Programming Research Group

A SCHEME FOR THE BSP SCHEDULING OF GENERIC LOOP NESTS

Radu Calinescu

PRG-TR-26-97

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

A Scheme for the BSP Scheduling of Generic Loop Nests

Radu Calinescu

August 1997

Abstract

This report presents a scheme for the bulk-synchronous parallel (BSP) scheduling of generic, untightly nested loops. Being targeted at the BSP model of computation, the novel parallelisation scheme yields parallel code which is scalable, portable, and whose cost can be accurately analysed. The scheme comprises three stages: data dependence analysis and potential parallelism identification, data and computation partitioning, and synchronisation and communication generation. New algorithms tackling each of the three stages are presented in the report, together with an algorithm for assessing the cost of the resulting BSP schedules.

1 Introduction

The prohibitive costs of parallel software design have led to an ever increasing interest in the automatic parallelisation of existing sequential code. As a result, remarkable advances have been made in areas such as data dependence analysis [6, 7, 24], code transformation [4, 22, 26], and potential parallelism identification [3, 18, 19]. Based on these theoretical advances, many parallelising compilers and tools have been devised within the last decade or so [2, 12, 15, 27, 29, 30]. Although the parallel code generated by these automatic parallelisers has shown to be effective in many cases, it nevertheless lacks portability and predictability.

Unlike the above mentioned approaches, the current report describes a new parallelisation strategy which succeeds in automatically generating parallel code which is scalable, portable, and whose cost can be accurately analysed. This is achieved by employing the bulk-synchronous parallel (BSP) model of computation [25, 20] as a target platform for the parallelisation and scheduling of generic loop nests.

The BSP model of computation provides a unifying framework for the development of both scalable parallel architectures and portable parallel software. A BSP computer comprises a set of processor-memory pairs, a communication network providing point-to-point message delivery, and a mechanism for the efficient barrier-style synchronisation of the processors. Three parameters fully characterise a BSP machine: p, the number of processor-memory components; L, the synchronisation periodicity, or the cost of a barrier synchronisation of all processors; and g, the ratio between the computation throughput of

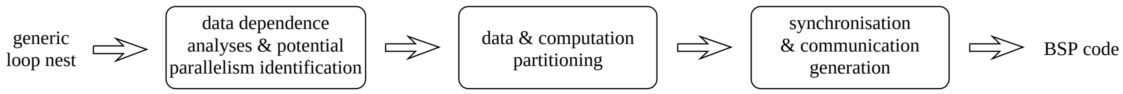

Figure 1: The BSP parallelisation scheme.

the processors, and the communication throughput of the router. A BSP program consists of a sequence of supersteps, each ended by a barrier synchronisation of all processors. In any superstep, the processors independently execute operations on locally held data, and/or initiate read/write requests for non-local data. These requests are guaranteed to be satisfied after the barrier synchronisation that ends the superstep. The BSP cost model is compositional: the cost of a BSP program is simply the sum of the costs of its constituent supersteps. Several equivalent expressions may be used for the cost of an individual superstep S [20]. In this report, we consider the expression which adds up the costs of synchronisation, computation, and communication:

$$cost(S) = L + w + gh, \tag{1}$$

where w represents the maximum number of local operations executed by any processor during superstep S, and h is the maximum number of words received or sent by any processor during superstep S.

Following the successful design and implementation of many BSP algorithms and applications [20, 25], recent research work studied the scheduling of perfect loop nests on BSP computers [21, 11, 8, 9, 10]. The current report extends this work by introducing a scheme for the BSP scheduling of generic, untightly nested loops whose loop bounds and referenced array subscripts are affine functions of the surrounding loop indices. The new parallelisation scheme (Figure 1) comprises three stages.

In the first stage, the dependence graph [7] of the loop nest is built based on the direction vector [28] representation of data dependences, and an extension of the parallelisation algorithms in [1, 2, 16, 29] is used to reveal the potential parallelism of the original loop nest. The extended algorithm integrates loop interchange into the basic algorithms in [1, 2, 16, 29], and employs an early termination test to eliminate the negative effects of unnecessary loop distribution. This early termination results in significant reductions in the overall synchronisation cost, and is done at the expense of sequentialising secondary statements in the loop nest (i.e., statements whose parallelisation is not essential for achieving a nearly optimal speedup). Furthermore, the new algorithm is targeted towards the identification of coarse-grained potential parallelism, parallelism which is best suited for the BSP setting.

In the second stage, the data and the computation are partitioned among the processors of the BSP computer. First, the automatic selection of a data distribution which partitions the arrays referenced within the loop body in dimensions corresponding to the parallel loops is attempted. When this is not possible, user expertise is employed to select an appropriate data distribution. Then, the "owner computes" rule [5, 14, 15, 30] is used to mechanically generate the computation partitioning.

Finally, in the third stage, the complete BSP code is generated by inserting appropriate calls to BSP synchronisation and communication primitives into the intermediate parallel code of stage two. Thus, the uses (i.e., the references that read the value of a variable) of each statement are examined to identify references to non-local data, and corresponding coalesced and vectorised [5, 14] calls to communication primitives are issued at the outermost possible level in the loop nest. At the same time, the dependence information is used to automatically insert appropriate barrier synchronisations into the code, a feature unique to our parallelisation algorithm.

The rest of the report is structured as follows. The three stages of the BSP parallelisation scheme are described in Sections 2 through 4. Then, in Section 5, we present an algorithm that computes the cost of a BSP schedule generated by the parallelisation scheme. Finally, several concluding remarks and a discussion of the further work directions are included in Section 6.

2 Data dependence analysis and potential parallelism identification

The first stage of the BSP parallelisation scheme aims at rewriting the original loop nest in a form that reveals the potential parallelism of the sequential code. In order to ensure that the transformed code yields the same results as the original one, the *data dependences* of the loop nest must be obeyed (i.e., the execution order of interdependent statements must be preserved).

Given two statements S_x and S_y, a data dependence exists between the two statements if both S_x and S_y refer the same variable, at least one of them modifying the value of the variable. There are three types of data dependence:

- *flow* (or *true*) *dependence* (denoted $S_x \delta S_y$), when statement S_x modifies the value of the commonly referenced variable, and then S_y uses this value;
- *antidependence* (denoted $S_x \overline{\delta} S_y$), when statement S_x reads the value of a variable later modified by statement S_y;
- *output dependence* (denoted $S_x \delta^o S_y$), when both statements (first S_x, and then S_y) modify the variable.

A generic data dependence between the two statements is denoted $S_x \delta^* S_y$.

When a statement is part of a loop body, data dependences are analysed in terms of *statement instances*. A statement instance represents the execution of a statement for a fixed value of the surrounding loop indices. Since distinct instances of a statement may refer the same variable, the statements involved in a data dependence need not to be different. For the loop nest in Figure 2 for example, the instance of statement S_1 corresponding to $(i, j) = (i_1, j_1)$ modifies $a[i_1, j_1]$, which is later read by the instance of S_1 corresponding to the iteration point $(i, j) = (i_1 + 1, j_1)$. Accordingly, a flow dependence $S_1 \delta S_1$ exists between the two instances of statement S_1.

```
l1     for i=1,n-1 do
l2       for j=0,n-1 do
S1:        a[i,j]=f1(a[i-1,j],b[i-1,i,j])
l3       for j=0,n-2 do
l4         for k=0,n-2 do
S2:          b[i,j,k]=f2(b[i-1,j+1,k],b[i-1,j,k+1])
```

Figure 2: Example of a generic, untightly nested loop.

In order to efficiently identify the parallelism available in a loop nest, we need a way of characterising the data dependences between two statement instances. For the purpose of our parallelisation scheme, the most appropriate means for describing a data dependence is the *direction vector* representation of Wolfe [28]. Thus, if S_x, S_y are two statements surrounded by m common loops indexed by i_1, i_2, $\ldots$, i_m, and a data dependence exists between the instance of S_x with $(i_1, i_2, \ldots, i_m) = (x_1, x_2, \ldots, x_m)$ and the instance of S_y with $(i_1, i_2, \ldots, i_m) = (y_1, y_2, \ldots, y_m)$, the direction vector of this dependence is $v \in \{'<', '=', '>'\}^m$, where

$$\forall j : 1..m \bullet v_j = \begin{cases} '<' & \text{if } x_j < y_j \\ '=' & \text{if } x_j = y_j \\ '>' & \text{if } x_j > y_j \end{cases}$$

For the flow data dependence $S_1 \delta S_1$ of the loop nest in Figure 2 for instance, the associated direction vector is $v = (<, =)$. This can be specified in a more compact way using the notation $S_1 \delta_{(<,=)} S_1$.

No new data dependence test (i.e., test for detecting the existence of a data dependence between two statement instances) is proposed in this report. We rely instead on the broad collection of dependence tests devised within the last two decades [2, 6, 7, 23, 28]. These tests can be applied when the variable commonly referenced by the statement instances whose interdependence is investigated is an array whose subscripts are affine functions of the surrounding loop indices. Some of the dependence tests are approximate and therefore conservative (i.e., whenever independence cannot be proved, dependence is unconditionally assumed), while others are exact (but computationally intensive).

After the data dependences of the loop nest (i.e., the dependences between all pairs of statements in the loop) are found, the dependence information is organised as a *dependence graph* [7]. This represents a directed graph comprising a vertex for each statement in the loop nest, and an edge between each pair of vertices that stand for interdependent statements. Each such edge is labeled with the type and direction vectors of the associated data dependence. The dependence graph for the loop nest in Figure 2 is depicted in Figure 3.

Following the construction of the dependence graph of a loop nest, the information in this graph permits the identification of the potential parallelism of the loop. The remainder of this section presents an algorithm performing this task. This algorithm

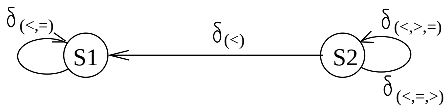

Figure 3: The dependence graph of the loop nest in Figure 2.

extends previous parallelisation algorithms by Kuck [16], Allen, Callahan and Kennedy [1, 2], and Zima and Chapman [29]. The novelty of our algorithm consists in the integration of loop interchange into the existing algorithms. This integration translates into an important increase in the granularity of the resulting potential parallelism, with beneficial effects in the BSP setting. At the same time, the new parallelisation algorithm employs an early termination test to eliminate the negative effects of unnecessary loop distribution.

First, we will introduce several theoretical results that underlie the various steps of the parallelisation algorithm. These results are introduced by the following two theorems.

Theorem 1 *Let S_x, S_y be two (not necessarily distinct) statements surrounded by m common loops. Assume that a data dependence $S_x \delta_v^* S_y$ with $v_j =$ '$<$' for some $1 \leq j \leq m$ exists between the two statements. Then, if the j-th loop surrounding S_x and S_y is sequentially executed in the outermost position of the m loops, all the restrictions imposed by this dependence are satisfied.*

Proof Consider two generic iteration points $x = (x_1, x_2, \dots, x_m)$ and $y = (y_1, y_2, \dots, y_m)$ such that the instance of S_x for the first iteration point refers the same data as the instance of S_y for the second iteration point. As the j-th component of the direction vector characterising the considered data dependence is '$<$', we necessarily have $x_j < y_j$.

Assume now that the j-th common loop surrounding S_x and S_y is brought into the outermost position and executed sequentially, with the other loops executed in parallel, in an arbitrary order. Then, since the iterations of the j-th loop are executed sequentially, the iteration point x will be executed before the iteration point y by the transformed loop nest. Hence, the transformed loop nest executes any interdependent iteration points in the correct order, and the restrictions imposed by the considered dependence are satisfied. $\square$

Theorem 1 provides the basis for the parallelisation of the inner loops of a loop nest when one of the loops which *carries* the data dependence is sequentialised in the outermost position. Consider for instance the following loop nest

```
         for i=1,n-1 do
           for j=2,n-1 do
S1:          a[i,j]=f(a[i-1,j-2])
```

Since both loops carry the only data dependence $S_1\delta_{(<,<)}S_1$ of this loop nest, the sequential execution of either loop in the outermost position allows the parallel execution of the other loop.

Theorem 2 *Consider again two statements S_x, S_y surrounded by m common loops. Assume that a data dependence $S_x \delta_v^* S_y$ with $v_j =$ '$=$' for some $1 \leq j \leq m$ exists between the two statements. Then, if the j-th loop surrounding S_x, S_y is concurrently executed in a randomly chosen position, and the other $m-1$ loops are executed such as to satisfy the restrictions imposed by the data dependence $S_x \delta^*_{(v_1,\ldots,v_{j-1},v_{j+1},\ldots,v_m)} S_y$, the constraints enforced by the original dependence on the entire loop nest are also satisfied.*

Proof Let $x = (x_1, x_2, \ldots, x_m)$ and $y = (y_1, y_2, \ldots, y_m)$ be two generic iteration points such that the data dependence occurs between the instance of S_x for the first iteration point and the instance of S_y for the second iteration point. Then, in any legal transformation of the loop nest, the execution order of two iteration points must remain unchanged (i.e., first x, then y).

Consider now that the j-th common loop surrounding S_x and S_y is brought into a randomly chosen position and executed in parallel. According to the hypothesis of the theorem, $x_j = y_j$. Then, the fact that iterations points with different j-th components are executed simultaneously does not influence the execution order of the iteration points x and y. Instead, the order in which x and y are performed depends on the other $m-1$ loops, i.e., it is given by the execution order of $x' = (x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_m)$ and $y' = (y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_m)$ when only the other $m-1$ loops are taken into account. As long as these loops are executed such as to respect the data dependence $S_x \delta^*_{(v_1,\ldots,v_{j-1},v_{j+1},\ldots,v_m)} S_y$, x' is performed before y'. Hence, the initially considered iteration points are also executed in the correct order, and the restrictions imposed by the original data dependence are obeyed.
□

Typically, the position in which a loop that satisfies the hypothesis of Theorem 2 is worth parallelising is the outermost level of its loop nest. Therefore, the second theorem provides a way of increasing the granularity of the potential parallelism of a loop nest by parallelising certain loops in the outermost position. The application of the theorem is illustrated by the following example:

```
            for i=1,n-1 do
              for j=0,n-1 do
                for k=0,n-2 do
S1:               a[i,j,k]=f(a[i-1,j,k+1])
```

The only data dependence of this loop nest is $S_1 \delta_{(<,=,>)} S_1$. Since the second component of the direction vector is '=', the loop indexed by j can be parallelised in the outermost position as long as the execution of the other loops does not violate the constraints imposed by the data dependence $S_1 \delta_{(<,>)} S_1$.

Before actually presenting the potential parallelism identification algorithm, let us make clear which are the goals of this algorithm. Consider the following example of a loop nest, in which the problem size n is much greater than p, the number of processors of the BSP computer:

```
for i=0,n-1 do
  S1
  for j=0,n-1 do
    S2
    for k=0,n-1 do
      S3
```

The sequential cost of executing the loop nest is $O(n^3)$, so what one expects from a p-processor BSP version of this loop nest is an execution cost of $O(n^3/p)$. Obviously, this can only be obtained if the execution of statement S_3—which is called the *main statement* of the loop nest—is parallelised. Furthermore, the fact that statements S_1 and, perhaps, S_2 are executed in parallel or not is irrelevant for the successful parallelisation of the loop nest as a whole. On the other hand, as we will show in Section 3, each main statement places constraints on the data partition computed in the second stage of the parallelisation scheme. Therefore, the chances of automatically detecting an appropriate data distribution decrease when the number of main statements is large. Consequently, a parallelisation algorithm must concentrate on the identification of potential parallelism corresponding to the main statements of a loop nest, possibly neglecting the less significant parts of the loop.

A second feature that must characterise a good parallelisation algorithm is illustrated by the following loop nest:

```
for i=0,n-1 do
  for j=0,2 do
    for k=0,n-1 do
      S1
```

Assuming again that the problem size n is much greater than the number of processors p, any successful BSP version of this loop nest must be able to concurrently execute the loop body for different iteration points along direction(s) i or/and k of the iteration space. Indeed, a parallelisation of the loop indexed by j is of little use in this case. Therefore, a parallelisation algorithm must focus on those loops whose iteration space size depends on the problem size—these are called the *main loops* of the loop nest—rather than on the loops whose index bounds are (small) constants.

Finally, assume that a main statement S of a loop nest is surrounded by several main loops which iterate from 0 to $n-1$, where n is the problem size, and suppose that only one of these main loops can be parallelised. Then, if the parallel loop is in the innermost position, a synchronisation is required after every n executions of the statement S. As a result, the BSP program induced by this parallelisation could be efficient only if the problem size is much greater than the synchronisation cost L, which is a very restrictive constraint. If, on the other hand, the parallel loop can be brought into a middle position (or, even better, in the outermost position), barrier synchronisations will be required far less often, and an efficient BSP schedule is possible. Accordingly, the parallelisation algorithm must pay attention to the parallelisation of loops in the outermost possible position, i.e., to the extraction of the coarsest possible parallelism.

The observations made so far are taken into account by the parallelisation algorithm in Figure 4. This algorithm identifies the potential parallelism of a generic loop nest $\mathcal{L}$ using the information in its dependence graph $\mathcal{G}$. In doing so, the algorithms attempts to parallelise the execution of a maximum number of main loops surrounding main statements of $\mathcal{L}$. Therefore, the set of main statements *main_stmts* and the set of main loops *main_loops* are provided as the third and the fourth parameters of the algorithm.

The algorithm (function `parallelise`) works as follows. In a first stage, the generic loop nest $\mathcal{L}$ is parallelised as a whole, without using loop distribution. The resulting parallel version of the loop nest is accepted—and the algorithm terminates—if each main statement is completely parallelised, i.e., if each main loop whose body contains at least one main statement is parallelised. The algorithm also terminates when $\mathcal{L}$ comprises a single statement, since in this case no further loop distribution is possible.

When none of the termination conditions is fulfilled, the algorithm enters a second stage, in which loop distribution is employed as in [29], in an attempt to increase the amount of available parallelism. First, the strongly connected components (or π-blocks [2, 17]) of $\mathcal{G}$ are identified. Then, if $\mathcal{G}$ comprises several strongly connected components, the parallelisation algorithm is recursively applied to each such component, in topologically sorted order. The distributed parallel version of the loop nest ($\mathcal{PL}2$) is obtained by concatenating the results of the successive applications of the algorithm.

If, on the other hand, $\mathcal{G}$ comprises a single π-block, the distributed parallel version of $\mathcal{L}$ is obtained by keeping the outermost loop in the parallelised "as a whole" version of the loop nest unchanged, and applying the algorithm recursively to the body of this loop. Before re-applying the algorithm, however, the edges that stand for dependences carried by the outermost loop are eliminated from $\mathcal{G}$. (Notice that if the outermost loop is parallel, it carries no dependence, so no test is needed to check whether the outermost loop is sequential or parallel.)

The result returned by the parallelisation algorithm is the best choice between $\mathcal{PL}1$, the parallelised "as a whole" version of $\mathcal{L}$, and $\mathcal{PL}2$, the distributed parallel version of $\mathcal{L}$. Thus, $\mathcal{PL}2$ is chosen if it brings extra parallelism, and $\mathcal{PL}1$ if the potential parallelism revealed by the two parallel versions of the original loop nest is identical.

For the parallelisation of a loop nest as a whole (function `parallelise_as_a_whole`), Theorems 1 and 2 are applied to the data dependences encoded by the dependence graph $\mathcal{G}$. First, the outermost perfectly nested loops of the loop nest are identified, and parallelised (function `parallelise_outermost_loops`; as an exception, the parameter $\mathcal{G}$ of this function is an input-output parameter; all other parameters are input only). Then, the algorithm is applied recursively to each component belonging to the body of these loops.

Finally, the parallelisation of a sequence of outermost loops is an iterative process working as follows. First, the algorithm attempts to find main loops that satisfy Theorem 2 and can therefore be parallelised in the outermost position. If such loops exist, they are brought in the outermost position and changed into parallel loops. Otherwise, an outermost sequential loop is chosen from the loops that satisfy Theorem 1 (at least one such loop, the current outermost loop, exists). The choice is made such as to maximise the number of parallel loops in the next iteration of the algorithm, a secondary loop

```
parallelise($\mathcal{L}$, $\mathcal{G}$, $main\_stmts$, $main\_loops$)
  $\mathcal{PL}1$=parallelise_as_a_whole($\mathcal{L}$, $\mathcal{G}$, $main\_loops$)
  if all main loops surrounding main statements in $\mathcal{PL}1$ are parallel
  or $\mathcal{L}$ comprises a single statement
    return $\mathcal{PL}1$
  identify the strongly connected components of $\mathcal{G}$
  if $\mathcal{G}$ has several such components
    $\mathcal{PL}2 = \{\}$
    for each strongly connected component $\mathcal{C}$, in topologically sorted order do
      $\mathcal{L}' \leftarrow$ reduction of $\mathcal{L}$ to the statements in $\mathcal{C}$
      $\mathcal{G}' \leftarrow$ reduction of $\mathcal{G}$ to the vertices in $\mathcal{C}$
      $\mathcal{PL}2 = \mathcal{PL}2 +\!\!+$ parallelise($\mathcal{L}'$, $\mathcal{G}'$, $main\_stmts$, $main\_loops$)
  else  /* $\mathcal{G}$ has a single strongly connected component */
    $\mathcal{L}' \leftarrow$ body of $loop$, the outermost loop in $\mathcal{PL}1$
    $\mathcal{G}' \leftarrow \mathcal{G}$ without the edges corresponding to dependences carried by $loop$
    $\mathcal{PL}2 \leftarrow loop +\!\!+$ parallelise($\mathcal{L}'$, $\mathcal{G}'$, $main\_stmts$, $main\_loops$)
  return best_choice($\mathcal{PL}1$, $\mathcal{PL}2$)

parallelise_as_a_whole($\mathcal{L}$, $\mathcal{G}$, $main\_loops$)
  if $\mathcal{L}$ is a simple statement
    return $\mathcal{L}$
  identify $loops$, the set of outermost loops surrounding the statements in $\mathcal{L}$
  $\mathcal{PL} \leftarrow$ parallelise_outermost_loops($loops$, $\mathcal{G}$, $main\_loops$)
  for each component $\mathcal{L}'$ in the loop body of $loops$ do
    $\mathcal{G}' \leftarrow$ reduction of $\mathcal{G}$ to the stmts in $\mathcal{L}'$
    $\mathcal{PL} \leftarrow \mathcal{PL} +\!\!+$ parallelise_as_a_whole ($\mathcal{L}'$, $\mathcal{G}'$, $main\_loops$)
  return $\mathcal{PL}$

parallelise_outermost_loops($loops$, $\mathcal{G}$, $main\_loops$)
  $V \leftarrow$ the set of non-null direction vectors corresponding to the edges of $\mathcal{G}$
  $\mathcal{PL} \leftarrow \{\}$
  while $loops \neq \{\}$ do
    identify $par\_loops$, the set of main loops from $loops$ with the property that
    their associated components in any direction vector in $V$ are '='
    if $par\_loops \neq \{\}$ then
      $\mathcal{PL} \leftarrow \mathcal{PL} +\!\!+$ parallelised $par\_loops$
      $loops \leftarrow loops \setminus par\_loops$
    else
      identify $seq\_loops$, the set of loops from $loops$ with the property that their
      associated components in any direction vector in $V$ are '=' or '<'
      choose $seq\_loop \in seq\_loops$ such that the elimination from $V$ of the direction
      vectors whose component corresponding to $seq\_loop$ is '<' permits the
      parallelisation of most main loops in the next step; among several
      candidate loops, prefer a secondary loop
      $\mathcal{PL} \leftarrow \mathcal{PL} +\!\!+ seq\_loop$
      $loops \leftarrow loops \setminus \{seq\_loop\}$
      $V \leftarrow V \setminus$ direction vectors whose component corresponding to $seq\_loop$ is '<'
      $\mathcal{G} \leftarrow \mathcal{G} \setminus$ edges corresponding to the eliminated direction vectors
  return $\mathcal{PL}$
```

Figure 4: The potential parallelism identification algorithm.

```
l1    for i=1,n-1 do
l3      forall j=0,n-2 do in parallel
l4        forall k=0,n-2 do in parallel
S2:         b[i,j,k]=f2(b[i-1,j+1,k],b[i-1,j,k+1])
l2    forall j=0,n-1 do in parallel
l1      for i=1,n-1 do
S1:       a[i,j]=f1(a[i-1,j],b[i-1,i,j])
```

Figure 5: The potential parallelism of the loop nest in Figure 2.

taking precedence over a main loop that would reveal the same amount of parallelism in the next iteration. After establishing the outermost loop(s), they are eliminated from the set of loops to be parallelised, and the algorithm is iterated until no loop is left in this set. When in the current iteration of the algorithm a sequential loop is selected, the dependences carried by this loop are eliminated before proceeding to the next iteration.

To illustrate the usage of the parallelisation algorithm, let us identify the potential parallelism of the loop nest $\mathcal{L}$ in Figure 2 by calling

$$\texttt{parallelise}(\mathcal{L},\mathcal{G},\{S1,S2\},\{l1,l2,l3,l4\})$$

where $\mathcal{G}$ represents the dependence graph in Figure 3. In the first stage, the loop nest is parallelised as a whole, yielding the following undistributed parallel version of $\mathcal{L}$:

```
l1    for i=1,n-1 do
l2      forall j=0,n-1 do in parallel
S1:       a[i,j]=f1(a[i-1,j],b[i-1,i,j])
l3      forall j=0,n-2 do in parallel
l4        forall k=0,n-2 do in parallel
S2:         b[i,j,k]=f2(b[i-1,j+1,k],b[i-1,j,k+1])
```

Indeed, all the data dependence are carried by loop $l1$, and the other loops can be parallelised once loop $l1$ is sequentialised in the outermost position.

As one of the main loops surrounding $S1$ and $S2$ is sequential, the algorithm proceeds to the second stage. In this stage, the two strongly connected components of $\mathcal{G}$ are parallelised in topologically sorted order, i.e., first the π-block comprising statement $S2$, then the π-block comprising $S1$. For the first strongly connected component, the algorithm yields

```
l1    for i=1,n-1 do
l3      forall j=0,n-2 do in parallel
l4        forall k=0,n-2 do in parallel
S2:         b[i,j,k]=f2(b[i-1,j+1,k],b[i-1,j,k+1])
```

and for the second one

```
l2    forall j=0,n-1 do in parallel
l1      for i=1,n-1 do
S1:       a[i,j]=f1(a[i-1,j],b[i-1,i,j])
```

Since in both cases the loops to be parallelised comprise a single statement, the algorithm terminates. The distributed parallel version of the original loop nest ($\mathcal{PL}2$) is then obtained by catenating the parallel code for the two components (Figure 5). This is also the final result, because the undistributed parallel version of $\mathcal{L}$ reveals less potential parallelism than the distributed one.

3 Data and computation partitioning

In the second stage of the BSP parallelisation scheme, the potential parallelism of the loop nest is mapped on the processor-memory pairs of the BSP computer. This is done by distributing the data and the computation of the loop such as to allocate operations that can be performed in parallel to distinct processors. As in the case of other parallelisation techniques targeted at distributed-memory systems [5, 14, 15, 30], we first build a data partition, and then apply the "owner computes" rule to partition the associated computation.

Most of the above mentioned approaches obtain the data distribution (or a starting point for its computation) from the user ([5] is an exception from this point of view). Unlike these approaches, we provide a simple algorithm that permits the automatic detection of an appropriate data distribution whenever the computation to be parallelised has a regular structure. Only when this algorithm is not able to generate a data partition is user expertise required for selecting the data distribution.

The partitioning algorithm is presented in Figure 6. It takes two input parameters, the parallelised loop nest ($\mathcal{PL}$) and the set of the main statements in the loop (*main_stmts*), and returns a distribution of the arrays referenced within the loop body. For each array, and each of its dimensions, the algorithms decides whether the array should be partitioned into blocks, or left *serial* (i.e., unpartitioned) in that dimension. The array dimensions that are candidates for block-partitioning are those corresponding to parallel loops, namely those whose associated subscript has the form $x * j + y$, where j is the index of a surrounding parallel loop and x, y are integer constants.

The algorithm is designed such as to first exploit the coarsest parallelism available. Therefore, it initially takes into account only the outermost layer of parallel loops surrounding the main statements of $\mathcal{PL}$. When an acceptable distribution (the criterion used to establish the acceptability of a distribution is described later in the text) cannot be obtained at this level, inner layers of parallelism are iteratively explored. The algorithm terminates when either an acceptable distribution is found, or no such distribution is available even when fully exploiting the potential parallelism of the loop.

The algorithm works as follows. First, $\mathcal{PL}'$, a copy of the parallelised loop nest is built. In this copy, the parallel loops belonging to layers greater than *par_loop_layer*, the currently considered layer of parallel loops, are sequentialised. Then, a stable data distribution is built in steps 1 to 3. In step 1, the distribution is initialised: for each array

```
compute_data_partition(PL, main_stmts)
  par_loop_layer ← 0; acceptable ← FALSE
  while acceptable=FALSE and par_loop_layer ≤ max_par_loop_layer do
    ready ← FALSE
    PL' ← PL with parallel loops in layers > par_loop_layer sequentialised
    while ready =FALSE do
      Step 1.  Initialise the partition
        for each array a in PL' and each dimension i of a do
          distribution[a, i] ← UNDEFINED
      Step 2.  Partition the arrays in dimensions corresponding to parallel loops
        for each main statement S do
          for each dimension i of an array a referenced by S do
            if the i-th subscript of a has the form x ∗ j + y, where j is the index
            of a surrounding parallel loop and x, y are integer constants
              if distribution[a, i] =UNDEFINED
                distribution[a, i] ← BLOCK
            else
              distribution[a, i] ← SERIAL
      Step 3.  Check if any parallel loop in PL' has to be sequentialised
        ready ← TRUE
        for each main stmt S and each parallel loop pl surrounding S in PL' do
          for each array a referenced by S do
            if a is the definition of a main statement and a has no
            partitioned dimension whose associated subscript has the form
            x ∗ j + y, with j the index of pl and x, y integer constants
              sequentialise pl
              ready ← FALSE
    Step 4.  Verify if current partition is acceptable
      acceptable ← TRUE
      for each main stmt S do
        if def(S) is partitioned in no dimension
          acceptable ← FALSE
    par_loop_layer ← par_loop_layer + 1
  if acceptable =TRUE
    return distribution, with undefined elements set to SERIAL
  else
    get data partition from user starting from the one described by distribution
```

Figure 6: The data partitioning algorithm.

and each dimension of that array the way of distributing the data is marked as undefined. In step 2, the partitioning of each array referenced by a main statement is attempted. Thus, array dimensions that correspond to parallel loops in $\mathcal{PL}'$ and whose distribution is undefined are marked as block-partitioned, and array dimensions that correspond to no parallel loop are marked as serial-partitioned.

Finally, in step 3, the algorithm checks that each parallel loop surrounding a main statement S corresponds to at least a partitioned dimension of each array referenced by

S, and which is also the *definition* of a main statement (i.e., the array written by a main statement). When this is not the case, the later usage of the "owner computes" rule for the partitioning of the computation will map all the iterations of the considered parallel loop on every processor. Or, in other words, the parallel loop would be executed as if it were sequential. Hence, the loop is explicitly sequentialised, and the flag *ready* is set to FALSE, to indicate that a stable data distribution has not been reached yet. The three steps are iterated until no further change of $\mathcal{PL}'$ is required.

Then, the acceptability of the resulting data distribution is verified in step 4. A data distribution is acceptable if for each main statement S, $def(S)$—the definition of statement S—is block-partitioned in at least one dimension. Indeed, if the array defined by a main statement were fully unpartitioned, that main statement would be scheduled for sequential execution on every processor, rendering the whole BSP program inefficient. Therefore, whenever such a main statement is identified, the algorithm searches for a better data distribution using one more layer of parallel loops.

The algorithm terminates when an acceptable data partition is found, or when such a distribution cannot be built even exploiting the entire potential parallelism of the loop nest. In the former case, the algorithm returns the acceptable distribution, while in the latter case user expertise is sought.

Let us illustrate the application of the data partitioning algorithm by using it to build a data distribution for the parallelised loop nest in Figure 5. The main statements of the loop (as considered when identifying the potential parallelism) are S_1, S_2. The loop nest has only one layer of parallel loops (namely loop l_2 surrounding S_1, and loops l_3, l_4 surrounding S_2). Hence, in the first and only application of the algorithm, $\mathcal{PL}'$ is initially identical to $\mathcal{PL}$. After initialising the data distribution to

$$\begin{array}{l} a[\mathrm{UNDEFINED}, \mathrm{UNDEFINED}] \\ b[\mathrm{UNDEFINED}, \mathrm{UNDEFINED}, \mathrm{UNDEFINED}] \end{array}$$

in step 1, step 2 is performed, first on statement S_1, and then on statement S_2. After performing step 2 on statement S_1, the data distribution is changed into:

$$\begin{array}{l} a[\mathrm{SERIAL}, \mathrm{BLOCK}] \\ b[\mathrm{SERIAL}, \mathrm{SERIAL}, \mathrm{BLOCK}] \end{array}$$

since array dimensions corresponding to the subscript j are marked as block-partitioned, and those corresponding to the subscripts i and $i-1$ are marked as serial-partitioned. The application of step 2 to statement S_2 brings no change into the data distribution.

However, in step 3, the algorithm identifies the fact that loop l_3 surrounding statement S_2 corresponds to no partitioned dimension of array b. Consequently, this loop is sequentialised, the distribution is marked as unstable, and a new iteration through steps 1 to 3 is performed. The new iteration yields the same data distribution, and is also found to be stable. Hence, the algorithm proceeds to step 4, and verifies the acceptability of the distribution. Since both $a = def(S_1)$ and $b = def(S_2)$ are partitioned in at least one dimension, the data distribution is acceptable and the algorithm successfully terminates. According to this data distribution, a generic processor me, $0 \leq me < p$ will be considered to "own"

$$
\begin{array}{l}
a[0..n-1, me\lceil n/p\rceil..\min\{n-1,(me+1)\lceil n/p\rceil-1\}] \\
b[0..n-1, 0..n-1, me\lceil n/p\rceil..\min\{n-1,(me+1)\lceil n/p\rceil-1\}]
\end{array}
$$

in the BSP schedule to be built in the remaining stages of the parallelisation scheme.

As for the generic case, when an m-dimensional array $a[n_1, n_2, \ldots, n_m]$ is block-partitioned in $1 \leq q \leq m$ dimensions, the p processors of the BSP computer are considered to be arranged into a q-dimensional hypercube of size $p^{1/q}$. Then, if the coordinates of processor me, $0 \leq me < p$, in this hypercube are $(me_1^q, me_2^q, \ldots, me_q^q)$, processor me will own

$$a[l_1..u_1, l_2..u_2, \ldots, l_m..u_m]$$

where the lower and upper bounds along the i-th dimension, $1 \leq i \leq m$ are

$$
(l_i,u_i)=\begin{cases}
(0, n_i-1) & \text{if } a \text{ is unpartitioned in dimension } i \\
(me_j^q\lceil n_i/p^{\frac{1}{q}}\rceil, \min\{n_i-1, (me_j^q+1)\lceil n_i/p^{\frac{1}{q}}\rceil-1\} & \text{if dimension } i \text{ is the } j\text{-th partitioned dimension of } a
\end{cases} \quad (2)
$$

After building a data distribution, the second stage of the parallelisation scheme continues with the partitioning of the computation of the loop nest. Typically, the computation partitioning algorithms based on the "owner computes" rule comprise two phases [15, 29]. In the first phase, they generate *masks* (i.e., boolean guards checking the ownership of the modified data) for each statement in the program. Then, in the latter phase, these algorithms attempt to adjust surrounding loop bounds such as to eliminate as many statement masks as possible.

The computation partitioning algorithm devised in this section (Figure 7) differs from other algorithms in that it computes the new loop bounds and the statement masks in a single phase. This is achieved by translating each data ownership constraint that must be satisfied by a statement instance which is executed locally either in a loop bound constraint or in a statement mask (step 1 of `computation_partition`). Thus, for each statement S, and for each block-partitioned dimension i of the definition of S, the subscript s corresponding to that dimension must satisfy the ownership constraint:

$$l_i \leq s \leq u_i \quad (3)$$

where l_i, u_i are given by eq. (2). As indicated in Figure 7, whenever $s = xj + y$, with j the index of a surrounding parallel loop pl, and x, y integer constants, this condition translates into

$$\lceil (l_i - y)/x \rceil \leq j \leq \lfloor (u_i - y)/x \rfloor$$

and is stored as a pair of lower/upper constraints associated with statement S and loop pl (i.e., $lower[pl, S]$, $upper[pl, S]$). If, however, s does not have this form, the constraint (3) is added to the mask of statement S.

The algorithm also comprises a second step in which the new loop bounds and the statement masks are inserted into the parallelised version of the loop nest. This step (function `add_masks&bounds`) is performed by applying the following rules on each component

```
computation_partition(PL, data_distribution)
  Step 0.  Initialise statement masks and loop bounds
    for each statement S do
      mask[S] ← {}
      for each loop loop surrounding S do
        lower[loop, S] ← {}; upper[loop, S] ← {}
  Step 1.  Compute loop bounds and statement masks
    for each statement S and each partitioned dimension i of def(S) do
      let s be the i-th subscript of def(S)
      if s = x * j + y where j is the index of a surrounding parallel loop pl,
      and x, y are integer constants
        lower[pl, S] ← max{lower[pl, S], ⌈(l_i−y)/x⌉}, with l_i given by eq. (2)
        upper[pl, S] ← min{upper[pl, S], ⌊(u_i−y)/x⌋}, with u_i given by eq. (2)
      else
        mask[S] ← mask[S] and l_i ≤ s ≤ u_i, with l_i, u_i given by eq. (2)
  Step 2.  Add loop bounds and statement masks to PL
    add_masks&bounds(PL, mask, lower, upper)

add_masks&bounds(PL, mask, lower, upper)
  for each component C of PL do
    if C is a parallel loop pl
      overall_lower ← {}; overall_upper ← {}
      for each statement S surrounded by loop pl do
        overall_lower ← min{overall_lower, lower[pl, S]}
        overall_upper ← max{overall_upper, upper[pl, S]}
      replace lower_bound(pl), upper_bound(pl) with max{lower_bound(pl), overall_lower}
      and min{upper_bound(pl), overall_upper}, respectively
      call add_masks&bounds for the body of the loop pl
    else if C is a sequential loop
      leave loop control unchanged
      call add_masks&bounds for the body of the loop
    else  /* C is a simple statement S */
      for each loop loop surrounding S do
        if lower[loop, S] ≠ {} and lower[loop, S] > lower_bound[loop]
          mask[S] ← mask[S] and lower[loop, S] ≤ index(loop)
        if upper[loop, S] ≠ {} and upper[loop, S] < upper_bound[loop]
          mask[S] ← mask[S] and index(loop) ≤ upper[loop, S]
      if mask[S] ≠ {}
        replace S with the statement ‘‘if mask[S] then S’’
```

Figure 7: The computation partitioning algorithm.

$\mathcal{C}$ of the loop nest. First, if $\mathcal{C}$ is a parallel loop pl originally iterating from $lower\_bound(pl)$ to $upper\_bound(pl)$, the iteration range of the loop is changed such that its new lower bound is

$$\max\left\{lower\_bound(pl), \min_{\text{stmts } S \text{ surrounded by } pl} lower[pl, S]\right\}$$

```
l1   for i=1,n-1 do
l3     forall j=0,n-2 do in parallel
l4       forall k=me*ceil(n/p),min(n-2,(me+1)*ceil(n/p)-1) do in parallel
S2:        b[i,j,k]=f2(b[i-1,j+1,k],b[i-1,j,k+1])
l2   forall j=me*ceil(n/p),min(n-1,(me+1)*ceil(n/p)-1) do in parallel
l1     for i=1,n-1 do
S1:      a[i,j]=f1(a[i-1,j],b[i-1,i,j])
```

Figure 8: The partitioned parallel version of the loop nest in Figure 2.

---

and its new upper bound is

$$\min\left\{ upper\_bound(pl), \max_{\text{stmts } S \text{ surrounded by } pl} upper[pl,S] \right\}$$

Then, the masks and bounds insertion algorithm is recursively applied to the body of the loop.

Second, if $\mathcal{C}$ is a sequential loop, the loop control is left unchanged, and the algorithm is recursively applied to the body of the loop. Finally, when $\mathcal{C}$ is an assignment statement, the mask of the statement is added as a guard to the statement. Before doing so, however, the mask is augmented with the loop bound constraints associated with the statement and which are not enforced by the new bounds of the surrounding loops.

To illustrate the application of the algorithm, we will partition the computation of the parallelised loop nest in Figure 5. The data distribution obtained in the first part of this section is $a[\text{SERIAL}, \text{BLOCK}]$, $b[\text{SERIAL}, \text{SERIAL}, \text{BLOCK}]$. Consequently, the first step of the algorithm yields:

$$\begin{aligned}
&lower[l_1, S_1] = \{\} \\
&upper[l_1, S_1] = \{\} \\
&lower[l_2, S_1] = me\lceil n/p \rceil \\
&upper[l_2, S_1] = (me+1)\lceil n/p \rceil - 1 \\
&mask[S_1] = \{\} \\
&lower[l_1, S_2] = \{\} \\
&upper[l_1, S_2] = \{\} \\
&lower[l_3, S_2] = \{\} \\
&upper[l_3, S_2] = \{\} \\
&lower[l_4, S_2] = me\lceil n/p \rceil \\
&upper[l_4, S_2] = (me+1)\lceil n/p \rceil - 1 \\
&mask[S_2] = \{\}
\end{aligned}$$

These data ownership constraints are then inserted into the parallel code in the second step of the algorithm, generating the partitioned parallel version of the loop nest (Figure 8).

4 Synchronisation and communication generation

In the third and final stage of the BSP parallelisation scheme, the complete BSP schedule of the original loop nest is built by inserting appropriate calls to BSP synchronisation and communication primitives into the partitioned parallel version of the loop nest.

Unlike other parallelisation techniques targeted at distributed-memory architectures, our parallelisation scheme has to take into account the decoupling of communication and synchronisation that characterises the BSP model of computation. Consequently, the typical approach of generating pairs of send/receive communication primitives that solve the requirements for non-local data and synchronise the two processors involved in the data transfer is not applicable in the BSP setting. Instead, we will generate calls to both BSP communication (i.e., `BSP_get`) and BSP synchronisation (i.e., `BSP_sync`) primitives [13], based on the information in the dependence graph of the loop nest. Before presenting an algorithm that generates these BSP primitive calls, let us analyse what non-local data a generic statement may need.

First, since the computation was partitioned using the "owner computes" rule, all statement definitions (i.e., variables written by statements) are local. Furthermore, output dependences always occur between local statement instances and can be neglected in this stage. The only variable references that may refer non-local data are therefore statement *uses* (i.e., variables that are read by statements). Accordingly, the algorithm will focus on the identification of those statement uses that refer non-local data, and will generate `BSP_get` calls that fetch this data.

In order to reduce the synchronisation and communication overheads, these communication primitives will be inserted into the code at the outermost possible position. When an use that refers non-local data is the sink of no flow data dependence, this "outermost possible position" is before the entire computation. Hence, antidependences can be ignored by the algorithm until the very last moment: non-local data which is the source of an antidependence will be fetched into the local memory before any computation takes place. When, however, an use that refers non-local data is the sink of at least one flow dependence, the non-local data will be fetched within the loop body of the innermost loop that carries such a dependence.

The insertion of the communication primitives at the outermost possible level reduces the BSP parallelisation overheads in several ways. First, it permits the *vectorisation* of remote data gets (i.e., the fetching of array blocks rather than of individual array elements). Second, the algorithm uses this communication generation strategy to avoid the fetching of multiple copies of the same remote data (this is called *communication aggregation*). Finally, the synchronisation overhead is minimised since the `BSP_sync` primitives that must follow each sequence of remote data gets will also be inserted at the outermost possible level.

The synchronisation and communication generation algorithm is presented in Figure 9. The algorithm takes three parameters: $\mathcal{PL}$, the partitioned parallel version of the loop nest, $\mathcal{G}$, its dependence graph, and *data_distribution*, the data partition. During its execution, appropriate BSP communication and synchronisation is inserted into $\mathcal{PL}$, which is an input-output parameter.

```
generate_sync&comm(PL, G, data_distribution)
  for each statement S and each variable use read by S do
    satisfied_use[S,use] ← FALSE
  add_sync&comm_to_block(PL,G,data_distribution, satisfied_use)
  for each statement S and each variable use read by S do
    if satisfied_use[S,use] = FALSE
      generate a BSP_get fetching the non-local elements of use before entire code
    if at least a BSP_get was generated before the entire code
      insert a BSP_sync call before the entire code

add_sync&comm_to_block(B, G, data_distribution, satisfied_use)
  for each component C in the block B do
    if C is a loop
      add_sync&comm_to_loop(C, G, data_distribution, satisfied_use)
    else  /* C is an assignment statement S */
      for each variable use read by S do
        if use is owned by the local processor for all instances of S
          satisfied_use[S,use] ← TRUE
  if B comprises more than one component
    taking into account all flow dependences whose sink use from statement S has
    satisfied_use[S,use] = FALSE, topologically sort the components in subsets S_1,
    S_2, .., S_m such that the components in S_k depend only on those in S_1, .., S_{k-1}
    rearrange the components in topologically sorted order
    for k=2,m do
      for each flow dependence between a component in S_1, .., S_{k-1} and one in S_k do
        let the variable use from statement S be the sink of the dependence
        if satisfied_use[S,use] = FALSE
          insert a BSP_get fetching the non-local elements of use before S_k
          satisfied_use[S,use] ← TRUE
      insert a BSP_sync before S_k

add_sync&comm_to_loop(L, G, data_distribution, satisfied_use)
  add_sync&comm_to_block(loop_body(L), G, data_distribution, satisfied_use)
  if L is a sequential loop
    for each flow dependence between two statements in loop_body(L)
      let the variable use from statement S be the sink of the dependence
      if satisfied_use[S,use] = FALSE and the dependence is carried by L
        insert a BSP_get fetching the non-local elements of use
        before the loop body of L
        satisfied_use[S,use] ← TRUE
    if at least a BSP_get was inserted before the loop body of L
      insert a BSP_sync before the loop body of L
  else  /* L is a parallel loop */
    if L is part of a perfect loop nest and communication primitives were
    generated before the body of an innerer loop
      move L between the communication primitives and the loop body,
      adjusting the communication primitives accordingly
```

Figure 9: The synchronisation and communication generation algorithm.

The algorithm works by initially marking each statement use in the code as not satisfied locally, and "solving" the non-local uses in a depth-first search fashion. Thus, the communication and synchronisation required by the use of non-local data that is the sink of flow dependences is first generated by calling the function `add_sync&comm_to_block` for the entire block of code $\mathcal{PL}$. Then, each statement use is examined, and, if necessary, an additional communication superstep solving the uses that are still unsatisfied at this outermost level is added before the entire code.

In order to generate the communication and synchronisation corresponding to a block of code (i.e., to a sequence of loops and/or simple statements), the algorithm is initially applied to each component of the block. Then—if the block comprises several components—communication satisfying dependences between statements in different components is generated. The application of the algorithm for a single component $\mathcal{C}$ depends on the type of that component. Thus, if $\mathcal{C}$ is a loop, the algorithm (function `add_sync&comm_to_loop`) is first applied to the loop body, and then communication and synchronisation accounting for the flow dependences carried by the loop itself is generated. If, on the other hand, the component is a simple statement (i.e., an assignment), the algorithm only marks locally held uses as satisfied.

Whenever a parallel loop is encountered during this depth-first search, the parallel loop may be usable to decrease the inner synchronisation overheads. Indeed, if the parallel loop belongs to a perfect loop nest and a communication superstep was previously inserted before the body of an innerer loop of this nest, Theorem 2 allows the moving of the parallel loop between the communication superstep and the loop body. Here is an example of such a situation. Consider that the following code is part of the parallel version of a loop nest:

```
l1  forall i=0,n-1 do in parallel
l2    for j=1,n-1 do
l3      forall k=0,n-2 do in parallel
S1:       a[i][j][k]=f(a[i][j-1][k-2])
```

and assume that the data distribution for array a is $a[\mathrm{SERIAL}, \mathrm{SERIAL}, \mathrm{BLOCK}]$. Then, after partitioning the computation and generating the communication and synchronisation for statement S_1 and loops l_2, l_3, the following BSP code is obtained:

```
l1  forall i=0,n-1 do in parallel
l2    for j=1,n-1 do
        if me>0
          BSP_get a[i,j-1,me*ceil(n/p)-2..me*ceil(n/p)-1]
                 from processor me-1
        BSP_sync
l3      forall k=me*ceil(n/p),min(n-2,(me+1)*ceil(n/p)-1) do in parallel
S1:       a[i][j][k]=f(a[i][j-1][k-2])
```

This BSP schedule comprises n^2 supersteps. By moving loop l_1 in an inner position as described above, the number of supersteps is decreased to n:

```
l1  for i=1,n-1 do
      if me<p-1 do
        BSP_get b[i-1,0..n-2,(me+1)*ceil(n/p)] from processor me+1
      BSP_sync
l3    for j=0,n-2 do
l4      for k=me*ceil(n/p),min(n-2,(me+1)*ceil(n/p)-1) do
S2:       b[i,j,k]=f2(b[i-1,j+1,k],b[i-1,j,k+1])
l2  for j=me*ceil(n/p),min(n-1,(me+1)*ceil(n/p)-1) do
l1    for i=1,n-1 do
S1:     a[i,j]=f1(a[i-1,j],b[i-1,i,j])
```

Figure 10: The BSP schedule of the loop nest in Figure 2.

```
l2  for j=1,n-1 do
      if me>0
        BSP_get a[0..n-1,j-1,me*ceil(n/p)-2..me*ceil(n/p)-1]
                from processor me-1
      BSP_sync
l1    forall i=0,n-1 do in parallel
l3      forall k=me*ceil(n/p),min(n-2,(me+1)*ceil(n/p)-1) do in parallel
S1:       a[i][j][k]=f(a[i][j-1][k-2])
```

It is also worth emphasising that, when a `BSP_get` call corresponding to the use of a statement S is generated and statement S acquired a mask in the previous stage of the parallelisation scheme, this mask is automatically inherited by the `BSP_get` call. For the sake of simplicity, this rule is not included in the algorithm description in Figure 9.

Let us now complete the BSP parallelisation of the loop nest in Figure 2, by adding the necessary communication and synchronisation to its partitioned parallel version in Figure 8. The algorithm described in this section is first applied to the loop nest containing statement S_2. The use $b[i-1, j+1, k]$ in this statement is identified as locally satisfied, but the use $b[i-1, j, k+1]$ refers non-local data for some values of k. Hence, appropriate communication and synchronisation is inserted before the body of loop l_1, the innermost loop that carries a flow dependence whose sink is $b[i-1, j, k+1]$:

```
l1 for i=1,n-1 do
     if me<p-1
       BSP_get b[i-1,0..n-2,(me+1)*ceil(n/p)] from processor me+1
     BSP_sync
l3   forall j=0,n-2 do in parallel
l4     forall k=me*ceil(n/p),min(n-2,(me+1)*ceil(n/p)-1) do in parallel
S2:      b[i,j,k]=f2(b[i-1,j+1,k],b[i-1,j,k+1])
```

Then, the algorithm is applied to the second loop nest, the one surrounding statement S_1. In this case, both uses in statement S_1 are local, so no modification of the code is

```
performance_analysis(𝒫ℒ)
  (sync_cost, comp_cost, comm_cost) ← (0, 0, 0)
  for each component 𝒞 of 𝒫ℒ do
    if 𝒞 is a simple assignment statement
      (sync_cost, comp_cost, comm_cost) ← (sync_cost, comp_cost + 1, comm_cost)
    else if 𝒞 is a barrier synchronisation
      (sync_cost, comp_cost, comm_cost) ← (sync_cost + L, comp_cost, comm_cost)
    else if 𝒞 is a BSP_get call
      (sync_cost, comp_cost, comm_cost) ← (sync_cost, comp_cost,
                                           comm_cost + g * fetched_data_size)
    else  /* 𝒞 is a loop l */
      (sync_cost, comp_cost, comm_cost) ← (sync_cost, comp_cost, comm_cost)+
          (upper_bound(l) − lower_bound(l) + 1)*performance_analysis(loop_body(l))
  return (sync_cost, comp_cost, comm_cost)
```

Figure 11: The performance analysis algorithm.

required. Since all statement uses are satisfied at this point, no further communication and synchronisation is added during the execution of the algorithm. The BSP schedule of the loop nest is depicted in Figure 10.

5 Performance analysis

As emphasised in the introductory section, one of the salient properties of the BSP model of computation is its *predictability*, i.e., the possibility to accurately analyse the cost of any BSP program. In this section, the BSP cost model is used to analyse the performance of a BSP schedule derived using the parallelisation scheme presented in the report. This cost analysis can readily be performed manually for the BSP schedule at hand. However, since the parallelisation scheme is intended for implementation into an automatic BSP scheduling tool, we will also introduce a simple algorithm that estimates the performance of the BSP version of a generic loop nest. As part of the above mentioned tool, this algorithm will be of great use, allowing the rapid assessment of the cost of a generic BSP program, and the comparison of several schedules of the same piece of sequential code.

The performance analysis algorithm (Figure 11) applies the BSP cost model to the schedule $\mathcal{PL}$ generated by the parallelisation scheme in a straightforward manner. The synchronisation, computation and communication costs of the considered BSP code are estimated by the algorithm in a depth-first search style. During the computation of these costs, each barrier synchronisation is charged a (synchronisation) cost of L, each assignment a (computation) cost of 1, and each fetching of a remote word of data a (communication) cost of g.

Since statement masks are not taken into account, the algorithm will provide an upper bound for the computation cost. On the other hand, since all processors must reach each barrier synchronisation, the synchronisation overhead of the schedule is accurately computed. As for the communication cost, its assesment is based on the assumption that, in each superstep, the maximum amount of data sent by any processor is closely approximated by the maximum amount of data received by any processor. This assumption is highly justified by the uniformity of the data distribution, and by the regularity of the communication pattern generated by the parallelisation scheme.

Applied to the BSP schedule in Figure 10, the performance analysis algorithm returns

$$\left(nL, \frac{n^3+n^2}{p}, gn^2\right)$$

with the three terms standing for the costs of synchronisation, computation, and communication, respectively. It is immediate to see that the three costs are accurate, and indicate that the implementation of the schedule will yield nearly optimal speedup on any BSP computer whose parameters satisfy the constraints

$$pL \ll n^2, \qquad pg \ll n$$

6 Conclusions and further work

The novel parallelisation scheme introduced in this report addresses the automatic mapping of generic loop nests onto bulk-synchronous parallel computers. Unlike other parallelisation strategies proposed so far, the new technique yields parallel code which is scalable, transportable across any shared or distributed memory parallel computer, and whose performance can be accurately and analytically evaluated. Furthermore, although targeted in the first instance at generic loop nests, the parallelisation scheme can be applied unchanged to entire sequential programs regarded as sequences of loop nests and simple statements.

The BSP parallelisation technique has been manually tested on a large number of untightly nested loops representing dense computations, the performance of the resulting schedules being comparable to that of hand-written BSP code. Finally, following the successful development of a BSP tool which implements the perfect loop nest scheduling techniques in [21, 11, 8, 9, 10], further work is currently being carried out to integrate the new parallelisation technique into the existing scheduling tool.

At the same time, several improvements of the parallelisation scheme are envisaged as necessary. Firstly, the data partitioning algorithm devised in Section 3 must be augmented in order to permit the cyclic distribution of arrays in dimensions corresponding to variable-length loops. Indeed, such an improvement would significantly decrease the workload imbalance that occurs when the loop nest to be parallelised has a triangular iteration space. Secondly, *array alignment* in the partitioned dimensions (i.e., the permutation of the processor coordinates and/or the addition of a constant displacement to the lower and upper bounds in eq. (2)) should be considered as a means to eliminate unnecessary statement masking and to decrease communication costs [14]. Thirdly, in order to bring the BSP pseudocode generated by the parallelisation scheme closer to real BSP code, the scheme must be enhanced to identify the locally referenced elements of each array, and to allocate local storage for these elements only. Finally, when the programming language used to implement the BSP schedules does not allow array subscripts varying within an arbitrary range, subscript translations must be operated into the BSP code to bring these subscripts within the permitted range.

References

[1] J.R. Allen et al. Conversion of control dependence to data dependence. In *Conference Record of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189. ACM Press, 1983.

[2] J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proc. 14th Annual ACM Symposium on Principles of Programming Languages*, pages 63–76. ACM Press, 1987.

[4] D F Bacon et al. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):346–420, December 1994.

[5] P. Banerjee et al. The PARADIGM compiler for distributed-memory multiprocessors. *Computer*, 28(10):37–47, October 1995.

[6] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[7] U. Banerjee. *Dependence Analysis.* Kluwer Academic Publishers, 1997.

[8] R. Calinescu. Bulk synchronous parallel scheduling of uniform dags. In L. Bouge et al., editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 555–562. Springer–Verlag, 1996.

[9] R. Calinescu. A BSP approach to the scheduling of tightly-nested loops. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS'97), April 1-5, 1997, Geneva, Switzerland*, pages 549–553. IEEE Computer Society Press, 1997.

[10] R. Calinescu. BSP scheduling of regular patterns of computation. Technical Report PRG-TR-1-97, Programming Research Group, Oxford University Computing Laboratory, January 1997.

[11] Y.-Z. Ding and D. Stefanescu. Efficient scheduling of loop nests for BSP programs. Technical Report TR-04-95, Center for Research in Computing Technology, Harvard University, 1995.

[12] M.W. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, December 1996.

[13] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: The BSP programming library. Technical report, Oxford University Computing Laboratory, May 1997.

[14] S. Hiranandani et al. Compiling FORTRAN D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[15] S.P. Johnson, C.S. Ierotheou, and M. Cross. Automatic parallel code generation for message passing on distributed memory systems. *Parallel Computing*, 22(2):227–258, 1996.

[16] D.J. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys*, 9(1):29–59, March 1977.

[17] D.J. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, 1978.

[18] L Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[19] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer–Verlag, 1993.

[20] W F McColl. General purpose parallel computing. In A. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation: 4, pages 337–391. Cambridge University Press, 1993.

[21] W.F. McColl. Scalable computing. In J. Van Leeuwen, editor, *Computer science today : recent trends and developments*, Lecture Notes in Computer Science 1000, pages 46–61. Springer-Verlag, 1995.

[22] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[23] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[24] R.A. Towle. *Control and data dependence for program transformations.* PhD thesis, Univ. of Illinois at Urbana-Champaign, 1976.

[25] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[26] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–470, October 1991.

[27] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[28] M J Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[29] H.P. Zima and B.M. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1990.

[30] H.P. Zima and B.M. Chapman. Compiling for distributed-memory systems. *Proc. of the IEEE*, 81(2):264–287, February 1993.