SMP Superscalar (SMPSs) User's Manual

Version 2.2

Barcelona Supercomputing Center

September 2008

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Contents

List of Figures

1 Introduction

This document is the user manual of the SMP Superscalar (SMPSs) framework, which is based on a source-to-source compiler and a runtime library. The programming model allows programmers to write sequential applications and the framework is able to exploit the existing concurrency and to use the different cores of a multi-core or SMP by means of an automatic parallelization at execution time. The requirements we place on the programmer are that the application is composed of coarse grain functions (for example, by applying blocking) and that these functions do not have collateral effects (only local variables and parameters are accessed). These functions are identified by annotations (somehow similar to the OpenMP ones), and the runtime will try to parallelize the execution of the annotated functions (also called tasks).

The source-to-source compiler separates the annotated functions from the main code and the library calls the annotated code. However, an annotation before a function does not indicate that this is a parallel region (as it does in OpenMP). To be able to exploit the parallelism, the SMPSs runtime builds a data dependency graph where each node represents an instance of an annotated function and edges between nodes denote data dependencies. From this graph, the runtime is able to schedule for execution independent nodes to different cores at the same time. Techniques imported from the computer architecture area like the data dependency analysis, data renaming and data locality exploitation are applied to increase the performance of the application.

While OpenMP explicitly specifies what is parallel and what is not, with SMPSs what is specified are functions whose invocations could be run in parallel, depending on the data dependencies. The runtime will find the data dependencies and will determine, based on them, which functions can be run in parallel with others and which not. Therefore, SMPSs provides programmers with a more flexible programming model with an adaptive parallelism level depending on the application input data and the number of available cores.

2 Installation

SMP Superscalar is distributed in source code form and must be compiled and installed before using it. The runtime library source code is distributed under the LGPL license v3 and the rest of the code is distributed under the GPL license v3. It can be downloaded from the SMPSs web page at http://www.bsc.es/smpsuperscalar.

2.1 Compilation requirements

The SMPSs compilation process requires the following system components:

- GCC 4.1 or later.
- GNU make
- Gfortran (install it with GCC). Only necessary when SMPSs Fortran support is required.

- PAPI 3.6 only if harware counter tracing is desired

Additionally, if you change the source code you may require:

- automake
- autoconf $\geq$ 2.60
- libtool
- rofi-bison[1]
- GNU flex
- Gperf

2.2 Compilation

To compile and install SMPSs please follow the following steps:

1. Decompress the source tarball.
   ```
   tar -xvzf SMPSs-2.2.tar.gz
   ```
2. Enter into the source directory.
   ```
   cd SMPSs-2.2
   ```
3. Run the configure script, specifying the installation directory as the prefix argument.
   ```
   ./configure --prefix=/opt/SMPSs
   ```
   The configure script also accepts the following optional parameters:
   - `--with-flavour=smp` Specifies that an SMPSs installation is to be performed
   - `--enable-papi` Specifies that tracing should include hardware counters. This option requires a recent installation of PAPI.
   - `--with-papi=prefix` Specifies the PAPI installation path.
   - `--enable-lazy-renaming` Specifies an installation with lazy renaming enabled (data renamings are only performed at task execution time, if necessary).

   More information can be obtained by running `./configure --help`.

   There are also some environment variables that affect the configuration behaviour.
   - SMPSs C compiler may be specified with the `SMPCC` variable.
   - SMPSs C compiler flags may be given with the `SMPCFLAGS` variable.
   - SMPSs C++ compiler may be specified with the `SMPCXX` variable.
   - SMPSs C++ compiler flags may be given with the `SMPCXXFLAGS` variable.

[1]Available at http://www.bsc.es/plantillaH.php?cat_id=351

- Fortran SMPSs compiler may be specified with the `SMPFC` variable.
- Fortran SMPSs compiler flags may be given with `SMPFCFLAGS`.

For example, setting `SMPCC=icc` and `SMPCXX=icc` will use `icc` as backend compiler for SMPSs.

4. Run make.

   ```
   make
   ```

5. Run make install.

   ```
   make install
   ```

2.3 User environment

If SMPSs has not been installed into a system directory, then the user must set the following environment variables:

1. The *PATH* environment variable must contain the bin subdirectory of the installation.

   ```
   export PATH=$PATH:/opt/SMPSs/bin
   ```

2. The *LD_LIBRARY_PATH* environment variable must contain the lib subdirectory from the installation.

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/SMPSs/lib
   ```

3 Programming with SMPSs

SMPSs applications are based on the parallelization at task level of sequential applications. The tasks (functions or subroutines) selected by the programmer will be executed in the different cores. Furthermore, the runtime detects when tasks are data independent between them and is able to schedule the simultaneous execution of several of them on different cores. All the above mentioned actions (data dependency analysis, scheduling and data transfer) are performed transparently to the programmer. However, to take benefit of this automation, the computations to be executed in the cores should be of certain granularity (about $50\mu s$). A limitation on the tasks is that they can only access their parameters and local variables. In case global variables are accessed the compilation will fail.

3.1 C Programming

The C version of SMPSs borrows the syntax from OpenMP in the way that the code is annotated using special preprocessor directives. Therefore, the same general syntax rules apply; that is, directives are one line long but they can span into multiple lines by escaping the line ending.

3.1.1 Task selection

In SMPSs, it is a responsibility of the application programmer to select tasks of certain granularity. For example, blocking is a technique that can be applied to increase the granularity of the tasks in applications that operate on matrices. Below there is a sample code for a block matrix multiplication:

```
void block_addmultiply (double C[BS][BS], double A[BS][BS],
                        double B[BS][BS])
{
    int i, j, k;

    for (i=0; i < BS; i++)
        for (j=0; j < BS; j++)
            for (k=0; k < BS; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

3.1.2 Specifying a task

A task is conceived in the form of a procedure, i.e. a function without return value. Then, a procedure is "converted" into a SMPSs task by providing a simple annotation before its declaration or definition:

```
#pragma css task [input(<input parameters>)]optional \
                 [inout(<inout parameters>)]optional \
                 [output(<output parameters>)]optional \
                 [reduction(<reduction parameters>)]optional \
                 [target (comm_thread)]optional \
                 [highpriority]optional
<function declaration or definition>
```

Where each clause serves the following purposes:

- `input` clause lists parameters whose input value will be read.
- `inout` clause lists parameters that will be read and written by the task.
- `output` clause lists parameters that will be written to.
- `reduction` clause lists parameters that are in a reduction. See more details in subsection 3.1.5.
- `target(comm_thread)` clause indicates that this task is a communication (MPI) task and that has to be executed by a communication thread.
- `highpriority` clause specifies that the task will be scheduled for execution earlier than tasks without this clause.

The parameters listed in the `input`, `inout` and `output` clauses are separated by commas. Only the parameter name and dimension(s) need to be specified, not the type. Although the dimensions must be omitted if present in the parameter declaration.

Examples

In this example, the "factorial" task has a single input parameter "n" and a single output parameter "result".

```
#pragma css task input(n) output(result)
void factorial (unsigned int n, unsigned int *result)
{
    *result = 1;
    for (; n > 1; n--)
        *result = *result * n;
}
```

The next example, has two input vectors "left", of size "leftSize", and "right", of size "rightSize"; and a single output "result" of size "leftSize+rightSize".

```
#pragma css task input(leftSize, rightSize) \
                 input(left[leftSize], right[rightSize]) \
                 output(result[leftSize+rightSize])
void merge (float *left,  unsigned int leftSize,
            float *right, unsigned int rightSize, float *result)
{
    ...
}
```

The next example shows another feature. In this case, with the keyword `highpriority` the user is giving hints to the scheduler: the jacobi tasks will be, when data dependencies allow it, executed before the ones that are not marked as high-priority.

```
#pragma css task input(lefthalo[32], tophalo[32], righthalo[32], \
                       bottomhalo[32]) inout(A[32][32]) highpriority
void jacobi (float *lefthalo,  float *tophalo,
             float *righthalo, float *bottomhalo, float *A)
{
    ...
}
```

In this example, the task `communication` is a task that calls MPI primitives and has to be executed in an specific thread devoted for communications.

```
#pragma css task input(partner, bufsend) output(bufrecv)\
                 target(comm_thread)
void communication(int partner, double bufsend[BLOCK_SIZE],
                 double bufrecv[BLOCK_SIZE])
```

```
{
  int ierr;
  MPI_Request request[2];
  MPI_Status status[2];

  ierr = MPI_Isend(bufsend, BLOCK_SIZE, MPI_DOUBLE, partner,
                   0, MPI_COMM_WORLD, &request[0]);
  ierr = MPI_Irecv(bufrecv, BLOCK_SIZE, MPI_DOUBLE, partner,
                   0, MPI_COMM_WORLD, &request[1]);
  MPI_Waitall(2, request, status);

}
```

3.1.3 Scheduling a task

Once all the tasks have been specified, the next step is to use them. The way to do it is as simple as it gets: just call the annotated function normally. However, there still exists a small requirement, in order to have the tasks scheduled by the SMPSs runtime, the annotated functions must be invoked in a block surrounded by these two directives:

```
#pragma css start
#pragma css finish
```

These two directives can only be used once in a program, i.e. it is not possible to annotate a `start` directive after[2] a `finish` directive has been annotated. They are also mandatory and must enclose all annotated function invocations.

Beware of the fact that the compiler will not detect these issues; the runtime will complain in some cases but may also incur into unexpected behaviour. Section 3.1.4 provides some simple examples.

3.1.4 Waiting on data

When code outside the tasks needs to handle data manipulated also by code inside the tasks, the automatic dependency tracking performed by the runtime is not enough to ensure correct read and write order. To solve this, SMPSs offers some synchronization directives.

As in OpenMP, there is a barrier directive:

```
#pragma css barrier
```

This forces the main thread to wait for the completion of all generated tasks so far. However, this kind of synchronization is too coarse grained and in many cases can be counter productive. To achieve a finer grained control over data readiness, the `wait on` directive is also available:

[2] *After* in the code execution path.

```
#pragma css wait on(<list of variables>)
```

In this case, the main thread waits (or starts running tasks) until all the values of the listed variables have been committed. Like in other clauses[3], multiple variable names are separated by commas.

The data unit to be waited on should be consistent with the data unit of the task. For example, if the task is operating on the full range of an array, we cannot wait on a single element `arr[i]` but on its base address `arr`.

Examples

The next example shows how a `wait on` directive can be used:

```
#pragma css task inout(data[size]) input(size)
void bubblesort (float *data, unsigned int size)
{
    ...
}

void main ()
{
    ...
    #pragma css start

    bubblesort(data, size);

    #pragma css wait on(data)

    for (unsigned int i = 0; i < size; i++)
        printf("%f ", data[i]);

    #pragma css finish
}
```

In this particular case, a barrier could have served for the same purpose since there is just one output variable.

3.1.5 Reductions

The reduction clause is used to specify that several tasks are performing a reduction operation on the parameters specified in the parameters list. The runtime will enable the different tasks performing a reduction to run concurrently (it does not insert the dependence edges that would be inserted due

[3]Clauses that belong to the `task` directive.

to the `inout` parameters), and the user is responsible of accessing the critical data by means of the `mutex` pragmas shown below. These are defined as auxiliary, and it is foreseen to be removed in the future by means of a more clever SMPSs compiler.

`#pragma css mutex lock (<`*`reduction parameters`*`>)`

`#pragma css mutex unlock(<`*`variable`*`>)`

A sample code that uses reductions is shown below. The function `accum` has been annotated with the reduction clause. By defining `sum` as a parameter under reduction, the `inout` dependences between the different instances of `accum` are not considered on the computation of the data dependences.

In this way the data dependences that would exist between each pair of `accum` tasks are eliminated. However, the dependences between all `accum` tasks and `scale_add` tasks are kept, to ensure that the correct value of sum is used.

```
#pragma css task input(A, B) output(C)
void vadd3 (float A[BS], float B[BS], float C[BS]);

#pragma css task input(sum, A) output(B)
void scale_add (float sum, float A[BS], float B[BS]);

#pragma css task input(A) inout(sum) reduction (sum)
void accum (float A[BS], float *sum);


main (){
      ...
      for (i=0; i<N; i+=BS)              // C=A+B
         vadd3 ( &A[i], &B[i], &C[i]);
      ...
      for (i=0; i<N; i+=BS)             // sum(C[i])
         accum (&C[i], &sum);
      ...
      for (i=0; i<N; i+=BS)             // B=sum*A
         scale_add (sum, &E[i], &B[i]);
      ...
      for (i=0; i<N; i+=BS)             // A=C+D
         vadd3 (&C[i], &D[i], &A[i]);
      ...
      for (i=0; i<N; i+=BS)             // E=G+F
         vadd3 (&G[i], &F[i], &E[i]);
      ...
}
```

Additionally, the tasks that are in a reduction must use the `mutex` pragma to ensure that only one of the tasks is accessing the argument resulting of the reductions. A possible implementation of the `accum` task is shown below. Right now the compiler does not perform any modification in the code for the reduction clause. Is only the runtime that is able to consider the data dependences in a different way. For this reason, the programmer is right now responsible for programming the local computation of the reduction and then of accessing the global value protected with the pragma `mutex`.

The task is accumulating the addition of vector A. Each task `accum` receives a chunk of A of BS elements, and variable `sum` accumulates the total value. The local accumulation of the chunck of vector A is perfomed on `local_sol` variable. Then, the value is added to the global value protecting the access with the pragma `mutex`.

```
#pragma css task input(A) inout(sum) reduction (sum)
void accum (float A[BS], float *sum)
{
        int i;
        int local_sol=0;

        for (i = 0; i < BS; i++)
                local_sol += A[i];
#pragma css mutex lock (sum)
                *sum = *sum + local_sol;
#pragma css mutex unlock (sum)
}
```

3.2 Fortran Programming

As in C, the Fortran version of SMPSs also is based on the syntax of OpenMP for Fortran-95. This version of SMPSs only supports free form code and needs some Fortran-95 standard features.

3.2.1 Task selection

In SMPSs it is responsibility of the application programmer to select tasks of a certain granularity. For example, blocking is a technique that can be applied to increase such granularity in applications that operate on matrices. Below there is a sample code for a block matrix multiplication:

```
subroutine block_addmultiply(C, A, B, BS)
   implicit none
   integer, intent(in) :: BS
   real, intent(in) :: A(BS,BS), B(BS,BS)
   real, intent(inout) :: C(BS,BS)
   integer :: i, j, k

   do i=1, BS
     do j=1, BS
       do k=1, BS
         C(i,j) = C(i,j) + A(i,k)*B(k,j)
```

```
      enddo
    enddo
  enddo
end subroutine
```

3.2.2 Specifying a task

A task is conceived in the form of a subroutine. The main difference with C SMPSs annotations it that in Fortran, the language provides the means to specify the direction of the arguments in a procedure. Moreover, while arrays in C can be passed as pointers, Fortran does not encourage that practice. In this sense, annotations in Fortran are simpler than in C. The annotations have the form of a Fortran-95 comment followed by a "`$`" and the framework sentinel keyword (`CSS` in this case). This is very similar to the syntax OpenMP uses in Fortran-95.

In Fortran, each subprogram calling tasks must know the interface for those tasks. For this purpose, the programmer must specify the task interface in the caller subprograms and also write some SMPSs annotations to let the compiler know that there is a task. The following requirements must be satisfied by all Fotran tasks in SMPSs:

- The task interface must specify the parameter directions of all parameters. That is, by using `INTENT (`*`<direction>`*`)`; where *<direction>* is one of: `IN`, `INOUT` or `OUT`.
- Provide an explicit shape for all array parameters in the task (caller subprogram).
- Provide a `!$CSS TASK` annotation for the caller subprogram with the task interface.
- Provide a `!$CSS TASK` annotation for the task subroutine.

The following example shows how a subprogram calling a SMPSs task looks in Fortran. Note that it is not necessary to specify the parameter directions in the task subroutine, they are only necessary in the interface.

```
subroutine example()
  ...
  interface
    !$CSS TASK
    subroutine block_add_multiply(C, A, B, BS)
      ! This is the task interface. Specifies the size and
      ! direction of the parameters.
      implicit none
      integer, intent(in) :: BS
      real, intent(in) :: A(BS,BS), B(BS,BS)
      real, intent(inout) :: C(BS,BS)
    end subroutine
  end interface
  ...
```

```
   !$CSS START
   ...
   call block_add_multiply(C, A, B, BLOCK_SIZE)
   ...
   !$CSS FINISH
   ...
end subroutine

!$CSS TASK
subroutine block_add_multiply(C, A, B, BS)
   ! Here goes the body of the task (the block multiply_add
   ! in this case)
   ...
end subroutine
```

It is also necessary, as the example shows, to call the tasks between `START` and `FINISH` annotation directives. These are executable statements that must be after the declarations, in the executable part of the subprogram. `START` and `FINISH` statements must only be executed once in the application.

Examples

The following example shows part of a SMPSs application using another feature. The `HIGHPRIORITY` clause is used to indicate that one task is high priority and must be executed before non-high priority tasks as soon as its data dependencies allow.

```
interface
   !$CSS TASK HIGHPRIORITY
   subroutine jacobi(lefthalo, tophalo, righthalo, bottomhalo, A)
     real, intent(in), dimension(32) :: lefthalo, tophalo, &
                                        righthalo, bottomhalo
     real, intent(inout) :: A(32,32)
   end subroutine
end interface
```

Next example shows how to annotate a communication task that encapsulates calls to MPI. The `TARGET(COMM_THREAD)` clause is used to indicate this and the runtime will schedule instances of this task in a communication thread.

```
interface
   !$CSS TASK TARGET(COMM_THREAD)
   subroutine checksum(i, u1, d1, d2, d3)
     implicit none
     integer, intent(in) :: i, d1, d2, d3
     double complex, intent(in) :: u1(d1*d2*d3)
   end subroutine
end interface
```

3.2.3 Waiting on data

SMPSs provides two different ways of waiting on data. These features are useful when the user wants to read the results of some tasks in the main thread. Since the main thread does not have the control over task execution, it does not know when a task has finished executing. With `BARRIER` and `WAIT ON`, the main program stops its execution until the data is available.

`BARRIER` is the most conservative option. When the main thread reaches a barrier, waits until all tasks have finished . The syntax is simple:

```
   ...
   do i=1, N
     call task_a(C(i),B)
   enddo
   ...
   !$CSS BARRIER
   print *, C(1)
   ...
```

The other way is to specify exactly which variables we want the program to wait to be available before the execution goes on.

```
!$CSS WAIT ON(<list of variables>)
```

Where the list of variables is a comma separated list of variable names whose values must be correct before continuing the execution.

Example

```
!$CSS TASK
subroutine bubblesort (data, size)
    integer, intent(in) :: size
    real, intent(inout) :: data(size)
    ...
end subroutine

program main
    ...
    interface
      !$CSS TASK
      subroutine bubblesort (data, size)
        integer, intent(in) :: size
        real, intent(inout) :: data(size)
      end subroutine
    end interface
    ...
    call bubblesort(data, size);

    !$CSS WAIT ON(data)

    do i=1, size
      print *, data(i)
    enddo
end
```

3.2.4 Fortran compiler restrictions

This is the first release of SMPSs with a Fortran compiler and it has some limitations. Some will disappear in the future. They consist of compiler specific and non-standard features. Also deprecated forms in the Fortran-95 standard are not supported and are not planned to be included in future releases.

- Case sensitiveness: The SMPSs Fortran compiler is case insensitive. However, task names must be written in lowercase.
- It is not allowed to mix generic interfaces with tasks.
- Internal subprograms cannot be tasks.
- Use of modules within tasks has not been tested in this version.
- Optional and named parameters are not allowed in tasks.
- Some non-standard common extensions like Value parameter passing are not supported or have not been tested yet. In further releases of SMPSs we expect to support a subset of the most common extensions.
- Only explicit shape arrays and scalars are supported as task parameters.
- The `MULTOP` parameter is not supported.
- Tasks cannot have an `ENTRY` statement.
- Array subscripts cannot be used as task parameters.
- `PARAMETER` arrays cannot be used as task parameters.

4 Compiling

The SMPSs compiler infastructure is composed of a C99 source-to-source compiler, a Fortran-95 source-to-source compiler and a common driver.

The driver is called `smpss-cc` and depending on each source filename suffix invokes transparently the C compiler or the Fortran-95 compiler. C files must have the "`.c`" suffix. Fortran files can have either the "`.f`", "`.for`", "`.f77`", "`.f90`", or "`.f95`" suffix.

The `smpss-cc` driver behaves similarly to a native compiler. It can compile individual files one at a time, several ones, link several objects into an executable or perform all operations in a single step.

The compilation process consists in processing the SMPSs pragmas, transforming the code according to those, compiling for the native architeture with the corresponding compiler and packing the object with additional information required for linking.

The linking process consists in unpacking the object files, generating additional code required to join all object files into a single executable, compiling it, and finally linking all objects together with the SMPSs runtime to generate the final executable.

4.1 Usage

The `smpss-cc` compiler has been designed to mimic the options and behaviour of common C compilers. We also provide a means to pass non-standard options to the platform compiler and linker.

The list of supported options is the following:

```
> smpss-cc -help

Usage: cellss-cc <options and sources>

Options:
  -D<macro>           Defines 'macro' with value '1' in the
                      preprocessor.
  -D<macro>=<value>   Defines 'macro' with value 'value' in the
                      preprocessor.
  -g                  Enables debugging.
  -h|--help           Shows usage help.
  -I<directory>       Adds 'directory' the list of preprocessor
                      search paths.
  -k|--keep           Keeps intermediate source and object files.
  -l<library>         Links with the specified library.
  -L<directory>       Adds 'directory' the list of library search
                      paths.
  -O<level>           Enables optimization level 'level'.
  -o <filename>       Sets the name of the output file.
  -c                  Specifies that the code must only be compiled
```

```
                        (and not linked).
  -t|--tracing          Enables run time tracing.
  -v|--verbose          Enables verbose operation.

SMP specific options:
  -Wp,<options>         Passes the comma separated list of options to
                        the C preprocessor.
  -Wc,<options>         Passes the comma separated list of options to
                        the native C compiler.
  -Wf,<options>         Passes the comma separated list of options to
                        the native Fortran compiler.
  -Wl,<options>         Passes the comma separated list of options to
                        the linker.
```

4.2 Building an MPI/SMPSs application

When building MPI applications, usually the `mpicc` wrapper is used. Whith the following commands, the command line that is executed with the wrapper is displayed:

- `mpicc -compile_info` if you use the openmpi library
- `mpicc -link_info` if you use the mpich library
- `mpicc -showme` if you use the openmpi library

For example, you may get:

```
gcc -I/opt/openmpi/include -pthread -L/opt/openmpi/lib -lmpi
```

Then, for compiling and linking mpi_smpss_test.c that is a MPI/SMPSs example:

```
> smpss-cc  -I/opt/openmpi/include -pthread -L/opt/openmpi/lib \
        -lmpi mpi_smpss_test.c -o bin_mpi_smpss_test
```

4.3 Examples

Contrary to previous versions of the compiler, now it is possible to generate binaries from multiple source files like any regular C or Fortran95 compiler. Therefore, it is possible to compile multiple source files directly into a single binary:

```
> smpss-cc -O3 *.c -o my_binary
```

Although handy, you may also use the traditional compilation methodology:

```
> smpss-cc -O3 -c code1.c
> smpss-cc -O3 -c code2.c
> smpss-cc -O3 -c code3.f90
> smpss-cc -O3 code1.o code2.o code3.o -o my_binary
```

This capability allows to easily adapting makefiles by just changing the C compiler, the Fortran compiler and the linker to point to `smpss-cc`. For instance:

```
CC = smpss-cc
LD = smpss-cc
CFLAGS = -O2 -g

SOURCES = code1.c code2.c code3.c
BINARY = my_binary

$(BINARY): $(SOURCES)
```

Combining the `-c` and `-o` options makes possible to generate objects with arbitrary filenames. However, changing the suffix to other than `.o` is not recommended since, in some cases, the compiler driver relies on them to work properly.

As already mentioned, the same binary serves as a Fortran95 compiler:

```
> smpss-cc -O3 matmul.f90 -o matmul
```

If there are no compilation errors, the executable file "matmul" (optimized) is created and can be called from the command line ("`> ./matmul ...`").

In some cases, it is desirable to use specific optimization options not included in the `-O`, `-O1`, `-O2`, or `-O3` set. This is possible by using the `-Wc` flag:

```
> smpss-cc -O2 -Wc,-funroll-loops,-ftree-vectorize \
           -Wc,-ftree-vectorizer-verbose=3 matmul-c -o matmul
```

In the previous example, the native options are passed directly to the native compiler (for example `c99`), to perform automatic vectorization of the code. Note: at the time this manual was written, vectorization seemed not to work properly on `gcc` with `-O3`.

Option `-k`, or `--keep`, will not delete the intermediate files (files generated by the preprocessor, object files, ...).

```
> smpss-cc -k cholesky.c -o cholesky
```

Finally, option `-t` enables executable instrumentation to generate a runtime trace to be analyzed later with the appropriate tool:

```
> smpss-cc -O2 -t matmul.c -o matmul
```

When executing "matmul", a trace file of the execution of the application will be generated. See section 8.1 for further information on trace analysis.

5 Setting the environment and executing

Depending on the path chosen for installation (see section 2), the *LD_LIBRARY_PATH* environment variable may need to be set appropriately or the application will not be able to run.

If SMPSs was configured with `--prefix=/foo/bar/SMPSs`, then *LD_LIBRARY_PATH* should contain the path */foo/bar/SMPSs/lib*. If the framework is installed in a system location such as */usr*, setting the loader path is not necessary.

5.1 Setting the number of CPUs and executing

Before executing a SMPSs application, the number of processors to be used in the execution have to be defined. The default value is 2, but it can be set to a different number with the *CSS_NUM_CPUS* environment variable, for example:

```
> export CSS_NUM_CPUS=6
```

SMPSs applications are started from the command line in the same way as any other application. For example, for the compilation examples of section 4.3, the applications can be started as follow:

```
> ./matmul <pars>
> ./cholesky <pars>
```

6 Programming examples

This section presents a programming example for the block matrix multiplication. The code is not complete, but you can find the complete and working code under *programs/matmul* in the installation directory. More examples are also provided in this directory.

6.1 Matrix multiply

This example presents a SMPSs code for a block matrix multiply. The block contains $BS \times BS$ floats.

```
#pragma css task input(A, B) inout(C)
static void block_addmultiply (float C[BS][BS], float A[BS][BS],
                               float B[BS][BS])
{
    int i, j, k;

    for (i = 0; i < BS; i++)
        for (j = 0; j < BS; j++)
            for (k = 0; k < BS; k++)
                C[i][j] += A[i][k] * B[k][j];
}

int main(int argc, char **argv)
{
    int i, j, k;

    initialize(argc, argv, A, B, C);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                block_addmultiply(C[i][j], A[i][k], B[k][j]);
    ...
}
```

The main code will run in the main thread while the `block_addmultiply` calls will be executed in all the threads. It is important to note that the sequential code (including the annotations) can be compiled with the native compiler, obtaining a sequential binary. This is very useful for debugging the algorithms.

7 SMPSs internals

When compiling a SMPSs application with `smpss-cc`, the resulting object files are linked with the SMPSs runtime library. Then, when the application is started, the SMPSs runtime is automatically invoked. The SMPSs runtime is decoupled in two parts: one runs the main user code and the other runs the tasks.

The most important change in the original user code is that the SMPSs compiler replaces calls to tasks with calls to the `css_addTask` function. At runtime, these calls will be responsible for the intended behavior of the application. At each call to `css_addTask`, the main thread will do the following actions:

- Add node that represents the called task in a task graph.
- Analyze data dependencies of the new task with other previously called tasks.

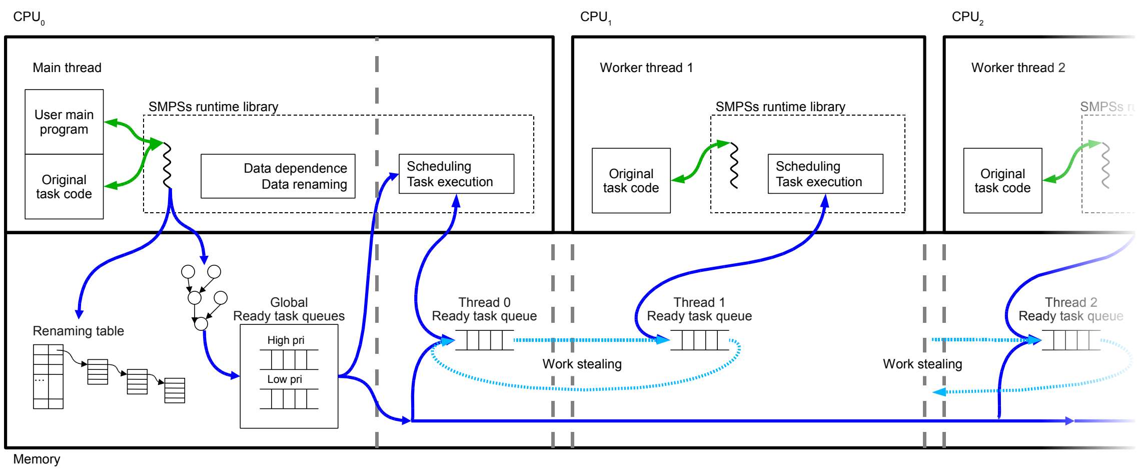

Figure 1: SMPSs runtime behavior

- Parameter renaming: similarly to register renaming, a technique from the superscalar processor area, we do renaming of the output parameters. For every function call that has a parameter that will be written, instead of writing to the original parameter location, a new memory location will be used, that is, a new instance of that parameter will be created and it will replace the original one, becoming a renaming of the original parameter location. This allows to execute that function call independently from any previous function call that would write or read that parameter. This technique allows to effectively remove some data dependencies by using additional storage, and thus improving the chances to extract more parallelism.

Every thread has its own ready task queue, including the main thread. There is also a global queue with priority. Whenever a task that has no predecessors is added to the graph, it is also added to the global ready task queue.

The worker threads consume ready tasks from the queues in the following order of preference:

1. High priority tasks from the global queue.
2. Tasks from its their own queue in LIFO order.
3. Tasks from any other thread queue in FIFO order.

Whenever a thread finishes executing a task, it checks what tasks have become ready and adds them to its own queue. This allows the thread to continue exploring the same area of the task graph unless there is a high priority task or that area has become empty.

In order to preserve temporal locality, threads consume tasks of their own queue in LIFO order, which allows them to reuse output parameters to a certain degree. The task stealing policy tries to minimise adverse effects on the cache by stealing in FIFO order, that is, it tries to steal the coldest tasks of the stolen thread.

The main thread purpose is to populate the graph in order to feed tasks to the worker threads. Nevertheless, it may stop generating new tasks for several conditions: too many tasks in the graph, a wait on, a barrier or and end of program. In those situations it follows the same role as the worker threads by consuming the tasks until the blocking condition is no longer valid.

8 Advanced features

8.1 Using paraver

To understand the behavior and performance of the applications, the user can generate Paraver [2] tracefiles of their SMPSs applications.

If the `-t/-tracing` flag is enabled at compilation time, the application will generate a Paraver tracefile of the execution. The default name for the tracefile is gss-trace-id.prv. The name can be changed by setting the environment variable *CSS_TRACE_FILENAME*. For example, if it is set as follows:

```
> export CSS_TRACE_FILENAME=tracefile
```

After the execution, the files: tracefile-0001.row, tracefile-0001.prv and tracefile-0001.pcf are generated. All these files are required by the Paraver tool.

The traces generated by SMPSs can be visualized and analyzed with Paraver. Paraver [2] is distributed independently of SMPSs.

Several configuration files to visualise and analyse SMPSs tracefiles are provided in the SMPSs distribution in the directory *<install_dir>/share/cellss/paraver_cfgs/*. The following table summarizes what is shown by each configuration file.

Configuration file	**Feature shown**
3dh_duration_phase.cfg	Histogram of duration for each of the runtime phases.
3dh_duration_tasks.cfg	Histogram of duration of tasks. One plane per task (Fixed Value Selector). Left column: 0 microseconds. Right column: 300 us. Darker colour means higher number of instances of that duration.
execution_phases.cfg	Profile of percentage of time spent by each thread (main and workers) at each of the major phases in the runt time library (i.e. generating tasks, scheduling, task execution, ...).
flushing.cfg	Intervals (dark blue) where each thread is flushing its local trace buffer to disk.
general.cfg	Mix of timelines.

Configuration file	Feature shown
task.cfg	Outlined function being executed by each thread.
task_distance_histogram.cfg	Histogram of task distance between dependent tasks.
task_number.cfg	Number (in order of task generation) of task being executed by each thread. Ligth green for the initial tasks in program order, blue for the last tasks in program order. Intermixed green an blue indicate out of order execution.
Task_profile.cfg	Time (microseconds) each thread spent executing the different tasks. Change statistic to: • #burst: number of tasks of each type by thread. • Average burst time: Average duration of each task type.
task_repetitions.cfg	Shows which thread executed each task and the number of times that the task was executed.

8.1.1 Generating traces of MPI/SMPSs applications

If the application we are developing is a MPI/SMPSs, additionally to the use of the -t flag, application traces should be extracted with the support of the MPItrace library.

The decision of which type of tracing to use is taken at runtime, if the MPItrace library is loaded it will use this library to generate the trace, if it is not loaded it will only use the built-in tracing.

The following table summarizes the different options:

		Runtime	
			LD_PRELOAD = libsmssmpitrace.so
Compile time		No tracing	Only MPItrace tracing, no StarSs events
	-t	Only StarSs tracing	MPI+StarSs tracing

If an application has been compiled with the flag `-t`, when starting the execution the runtime will check if the "MPItrace_eventandcounters" function is available. If so the runtime will use the MPI-TRACE tracing. If not it will use the built-in SMPSs tracing.

A consideration is that the pragma `start` must be placed before the `MPI_Init` and the pragma `finish` must be placed before the `MPI_Finalize`. A current limitation of the tracing infrastructure is that the very first events of StarSs may be lost and will not appear in the Paraver tracefile.

In summary, to get a Paraver tracefile of an MPI/SMPSs application, follow these steps:

- Compile the application with the `-t` flag

- Set the the LD_PRELOAD environment variable, being MPItrace_PATH the path where MPI-trace is installed:

  ```
  > export LD_PRELOAD=$MPItrace_PATH/libsmpssmpitrace.so
  ```

- Run the application
- The execution will generate several mpit files, one per MPI process in the execution and one StarSs.pcf file. To generate a Paraver tracefile from them, the `mpi2prv` utility is used:

  ```
  > export MPTRACE_LABELS=StarSs.pcf
  > mpi2prv -f *mpits -syn -o tracefile_name.prv
  ```

8.2 Configuration file

With the objective of tuning the behaviour of the SMPSs runtime, a configuration file where some variables are set is introduced. However, we do not recommend to play with them unless the user considers that it is required to improve the performance of her/his applications. The current set of variables is the following (values between parenthesis denote the default value):

- scheduler.initial_tasks (0): this number specifies how many tasks must be ready before scheduling for the first time.
- task_graph.task_count_high_mark (1000): defines the maximum number of non-executed tasks that the graph will hold.
- task_graph.task_count_low_mark (900): whevever the task graph reaches the number of tasks defined in the previous variable, the task graph generation is suspended until the number of non-executed tasks goes below this amount.
- renaming.memory_high_mark (∞): defines the maximum amount of memory used for renaming in bytes.
- renaming.memory_low_mark (1): whenever the renaming memory usage reaches the size specified in the previous variable, the task graph generation is suspended until the renaming memory usage goes below the number of bytes specified in this variable.
- tracing.papi.hardware_events: Specifies a series of hardware counters that will be measured when tracing is enabled and the runtime has been compiled with PAPI support. The list of available hardware counters can be obtained with the PAPI command "papi_avail -a" and "papi_native_avail -a". The elements in the list can be separated by comas or spaces. Note that not all counter combinations are valid.

This variables are set in a plain text file, with the following syntax:

```
    task_graph.task_count_high_mark = 2000
    task_graph.task_count_low_mark  = 1500
    renaming.memory_high_mark       = 134217728
    renaming.memory_low_mark        = 104857600
    tracing.papi.hardware_events    = PAPI_TOT_CYC,PAPI_TOT_INS
```

The file where the variables are set is indicated by setting the *CSS_CONFIG_FILE* environment variable. For example, if the file "file.cfg" contains the above variable settings, the following command can be used:

```
> export CSS_CONFIG_FILE=file.cfg
```

Some examples of configuration files for the execution of SMPSs applications can be found at location *<install_dir>/share/docs/cellss/examples/*.

References

[1] Barcelona Supercomputing Center. SMP Superscalar website. http://www.bsc.es/smpsuperscalar.

[2] CEPBA/UPC. Paraver website. http://www.bsc.es/paraver.

[3] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A flexible and portable programming model for SMP and multi-cores. Technical report, Barcelona Supercomputing Center – Centro Nacional de Supercomputación, June 2007.