Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 105-124

PARALLEL ELEMENTWISE PROCESSING – A NOVEL VERSION

Á. Fóthi, Z. Horváth and T. Kozsik
(Budapest, Hungary)

Abstract. In this paper a parallel algorithm for computing the values of elementwise processable functions will be presented. The main task is to prepare a dense total disjoint decomposition of the input. A relational model of sequential programs and its extension to parallel programs will be used to formalize this algorithm. By this way a programming theorem is produced. We investigate the computational and communication costs when implemented on a distributed system.

1. Introduction

Elementwise processable functions form a class of many often-used operations such as merging ordered sequences, computing the union of sets and updating a database. Elementwise processability means that the output can be obtained by processing each single data item from the input one after the other [1, 6]. A formal definition will be introduced in Section 2. In general the domain and the image of such functions consist of ordered sequences (possibly represented by sequantial or direct files) or sets. Throughout this paper we assume that the domain is some ordered sequences while the image is some sets. (See [6] and Section 2 for details on sequences.)

To describe our algorithm we use a relational model of sequential programs [3, 4, 5] and its extension to parallel ones [2]. This latter model is closely related to UNITY introduced in [8], but is completed with some important features of the former. The two models formalize the notions of state space, problem, program, solution, specification, refinement, etc.

Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T017800.

These models are based on the top-down refinement of specifications. As the starting point of the program design we produce a formal specification to the problem. The proof of the correctness of the solution is developped parallel to the step-by-step refinement of this specification. In the end an abstract program is obtained. We say that the specification and the solution is a programming theorem.

Section 2 enumerates some notations, introduces some basic definitions and the formal specification to the problem. A sequential solution can be found in Section 3, while Section 4 discusses the possibilities of parallelism. In Section 5 we present a parallel algorithm which is the main goal of this paper. We give a formal description of this algorithm in Section 6. Finally we draw the conclusions in Section 7.

2. Formal approach

In the following we employ the terminology used also in [1, 2, 4, 6]. The names of types and components of the state space are given in capital letters. Variables are usually named after their types, but they are in small letters. The 'variables' of the parameter space are denoted by the corresponding variable with a $'$ sign, such as x'. We use P, Q, R as the name of predicates.

Let us denote the abstract data type *sequence* by $seq(T)$, where T is the base type of the sequence. The methods of $seq(T)$, viz. $lob(s)$, $hib(s)$, $lov(s)$, $hiv(s)$, $dom(s)$, $lorem(s)$, $hirem(s)$, $loext(s,t)$ and $hiext(s,t)$ are defined in [6]. In addition we will use the following notations:

range(s): the interval $[lob(s)..hib(s)]$

s(i): the element of s which is indexed by i (the $(i - lob(s))$-th element of s)

s(i..j): a subsequence of s

s ∘ z: concatenation of the two sequences

The $\ll\gg$ denotes the empty sequence, $\ll a, b \gg$ the one with two elements, a and b. As it can be seen above, we write $s(i)$ when refering to one element of the sequence s. For indexing *vectors* (which is another abstract data type presented in [6]) we use square brackets (e.g. $v[i]$). The notation x_i means the ith component of a *direct product* x. With a direct product we are usually willing to note variables stored on different processors in a distributed system. In contrary we apply the notation f_i in Section 5 for the sake of brevity.

2.1. Elementwise processability

In the following we present a generalized definition of total disjoint decomposition and a type transformed version of elementwise processable functions and elementwise processing (cf. [6]). Let H denote a set with a complete ordering '<' among its elements, $S = seq(H)$ and $P = \mathcal{P}(H)$. Let $X = S^k$ (a direct product with k components), $Y = P^l$, where $k, l \in \mathcal{N}$.

Definition 2.1. Total disjoint decomposition: *Let $x \in X$. We call $x^{(1)}, x^{(2)}, \ldots, x^{(r)}$ a total disjoint decomposition of x, abbreviated by $cdd(x, (x^{(1)}, \ldots, x^{(r)}))$ iff for all different blocks ($i \neq j \in [1..r]$) and for all components $u, v \in [1..k]$ it holds, that*

$$(1) \qquad \forall s \in range(x_u^{(i)}) : \forall t \in range(x_v^{(j)}) : x_u^{(i)}(s) \neq x_v^{(j)}(t),$$

$$(2) \qquad \exists i \in Perm(1, 2, \ldots, r) : \forall u \in [1..k] : x_u = x_u^{(i(1))} \circ x_u^{(i(2))} \circ \ldots \circ x_u^{(i(r))}.$$

Remark 2.1. *Rule (1) can be replaced by the following: $\exists$ $H_1, H_2, \ldots, H_r$ pairwise disjoint decomposition of H, such that $\forall i \in [1..r] : \forall u \in [1..k] : x_u^{(i)} \in \in seq(H_i)$.*

Definition 2.2. Elementwise processable function: *We call an $F : X \longmapsto Y$ function elementwise processable iff*

for all $x, x^{(1)}, x^{(2)} \in X$, such that $cdd(x, (x^{(1)}, x^{(2)}))$ and for all $i \in [1..l]$:

$$(3) \qquad F(x^{(1)})_i \cup F(x^{(2)})_i = F(x)_i,$$

$$(4) \qquad F(x^{(1)})_i \cap F(x^{(2)})_i = \emptyset.$$

It is obvious that a similar assertion is true for every $(x^{(1)}, \ldots, x^{(r)})$ total disjoint decomposition of x. Now we can formalize the specification of the problem. Let F be the elementwise processable function, the value of which is to be computed.

Specification 2.1. *Let A be the state space, B the parameter space and the Q predicates as follows:*

$$A = \underset{x}{X} \times \underset{y}{Y} \qquad B = \underset{x'}{X} \qquad Q(x') = (x = x')$$

$$(5) \qquad Q(x') \in INIT_{x'}$$

(6) $$Q(x') \in TERM_{x'}$$

(7) $$FP_{x'} \Rightarrow (y = F(x')).$$

3. A sequential solution

To produce a solution we first refine specification 2.1 by weakening the fixed-point condition (7) and introducing an invariant. We utilize the main feature of F, viz. that it is elementwise processable. We devide x' into two parts: the items of x' not yet processed are collected in x while the ones already processed are denoted by $x' - x$. We keep it invariant, that x and $x' - x$ form a total disjoint decomposition of x'. The program can make progress by simply processing an item from x until it becomes empty.

Specification 3.1.

(8) $$Q(x') \in INIT_{x'}$$

(9) $$Q(x') \in TERM_{x'}$$

(10) $$inv_{x'} : cdd(x', (x, x' - x))$$

(11) $$inv_{x'} : (y = F(x' - x))$$

(12) $$FP_{x'} \Rightarrow (x = \ll\gg, \ldots \ll\gg)$$

It is easy to prove that this specification is a refinement of Specification 2.1. We only have to show that (7) is a consequence of (10), (11) and (12). (In fact all we need are (11) and (12).) Knowing that the invariants hold in any fixed-point of the program the conclusion can be drawn that $FP_{x'} \Rightarrow (y = F(x' - (\ll\gg, \ldots, \ll\gg)))$, thus $FP_{x'} \Rightarrow (y = F(x'))$. Now let us see the following abstract program:

Program 3.1. Sequential elementwise processing

$$s_0: \quad y,\ ch := (\emptyset, \ldots, \emptyset),\ false$$

$$\{ \quad \overline{\sqcup}\ e,\ ch := min\{lov(x_i) \mid i = 1..k \wedge x_i \neq \ll\gg \},\ true$$
$$if\ x \neq (\ll\gg, \ldots, \ll\gg) \wedge \neg ch$$

$$\overline{\sqcup}\ (\ y, ch := (y_i \cup F(sl(x,e))_{i=1}^{l},\ false\ \ ||$$
$$\|_{i=1}^{k}\ (x_i := lorem(x_i),\ if\ x_i \neq \ll\gg\ \wedge lov(x_i) = e)$$
$$),\ if\ ch$$

$$\}$$

where $\forall i = 1..k$:

$$sl(x,e)(i) ::= \begin{cases} \ll e \gg, & \text{if } x_i \neq \ll\gg \wedge lov(x_i) = e \\ \ll\gg, & \text{otherwise} \end{cases}$$

Again, it is not difficult to verify that this program solves the latter specification. To prove (9) we can use the pair $v = (\sum_{i=1}^{k} dom(x_i),\ khi(ch))$ with lexicographic ordering as a variant function. ($khi(b)$ equals to 1 if $b = true$, otherwise it is 0.) For (10) and (11) we have to compute the strongest postcondition of the initial statement s_0 and the weakest preconditions of the other two statements. Finally we can calculate the fixed-point of the program and show that (12) is a consequence of it.

4. The ways of parallelism

Before searching parallel solutions first we have to fix how many processors we have and how big the input is. To simplify the computation of costs we assume that the dimension of X is equal to the dimension of Y and to the number of processors: $k = l = p$. We describe the size of the input with two numbers:

(13) $$N = \sum_{i=1}^{k} dom(x_i)$$

$$M = \left| \bigcup_{i=1}^{k} \{x_i(j) | j \in range(x_i)\} \right| \tag{14}$$

We call N the 'bag-size', and M the 'set-size' of the input. Since x_i is a strictly monotonic sequence, it is clear that $M \leq N \leq k * M$. We can also assume that $k \ll M$ (in general k is about $2 - 64$ while M can even be some million).

The sequential program described in the previous section needed $O(M)$ steps to reach the fixed-point. But we must not forget that the first statement needed $O(k)$ time to be executed as it contained a minimum-search. With k processors one can compute the minimum of k numbers in $O(log(k))$ steps ([7]), hence executing this statement parallelwise seems to be a good way to reduce computational costs. However, we take the computation of $F(sl(x, e))$ the dominant cost, so we reject this solution. We are willing to find a solution in which all the processors have to take approximetely M/k dominant steps and they work parallelwise. On the other hand we do not want to ignore the other costs either: we will let the programmer decide to what extent he wants to regard them.

Our plan is to provide each processor with a block from a balanced total disjoint decomposition and let them all work. Therefore we should divide x into k parts, each part having a 'set-size' of M/k. Notice that deciding whether a total disjoint decomposition is balanced or not, needs an elementwise processing, viz. computing the union of the x_i-s so that we can determine the value of M. Our job is *catch-22*. How can we produce a balanced total disjoint decomposition for our effective elementwise processing algorithm if it is an elementwise processable function to create it?

We can avoid this trap by making a compromise. One way to do this is to use N, the 'bag-size' to cut x: each block having the 'bag-size' of N/k. Unfortunately this way we may get blocks with huge difference in 'set-size'. In worst case we may have a block with k times larger 'set-size' than the others. To solve this problem we give up one more assumption, notably that the number of blocks equals to the number of processors. The smaller blocks we have the more balanced the occupation of the processors is. On the other hand, the more blocks we have the more administrative costs arise, not to mention the increasing communication costs in a distributed system. It is the job of the programmer to estimate the constants behind the O signs and find a balance. In the following section we discuss this idea more preciously.

5. A parallel solution

First we sum up the assumptions. We have k processors in a distributed system. On the i-th processor we have the sequence x_i. The total 'bag-size' of x is N, the total 'set-size' is M. We have to receive the value of an F elementwise processable function, that is y_i on the i-th processor. ($p = k = l \ll M \leq N \leq k * M$).

Our strategy is as follows: we devide the input into blocks, orthogonally to the sequences. These blocks form a total disjoint decomposition of x, therefore each block can be processed independently from the others. The distribution of the blocks among the processors is dynamic. At the beginning every processor is given a block which it can process with the sequential program discussed in Section 3. When ready it is provided with another block, until we run out of blocks. In the end we have to compute y from the partial results appeared on the k processors.

It seems to be rather difficult to figure out how many processing steps a processor has to take. But it turns out that it is quite simple to determine how much it differs from the optimum. We deviate from the optimum at the moment when we run out of blocks. From that moment on we have processors without a job, while the others are still working. Fortunately this does not last long, as all the working processors have only to complete the block they are working on and then terminate. That is why it takes no more than $M/k + B$ steps to take for the k processors to compute the value of F, where B is the 'bag-size' of the largest block. (In fact even 'set-size' could be used.) This proves the following lemma:

Lemma 5.1. *Our algorithm takes no more than $M/k+B$ processing steps with k processors, where B is the 'bag-size' of the largest block.*

It is worth to choose B as small as possible, but we must not forget about the other costs which increase with the number of the blocks: the number of cuts, the number of communication steps, etc. Since the number of blocks depends on b, the 'bag-size' of the smallest block (viz. not more than N/b), we should ensure that the difference between B and b is as small as possible. In subsection 5.1 we present an algorithm which ensures that $b \geq B/8$.

5.1. Preparing the total disjoint decomposition

During the algorithm we have two sets for storing the blocks bigger than B in one of them and the blocks smaller than B in the other. For representing these sets we will use sequences (the reason can be found in Section 6). Initially

we have only one block, containing the whole x. At the beginning every processor sends the necessary information about its sequence to one of the processors, let us say to the first one. This processor computes N, determines the value of B, puts the first block into the first set of blocks and initializes the second set as an empty one. While the first set is not empty we take a block greater than B and cut it into two. We put the resulting two blocks in the first set or in the other one depending on the size.

In subsection 5.1.1 we investigate where a block should be cut (how to find the so called cut-value) so that the resulting two blocks do not differ too much in size. After determining the cut-value the first processor sends it to all the processors. Each processor cuts his own part of the above-mentioned block by processing a binary search on x_i. Finally they send the required information about the two originating blocks to the first processor.

5.1.1. Where to cut a block

Let us suppose we have a block with 'bag-size' n ($n > B$). We show how to find an $h \in H$ so that cutting the block into two according to h, the smaller originating block has 'bag-size' at least $n/8$. We assume to know the following information about the block:

d_i ($i = 1..k$): the size of the ith part of the block, that is the length of the appropriate subsequence in x_i.

f_i ($i = 1..k$): the median value of the ith part of the block, that is $x_i((s + e)/2)$ where s is the starting, e is the ending index of the appropriate subsequence.

First let us sort the f_i values. This way we obtain an $i(j)$ ($j = 1..k$) permutation of the $1, 2, \ldots, k$ numbers, such that $f_{i(j)} \leq f_{i(j+1)}$. Now we are looking for $r \in [1..k]$ for which $|(\sum_{j=1}^{r-1} d_{i(j)}) - (\sum_{j=r+1}^{k} d_{i(j)})|$ is minimal. We are going to choose $f_{i(r)}$ as h, the cut-value: we hope, $f_{i(r)}$ will not be far from the median of the block.

Algorithm 5.1. *Let $1 \leq p \leq q \leq k$, and let us keep it invariant, that*

$$g = \left(\sum_{j=1}^{p-1} d_{i(j)}\right) + d_{i(p)}/2, \tag{15}$$

$$h = \left(\sum_{j=q+1}^{k} d_{i(j)}\right) + d_{i(q)}/2. \tag{16}$$

Initially we can provide this invariant by setting $p = 1$, $q = k$ and $g = d_{i(1)}/2$, $h = d_{i(k)}/2$. While $p \neq q$ increase p by one (and increase g), if $g < h$ and

decrease q (and increase h), if $h < g$. When p and q become equal, we have r, namely $r = p = q$.

Theorem 5.1. *The algorithm 5.1 is optimal in the sense that $|g - h|$ is minimal among all possible cuts of $d_{i(j)}$-s.*

Proof. Let us suppose that the cut specified by the pair (u, v) is optimal, that is $|u - v|$ is minimal. Let us suppose that $u \neq g$ (and, of course, $v \neq h$). Since $u + v = n = g + h$, either $u < g$ or $v < h$ holds. Let us suppose e.g. that $v < h$ (and $u > g$). In this case v contains a shorter postfix of the $d_{i(j)}$-s. Our algorithm decreased q in each step by at most one, hence there was a period of time, when the variable h had the value v. Later on there was a moment when h would become greater then v. Let us denote the values of our variables at that moment by p', g', q' and h' ($h' = v$). At this step the algorithm chose q' for decrementation - this means, that $g' \geq h'$. It is also clear, that g' contains a shorter prefix of the $d_{i(j)}$-s than g, that is why $g' \leq g$. Collating these two relations we get $g - h' \geq 0$. Of course $h' \leq h$, hence $h - h' \geq 0$, too. Now let us compare $|g - h|$ and $|u - v|$.

- $|g - h| = |g - h + h' - h'| = |(g - h') + (h' - h)| \leq ||g - h'| + |h' - h|| = |g - h'| + |h - h'| = (g - h') + (h - h')$, and
- $|u-v| = |u+v-v-v| = |g+h-v-v| = |g+h-h'-h'| = |(g-h')+(h-h')| = ||g - h'| + |h - h'|| = |g - h'| + |h - h'| = (g - h') + (h - h')$.

Hence we got $|g - h| \leq |u - v|$, that is $|g - h|$ is minimal. The other case, namely when $u < g$ can be proven similarly.

Theorem 5.2. *$|g - h| \leq n/2$ holds at the end of algorithm 5.1.*

Proof. We present an algorithm producing a cut for which the difference of the two parts is not more than $n/2$. Since algorithm 5.1 produced an optimum on the difference of the sizes of the two resulting parts, Theorem 5.3 completes this proof.

Algorithm 5.2. *Let $1 \leq p \leq q \leq k$, $g = \sum_{j=1}^{p-1} d_{i(j)}$, and $h = \sum_{j=q+1}^{k} d_{i(j)}$. Initially $p = 1$ and $q = k$. If $g \leq h$ and $g + d_{i(p)} \leq h + n/2$, then let us increase p by one (and increase g). Similarly, if $h \leq g$ and $h + d_{i(q)} \leq g + n/2$, then let us decrease q and increase h. Let us continue this until $p = q$. The last $d_{i(j)}$, viz. $d_{i(p)}$ should be divided between g and h, thus preserving the invariant $|g - h| \leq n/2$.*

If it happens to occur during this algorithm that we cannot increase the smaller part keeping the invariant true (e.g. $g \leq h$, but $g + d_{i(p)} > h + n/2$), then we do as follows. We add all the remaining $d_{i(j)}$-s to h, so that q becomes equal to p. We divide $d_{i(p)}$ by 2 and add one half to g and the other to h. Since $d_{i(p)} > n/2$ (because of the previous $g \leq h$ and $g + d_{i(p)} > h + n/2$ conditions),

the half of it is at least $n/4$. Therefore at the end of the algorithm both g and h is at least $n/4$, thus their difference is at most $n/2$.

Theorem 5.3. *$|g - h| \leq n/2$ holds at the end of algorithm 5.2.*

Proof. $|g - h| \leq n/2$ is invariant (cf. the description of algorithm 5.2).

Theorem 5.4. *At algorithm 5.1 the lower estimation $n/2$ for $|g - h|$ is strict.*

Proof. We show an input for which the optimal cut produces a difference $n/2$ in the size of the two originating parts. Let $d_{i(1)} = d_{i(2)} = n/2$, $d_{i(3)} = \ldots = d_{i(k)} = 0$. For this input the optimal cut is either $r = 1$ or $r = 2$. In both cases the difference $|g - h|$ is exactly $n/2$.

5.1.2. About the sizes of the blocks

Now we show that the 'bag-size' of the smallest block is at least $b = B/8$, while the 'bag-size' of the largest block is at most B, where B is a constant chosen in advance. The second part of our assertion is evident since the applied algorithm cuts all the blocks with size greater than B into two. On the other hand it is enough to show, that the smaller originating block has the size of at least $n/8$. Since n, the size of the original block is at least B, we never receive a block with size under $B/8$.

Theorem 5.5. *When cutting a block, the size of the smaller originating block is at least $n/8$ (where n is the size of the original block).*

Proof. We will use the notations of algorithm 5.1. Since $f_{i(j)} \leq f_{i(j+1)}$, the lower originating block contains $f_{i(r-1)}, \ldots, f_{i(1)}$, and the higher one $f_{i(r+1)}, \ldots f_{i(k)}$. The lower block contains not only $f_{i(j)}$, but the lower half of the $d_{i(j)}$ element of the appropriate subsequence of $x_{i(j)}$ (for all $j = 1..r-1$), since these elements are not greater than $f_{i(j)}$. Similar assertion holds for the higher block (for all $j = r+1..k$). In addition both the lower block and the upper one contain $d_{i(r)}/2$ elements from $x_{i(r)}$. Therefore the lower block contains at least $\sum_{j=1}^{r} d_{i(j)}/2 = g/2$ elements, and the higher one at least $\sum_{j=r}^{k} d_{i(j)}/2) = h/2$ elements. Since $|g - h| \leq n/2$, both g and h is at least $n/4$, hence both originating blocks are containing at least $n/8$ elements.

5.2. Costs

We have pointed out that our algorithm takes $M/k + B$ processing steps on k processors. The reason is that the period of time, when we have processors without a job is at most the time a processor needs to process one block.

Since our largest block has the 'bag-size' of at most B, the 'set-size' cannot be bigger than the 'bag-size' and the time needed to process a block is directly proportional to the 'set- size' of the block, we have that the difference from the optimum cost is at most B.

Now we sum up the other costs. To prepare the total disjoint decomposition we have to cut into two all the blocks greater than B. To do this we need no more cuts than the number of the blocks, which is not more than N/b. In addition to this we need k messages to collect the necessary information about x on the first processor and $k+1$ steps to compute N and B.

We have to quantify the costs of one cut. First we have to sort the f_i-s. This takes $k * log(k)$ steps on one processor. (It can be done in $O(log(k)* *log(k))$ steps on k processors, but considering that k is rather small, we should better complete the sort on one processor.) To find r we apply algorithm 5.1, which requires $O(k)$ steps. We send a message to each processor with data for the binary search, this means k messages. The binary searches complete in $O(log(M))$ steps on the k processors. After that we again need k messages to transfer the description of the arising blocks to the first processor. Altogether $2 * k$ messages and $O(k * log(k) + log(M))$ steps are required.

After constructing the total disjoint decomposition we can start processing the blocks. To process a block a processor has to require it from the first processor. After obtaining the beginning and ending indices of the parts of the block it can ask the missing subsequences from the other $k-1$ processors. Summing up all these, we need $2 * k$ messages to process a block. Finally we have to prepare y from the partial results stored on the k processors. Each part of y_i should be transfered to the processor i and there they should be united. This requires $k * k$ messages (k messages to each processor).

Now we can give a summary of the costs:

communication costs: $4 * (N/b) * k + k^2$ messages

processing steps: $B + M/k$

other steps: $(N/b) * O(k * log(k) + log(M))$

In the last row we can see $k * log(k) + log(M)$. If $O(M)$ is greater than $O(k^k)$, then $log(M)$ is the dominant, otherwise $k * log(k)$.

In general the dominant costs are the processing steps, that is why $B = = M/k^2$ is a good choice. Anyway, we leave it to the programmer to decide to what extent he takes into account the other costs and to determine the value of B. For example if we do not distinguish processing steps from the others, we can find the following. Since one processing step requires k elementary steps (it covers a minimum search), the overall number of steps needed is less than $O(M + B * k + M * k^3/B)$. We would like to have M as the dominant component in this sum. Let us suppose we choose B as M/k. In this case M should be at

least $O(k^4)$ which is quite probable. If we would like to ensure that the costs are $M + O(M/k)$, we need an input with size at least $O(k^7)$.

6. Derivation of the parallel program

Our solution is a sequence of three programs. The first one prepares the total disjoint decomposition, the second one completes the elementwise processing and the third one collects the partial results. We specify these three programs one after the other, so that the precondition of one is a consequence of the postcondition of the previous, and altogether they form a refinement of specification 2.1. Verification is usually left to the reader.

First of all we describe the types and predicates used in our specifications.

- IND is the indextype of the x_i-s (possibly $IND = \mathcal{N}$)
- $BLOCK = (s : IND, d : \mathcal{N}, f : H)$, descriptor of a blockpart
- $BV = vector\ [1..k]\ of\ BLOCK$, descriptor of a whole block
- $BVS = seq(BV)$, the description of a total disjoint decomposition
- $BVSP = BVS^k$, for storing the actually processed block (as a sequence of one length) on each processor
- $YV = (\ vector\ [1..l]\ of\ \mathcal{P}(H)\ )^k$, for collecting the partial results in (remember that $k = l$)
- $opt : \mathcal{N} \longmapsto \mathcal{N}$, used to determine the optimal value of the size of the largest block, viz. B
- $CDD : X \times BVS \longmapsto \mathcal{L}$,
  $CDD(x, b) = cdd(x, ((x_i(b(j)[i].s..b(j)[i].s + b(j)[i].d - 1))_{j \in range(b)})_{i=1}^k)$
- $DENSE : BVS \times \mathcal{N} \longmapsto \mathcal{L}$,
  $DENSE(b, B) = (\forall j \in range(b) : B/8 \ \leq \ \sum_{i=1}^{k} b(j)[i].d \ \leq \ B)$

Remark 6.1. *For example* $CDD(x, \ll ((lov(x_i), dom(x_i), f_i)_{i=1}^k) \gg)$, *since* $cdd(x, (x))$ *holds. (Notice that considering* CDD *the values of the* f_i*-s are indifferent.)*

6.1. Total disjoint decomposition

Specification 6.1.

$$\underset{x}{A = X} \times \underset{bb}{BVS} \times \underset{N}{\mathcal{N}} \times \underset{B}{\mathcal{N}} \qquad \underset{x'}{B = X} \qquad Q(x') = (x = x')$$

$$Q(x') \in INIT_{x'} \qquad Q(x') \in TERM_{x'} \qquad inv_{x'} : Q(x')$$

$$FP_{x'} \Rightarrow (\; N = \sum_{i=1}^{k} dom(x'_i) \; \wedge \; B = opt(N) \; \wedge \; CDD(x', b) \; \wedge \; DENSE(b, B) \;)$$

We are looking for a solution which is a sequence of two programs. The first one computes N and B, and initializes the block administration. We introduce a new component in the state space, $bb : BVS$. In this component we store the blocks that are bigger than B. The first program puts one element into bb: the one that corresponds to x.

6.1.1. Initializing part

Specipication 6.2.

$$A = \underset{x}{X} \times \underset{bb}{BVS} \times \underset{N}{\mathcal{N}} \times \underset{B}{\mathcal{N}} \qquad B = \underset{x'}{X} \qquad Q(x') = (x = x')$$

$$Q(x') \in INIT_{x'} \qquad Q(x') \in TERM_{x'} \qquad inv_{x'} : Q(x')$$

$$(17) \qquad FP_{x'} \Rightarrow (\; N = \sum_{i=1}^{k} dom(x'_i) \; \wedge \; B = opt(N) \; \wedge \; CDD(x', bb) \;)$$

We refine this specification by introducing $j : \mathcal{N}$ and $blk : BV$, and by exchanging the fixedpoint condition to a new one and an invariant (cf. Remark 6.1).

$$inv_{x'} : (j \in [0..k] \; \wedge \; N = \sum_{i=1}^{j} dom(x'_i) \; \wedge \; \forall i \in 1..j :$$

$$(18) \qquad blk[i] = (\; lov(x_i), \; dom(x_i), \; x_i((lov(x_i) + dom(x_i) - 1)/2) \;) \;)$$

$$(19) \qquad FP_{x'} \Rightarrow (\; j = k \; \wedge \; B = opt(N) \; \wedge \; CDD(x', bb) \; \wedge \; bb = \ll blk \gg \;)$$

Program 6.1

s_0 : $j, \; N \; := \; 0, \; 0$

{

$\square \; N, \; j, \; blk[j+1] \; :=$

$N + dom(x_{j+1}), \quad j+1,$

$(\; lov(x_{j+1}), \; dom(x_{j+1}), \; x_{j+1}(\; (lov(x_{j+1}) + dom(x_{j+1}) - 1) \; /2 \;) \;),$

$if \quad j < k$

$\square \; B, \; bb \; := \; opt(N), \; \ll blk \gg, \quad if \quad j = k$

}

The assertions, that the original fixedpoint condition (17) is a consequence of the introduced invariant (18) and fixedpoint condition (19) and that the program solves the refined specification are obvious.

6.1.2. Consuming bb

Specification 6.3.

$$\underset{x}{A} = \underset{x}{X} \times \underset{b}{BVS} \times \underset{bb}{BVS} \times \underset{N}{\mathcal{N}} \times \underset{B}{\mathcal{N}} \qquad \underset{x'}{B} = \underset{x'}{X} \times \underset{bb'}{BVS} \times \underset{N'}{\mathcal{N}} \times \underset{B'}{\mathcal{N}}$$

$$Q(x', bb', N', B') = (\ x = x' \ \wedge \ N = N' \ \wedge \ B' = B \ \wedge \ N = \sum_{i=1}^{k} dom(x'_i) \ \wedge \ B = opt(N)\ )$$

$$R(x', bb', N', B') = (bb = bb' \ \wedge \ CDD(x', bb))$$

$$Q(x', bb', N', B') \wedge R(x', bb', N', B') \in INIT_{x', bb', N', B'} \tag{20}$$

$$Q(x', bb', N', B') \wedge R(x', bb', N', B') \in TERM_{x', bb', N', B'} \tag{21}$$

$$inv : \ Q(x', bb', N', B') \tag{22}$$

$$FP_{x', bb', N', B'} \Rightarrow (CDD(x', b) \wedge DENSE(b, B)) \tag{23}$$

We introduce the predicate $HALFVALUE$ and refine this specification as follows:

$$HALFVALUE(x, b) = (\forall j \in range(b) : \forall i \in 1..k : b(j)[i].f \ = \ x_i(b(j)[i].s + b(j)[i].d/2))$$

Specification 6.4.

$$inv : CDD(x', b \circ bb) \tag{24}$$

$$inv : DENSE(b, B) \tag{25}$$

$$inv : HALFVALUE(x, bb \circ b) \tag{26}$$

(27) $$FP \Rightarrow (bb = \ll\gg)$$

This specification can be solved by a loop: we choose (24-26) as loop invariant, $(bb \neq \ll\gg)$ as loop condition and $dom(bb)$ as variant function.

Program 6.2.

$$\begin{aligned} &\textit{while } dom(bb) \neq 0 \textit{ loop} \\ &\quad h := CutValue(lov(bb)) \\ &\quad ub, lb := Cut(x, lov(bb), h) \\ &\quad bb, b := Update(bb, b, ub, lb) \end{aligned}$$

where $ub, lb : BV$ and for $bv : BV$ the $\{h := CutValue(bv)\}$ is the sequence of $\{i := Sorting(bv)\}$ (which computes i, a permutation of $\{1..k\}$), and $\{h := := Middle(bv, i)\}$ (which can be found in subsection 5.1.1). Furthermore,

$$Update(bb, b, ub, lb) =$$

$$= \begin{cases} hiext(hiext(lorem(bb), ub), lb),\ b & \text{if } bagsize(ub) > B \wedge \\ & \quad bagsize(lb) > B \\ hiext(lorem(bb), ub),\ hiext(b, lb) & \text{if } bagsize(ub) > B \wedge \\ & \quad bagsize(lb) \leq B \\ etc. \end{cases}$$

Specification 6.5. *Now we specify* $\{ub, lb := Cut(bv, h)\}$*:*

$$\begin{array}{cccccc} A = X & \times & BV & \times & BV & \times & BV & \times & H \\ & x & & ub & & lb & & bv & & h \end{array} \qquad \begin{array}{ccccc} B = X & \times & BV & \times & H \\ & x' & & bv' & & h' \end{array}$$

$$Q(x', bv', h') = (x = x' \wedge bv = bv' \wedge h = h')$$

$$Q(x', bv', h') \in INIT \quad Q(x', bv', h') \in TERM \quad inv : Q(x', bv', h')$$

$$\begin{aligned} FP \Rightarrow (HALFVALUE(x', \ll lb.\, ub \gg) \ \wedge \ \forall i \in 1..k : \\ lb[i].s = bv'[i].s \ \wedge \ lb[i].s + lb[i].d = ub[i].s \wedge \\ lb[i].d + ub[i].d = bv'[i].d \wedge \\ \forall j \in lb[i].s..lb[i].s + lb[i].d - 1 : \ x'_i(j) \leq h \ \wedge \\ \forall j \in ub[i].s..ub[i].s + ub[i].d - 1 : \ x'_i(j) > h \ ) \end{aligned}$$

This can be solved parallel on k processors by independently executing the following program:

Program 6.3.

$$\bigsqcup_{i=1}^{k} v_i := BinSearch(x_i(bv[i].s..bv[i].s + bv[i].d - 1), h)$$

$$\bigsqcup_{i=1}^{k} lb[i], ub[i] :=$$

$$(bv[i].s,\ v_i - bv[i].s + 1,\ x_i(bv[i].s + v_i)/2)),$$

$$(v_i + 1,\ bv[i].s + bv[i].d - v_i,\ x_i((v_i + bv[i].s + bv[i].d)/2))$$

}

Specification and solution of the remaining subprograms can be completed by regarding Section 5.1.1. We also leave it to the Reader to prove that the presented programs solve the appropriate specifications.

6.2. Elementwise processing

Specification 6.6.

$$\underset{x}{A} = \underset{x}{X} \times \underset{b}{BVS} \times \underset{yv}{YV} \qquad \underset{x'}{B} = \underset{x'}{X} \times \underset{b'}{BVS}$$

$$Q(x', b') = (x = x') \qquad R(x', b') = (b = b' \wedge CDD(x', b'))$$

(28) $$Q(x', b') \wedge R(x', b') \in INIT_{x', b'}$$

(29) $$Q(x', b') \wedge R(x', b') \in TERM_{x', b'}$$

(30) $$inv :\ Q(x', b')$$

(31) $$FP_{x', b'} \Rightarrow (\forall i = 1..k :\ \bigcup_{j=1}^{k} yv_j[i] = F(x')_i)$$

Let us define the functions $Leftout : X \times BVS \longmapsto X$ and $R : X \times \mathcal{N} \times \times IND \times BVS \longmapsto S$ this way:

$$Leftout(x, b) = (\underset{j \in range(x_i)}{\circ} R(x, i, j, b))_{i=1}^{k},$$

$$R(x, i, j, b) =$$

$$= \begin{cases} \ll\gg, & \text{if } \exists t \in range(b) : b(t)[i].s \leq j < b(t)[i].s + b(t)[i].d \\ \ll x_i(j) \gg, & \text{otherwise} \end{cases}$$

Now we introduce $bv : BVSP$ with the following:

$$(32) \qquad inv : (\forall j = 1..k : \bigcup_{i=1}^{k} yv_i[j] = F(Leftout(x', (\overset{k}{\underset{i=1}{\circ}} bv_i) \circ b))_j)$$

$$(33) \qquad inv : CDD(x', (\overset{k}{\underset{i=1}{\circ}} bv_i) \circ b)$$

$$(34) \qquad FP_{x',b'} \Rightarrow (\forall i \in 1..k : bv_i = b = \ll\gg)$$

We can choose the following pair as variant function:

$$(35) \qquad v \; = \; (\; dom(b), \; \sum_{i=1}^{k} dom(bv_i) \;)$$

Let us suppose, we have a sequential program named EP (Elementwise Processing), which satisfies:

$$\{ \; \forall i \in 1..k : blk[i].s, blk[i].s + blk[i].dom - 1 \in range(x_i) \; \}$$

$$v := EP(u, x, blk)$$

$$\{ \; \forall j = 1..k : \; v[j] = u[j] \cup F(\; (x_i(blk[i].s..blk[i].s + blk[i].d - 1))_{i=1}^{k} \;)_j \; \}$$

A program that solves the specification above can be obtained by a little modification of program 3.1. Now we can assemble program 6.4:

Program 6.4.

$$s_0 : \quad \|_{i=1}^{k} bv_i, yv_i := \ll\gg, \ \ll\gg$$

$$\{ \quad \sqcup_{i=1}^{k} bv_i, b := hiext(bv_i, lov(b)), \ lorem(b) \qquad if\ bv_i = \ll\gg \wedge b \neq \ll\gg$$

$$\sqcup_{i=1}^{k} yv_i, bv_i := EP(yv_i, x, lov(bv_i)), \ lorem(bv_i), \quad if\ bv_i \neq \ll\gg$$

$$\}$$

6.3. Collecting the partial results

Specification 6.7.

$$A = \underset{y}{Y} \times \underset{yv}{YV} \qquad B = \underset{yv'}{YV} \qquad Q(yv') = (yv = yv')$$

$$Q(yv') \in INIT_{yv'} \tag{36}$$

$$Q(yv') \in TERM_{yv'} \tag{37}$$

$$FP_{yv'} \Rightarrow (\ \forall j \in 1..k : \ y_j \ = \underset{i=1}{\overset{k}{\cup}} yv'_i[j]) \tag{38}$$

Let us introduce the following invariant and fixedpoint condition. (39-40) is a refinement of (38).

Specification 6.8.

$$inv : (\forall j \in 1..k : \underset{i=1}{\overset{k}{\cup}} yv_i[j]) \cup \ y_j \ = \underset{i=1}{\overset{k}{\cup}} yv'_i[i]) \tag{39}$$

$$FP_{yv'} \ \Rightarrow \ (\ \forall i, j \in 1..k : \ yv_i[j] \ = \ \emptyset \ ) \tag{40}$$

Program 6.5.

$$s_0 : \quad \sqcup_{j=1}^{k} \ y_j \ := \ \emptyset$$

$$\{ \sqcup_{i,j=1}^{k} \ y_j, \ yv_i[j] \ := \ y_j \cup yv_i[j], \ \emptyset \ \ \}$$

7. Summary

We have intended to create a programming theorem for the problem of evaluating an elementwise processable function. We have searched for a parallel solution. We have given a formal specification to the problem, using a relational model of programming. We have invented an algorithm which provided that k processors could work on the input concurrently, each processing some parts of it. These parts form a total disjoint decomposition of the input: constructing this partitioning is the main task of our algorithm.

We have pointed out that with a dense total disjoint decomposition we need only $M/k + B$ processing steps, where M is the size of the input (proportional to the number of steps that the sequential program has to take), k is the number of processors and B is the size of the largest block. (Notice, that the optimum would be M/k.) During the algorithm we send no more than $32 * k^2 * M/B + k^2$ messages in a distributed system and the other costs are below $O(M * k^2 * log(k)/B)$. We leave it to the programmer to choose the value of B. However, if $M = O(k^7)$, then $B = M/k^2$ is a good choice.

We have demonstrated how to derive an abstract program that solves the formal specification. During the program derivation we have restricted ourselves to the parallel subprograms and to presenting how they can be integrated in sequential programs and vice versa. Finally we would like to draw it to the Readers attention, that combining the tools of the sequential model and that of the parallel one can facilitate the derivation of a complex program.

References

[1] **Fóthi Á. and Horváth Z.,** A Parallel Elementwise Processing, *Proceedings of the 2nd Austrian-Hungarian Workshop on Transputer Applications*, eds. Ferenczi Sz. and Kacsuk P., Budapest, Hungary, KFKI-1995-2/M,N Report

[2] **Horváth Z.,** The Formal Specification of a Problem Solved by a Parallel Program – a Relational Model, *Proceedings of the Fourth Symposium on Programming Languages and Software Tools, Visegrád, Hungary, June 8-14, 1995*, ELTE, Budapest, 1995, 165-179.

[3] **Fóthi Á.,** *Bevezetés a programozáshoz*, egyetemi jegyzet, ELTE TTK, Budapest, 1983.

[4] **Fóthi Á.,** A Mathematical Approach to Programming, *Annales Univ. Sci. Bud. Sect. Comp.*, **9** (1988), 105-114.

[5] **Fóthi Á. and Horváth Z.,** The Weakest Precondition and the Theorem of the Specification, *Proceedings of the Second Symposium on Programming Languages and Software Tools, Pirkkala, Finland, August 21-23, 1991,* Report A-1991-5, University of Tampere, Department of Computer Science, 1991, 39-47. (to appear in *Acta Cybernetica*).

[6] **Workgroup on Relational Models of Programming,** Some concept of a Relational Model of Programming, *Proceedings of the Fourth Symposium on Programming Languages and Software Tools, Visegrád, Hungary, June 8-14, 1995,* 434-446.

[7] **Horváth Z.,** The asynchronous computation of the values of an associative function, *Acta Cybernetica* (to appear)

[8] **Chandy K.M. and Misra J.,** *Parallel program design: a foundation*, Addison-Wesley, 1989.

Á. Fóthi, Z. Horváth and T. Kozsik
Department of General Computer Science
Eötvös Loránd University
VIII. Múzeum krt. 6-8.
H-1088 Budapest, Hungary
kto@dtalk.elte.hu