Annals of the University of Craiova, Mathematics and Computer Science Series
Volume 38(4), 2011, Pages 12–23
ISSN: 1223-6934, Online 2246-9958

Study on Parallel Merging of Arrays using PVM

Paraschiva Popovici

Abstract. The algorithm for merging is inspired by the method proposed in [1]. In the algorithm we partition the two arrays in subpartitions, and we sent each subpartition (Ai,Bi) to be merged by the child processes created on other hosts of the virtual machine. Then with the merged subpartitions received from the child processes we directly assemble in the parent process the merge of the two primary sequences. For partitioning there the following alternatives:

(1) We compute the partitions in the master process and we send only the two partitions for merging to a slave process. In this case the disadvantage is that the master performs a greater effort to compute the partitions but sends less data to the children (sends only the partitions).
(2) We send at least one of the sets in its entirety to the child for computing the partitioning. In this case more data is sent but the parent does not have to compute

In the implementation of the algorithm we chose the first alternative, because in a real application the chosen variant varies in each case with the speed of the processors or the speed of data transmission

2010 Mathematics Subject Classification. Primary 65Y05; Secondary 68W10.
Key words and phrases. parallel calculation, merging.

1. Solving Algorithm

The merging algorithm is inspired according to the proposed method in [1]. The processes' architecture is a star one meaning that there is a master process and more other sons processes that are rolling in parallel on the virtual machine's hosts.

Received January 10, 2011. Revision received September 18, 2011.

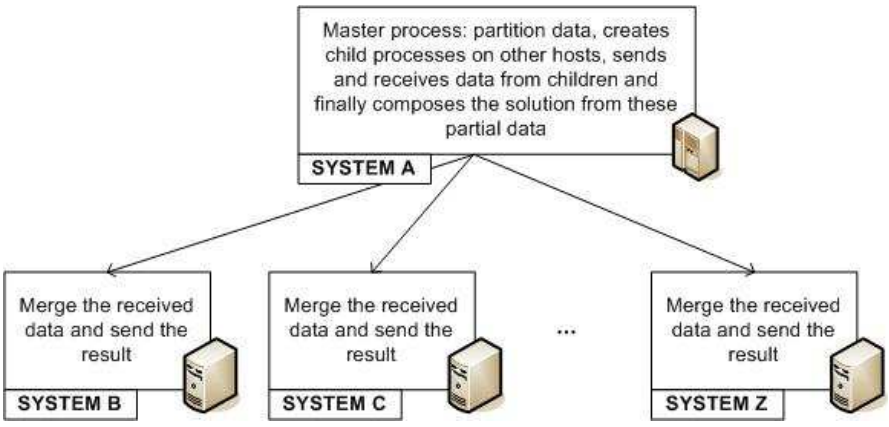

Figure 1. The processes architecture

2. Topics

The Annals of the University of Craiova - Mathematics and Computer Science series - publishes original research papers in any branch of Mathematics and Computer Science. The publishing policy of AMI journal encourages particularly the publishing of scientific papers that are focused on the convergence of the mathematics and computer science.

In case of merging problem this is an appropriate architecture but if we have to implement sorting through merging in parallel mode then we have to use an arborescent architecture of the processes in which there is a 0 level process that partition the list into 2 equal subpartitions and then transmits them to the son processes. The sons partition also the received list and send it to other processes that they create obtaining this way an arborescent processes structure in which the result turns overhand through merging the intermediary results on each process level.

We present below the problem and the solving method: Let us consider $A = (a_1, a_2, ..., a_m)$ and $B = (b_1, b_2, ..., b_n)$ two sorted sequences. It is requested to determine the ordered sequence $C = (c_1, c_2, ..., c_{n+m})$ which contains all the elements of the two ordered sequences.

3. Sequential Algorithm

A and B are partitioned on subsequences pairs so that to obtain the ordered sequence through pairs merging.

4. Parallel Algorithm

More mergings are being simultanely created. It's being noted with grade $(a_i : A)$ the number of A's components, smaller or equal with a_i. The grade can be determined through binary or parallel searching. Taking for example the case in which n (it's not prime number) has a k divider. Then we are searching k sequences pairs (A_i, B_i) of A and B so that:

a) the number of B_i elements is n/k

b) each element of $A_i \cup B_i$ is bigger than any of its elements:

$$A_{i-1} \cup B_{i-1} \ for \ 1 \leq i \leq k-1$$

One solution is partitioning:

$$B_i = (b_{i*n/k+1}, ..., b_{(i+1)*n/k}), Ai = (a_{j(i)+1}, ..., a_{j(i+1)})$$

where

$$j(i) = grade(b_{i*n/k} : A)$$

As it can be observed from b) property, if we merge each subpartitions pair, then we can directly compose the merging of the two primary A and B sequences.

This is the reason why we have been partitioned inside the algorithm the two arrays in subpartitions and each partition (Ai, Bi) was submitted to be "merged" by the sons processes created on other hosts of the virtual machine. Then with the merged subpartitions received from the sons processes we directly assemble in the parent process the merging of the two primary sequences. In our case for partitioning there are two options:

(1) We calculate the partitions in the master process (owner in this current case) and then we submit only the two partitions for merging a slave process (son in the current case). In this situation the disadvantage is that the master has more work to do in order to calculate the partitions but sends less data tpwards the son processes (sends only the partitions).
(2) We send at least one of the data sets completely toward the son process, in order to calculate its partition. In this case more data are being sent but the father is no longer busy with the calculation.

In a real application the chosen option depends from case to case by the speed of the processors or the data transmission speed. In the implementation of the algorithm we have chosen option 1.

```
#include <stdio.h>
#include <conio.h>
#include "pvm3.h"
/* The maximum length of a sequence.*/
#define LMAX 100
/* The exact process number is equal to the partitions number.*/
#define MAX_PROCESSES 50
/* The mark is being used as a signature for the sent messages so that
    the one who tries to read the message must provide the same mark.
 */
#define MARK 1
/* Merges the two sorted sublists into one single list.  */
void merge(int a[], int left, int center, int right) {
    int i,j,k,b[2*LMAX];
    i=left;
    j=center+1;
    k=0;
    while ((i<=center) && (j<=right)) {
      if (a[i] <= a[j]) {
         b[k] = a[i];
         i++;
      }
      else {
         b[k] = a[j];
         j++;
      }
    k++;
    }
    /* if there are other elements in one of the two sublists,
    these are being directly copied because they are already ordered
    */
    while (i<=center) {
         b[k] = a[i];
         i++;
         k++;
    }
    while (j<=right) {
         b[k] = a[j];
```

```
         j++;
         k++;
    }
    /* then the unordered list is being over written with the ordered
       one from the b panel
    */
    for (k=left; k<=right; k++)
         a[k] = b[k-left];
}
void mergesort(int a[], int left, int right) {
      int center;
      if(left<right) {
         center = (left + right)/2; //este un intreg
         mergesort(a, left, center);
         mergesort(a, center +1, right);
         merge(a, left, center, right);
      }
}
int grade (int a[], int m, int e) {
     int i, grade = 0;
     for (i=1;i<=m;i++)
         if (a[i]<=e)
             grade++;
     return grade;
}
void displays(int a[], int m, char t[20], int k = 1) {
     int i;
     for (i=k;i<=m-(1-k);i++)
         printf("\n%s[%d]=%d",t,i,a[i]);
}
int is_Prime(int n) {
    for (int i = 2;i<=n/2;i++)
       if ((n % i) == 0)
          return 0;
       return 1;
}
int finds_the_Maximum_Divider(int n) {
     int divMax = 2, sup;
     //the upper limit till where the search is being made
     if (MAX_PROCESSES>(n/2))
       sup = n/2;
       else sup = MAX_PROCESSES;
     for (int i = 2;i<=sup;i++)
        if ((n % i) == 0) divMax = i;
     return divMax;
}
void partition (int a[], int m, int b[], int n, int partition A[],
                        int partition B[], int noProcesses) {
      int i, sup;
```

```
      /* CALCULATION OF PARTITIONS
         Partition of B.
      */
      for (i=1;i<=nrProcesses;i++) {
         // the maximum index/factor of each subpartition
         Partition B[i]=n/noProcesses*i;
      }
      /* Partition of A.
         It's being performed according to the partition of B
      */
      // adding a fictive element (one of the subpartitions is void)
      PartitionA[0]=1;
      for (i=1;i<=noProcesses;i++){
     //printf("\n-- %d -- %d ",partition B[i],b[partition of B[i]]);
         sup = grade(a,m,b[partitionB[i]]);
         // if the subpartition from the other set is void then
         // index/factor is kept
         if (sup == 0)
            partitionA[i] = partitionA[i-1];
         else
            partitionareaA[i]= sup;
      }
      /* in case there are elements in A bigger than the elements
         from B the last index/factor limits the partition
      */
      partitionA[noProcesses]=m;
}

void main() {
    int a[LMAX],b[LMAX],merging[LMAX*2],n,m,i,j,k,info,lung1,lung2;
    int idProcess, idHost, noProcesses, np, idProcesses[MAX_PROCESSES];
    /* The partitions are indicated by some indexes/factors that
       represents the beginning and the end of each partition
    */
    int partition A[MAX_PROCESSES], partition B[MAX_PROCESSES];
    // We read the two full arrays as input data
    while (1) {
       printf("Number m of elements of the first array:");
    scanf("%d",&m);
    printf("Number n of elements of the second array:");
    scanf("%d",&n);
 if ((m<=1)||(n<=1)||(m>LMAX)||(n>LMAX)||(isPrime(n) && isPrime(m))){
          /* It can be considered also an additional pseudo-element
             in order to eliminate the condition not to be prime,
             but this increases the difficulty of the algorithm.
          */
          printf("\nError: n and m must be natural numbers >1") ;
  printf(" and <%d at least one must not be a prime number!\n",LMAX);
          continue;
```

```
   }
  // All the elements starting with position 1 from the board/panel
      for (i=1;i<=m;i++)
          printf("a[%d]=",i); scanf("%d",&a[i]);
      for (i=1;i<=n;i++)
          printf("b[%d]=",i); scanf("%d",&b[i]);
      break;
   }
   /* We ensure that the data from the two input arrays are sorted out
      (this for guarantee the correcteness of merging)
   */
    mergesort(a,1,m);
    mergesort(b,1,n);
    display(a,m,"a");
    display(b,n,"b");

    // Determining the necessary processes number
    if (! isPrime(n)){
       printf("\n It's being partitioned after the second array (b)");
       noProcesses = finds Maximum Divider(n);
       partition(a,m,b,n,partitionA,partitionB,noProcesses);
    }
    else {
       printf("\n it's being partitioned after the first array (a)");
       noProcesses = finds Maximum Divider(m);
       partition(b,n,a,m,partitionA,partitionB,noProcesses);
    }
    displays(partitionA,noProcesses,"partitionA");
    diaplays(partitionB,noProcesses,"partitionB");
    idProcess = pvm_mytid();
    // Ids of the hosts are on "DTID" column on hexadecimal.
    idHost = pvm_tidtohost(idProcess);
    //we verify if there are any errors
    if ((idProcess < 0) || (idHost < 0)) {
        //display the error
        printf("Error at obtaining the process identifiers");
        exit(1);
    }
    printf("Process with id %d are rolling on the host with the id",
           idProcess,idHost);
np = pvm_spawn("merging", (char**)0, 0, "", noProcesses, idProcesses);
    for (i=0;i<noProcesses;i++)
        if (idProcesses[i]<0){
           // error in decimal
           printf("Error (id process < 0): %d",idProcesses[i]);
           pvm_exit();
           exit(1);
        }
 // the process creation was not succedded or there have been created
```

```
    // less processes than needed
    if ((np == 0) || (np != noProcesses)) {
        pvm_exit();
        exit(1);
    }
    printf("\n we have created  %d processes",np);
    dispaly(idProcesses,noProcesses,"idProcesses",0);

 /* We need auxiliary elements for determine the length of the first
      subpartition
    */
   partition A[0] = 0;
   partitionB[0] = 0;
   // we send the data to each process
   for (i=1;i<=noProcesses;i++) {
       info = pvm_initsend(PvmDataDefault);
       if (info < 0) {
          printf("Error when call initsend!");
          pvm_exit();
          exit(1);
       }
       printf("\n Send to the process %d",i);
       // Send subpartition i from A.
       length1 = partitionA[i]-partitionA[i-1];
       info = pvm_pkint(&length1,1,1);
   //printf("\n compress length= %d,%d,%d",length1,partitionA[i],
                    partitionA[i-1]);
       if (info < 0) {
          printf("Error when compressing the length of A!");
          pvm_exit();
          exit(1);
       }
       // first we set the index, then we transmit the elements
        k = partitionA[i-1]+1;
        for (j=k;j<=partitionA[i];j++) {
            //printf("\n Compressing a[%d]=%d",j,a[j]);
            info = pvm_pkint(&a[j],1,1);
            if (info < 0) {
               printf("Error when compressing the length of A!");
               pvm_exit();
               exit(1);
            }
        }
       /* Sending subpartition i from B. */
       Length 2 = partitionB[i]-partitionB[i-1];
       info = pvm_pkint(&length2,1,1);
   //printf("\n Compressing length =%d,%d,%d",length2,partitionB[i],
                    partitionB[i-1]);
       if (info < 0) {
```

```
            printf("Error when compressing the length of B!");
            pvm_exit();
            exit(1);
         }
         // first we set the index from which we start
         // then we transmit the elements
         k = partitionB[i-1]+1;
         for (j=k;j<=partitionB[i];j++) {
             //printf("\n Compressing b[%d]=%d",j,b[j]);
             info = pvm_pkint(&b[j],1,1);
             if (info < 0) {
                printf("Error when compressing the length of B!");
                pvm_exit();
                exit(1);
             }
         }
        /* now we execute the submission of data:
           the index of the processes' panel/board starts from 0 => i-1
        */
        info = pvm_send(idProcesses[i-1],MARK);
        if (info < 0) {
           printf("Error when sending the data!");
           pvm_exit();
           exit(1);
        }
    }
    printf("\n Sent all data.");
    /* We receive the intermediary results (meaning the merged
     subpartitions) from each process. We will use k to indicate
     the current position from the resulted array.
    */
printf("\n Receiving the subpartitions mergings from each process.");
    k = 0;
    for (i=1;i<=noProcesses;i++) {
        //printf("\n ---- From the process %d",i);
        info = pvm_recv(idProcesses[i-1],MARK);
        if (info < 0) {
           printf(" can not receive the data!");
           pvm_exit();
           exit(1);
        }
    /* No longer waiting to receive the length of the sequence because
      I know that it represents the sum of the number of elements of
      the two sent subpartitions.
        */
        length1 = partitionA[i]-partitionA[i-1];
        length2 = partitionB[i]-partitionB[i-1];
        for (j=1;j<=(length1+length2);j++) {
            //printf("\n trying another reception");
```

```
            info = pvm_upkint(&merging[k+j],1,1);
            if (info < 0) {
               printf(" I can not receive the data!");
               pvm_exit();
               exit(1);
            }
            //printf("\n received %d",merging[k+j]);
        }
        k+=length1+length2;
    }
    pvm_exit();
    // result
    display(merging, m+n,"Merging");  getch();
}
```

We formulate the following observations regarding the Algorithm presented in this section:

(1) The partitions number equals to a k divider of the elements' number of one of the sequences. This way the `MAX_PROCESSES` constant will represent the maximum divider that we will choose for that sequence. We will try to find this way the greatest divider, smaller or equal to `MAX_PROCESSES`. The `MAX_PROCESSES` constant depends on practice on the number of hosts that compose the virtual machine and on the problem itself (for example the load -load balancing, how big the data flow is, the speed of the data transmission ways between the processes- the more processes we have the faster the data are being spread, the speed of the nodes etc).

(2) The merge function uses at ordering the input data for ensuring the correcteness of the parallel merging.
* parameter `a` - the panel screen with the sublist that has to be sorted out
* parameter `left` - left index/factor of the sublist (the begining)
* parameter `right` - right index/factor of the sublist (the end)
* parameter `center` - is the index/factor that sets out the two sublists

The merging process continues as long as we did not get to the end of neither of the two sublists. The elements from correspondent positions are being compared and are being ordered in the auxiliary panel `b`, increasing the indexes/factors according to the element that it's being copied.

(3) The `mergesort` routine divides the unsorted list in two lists with approximately same length. Then we divide the two sublists recursively until we obtain lists of length 1, in this last case actually the list itself being turned upside down ([6]). Pay attention still because the operations are being performed on the same list, but the indexes/factors will indicate the beginning/end of the sublists.
* parameter `a` - the screen panel with the sublist that has to be sorted out
* parameter `left` - left index/factor of the sublist (the beginning)
* parameter `right` - right index/factor of the sublist (the end)

(4) The `grade` function calculates the grade of an element in a panel:
* parameter `a` - the panel/board
* parameter `m` - the number of elements of the panel/board
* parameter `e` - the element whose grade it's being calculated

This function returns the grade of `e` element in the `a` list (the number of elements from `a` <= `e`)

(5) The function `display` print the elements of a panel/board:
* parameter `a` - the panel/board
* parameter `m` - the number of elements of the panel/board
* parameter `t` - character that it's being used for the name of the panel/board
* parameter `k` - specifies the position from which the display starts

(6) The function `is_Prime` determines if a number is a prime number or not
* parameter `n` - the number that is being tested to see if it's a prime number or not
* the function returns 1 if `n` is a prime number and returns 0 if `n` it's not a prime number

(7) The function `finds_the_Maximum_Divider` determines the maximum divider of a number smaller than MAX_PROCESSES
* parameter `n` - the number
* the function returns the maximum divider smaller than MAX_PROCESSES.

(8) The function `partition` partitions the two sequences
* parameter `a` - the panel/board with the first sequence
* parameter `m` - number of elements from the first sequence
* parameter `b` - the panel/board with the first sequence
* parameter `n` - number of elements from the second sequence
* parameter `partitionA` - represents the partition of the first sequence (remember the indexes/factors that indicate the upper boundary of each subpartition)
* parameter `partitionB` - represents the partition of the second sequences (remember the indexes/factors that indicate the upper boundary of each subpartition)
* parameter `no`Processes - represents the number of processes (task-uri) that will be created

(9) Function `main` is responsible for processes initialization which will implement the algorithm and the transmission of data to them. The architecture of the processes is a star one or master-slave (there is no communication between the son processes). Here the final result will be also obtained.
The subpartitions sending process takes place as follows: the first compressed parameter will be the length of each subpartition. Then the elements of the subpartition follows. In the created processes the reading of the elements will be stopped based on this first element submitted.

(10) Function `pvm_spawn` starts `noProcesses` (tasks) that are independently allocated on the virtual machine by the PVM daemon.
* the first parameter represents the executable "interclasare.exe" that must exist in the PVM\bin\win32 director on each of the virtual machine's systems. This program executes the merging of each subpartition pairs and communicates the merged sequence.
* the second parameter is a pointer to a panel/boards of arguments (with the last element NULL) to be transmitted to the process.
* the third parameter is a flag that establishes or not establishes a host where the process to be created(in the current case through 0 (PvmTaskDefault)
* we launched the PVM deamon to decide where the processes will be created. The forth parameter would indicate the host if we have been chosen through flag to specify the hosts.

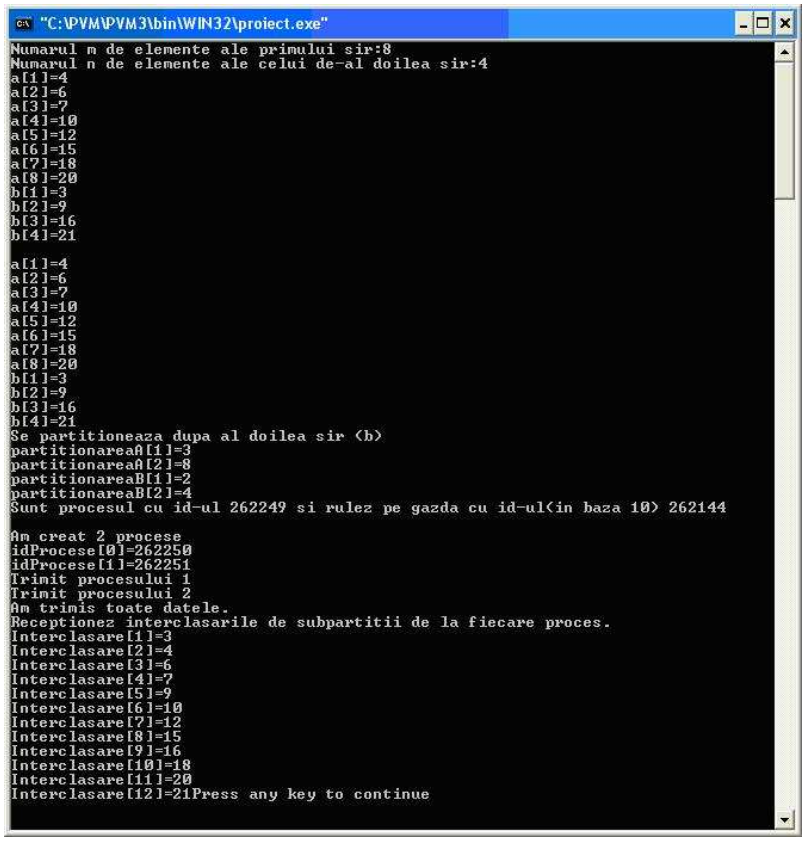

FIGURE 2. The resulted output

* The last but one parameter represents the number of processes that has to be created and the last one is a panel that will remember the id's of the processes.

5. Examples

Being $A = (4, 6, 7, 10, 12, 15, 18, 20)$, $B = (3, 9, 16, 21)$. Then $n = 4$, $k = 2$. As $grade(9 : A) = 3$ the partition is being realised as follows: $A_1 = (4, 6, 7)$, $B_1 = (3, 9)$ and $A_2 = (10, 12, 15, 18, 20)$, $B_2 = (16, 21)$ that are then concurrently merged by the son processes.

6. Conclusion

The most important implementation problems that we face are related to the correct partition of the two arrays and also at their reception by the *master*. The order for transmitting the data toward a process must correspond to the order in which the reception is made in the destination process, otherwise the results are being imprevisible. In Fig. 2 we have presented the rolling modality for the given example.

References

[1] D. Petcu, *Parallel Algorithms*, Timisoara University Typography, 1994.
[2] P. Popovici, *Liniar and Arborescent Data Structures*, Eurostampa Publishing House, 2009.
[3] http://www.netlib.org/pvm3/win32/
[4] http://www.netlib.org/pvm3/book/
[5] http://wwwcs.uni-paderborn.de/pc2/fileadmin/services/pvm_book.pdf
[6] http://en.wikipedia.org/wiki/PVM

(Paraschiva Popovici) Universitatea de Vest din Timisoara, B-dul Prvan Nr.4, Tel. 0256/592219, Romania
E-mail address: popovici@info.uvt.ro