Programming Research Group

A BSP SCHEDULING TOOL FOR LOOP NEST PARALLELISATION

Radu Calinescu

PRG-TR-36-97

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

A BSP Scheduling Tool for Loop Nest Parallelisation

Radu Calinescu

November 1997

Abstract

This report introduces BSPscheduler, a new tool for the automatic parallelisation of nested loops. The novel parallelisation tool generates bulk synchronous parallel (BSP) code, by automatically scheduling the data and the computation of a sequential loop nest for concurrent execution on a BSP computer. Being targeted at the BSP model of computation, the resulting parallel code is scalable, portable, and its cost can be accurately analysed. The current report describes the two-phase scheduling strategy underlying the implementation of the tool, and presents its structure and an example of a parallelisation session with BSPscheduler.

1 Introduction

Within the last few years, a number of realistic models of parallel computation have been proposed, with the aim of providing a generic framework for the design of parallel architectures and applications. These models, which encompass many of the characteristics that made the von Neumann model so successful in the realm of sequential computing, include the bulk synchronous parallel (BSP) model of Valiant [30, 24], the LogP model of Culler et al. [12], the Weakly coherent PRAM (WPRAM) model of Nash et al. [27], etc.

Due to its simplicity, elegance and generality, the BSP model represents one of the most attractive of these proposals. This position is further justified by the recent release of a worldwide standard BSP programming library [16], and of a BSP application development toolset [15, 17] comprising powerful profiling tools. As a result, parallel code which is scalable, portable, and whose cost can be accurately analysed, can be developed using the BSP library and tuned for best performance with the BSP profiling tools.

The above mentioned advances represent important steps forward in the attempt to make the transition from sequential to parallel computing as smooth and unpainful as possible. The current report takes another step in this direction, by introducing a tool intended to assist the programmer in the development of BSP applications. The new tool, called *BSPscheduler*, is an automatic loop nest paralleliser targeted at the BSP model of computation. Therefore, unlike other parallelising compilers and tools devised within the last decade or so [2, 14, 19, 32, 35, 36], the novel parallelisation tool generates parallel code that presents all the features characteristic to BSP code: portability, scalability, costability.

The BSP parallelisation tool distributes the data and the computation of a sequential loop nest among the processors–memory units of a BSP computer using a two–phase scheduling strategy. In the first phase of this strategy, the tool attempts to use *template–matching scheduling*, i.e., to match the computation described by the loop nest against one of several templates of regular computation and to use a predefined BSP schedule for its parallelisation. The loop nest templates recognised by the tool are introduced in [13, 25, 8, 9, 10] and overviewed in Section 2.1. When this first phase succeeds in identifying a pattern in the sequential code to be parallelised, the tool immediately generates an efficient (or even optimal) BSP schedule. When, however, this does not happen, the tool proceeds to a second phase, in which the generic technique for the BSP parallelisation of nested loops developed in [11] and briefly described in Section 2.2 is used to generate a BSP schedule for the sequential loop nest.

BSPscheduler has a modular structure, and was implemented as an X Window application. The structure of tool, and each of its constituent blocks are described in Section 3, together with details on their implementation. An example of a scheduling session with the new BSP tool is then presented in Section 4. Finally, Section 5 completes the report with several concluding remarks and a discussion of the possible additions to the current version of the parallelisation tool.

2 A two–phase strategy for the automatic parallelisation of nested loops

The loop nest parallelisation techniques developed so far fall into two broad classes. The first class comprises techniques that approach the parallelisation of regular computations surrounded by a set of perfectly nested loops [7, 9, 13, 21, 22, 23, 28, 29, 31, 34], i.e., techniques that are referred to in this report as *template–matching scheduling*. As the techniques in this class build a parallel version of the the sequential loop nest starting from a predefined schedule, they present the advantage of inexpensively generating efficient, usually optimal, parallel code. Nevertheless, when faced with the parallelisation of a nested loop whose structure cannot be matched against any of the available computation templates, template–matching scheduling is unable to deliver any parallel schedule whatsoever.

The latter class of techniques, on the other hand, tackles the automatic parallelisation of generic loop nests [2, 3, 11, 36, 35]. Consequently, the techniques belonging to this class provide a means of parallelising the most various types of nested loops. Unfortunately, generic loop nest scheduling has its own disadvantages. Thus, it typically comprises several distinct steps (e.g., data dependence analysis, potential parallelism identification, data and computation partitioning, etc.), therefore being very intricate and computationally intensive. Then, generic parallelisation techniques are prone to yield suboptimal schedules even when dealing with very simple nested loops.

In order to take advantage of the benefits of both types of parallelisation techniques,

```
for i=1,n-1 do
  for j=0,n-1 do
    a[i][j]=f(a[i-1][j])
```

a

```
forall j=0,n-1 do in parallel
  for i=1,n-1 do
    a[i][j]=f(a[i-1][j])
```

b

```
for j=me*⌈n/p⌉,min{n-1,(me+1)*⌈n/p⌉-1} do
  for i=1,n-1 do
    a[i][j]=f(a[i-1][j])
```

c

Figure 1: A 2–level fully parallel loop nest (a) whose only data dependence of distance vector $(1,0)$ permits the parallelisation of the loop indexed by j in the outermost position (b). Hence, an optimal one-superstep BSP schedule can be obtained by distributing the iterations of the parallel loop among the p processors, with each processor me, $0 \leq me < p$ computing an equally sized block of the iteration space (c).

the BSP scheduling tool presented in this report implements a two-phase, hybrid scheduling strategy. In the first phase of this strategy, the application of template-matching scheduling is attempted. If this attempt is successful, the parallelisation is completed. Otherwise, generic loop nest scheduling is employed in a second phase of the strategy. The template–matching and generic loop nest scheduling techniques used in the two phases of the BSP scheduling strategy are introduced in [13, 25, 8, 9, 10] and [11], respectively, and are briefly reviewed in the remainder of this section.

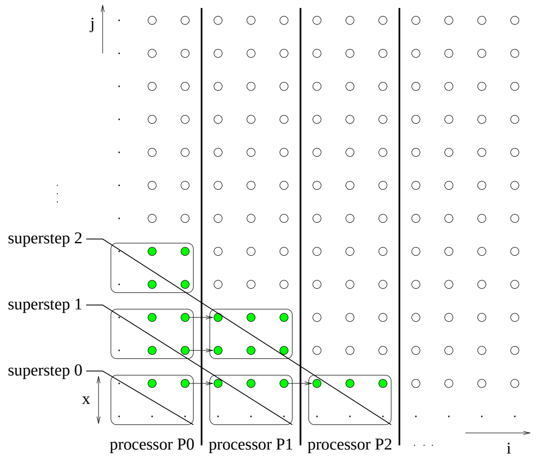
2.1 Template–matching BSP scheduling

The current version of parallelisation tool implements four of the template–matching BSP scheduling techniques introduced in [13, 25, 8, 9, 10]. The first and most simple template recognised by the tool is called a *fully parallel loop nest* [13, 9, 10]. As illustrated in Figure 1, a fully parallel loop nest represents a perfectly nested loop whose data dependence pattern permits the parallelisation of (at least) one of its loops in the outermost position. Since the loops parallelised in the outermost position carry no data dependence, the BSP schedule of a fully parallel loop nest (Figure 1c) partitions the iteration space of the nested loop into p equally sized independent blocks, and assigns each processor the computation of one such block. The whole computation takes a single superstep, and is one-optimal.

A second template–scheduling technique implemented by the tool is called *wavefront* (or *dag*) *scheduling* [25, 8, 10]. This technique is used to schedule perfectly nested loops whose data dependences have a regular structure and can be described by distance vec-

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

a



b

```
for superstep=0,⌈n/x⌉+p-2 do
  tile=superstep-me
  if me>0 and 0<=tile<⌈n/x⌉ then
    BSP_get a[me*⌈n/p⌉-1][max{1,tile*x}..min{n-1,(tile+1)*x-1}]
            from processor me-1
  BSP_sync
  if 0<=tile<⌈n/x⌉ then
    for i=max{1,me*⌈n/p⌉}, min{n-1,(me+1)*⌈n/p⌉-1} do
      for j=max{1,tile*x},min{n-1,(tile+1)*x-1} do
        a[i][j]=f(a[i-1][j],a[i][j-1])
```

c

Figure 2: A 2–level loop nest whose data dependences are described by the set of distance vectors $\{(1,0),(0,1)\}$ (a), the partitioning of its iteration space in wavefront scheduling (b), and the BSP pseudocode executed by processor me, $0 \leq me < p$ (c).

tors with non–negative components[1] (Figure 2a). Wavefront scheduling works as follows (Figure 2b-c). First, the iteration space of the loop is partitioned into p equally sized rectangular blocks along $k-1$ arbitrarily chosen directions of the iteration space, where k is the dimensionality of the iteration space. Each of the resulting blocks is assigned to a different processor, and is further subdivided into tiles of size x along the remaining direction of the iteration space. The BSP schedule starts with the computation of the tile

[1]in fact, wavefront scheduling can also be used when the distance vectors comprise negative elements; however, BSPscheduler does not employ wavefront scheduling in such cases because it would induce a partitioning of the iteration space into non-rectangular blocks, i.e., into blocks whose manipulation requires an expensive and intricate index arithmetic.

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

a

```
for i=1,n-1 do
  if me<p-1 then
    BSP_get a[i-1][(me+1)*⌈n/p⌉] from processor me+1
  BSP_sync
  for j=me*⌈n/p⌉,min{n-2,(me+1)*⌈n/p⌉-1} do
    a[i][j]=f(a[i-1][j],a[i-1][j+1])
```

b

Figure 3: A 2–level nested loop whose outermost loop carries the two data dependences of the loop nest (a), and its BSP iterative schedule (b).

---

that contains the first iteration point of the nested loop: in a first superstep, the processor that owns this tile is active, and all the other processors are idle. Then, the computation advances across the iteration space like a wavefront: in each subsequent superstep, the processors that own tiles neighbouring a tile computed in the previous superstep execute these tiles. As shown in Figure 2b-c, every tile computation is preceded by a boundary data transfer between processors that own neighbouring blocks. By appropriately choosing the tile size x, one-optimality can be approached when the problem size is large enough.

The third template–matching scheduling technique implemented by the tool tackles the BSP parallelisation of perfect loop nests comprising a loop that carries all the data dependences of the nested loop. This technique is called *iterative scheduling*, and works by bringing the loop that carries all the data dependences in the outermost position, and executing its iterations in different, consecutive supersteps [10]. In order to distribute the computation among the p processors, the iteration space of the nested loop is partitioned into p blocks of equal size in the dimensions corresponding to the inner loops. As a result, in each superstep (or *iteration*) of the resulting BSP schedule, the p processors execute the loop body for an equal number of iteration points, namely for the iteration points belonging to the block they own. As shown by the example in Figure 3, boundary data exchanges between neighbour processors are necessary in each superstep, to account for the usage of non-local data. A BSP schedule generated through iterative scheduling is one-optimal in computation, but involves a high synchronisation overhead. Accordingly, iterative scheduling can only be efficient when the problem size is very large, or when the dimensionality of the iteration space to be parallelised is higher than 2.

There is sometimes the case that two or even all the scheduling techniques presented so far can be used to parallelise a nested loop. In this situation, the tool first attempts to

```
for i=1,n do
  for j=1,n do
    s=s+f(i,j)
```

a

```
for i=me0*⌈n/p0⌉+1,min{n,(me0+1)*⌈n/p0⌉} do
  for j=me1*⌈n/p1⌉+1,min{n,(me1+1)*⌈n/p1⌉} do
    s=s+f(i,j)
BSP_combine(s,PLUS)
```

b

Figure 4: A 2–level loop nest implementing a simple reduction (a), and its BSP schedule (b). The iteration space of the nested loop was block–partitioned in both dimensions, hence the processors must be viewed as arranged in an array of size $p_0 \times p_1$, $p_0 \cdot p_1 = p$, with processor me, $0 \leq me < p$ having the coordinates (me_0, me_1), $0 \leq me_0 < p_0$, $0 \leq me_1 < p_1$.

schedule the nested loop as a fully parallel loop nest. When, however, the set of scheduling techniques that can be applied only include wavefront and iterative scheduling, it is the user's responsibility to choose between the two.

The last template–matching scheduling technique implemented by BSPscheduler deals with the scheduling of perfect loop nests describing a reduction operation (Figure 4). In *reduction scheduling* [10], the iteration space is partitioned into p identical blocks, and each processor is assigned the computation of a partial reduction. This one-superstep local stage in which each processor computes a partial reduction is then followed by a global stage, in which a BSP combine operation is used to build the overall result. As long as the problem size is large enough, the cost of the local stage dominates the overall cost of the schedule, and the resulting BSP code approaches one-optimality.

2.2 BSP parallelisation of generic loop nests

The generic loop nest parallelisation scheme employed by the BSP tool when no template–matching scheduling technique is applicable comprises four steps (Figure 5). The first of these steps deals with the analysis of the data dependences in the nested loop. Then, this dependence information is used to identify the parallelism available in the sequential code in a second step. In a third step, the data and the computation are partitioned among the processor/memory units of the BSP computer. Finally, in the fourth step, communication and synchronisation is inserted into the incomplete parallel code generated in step 3. This section includes a brief summary of what is going on in each of the four steps; for a thorough description of the algorithms used in each of these steps, the reader is referred to [11].

Two statements are said to be data dependent if they access the same variable, at least one of them modifying the value of the variable. The existence of a data dependence between two statements requires that the execution order of the two statements be preserved in any parallel version of the original code. Therefore, the aim of the dependence analysis step is to identify all the data dependences in the input sequential code, and to build the dependence graph [6] of the nested loop based on the *direction vector* [33]

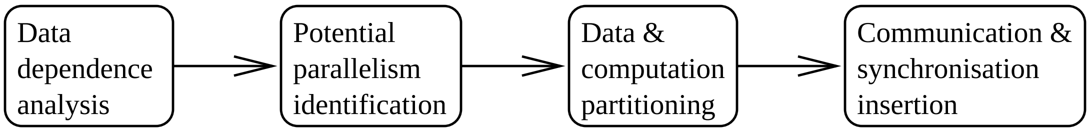


Figure 5: The 4–step scheme for the BSP scheduling of generic loop nests.

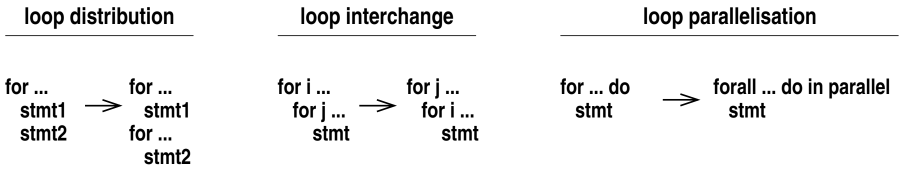


Figure 6: Code transformation techniques used in the potential parallelism identification step.

representation of data dependences. The dependence analysis process is conservative, namely dependence is assumed whenever independence cannot be proved.

In the second step of the BSP parallelisation scheme, a set of rules describing the code transformations allowed by the existing data dependences are used to alter the original code in order to exhibit its potential parallelism. Three code transformation techniques are employed by the scheme (Figure 6). The first one, called *loop distribution*, duplicates a loop control and distributes it over the statements forming the loop body. This transformation is typically used to break a data dependence between two statements in the loop body. The second transformation, called *loop interchange*, swaps the controls of two loops in a nested loop. It is used either to allow the parallelisation of the loop that is brought into the inner position, or to move a parallel loop into the outermost position. Finally, the third transformation, called *loop parallelisation*, replaces a sequential loop control with a parallel loop control, indicating that all the iterations of that loop can be executed concurrently, without any synchronisation or communication.

Once the potential parallelism of the sequential code is identified, and a PRAM–style version of the original loop nest is available, the data and the computation are distributed among the p processors of a BSP computer such as to maximise parallelism and to minimise communication and synchronisation. The partitioning algorithm applied in the third step of the scheme is based on two rules. According to the first rule, the algorithm attempts to partition the arrays used within the loop body in dimensions corresponding to subscripts that are linear combinations of parallel loop indices. Block distribution in one or several dimensions is available. The second rule, called the "owner computes" rule [4, 18, 19, 35], specifies that the computation must be partitioned in such a way that any statement computation is assigned to the processor that owns the variable modified by that statement computation.

Finally, in the fourth step of the parallelisation scheme, the complete BSP code is generated by inserting appropriate calls to BSP synchronisation and communication prim-

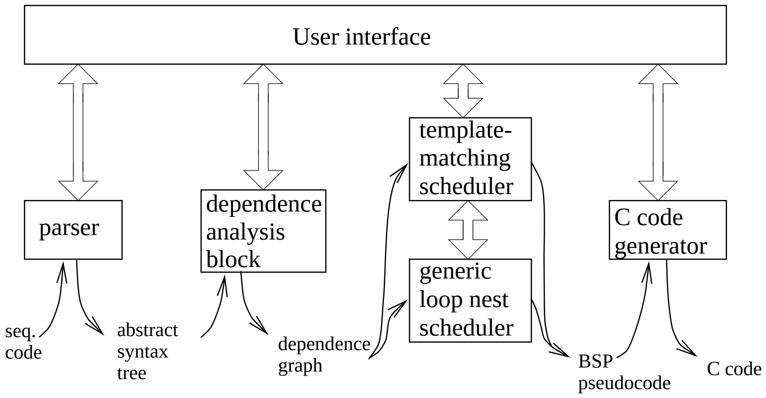


Figure 7: The structure of BSPscheduler.

itives into the intermediate parallel code devised in step 3. 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 [4, 18] 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.

3 The structure of the BSP parallelisation tool

The structure of the parallelisation tool is depicted in Figure 7. First, it comprises a parser block which analyses the sequential loop nest fed into the tool and builds an internal, abstract syntax tree representation of this code. All other modules then operate on this internal representation of the code rather than on the original sequential code. The parser module also generates a symbol table and other compiler–like data structures, and operates minor changes into the original code in order to bring it to a normalised form.

The dependence analysis block which follows, applies a series of dependence tests on the internal representation of the loop nest, building the adjacency matrix of its dependence graph. The dependence information conveyed by the dependence graph is then passed on to two scheduler modules, one for each type of scheduling implemented by the tool. At this level, a decision is taken on which scheduling block is in charge with the parallelisation, and this block is responsible for generating the BSP version of the original loop nest. This BSP version is described in some simple pseudocode, and can be converted into real C code comprising BSP primitive calls by a C code generator block. Finally, the BSP tool also comprises a user interface that transfers information between the user and the other modules of the tool.

This section provides details about each of the BSPscheduler constituent blocks, with the emphasis placed on their interaction with the user.

3.1 The parser module

The parser module has the role of analysing the sequential loop nest to be parallelised, and of building its abstract syntax tree, together with several compiler–like data structures used by the other modules of the tool. These data structures include a symbol table, as well as a loop table and a statement table comprising important information about each loop and each assignment statement in the original code, respectively.

The sequential loop nest must be described in a simple, FORTRAN–style pseudocode. A formal description of the BSPscheduler input language is provided in Appendix A. In addition to the syntactic rules of this minimal imperative language, the parser checks the following semantic rules:

Rule 1: Loop indices cannot be modified (i.e., assigned to) within the loop body.

Rule 2: Any identifier used to define the size of an array in an array declaration is a symbolic constant, and cannot be used as the left hand side of an assignment or as a loop index.

Furthermore, the tool allows the usage of user functions, but requires that function parameters are scalars passed as values, function executions have no side effects, and functions return a scalar value. Under these circumstances, the exact definitions of user functions are irrelevant for the parallelisation of the loop nest, and therefore not provided to the tool.

When analysing a sequential loop nest, the parser stops at the first syntax or semantic error it encounters, issuing a message that specifies the type of the error and the line in the input code where the error was found. The user can then restart the parser after correcting the error.

During the analysis of the input code, the parser brings the loop nest to a normalised form, i.e., to a form in which control dependences are converted into data dependences [1], and conditional statement controls are distributed over the statements in the body of the conditional statement (Figure 8). In the normalisation process, auxiliary array variables are used to eliminate control dependences. These variables, whose name is formed by adding a number generated by a special counter to the prefix `_cond`, are automatically introduced by the parser, but their declarations have to be completed by the user.

As shown in Figure 8, in addition to the normalisation of the input code, the parser numbers each loop and statement in the code. These numbers are then used by the tool to refer to a specific loop or statement.

3.2 The dependence analysis module

Using the information in the abstract syntax tree and in the other data structures generated by the parser block, the dependence analysis module applies dependence tests in order to identify all the data dependences between pairs of statements that refer the same variable. The following data dependence tests are used by the current version of the dependence analysis block:

```
scalar i,j,k,n
array a[n][n],b[n][n]
for i=0,n-1 do
 for j=0,n-1 do
  if a[i][j-1]<0 then
   a[i][j]=...
   b[i][j]=...
  else
   a[i][j]=...
  endif
 endfor
endfor
```

a

```
    scalar i, j, k, n
    array a[n][n], b[n][n], _cond1[n][n]
l1  for i=0,n-1 do
l2   for j=0,n-1 do
S1:   _cond1[i][j]=a[i][j-1]<0
S2:   if _cond1[i][j] then a[i][j]=...  endif
S3:   if _cond1[i][j] then b[i][j]=...  endif
S4:   if not _cond1[i][j] then a[i][j]=...  endif
     endfor
    endfor
```

b

Figure 8: A nested loop (a), and its normalised version generated by the parser (b).

- the *separability test* [36], an exact test that can be used to compare two array subscripts with the form $xi + y$, $x'i + y'$, where x, x', y, y' are integer constants and i is a surrounding loop index;
- the *gcd test* [2, 36], an approximate test used to compare two array subscripts which are linear combinations of the surrounding loop indices;
- the *Banerjee test* [2, 5, 6], another approximate test applicable to two array subscripts that are linear combinations of the surrounding loop indices.

Due to the modular structure of BSPscheduler, additional dependence tests can be easily added at any time, without modifying any of the other blocks of the tool.

As a result of the dependence analysis, the adjacency matrix of the dependence graph [6] of the loop nest is build. Each entry in this matrix describes all the data dependences involving the two statements associated with the entry. Thus, for every data dependence, the adjacency matrix provides the following information:

- the type of the dependence (i.e., true, anti–, or output dependence);
- the variable references that form the source and the sink of the data dependence;
- the distance vector [20] and the direction vector [33] representations of the data dependence.

The dependence analysis process is conservative, namely whenever the dependence tests fail to prove that two data references are independent, dependence is unconditionally assumed. Therefore, the user is shown a list presenting all the data dependences identified in the input code, and is allowed to delete any of these dependences. Based on the dependence information, the user can also decide to modify the original loop nest in order to eliminate a real data dependence that limits or even prohibits the parallelisation of the loop nest.

3.3 The scheduling modules

Based on the dependence information provided by the dependence analysis block, and on the structure of the abstract syntax tree built by the parser, the scheduling modules decide on the type of scheduling that suits the nested loop to be parallelised. Then, the scheduling module that implements the appropriate type of scheduling is in charge of generating the parallel version of the loop nest. While this is the default scheduling strategy employed by the tool, the user is able to change it, by choosing to bypass template-matching scheduling and to rely entirely on generic loop nest scheduling.

When the template-matching scheduler is assigned the parallelisation of the original loop nest, it builds the abstract syntax tree of the BSP schedule starting for a predefined BSP code skeleton. At the same time, the scheduler generates a data partition specifying the distribution of all arrays used within the loop body, and, whenever possible, a version of the loop nest in which the available potential parallelism is explicitly shown.

When, on the other hand, template-matching scheduling is not possible, or the user chooses to bypass it, the generic loop nest scheduler becomes active. In order to parallelise the loop nest, this module first asks the user to indicate [11] the *main statements* (i.e., the statements whose execution distribution among processors is essential) and the *main loops* (i.e., the loops whose parallelisation makes sense) of the nested loop.

Then, the scheduler uses the three code transformation techniques described in Section 2.2 to expose the potential parallelism of the loop nest, concentrating on the parallelisation of main loops that surround main statements. The resulting intermediate parallel version of the loop nest is then used in the automatic generation of a data partition. Block distribution of array elements, in one or several dimensions, is implemented in the current version of the tool. As the "owner computes" rule described in Section 2.2 is subsequently applied to partition the computation, the data partition is considered acceptable if the *definition* (i.e., the left hand side) of each main statement is block–distributed in at least one dimension. If an acceptable data partition cannot be derived automatically, the user is shown the potential parallelism of the loop nest and the data partition reached by the tool, and is asked to use his or her expertise to derive a final data partition.

In order to decrease the synchronisation overhead of the resulting BSP schedule, the BSP tool is tuned to minimise the number of loop distributions used in the potential parallelism identification step. However, the user can override this setting and ask that a maximum number of loops are distributed over the statements in their loop body, thus increasing the chance of an acceptable data partition being automatically derived. Furthermore, the user has the option of checking and modifying the automatically generated data partition before the tool proceeds to the distribution of computation, even when this partition is acceptable.

Once the abstract syntax tree representation of the BSP schedule of the loop nest is generated by one of the scheduling modules, the user interface block translates this internal representation into BSP pseudocode. The language used to describe the BSP pseudocode is entirely similar to the BSPscheduler input language, but includes new types of statement for the specification of BSP function calls. The syntax of these new types of statement is formally defined in Appendix B. Beyond the new types of statement, the tool

identifier	type	description
`_p`	variable	number of processors executing the BSP schedule
`_me`	variable	the number of the current processor, $0 \leq$`_me`$<$`_p`
`_p`x`_`y	variable	when the processors are arranged in an x–dimensional array, number of processors in the $(y+1)$-th dimension of the array
`_me`x`_`y	variable	when the processors are arranged in an x–dimensional array, the $(y+1)$-th coordinate of the current processor in the array
`_ceil`	function	returns the ceiling of its single parameter
`_floor`	function	returns the floor of its single parameter
`_min`	function	returns the minimum over its set of arguments
`_max`	function	returns the maximum over its set of arguments

Table 1: The auxiliary functions and variables used by the scheduling tool. An underscore prefixes the name of each of these functions and variables in order to differentiate them from the user defined symbols.

introduces several new functions and auxiliary scalar variables in the BSP pseudocode it generates; an account of these functions and variables is presented in Table 1.

In addition to the BSP schedule of the original loop nest, the user is shown the data partition used by the tool, and, whenever possible, an intermediate parallel version revealing the potential parallelism of the original loop nest. The data partition is presented by indicating, for each array and each dimension of that array, whether the array is partitioned or not in that dimension. For instance, if a two–dimensional array a is block–partitioned in its first dimension and is unpartitioned in its second dimension, the line describing the partitioning of array a will be:

$$a(\text{block}, *)$$

Accordingly, if the array size is $n_1 \times n_2$, the current processor $\_me$, $0 \leq \_me < \_p$ will own the rectangular block:

$$a[\_me\lceil n_1/\_p\rceil..\min\{n_1-1, (\_me+1)\lceil n_1/\_p\rceil - 1\}][0..n_2-1]$$

and will execute all statements that modify these elements of a. When an array is block–partitioned in $k > 1$ dimensions, the processors are viewed as arranged in a k-dimensional array of size $\_p_0^k \times \_p_1^k \times \cdots \times \_p_{k-1}^k$, with the current processor having the coordinates $(\_me_0^k, \_me_1^k, \ldots, \_me_{k-1}^k)$ and owning a similarly computed rectangular block of the array.

3.4 The code generation module

During a scheduling session with the BSP tool, the user can repeatedly modify the various scheduling options and/or the input code in order to generate several BSP schedules for the input loop nest. When the user decides for one of the possible schedules, the C

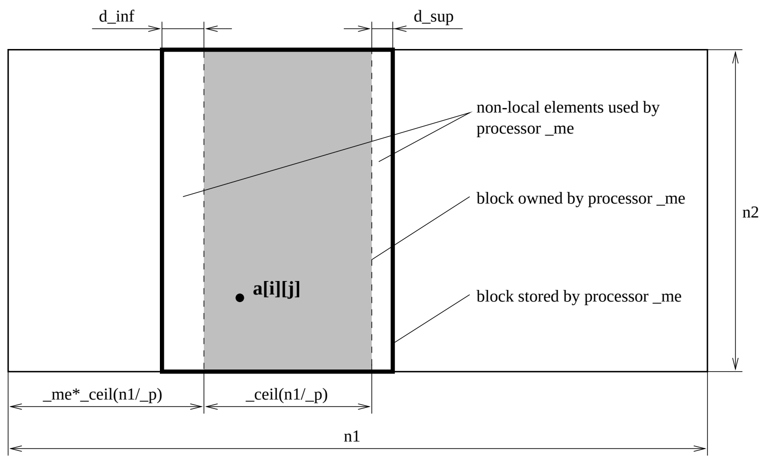


Figure 9: A two-dimensional array a of size $n_1 \times n_2$ which is block-partitioned in its first dimension, and the array block stored locally by processor $\_me$, $0 \leq \_me < \_p$. With the notations in the diagram, a reference to a generic array element $a[i][j]$ is converted into $a[i - \_me\lceil n_1/\_p\rceil + d\_inf][j]$.

---

code generator module can be used to translate the BSP pseudocode into real code. The resulting code is written in C and comprises calls to functions from the standard BSP library BSPlib [16]. The output code is generated into a file and includes the following parts:

a. a header comprising a description of the data partition and of the array blocks stored on each processor;
b. a part comprising `#include` directives and macro definitions;
c. a function for initialising the data owned by each processor (`initialise_data`), and a function for collecting the results after the entire computation is over (`output_data`);
d. the user functions;
e. a function implementing the BSP schedule (`parallel_code`).

The remainder of this section provides a brief description of each of these elements.

The header of the output file lists, for each array in the loop nest, the partition of the array and the array block stored on each processor. While the array partitions are the ones generated during the parallelisation of the loop nest, the array blocks stored locally represent the smallest rectangular blocks that contain all the array elements referred by the considered processor during the entire computation. Such an array block includes the block owned by the current processor, as well as all the non–local elements whose value is used locally. Whenever a processor keeps in its memory only a part of an array, the subscripts of each array reference must be accordingly adjusted (Figure 9). Furthermore, the arrays used in the loop nest are vectorised in the C code to simplify the manipulation of array references. Consequently, a reference $a[s_1][s_2]\ldots[s_m]$ to a generic element of an m-dimensional array a is further replaced with $a[s_1 n_2 n_3 \ldots n_m + s_2 n_3 \ldots n_m + ... + s_m]$, where the size of the array block stored locally is $n_1 \times n_2 \times \cdots \times n_m$.

The second part of the output code comprises directives that include the BSPlib header file and several ANSI C header files into the code, as well as a series of macro definitions. These macros define the auxiliary functions introduced by the tool (Table 1), and the array vectorisation primitives used to translate multidimensional array coordinates into linear coordinates (e.g., `_lin2(i1,i2,n2)=i1*n2+i2`, etc.).

The body of the input/output functions in the third part of the code, and the body of the user functions in its forth part are left empty by the tool. Before actually including the automatically generated code into a BSP application, the user has the responsibility of writing the code for these functions. In doing so, the user must ensure that the input function correctly initialises the data which is locally owned by each processor, and that the output function saves all the computation results required by the user.

Finally, the actual computation is implemented as a C function that forms the last part of the code. This function takes as arguments the symbolic constants used in array declarations, and the number of processors on which the code is executed. When due to the partitioning of an array in more than one dimension, the processors are viewed as arranged into an array, the function also takes as arguments the sizes of the processor array (i.e., the appropriate _px_y variables described in Table 1). The user is responsible for calling this function with appropriate arguments, after initialising the BSP application.

3.5 The user interface

The BSP scheduling tool was implemented as an X Window application, with the user interface (Figure 10) developed on top of the X Toolkit Intrinsics and the Athena widget set [26]. The commands of the tool are organised under five categories.

The file–related commands provide the usual facilities for loading or saving a source file (i.e., a file containing the sequential code to be parallelised). The tool also provides a basic text editor that can be used to modify an existing loop nest and/or to type in a new one. The text editor is disabled during the actual parallelisation, but can return to the edit mode at any moment (command File/Modify). However, any BSP schedule or intermediate scheduling results computed up to that moment are deleted before re–enabling the text editor.

At any time, the BSPscheduler window can show one or several panes comprising various scheduling information. The following panes are available:

- the source pane containing the original loop nest;
- the normalised source pane containing the normalised version of the loop nest;
- the data dependence pane listing the dependences identified in the sequential code;
- the intermediate results pane providing details on the potential parallelism of the nested loop and on the final data partition;
- the BSP pseudocode pane comprising the BSP version of the original loop nest.

The user can include a new pane among those shown by the BSPscheduler window or eliminate a visible pane by using the View commands. Clearly, some of the panes are only available after several (or all) steps of the parallelisation process have been performed. For instance, the BSP pseudocode pane is created after the entire scheduling process is

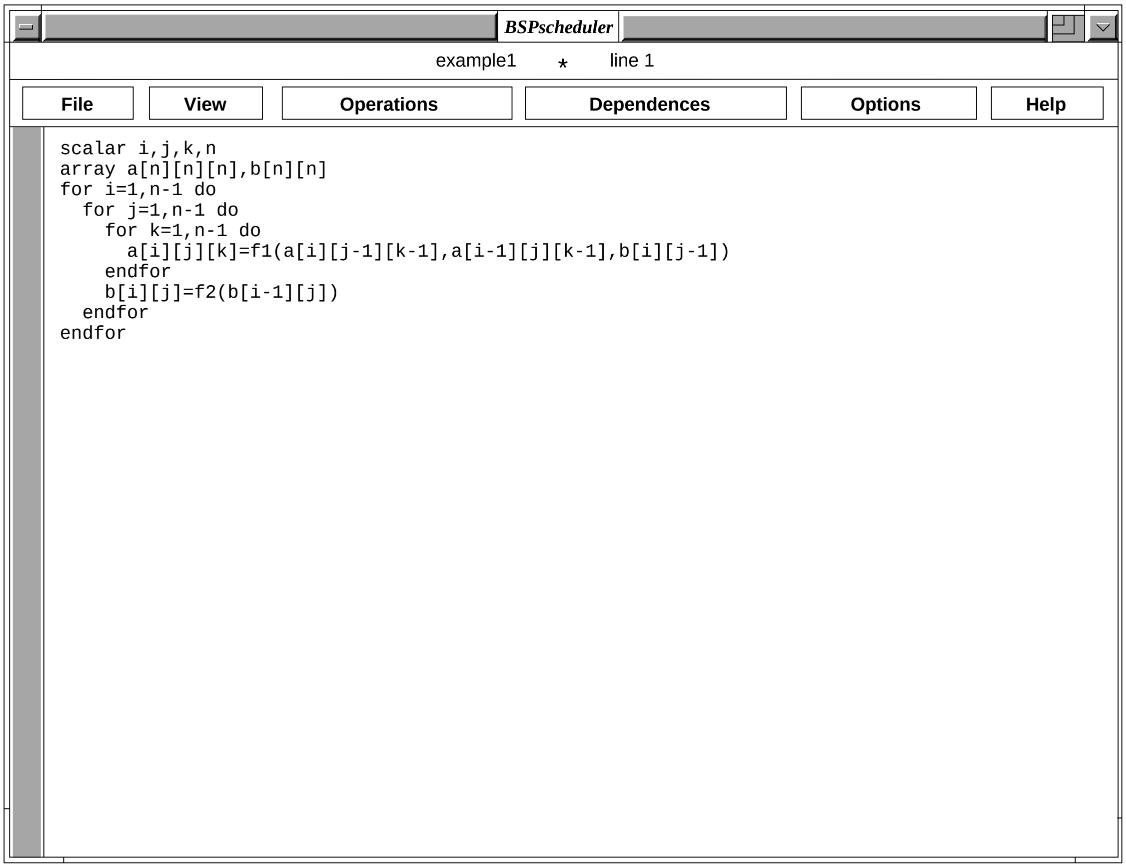


Figure 10: The user interface of the BSP scheduling tool; a sample sequential loop nest is loaded into the tool.

completed. To distinguish among these panes, all but the source pane comprise a title line.

The commands that guide the parallelisation are gathered under the Operations group. They can be used to start the parser module and to initiate the dependence analysis step, as well as to activate the scheduling and the C code generation modules. The last couple of operations can be performed repeatedly, for different scheduling options and output files, respectively.

Since the dependence analysis process is conservative (see Section 3.2 for details), delete/undelete dependence commands are provided by the tool in the dependence–related command group. At the same time, the user can tune the scheduling results by using the commands grouped under the Options heading. Thus, the user is allowed to enable or disable the usage of template–matching schedule, to check and/or modify the data partition automatically generated by the tool, and to force the distribution of a maximum number of loops over the statements belonging to their loop body. Finally, on–line help information on BSPscheduler can be obtained using the Help group of commands.

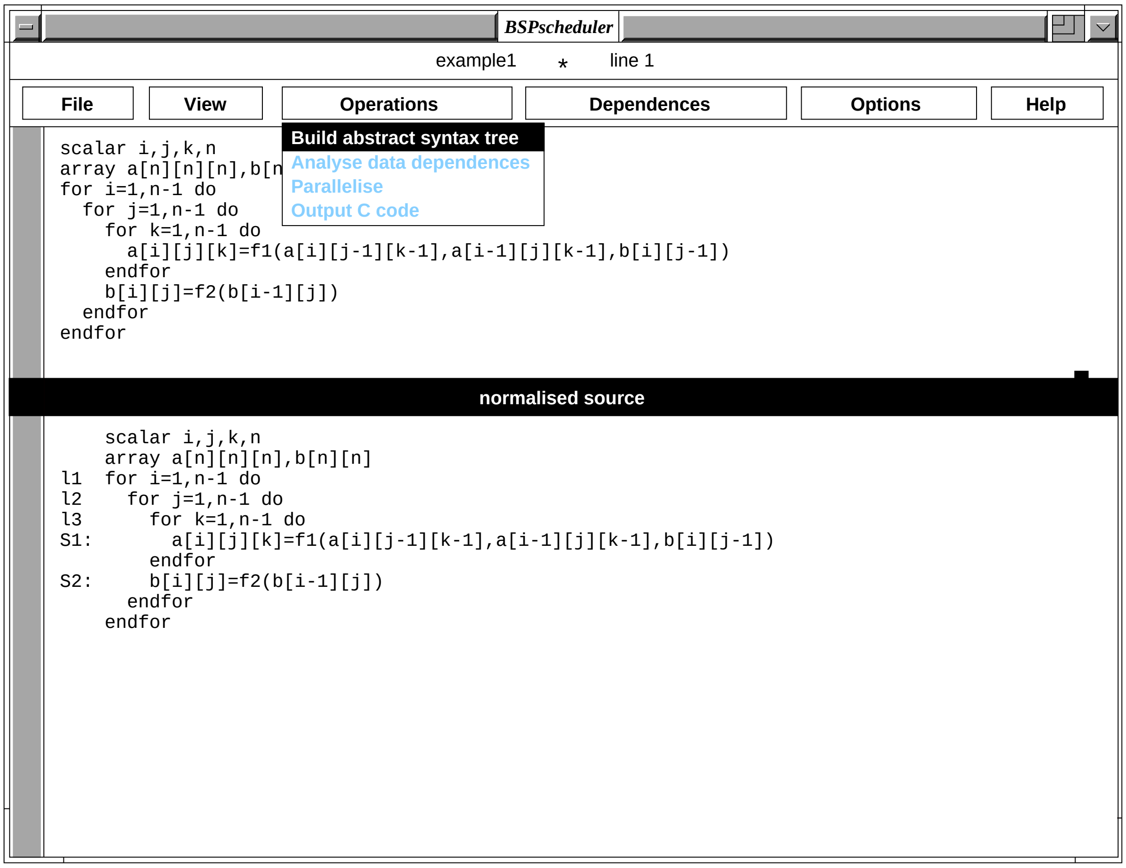


Figure 11: The translation of the sequential loop nest into the BSPscheduler internal representation is preceded by a normalisation of the source code; the loop nest example in Figure 10 is used to illustrate this step.

4 Example of a scheduling session with the BSP tool

A scheduling session with the BSP tool comprises several steps. In the first step (Figure 10), the user is required to provide a sequential loop nest described in the pseudocode defined in Appendix A, either by loading this source code from a file or by typing it into the BSPscheduler editor. Then, in a second step, the abstract syntax tree representation of the loop nest must be built (command Operations/Build abstract syntax tree); at the same time, the normalised version of the source code is generated (Figure 11).

In the following step of the illustrated scheduling session, the data dependences in the loop nest are computed (command Operations/Analyse data dependences), and the data dependence pane is inserted into the BSPscheduler window (Figure 12). At this point, the user can study each data dependence and delete any dependence that was conservatively assumed by the tool but does not exist. Alternately, the user may want to alter the source code (command File/Modify) to eliminate some data dependences.

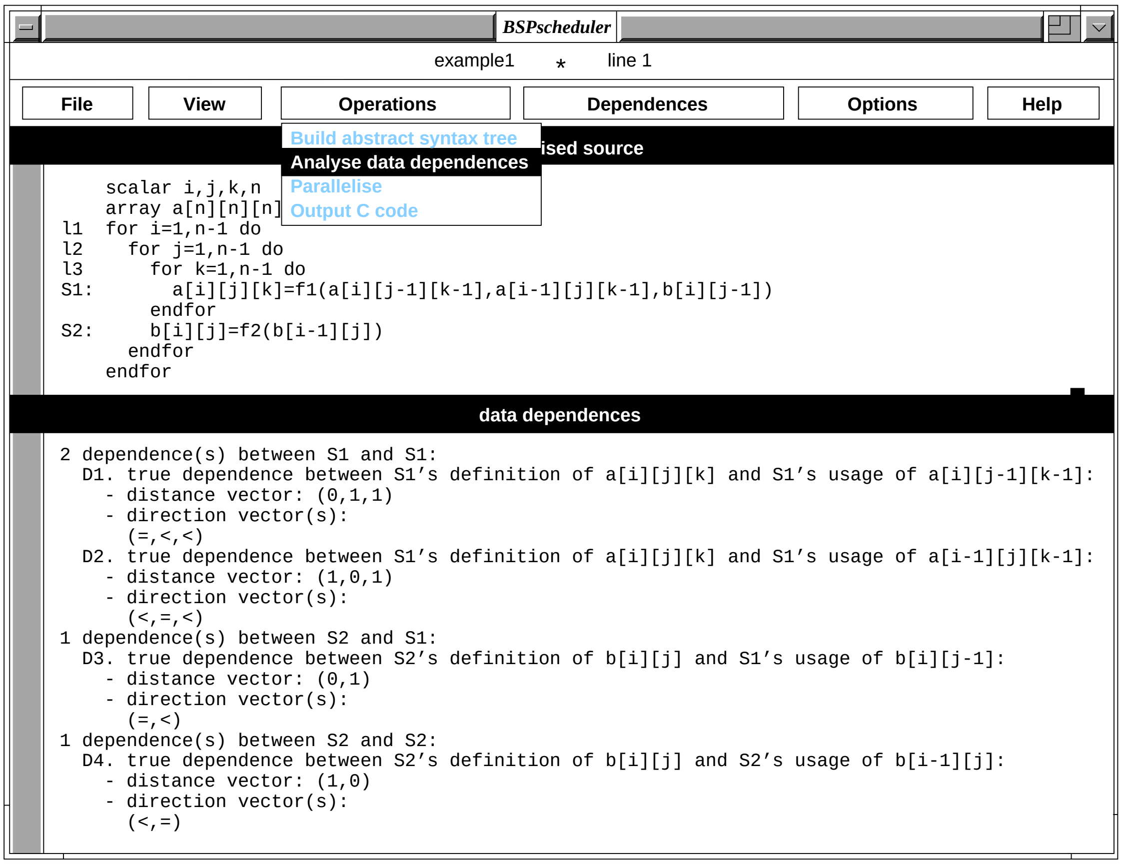


Figure 12: The dependence analysis step.

Then, in the next step of a scheduling session, the actual BSP schedule of the loop nest is obtained (command Operations/Parallelise). During this step, the tool may request some additional information from the user. The exact nature of this information depends on the type of scheduling used by the tool. Thus, when template–matching scheduling is used, a choice may exist on the loop whose iterations are distributed among the processors, on the loop to be brought into the outermost position, etc. Whenever such a choice exists, the tool issues an explanatory message and asks for a user choice. For generic loop nest scheduling, on the other hand, the tool needs to know the main loops and the main statements of the nested loop, and, sometimes, to obtain a final decision on the data partition to be used in the parallelisation process. The user is prompted to input all this information. Following the completion of this step, the BSP pseudocode pane is inserted into the BSPscheduler window (Figure 13). Additional information about the parallelisation (i.e., potential parallelism and data partition information) can be studied in the intermediate results pane (command View/Show intermediate results)—see Figure 14.

The parallelisation step can be repeated, possibly for different scheduling parameter settings, as many times as desired. When an acceptable BSP schedule is obtained, a

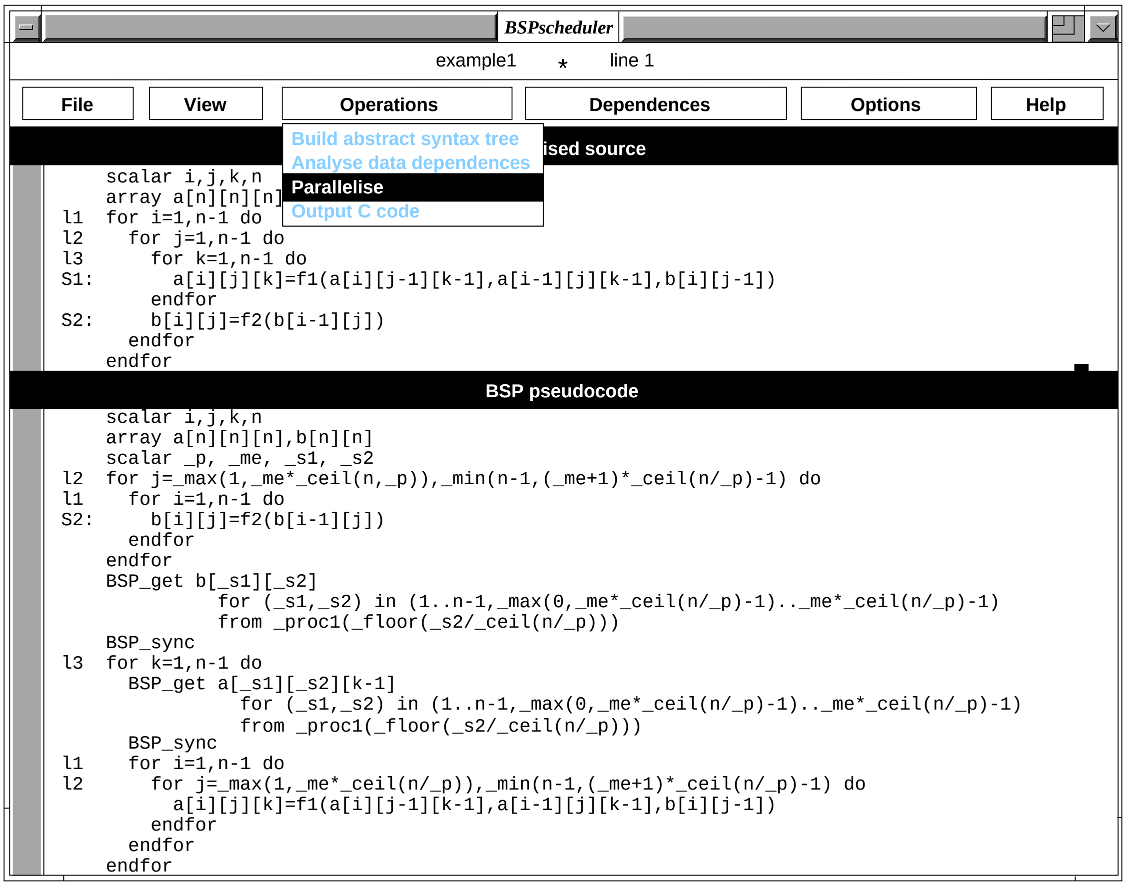


Figure 13: The actual parallelisation step.

real code version of it may be written into an output file. The structure of the resulting output C code is presented in Section 3.4, and the output code for the sample loop nest whose scheduling was illustrated in this section is included in Appendix C.

The sequential cost of the computation whose BSP parallelisation was described in this section is $n^3(\mathrm{cost}(f1) + \mathrm{o}(1))/p$, where $\mathrm{cost}(f1)$ represents the cost of computing function $f1$ for a single point of the iteration space. The BSP cost of the resulting schedule is

$$n(1 + \mathrm{o}(1))L + \frac{n^3(\mathrm{cost}(f1) + \mathrm{o}(1))}{p} + gn^2(1 + \mathrm{o}(1))$$

which approaches one–optimality when the problem size n is large enough to ensure that the computation cost dominates the communication and the synchronisation overheads (i.e., $n \gg pg/\mathrm{cost}(f1)$, $n \gg \sqrt{pL/\mathrm{cost}(f1)}$). Furthermore, the BSP schedule is one–optimal in its space complexity, requiring $n^3(1+\mathrm{o}(1))/p$ space compared to the $n^3(1+\mathrm{o}(1))$ space complexity of the sequential loop nest. The BSP code in Appendix C was tested on a 4–processor SGI Power Challenge shared–memory computer, and the experimental results (Figure 15) confirm the efficiency of the automatically generated BSP schedule.

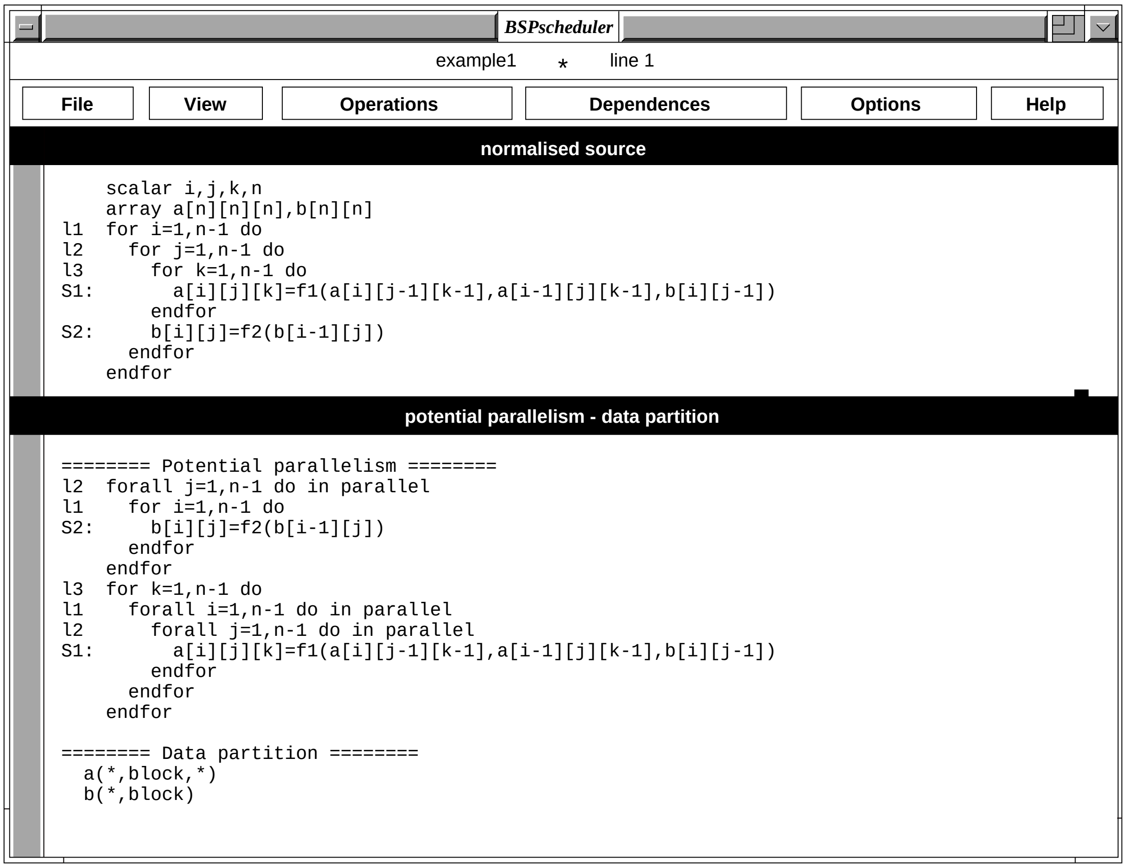


Figure 14: The intermediate results pane shows the potential parallelism of the sequential code, and the data partition used in the generation of the BSP schedule.

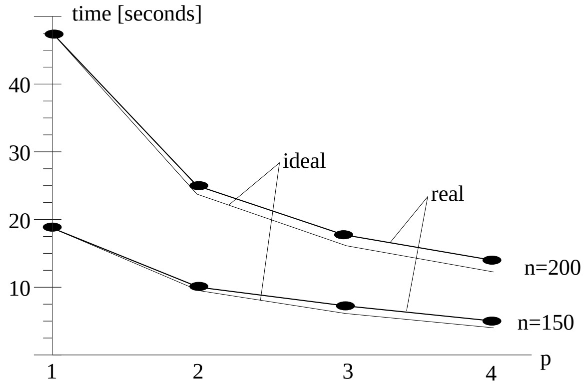


Figure 15: The execution time of the BSP code in Appendix C on a 4–processor SGI Power Challenge multiprocessor; the real execution time closely approaches the ideal one for both problem sizes considered in the diagram.

5 Conclusions and further work

The BSP tool presented in the current report approaches the automatic parallelisation of nested loops. By implementing a two-phase, hybrid scheduling strategy, the tool succeeds in exploiting the existing knowledge on the efficient BSP parallelisation of nested loops that match certain computation templates, being at the same time able to parallelise generic loop nests. The tool is able to produce both BSP pseudocode—for the facile examination and analysis of the parallel schedule, and real parallel code—for immediate inclusion into a user application. Most important, being targeted at the BSP computation model, the automatically generated parallel code is scalable, portable, and its cost can be analytically analysed, characteristics which are unique among the existent parallelising compilers and tools.

Due to its modular structure, the various blocks of the tool can be independently developed to include new features. The remainder of this section lists several of these possible improvements. Firstly, the template–matching scheduling module can be upgraded to implement more of the scheduling techniques described in [10], and perhaps other techniques as well. At the same time, both scheduling modules should consider the block–cyclic distribution of data in cases when non–rectangular iteration spaces are parallelised and a simple block distribution leads to unbalanced schedules. Also, array alignment in the partitioned dimensions should be considered as a means to eliminate unnecessary statement masking and to decrease communication costs [18].

Other improvements should be directed at the dependence analysis module, which currently uses only part of the available dependence tests; in particular, the inclusion of symbolic analysis and more exact dependence tests are deemed necessary. Finally, the addition of a complexity analysis module implementing the performance analysis algorithm described in [11] would significantly increase the usefulness of the BSP scheduling tool.

Acknowledgements

The author would like to thank Prof. W.F. McColl for his continuous advice and encouragement. I am also indebted to Prof. D.J. Evans, and to the members of the Oxford BSP research group for their valuable comments on my work.

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

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

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

[7] P. Boulet et al. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, August 1994.

[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] R. Calinescu. A scheme for the BSP scheduling of generic loop nests. Technical Report PRG-TR-26-97, Programming Research Group, Oxford University Computing Laboratory, August 1997.

[12] D. Culler et al. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 19–22. ACM Press, 1993.

[13] 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.

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

[15] Jonathan M. D. Hill, Paul I. Crumpton, and David A. Burgess. Theory, practice, and a tool for BSP performance prediction. In *EuroPar'96*, number 1124 in Lecture Notes in Computer Science, pages 697–705, Lyon, France, August 1996. Springer-Verlag.

[16] 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.

[17] Jonathan M.D. Hill, Stephen Jarvis, Constantinos Siniolakis, and Vasil P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*. IEEE Computer Society Press, January 1998.

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

[19] 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.

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

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

[22] 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.

[23] A.W. Lim and M.S. Lam. Communication-free parallelization via affine transformations. In K. Pingali et al., editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 892, pages 92–106. Springer-Verlag, 1995.

[24] 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.

[25] 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.

[26] J. McCormack et al. *X toolkit : the intrinsics & Athena Widgets.* Silicon Press, 1990.

[27] J.M. Nash et al. Parallel algorithm design on the WPRAM model. Technical Report 94.24, Division of Computer Science, University of Leeds, July 1994.

[28] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, October 1992.

[29] R. Schreiber and J.J. Dongarra. Automatic blocking of nested loops. Technical Report 90-38, Univ. of Tennessee at Knoxville, May 1990.

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

[31] 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.

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

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

[34] M.J. Wolfe. More iteration space tiling. In *Proceedings Supercomputing '89*, pages 655–664. ACM Press, 1989.

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

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

Appendix A: Syntax of the BSPscheduler input language

The following notation is used to define the syntax of the BSPscheduler input language. Non–terminal symbols are indicated by *italic* type. Terminal symbols are indicated by **bold** type. A colon : following a non–terminal introduces its definition. Alternative definitions are separated by a vertical line |.

BSPscheduler-input	:	*var-declarations loop-nest*
var-declarations	:	*var-declaration* | *var-declaration var-declarations*
var-declaration	:	*array-declaration* | *scalar-declaration*
array-declaration	:	**array** *array-decl-list*
array-decl-list	:	*array-decl* | *array-decl* **,** *array-decl-list*
array-decl	:	*identifier size-decl-list*
identifier	:	*letter* | *identifier letter* | *identifier digit*
letter	:	**a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**
digit	:	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
size-decl-list	:	*size-decl* | *size-decl size-decl-list*
size-decl	:	[*pos-integer-const*] | [*identifier*]
pos-integer-const	:	*nonzero-digit* | *nonzero-digit digits*

nonzero-digit : **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

digits : *digit* |
digit digits

scalar-declaration : **scalar** *scalar-decl-list*

scalar-decl-list : *identifier* |
identifier **,** *scalar-decl-list*

loop-nest : *for-stmt*

for-stmt : **for** *identifier* **=** *aritm-expr* **,** *aritm-expr* **do** *stmts* **endfor**

stmts : *stmt* |
stmt stmts

stmt : *for-stmt* |
if-stmt |
assign-stmt

if-stmt : **if** *expr* **then** *assign-stmts* **endif** |
if *expr* **then** *assign-stmts* **else** *assign-stmts* **endif**

expr : **not** *expr* |
logic-term |
logic-term **or** *expr*

logic-term : *logic-factor* |
logic-factor **and** *logic-term*

logic-factor : *rel-expr*

rel-expr : *aritm-expr* |
aritm-expr rel-operator aritm-expr

rel-operator : < | > | = | <= | >= | <>

aritm-expr : *aritm-term* |
− *aritm-term* |
aritm-expr + *aritm-term* |
aritm-expr − *aritm-term*

aritm-term	:	*aritm-factor* |
		aritm-term $*$ *aritm-factor* |
		aritm-term / *aritm-factor*
aritm-factor	:	*integer-const* |
		float-const |
		variable |
		(*expr* **)** |
		function-call
integer-const	:	**0** |
		pos-integer-const |
		$-$ *pos-integer-const*
float-const	:	*integer-const* **.** *digits*
variable	:	*identifier* |
		identifier subscripts
subscripts	:	*subscript* |
		subscript subscripts
subscript	:	[*expr*]
function-call	:	*identifier* **( )** |
		identifier **(** *parameter-list* **)**
parameter-list	:	*expr* |
		expr **,** *parameter-list*
assign-stmts	:	*assign-stmt* |
		assign-stmt assign-stmts
assign-stmt	:	*variable* **=** *expr*

Appendix B: Syntax of the BSPscheduler output language

The language used to describe the BSP pseudocode generated by BSPscheduler differs from the input language of the tool in the definition of the *stmt* non–terminal symbol. The notations introduced in Appendix A are used to define the modified syntax of this non–terminal symbol. Non–terminal symbols whose definitions are identical to the definitions provided in Appendix A are no longer defined here.

stmt	:	*for-stmt* | *if-stmt* | *assign-stmt* | **BSP_sync** | *BSP-get-stmt* | *BSP-combine-stmt*
if-stmt	:	**if** *expr* **then** *assign-stmt* **endif** | **if** *expr* **then** *BSP-get-stmt* **endif**
BSP-get-stmt	:	**BSP_get** *variable* **for (** *index-list* **) in (** *index-range-list* **)** **from** *processor*
identifier	:	*letter* | *identifier letter* | *identifier digit* | _ *identifier*
index-list	:	*identifier* | *identifier* **,** *index-list*
index-range-list	:	*index-range* | *index-range* **,** *index-range-list*
index-range	:	*aritm-expr* **..** *aritm-expr* | *aritm-expr* **..** *aritm-expr* **;** *pos-integer-const*
processor	:	**_proc** *pos-integer-const* **(** *processor-coordinates* **)**
processor-coordinates	:	*aritm-expr* | *aritm-expr* **,** *processor-coordinates*
BSP-combine-stmt	:	**BSP_combine (** *variable* **,** *combine-operator* **)**
combine-operator	:	**PLUS** | **MULT** | **OR** | **AND**

Appendix C: The C code generated by BSPscheduler for the sample loop nest in Section 4

```
/*************************************************************************
*
* This code was generated by BSPscheduler 1.0
*
* Here is an account of the array distributions and of the locally owned
* array blocks:
*
* > a[*][block][*]:
*   - processor _me owns (and, if necessary, must initialise)
*     a[0..n-1]
*      [_me*_ceil(n,_p)..(_me+1)*_ceil(n,_p)-1]
*      [0..n-1]
*   - element a[i1][i2][i3] can be referred to as
*     a[_lin3(i1,
*             i2-_me*_ceil(n,_p)+1,
*             i3,
*             (_ceil(n,_p)+1),
*             n)]
*
* > b[*][block]:
*   - processor _me owns (and, if necessary, must initialise)
*     b[0..n-1]
*      [_me*_ceil(n,_p)..(_me+1)*_ceil(n,_p)-1]
*   - element b[i1][i2] can be referred to as
*     b[_lin2(i1,
*             i2-_me*_ceil(n,_p)+1,
*             (_ceil(n,_p)+1))]
*
*************************************************************************/

#include <stdlib.h>

#include "bsp.h"

#define _floor(x) ((int)(x))
#define _ceil(x,y)  ((x+y-1)/(y))
#define _min2(x1,x2) (((x1)<(x2))?(x1):(x2))
#define _max2(x1,x2) (((x1)>(x2))?(x1):(x2))
#define _lin2(i1,i2,n2) ((i1)*(n2)+(i2))
#define _lin3(i1,i2,i3,n2,n3) (((i1)*(n2)+(i2))*(n3)+(i3))

void initialise_data(int* i,int* j,int* k,float* a,float* b,int n,
int _p,int _me){

}
```

```
void output_data(int i,int j,int k,const float* a,const float* b,
int n,int _p,int _me){

}

float f1(float _x1,float _x2,float _x3){

}

float f2(float _x1){

}

void parallel_code(int n,int _p){
  int i,j,k,_me,_s1,_s2,_aux;
  float *a,*b;
  _me=bsp_pid();
  a=(float*)calloc(n*(_ceil(n,_p)+1)*n,sizeof(float));
  b=(float*)calloc(n*(_ceil(n,_p)+1),sizeof(float));
  initialise_data(&i,&j,&k,a,b,n,_p,_me);
  bsp_push_reg(a,n*(_ceil(n,_p)+1)*n*sizeof(float));
  bsp_push_reg(b,n*(_ceil(n,_p)+1)*sizeof(float));
  bsp_sync();
  for(j=_max2(1,_me*_ceil(n,_p));j<=_min2(n-1,(_me+1)*_ceil(n,_p)-1);j++)
    for(i=1;i<=n-1;i++)
      b[_lin2(i,j-_me*_ceil(n,_p)+1,(_ceil(n,_p)+1))]=
             f2(b[_lin2(i-1,j-_me*_ceil(n,_p)+1,(_ceil(n,_p)+1))]);
  for(_s1=1;_s1<=n-1;_s1++)
    for(_s2=_max2(0,_me*_ceil(n,_p)-1);_s2<=_me*_ceil(n,_p)-1;_s2++)
      bsp_get(
        _floor(_s2/_ceil(n,_p)),
        b,
        (_lin2(_s1,_s2-(_floor(_s2/_ceil(n,_p)))*_ceil(n,_p)+1,
                (_ceil(n,_p)+1)))*sizeof(float),
        &b[_lin2(_s1,_s2-_me*_ceil(n,_p)+1,(_ceil(n,_p)+1))],
        sizeof(float));
  bsp_sync();
  for(k=1;k<=n-1;k++){
    for(_s1=1;_s1<=n-1;_s1++)
      for(_s2=_max2(0,_me*_ceil(n,_p)-1);_s2<=_me*_ceil(n,_p)-1;_s2++)
        bsp_get(
          _floor(_s2/_ceil(n,_p)),
          a,
          (_lin3(_s1,_s2-(_floor(_s2/_ceil(n,_p)))*_ceil(n,_p)+1,k-1,
                 (_ceil(n,_p)+1),n))*sizeof(float),
          &a[_lin3(_s1,_s2-_me*_ceil(n,_p)+1,k-1,(_ceil(n,_p)+1),n)],
          sizeof(float));
    bsp_sync();
```

```
    for(i=1;i<=n-1;i++)
      for(j=_max2(1,_me*_ceil(n,_p));j<=_min2(n-1,(_me+1)*_ceil(n,_p)-1);j++)
        a[_lin3(i,j-_me*_ceil(n,_p)+1,k,(_ceil(n,_p)+1),n)]=
            f1(a[_lin3(i,j-1-_me*_ceil(n,_p)+1,k-1,(_ceil(n,_p)+1),n)],
               a[_lin3(i-1,j-_me*_ceil(n,_p)+1,k-1,(_ceil(n,_p)+1),n)],
               b[_lin2(i,j-1-_me*_ceil(n,_p)+1,(_ceil(n,_p)+1))]);
  }
  output_data(i,j,k,a,b,n,_p,_me);
  bsp_pop_reg(a);
  bsp_pop_reg(b);
  bsp_sync();
  free(a);
  free(b);
}
```