Nakano, Junji

Working Paper

Parallel computing techniques

Papers, No. 2004,27

Provided in Cooperation with:
CASE - Center for Applied Statistics and Economics, Humboldt University Berlin

Suggested Citation: Nakano, Junji (2004) : Parallel computing techniques, Papers, No. 2004,27, Humboldt-Universität zu Berlin, Center for Applied Statistics and Economics (CASE), Berlin

This Version is available at:
https://hdl.handle.net/10419/22200

Parallel computing techniques

Junji Nakano[1]

The Institute of Statistical Mathematics nakanoj@ism.ac.jp

1 Introduction

Parallel computing means to divide a job into several tasks and use more than one processor simultaneously to perform these tasks. Assume you have developed a new estimation method for the parameters of a complicated statistical model. After you prove the asymptotic characteristics of the method (for instance, asymptotic distribution of the estimator), you wish to perform many simulations to assure the goodness of the method for reasonable numbers of data values and for different values of parameters. You must generate simulated data, for example, 100 000 times for each length and parameter value. The total simulation work requires a huge number of random number generations and takes a long time on your PC. If you use 100 PCs in your institute to run these simulations simultaneously, you may expect that the total execution time will be 1/100. This is the simple idea of parallel computing.

Computer scientists noticed the importance of parallel computing many years ago (Flynn, 1966). It is true that the recent development of computer hardware has been very rapid. Over roughly 40 years from 1961, the so called "Moore's law" holds: the number of transistors per silicon chip has doubled approximately every 18 months (Tuomi, 2002). This means that the capacity of memory chips and processor speeds have also increased roughly exponentially. In addition, hard disk capacity has increased dramatically. Consequently, modern personal computers are more powerful than "super computers" were a decade ago. Unfortunately, even such powerful personal computers are not sufficient for our requirements. In statistical analysis, for example, while computers are becoming more powerful, data volumes are becoming larger and statistical techniques are becoming more computer intensive. We are continuously forced to realize more powerful computing environments for statistical analysis. Parallel computing is thought to be the most promising technique.

However, parallel computing has not been popular among statisticians until recently (Schervish, 1988). One reason is that parallel computing was

available only on very expensive computers, which were installed at some computer centers in universities or research institutes. Few statisticians could use these systems easily. Further, software for parallel computing was not well prepared for general use.

Recently, cheap and powerful personal computers changed this situation. The Beowulf project (Sterling et al., 1999), which realized a powerful computer system by using many PCs connected by a network, was a milestone in parallel computer development. Freely available software products for parallel computing have become more mature. Thus, parallel computing has now become easy for statisticians to access.

In this chapter, we describe an overview of available technologies for parallel computing and give examples of their use in statistics. The next section considers the basic ideas of parallel computing, including memory architectures. Section 3 introduces the available software technologies such as process forking, threading, OpenMP, PVM (Parallel Virtual Machine), MPI (Message Passing Interface) and HPF (High Performance Fortran). The last section describes some examples of parallel computing in statistics.

2 Basic ideas

Two important parts of computer hardware are the processor, which performs computations, and memory, in which programs and data are stored. A processor is also often called a central processing unit (CPU). Modern computer systems adopt a stored programming architecture: all the program instructions are stored in memory together with processed data and are executed sequentially by a processor according to the instructions.

In a traditional single processor computer, a single stream of instructions is generated from the program, and these instructions operate on a single stream of data. Flynn (1966) called this arrangement a single instruction stream–single data stream (SISD) computer.

On the other hand, a parallel computer system uses several processors, and is realized as a single instruction stream–multiple data stream (SIMD) computer or a multiple instruction stream–multiple data stream (MIMD) computer. SIMD refers to a form of parallel execution in which all processors execute the same operation on different data at the same time, and is often associated with performing the same operation on every element of a vector or array. MIMD refers to parallel execution in which each processor works independently; for example, one processor might update a database file while another processor handles a graphic display.

The fundamental software of a modern computer system is an operating system such as UNIX or Microsoft Windows. They support multiple users and multiple tasks, even on single processor systems, by adopting time-slicing mechanisms, in which a processor executes tasks cyclically. In parallel computer systems, some tasks are executed on different processors simultaneously.

2.1 Memory architectures of parallel computers

The traditional computer system has a single processor (or CPU) that can access all of the memory (Fig. 1). Parallel computers use more than one processor simultaneously for a single calculation task. There are two simple methods to increase the number of available processors in a single system. One method is to add processors to the traditional single processor system without changing other parts. Because all the memory is shared by all processors, such systems are called shared memory systems (Fig. 2). An example of a shared memory system is a dual processor personal computer, where the motherboard has two sockets for CPUs. When we mount one CPU, it works as a traditional single processor system. If we mount two CPUs, both processors can access all the memory in the PC, and it works as a shared memory system. A second method is to connect traditional single processor computers by a network. This is called a distributed memory system, because the memory is used by a single processor locally and is "distributed" over the whole system (Fig. 3). An example of a distributed memory system is a network of workstations, in which each node computer works independently and communicates with the others through a network to solve a single problem.

Integration of shared memory and distributed memory is possible (Fig. 4). Network-connected PCs that each have two processors can be considered a distributed shared memory system.

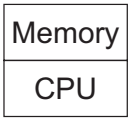

Fig. 1. Traditional system

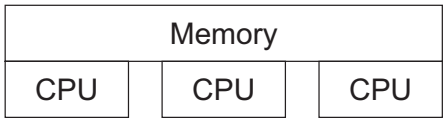

Fig. 2. Shared memory system

Shared memory systems

In the simple shared memory realization, all the processors can access all the memory at the same speed by using a common memory bus. This is known as a uniform memory access (UMA) configuration. Performance in a UMA system

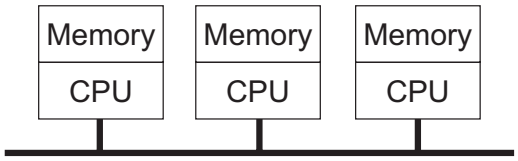

Fig. 3. Distributed memory system

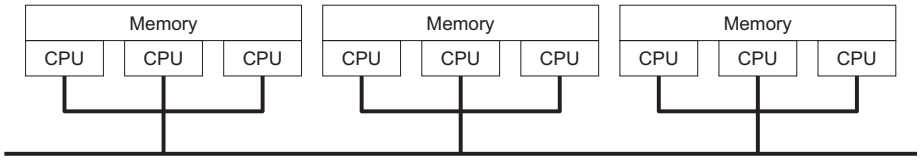

Fig. 4. Distributed shared memory system

is limited by the memory bus bandwidth; adding processors to the system beyond some point does not increase performance linearly, because signals from processors flow on the same memory bus and often cause collisions. Typically, UMA configurations do not scale well beyond 10 to 20 processors.

To improve communication between processors and memory, a non-uniform memory access (NUMA) configuration is used. In NUMA systems, all processors have access to all the memory, but the cost of accessing a specific location in memory is different for different processors, because different regions of memory are on physically different buses. Even if we adopt a NUMA configuration, it is not efficient to use more than 100 processors in a shared memory system.

A shared memory system is also a symmetric multiprocessor (SMP) system, in which any processor can do equally well any piece of work.

In a shared memory system, a single copy of an operating system is in charge of all the processors and the memory. It usually uses a programming model called "fork–join". Each program begins by executing just a single task, called the master. When the first parallel work is reached, the master spawns (or forks) additional tasks (called slaves or workers), which will "join" to the master when they finish their work (the middle figure in Fig. 5). Such activities can be programmed by using software technologies such as process, thread or OpenMP, which will be explained in the next section.

Distributed memory systems

In a distributed memory system, each node computer is an independent computer that has, at least, processor and memory, and the nodes are connected together by a network. This so called "network of workstations" (NOW) is the cheapest way to construct a distributed memory system, because we can utilize many different kinds of workstations available, connected by a network, without adding any new hardware. However, NOW is sometimes ineffective

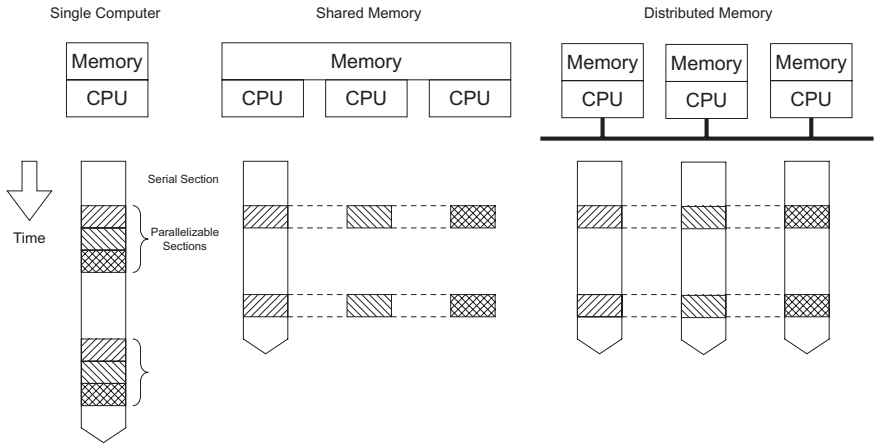

Fig. 5. Typical parallel computing execution

for heavy computation, because, for example, general purpose networks are slow, and nodes may be unexpectedly used for other work, so that it is difficult to schedule them efficiently.

Nowadays, "Beowulf class cluster computers" are popular for distributed memory parallel computing (Sterling et al., 1999). These are a kind of NOW, but there are slight differences. First, the nodes in the cluster are the same kind of workstation or PC, and are dedicated to the cluster calculation tasks. Typically, node computers share the working directory on the hard disk and have no display or keyboard. The interconnection network is isolated from external networks and is also dedicated to the cluster, and communication among the nodes can be done without further authentication. Operating system parameters are tuned to improve the total performance for parallel computing. All these characteristics help the performance of the parallel computing on the cluster.

Distributed memory systems have no memory bus problem. Each processor can use the full bandwidth to its own local memory without interference from other processors. Thus, there is no inherent limit to the number of processors. The size of the system is constrained only by the network used to connect the node computers. Some distributed memory systems consist of several thousand processors.

As nodes in a distributed memory system share no memory at all, exchange of information among processors is more difficult than in a shared memory system. We usually adopt a message passing programming model on a distributed memory system; we organize a program as a set of independent tasks that communicate with each other via messages. This introduces two sources of overhead: it takes time to construct and send a message from one

processor to another, and the receiving processor must be interrupted to deal with messages from other processors.

Available message passing libraries are PVM and MPI The right figure in Fig. 5 shows an execution image of MPI. HPF is also mainly used in distributed memory systems. These libraries are illustrated in the next section.

2.2 Costs for parallel computing

We expect that the calculation speed increases n times if we use n processors instead of one. We also wish to use multiprocessor systems just like an ordinary single processor system. However, some costs are incurred in realizing parallel computing. They include the non-parallel characteristics of the problem, communication costs such as distributing and gathering data and/or programs, the difficulty of programming for synchronization among executions and unexpected influences of cache memory. All these factors reduce the effect of parallelization.

Amdahl's law

All programming tasks include non-parallelizable or serial parts, which cannot be executed on several processors, for example, summarizing calculation results and writing them to the display or file. Assume the ratio of computing time for the serial parts to the whole task is f $(0 < f < 1)$. If a single processor requires t_s time to complete the task, $(1 - f)t_s$ computation time is used for the parallelizable task and ft_s computation time is used for the serial task. If we use n processors, the elapsed time for execution of the parallelizable task will be at least $(1 - f)t_s/n$, while the execution time of the serial task remains ft_s. Thus, the ratio of execution time for n processors to that for one processor, $S(n)$, which is called the speedup factor, is

$$S(n) = \frac{t_s}{ft_s + (1 - f)t_s/n} = \frac{n}{1 + (n - 1)f}.$$

This equation is known as "Amdahl's law" (Amdahl, 1967). When n is large, it converges to $1/f$, that is, the effect of parallel computing is limited. For example, if $f = 5\%$, the maximum possible speedup is 20, even if we use an infinite number of processors. This may discourage the use of parallel computing.

Of course, as f goes to zero, $S(n)$ converges to n, which is an ideal situation.

Gustafson's law

Amdahl's law considers the situation where the task size is fixed and the number of processors increases. In real problems, however, we wish to perform larger tasks when the number of processors increases. For example, assume time s is required for preparing a task, and time p is required for the

(moderate) simulation task. When a parallel computer is available, we wish to perform more simulations, typically, n times larger simulations than the original ones by n processors. To perform this simulation, a single processor system requires $s + np$ time, while the n-processor system requires $s + p$ time. The speedup factor is

$$S(n) = \frac{s + np}{s + p}.$$

This equation is called "Gustafson's law" (Gustafson, 1988). Note that if we define $f = s/(s + np)$, this is as same as Amdahl's law. However, when n becomes large, $S(n)$ becomes large linearly. This means that parallel computing is useful for large-scale problems in which the serial part does not increase as the problem size increases. If s approaches zero, $S(n)$ converges to n, the ideal situation.

Other costs

If we divide one task into several small tasks and execute them in parallel, we must wait until all the child tasks have been completed: we must synchronize executions. As the slowest child task determines the total execution time, child tasks should be designed to have almost the same execution times, otherwise some processors may be idle while others have tasks queuing for execution. Techniques that aim to spread tasks among the processors equally are called load balancing and are not easy.

In a shared memory system, exchange of information among processors is performed by variables stored in the shared memory. If several tasks use one variable almost simultaneously, it may cause trouble. Consider two tasks trying to decrease the value of variable x by one. Assume $x = 3$; task 1 obtains this value, decreases it and writes 2 into x. If task 2 tries to do the same task before task 1 finishes its work, task 2 also obtains the value 3, and writes 2 into x. Thus, the final result is 2, although x should have decreased twice. To avoid such a maloperation, task 2 must wait until task 1 finishes. All parallel computing software can handle this synchronization problem, typically by using a lock-unlock mechanism.

An important hardware aspect of shared memory systems is cache memory. As the advances in main memory technology do not keep up with processor innovations, memory access is very slow compared with processor speed. In order to solve this problem, another layer of memory has been added between a processor and main memory, called the cache. It is a small amount of very fast, expensive memory, that operates close to the speed of the processor. A separate cache controller monitors memory accesses and loads data and instructions in blocks of contiguous locations from memory into the cache. Once the content of memory is stored in the cache, the processor can operate at full speed by using them. Sometimes, the cache contents are different from the necessary ones. In these cases, the processor is stalled and has to wait

while the necessary data is newly loaded from memory into the cache. This mechanism works well in a single processor system.

All processors in a shared memory system have their own caches. Suppose several processors access the same location of memory and copy them into their caches. If one processor changes the value of the memory in that location, other processors should not use the value in their caches. A cache coherence protocol is used to notify this information among caches. A common cache coherence protocol is an invalidate policy; when one copy of memory is altered, the same data in any other cache is invalidated (by resetting a valid bit in the cache). In shared memory systems, cache coherence is done in the hardware and the programmer need not worry about cache coherence. However, it may cause the slowdown of the calculation speed. Note that caches handle blocks of memory. If one processor writes to one part of the block, copies of the whole block in other caches are invalidated though the actual data is not shared. This is known as false sharing and can damage the performance of the cache in a shared memory system. We are sometimes required to write programs considering the amount of the cache memory in a shared memory system to achieve enough performance.

Distributed memory systems require communication among node computers. Such communication is affected by several factors, including network bandwidth, network latency and communication latency. Network bandwidth is the number of bits that can be transmitted in unit time. Network latency is the time to prepare a message for sending it through the network. Communication latency is the total time to send the message, including software overhead and interface delays. Generally, communication is expensive compared with processor work.

If a problem can be divided into small tasks that are completely independent and require no or very little synchronization and communication, the problem is called "embarrassingly parallel". Clearly, embarrassingly parallel problems are particularly suitable for parallel computing.

3 Parallel computing software

Several well-designed software technologies are available for utilizing parallel computing hardware. Note that each of them is suitable for a specific hardware architecture.

In this section, we use as an example the calculation of the value of π by the approximation formula

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \sim \frac{1}{n} \sum_{i=1}^{n} \frac{4}{1+(\frac{i-0.5}{n})^2}.$$

The case $n = 10$ is illustrated in Fig. 6.

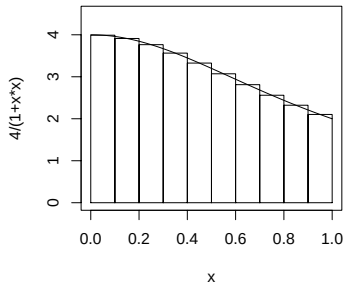

Fig. 6. Calculation of π

A C program to calculate the last term is given in Listing 1. The main calculation is performed in the `for` statement, which is easily divided into parallel-executed parts; this is an example of an embarrassingly parallel problem. We show several parallel computing techniques by using this example in this section. We choose this simple example to keep the length of following example source codes as small as possible and to give a rough idea of parallel computing techniques. Note that this example is so simple that only the most fundamental parts of each technique will be used and explained. Many important details of each technique are left to references.

Listing 1

```
#include <stdio.h>

main(int argc, char **argv)
{
  int n, i;
  double d, s, x, pi;
  n = atoi(argv[1]);
  d = 1.0/n;
  s = 0.0;
  for (i=1; i<=n; i++){
    x = (i-0.5)*d;
    s += 4.0/(1.0+x*x);
  }
  pi = d*s;
}
```

3.1 Process forking

Modern operating systems have multi-user and multi-task features even on a single processor; many users can use a single processor system and can

seemingly perform many tasks at the same time. This is usually realized by multi-process mechanisms (Tanenbaum, 2001).

UNIX-like operating systems are based on the notion of a process. A process is an entity that executes a given piece of code, has its own execution stack, its own set of memory pages, its own file descriptors table and a unique process ID. Multiprocessing is realized by time-slicing the use of the processor. This technology repeatedly assigns the processor to each process for a short time. As the processor is very fast compared with human activities, it looks as though it is working simultaneously for several users. In shared memory systems, multiprocessing may be performed simultaneously on several processors. Multiprocessing mechanisms are a simple tool for realizing parallel computing.

We can use two processes to calculate the `for` loop in Listing 1, by using the process-handling functions of UNIX operating systems: `fork()`, `wait()` and `exit()`. The function `fork()` creates a new copy process of an existing process. The new process is called the child process, and the original process is called the parent. The return value from `fork()` is used to distinguish the parent from the child; the parent receives the child's process id, but the child receives zero. By using this mechanism, an `if` statement, for example, can be used to prescribe different work for the parent and the child. The child process finishes by calling the `exit()` function, and the parent process waits for the end of the child process by using the `wait()` function. This fork–join mechanism is fundamental to the UNIX operating system, in which the first process to start invokes another process by forking. This procedure is repeated until enough processes are invoked. Although this mechanism was originally developed for one processor and a time-slicing system, UNIX operating systems that support shared memory can run processes on different processors simultaneously.

As processes are independent and share only a limited set of common resources automatically, we must write a program for information exchange among processes. In our example, we use functions to handle shared memory segments: `shmget()`, `shmat()` and `shmctl()`. `shmget()` allocates a shared memory segment, `shmat()` attaches the shared memory segment to the process, and `shmctl()` allows the user to set information such as the owner, group and permissions on the shared memory segment. When the parent process uses `fork()`, the shared memory segment is inherited by the child process and both processes can access it.

Listing 2 shows a two-process version of Listing 1. In the `for` statement, the parent process works for $i = 2, 4, 6, \ldots$, while the child process works for $i = 1, 3, 5, \ldots$. The child process stores its result to `*shared` and the parent process receives the value and adds it to its own result, then prints the final result.

Listing 2

```
#include <stdio.h>
#include <sys/types.h>
```

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

main(int argc, char **argv)
{
  int n, i;
  double d, s, x, pi;
  int shmid, iproc;
  pid_t pid;
  double *shared;
  n = atoi(argv[1]);
  d = 1.0/n;
  shmid = shmget(IPC_PRIVATE,
                 sizeof(double), (IPC_CREAT | 0600));
  shared = shmat(shmid, 0, 0);
  shmctl(shmid, IPC_RMID, 0);
  iproc = 0;
  if ((pid = fork()) == -1) {
    fprintf(stderr, "The fork failed!\n");
    exit(0);
  } else {
    if (pid != 0) iproc = 1 ;
  }
  s = 0.0;
  for (i=iproc+1; i<=n; i+=2) {
    x = (i-0.5)*d;
    s += 4.0/(1.0+x*x);
  }
  pi = d*s;
  if (pid == 0) {
    *shared = pi;
    exit(0);
  } else {
    wait(0);
    pi = pi + *shared;
  }
}
```

Forking, however, is not appropriate for parallel computing. Much time and memory is required to duplicate everything in the parent process. Further, a complete copy is not always required, because, for example, the forked child process starts execution at the point of the fork.

3.2 Threading

As a process created using the UNIX `fork()` function is expensive in setup time and memory space, it is sometimes called a "heavyweight" process. Often a partial copy of the process is enough and other parts can be shared. Such copies can be realized by a thread or "lightweight" process. A thread

is a stream of instructions that can be scheduled as an independent unit. It is important to understand the difference between a thread and a process. A process contains two kinds of information: resources that are available to the entire process such as program instructions, global data and working directory, and schedulable entities, which include program counters and stacks. A thread is an entity within a process that consists of the schedulable part of the process.

In a single processor system, threads are executed by time-slicing, but shared memory parallel computers can assign threads to different processors.

Pthread library

There were many thread libraries in the C language for specific shared memory systems. Now, however, the Pthread library is a standard thread library for many systems (Butenhof, 1997). The Pthread API is defined in the ANSI/IEEE POSIX 1003.1-1995 standard, which can be purchased from IEEE.

Listing 3 is an example program to calculate π by using the Pthread library. The program creates a thread using the function `pthread_create()`, then assigns a unique identifier to a variable of type `pthread_t`. The caller provides a function that will be executed by the thread. The function `pthread_exit()` is used to terminate itself. The function `pthread_join()` is analogous to `wait()` for forking, but any thread may join any other thread in the process, that is, there is no parent–child relationship.

As multi-threaded applications execute instructions concurrently, access to process-wide (or interprocess) shared memory requires a mechanism for coordination or synchronization among threads. It is realized by mutual exclusion (mutex) locks. Mutexes furnish the means to guard data structures from concurrent modification. When one thread has locked the mutex, this mechanism precludes other threads from changing the contents of the protected structure until the locker performs the corresponding mutex unlock. Functions `pthread_mutex_init()`, `pthread_mutex_lock()` and `pthread_mutex_unlock()` are used for this purpose.

The compiled executable file is invoked from a command line with two arguments: `n` and the number of threads, which is copied to the global variable `num_threads`. The ith thread of the function `PIworker`, which receives the value i from the original process, calculates a summation for about `n/num_threads` times. Each thread adds its result to a global variable `pi`. As the variable `pi` should not be accessed by more than one thread simultaneously, this operation is locked and unlocked by the mutex mechanism.

Listing 3

```
#include <stdio.h>
#include <pthread.h>
int n, num_threads;
```

```
double d, pi;
pthread_mutex_t reduction_mutex;
pthread_t *tid;

void *PIworker(void *arg)
{
  int i, myid;
  double s, x, mypi;
  myid = *(int *)arg;
  s = 0.0;
  for (i=myid+1; i<=n; i+=num_threads) {
    x = (i-0.5)*d;
    s += 4.0/(1.0+x*x);
  }
  mypi = d*s;
  pthread_mutex_lock(&reduction_mutex);
  pi += mypi;
  pthread_mutex_unlock(&reduction_mutex);
  pthread_exit(0);
}

main(int argc, char **argv)
{
  int i;
  int *id;
  n = atoi(argv[1]);
  num_threads = atoi(argv[2]);
  d = 1.0/n;
  pi = 0.0;
  id = (int *) calloc(n,sizeof(int));
  tid = (pthread_t *) calloc(num_threads,
                             sizeof(pthread_t));
  if(pthread_mutex_init(&reduction_mutex,NULL)) {
    fprintf(stderr, "Cannot init lock\n");
    exit(0);
  };
  for (i=0; i<num_threads; i++) {
    id[i] = i;
    if(pthread_create(&tid[i],NULL,
                      PIworker,(void *)&id[i])) {
      exit(1);
    };
  };
  for (i=0; i<num_threads; i++)
    pthread_join(tid[i],NULL);
}
```

We note that it is not easy to write multi-threaded applications in the C language, even if we use the Pthread library. As the Pthread library was added to the C language later, there are no assurances that original basic libraries are “thread-safe”. The term thread-safe means that a given library function

is implemented in such a manner that it can be executed correctly by multiple concurrent threads of execution. We must be careful to use thread-safe functions in multi-thread programming. The Pthread library is mainly used by professional system programmers to support advanced parallel computing technologies such as OpenMP.

Java threads

The Java language supports threads as one of its essential features (Oaks and Wong, 1999). The Java library provides a `Thread` class that supports a rich collection of methods: for example, the method `start()` causes the thread to execute the method `run()`, the method `join()` waits for the thread to finish execution. The lock–unlock mechanism can be easily realized by the `synchronized` declaration. All fundamental libraries are thread-safe. These features make Java suitable for thread programming.

Listing 4

```
public class PiJavaThread {
  int n, numThreads;
  double pi = 0.0;
  synchronized void addPi(double p) {
    pi += p;
  }
  public PiJavaThread(int nd, int nt) {
    n = nd;
    numThreads = nt;
    Thread threads[] = new Thread[numThreads];
    for (int i=0; i<numThreads; i++) {
      threads[i] = new Thread(new PIworker(i));
      threads[i].start();
    }
    for (int i=0; i<numThreads; i++) {
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
  class PIworker implements Runnable {
    int myid;
    public PIworker(int id) {
      myid = id;
    }
    public void run() {
      double d, s, x;
      d = 1.0/n;
      s = 0.0;
      for (int i=myid+1; i<=n; i+=numThreads) {
        x = (i-0.5)*d;
```

```
        s += 4.0/(1.0+x*x);
      }
      addPi(d*s);
    }
  }
  public static void main(String[] args) {
    PiJavaThread piJavaThread
       = new PiJavaThread(Integer.parseInt(args[0]),
                          Integer.parseInt(args[1]));
    System.out.println(" pi = " + piJavaThread.pi);
  }
}
```

Listing 4 is an example program to calculate the value of π using the Java language. This program is almost the same as the Pthread example. As the method declaration for `addPi()` contains the keyword `synchronized`, it can be performed by only one thread; other threads must wait until the `addPi()` method of the currently executing thread finishes.

Although the Java language is designed to be thread-safe and provides several means for thread programming, it is still difficult to write efficient application programs in Java. Java's tools are generally well suited to system programming applications, such as graphical user interfaces and distributed systems, because they provide synchronization operations that are detailed and powerful, but unstructured and complex. They can be considered an assembly language for thread programming. Thus, it is not easy to use them for statistical programming.

3.3 OpenMP

OpenMP is a directive-based parallelization technique (Chandra et al., 2001) that supports fork–join parallelism and is mainly for shared memory systems. The MP in OpenMP stands for "Multi Processing". It supports Fortran (77 and 90), C and C++, and is suitable for numerical calculation, including statistical computing. It is standardized for portability by the OpenMP Architecture Review Board (OpenMP Architecture Review Board, 2004). The first Fortran specification 1.0 was released in 1997, and was updated as Fortran specification 1.1 in 1999. New features were added as Fortran specification 2.0 in 2000. Several commercial compilers support OpenMP.

We use the Fortran language for our examples in this section, because Fortran is still mainly used for high-performance computers focused on large numerical computation. Fortran is one of the oldest computer languages and has many reliable and efficient numerical libraries and compilers. The Fortran program for the simple π computation is shown in Listing 5.

We note that C (and C++) are also used for large numerical computations and are now supported to the same extent as Fortran. The following examples

can easily be replaced by C programs (except the HPF examples) but we omit them for space reasons.

Listing 5

```
      integer n, i
      double precision d, s, x, pi
      write(*,*) 'n?'
      read(*,*) n
      d = 1.0/n
      s = 0.0
      do i=1, n
        x = (i-0.5)*d
        s = s+4.0/(1.0+x*x)
      enddo
      pi = d*s
      write(*,100) pi
 100  format(' pi = ', f20.15)
      end
```

We can parallelize this program simply by using OpenMP directives (Listing 6).

Listing 6

```
      integer n, i
      double precision d, s, x, pi
      write(*,*) 'n?'
      read(*,*) n
      d = 1.0/n
      s = 0.0
!$OMP PARALLEL PRIVATE(x), SHARED(d)
!$OMP& REDUCTION(+: s)
!$OMP DO
      do i = 1, n
         x = (i-0.5)*d
         s = s+4.0/(1.0+x*x)
      end do
!$OMP END DO
!$OMP END PARALLEL
      pi = d*s
      write(*,100) pi
 100    format(' pi = ', f20.15)
      end
```

Lines started by `!$OMP` are OpenMP directives to specify parallel computing. Each OpenMP directive starts with `!$OMP`, followed by a directive and, optionally, clauses. For example, "`!$OMP PARALLEL`" and "`!$OMP END PARALLEL`" encloses a parallel region and all code lexically enclosed is executed by all threads. The number of threads is usually specified by an environmental variable `OMP_NUM_THREADS` in the shell environment. We also require a process

distribution directive "`!$OMP DO`" and "`!$OMP END DO`" to enclose a loop that is to be executed in parallel. Within a parallel region, data can either be private to each executing thread, or be shared among threads. By default, all data in static extents are shared (an exception is the loop variable of the parallel loop, which is always private). In the example, shared scope is not desirable for `x` and `s`, so we use a suitable clause to make them private: "`!$OMP PARALLEL PRIVATE (x, s)`". By default, data in dynamic extent (subroutine calls) are private (an exception is data with the `SAVE` attribute), and data in `COMMON` blocks are shared.

An OpenMP compiler will automatically translate this program into a Pthread program that can be executed by several processors on shared memory systems.

3.4 PVM

PVM (Parallel Virtual Machine) is one of the first widely used message passing programming systems. It was designed to link separate host machines to form a virtual machine, which is a single manageable computing resource (Geist et al., 1994). It is (mainly) suitable for heterogeneous distributed memory systems. The first version of PVM was written in 1989 at Oak Ridge National Laboratory, but was not released publicly. Version 2 was written at the University of Tennessee Knoxville and released in 1991. Version 3 was redesigned and released in 1993. Version 3.4 was released in 1997. The newest minor version, 3.3.4, was released in 2001 (PVM Project Members, 2004).

PVM is freely available and portable (available on Windows and several UNIX systems). It is mainly used in Fortran, C and C++, and extended to be used in many other languages, such as Tcl/Tk, Perl and Python.

The PVM system is composed of two parts: a PVM daemon program (pvmd) and libraries of PVM interface routines. Pvmd provides communication and process control between computers. One pvmd runs on each host of a virtual machine. It serves as a message router and controller, and provides a point of contact, authentication, process control and fault detection. The first pvmd (which must be started by the user) is designated the master, while the others (started by the master) are called slaves or workers.

PVM libraries such as `libpvm3.a` and `libfpvm3.a` allow a task to interface with the pvmd and other tasks. They contain functions for packing and unpacking messages, and functions to perform PVM calls by using the message functions to send service requests to the pvmd.

Example Fortran programs are in Listings 7.1 and 7.2.

Listing 7.1

```
program pimaster
include '/usr/share/pvm3/include/fpvm3.h'
integer n, i
double precision d, s, pi
integer mytid,numprocs,tids(0:32),status
```

```
      integer numt,msgtype,info
      character*8 arch
      write(*,*) 'n, numprocs?'
      read(*,*) n, numprocs
      call PVMFMYTID(mytid)
      arch = '*'
      call PVMFSPAWN('piworker',PVMDEFAULT,arch,
     $               numprocs,tids,numt)
      if( numt .lt. numprocs) then
         write(*,*) 'trouble spawning'
         call PVMFEXIT(info)
         stop
      endif
      d = 1.0/n
      msgtype = 0
      do 10 i=0, numprocs-1
        call PVMFINITSEND(PVMDEFAULT,info)
        call PVMFPACK(INTEGER4, numprocs, 1, 1, info)
        call PVMFPACK(INTEGER4, i,        1, 1, info)
        call PVMFPACK(INTEGER4, n,        1, 1, info)
        call PVMFPACK(REAL8,    d,        1, 1, info)
        call PVMFSEND(tids(i),msgtype,info)
10    continue
      s=0.0
      msgtype = 5
      do 20 i=0, numprocs-1
         call PVMFRECV(-1,msgtype,info)
         call PVMFUNPACK(REAL8,x,1,1,info)
         s = s+x
20    continue
      pi = d*s
      write(*,100) pi
100   format(' pi = ', f20.15)
      call PVMFEXIT(info)
      end
```

Listing 7.2

```
      program piworker
      include '/usr/share/pvm3/include/fpvm3.h'
      integer n, i
      double precision s, x, d
      integer mytid,myid,numprocs,msgtype,master,info
      call PVMFMYTID(mytid)
      msgtype = 0
      call PVMFRECV(-1,msgtype,info)
      call PVMFUNPACK(INTEGER4, numprocs, 1, 1, info)
      call PVMFUNPACK(INTEGER4, myid,     1, 1, info)
      call PVMFUNPACK(INTEGER4, n,        1, 1, info)
      call PVMFUNPACK(REAL8,    d,        1, 1, info)
      s = 0.0
      do 10 i = myid+1, n, numprocs
        x = (i-0.5)*d
```

```
        s = s+4.0/(1.0+x*x)
10    continue
      call PVMFINITSEND(PVMDEFAULT,info)
      call PVMFPACK(REAL8, s,1,1, info)
      call PVMFPARENT(master)
      msgtype = 5
      call PVMFSEND(master,msgtype,info)
      call PVMFEXIT(info)
      end
```

Listing 7.1 is the master program, and Listing 7.2 is the slave program, and its compiled executable file name should be `piworker`. Both programs include the Fortran PVM header file `fpvm3.h`.

The first PVM call `PVMFMYTID()` in the master program informs the pvmd of its existence and assigns a task id to the calling task.

After the program is enrolled in the virtual machine, the master program spawns slave processes by the routine `PVMFSPAWN()`. The first argument is a string containing the name of the executable file that is to be used. The fourth argument specifies the number of copies of the task to be spawned and the fifth argument is an integer array that is to contain the task ids of all tasks successfully spawned. The routine returns the number of tasks that were successfully created via the last argument.

To send a message from one task to another, a send buffer is created to hold the data. The routine `PVMFINITSEND()` creates and clears the buffer and returns a buffer identifier. The buffer must be packed with data to be sent by the routine `PVMFPACK()`. The first argument specifies the type of data to be packed. The second argument is the first item to be packed, the third is the total number of items to be packed and the fourth is the stride to use when packing. A single message can contain any number of different data types; however, we should ensure that the received message is unpacked in the same way it was originally packed by the routine `PVMFUNPACK()`. The routine `PVMFSEND()` attaches an integer label of `msgtype` and sends the contents of the send buffer to the task specified by the first argument.

After the required data have been distributed to each worker process, the master program must receive a partial sum from each of the worker processes by the `PVMFRECV()` routine. This receives a message from the task specified by the first argument with the label of the second argument and places it into the receive buffer. Note that a value of `-1` for an argument will match with any task id and/or label. The master program expects a label value of `5` on messages from the worker tasks.

The unpacking routine `PVMFUNPACK()` has the same arguments as `PVMFPACK()`. The second argument shows where the first item unpacked is to be stored.

After the sum has been computed and printed, the master task informs the PVM daemon that it is withdrawing from the virtual machine. This is done by calling the routine `PVMFEXIT()`.

The worker program uses the same PVM routines as the master program. It also uses `PVMFPARENT()` routine to find the task id of the master task that spawned the current task.

When we compile Fortran PVM codes, we must link in both the PVM Fortran library and the standard PVM library compiled for the target machine architecture. Before executing the program, the executables of the worker program should be available in a specific directory on all the slave nodes. The default authentication is performed by `rsh` call.

3.5 MPI

MPI (Message Passing Interface) is the most widely used parallel computing technique. It specifies a library for adding message passing mechanisms to existing languages such as Fortran or C. MPI is mainly used for homogeneous distributed memory systems.

MPI appeared after PVM. PVM was a research effort and did not address the full spectrum of issues: it lacked vendor support, and was not implemented at the most efficient level for a particular hardware. The MPI Forum (Message Passing Interface (MPI) Forum, 2004) was organized in 1992 with broad participation by vendors (such as IBM, Intel, SGI), portability library writers (including PVM), and users such as application scientists and library writers. MPI-1.1 was released in 1995, MPI-1.2 was released in 1997, and MPI-2 was released in 1997.

MPI-1 has several functions that were not implemented in PVM. Communicators encapsulate communication spaces for library safety. Data types reduce copying costs and permit heterogeneity. Multiple communication modes allow precise buffer management. MPI-1 has extensive collective operations for scalable global communication, and supports process topologies that permit efficient process placement and user views of process layout (Gropp et al., 1999a).

In MPI-2, other functions were added: extensions to the message passing model, dynamic process management, one-sided operations (remote memory access), parallel I/O, thread support, C++ and Fortran 90 bindings, and extended collective operations (Gropp et al., 1999b).

MPI implementations are released from both vendors and research groups. MPICH (MPICH Team, 2004) and LAM/MPI (LAM Team, 2004) are widely used free implementations.

Although MPI has more than 150 routines, many parallel programs can be written using just six routines, only two of which are non-trivial: `MPI_INIT()`, `MPI_FINALIZE()`, `MPI_COMM_SIZE()`, `MPI_COMM_RANK()`, `MPI_SEND()` and `MPI_RECV()`. An example program is shown in Listing 8.

Listing 8

```
include 'mpif.h'
integer n, i
```

```
      double precision  d, s, x, pi, temp
      integer myid, numprocs, ierr, status(3)
      integer sumtag, sizetag, master
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
      sizetag = 10
      sumtag = 17
      master = 0
      if (myid .eq. master) then
        write(*,*) 'n?'
        read(*,*) n
        do i = 1, numprocs-1
          call MPI_SEND(n,1,MPI_INTEGER,i,sizetag,
     $          MPI_COMM_WORLD,ierr)
        enddo
      else
        call MPI_RECV(n,1,MPI_INTEGER,master,sizetag,
     $        MPI_COMM_WORLD,status,ierr)
      endif
      d = 1.0/n
      s = 0.0
      do i = myid+1, n, numprocs
         x = (i-0.5)*d
         s = s+4.0/(1.0+x*x)
      enddo
      pi = d*s
      if (myid .ne. master) then
        call MPI_SEND(pi,1,MPI_DOUBLE_PRECISION,
     $        master,sumtag,MPI_COMM_WORLD,ierr)
      else
        do i = 1, numprocs-1
          call MPI_RECV(temp,1,MPI_DOUBLE_PRECISION,
     $          i,sumtag,MPI_COMM_WORLD,status,ierr)
          pi = pi+temp
        enddo
      endif
      if (myid .eq. master) then
        write(*, 100) pi
  100   format(' pi = ', f20.15)
      endif
      call MPI_FINALIZE(ierr)
      end
```

MPI follows the single program–multiple data (SPMD) parallel execution model. SPMD is a restricted version of MIMD in which all processors run the same programs, but unlike SIMD, each processor may take a different flow path in the common program.

If the example program is stored in file `prog8.f`, typical command lines for executing it are

```
f77 -o prog8 prog8.f -lmpi
```

```
mpirun -np 5 prog8
```

where the command `mpirun` starts five copies of process `prog8` simultaneously. All processes communicate via MPI routines.

The first MPI call must be `MPI_INIT()`, which initializes the message passing routines. In MPI, we can divide our tasks into groups, called communicators. `MPI_COMM_SIZE()` is used to find the number of tasks in a specified MPI communicator. In the example, we use the communicator `MPI_COMM_WORLD`, which includes all MPI processes. `MPI_COMM_RANK()` finds the rank (the name or identifier) of the tasks running the code. Each task in a communicator is assigned an identifying number from 0 to `numprocs-1`.

`MPI_SEND()` allows the passing of any kind of variable, even a large array, to any group of tasks. The first argument is the variable we want to send, the second argument is the number of elements passed. The third argument is the kind of variable, the fourth is the id number of the task to which we send the message, and the fifth is a message tag by which the receiver verifies that it receives the message it expects. Once a message is sent, we must receive it on another task. The arguments of the routine `MPI_RECV()` are similar to those of `MPI_SEND()`. When we finish with the message passing routines, we must close out the MPI routines by the call `MPI_FINALIZE()`.

In parallel computing, collective operations often appears. MPI supports useful routines for them. `MPI_BCAST` distributes data from one process to all others in a communicator. `MPI_REDUCE` combines data from all processes in a communicator and returns it to one process. In many numerical algorithms, `SEND/RECEIVE` can be replaced by `BCAST/REDUCE`, improving both simplicity and efficiency. Listing 8 can be replaced by Listing 9 (some parts of Listing 8 are omitted).

Listing 9

```
      ...
 master = 0
 if (myid .eq. master) then
   write(*,*) 'n?'
   read(*,*) n
 endif
 call MPI_BCAST(n,1,MPI_INTEGER,master,
$      MPI_COMM_WORLD,ierr)
 d = 1.0/n
 s = 0.0
      ...
 enddo
 pi = d*s
 call MPI_REDUCE(pi,temp,1,MPI_DOUBLE_PRECISION,
$      MPI_SUM,master,MPI_COMM_WORLD,ierr)
 pi = temp
 if (myid .eq. master) then
   write(*, 100) pi
     ...
```

In distributed shared memory systems, both OpenMP and MPI can be used together to use all the processors efficiently. Again, Listing 8 can be replaced by Listing 10 (the same parts of Listing 8 are omitted) to use OpenMP.

Listing 10

```
          ...
      d = 1.0/n
      s = 0.0
!$OMP PARALLEL PRIVATE(x), SHARED(d)
!$OMP& REDUCTION(+: s)
!$OMP DO
      do i = myid+1, n, numprocs
         x = (i-0.5)*d
         s = s+4.0/(1.0+x*x)
      enddo
!$OMP END DO
!$OMP END PARALLEL
      pi = d*s
      if (myid .ne. master) then
         ...
```

3.6 HPF

HPF (High Performance Fortran) is a Fortran 90 with further data parallel programming features (Koelbel et al., 1993). In data parallel programming, we specify which processor owns what data, and the owner of the data does the computation on the data (Owner-computes rule).

Fortran 90 provides many features that are well suited to data parallel programming, such as array processing syntax, new functions for array calculations, modular programming constructs and object-oriented programming features.

HPF adds additional features to enable data parallel programming. We use compiler directives to distribute data on the processors, to align arrays and to declare that a loop can be calculated in parallel without affecting the numerical results. HPF also has a loop control structure that is more flexible than DO, and new intrinsic functions for array calculations.

The High Performance Fortran Forum (HPFF) (High Performance Fortran Forum, 2004) is a coalition of industry, academic and laboratory representatives, and defined HPF 1.0 in 1993. HPF 1.1 was released in 1994 and HPF 2.0 was released in 1997. Several commercial and free HPF compilers are now available.

Listing 11 is an example program for calculating π in HPF.

Listing 11

```
      integer n, i
      double precision  d, s, pi
```

```
      double precision, dimension (:),
     $           allocatable :: x, y
!HPF$ PROCESSORS procs(4)
!HPF$ DISTRIBUTE x(CYCLIC) ONTO procs
!HPF$ ALIGN y(i) WITH x(i)
      write(*,*) 'n?'
      read(*,*) n
      allocate(x(n))
      allocate(y(n))
      d = 1.0/n
!HPF$ INDEPENDENT
      FORALL (i = 1:n)
        x(i) = (i-0.5)*d
        y(i) = 4.0/(1.0 + x(i)*x(i))
      end FORALL
      pi = d*SUM(y)
      write (*, 100) pi
 100  format(' pi = ', f20.15)
      deallocate(x)
      deallocate(y)
      end
```

`!HPF$` is used for all HPF compiler directives. We note that this is a comment to non-HPF compilers and is ignored by them. The `PROCESSORS` directive specifies the shape of the grid of abstract processors. Another example "`!HPF$ PROCESSORS exprocs(6,2)`" specifies a 6×2 array of 12 abstract processors labelled `exprocs`.

The `DISTRIBUTE` directive partitions an array by specifying a regular distribution pattern for each dimension `ONTO` the arrangement of abstract processors. The `CYCLIC` pattern spreads the elements one per processor, wrapping around when it runs out of processors, i.e., this pattern distributes the data in the same way that the program in Listing 8 performs. Another pattern is `BLOCK`, which breaks the array into equal-sized blocks, one per processor. The rank of the abstract processor grid must be equal to the number of distributed axes of the array.

The `ALIGN` directive is used to specify relationships between data objects. In the example program, elements of x and y that have the same index are placed on the same processor.

The `INDEPENDENT` directive informs the compiler that in the execution of the `FORALL` construct or the do loop, no iteration affects any other iteration in any way.

The `FORALL` statement is a data parallel construct that defines the assignment of multiple elements in an array but does not restrict the order of assignment to individual elements. Note that the do loop executes on each element in a rigidly defined order.

The `SUM` intrinsic function performs reduction on whole arrays.

We may compare HPF with OpenMP, because both systems use compiler directives in a standard language (Fortran) syntax. In OpenMP, the user specifies the distribution of iterations, while in HPF, the user specifies the distribution of data. In other words, OpenMP adopts the instruction parallel programming model while HPF adopts data parallel programming model. OpenMP is suitable for shared memory systems whereas HPF is suitable for distributed memory systems.

4 Parallel computing in statistics

4.1 Parallel applications in statistical computing

The most important thing in parallel computing is to divide a job into small tasks for parallel execution. We call the amount of independent parallel processing that can occur before requiring some sort of communication or synchronization the "granularity". Fine granularity may allow only a few arithmetic operations between processing one message and the next, whereas coarse granularity may allow millions. Although the parallel computing techniques described above can support programming of any granularity, coarse granularity is preferable for many statistical tasks. Fine granularity requires much information exchange among processors and it is difficult to write the required programs. Fortunately, many statistical tasks are easily divided into coarse granular tasks. Some of them are embarrassingly parallel.

In data analysis, we often wish to perform the same statistical calculations on many data sets. Each calculation for a data set is performed independently from other data sets, so the calculations can be performed simultaneously. For example, Hegland et al. (1999) implemented the backfitting algorithm to estimate a generalized additive model for a large data set by dividing it into small data sets, fitting a function in parallel and merging them together. Beddo (2002) performed parallel multiple correspondence analysis by dividing an original data set and merging their calculation results.

Another embarrassingly parallel example is a simulation or a resampling computation, which generates new data sets by using a random number generating mechanism based on a given data set or parameters. We calculate some statistics for those data sets, repeat such operations many times and summarize their results to show empirical distribution characteristics of the statistics. In this case, all calculations are performed simultaneously except the last part. Beddo (2002) provided an example of bootstrapping from parallel multiple correspondence analysis.

We must be careful that random numbers are appropriately generated in parallel execution. For example, random seeds for each process should all be different values, at least. SPRNG (Mascagni, 1999) is a useful random number generator for parallel programming. It allows for the dynamic creation

of independent random number streams on parallel machines without interprocessor communication. It is available in the MPI environment and the macro `SIMPLE_SPRNG` should be defined to invoke the simple interface. Then the macro `USE_MPI` is defined to instruct SPRNG to make MPI calls during initialization. Fortran users should include the header file `sprng_f.h` and call `sprng()` to obtain a double precision random number in $(0, 1)$. In compiling, the libraries `liblcg.a` and the MPI library should be linked.

The maximum likelihood method requires much computation and can be parallelized. Jones et al. (1999) describes a parallel implementation of the maximum likelihood estimation using the EM algorithm for positron emission tomography image reconstruction. Swann (2002) showed maximum likelihood estimation for a simple econometric problem with Fortran code and a full explanation of MPI. Malard (2002) solved a restricted maximum likelihood estimation of variance-covariance matrix by using freely available toolkits: the portable extensible toolkit for scientific computation (PETSc) and the toolkit for advanced optimazation (TAO) (Balay et al., 2001) which are built on MPI.

Optimization with dynamic programming requires much computation and is suitable for parallel computing. Hardwick et al. (1999) used this technique to solve sequential allocation problems involving three Bernoulli populations. Christofides et al. (1999) applied it to the problem of discretizing multidimensional probability functions.

Racine (2002) demonstrated that kernel density estimation is also calculated efficiently in parallel.

4.2 Parallel software for statistics

Several commercial and non-commercial parallel linear algebra packages that are useful for statistical computation are available for Fortran and/or C. We mention two non-commercial packages with freely available source codes: ScaLAPACK (Blackford et al., 1997) supports MPI and PVM, and PLAPACK (van de Geijin, 1997) supports MPI. Murphy et al. (1999) described the work to transfer sequential libraries (Gram-Schmidt orthogonalization and linear least squares with equally constraints) to parallel systems by using Fortran with MPI.

Although we have many statistical software products, few of them have parallel features. Parallel statistical systems are still at the research stage. Bull et al. (1999) ported a multilevel modeling package MLn into a shared memory system by using C++ with threads. Yamamoto and Nakano (2002) explained a system for time series analysis that has functions to use several computers via Tkpvm, an implementation of PVM in the Tcl/Tk language.

The statistical systems R (The R Development Core Team, 2004) and S (Chambers, 1998) have some projects to add parallel computing features. Temple Lang (1997) added thread functions to S. PVM and MPI are directly available from R via the rpvm (Li and Rossini, 2001) and Rmpi (Yu, 2002) packages. They are used to realize the package "snow" (Rossini et al., 2003),

which implements simple commands for using a workstation cluster for embarrassingly parallel computations in R. A simple example session is:

```
> cl <- makeCluster(2, type = "PVM")
> clusterSetupSPRNG(cl)
> clusterCall(cl, runif, 3)
[[1]]
[1] 0.749391854 0.007316102 0.152742874

[[2]]
[1] 0.8424790 0.8896625 0.2256776
```

where a PVM cluster of two computers is started by the first command and the SPRNG library is prepared by the second command. Three uniform random numbers are generated on each computer and the results are printed by the third command.

The statistical system "Jasp" (Nakano et al., 2000) is implementing experimental parallel computing functions via network functions of the Java language (see also `http://jasp.ism.ac.jp/`).

References

Amdahl, G. M. (1967). Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483 – 485.

Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M., McInnes, L. C., Smith, B. F., and Zhang, H. (2001). PETSc home page. `http://www.mcs.anl.gov/petsc`.

Beddo, V. (2002). *Applications of parallel programming in Statistics*. Ph.D. dissertation, University of California, Los Angeles. `http://theses.stat.ucla.edu/19/parallel_programming_beddo.pdf`.

Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. (1997). *ScaLAPACK Users' Guide*. SIAM Press.

Bull, J. M., Riley, G. D., Rasbash, J., and Goldstein, H. (1999). Parallel implementation of a multilevel modelling package. *Computational Statistics & Data Analysis*, 31(4):457 – 474.

Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison Wesley.

Chambers, J. M. (1998). *Programming with Data: A Guide to the S Language*. Springer-Verlag.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel Programming in OpenMP*. Morgan Kaufman.

Christofides, A., Tanyi, B., Christofides, D., Whobrey, D., and Christofides, N. (1999). The optimal discretization of probability density functions. *Computational Statistics & Data Analysis*, 31(4):475 – 486.

Flynn, M. (1966). Very high-speed computing systems. *Proc. IEEE*, 54(12):1901 – 1909.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. S. (1994). *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press.

Gropp, W., Lusk, E., and Skjellum, A. (1999a). *Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd Edition*. MIT Press.

Gropp, W., Lusk, E., and Thakur, R. (1999b). *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Comm. ACM*, 31(5):532 – 533.

Hardwick, J., Oehmke, R., and Stout, Q. F. (1999). A program for sequential allocation of three bermoulli populations. *Computational Statistics & Data Analysis*, 31(4):397 – 416.

Hegland, M., McIntosh, I., and Turlach, B. A. (1999). A parallel solver for generalized additive models. *Computational Statistics & Data Analysis*, 31(4):377 – 396.

High Performance Fortran Forum (2004). HPF: The high performance fortran home page. `http://www.crpc.rice.edu/HPFF/`.

Jones, H., Mitra, G., Parkinson, D., and Spinks, T. (1999). A parallel implementation of the maximum likelihood method in positron emission tomography image reconstruction. *Computational Statistics & Data Analysis*, 31(4):417 – 439.

Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Steele, J. G. L., and Zosel, M. E. (1993). *The High Performance Fortran Handbook*. MIT Press.

LAM Team (2004). LAM/MPI parallel computing. `http://www.lam-mpi.org/`.

Li, N. and Rossini, A. (2001). RPVM: Cluster statistical computing in R. *R News*, 1(3):4 – 7. `http://CRAN.R-project.org/doc/Rnews/`.

Malard, J. M. (2002). Parallel restricted maximum likelihood estimation for linear models with a dense exogenous matrix. *Parallel Computing*, 28(2):343 – 353.

Mascagni, M. (1999). SPRNG: A scalable library for pseudorandom number generation. In Spanier, J. et al., editor, *Proceedings of the Third International Conference on Monte Carlo and Quasi Monte Carlo Methods in Scientific Computing*. Springer Verlag.

Message Passing Interface (MPI) Forum (2004). Message passing interface (MPI) forum home page. `http://www.mpi-forum.org/`.

MPICH Team (2004). MPICH - A portable mpi implementation. `http://www-unix.mcs.anl.gov/mpi/mpich/`.

Murphy, K., Clint, M., and Perrott, R. H. (1999). Re-engineering statistical software for efficient parallel execution. *Computational Statistics & Data Analysis*, 31(4):441 – 456.

Nakano, J., Fujiwara, T., Yamamoto, Y., and Kobayashi, I. (2000). A statistical package based on Pnuts. In Bethlehem, J. G. and van der Heijden, P.

G. M., editors, *COMPSTAT 2000 Proceedings in Computational Statistics*, pages 361 – 366. Physica-Verlag.

Oaks, S. and Wong, H. (1999). *Java Threads, 2nd edition*. O'Reilly.

OpenMP Architecture Review Board (2004). OpenMP: Simple, portable, scalable SMP programming. `http://www.openmp.org/`.

PVM Project Members (2004). PVM: Parallel virtual machine. `http://www.csm.ornl.gov/pvm/pvm_home.html`.

Racine, J. (2002). Parallel distributed kernel estimation. *Computational Statistics & Data Analysis*, 40(2):293 – 302.

Rossini, A., Tierney, L., and Li, N. (2003). Simple parallel statistical computing in R. UW Biostatistics working paper series, Paper 193, University of Washington. `http://www.bepress.com/uwbiostat/paper193`.

Schervish, M. J. (1988). Applications of parallel computation to statistical inference. *J. Aner. Statist. Assoc.*, 83:976 – 983.

Sterling, T., Salmon, J., Becker, D. J., and Savarese, D. F. (1999). *How to Build a Beowulf : A Guide to the Implementation and Application of PC Clusters*. MIT Press.

Swann, C. A. (2002). Maximum likelihood estimation using parallel computing: An introduction to MPI. *Computational Economics*, 19:145 – 178.

Tanenbaum, A. S. (2001). *Modern Operating Systems, 2nd Edition*. Prentice Hall.

Temple Lang, D. (1997). *A multi-threaded extension to a high level interactive statistical computing environment*. Ph.D. dissertation, University of California, Berkeley. `http://cm.bell-labs.com/stat/doc/multi-threaded-S.ps`.

The R Development Core Team (2004). The R project for statistical computing. `http://www.r-project.org/`.

Tuomi, I. (2002). The lives and death of moore's law. *First Monday*, 7(11). `http://firstmonday.org/issues/issue7_11/tuomi/index.html`.

van de Geijin, R. A. (1997). *Using PLAPACK*. MIT Press.

Yamamoto, Y. and Nakano, J. (2002). Distributed processing functions of a time series analysis system. *Journal of the Japanese Society of Computational Statistics*, 15(1):65 – 77.

Yu, H. (2002). Rmpi: Parallel statistical computing in R. *R News*, 2(2):10 – 14. `http://CRAN.R-project.org/doc/Rnews/`.

Index