The Journal of Supercomputing, 29, 5–25, 2004

On Data Distributions in the Construction of Parallel Programs

VIRGINIA NICULESCU vniculescu@cs.ubbcluj.ro

Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

Abstract. Data distributions have a serious impact on time complexity of parallel programs, developed based on domain decomposition. A new kind of distributions—set distributions, based on set-valued mappings, is introduced. These distributions assign a data object to more than one process. The set distributions can be used especially when the number of processes is greater than the data input size, but, sometimes using set distributions can lead to efficient general parallel algorithms. The work-load properties of these distributions and their impact on the number of communications are discussed. In order to illustrate the implications of data distributions in the construction of parallel programs, some examples are presented. Two parallel algorithms for computation of Lagrange interpolation polynomial are developed, starting from simple distributions and set distributions.

Keywords: parallel computation, distribution, complexity, correctness, numerical algorithms

1. Introduction

Many models of parallel programs use distributed data objects, like arrays or matrices. In a parallel program, processes can independently perform operations on their data until a global information is needed. Communication processes are responsible for combining and collecting global information. This causes synchronization points in the parallel program, which usually lead to waiting times. The impact of distributions on the time complexity of a parallel program is therefore an important issue.

The model for parallel programs design, used here, is based on that presented in Loyens [6]. The model considers a parallel program as a number of cooperating parameterized processes with similar structures.

1.1. Notations

A notation for quantifications is used:

$$(\odot k : Q : E),$$

where $\odot$ is a quantifier $(\sum, \forall, \max, \ldots)$, k is the list of bound variables, Q is the predicate describing the domain of the bound variables, and E is an expression. Function application is denoted by an infix, left associative dot "." operator. The lambda calculus is used for the definition of distribution functions. The set

$(\forall i : 0 \leq i < n : i)$ is denoted by $\overline{n}$. The integer division and remainder use the symbols / and % respectively. Also, the proof notation of W. F. H. Feijen is used.

1.2. *Program notation*

The program notation is based on Dijkstra's guarded command language. The programs use variables declarations in a Pascal-like style, extended with local scope rules. The symbols $||[x \ldots]||$ delimit the scope of variable x. Variables usually have the type "int" or "real". An array f of length n with base type "real" is declared as "$f(i : 0 \leq i < n)$: *array of real*".

The programs use the following constructs:

- *abort*—stop forever;
- *skip*—do nothing;
- $x := e$—assignment;
- $S0$; $S1$—sequential composition;
- *if* $B0 \rightarrow S0 [] B1 \rightarrow S1$ *fi*—alternative construct;
- *do* $B0 \rightarrow S0 [] B1 \rightarrow S1$ *od*—repetition.

An extension to Dijkstra's notation is introduced for the *for all* statement:

- *for all* $i : i \in Set : S.i$ *lla rof*—arbitrary order.

Parallel composition is denoted by:

- *par* $q : 0 \leq q < p : S.q$ *rap*—parallel composition.

Communication is expressed by the statements:

- $r!e$—output to process r the value of expression e;
- $s?x$—input from process s of a value, which is assigned to x.

1.3. *Time complexity*

For a parallel program with the input size n and the number of processes p, we consider that the time complexity is expressed by:

$$T.p.n = T_f.p.n + \alpha * T_c.p.n,$$

where

- $T_f.p.n =$ the computation complexity (measured in t_f);
- $T_c.p.n =$ the communication complexity (measured in t_c);
- $t_c =$ the time required to communicate a single value;

- t_f = the time required to execute a single operation;
- $\alpha = t_c/t_f$.

The communication complexity depends on the communication network. We will consider here an ideal communication network—a process is directly connected with all the others. Other kinds of networks may also be considered: chain, mesh, hypercube, ...; for these, the communication processes have to be evaluated depending on their structure.

Also, we consider idealized communications—no message startup time. So, we will not analyze the possibilities of coupling some communication processes, in order to reduce the total real communication time. These analyses can be done at the implementation phase.

1.4. *Program derivation*

A parallel program is considered to be formed by many parameterized processes $S.q(0 \leq q < p)$, which are running in parallel. There is no shared memory. A parameterized process is much like a procedure in sequential programming. The difference is that, instead of having only one instantiation in a sequential program, we have many instantiations in a parallel program.

In sequential programming the Hoare-triple [5]

$$\{Q\}S\{R\},$$

is commonly used to denote a formal specification of a program S. This notation expresses that if the program S starts in a state described by the predicate Q, and the program terminates, then, upon completion the predicate R is satisfied.

For specifying a parallel program, both pre- and post-conditions, Q and R, are split up as conjunctions of $p(p > 0)$ local pre- and post-conditions, and a process is associated with each such pair:

$$\{Q.q\}S.q\{R.q\}, \quad \forall q : 0 \leq q < p.$$

So, we may specify a parallel program as follows:

```
|[
{Q}
  par q : 0 ≤ q < p :
  {Q.q}
    S.q
  {R.q}
  rap
{R}
]|
```

A parameterized specification usually refers some local variables representing a part of a distributed object. Therefore, the parameters of a specification are q, p and

a data distribution D. Many choices are possible for D, each of them having an impact on the complexity of the parallel program.

Such a specification forms the starting point for a parallel program derivation, a formal construction of parameterized processes constituting a parallel program.

Our approach for obtaining a parameterized process S from a functional specification is similar to the methods used in sequential programming. These methods obtain an invariant from a specification, in a calculational style. The programs are derived by calculating the necessary conditions to maintain the invariant. Because we have parameterized specifications, we will derive parameterized invariants.

A global specification requires some form of coordination between processes, so in order to satisfy the specification, several processes have to interact with one another, via message exchanging.

A parameterized process is refined into a sequence of ordinary sequential programs and communication processes. A parallel program is, thus, decomposed into layers of process instances.

1.5. Distributions

Static distributions are going to be discussed; in a static distribution the assignments are not changed during the execution of the program.

Two kinds of distributions are presented:

- simple distributions, defined by single-valued mappings;
- set distributions, defined by set-valued mappings.

2. Simple distributions

The simple distributions are characterized by:

- the number of data elements assigned to one process;
- the data distribution on processes.

If the data input is a vector, the simple distribution is called 1D distribution, and if it is a multi-dimensional array, the distribution is called Cartesian distribution [6].

2.1. 1D distributions

Definition 1 $D = (\delta, A, B)$ is called a distribution if A and B are finite sets, and δ is a mapping from A to B. Set A specifies the set of data objects (an array with n elements), and the set B specifies the set of processes, which is usually $\overline{p}$. The function δ assigns each index $i(0 \leq i < n)$, and its corresponding element, to a process number.

Well-known ways of distributing an array are: every element to one unique process (identity), assigning p equally-sized consecutive array segments (linear), and assigning elements cyclically (cyclic):

$$\begin{aligned}\text{identity} &= ((\lambda\ i \cdot i), \overline{n}, \overline{p});\\ \text{linear} &= (\lambda\ i \cdot i/(n/p), \overline{n}, \overline{p}),\ \text{provided that } p \mid n;\\ \text{cyclic} &= (\lambda\ i \cdot (i\%p), \overline{n}, \overline{p}).\end{aligned}$$

Definition 2 Considering a distribution $(\delta, \overline{n}, \overline{p})$, the set of elements from $\overline{n}$, assigned to process $q(0 \leq\ q < p)$, is given by $O.q$:

$$O.q = (\forall i : i \in \overline{n} \wedge \delta.i = q : i).$$

Counting the cardinalities of the sets $O.q$ is meaningful in time complexity analysis. That gives a good indication of the amount of work per process. The sets $O.q$ form a partition of $\overline{n}$.

Definition 3 The maximum/minimum number of data objects assigned to a process for a distribution $(\delta, \overline{n}, \overline{p})$ are defined by:

$$\begin{aligned}\mathrm{Ma}(\delta) &= (\max q : 0 \leq q < p : |O^{\delta}.q|),\\ \mathrm{Mi}(\delta) &= (\min q : 0 \leq q < p : |O^{\delta}.q|).\end{aligned}$$

A distribution $(\delta, \overline{n}, \overline{p})$ is called w-balanced, $w \geq 0$ iff

$$\mathrm{Ma}(\delta) - \mathrm{Mi}(\delta) \leq w.$$

Distributions may be composed to obtain new distributions. Practical applications of composed distributions are, for example, parallel programs using different data distributions. In this way, it is possible to trade off the load balance for each individual part and to avoid expensive redistributions during computation.

2.2. *Cartesian distributions*

Distributions of multi-dimensional arrays may be modeled by Cartesian distributions. In what follows, it is assumed that an $m \times n$ matrix is distributed across processes.

Definition 4 A Cartesian distribution is defined by a Cartesian product of 1D distributions. The Cartesian product of two 1D distributions $D0 = (\delta 0, \overline{m}, \overline{M})$, $D1 = (\delta 1, \overline{n}, \overline{N})$ is defined by:

$$D0 \times D1 = (\delta 0 \times \delta 1, \overline{m} \times \overline{n}, \overline{M} \times \overline{N}),$$

where the function $\delta 0 \times \delta 1$ assigns a pair of process numbers to each array index pair.

Written formally, we have:

$$\delta 0 \times \delta 1 = (\lambda\ i, j \cdot (\delta 0.i, \delta 1.j)).$$

The Cartesian product of two 1D distributions uses a process pair as identification for a process. Cartesian distributions of matrices can be obtained by distributing the rows of the matrix independently from the columns. Since the processes number p is set, we can consider all decomposition such that $p = M * N$.

Some examples of Cartesian distributions, for $p = M * N$ processes, are:

$$\begin{aligned}
\text{linear}^2 &= (\delta^{\text{linear}}, \overline{m}, \overline{M}) \times (\delta^{\text{linear}}, \overline{n}, \overline{N}), \text{with } M = N\text{—also called block;} \\
\text{row} &= \text{linear}^2 \text{ with } N = 1; \\
\text{column} &= \text{linear}^2 \text{ with } M = 1; \\
\text{cyclic}^2 &= (\delta^{\text{cyclic}}, \overline{m}, \overline{M}) \times (\delta^{\text{cyclic}}, \overline{n}, \overline{N}), \text{with } M = N\text{—also called grid;} \\
\text{cyclic} - \text{row} &= \text{cyclic}^2 \text{ with } N = 1; \\
\text{cyclic} - \text{column} &= \text{cyclic}^2 \text{ with } M = 1.
\end{aligned}$$

2.3. *Counting communications*

During a communication, processes need values that are not available locally, values that have been assigned to some different processes, and hence have to be communicated. The distribution determines the total number of communications. It is also important that the communications are spread evenly across the processes in such way that many communications take place in parallel—if the communication network offers enough freedom to implement communication processes efficiently.

Given a program's postcondition and a distribution, the evaluation of the total number of communications, before developing the program, is possible. The program's postcondition is split in p local postconditions according to the distribution used. For each local postcondition a process, establishing it, is created and associated to. Under the hypothesis that every datum is assigned to one unique process, the total number of postconditions that refer to a particular datum is a measure of the number of communications of that datum. However, it may happen that a subexpression containing several data occurs in different postconditions. One process can compute such a subexpression and store the result in a variable, which is communicated to the other processes. And, thus, communication is reduced. This case is excluded from the following counting technique.

For the datum e, the quantity $NOcc.e$ is introduced:

$NOcc.e$ = the number of local postconditions in which e occurs.

By summing over all e, the total number of communications $NCom$ will be obtained:

$$NCom = \left(\sum e :: NOcc.e - R.e\right),$$

where

$$R.e = \begin{cases} 1, & \text{if } e \text{ occurs in the postcondition of the process that contain it;} \\ 0, & \text{otherwise.} \end{cases}$$

The value of $NCom$ is only determined by the way the program's postcondition is split up, and the distribution used. The communication complexity is bounded from below by $(NCom + p - 1)/p$, if a process can perform only one communication action at each moment.

This technique of counting communications allows a comparison of the distributions on the basis of their communication overhead. Like we mentioned before, the applicability of the technique is not possible when common subexpressions exist, but generally in the construction of the program stepwise refinement is used, and so these cases may be many times excepted. The technique is useful especially because the results that can be obtained are independent of any communication network.

To illustrate this technique, an example is given.

Example 1 (Matrix multiplication) Let us consider two matrices A and B of dimensions $m \times o$ and $o \times n$, respectively. Our goal is to compute the $m \times n$ matrix C, satisfying postcondition R:

$$R : C = A \times B.$$

We consider $p = M * N$ processes, each process being identified by an ordered pair $(s,t), 0 \leq s < M, 0 \leq t < N$. For the matrix C a Cartesian distribution $D0 \times D1$ is used, and we consider that the folowing conditions hold: $M < o$, $N < o$, $M < m$, $N < n$.

The local postconditon $R.s.t$ is:

$$R.s.t : (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge \delta 0.i = s \wedge \delta 1.j = t : c(i,j) = \left(\sum k : 0 \leq k < o : a(i,k) * b(k,j)\right).$$

Note that

$$(\forall s,t : 0 \leq s < M \wedge 0 \leq t < N : R.s.t) \Rightarrow R.$$

In order to count the number of communications, quantities $NOcc.a(i,k)$ and $NOcc.b(k,j)$ are calculated:

$$\begin{aligned} & NOcc.a(i,k) \\ = \; & |(\forall s,t : 0 \leq s < M \wedge 0 \leq t < N \wedge \delta 0.i = s \wedge (\exists j :: \delta 1.j = t) : (s,t))|. \end{aligned}$$

If the $\delta 1$ is surjective:

$$
\begin{aligned}
& NOcc.a(i,k) \\
= & \quad \{\text{definition } NOcc,\ \delta 1 \text{ is surjective}\} \\
& |(\forall s,t : 0 \le s < M \wedge 0 \le t < N \wedge \delta 0.i = s \wedge \text{true} : (s,t))| \\
= & \quad \{\text{calculus}\} \\
& N * |(\forall s : 0 \le s < M \wedge \delta 0.i = s : s)| \\
= & \quad \{\delta 0 \text{ is a function}\} \\
& N.
\end{aligned}
$$

Similarly $NOcc.b(i,k) = M$, if $\delta 0$ is surjective.

If we assume that A is distributed using a Cartezian distribution $D0 \times D2$, and B is distributed using a Cartezian distribution $D3 \times D1$, then:

$$
\begin{aligned}
& NCom \\
= & \quad \{\text{definition of } NCom\} \\
& \left(\sum i,k : 0 \le i < m \wedge 0 \le k < o : NOcc.a(i,k) - 1\right) \\
& \quad + \left(\sum k,j : 0 \le k < o \wedge 0 \le j < n : NOcc.b(k,j) - 1\right) \\
= & \quad \{\text{calculus}\} \\
& om(N-1) + on(M-1).
\end{aligned}
$$

Remark:

- For $M = N = p = 1$, $NCom = 0$ and no communications are necessary.
- $NCom$ is independent of particular choices of $\delta 0$ and $\delta 1$.
- It is possible to determine M and N, $p = M * N$ such that $NCom$ is minimal. All possible values (M,N) are integer points on the hyperbola $p = M * N$, $1 \le M, N \le p$, and the values of $NCom$ for fixed m, n, o lie on the line with a slope dependent on m/n. Hence, the minimal value for $Ncom$ depends on the ratio m/n and in particular for $m = n$, $Ncom$ has a minimal value if p is square. This result confirm the results obtained by evaluation of parallel programs for matrix multiplication, based on 1D decomposition and 2D decomposition [3].

A similar counting technique is presented in Loyens [6], but instead of the function $R.e$, the constant value 1 is used there. This means that the process that contains a datum has to always use it in its postcondition. This is almost always true; there are, however, exceptions to this, and the following example illustrates one such exception.

Example 2 (Matrix multiplication) We consider the same problem, but with the matrix A distributed with a $D0 \times D2$ distribution, and we distribute the transpose of matrix B, using a distribution $D3 \times D2$ $(M < m, M < n, N < n, N < o)$.

The local postcondition is the same as in the first case. But, now we rewrite the postcondition in a different way:

$$
\begin{aligned}
&\Big(\forall i,j : 0 \le i < m \wedge 0 \le j < n \wedge \delta 0.i = s \wedge \delta 1.j = t : \\
&\quad c(i,j) = \Big(\sum k : 0 \le k < o : a(i,k) * b(k,j)\Big)\Big) \\
&= \{\text{calculus}\} \\
&\Big(\forall i,j : 0 \le i < m \wedge \delta 0.i = s \wedge 0 \le j < n \wedge \delta 1.j = t : \\
&\quad c(i,j) = \Big(\sum v : 0 \le v < N : \Big(\sum k : 0 \le k < o \wedge \delta 2.k = v : a(i,k) * b(k,j)\Big)\Big)\Big) \\
&= \Big\{w(i,j,v) \stackrel{\text{not}}{=} \Big(\sum k : 0 \le k < o \wedge \delta 2.k = v : a(i,k) * b(k,j)\Big)\Big\} \\
&\Big(\forall i,j : 0 \le i < m \wedge \delta 0.i = s \wedge 0 \le j < n \wedge \delta 1.j = t : \\
&\quad c(i,j) = \Big(\sum v : 0 \le v < N : w(i,j,v)\Big)\Big).
\end{aligned}
$$

Therefore, the program has two stages: the first for the computation of $w(i,j,v)$ values, and the second that combines these values.

The local postcondition for the first stage is:

$$
\begin{aligned}
R0.s.t : \Big(&\forall i,j : 0 \le i < m \wedge \delta 0.i = s \wedge 0 \le j < n : \\
&w(i,j,t) = \Big(\sum k : 0 \le k < o \wedge \delta 2.k = t : a(i,k) * b(k,j)\Big)\Big).
\end{aligned}
$$

The element $a(i,k)$ only appears in the postcondition of the process containing it, and so, $NOcc.a(i,k) = 1, R.a(i,k) = 1$. The element $b(k,j)$ appears in M postconditions: $NOcc.b(k,j) = M, R.b(k,j) = 1$.

$$
\begin{aligned}
&NOcc.b(k,j) \\
&= \{\text{definition}\} \\
&|(\forall s,t : 0 \le s < M \wedge 0 \le t < N \wedge \delta 2.k = t \wedge (\exists i :: \delta 0.i = s) : (s,t))| \\
&= \{\delta 0 \text{ is surjective}\} \\
&M.
\end{aligned}
$$

Hence $NCom = on(M-1)$.

For the second stage the postcondition is:

$$
\begin{aligned}
R1.s.t : \Big(&\forall i,j : 0 \le i < m \wedge 0 \le j < n \wedge \delta 0.i = s \wedge \delta 1.j = t : \\
&c(i,j) = \Big(\sum v : 0 \le v < N : w(i,j,v)\Big)\Big).
\end{aligned}
$$

We have $NOcc.w(i,j,v) = 1$. In order to compute $NCom$ corresponding to the

second stage, we have to establish first $R.w(i,j,v)$:

$$R.w(i,j,v) = \begin{cases} 1, & \text{if } v = \delta 1.j, \\ 0, & \text{if } v \neq \delta 1.j. \end{cases}$$

Therefore $NCom = mn(N-1)$.

Thus, we obtain the following result for the whole program:

$$NCom = on(M-1) + mn(N-1).$$

Remark: If we compare the results of these two examples for matrix multiplication, we may conclude that choosing the best distribution from the communication point of view depends on the values of n and o.

3. Set distributions

When the processes number is greater than the data input size, it is desirable to assign a datum to more than one process. Also, when a data object is used in more than one computation, this kind of distributions may lead to efficient algorithms.

Definition 5 A set distribution for n data input objects on p processes is defined by a set-valued mapping $\theta : \overline{n} \multimap \overline{p}$; $\theta.i$ represents the set of processes containing the data object i.

The set distributions are characterized by:

- the number of processes containing one data object;
- the data distribution on processes.

Examples of set distributions, when $p > n$, are $\overline{\text{linear}}$ and $\overline{\text{cyclic}}$:

$$\theta^{\overline{\text{linear}}}.i = (\forall k : 0 \leq k < p/n : i(p/n) + k), \quad \text{if } n \mid p,$$
$$\theta^{\overline{\text{cyclic}}}.i = (\forall k : 0 \leq k < p/n : kn + i), \quad \text{if } n \mid p.$$

Similar properties about load imbalance, with those for the simple distributions, may be established.

We may also define a function ΘInd that allows us to obtain the processes that contain a particular datum, in a particular order. For a set distribution $\theta : \overline{n} \multimap \overline{p}$, the function ΘInd is defined as:

$$\Theta Ind : \overline{n} \times \overline{(p/n)} \to \overline{p}.$$

For $\overline{\text{linear}}$ and $\overline{\text{cyclic}}$ the ΘInd functions are:

$$\Theta Ind^{\overline{\text{linear}}}.i.k = i(p/n) + k,$$
$$\Theta Ind^{\overline{\text{cyclic}}}.i.k = k * n + i.$$

If we use a set distribution, we may formally define a broadcast, using these functions.

Example 3 (Broadcast) We consider p processes and an array $x(i : 0 \leq i < n) :$ *array of int*, $n < p$, distributed over the processes by a set distribution θ. A broadcast of a value $x(i)$ may be defined by the following parameterized communication processes:

```
C.q ::
|[ a : int;
{a = x(i) ∨ ¬(q ∈ θ.i)}
    if (q = ΘInd.i.0) →
      par u : 0 ≤ u < p ∧ ¬(u ∈ θ.i) :
        u!a
      rap
    [] ¬(q ∈ θ.i) →
      ΘInd.i.0?a
    fi
  {a = x(i)}
]|
```

There are cases when, in order to easier express the sets of the processes that contain a datum, a set distribution may be defined using a simple distribution, which is, for example, multiplicated.

3.1. *Counting communications*

The counting communications technique used for simple distributions may be also used for set distributions, with the difference that for a datum e, the value $R.e$ depends on the number of processes where e is assigned:

$$R.e = \left| \left(\sum q : 0 \leq q < p \wedge q \in \theta.e : A.q.e \right) \right|,$$

where

$$A.q.e = \begin{cases} 1, & \text{if } e \text{ occurs in the postcondition of the process } q, \\ 0, & \text{otherwise.} \end{cases}$$

Example 4 (Matrix multiplication) We consider again two matrices A and B of dimensions $m \times o$ and $o \times n$, respectively (with $p \leq m * n * o$). To compute the $m \times n$ product matrix C, the postcondition R must be satisfied:

$$R : C = A \times B.$$

We will use a set distribution, but in order to simplify the specification of the set of processes containing the same data, we choose another approach, different than the set-valued mapping.

If the data input is an $m \times n$ matrix, the processes number p is factorized such that $p = M * N * Q$, where $M \leq m$ and $N \leq n$; each process is identified by an ordered triple $(s,t,r), 0 \leq s < M, 0 \leq t < N, 0 \leq r < Q$.

For matrix C, a Cartesian distribution $D0 \times D1, D0 = (\delta 0, \overline{m}, \overline{M}), D1 = (\delta 1, \overline{n}, \overline{N})$ is used; the matrix C is replicated Q times. Another distribution $D2 = (\delta 2, \overline{o}, \overline{Q})$ is used. The matrix A is distributed using the distribution $D0 \times D2$, and is replicated N times; the matrix B^t is distributed using $D1 \times D2$, and is replicated M times.

The parallel program is split in two stages. In the first stage all processes work to compute partial results, and in the second, the partial results are combined. To compute these partial values, $M * N * Q$ partial parameterized postconditions are defined.

The local parameterized postconditon $R.s.t.0$ is:

$$R.s.t.0 : \Big(\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge \delta 0.i = s \wedge \delta 1.j = t \\ : c(i,j) = \Big(\sum r : 0 \leq r < Q : sum.i.j.r\Big)\Big),$$

where $sum.i.j.r = (\sum k : 0 \leq k < o \wedge \delta_2.k = r : a(i,k) * b(k,j))$.

Note that

$$(\forall s,t : 0 \leq s < M \wedge 0 \leq t < N : R.s.t.0) \Rightarrow R.$$

Values $sum.i.j.r$ are calculated based on the following parameterized postconditions:

$$R0.s.t.r : (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge \delta 0.i = s \wedge \delta 1.j = t \\ : w(i,j,r) = sum.i.j.r).$$

For the first stage, provided that $\delta 1$ and $\delta 0$ are surjective, the number of communications is determined by:

$$\begin{aligned} & NOcc.a(i,k) \\ = {} & |(\forall s,t,r : 0 \leq s < M \wedge 0 \leq t < N \wedge 0 \leq r < Q \wedge \delta 0.i = s \wedge \delta 2.k \\ = {} & r \wedge (\exists j :: \delta 1.j = t) : (s,t,r))| \\ = {} & N. \end{aligned}$$

Analog $NOcc.b(k,j) = M$. Because $R.a(i,k) = N$ and $R.b(k,j) = M \Rightarrow NCom = 0$.

For the second stage:

$$\begin{aligned} & NOcc.(w(i,j,r)) \\ &= |(\forall s,t : 0 \le s < M \wedge 0 \le t < N \wedge \delta 0.i = s \wedge \delta 1.j = t : (s,t))| \\ &= 1 \end{aligned}$$

and

$$R.w(i,j,r) = \begin{cases} 1, & \text{if } r = 0, \\ 0, & \text{if } r \neq 0. \end{cases}$$

Hence $NCom = mn(Q-1)$.

Remark:

- If Q is taken the smallest possible, the number of communications decreases, but the work-load of the processes increases ($p = M * N * Q$).
- $M * N$ communications can be performed in parallel, therefore, the communication complexity depends on $(m * n)/(M * N) * (Q-1)$.
- If we consider a tree-like computation algorithm for the summation of partial sums, the factor $Q-1$ will be replaced with $\log_2 Q$.
- The distribution functions have no implications on the communication number.

For matrix-matrix multiplication, this idea of replication on a third dimension is not new. But, by using these distributions the flexibility is increased—a general algorithm which does not depend on the values of m, n, o, and p, may formally be developed. We may also choose the best division of p: $p = M * N * Q$.

4. Lagrange polynomial

In this section, a parallel algorithm for the computation of Lagrange polynomial on a given value is developed. Two variants are constructed starting by selecting different types of distribution: simple and set.

4.1. *The problem*

Let $[a,b] \subset \mathbb{R}, x(i) \in [a,b], 0 \le i < m$, such that $x(i) \neq x(j)$ for $i \neq j$ and $f : [a,b] \to \mathbb{R}$.

Lagrange interpolation polynomial is defined as:

$$(L_{(m-1)}f).x = \left(\sum i : 0 \le i < m : l_i.x * f.x(i)\right),$$

where $l_i, 0 \le i < m$ are the fundamental Lagrange interpolation polynomials:

$$l_i.x = \frac{(x - x(0)) \dots (x - x(i-1))(x - x(i+1)) \dots (x - x(m-1))}{(x(i) - x(0)) \dots (x(i) - x(i-1))(x(i) - x(i+1)) \dots (x(i) - x(m-1))}$$
$$= \frac{u.x}{(x - x(i))} \cdot \frac{1}{(x(i) - x(0)) \dots (x(i) - x(i-1))(x(i) - x(i+1)) \dots (x(i) - x(m-1))}.$$

The global postcondition for the computation of the value $L_{(m-1)}f.x$ is

$$R : lx = (L_{m-1}f).x.$$

4.2. *Variant 1—simple distributions*

We consider the distribution $\delta : \overline{m} \to \overline{p}$, for the data $x(i), f(i), 0 \le i < m$. The x value is distributed to all the processes (or x is distributed to process 0, and then communicated to all other processes, by a broadcast).

Using stepwise refinement, the following stages can be considered:

1. Compute the value $u.x$.
2. Compute the fundamental polynomials $l_i.x$.
3. Compute the value $(L_{m-1}f).x$.

The local postconditions for the three stages are:

$$R0.q : ux = u.x \wedge (\forall i : 0 \le i < m \wedge \delta.i = q : xx(i) = x - x(i))$$
$$R1.q : \ (\forall i : 0 \le i < m \wedge \delta.i = q : l(i) = l_i.x)$$
$$R2.q : lx = (L_{m-1}f).x.$$

The first and the last stages represent computations of a product and a sum, so a classic algorithm for a combine computation in a tree-like manner, may be used.

The postcondition for the second stage may be rewritten in the following way:

$$R1.q : \left(\forall i : 0 \le i < m \wedge \delta.i = q : l(i) = \frac{ux}{x - x(i)} * prod.i.m\right),$$

where $prod.i.k = (\prod j : 0 \le j < k \wedge j \ne i : x(i) - x(j))$.

The communications counting leads to the conclusion that the total number of communications $(NCom = m * (p-1))$ is not influenced by the distribution function. We consider the cyclic distribution with the assumption $m\%p = 0$. To simplify the notation, we use the set of the all local indexes $O.q = (\forall i : 0 \le i < m \wedge i\%p = q : i)$.

For the derivation of the program it is necessary to define parameterized invariants. For this, a variable $k : 0 \le k < m \wedge k\%p = 0$ is introduced. So, the

invariants are:

$$P1.q : P1.0.q \wedge P1.1.q$$
$$P1.0.q : 0 \le k < m \wedge k\%p = 0$$
$$P1.1.q : (\forall i : i \in O.q : pr(i) = prod.i.k).$$

If k is initialized with 0 and $pr(i)$ with 1, the invariants hold.
The progress is made by increasing the variable k by the increment p:

$$P1.1.q(k := k + p)$$
$$= \{\text{substitution}\}$$
$$(\forall i : i \in O.q : pr(i) = prod.i.(k + p))$$
$$= \{\text{range spliting}\}$$
$$\Big(\forall i : i \in O.q : pr(i) = prod.i.k * \Big(\prod j : k \le j < k + p \wedge j \ne i : x(i) - x(j)\Big)\Big).$$

So, we get the program:

```
|[ p, m : int; x, ux : real;
  l, xx, x(i : 0 ≤ i < m) : array of real;
  par q : 0 ≤ q < p :
    |[ {Q.q : (∀i : i ∈ O.q : xx(i) = x − x(i)) ∧ ux = u.x)}
         S.q
       {R1.q : l(i) = l_i.x}
    |]
  rap
|]
```

```
S.q ::
|[ pr(i : 0 ≤ i < m) : array of real;
  a(i : 0 ≤ i < m) : array of real;
  for all i : i ∈ O.q :
    a(i) := x(i);
    pr(i) := 1
  lla rof
  {(∀i : i ∈ O.q : a(i) = x(i))}
  k := 0;   {P1.q(k := 0)}
  do (k ≠ m) →
    RestoreP1.1.q {P1.1.q(k := k + p)}
    ; k := k + p   {P1.q(k := k + p)}
  od
  for all i : i ∈ O.q :
    l(i) := (ux/xx(i))/pr(i)
  lla rof   {R1.q}
]|
```

```
RestoreP1.1.q ::
|[
  C0.q {(∀i : 0 ≤ i < m ∧ i < k + p : a(i) = x(i))}
  ;S0.q
]|
```

The description of the parameterized processes $C0.q$ and $S0.q$ is given in Appendix A.

4.2.1. Time complexity. The procedure $RestoreP1.q$ is called m/p times, and it contains a communication process and a computational process. In the communication process every process brodcasts a value to the other processes. If we consider that a brodcast lasts one communication time unit, then $T_c.p.m = m/p * p = m$. The computation process executes p operations $(-, *)$ m/p times. So, the time complexity for the second stage is:

$$T.p.m = \frac{m}{p} * \left[p * \alpha + 2p * \frac{m}{p}\right] + 2\frac{m}{p}$$
$$= m * \alpha + 2\frac{m^2}{p} + 2\frac{m}{p}.$$

4.3. *Variant 2—set distribution*

Let $p = M * M$ and all the processes are identified by a pair $(s, t), 0 \le s, t < M$.

We consider a cyclic distribution $\delta : \overline{m} \to \overline{M}$, and M permutations $\pi_t : \overline{M} \to \overline{M}, 0 \le t < M$, defined by $\pi_t.i = (i + t)\%M$.

The set distribution is defined by:

$$x(i) \in O.s.t \Leftrightarrow \delta.i = \pi_t.s \Leftrightarrow i\%M = (s + t)\%M,$$

where $O.s.t$ is the set of data elements that are assigned to the process (s, t).

For $m = 9$ and $M = 3$ the data distribution is shown in the Figure 1.

There are again three stages defined by the following local postconditions:

$$R0.s.t : (ux = u.x \wedge (\forall i : i \in O.s.t : xx(i) = x - x(i)))$$
$$R1.s.t : t \neq 0 \vee (\forall i : i \in O.s.t : l(i) = l_i.x)$$
$$R2.s.t : (lx = (L_{m-1}f).x).$$

The first and the last stages may be computed using a tree-like computation, with slight modification over the classic one. The difference is that a different numbering of the processes is used.

s\t	0	1	2
0	x_0, x_3, x_6	x_1, x_4, x_7	x_2, x_5, x_8
1	x_1, x_4, x_7	x_2, x_5, x_8	x_0, x_3, x_6
2	x_2, x_5, x_8	x_0, x_3, x_6	x_1, x_4, x_7

Figure 1. The data distribution for $m = 9$ and $M = 3$.

For example, for *ux* computation, we may consider the following derivation:

$$\begin{aligned}
&\Big(\prod i : 0 \le i < m : xx(i)\Big)\\
&\quad = \{\forall i, \exists!(s,t) \text{ such that } i\%M\\
&\quad = (s+t)\%M \wedge i\%M^2 = s * M + (s+t)\%M\}\\
&\Big(\prod s,t : 0 \le s,t < M : \Big(\prod i : 0 \le i < m \wedge i\%M\\
&\quad = (s+t)\%M \wedge i\%M^2 = s * M + (s+t)\%M : xx(i))\Big)\Big)\\
&\quad = \{O.s.t = (\forall i : 0 \le i < m \wedge i\%M = (s+t)\%M : i)\}\\
&\Big(\prod s,t : 0 \le s,t < M : \Big(\prod i : i \in O.s.t \wedge i\%M^2\\
&\quad = s * M + (s+t)\%M : xx(i)\Big)\Big).
\end{aligned}$$

The postconditions for the partial products are:

$$R01.s.t : ux.s.t = \Big(\prod i : i \in O.s.t \wedge i\%M^2 = s * M + (s+t)\%M : xx(i)\Big).$$

So, we can conclude that

$$ux = \Big(\prod i : 0 \le i < m : xx(i)\Big) = \Big(\prod s,t : 0 \le s,t < M : ux.s.t\Big).$$

The invariant is constructed using a variable k, which is initialized first with $s \cdot M + (s+t)\%M$; the progress is made by increasing the variable k with M^2. The final value is obtained by multiplication of the partial products. So, the time complexity is the same as for the variant 1.

This derivation can be also obtained starting from the definition of the corresponding ΘInd function: $\Theta Ind.i.k = (k, (i-k)\%M)$; in this case, the corresponding partial postconditions are:

$$R01.s.t : ux.s.t = \Big(\prod i : 0 \le i < m \wedge \Theta Ind.i.((i/M)\%M) = (s,t) : xx(i)\Big),$$

which are equivalent with those written above.

In what follows, we discuss about the second stage, in more detail.

As for matrix-matrix multiplication, we consider two sub-stages: one for partial computations, and one for combining the partial computations.

Each row $(s,.)$ computes the values $l(i), \forall i : 0 \le i < m \wedge \delta.i = s$.

The postcondition $R1.s.t$ may be rewritten as:

$$R1.s.t : t \neq 0 \vee (\forall i : i \in O.s.t : l(i) = ux/xx(i) * 1/prod.i.m),$$

where $prod.i.m = (\prod j : 0 \leq j < m \wedge i \neq j : (x(i) - x(j)))$.

To compute the products $prod.i.m$ we split them in M products. Each of these subproducts corresponds to the set of elements assigned to a process.

So, we rewrite the products $prod.i.m$ as:

$$\begin{aligned}
& prod.i.m \\
&= \{\text{range spliting}\} \\
& \left(\prod t : 0 \leq t < M : \left(\prod j : j \in O.s.t \wedge i \neq j : (x(i) - x(j))\right)\right) \\
&= \{(j \in O.s.t \Leftrightarrow 0 \leq j < m \wedge j\%M = (s+t)\%M) \wedge s = \delta.i\} \\
& \left(\prod t : 0 \leq t < M : \left(\prod j : 0 \leq j < m \wedge \delta.j = \pi_t.(\delta.i) \wedge i \neq j : (x(i) - x(j))\right)\right) \\
&= \left\{parprod.i.t \stackrel{\text{not}}{=} \left(\prod j : 0 \leq j < m \wedge \delta.j = \pi_t.(\delta.i) \wedge i \neq j : (x(i) - x(j))\right)\right\} \\
& \left(\prod t : 0 \leq t < M : parprod.i.t\right).
\end{aligned}$$

The values $prod.i.m$ are obtained using a tree-like computation on each row.

The invariants for partial products computations are defined by introduction of the variable k, which is incremented by M:

$$\begin{aligned}
& P1.s.t : P1.0.s.t \wedge P1.1.s.t \\
& P1.0.s.t : 0 \leq k < m \wedge k\%M = 0 \\
& P1.1.s.t : (\forall i : 0 \leq i < k \wedge \delta.i = s : ppr(i) = parprod.i.t).
\end{aligned}$$

The algorithm for partial products computation is defined by the following parameterized processes:

```
S.s.t ::
|[ppr(i : 0 ≤ i < m): array of real;
  a(i : 0 ≤ i < m): array of real;
  for all i : i ∈ O.s.t:
    a(i) := x(i);
    ppr(i) := 1
  lla rof
  {(∀i : i ∈ O.s.t : a(i) = x(i))}
  k := 0;   {P1.q(k := 0)}
  do (k ≠ m)→
    RestoreP1.1.s.t {P1.1.s.t(k := k + M)}
    ;k := k + M {P1.s.t(k := k + M)}
  od
]|
```

$RestoreP1.1.s.t$ ::
|[
  $C0.s.t$ $\{a(s+k) = x(s+k)\}$
  ;$S0.s.t$
  $\{ppr(s+k) = (\prod i : i \in O.s.t \wedge i \neq s+k : a(s+k) - a(i))\}$
]|

The $C0.s.t$ and $S0.s.t$ processes are presented in Appendix B.

4.3.1. Time complexity. For this variant, the time complexity for the second stage is:

$$T.p.m = \frac{m}{M}\left[\alpha + 2\frac{m}{M} + \log_2 M(\alpha+1)\right] + 2\frac{m}{M}$$
$$= 2\frac{m^2}{p} + \frac{m}{M}(\log_2 M + 2) + \alpha\frac{(1+\log_2 M)}{M}m.$$

The communication complexity is lower than that for the previous algorithm.

Remark:

- Both variants are built starting from specifications and using correct derivation rules.
- Different types of distributions lead to different algorithms.
- The second algorithm can be used in both cases: $p \leq m$, or $p > m$ and it have a lower communication complexity.
  If $p > m$ the advantage of using the second algorithm is obvious.
  If $p \leq m$, the comparison between time complexities in the two cases leads to the conclusion that

  $$\alpha > 3.5 \text{ (true in many case)}$$
  $$\Rightarrow T_{\text{set-distribution}} < T_{\text{simple-distribution}}, \quad \forall M \geq 4.$$

  If M is greater, the inequality is true even if α is smaller.
- If we do not consider an ideal communications network (if we consider a hypercube, or a mesh), the second algorithm is even much efficient.

5. Conclusions

Static distributions of arrays and matrices, and their implications in the construction of parallel programs are discussed. We have defined a new kind of distributions, based on set-valued mappings.

Counting communications before developing the program enables the evaluation of the distributions impact on the resulted program. This technique allows us to choose the best distribution from the beginning. We use matrix-matrix multiplication

case to emphasize this. The obtained results are independent of any communication network.

The algorithm for Lagrange interpolation polynomial developed starting from set distribution is better than the algorithm based on simple distribution.

If the number of processes is greater than the data input size, it is more convenient using set distribution. Even if the processors number does not exceed the data input size, we may define more processes than processors, and map more processes to one processor; that leads to overlapping computation and communication.

Generally, if each datum appears more than once in the local postconditions, using set distributions we may get better results than using simple distributions. The data replication was used before, but not in a formalized way. This formalism introduced here, based on set-valued mappings, helps deriving parallel programs in a formal way, and also allows the evaluation of communication costs before program development; this leads to the possibility of choosing the best distribution.

Also, the applications are not always so regular, and using set distributions does not mean always adding a new dimension in the organization of the processes.

We conclude that the distributions determine the construction of parallel programs, and the set distributions can lead to efficient and general parallel algorithms.

Appendix A

```
C0.q ::                                  | S0.q ::
|[                                       | |[ for all i : i∈O.q :
  par u : 0 ≤ u < p ∧ u ≠ q :            |      for all u : 0 ≤ u < p :
    u!a(k + q) {transmission}            |      if (i ≠ k + u)→
  rap                                    |        pr(i) := pr(i) * (a(i) − a(k + u))
  ;for all u : 0 ≤ u < p ∧ u ≠ q :       |      fi
    v?a(k + u) {reception}               |      lla rof
lla rof                                  |   lla rof
]|                                       | ]|
```

Appendix B

```
C0.s.t ::                                | S0.s.t ::
|[                                       | |[ ppr(s + k) := 1;
  if (t = 0)→                            |   for all j : j∈O.s.t :
    par v : 0 < v < M :                  |     if (j ≠ s + k)→
      (s, v)!a(k + s)                    |       ppr(s + k) := ppr(s + k) * (a(s + k) − a(j))
    rap                                  |     fi
  [ ](t ≠ 0)→                            |   lla rof
    (s, 0)?a(k + s)                      | ]|
  fi
]|
```

References

1. R. C. Agarwal, F. G. Gustavson, S. M. Balle, M. Joshi, and P. Palkar. *A High Performance Matrix Multiplication Algorithm for MPPs*, PARA'95, pp. 1–7.
2. E. W. Dijkstra. *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
3. I. Foster. *Designing and Building Parallel Programs*, Addison-Wesley, 1995.
4. L. Barbara and J. Guttag. *Abstractions and Specification in Program Development*, Massachusetts Institute of Technology, 1986.
5. C. A. R. Hoare. *An Axiomatic Basis for Computer Programming*, Communications of the ACM, 12(10):576–580, 1969.
6. L. D. Loyens. *A Design Method for Parallel Programs*, Technische Universiteit Eindhoven, 1992.
7. C. Morgan. *Programming from Specifications*, Prentice Hall, 1990.
8. D. B. Skillicorn and D. Talia. *Models and Languages for Parallel Computation*, ACM Computer Surveys, 30(2):123–136, June 1998.
9. J. R. Smith. *The Design and Analysis of Parallel Algorithms*, Oxford University Press, 1993.
10. E. F. Van de Velde. *Concurrent Scientific Computing*, Spring-Verlag, New-York Inc., 1994.
11. H. S. Wilf. *Algorithms and Complexity*, Mason & Prentice Hall, 1985.