

BSF-skeleton: user manual

Leonid B. Sokolinsky

*South Ural State University (national research university),
76, Lenin prospekt, Chelyabinsk, Russia, 454080
E-mail: leonid.sokolinsky@susu.ru*

Content

1. Main specifications and application scope	3
2. Theoretical basis	3
3. Source code structure of BSF-skeleton.....	5
4. Predefined problem-dependent parameters.....	6
5. Predefined BSF types.....	7
6. Extended reduce-list.....	7
7. Skeleton variables	7
8. Functions.....	8
8.1 Key problem-independent functions (prefix <i>BC_</i>).....	8
8.2 Predefined problem-dependent bsf-functions (prefix <i>PC_bsf_</i>)	9
8.2.1 PC_bsf_CopyParameter	9
8.2.2 PC_bsf_Init	9
8.2.3 PC_bsf_IterOutput	10
8.2.4 PC_bsf_MapF	10
8.2.5 PC_bsf_ParametersOutput	11
8.2.6 PC_bsf_ProblemOutput	11
8.2.7 PC_bsf_ProcessResults.....	12
8.2.8 PC_bsf_ReduceF.....	13
8.2.9 PC_bsf_SetListSize.....	14
8.2.10 PC_bsf_SetInitParameter	14
8.2.11 PC_bsf_SetMapSubList.....	14
8.2.12 PC_bsfAssignAddressOffset.....	15
8.2.13 PC_bsfAssignIterCounter	15
8.2.14 PC_bsfAssignJobCase	15
8.2.15 PC_bsfAssignMpiRank.....	16
8.2.16 PC_bsfAssignNumberInSublist	16
8.2.17 PC_bsfAssignNumOfWorkers.....	16
8.2.18 PC_bsfAssignParameter	16
8.2.19 PC_bsfAssignSublistLength	16

9.	Step-by-step instruction	17
10.	Example of using the BSF-skeleton: Jacobi algorithm.....	17
11.	Workflow support	19
12.	Using OpenMP.....	19
13.	Applying BSF model to Jacobi method	19
14.	Using Map without Reduce	19
	References	19

1. Main specifications and application scope

The BSF-skeleton is designed for creating parallel programs in C++ using the MPI library. The scope of the BSF-skeleton is cluster computing systems and iterative numerical algorithms of high computational complexity. The BSF-skeleton completely encapsulates all aspects that are associated with parallelizing a program on a cluster computing system. The source code of the BSF-skeleton is freely available on Github at <https://github.com/leonid-sokolinsky/BSF-skeleton>.

2. Theoretical basis

The theoretical basis of the BSF-skeleton is the BSF (Bulk Synchronous Farm) model of parallel computations [1], which allows predicting the scalability boundary* of a parallel algorithm/program at an early stage of its development.

The BSF-skeleton uses the master/worker (master/slave) paradigm to organize interaction between MPI processes (see Fig. 1). This means that worker processes can only exchange messages with the master process.

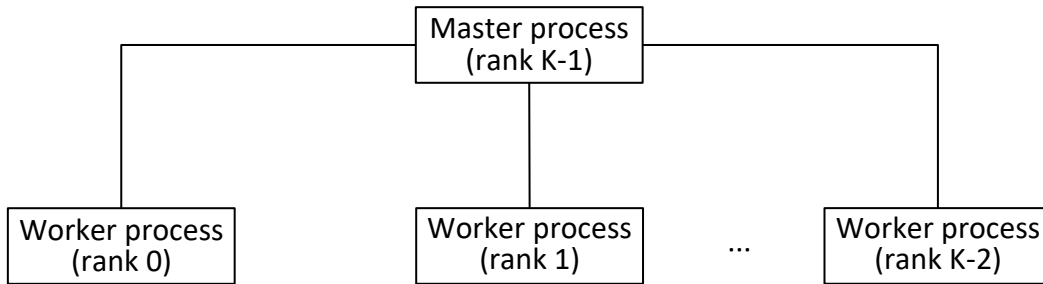


Fig. 1. Interaction of K MPI processes in the BSF-skeleton.

To use the BSF-skeleton, you must represent your algorithm in the form of operations on lists using the higher-order functions *Map* and *Reduce* [2]. The higher-order function $Map(f, \mathbb{A})$ applies the function f to each element of list $\mathbb{A} = [a_1, \dots, a_n]$ converting it to the list $\mathbb{B} = [f(a_1), \dots, f(a_n)]$. The higher-order function $Reduce(\oplus, \mathbb{B})$ taking an associative binary operation \oplus and a list $\mathbb{B} = [b_1, \dots, b_n]$ as parameters calculates the element $b = b_1 \oplus \dots \oplus b_n$. You should use the template shown in Fig. 2 to represent your algorithm in sequential form. The BSF skeleton encapsulates (contains inside) all actions related to parallelizing the algorithm using the MPI library. Let us comment on Algorithm 1.

Algorithm 1.	
1:	input $A, x^{(0)}$
2:	$i := 0$
3:	$B := Map(F_{x^{(i)}}, A)$
4:	$s := Reduce(\oplus, B)$
5:	$x^{(i+1)} := Compute(x^{(i)}, s)$
6:	$i := i + 1$
7:	if $StopCond(x^{(i)}, x^{(i-1)})$ goto 9
8:	goto 3
9:	output $x^{(i)}$
10:	stop

Fig. 2. Generic BSF-algorithm template.

* The scalability boundary is the maximum number of processor nodes that speedup increases to.

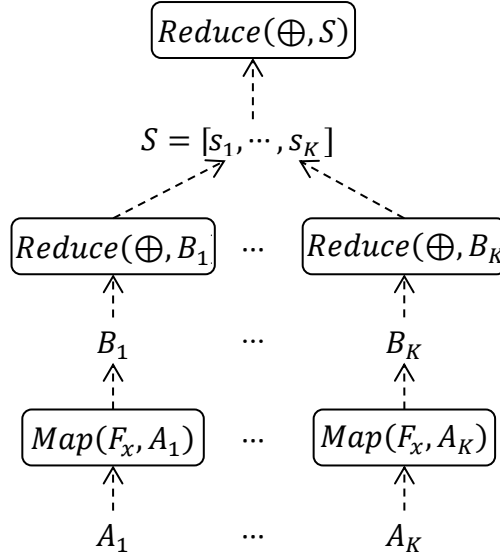


Fig. 3. BSF-skeleton parallelization schema.

The variable i denotes the iteration number; $x^{(0)}$ is an initial approximation; $x^{(i)}$ is the i -th approximation (the approximation can be a number, a vector, or any other data structure); A is the list of elements of a certain set \mathbb{A} , which represents the source data of the problem; $F_x: \mathbb{A} \rightarrow \mathbb{B}$ is a parameterized user function (the parameter x is the current approximation) that maps the set \mathbb{A} to a certain set \mathbb{B} ; B is a list of elements of the set \mathbb{B} calculated by applying the function F_x to each element of the list A ; \oplus is a binary associative operation on the set \mathbb{B} . Step 1 reads the input data of the problem and the initial approximation. Step 2 assigns the zero value to the iteration counter i . Step 3 calculates the list B by invoking the higher-order function $Map(F_{x^{(i)}}, A)$. Step 4 assigns the result of the higher-order function $Reduce(\oplus, B)$ to the intermediate variable s . Step 5 invokes the user function $Compute$ that calculates the next approximation $x^{(i+1)}$ taking two parameters: the current approximation $x^{(i)}$ and the result s of the higher-order function $Reduce$. Step 6 increases the iteration counter i by one. Step 7 checks a termination criteria by invocation of the user Boolean function $StopCond$, which takes two parameters: the new approximation $x^{(i)}$ and the previous approximation $x^{(i-1)}$. If $StopCond$ returns true, the algorithm outputs $x^{(i)}$ as an approximate problem solution and stops working. Otherwise, the control is passed to Step 3 starting the next iteration.

The BSF-skeleton automatically parallelizes Algorithm 1 by splitting the list A into K sublists of equal length (± 1):

$$A = A_1 \# \dots \# A_K,$$

where K is the number of worker processes and $\#$ denotes the operation of list concatenation. This uses the parallelization scheme shown in Fig. 3. The result is the parallel algorithm, shown in Fig. 4. It includes $K+1$ parallel processes: one master process and K worker processes. In Step 2, the master process sends the current approximation $x^{(i)}$ to all worker processes. After that, every j -th worker process independently applies higher-order function Map and $Reduce$ to its sublist (the steps 3 and 4). In the steps 3 and 4, the master process is idle. In Step 5, all worker processes send the partial foldings s_1, \dots, s_K to the master process. In the steps 6-9, the master process performs the following actions: executes the higher-order function $Reduce$ over the list of partial foldings $[s_1, \dots, s_K]$; invokes the user function $Compute$ that calculates the next approximation; checks the termination criteria by using the user Boolean function $StopCond$ and assigns its result to the Boolean variable $exit$. In the steps 6-9, the worker processes are idle. In Step 10, the master process sends the $exit$ value to all worker-processes. If the $exit$ value is false, the master process and worker processes go to the next iteration, otherwise the master processes outputs the result and the computation stops. Note that, in the steps 2 and 10, all processes perform the implicit global synchronization.

Algorithm 2.	
Master	j -th worker ($j=1,\dots,K$)
1: input $x^{(0)}$; $i := 0$	1: input A_j
2: $\text{SendToAllWorkers}(x^{(i)})$	2: $\text{RecvFromMaster}(x^{(i)})$
3:	3: $B_j := \text{Map}(F_{x^{(i)}}, A_j)$
4:	4: $s_j := \text{Reduce}(\oplus, B_j)$
5: $\text{RecvFromWorkers}(s_1, \dots, s_K)$	5: $\text{SendToMaster}(s_j)$
6: $s := \text{Reduce}(\oplus, [s_1, \dots, s_K])$	6:
7: $x^{(i+1)} := \text{Compute}(x^{(i)}, s)$	7:
8: $i := i + 1$	8:
9: $\text{exit} := \text{StopCond}(x^{(i)}, x^{(i-1)})$	9:
10: $\text{SendToAllWorkers}(\text{exit})$	10: $\text{RecvFromMaster}(\text{exit})$
11: if exit goto 2	11: if exit goto 2
12: output $x^{(i)}$	12:
13: stop	13: stop

Fig. 4. BSF-skeleton parallelization template.

3. Source code structure of BSF-skeleton

The BSF-skeleton is a compilable but not executable set of files. This set is divided into two groups:

- 1) files with the “*BSF*” prefix contain problem-independent code and are not subject to changes by the user;
- 2) files with the “*Problem*” prefix are intended for filling in problem-dependent parts of the program by the user.

Descriptions of all source code files are given in Table 1.

Table 1. Source code files of the BSF-skeleton.

File	Description
<i>Problem-independent code</i>	
<i>BSF-Code.cpp</i>	Implementations of the <i>main</i> function and all problem-independent functions
<i>BSF-Data.h</i>	Problem-independent variables and data structures
<i>BSF-Forwards.h</i>	Declarations of the problem-independent functions
<i>BSF-Include.h</i>	The inclusion of problem-independent libraries
<i>BSF-SkeletonVariables.h</i>	Definitions of the skeleton variables (see Section 7)
<i>BSF-ProblemFunctions.h</i>	Declarations of predefined functions with problem-dependent implementation
<i>BSF-Types.h</i>	Definitions of problem-independent types
<i>Problem-dependent code</i>	
<i>Problem-bsfCode.cpp</i>	Implementations of the problem-dependent functions (see Section 8)
<i>Problem-bsfParameters.h</i>	Predefined problem-dependent parameters (see Section 4)
<i>Problem-bsfTypes.h</i>	Predefined BSF types (see Section 5)
<i>Problem-Data.h</i>	Problem-dependent variables and data structures
<i>Problem-Forwards.h</i>	Declarations of the problem-dependent functions
<i>Problem-Include.h</i>	Inclusion of problem-dependent libraries
<i>Problem-Parameters.h</i>	Parameters of the problem
<i>Problem-Types.h</i>	BSF types

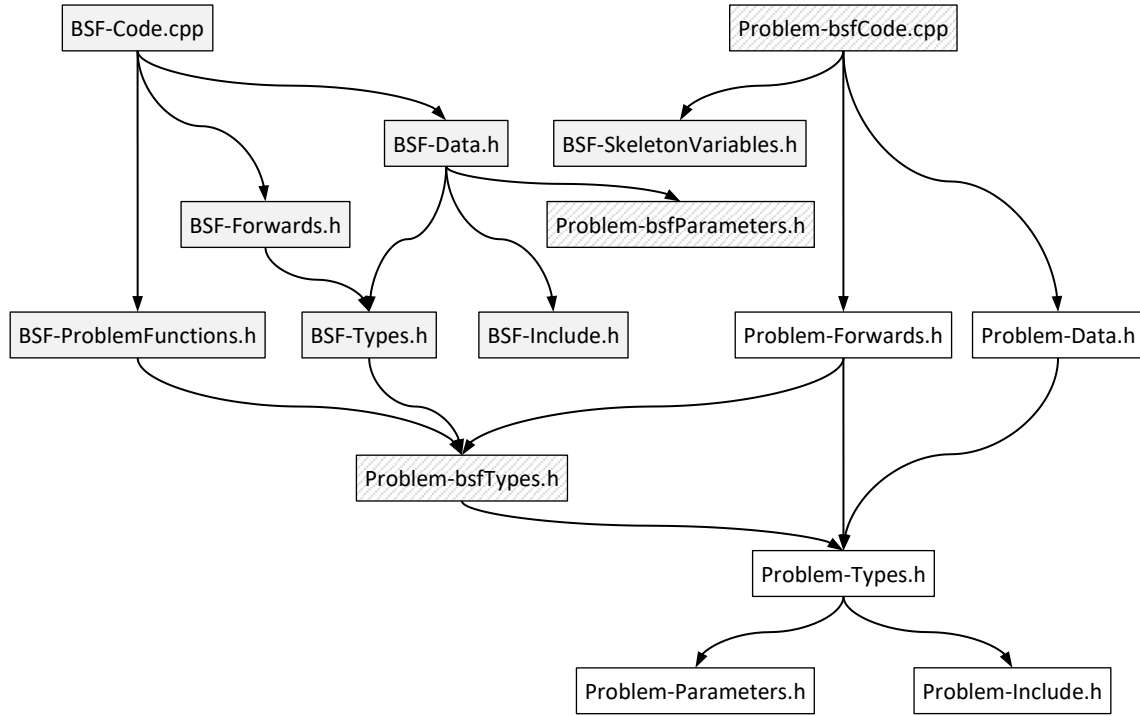


Fig. 5. Dependency graph of the source code files by the directive `#include`.

The dependency graph of the source code files by the directive `#include` is shown in Fig. 1. The gray rectangles indicate the code files that do not allow changes. The rectangles with striped shading indicate the code files containing predefined declarations that must be defined (filled in) by the user. The white rectangles indicate the code files that should be fully implemented by the user.

4. Predefined problem-dependent parameters

The predefined problem-dependent parameters are declared as macroses in the file *Problem-bsfParameters.h*. They are used in the *BSF-Code.cpp* and should be set by the user. All these parameters are presented in Table 2.

Table 2. Predefined problem-dependent parameters.

ID	Description	Default value
<i>PP_MAX_MPI_SIZE</i>	Defines the maximum possible number of MPI processes (the result returned by the function <i>MPI_Comm_size</i> cannot exceed this number).	500
<i>PP_BSF_PRECISION</i>	Sets the decimal precision to be used to format floating-point values on output operations.	4
<i>PP_BSF_ITER_OUTPUT</i>	If this macros is defined, at the end of each <i>k</i> -th iteration, the master process will invoke the predefined bsf-function <i>PC_bsf_IterOutput</i> that outputs intermediate results. The number <i>k</i> is defined by the macros <i>PP_BSF_TRACE_COUNT</i> .	#undef
<i>PP_BSF_TRACE_COUNT</i>	Defines the number <i>k</i> mentioned in the description of the macros <i>PP_BSF_ITER_OUTPUT</i> .	1
<i>PP_BSF_MAX_JOB_CASE</i>	Defines the maximum number of activities (jobs) in workflow minus 1. See “Workflow Support” in Section 11.	0
<i>PP_BSF_FRAGMENTED_MAP_LIST</i>	If this macros is defined, each worker process stores only its part of the map-list [†] . Otherwise, each worker process stores the entire map-list.	#define
<i>PP_BSF_OMP</i>	If this macros is defined, the worker processes use <i>#pragma omp parallel for</i> to perform the higher-order function <i>Map</i> .	#undef
<i>PP_BSF_NUM_THREADS</i>	If this macros is defined, <i>omp parallel for</i> uses the specified number of threads to perform the higher-order function <i>Map</i> . If this macros is defined, <i>omp parallel for</i> uses the maximum possible number of threads.	#undef

[†] *Map-list* is the list being the second parameter of the higher-order function *Map*.

5. Predefined BSF types

The predefined BSF types are declared as data structures in the file *Problem-bsfTypes.h*. They are used in the *BSF-Code.cpp* and should be set by the user. All types parameters are presented in Table 3.

Table 3. Predefined BSF types (file *Problem-bsfTypes.h*).

Type ID	Data type	Description	Mandatory to fill in
<i>PT_bsf_parameter_T</i>	struct	Defines the structure (set of data elements) that is transferred by the master process to all the worker processes in Step 2 of Algorithm 2 (see Fig. 4) and includes the order parameters (usually the current approximation).	Yes
<i>PT_bsf_mapElem_T</i>	struct	Defines the record that represents an item in the map-list (list <i>A</i> in Algorithm 1).	Yes
<i>PT_bsf_reduceElem_T</i>	struct	Defines the record that represents an item in the reduce-list [‡] (list <i>B</i> in Algorithm 1).	Yes
<i>PT_bsf_reduceElem_T_1</i> , <i>PT_bsf_reduceElem_T_2</i> , <i>PT_bsf_reduceElem_T_3</i>	struct	Alternative types of the reduce-list items that are used to organize the workflow (see section 11).	No

6. Extended reduce-list

The BSF-skeleton appends to each element of the reduce-list the additional integer field called *reduceCounter*. This extended reduce-list is presented by the pointer *BD_extendedReduceList* declared in *BSF-Data.h*. When performing the *Reduce* function (see *BC_ProcessExtendedReduceList* in Section 8.1), the elements that have this field equal to zero are ignored. For elements where *reduceCounter* is not zero, the values of the *reduceCounter* are added together. By default, the function *BC_WorkerMap* (see Section 8.1) sets the *reduceCounter* to 1. The user can set the value of this field to 0 by setting the parameter **success* of the function *PC_bsf_MapF* (see Section 8.2.3) to 0.

7. Skeleton variables

The skeleton variables are declared in the file *BSF-SkeletonVariables.h*. The user can exploit these variables for the sake of debugging, tracing, and non-standard implementing (see, for example, Section 14). The user should not change the values of these variables. All skeleton variables are presented in Table 4.

Table 4. Skeleton variables (file *BSF-SkeletonVariables.h*).

Skeleton variable	Type	Description
<i>BSF_sv_addressOffset</i>	<i>int</i>	Contains the number of the first element of the map-sublist appointed to the current worker process.
<i>BSF_sv_iterCounter</i>	<i>int</i>	Contains the number of iterations performed so far.
<i>BSF_sv_jobCase</i>	<i>int</i>	Contains the number of the current activity (job) in workflow (see Section 11).
<i>BSF_sv_mpiRank</i>	<i>int</i>	Contains the rank (number) of current MPI process.
<i>BSF_sv_numberInSublist</i>	<i>int</i>	If the macros <i>PP_BSF_FRAGMENTED_MAP_LIST</i> (see Section 4) is defined then this variable contains the relative number of the element in the map-sublist that the function <i>Map</i> is currently applied to. Otherwise, this variable contains the absolute number of the map-list element that the function <i>Map</i> is currently applied to.
<i>BSF_sv_numOfWorkers</i>	<i>int</i>	Contains the total number of the worker processes.
<i>BSF_sv_parameter</i>	<i>PT_bsf_parameter_T</i>	Structure that contains the order parameters.
<i>BSF_sv_sublistLength</i>	<i>int</i>	Contains the length of a map-sublist appointed to a worker process.

[‡] *Reduce-list* is the list being the second parameter of the higher-order function *Reduce*.

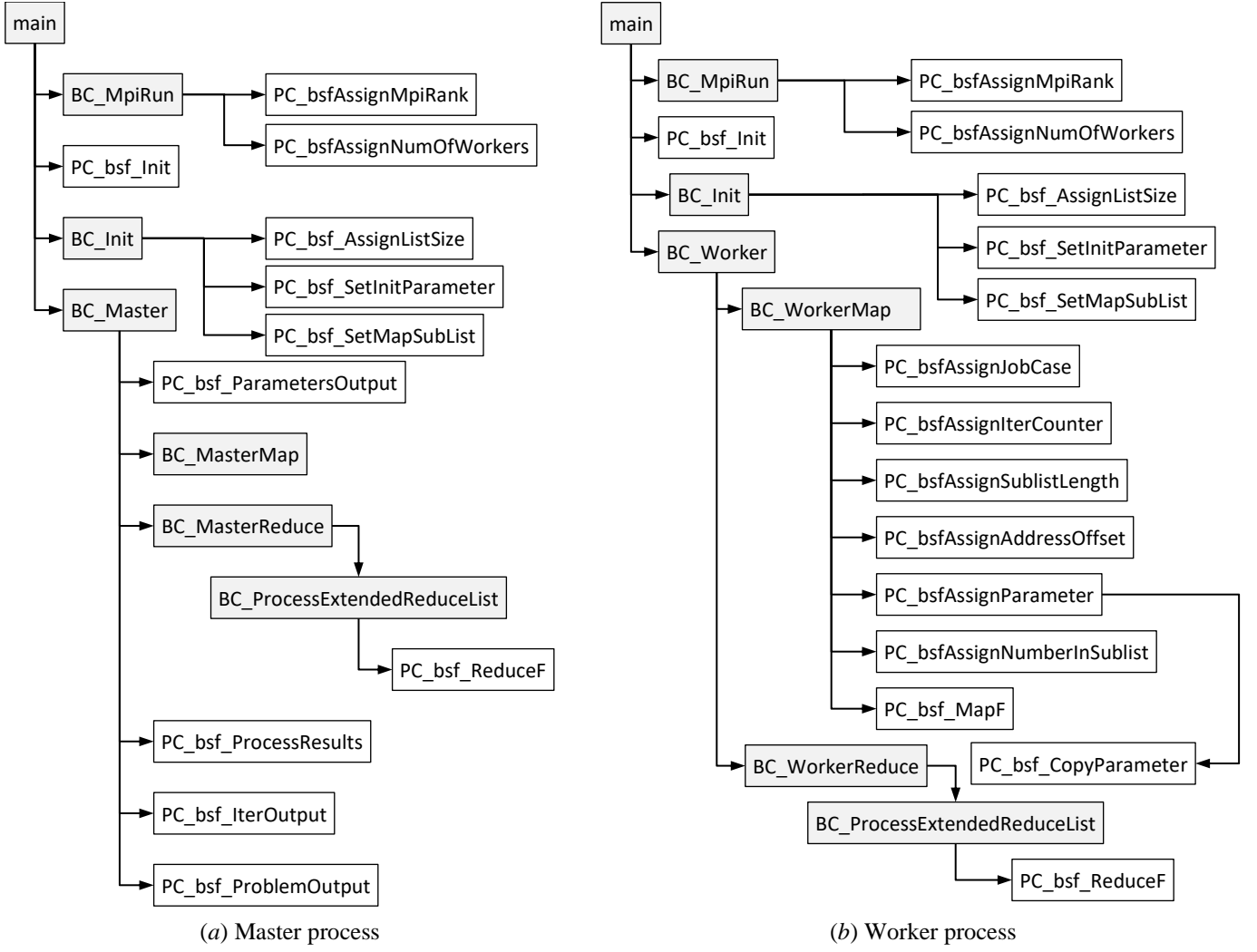


Fig. 6. Hierarchy of the key function calls.

8. Functions

The skeleton functions are divided into two groups:

- 1) problem-independent functions with the prefix *BC_* that have implemented in the file *BSF-Code.cpp*;
- 2) problem-dependent functions (*predefined bsf-functions*) with the prefix *PC_bsf_* that have declared in the file *Problem-Code.cpp*.

The user cannot change the headers and bodies of the functions with the prefix *BC_*. The user also cannot change function headers with the prefix *PC_bsf_* but must write an implementation of these functions. The body of a predefined bsf-function cannot include calls of problem-independent functions with the prefix *BC_*. The hierarchy of the key function calls is presented in Fig. 6.

8.1 Key problem-independent functions (prefix *BC_*)

The implementations of all problem-independent functions can be found in the file *BSF-Code.cpp*. Descriptions of some key problem-independent functions are presented in Table 5.

Table 5. Key problem-independent functions (file *BSF-Code.cpp*).

Function	Description
<i>BC_Init</i>	Performs the memory allocation and the initialization of the skeleton data structures and variables.
<i>BC_Master</i>	The head function of the master process.
<i>BC_MasterMap</i>	Forms an order and sends it to the worker processes to perform the Map function in the current iteration.
<i>BC_MasterReduce</i>	Receives the results from worker processes, collects them in a list, and performs the function Reduce on this list.
<i>BC_MpiRun</i>	Executes the MPI initialization. After it, the number of worker processes is accessible by the skeleton variable <i>BSF_sv_numOfWorkers</i> ; total number of MPI processes (<i>MPI_Comm_size</i>) is equal to (<i>BSF_sv_numOfWorkers</i> + 1); the rank of a MPI process (<i>MPI_Comm_rank</i>) is accessible by the skeleton variable <i>BSF_sv_mpiRank</i> . The MPI ranks of the worker processes have values from 0 to (<i>BSF_sv_numOfWorkers</i> – 1). The MPI rank of the worker process is equal to <i>BSF_sv_numOfWorkers</i> .
<i>BC_ProcessExtendedReduceList</i>	This function finds the first element in the extended reduce-list with the <i>reduceCounter</i> not equal to zero and adds to it all other elements that have the <i>reduceCounter</i> not equal to zero. For pairwise addition of elements of the original reduce-list, the function <i>PC_bsf_ReduceF</i> (see Section 8.2.8) is used.
<i>PC_bsf_ReduceF</i>	The head function of a worker process.
<i>BC_WorkerMap</i>	Receives an order from the master process, assigns the skeleton variables (see Section 7), and applies the function <i>PC_bsf_MapF</i> to the appointed map-sublist producing the corresponding part of the reduce-list.
<i>BC_WorkerReduce</i>	Sends to the master process the element that is the sum of all reduce-sublist elements.

8.2 Predefined problem-dependent bsf-functions (prefix *PC_bsf_*)

This section contains detailed descriptions of the predefined problem-dependent bsf-functions with the prefix *PC_bsf_* declared in *Problem-bsfCode.cpp*. The user must implement all these functions. Step-by-step instruction is presented in Section 8.2.9. An example is presented in Section 10.

8.2.1 *PC_bsf_CopyParameter*

Copies order parameters from the in-structure to the out-structure.

Syntax

```
void PC_bsf_CopyParameter(
    PT_bsf_parameter_T parameterIn,
    PT_bsf_parameter_T* parameterOutP
);
```

In parameters

parameterIn

The structure from which parameters are copied.

Out parameters

parameterOutP

The pointer to the structure to which parameters are copied.

8.2.2 *PC_bsf_Init*

Initializes the problem-depended variables and data structures defined in *Problem-Data.h*.

Syntax

```
void PC_bsf_Init(
    bool* success
);
```

Out parameters

**success*

Must be set to *false* if the initialization failed. The default value is *true*.

8.2.3 PC_bsf_IterOutput

Outputs intermediate results of the current iteration.

Syntax

```
void PC_bsf_IterOutput(  
    PT_bsf_reduceElem_T* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T parameter,  
    double elapsedTime,  
    int newJobCase  
);  
void PC_bsf_IterOutput_1(  
    PT_bsf_reduceElem_T_1* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T parameter,  
    double elapsedTime,  
    int newJobCase  
);  
void PC_bsf_IterOutput_2(  
    PT_bsf_reduceElem_T_2* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T parameter,  
    double elapsedTime,  
    int newJobCase  
);  
void PC_bsf_IterOutput_3(  
    PT_bsf_reduceElem_T_3* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T parameter,  
    double elapsedTime,  
    int newJobCase  
);
```

In parameters

reduceResult

Pointer to the structure that contains the result of executing the Reduce function.

reduceCounter

The number of summed (by \oplus) elements in the reduce-list. This number matches the number of extended reduce-list elements that have the value 1 in the field *reduceCounter* (see Section 6).

Remarks

The functions *PC_bsf_IterOutput_1*, *PC_bsf_IterOutput_2* and *PC_bsf_IterOutput_3* are used to organize a workflow (optional).

8.2.4 PC_bsf_MapF

Implements the function that is applied to the map-list elements when performing the higher-order function *Map*.

Syntax

```
void PC_bsf_MapF(  
    PT_bsf_mapElem_T* mapElem,  
    PT_bsf_reduceElem_T* reduceElem,  
    int* success  
);  
void PC_bsf_MapF_1(  
    PT_bsf_mapElem_T* mapElem,  
    PT_bsf_reduceElem_T_1* reduceElem,  
    int* success  
);  
void PC_bsf_MapF_2(  
    PT_bsf_mapElem_T* mapElem,  
    PT_bsf_reduceElem_T_2* reduceElem,  
    int* success  
);  
void PC_bsf_MapF_3(  
    PT_bsf_mapElem_T* mapElem,  
    PT_bsf_reduceElem_T_3* reduceElem,  
    int* success  
);
```

In parameters

mapElem

The pointer to the structure that is the current element of the map-list.

Out parameters

reduceElem

The pointer to the structure that is the corresponding reduce-list element to be calculated.

**success*

Must be set to *false* if the corresponding reduce-list element must be ignored when the *Reduce* function will be executed. The default value is *true*.

Remarks

The functions *PC_bsf_MapF_1*, *PC_bsf_MapF_2* and *PC_bsf_MapF_3* are used to organize a workflow (optional).

8.2.5 PC_bsf_ParametersOutput

Outputs parameters of the problem before starting the iterative process.

Syntax

```
void PC_bsf_ParametersOutput(  
    PT_bsf_parameter_T parameter  
);
```

In parameters

parameter

The structure containing the parameters of the problem.

8.2.6 PC_bsf_ProblemOutput

Outputs the results of solving the problem.

Syntax

```
void PC_bsf_ProblemOutput(  
    PT_bsf_reduceElem_T* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T parameter,  
    double t  
);  
void PC_bsf_ProblemOutput_1(  
    PT_bsf_reduceElem_T_1* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T parameter,  
    double t  
);  
void PC_bsf_ProblemOutput_2(  
    PT_bsf_reduceElem_T_2* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T parameter,  
    double t  
);  
void PC_bsf_ProblemOutput_3(  
    PT_bsf_reduceElem_T_3* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T parameter,  
    double t  
);
```

In parameters

reduceResult

The pointer to the structure that is the result of executing the higher-order function *Reduce*.

parameter

The structure containing the parameters of the final iteration.

Remarks

The functions PC_bsf_ProblemOutput_1, PC_bsf_ProblemOutput_2 and PC_bsf_ProblemOutput_3 are used to organize a workflow (optional).

8.2.7 PC_bsf_ProcessResults

Processes the results of the current iteration: computes the order parameters for the next iteration, and checks the stop condition.

Syntax

```
void PC_bsf_ProcessResults(  
    PT_bsf_reduceElem_T* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T* parameter,  
    int* newJobCase,  
    bool* exit  
);  
void PC_bsf_ProcessResults_1(  
    PT_bsf_reduceElem_T_1* reduceResult,  
    int reduceCounter,  
    PT_bsf_parameter_T* parameter,  
    int* newJobCase,  
    bool* exit  
);
```

```

void PC_bsf_ProcessResults_2(
    PT_bsf_reduceElem_T_2* reduceResult,
    int reduceCounter,
    PT_bsf_parameter_T* parameter,
    int* newJobCase,
    bool* exit
);
void PC_bsf_ProcessResults_3(
    PT_bsf_reduceElem_T_3* reduceResult,
    int reduceCounter,
    PT_bsf_parameter_T* parameter,
    int* newJobCase,
    bool* exit
);

```

In parameters

reduceResult

The pointer to the structure that is the result of executing the higher-order function *Reduce*.

reduceCounter

The number of summed (by \oplus) elements in the reduce-list. This number matches the number of extended reduce-list elements that have the value 1 in the field *reduceCounter* (see Section 6).

In/out parameters

parameter

The pointer to the structure containing the parameters of the current iteration. This structure must be modified by setting the parameters for the next iteration.

Out parameters

**nextJob*

If a workflow is used (see Section 11), then this variable must be assigned the number of the next action (job). Otherwise, this parameter is not used.

**exit*

If the stop condition holds, then this variable must be assigned *true*. The default value is *false*.

Remarks

The functions *PC_bsf_ProcessResults_1*, *PC_bsf_ProcessResults_2* and *PC_bsf_ProcessResults_3* are used to organize a workflow (optional).

8.2.8 PC_bsf_ReduceF

Implements the operation $z = x \oplus y$ (see Section 2).

Syntax

```

void PC_bsf_ReduceF(
    PT_bsf_reduceElem_T* x,
    PT_bsf_reduceElem_T* y,
    PT_bsf_reduceElem_T* z
);
void PC_bsf_ReduceF_1(
    PT_bsf_reduceElem_T_1* x,
    PT_bsf_reduceElem_T_1* y,
    PT_bsf_reduceElem_T_1* z
);

```

```

void PC_bsf_ReduceF_2(
    PT_bsf_reduceElem_T_2* x,
    PT_bsf_reduceElem_T_2* y,
    PT_bsf_reduceElem_T_2* z
);
void PC_bsf_ReduceF_3(
    PT_bsf_reduceElem_T_3* x,
    PT_bsf_reduceElem_T_3* y,
    PT_bsf_reduceElem_T_3* z
);

```

In parameters

x

The pointer to the structure that presents the first term.

y

The pointer to the structure that presents the second term.

Out parameters

z

The pointer to the structure that presents the result of the operation.

Remarks

The functions *PC_bsf_ReduceF_1*, *PC_bsf_ReduceF_2* and *PC_bsf_ReduceF_3* are used to organize a workflow (optional).

8.2.9 PC_bsf_SetListSize

Sets the length of the list.

Syntax

```

void PC_bsf_SetListSize(
    int* listSize
);

```

Out parameters

**listSize*

Must be assigned a positive integer that specifies the length of the list.

Remarks

The list size should be greater than or equal to the number of workers.

8.2.10 PC_bsf_SetInitParameter

Sets initial order parameters for the workers in the first iteration.

Syntax

```

void PC_bsf_SetInitParameter(
    PT_bsf_parameter_T* parameter
);

```

Out parameters

parameter

The pointer to the structure that the initial parameters should be assigned to.

8.2.11 PC_bsf_SetMapSubList

Fills in the map-sublist that is appointed to the current worker.

Syntax

```
void PC_bsf_SetMapSubList(  
    PT_bsf_mapElem_T* sublist,  
    int sublistLength,  
    int offset  
);
```

In parameters

sublist

The array that represents the map-sublist.

sublistLength

The length of the map-sublist.

offset

A non-negative number that is the index of the first element of the map-sublist.

8.2.12 PC_bsfAssignAddressOffset

Assigns the number of the first element of the map-sublist to the skeleton variables *BSF_sv_addressOffset* (see Section 7).

Syntax

```
void PC_bsfAssignAddressOffset(int value);
```

In parameters

value

Non-negative integer value.

Remarks

Important: The user should not use this function.

8.2.13 PC_bsfAssignIterCounter

Assigns the number of the first element of the map-sublist to the skeleton variables *BSF_sv_iterCounter* (see Section 7).

Syntax

```
void PC_bsfAssignIterCounter(int value);
```

In parameters

value

Non-negative integer value.

Remarks

Important: The user should not use this function.

8.2.14 PC_bsfAssignJobCase

Assigns the number of the current activity (job) in workflow to the skeleton variables *BSF_sv_jobCase* (see Section 7).

Syntax

```
void PC_bsfAssignJobCase(int value);
```

In parameters

value

Non-negative integer value.

Remarks

Important: The user should not use this function.

8.2.15 PC_bsfAssignMpiRank

Assigns the rank of current MPI process to the skeleton variables *BSF_sv_mpiRank* (see Section 7).

Syntax

```
void PC_bsfAssignMpiRank(int value);
```

In parameters

value

Non-negative integer value.

Remarks

Important: The user should not use this function.

8.2.16 PC_bsfAssignNumberInSublist

Assigns the number of the current element in the map-sublist to the skeleton variables *BSF_sv_numberInSublist* (see Section 7).

Syntax

```
void PC_bsfAssignNumberInSublist(int value);
```

In parameters

value

Non-negative integer value.

Remarks

Important: The user should not use this function.

8.2.17 PC_bsfAssignNumOfWorkers

Assigns the total number of the worker processes to the skeleton variables *BSF_sv_numOfWorkers* (see Section 7).

Syntax

```
void PC_bsfAssignNumberInSublist(int value);
```

In parameters

value

Non-negative integer value.

Remarks

Important: The user should not use this function.

8.2.18 PC_bsfAssignParameter

Assigns the order parameters to the structure *BSF_sv_parameter* (see Section 7).

Syntax

```
void PC_bsfAssignParameter(PT_bsf_parameter_T parameter);
```

In parameters

parameter

The structure from which the order parameters are taken.

Remarks

Important: The user should not use this function.

8.2.19 PC_bsfAssignSublistLength

Assigns the length of the current map-sublist to the skeleton variables *BSF_sv_sublistLength* (see Section 7).

Syntax

```
void PC_bsfaAssignSublistLength(int value);
```

In parameters

value

Non-negative integer value.

Remarks

Important: The user should not use this function.

9. Step-by-step instruction

10. Example of using the BSF-skeleton: Jacobi algorithm

In this section, we show how to apply the BSF model to estimate the algorithm scalability boundary using the iterative Jacobi method as an example. The *Jacobi method* [3] is a simple iterative method for solving a system of linear equations. This method was originally described by the German mathematician Carl Gustav Jacob Jacobi in [4]. Let us give a brief description of the Jacobi method.

Let a joint square system of linear equations in a matrix form be given in Euclidean space \mathbb{R}^n :

$$Ax = b, \tag{1}$$

where

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix};$$
$$x = (x_1, \dots, x_n);$$
$$b = (b_1, \dots, b_n).$$

It is assumed that $a_{ii} \neq 0$ for all $i = 1, \dots, n$. Let us define the matrix

$$C = \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix}$$

in the following way:

$$c_{ij} = \begin{cases} -\frac{a_{ij}}{a_{ii}}, \forall j \neq i; \\ 0, \forall j = i. \end{cases}$$

Let us define the vector $d = (d_1, \dots, d_n)$ as follows: $d_i = b_i / a_{ii}$. The Jacobi method of finding an approximate solution of system (1) consists of the following steps:

Step 1. $k := 0$; $x^{(0)} := d$.

Step 2. $x^{(k+1)} := Cx^{(k)} + d$.

Step 3. If $\|x^{(k+1)} - x^{(k)}\|^2 < \varepsilon$, go to Step 5.

Step 4. $k := k + 1$; go to Step 2.

Step 5. Stop.

Here, $\|\cdot\|$ denotes the Euclidian norm. In the Jacobi method, an arbitrary vector $x^{(0)}$ can be taken as the initial approximation. In Step 1, the initial approximation $x^{(0)}$ is assigned by the vector d . In Step 3, the Euclidean norm $\|\cdot\|$ is used in the termination criteria. The *diagonal dominance* of the matrix A is a sufficient condition for the convergence of the Jacobi method:

$$|a_{ii}| \geq \left(\sum_{j=1}^n |a_{ij}| \right) - |a_{ii}|$$

for all $i = 1, \dots, n$, and at least one inequality is strict. In this case, the system (1) has a unique solution for any right-hand side.

Let us represent the Jacobi method in the form of algorithm on lists. Let c_j denotes the j -th column of matrix C :

$$c_j = \begin{pmatrix} c_{1j} \\ \vdots \\ c_{nj} \end{pmatrix}.$$

Let $G = [1, \dots, n]$ be the list of natural numbers from 1 to n . For any vector $x = (x_1, \dots, x_n) \in \mathbb{R}^n$, let us define the function $F_x : 1, \dots, n \rightarrow \mathbb{R}^n$ as follows:

$$F_x(j) = x_j c_j = \begin{pmatrix} x_j c_{1j} \\ \vdots \\ x_j c_{nj} \end{pmatrix}, \quad (2)$$

i.e. the function $F_x(j)$ multiplies the j -th column of the matrix C by the j -th coordinate of the vector x . The BSF-implementation of the Jacobi method shown in Fig. 7 can be easily obtained from the generic BSF-algorithm template shown in Fig. 2. In the algorithm 3, $\vec{+}$ and $\vec{-}$ denote the operations of vector addition and subtraction, respectively. Note that the matrix C entered in line 1 is implicitly used to calculate the values of the function $F_{x^{(k)}}$ in line 3.

Algorithm 3.

```

1:  input  $C, d$ 
2:   $k := 0; x^{(0)} := d; G := [1, \dots, n]$ 
3:   $B := \text{Map}(F_{x^{(k)}}, G)$ 
4:   $s := \text{Reduce}(\vec{+}, B)$ 
5:   $x^{(k+1)} := s \vec{+} d$ 
6:   $k := k + 1$ 
7:  if  $\|x^{(k)} - x^{(k-1)}\|^2 < \varepsilon$  goto 9
8:  goto 3
9:  output  $x^{(i)}$ 
10: stop

```

Fig. 7. BSF-Jacobi algorithm.

11. Workflow support

12. Using OpenMP

13. Applying BSF model to Jacobi method

14. Using Map without Reduce

References

- [1] L.B. Sokolinsky, BSF: a parallel computation model for scalability estimation of iterative numerical algorithms on cluster computing systems, Chelyabinsk, Russia, 2020. <http://arxiv.org/abs/2008.03485>.
- [2] R.S. Bird, Lectures on Constructive Functional Programming, in: M. Broy (Ed.), Constr. Methods Comput. Sci. NATO ASI Ser. F Comput. Syst. Sci. Vol. 55, Springer, Berlin, Heidelberg, 1988: pp. 151–216.
- [3] H. Rutishauser, The Jacobi Method for Real Symmetric Matrices, in: Bauer F.L. (Ed.), Handb. Autom. Comput. Vol 2. Linear Algebr., Springer Berlin Heidelberg, Berlin, Heidelberg, 1971: pp. 202–211. https://doi.org/10.1007/978-3-662-39778-7_12.
- [4] C.G.J. Jacobi, Ueber eine neue Auflösungsart der bei der Methode der kleinsten Quadrate vorkommenden lineären Gleichungen, Astron. Nachrichten. 22 (1845) 297–306. <https://doi.org/10.1002/asna.18450222002>.