, protein folding [1] and others. The accuracy of such simulations may often be improved by increasing the amount of computations. In many cases it is required that the result is known after a certain amount of time, so the execution time is limited. An example application of this kind is weather forecasting, which has to be ready for evening news, or climate change simulation, which must be ready for instance before preparing an annual report of a grant.

Traditional computer programs are sequences of instructions executed by a processor (also called a *central processing unit* — CPU) one by one. This model imposes a limit on the amount of computations performed per second. To go beyond this limit multiple CPUs or multiple computers have to be used simultaneously.

Some problems may be decomposed into smaller, independent sub-problems. In such case, each processor gets its own part and solves it independently of the other ones. This may be done, for instance, in High-Energy Physics area, with a simulation of a detector's response for 10000 events — the events may be distributed among processors. Other problems, for instance the one presented in section 1.2, cannot be decomposed in such a way. This means that processors have to cooperate to solve a problem of this kind. A parallel program uses multiple CPUs for computations and manages the communication between the processors.

Parallel programs may be run on various architectures. The architectures are often divided into two categories: single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD) [2] (see also [3] and [4]). In the first case, a single CPU instruction operates on multiple data — usually arrays of numbers. In the second case, the programs for each processor are separate, and a single instruction may operate only on single operands.

1.2 An example problem — successive overrelaxation

Before describing the parallel architectures more precisely, an example application which is to be parallelized will be described. This will be the base for examples of further sections of this lecture. The problem is rather simple, but it shows many the problems faced in the high performance computing area.

The *Successive Overrelaxation* (SOR) algorithm can be applied for example in the following problem. There is a rectangular drum made of elastic skin fastened to each side of a frame. The frame has various height in each point (Fig. 1). The goal is to predict numerically what the shape of the drum's surface (the height of the skin in each point) will be.

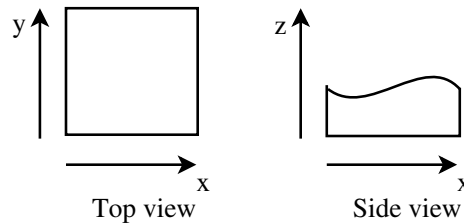


Fig. 1: Drum

Following [5], it can be assumed that the equation of the z coordinate of the skin follows (approximately) the Laplace equation:

$$\frac{d^2}{d^2x}z + \frac{d^2}{d^2y}z = 0. \quad (1)$$

The equation may be solved numerically, using the SOR method. In this method, the z function is evaluated in a regular grid. The function values form a matrix. The values on the border (on the frame) are known and fixed, because they are equal to the height of the frame at the appropriate point. At the beginning the function values inside the rectangle are initialized to some value (for example the average height of the border). In each iteration the values are replaced by the average of values of its four neighbors (Fig. 2 and equation (2)). The computations are finished when the values stabilize (within a certain accuracy).

$$f(i, j) := \frac{1}{4} (f(i + 1, j) + f(i - 1, j) + f(i, j + 1) + f(i, j - 1)) \quad (2)$$

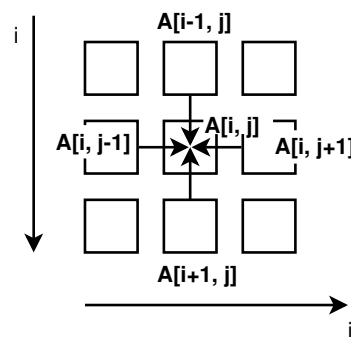


Fig. 2: A step of the SOR algorithm

1.3 SIMD architectures

The SIMD architecture, often called *Vector Architecture*, targets to optimize the typical instruction flow of a processor. When a CPU adds two numbers, it executes many internal operations: fetches the add instruction from the program memory, decodes the instruction, finds the operands and fetches them from the data memory, performs the addition and stores the result. When numbers from two arrays (two vectors) are added one by one, all those steps are repeated for each element of the result array.

If instructions are allowed to operate on arrays of numbers, redundant steps such as fetching and decoding instructions can be eliminated. Also the data may be read from memory or written to memory in bigger chunks, which would increase the throughput. A computer may contain many arithmetic units which execute such a vector operation together, in parallel. This is the idea behind the SIMD architecture.

On a vector computer the SOR method may be implemented by updating a whole row at once. To compute its new value the row above is read, the row below is added, then the current row shifted by +1 is added and the current row shifted by -1 as well. (Fig. 3). In this way, each cell will be the sum of its four neighbors. Then the elements are divided by four and stored as the result of the current iteration.

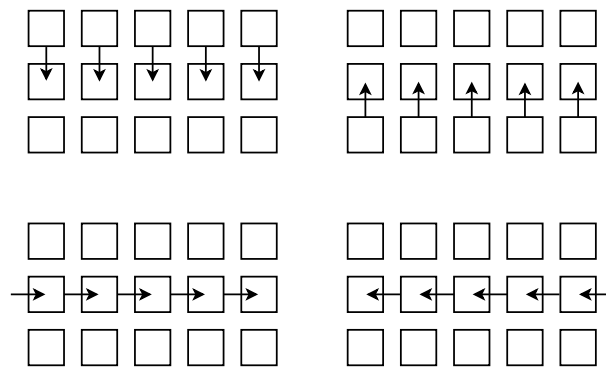


Fig. 3: SOR algorithm implementation on SIMD computers

Vector architectures are well suited for matrix operations, which are common in scientific computing. They do not work very well for every kind of computations.

The SIMD architecture was popular in the 80's. Now, nevertheless, such computers are rare. Some installations are still being introduced, usually dedicated to specialized applications, as this kind of computers has more potential for hardware optimizations.

1.4 MIMD architectures

In the MIMD architecture there are many CPUs working on one problem, but executing their own programs and communicating with one another in order to provide a single result.

The SOR algorithm can also be implemented in this model. In each iteration a new value has to be computed in every point of the grid. The computations may be divided between CPUs by assigning a group of points to each of them. A processor will update and manage only the values of its own points. First few rows may be assigned to the first processor, next rows to the second one, and so on.

The processors have to communicate to get the values of points assigned to other CPUs (picture 4). For example, CPU1 should give the content of its last row to CPU2. CPU2 should give the content of its first row to CPU1 and the content of its last row to CPU3, and so on. In a parallel program this communication usually has to be coded explicitly by a programmer.

The way of exchanging data depends on the architecture. On one side, there are computers with multiple CPUs and common memory (for example SMP machines, see next paragraph). In this case

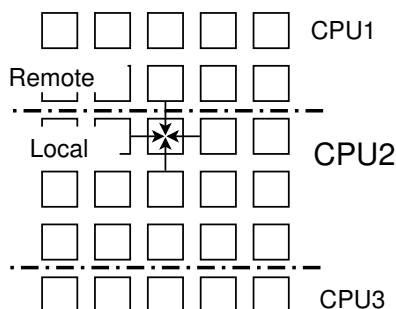


Fig. 4: SOR implementation on MIMD computers

processors may write data to be read by other CPUs. On the other side there are clusters of single-CPU computers communicating by messages sent via a network (message passing). There are also some variations in the middle. Communication in a parallel application on a MIMD machine is described in section 3.

Symmetric Multiprocessing (SMP) is an architecture in which all CPUs can access a common memory on the same rights. SMP is the technology present in desktop computers, for example in computers with multi-core CPUs [6] or double-processor main boards. Currently desktops with 2 or 4 processors are available on the market for a reasonable price. More expensive servers may use even 64 CPUs. Unfortunately the number of processors in a single computer does not scale well, because the memory access becomes a bottleneck. This means that this architecture cannot be used for constructing a computer with thousands of processors,

A different architecture which also permits a direct access to the whole memory is *NUMA* (Non Uniform Memory Access, see e.g. [7]). In this model, each processor has its own local memory, which can be accessed with maximum speed. The memory of the other processors can also be accessed directly, but with a larger delay. Efficient programming of machines with this architecture requires that data is kept "close" to the processor which operates on it most frequently.

The simplest way of building a large but relatively cheap parallel system is to connect many computers into a cluster [8]. Each computer (a node) has separate memory and operating system. The nodes exchange data using a fast network, for example Gigabit Ethernet, Myrinet or Infiniband. Usually special libraries are used in order to facilitate messages transmission.

The most efficient supercomputers use the MIMD architecture with proprietary networks connecting the processors. These computers may also implement some operations in hardware. For example *IBM Blue Gene/L* has a support network dedicated to global reduction operations, such as a sum or maximum of values in each CPU, another network for global barriers, used to synchronize the execution of a parallel program, and yet another one for disk I/O [9].

1.5 Public resource computing

Desktop computer are steadily becoming faster and faster. While browsing the Internet or doing office work only a fraction of CPU power is used. Very often a computer is turned on when numbers from two arrays (two vectors) are added one by one. Seti@home [10] pioneered the use of the power of idle desktops to perform scientific computations. The creators of this project constructed a screen-saver which did calculations. Thanks to nice visualization on the screen and a ranking system for users, the project became popular.

There are other projects based on the same idea. One of them is *LHC@home* [11], whose first application tests the stability of the beam in LHC. The project is based on Berkley Open Infrastructure for

Network Computing [12] (*BOINC*)—a framework for constructing software of this kind. The authors claim [13] that with 12000 hosts they achieve an average computing power of 3000 GFLOPS, which would place it at around the 400th position on the list of the fastest supercomputers in the world [14].

Unfortunately, this way of computing is not well suited to all tasks. Such applications must not use much bandwidth, so the use is limited only to computational-intensive applications with little communication between CPUs.

1.6 Parallel applications on desktop computers

The market requires that processor manufacturers produce faster and faster CPUs for desktop computers. It seems that there are technological obstacles which prevent the producers from increasing processors frequency at the same rate as a few years ago [15]. A workaround for this problem is to produce chips with several processors (cores) in it—the so called multi-core CPUs [6]. The manufacturers have already introduced units with two and four cores. More cores on a single chip may be expected in the coming years.

A double-core CPU has approximately twice the computing power of a standard CPU of the same family. But it does not mean programs run twice fast. An application designed to work on a single CPU does not use the second one at all and the performance is still limited by the speed of just one CPU. In order to use fully the performance of the computer, an application must run on all the processors in parallel. It means that parallel programming has become important when running scientific analysis even on desktops. These computers use the SMP architecture described in section 1.4. More demanding analysis may be run on a couple of desktops, connected with a standard Ethernet, or Wi-Fi network.

2 Creating a parallel application

Writing a parallel application is much more difficult than a sequential one. Several factors have to be taken into account in order to provide a fast and robust solution. First of all, the processors have to communicate in order to solve the problem together. The communication has to be carefully designed and tested and should take into account the architecture of the parallel computer. It also introduces an additional overhead, not present in a sequential program. The time when a CPU waits for data from another processor should be eliminated, or at least minimized. The following subsections describe these factors more precisely and give a general scheme which may be applied when designing or tuning a parallel application.

2.1 Performance metrics

The goal of using many CPUs at the same time is to reduce the time needed to solve a problem. If a program requires 100 hours on a single CPU, it could be expected that 20 CPUs would solve it 5 hours. In reality it takes longer, because of the additional communication overhead.

Performance of a parallel application is often measured in comparison to a sequential program that solves the same problem. *Speedup* (see e.g. [16]) for a program that runs on P processors is defined by:

$$\text{Speedup} = (\text{time on one processor}) / (\text{time on } P \text{ processors}). \quad (3)$$

In a perfect case the speedup can be expected to be equal P . In the case, mentioned before (100 hours on 1 CPU and 5 hours on 20 CPUs), the speedup is equal 20. On the other hand, if it took 10 hours with 20 CPUs to get the result, the speedup would be 10. *Efficiency* of a parallel program may be defined as the percentage of time the processors spend on performing calculations (the time spent on communication or waiting is not taken into account). It may be written as:

$$\text{Efficiency} = \text{Speedup} / P. \quad (4)$$

In the example of the previous paragraph the efficiency is 100% in the perfect case, or 50% in the other case. The goal is to achieve efficiency equal, or close to, 100%.

Neither the speedup nor the efficiency is a complete characterization of a parallel program. Both parameters depend on both the processors' and network's speed. The same program run with a faster network (or slower CPUs), will have larger speedup. Faster CPUs (or slower network) will result in smaller speedup.

The sequential program that is compared to the parallel program is important as well. For instance, *Bubble-Sort* is relatively easy to parallelize, but the parallel version should be compared rather to sequential *Quick-Sort* – which is much faster in the sequential case.

2.2 Parallelism overhead

The primary source of overhead in a parallel application is communication. In the example of section 2.1 there are 20 processors solving a 100-hours problem. It means that each processor will spend 5 hours on calculations. Additional time is needed for communication. If it is 1 hour per processor, the speedup is $100 / 6 = 16,7$, which gives 83% efficiency.

Another important issue is *load imbalance*. The goal is to divide the work equally among all processors, so that none of them is idle. This guarantees to finish the whole task as soon as possible. On the other hand, if one processor has twice the work of other ones, the time needed to finish the program is determined by the time needed by that processor to finish its part. If, for instance, the problem partition were changed to give a 6-hour job to one processor and to divide the rest equally into 19 jobs of about 4.95 hours, the speedup would drop to 83% (without the communication overhead). The problem seems to be artificial, but often the amount of work in parts of a problem is not known until the computations are performed. A possible solution is to use dynamic load balancing — the work is divided into smaller parts and assigned to processors as soon as they finish their previous parts. The way of doing load balancing depends on an application.

2.3 Designing a parallel application

Designing an efficient parallel application is always challenging. There is no standard procedure for converting a sequential program into a parallel one. There are however steps which are frequently done in such cases (following [3]). We shall go through these steps looking at the example from section 1.2.

First of all, the problem should be decomposed into small tasks. In most cases, a task will depend on other tasks. In the SOR example a task may be to perform computations for a single matrix element. The task will then depend on neighboring tasks, which have the values of neighboring matrix elements.

The second step is to analyze the communication required to coordinate the tasks. If two tasks are dependent on each other, they need to exchange data. In our case, a task will need the values in its neighboring cells, so it has to receive in each iteration a value from the neighboring tasks. In each iteration a task will also have to send its own value to the neighboring tasks.

In the third step the tasks are agglomerated into bigger groups. The point is to reduce communication by increasing the granularity and improving locality of dependency between tasks. The tasks which communicate the most intensively should be merged. For example a whole row in the SOR problem may be merged into a larger task. By grouping the elements in such a way, the overall communication needs are reduced — now the values which have to be fetched from other tasks are only the values of the elements above and below.

Finally the tasks are mapped to processors (so there must be more tasks than processors). Frequently communicating tasks should be mapped onto the same processor. Concurrent tasks may be placed on different processors. When mapping the tasks load balancing must also be taken into account. All the processors should have equal amounts of work. An idle processor wastes computational

resources. In our example, if there are 100 rows and 20 processors, the first 5 rows can be mapped to the first processor, next 5 rows to the second one, and so on.

The scheme presented here helps to organize the process of designing a parallel application, but what really matters are the skills and experience of a programmer.

2.4 Difficulties

As mentioned before, parallel programming introduces several factors not present in standard programming. Careless communication may result in *deadlocks*. A deadlock is a state of processors (or processes, in general) in which one of them waits for an action of others, and vice versa. Lets consider two CPUs sending messages to each other. It can be assumed that the CPU sending a message, may continue only when the other one has received the message (these are called *synchronous messages*).

CPU1: Send (CPU2, msg1) Receive(CPU2, msg2)	CPU2: Send (CPU1, msg2) Receive(CPU2, msg1)
---	---

Fig. 5: Deadlock

The programs presented in Fig. 5 will cause a deadlock — CPU1 will wait for CPU2 to receive his message 1, but CPU2 can start receiving the message only after CPU1 gets the message of CPU2. In real applications communication scheme is often more complicated, which makes this kind of errors awfully difficult to spot.

A programmer has to be aware that messages sent over a network may arrive in different order than in which they have been sent. An example is shown in Fig. 6.

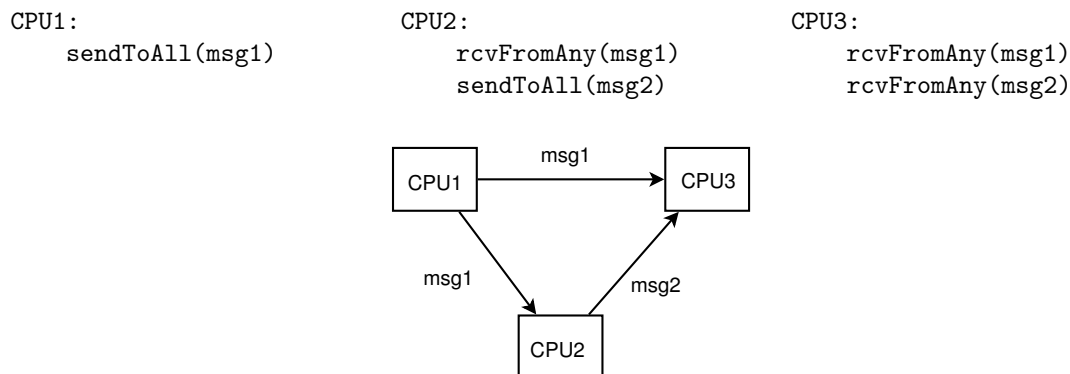


Fig. 6: Undetermined message ordering

In this case it may be expected that processor 3 receives message 1 before message 2. In reality, for example because of an overload on the connection between CPU1 and CPU2, message 2 may arrive earlier than message 1.

If two CPUs are writing a value to the same global variable (a global variable may be stored in a global memory or may be maintained by one processor; in the latter case the CPUs are sending messages to the CPU responsible for the variable), the result depends on the order in which the operations were executed. If that order is not controlled by the programmer, it is called a *race condition* [17]. The following example shows a race condition. A global variable stores the number of interesting events found in a bulk of detector events being analyzed in parallel. If a CPU finds an interesting event, it reads the number of interesting events found so far, adds one and stores the value back. A race condition will

appear if two CPUs want to update the global variable at the same time. They will both read the same value, n , both increase it by one and both store the result, $n + 1$, in the variable. But if one processor executed his code a bit earlier, the other one would read the value $n + 1$ and change it to $n + 2$, which would be correct.

All the errors described here are difficult to find and repair. An error may appear only under certain conditions and may not be discovered during tests.

3 Communication in a parallel application

Communication in a parallel application depends on the architecture. If a shared memory is present, the CPUs usually communicate using this memory. Otherwise a general scheme called message passing is used. There are also higher level methods and parallel programming languages.

3.1 Shared memory systems

Exchanging data via a global shared memory requires *synchronization* of memory access to prevent race conditions, such as presented in section 2.4. Means of synchronization are provided by the operating system or by the programming environment. The most popular ones are *mutexes*, *semaphores* [18] and *monitors* [19,20].

Mutexes allow only to lock and unlock access to global resources (for instance, to a piece of memory or to a file) i.e. resources accessible by many processes ¹. Before accessing such a resource, a process locks a mutex. If the mutex had already been locked, the process is suspended until the mutex gets unlocked. Then the process may access the global resource. This part of code is called a *critical section*. When the code finishes, the process unlocks the mutex to let other processes in. While a process is suspended, the processor may execute other processes running on the same computer.

Example pseudo-code which uses mutexes is presented in Fig. 7. It solves the race condition problem of the example of section 2.4.

```
Global Mutex m;
Global int NoInterestingEvents;

void interestingEventFound () {
    m.lock()
    int x = NoInterestingEvents;           // \
    x = x + 1;                             // | critical section
    NoInterestingEvents = x;              // /
    m.unlock();
}
```

Fig. 7: Using a mutex to guard a critical section

The critical section must be programmed carefully. If an error causes a process to skip unlocking a mutex, the mutex will be locked for other processes, causing a failure of all the computations.

There are some cases when synchronization with mutexes is difficult. Lets consider a process (the producer) which produces data to be processed by another process (the consumer; there may also be groups of processes of each kind). For instance, one process generates events and another process simulates a detector with these events. The data is exchanged by a global memory buffer. When the producing process prepares its piece of data, it puts it into the buffer. The other process takes data from the buffer. But if the buffer is empty, the consumer has to wait for the producer to put some data. To allow the producer to access the buffer, the consumer will have to do the following in a loop: lock the mutex,

¹here the term process is used because the same problems and mechanisms are present in multitasking single-CPU systems

check if the data has been supplied, unlock the mutex (the mutex has to be unlocked for the producer to be able to access the buffer) . Such a loop wastes CPU time, which may be needed by other processes running on the same computer. This problem may be solved in a correct way with two mutexes, but the solution is nontrivial.

Using *semaphores* [18], the problem can be solved directly. A semaphore contains a value and has two operations: P, which decreases the value by one, and V, which increases the value by one. The value is always greater or equal to zero. If it would drop below zero when calling P, the semaphore blocks the caller until the value is increased by another process which called V. A simple solution to the producer-consumer problem can be found using a semaphore. The semaphore's value will be initialized to zero and will hold the number of pieces of data in the buffer. The producer, after putting data into the buffer, calls V. The consumer calls P and then takes data from the buffer. The semaphore allows the consumer to access the buffer only when there is data inside. It is still important, however, that the code which accesses the buffer is guarded with a mutex. Fig. 8 presents a way of solving the producer-consumer problem using a semaphore. The reader may try to discover how to use two semaphores to solve the same problem, but with a bounded buffer.

```
Global data Buffer[];
Global int noElements = 0;
Global Semaphore s;
Global Mutex m;

put(data d) {
    m.lock();
    Buffer[noElements++] = d;
    m.unlock();
    s.V();
}

data get() {
    s.P();
    m.lock();
    d = Buffer[--noElements];
    m.unlock();
    return d;
}
```

Fig. 8: Using a semaphore for synchronization

A more structured component for synchronization is a *monitor* [19]. It is an object with two functions: *wait* and *notify*, and additional custom operations implemented by a programmer. The calls to all the custom operations are serialized, which means that there may be at most one process at a time which executes any of them. If another process tries to call such an operation, it will be suspended until the previous one finishes its operation (releases the monitor). The functions *wait* and *notify* may be called only from the custom operations. The *wait* function suspends the process and releases the monitor. Other processes may then execute custom operations. The *notify* function wakes up one of the suspended processes. That process will be able to continue from the point where it was suspended, but after the process which called *notify* finishes the monitor's custom operation. An example pseudo-code which solves the bounded buffer problem using a monitor is presented in Fig. 9.

3.2 Message passing

Message passing [21] is usually used in parallel systems, which do not have a common memory, for instance in clusters of computers or in distributed memory supercomputers. This model is based on messages from a CPU to another CPU (see Fig. 10). There are several variations of this scheme, such as synchronous or asynchronous message passing and global communication.

In *synchronous* message passing, the sender may continue only when the whole message has been sent. The recipient may continue only if it has received the whole message. Example of such communication was presented in section 2.4 (Fig. 5).

```

Monitor m {
    data Buffer[];

    put(data d) {
        Buffer[noElements++] = d;
        notify();
    }

    data get() {
        if (noElements == 0)
            wait();
        return = Buffer[--noElements];
    }
}

```

Fig. 9: Using a monitor for synchronization

```

CPU1:                CPU2:
Send(CPU2, dataPointer)  Receive(CPU1, rcvBuffer)
// continue                // continue

```

Fig. 10: Message passing

Often the operating system buffers messages. The send function just copies the message to the system buffer and the process may continue immediately. The process will wait only if the whole message does not fit into the system buffer.

A more flexible scheme is *asynchronous* message passing (see Fig. 11). The send function just tells the operating system to transfer the data to the recipient. The function returns immediately and data is being transferred in background. The process may assure that the message has been send by calling the wait function — it then will be suspended until the data transfer has finished. The recipient may receive messages asynchronously as well. The receive function provides the operating system with a memory location where the received data will be stored. The receipt is being done in background and calling wait assures that the message has arrived — the process will be suspended until the whole message has been received.

```

CPU1:                CPU2:
int outBuffer[SIZE];    int inBuffer[SIZE];
doComputations1();      doComputations2();
id1 = sendAsync(CPU2, outBuffer, SIZE);  id2 = receiveAsync(CPU1, inBuffer, SIZE);
continueComputations1();  continueComputations2();
wait(id);                wait(id2);
reuseOutBuffer();        useInBuffer();

```

Fig. 11: Asynchronous message passing

Message passing may also involve global communication, where messages are sent to all (broadcasting) or selected (multicasting) CPUs.

A programmer may use *MPI* [22], which is a standard interface for message passing. It includes the modes of message passing mentioned before. In addition it allows global reduction operations — for instance to compute a sum of values kept in each processor. MPI has implementations on many hardware platforms and for many languages. Programs written with MPI may be easily ported from one platform on another. There are also other libraries which support message passing.

3.3 Higher level methods

Programming with messages is flexible, but often difficult. There are other ways of communication, which make it more natural and easy to program. *RPC* [23] is a way of executing a procedure on a remote computer. The procedure will be run in a dedicated process. Such a process may communicate with a process running on that computer using common memory, as described in section 3.1. *RMI* [24] is an object-oriented way of doing RPC. It is included in the Java programming language. *High Performance Fortran* (HPF) [25] is an extension of Fortran, dedicated to parallel computations. It works well for matrix or array operations. A programmer specifies how portions of arrays will be distributed and the compiler parallelizes the operations on arrays. HPF is easy to program, because it handles all the communication. Unfortunately it cannot be applied to irregular data structures. There are many other types of parallel languages, for instance OpenMP, Occam or Linda.

4 Parallel computing in high-energy physics

Computers are extensively used in High-Energy Physics. This section will discuss only a small fraction of HEP applications and should not be regarded as an overview of HEP computing.

4.1 Event processing

Event processing, i.e. simulation, reconstruction and analysis, in most cases involves processing a large number of events — of the order of thousands, millions and more. The events are independent of each other, so such an application may be trivially parallelized by distributing events on CPUs. The CPUs work independently and at the end the results are merged.

These applications are often data intensive, so the performance of a parallel application may be limited by the throughput to a data storage system. This also makes using public resource computing almost impossible for this area.

4.2 Batch data processing

Event processing is often done off-line using batch systems. Such a system manages a cluster of computers and allows a user to submit a program (called a *job* in this case) to be executed. A batch system starts a new job as soon as there is an available processor. Otherwise the job waits in a queue. An example of a batch system is the *lxbatch* cluster at CERN. Also a computing grid may be used as a large batch system.

For processing events, they are often divided by a user into parts and each part is submitted separately. This allows a batch system to process events in parallel, on different computers of the cluster. Another advantage of splitting events is that if processing one event fails, it should not influence processing of other parts.

The time needed to process a single event may vary. It follows that splitting events into equal parts may introduce a load imbalance. Such an imbalance is tolerable because the delay caused by queuing in a batch system is often much longer.

4.3 Interactive data analysis

Interactive computing is opposite to batch computing. The user is given immediate feedback of the computations, may modify parameters and start the processing again. In this case the time of computation is crucial, so load balancing a parallel application is important.

An example parallel application for interactive event analysis, *PROOF* [26], is described in section 5.

4.4 Other parallel applications in HEP

Apart from the trivially parallelizable programs discussed before, there are also problems, for which the parallelization is more complicated, for instance multivariate function fitting or lattice QCD simulation. For the latter there are even custom parallel processors being used (e.g. the *apeNEXT* project, [27]).

5 Example application — PROOF

PROOF [26], Parallel ROOT Facility, is a part of the *ROOT* project [28]. *ROOT* is a framework for HEP data analysis. It includes functionalities such as multidimensional histogramming, fitting, visualization or I/O. *PROOF* allows a user to run interactive *ROOT* analysis in a parallel environment — either on a cluster of computers or on a single multi-CPU machine. *PROOF* should be regarded rather as a large distributed system and not only a parallel program.

5.1 Overview

A typical event analysis program in *ROOT* iterates over a set of events and fills histograms. The values which are histogrammed are results of processing a single event. Each event is independent of the other ones. A user may specify an expression to be histogrammed and give the range of events to be processed. Such an expression may use the parameters of an event. For instance, if an event contains the coordinates of the momentum, px , py and pz , the expression may compute its absolute value, $\sqrt{px * px + py * py + pz * pz}$. More complicated expressions and custom analysis code may be wrapped into a C++ program. *ROOT* contains a C++ compiler able to compile a source program and load it at runtime.

ROOT itself is a sequential program. An analysis on a multi-core CPU would use only one core. *ROOT* may be started in several instances, but manually splitting event ranges and then adding histograms is inconvenient. The same applies to batch analysis on a large dataset.

The primary goal of *PROOF* was to allow analysis to be runned transparently (without an additional effort) on a cluster of computers, in parallel. When multi-core CPUs emerged, it became obvious that *PROOF* would also be useful on a single machine. *PROOF* divides the set of events into small packets and distributes those packets among CPUs. Each processor fills its own histograms, based on events assigned it. *PROOF* merges the results into one histogram and give it as a single output.

The parallelization of computations is trivial in this case. *PROOF*'s main success is transparency and interactivity. A user may use the same GUI and functions to draw histograms in parallel as he used in the standard (sequential) mode [29]. In some cases a program has to be modified to comply with the standards introduced for *PROOF*. Such a modified program may be run in both parallel and sequential modes. When running an analysis the user may view the progress of the work and the temporary results.

5.2 Architecture

The *PROOF* system consists three primary layers (tiers): *PROOF client*, *PROOF master* and *PROOF slaves* (see Fig. 12). Each tier is a separate process and may run on different CPUs. The client is attached to the standard *ROOT* program. The master and the slaves are located on the cluster where the computations take place. The master manages the work of the slaves. The slaves do the computations.

The client supplies the master with the input for the analysis: the data files (or rather their locations — files are accessed through a network), the analysis code and the events' range. The master hands over the analysis code to the slaves and divides the events range into smaller packets. There are more packets than the number of slaves.

Each slave compiles the analysis code using *CINT* — the *ROOT*'s internal compiler. Then it gets a packet from the master. After finishing it, it gets another one, and so on. This dynamic load balancing reduces the idle time of slaves and is especially useful if computers have different speed. When there are no more packets, a slave sends the result to the master. The master waits until it receives results from all

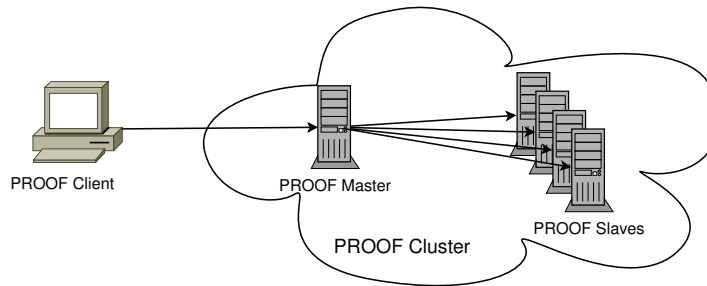


Fig. 12: PROOF's architecture

slaves, adds the histograms and sends the result to the client. The client draws the histograms as if the analysis were done locally.

Analysis may also be done in a preview mode, in which the user sees his histograms growing during computations. In this case the slaves send their temporary results to the master repeatedly, after a certain time. The master merges the results and sends the preview to the client, where the histograms are drawn.

The communication overhead in PROOF is small, because the slaves are independent of each other. The amount of data sent from each slave to the master is small (when the preview mode is off; if it is on, more data is sent). The bottleneck in this case may be the file access. The files are usually located on a central data server and the bandwidth between the cluster's nodes and the data server may be insufficient if the number of slaves is large. This problem can be eliminated by replicating files on cluster nodes.

5.3 Other features

PROOF has many other functionalities than described in the previous subsections, for instance batch processing support [REF], authentication mechanisms, multi-user support, GUI. This lecture described only the features related to parallel programming.

6 Summary

Parallel programming is important in high performance computing, and, due to multi-core CPU technology, also in desktop computing. It allows computations to be done faster by using more processors or computers at the same time. Writing a parallel application is more difficult than writing a sequential one. Not only there are additional sources of errors, for instance communication between processors, but also a parallel program must efficiently use the CPU time. Most applications in HEP, such as those presented in section 4, are trivial to parallelize. But even in those application there are some other concept, such as easiness of use or transparency, which should be taken care of.

Acknowledgements

The material covered in this lecture was presented by the author at the *inverted CERN School of Computing* (iCSC) 2006 initiated and organised by François Flückiger from CERN. The publication of the material of iCSC was initiated by Liliana Teodorescu from the Brunel University, United Kingdom. Parts of this lecture were based on the lecture *Parallel Programming* given by prof. Henri Bal at Vrije Universiteit in Amsterdam. The author thanks the people mentioned here.

References

- [1] F. Allen *et al.*, *IBM Systems J.*, **40(2)** (2001) 310;
<http://www.research.ibm.com/journal/sj/402/allen.pdf>.
- [2] M.J. Flynn, *IEEE Trans. on Comput.*, **C-21** (1972) 948.
- [3] I. Foster, *Designing and Building Parallel Programs*, (Addison-Wesley, 1995); online book also available at the URL <http://www-unix.mcs.anl.gov/dbpp/>.
- [4] TOP500.org, *The Main Architectural Classes*,
<http://www.top500.org/orsc/2006/architecture.html>.
- [5] D. Griffiths, *Introduction to Electrodynamics*, 3rd ed. (Prentice-Hall, NJ, 1999).
- [6] R.M. Ramanathan, *Technology@Intel Mag.* (Sept. 2005); <http://www.intel.com/technology/magazine/computing/multi-core-0905.pdf>.
- [7] TOP500.org, *ccNUMA machines*, <http://www.top500.org/orsc/2006/ccnuma.html>.
- [8] TOP500.org, *Clusters*, <http://www.top500.org/orsc/2006/clusters.html>.
- [9] A. Gara *et al.*, *IBM J. Research and Development*, **49(2/3)** (2005);
<http://www.research.ibm.com/journal/rd/492/gara.pdf>.
- [10] Seti@home project's website, <http://setiathome.berkeley.edu>.
- [11] LHC@home project's website, <http://lhcatome.cern.ch>.
- [12] BOINC project's website, <http://boinc.berkeley.edu>.
- [13] J. Klem, *Public Resource Computing at CERN — LHC@home*, talk at CHEP 2006, Mumbai, India.
- [14] TOP500.org, <http://www.top500.org>.
- [15] Geoff Koch, *Technology@Intel Magazine* (July 2005);
<http://www.intel.com/technology/magazine/computing/multi-core-0705.pdf>.
- [16] M.J. Quinn, *Parallel Computing — Theory and Practice* (McGraw-Hill, 1994), Chap. 3.
- [17] T.W. Christopher and G.K. Thiruvathukal, *High-Performance Java Platform Computing* (The SUN Microsystems Press, 2001), Chap. 3;
<http://java.sun.com/developer/Books/performance2/chap3.pdf>
- [18] E.W. Dijkstra, *Cooperating sequential processes*, (Technological University, Eindhoven, The Netherlands, Sept. 1965). Reprinted in *Programming Languages*, F. Genuys, Ed., (Academic Press, New York, 1968), 43-112.
- [19] C.A.R. Hoare, *Commun. of the ACM*, **17(10)** (1974) 549.
- [20] T.W. Christopher and G.K. Thiruvathukal, *High-Performance Java Platform Computing* (The SUN Microsystems Press, 2001), Chap. 4;
<http://java.sun.com/developer/Books/performance2/chap4.pdf>
- [21] H.E. Bal, *Interprocess Communication and Synchronization based on Message Passing* (Vrije Universiteit, Amsterdam, 1995).
- [22] J.J. Dongarra, *et al.*, *Commun. of the ACM* **39(7)** (1996) 84.
- [23] R. Srinivasan, *RPC: Remote Procedure Call Protocol Specification Version 2*, Request for Comments: 1831, <http://tools.ietf.org/html/rfc1831>.
- [24] J. Waldo, *IEEE Concurrency* **6(3)** (1998) 5.
- [25] H. Richardson, *Tech. Rep. TMC-261* (Thinking Machines Corporation, April 1996).
- [26] M. Ballintijn *et al.*, The PROOF — Distributed Parallel Analysis Framework based on ROOT, Proceedings CHEP03 Workshop, La Jolla, California, 2003;
<http://arxiv.org/abs/physics/0306110>.
- [27] R. Alfieri *et al.*, apeNEXT: A Multi-Tflops LQCD Computing Project,
eprint [arXiv:hep-lat/0102011](http://arxiv.org/abs/hep-lat/0102011).
- [28] R. Brun and F. Rademakers, *Nucl. Instrum. Methods Phys. Res. A* **389** (1997) 81. See also
<http://root.cern.ch>.
- [29] M. Ballintijn *et al.*, *Nucl. Instrum. Methods Phys. Res. A* **559** (2006) 13.