Chapter 7
Signal Processing: Radar

Michel Barreteau and Claudia Cantini

7.1 Brief Description of the RT-STAP Algorithm

Space-time adaptive processing (STAP) is a processing technique operating in the space-time domain that allows the simultaneous cancellation of clutter and jamming via the computation of a 2D cancellation filter.

Essentially, the radar is required to have an array (for instance, a linear array along the aircraft axis) of L antennas each receiving K echoes from a transmitted train of K coherent pulses PRT (pulse repetition time) seconds far apart. The STAP filter operates the simultaneous processing of the spatial samples, i.e., the different channels from different antenna elements, and the temporal samples collected from multiple (consecutive) pulses of the transmitted train. Processing data from multiple channels enables the control of the directional response of the system, while processing data from multiple pulses enables the separation of signals based upon their Doppler frequencies. Unwanted received signals, such as ground clutter and jamming signals, can thereby be efficiently suppressed. The STAP application comprises calculations of adaptive weights (w) and application of these weights on the input data (vector x) (Fig. 7.1). Under the hypothesis of disturbance having a Gaussian probability density function and a target with a certain Doppler frequency and direction of arrival, the output signal of the optimum processor is provided by the linear combination of the LK echoes x (vector representing the physical data cube at a certain range cell under test, CUT) with weights $w = M^{-1}s^*$. M is the noise covariance matrix estimation, i.e., $M = E\{x^*x^T\}$ where x

M. Barreteau
Thales Research & Technology, Campus Polytechnique - 1 avenue Augustin Fresnel, 91767, Palaiseau Cedex, France
e-mail: michel.barreteau@thalesgroup.com

C. Cantini (✉)
Selex ES, Via Tiburtina Km. 12,400, 00131, Roma RM, Italy
e-mail: claudia.cantini@selex-es.com

M. Torquati et al. (eds.), *Smart Multicore Embedded Systems*,
DOI 10.1007/978-1-4614-8800-2_7,

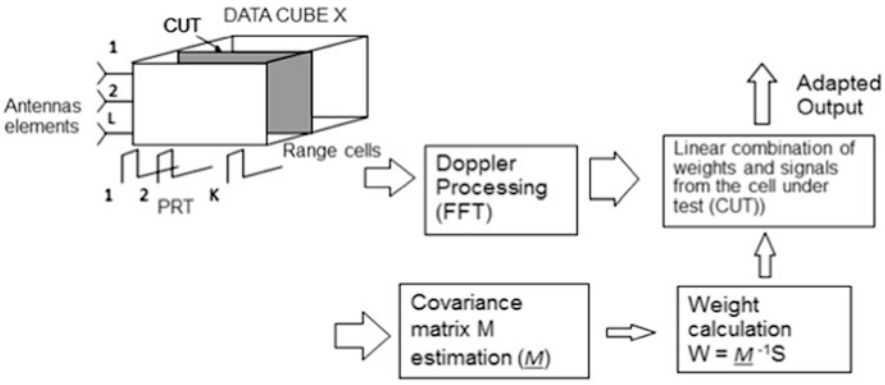

Fig. 7.1 Simplified schema of the real-time STAP radar application

(dimension $LK \times 1$) is the collection of the LK disturbance echoes in a range cell, and s—the space-time steering vector—is the collection of the LK samples expected by the target. The superscripts * and ^{T} stand, respectively, for complex conjugate and transpose. Such a processing technique is highly demanding in terms of computational power requesting the covariance matrix inversion for the weight calculation [1].

From the computational viewpoint, the input data structure is a data cube, i.e., a stream, of thousands of vectors of typically 512 or 1,024 complex single-precision numbers that are used for calculating the covariance matrix estimation. The most critical phase (bottleneck) in the application workflow is the weight calculation through covariance matrix inversion, i.e., the resolution of a very large system of N linear equations on complex numbers, of order $O(N^3)$. This computation is applied to each covariance matrix calculated from each vector belonging to the data cube, according to a streamlike behavior. As performance measures, we are interested mainly in the throughput parameter, i.e., the average number of computed cubes per second. Due to the computational characteristics of this application, this throughput measure is obtained by the evaluation of the service time per matrix. On a current machine, the sequential execution of a system of linear equations on complex numbers, of the sizes indicated above, has a service time per matrix equal to about 10^2–10^4 ms for single-precision numbers. For a true real-time exploitation of STAP, our target service time per matrix is required to be of the order of 10^0–10^1 ms; thus our requirement is a performance improvement of two orders of magnitude to be achieved through parallelization strategies. This goal can be met only if we are able to find almost linear scalable solutions for the parallelization (not a simple task for our target problem due to the large problem size and to the heavy data dependencies in a nested loop computation). With lower priority, we are also interested in the latency per matrix. In some applications we could also accept a latency of the same order of magnitude of the sequential version, while in other cases a sensible latency decrease could be required too. All the other phases of the

applications, though potential and interesting candidates for parallelization (e.g., Doppler processing), have much less stringent performance requirements (they are at least one order of magnitude faster than the weight calculation) and, nowadays, they can be implemented according to very efficient sequential algorithms and libraries (e.g., FFT). However, in massively parallel implementation of the whole application, also these phases could become candidates for parallelization [1].

7.1.1 Detailed Description of the Computational Phases

In the following, for the reasons discussed above, we will consider the weight calculation phase only. For the resolution of the linear system of equations, we use the Cholesky factorization direct method, which applies correctly to all the occurrences of this problem in STAP applications and is characterized by lower complexity compared to other direct methods (QR versions). For our purposes, the Cholesky factorization is considered the application bottleneck. As discussed above, the Cholesky factorization operates on matrices of 512×512 or $1,024 \times 1,024$ single-precision complex numbers organized in streams, where the generic stream element is a vector of 512 or 1,024 single-precision complex numbers.

The classical Cholesky factorization transforms a hermitian positively defined matrix A into the product of a lower triangular matrix L and of its conjugate transpose upper triangular matrix L^T:

$$A = LL^T$$

The basic algorithm applies the method definition directly. It is described by the following algorithmic pseudocode to generate matrix L from matrix A:

```
for (j = 0; j < n; j++) {
    sum = 0;
    for (k = 0; k < j; k++) {
        sum += L_jk^2;
    }
    L_jj = sqrt(A_jj − sum);
    for (i = j + 1; i < n; i++) {
        sum = 0;
        for (k = 0; k < j; k++) {
            sum += L_ik * L_jk;
        }
        L_ij = (A_ij − sum)/L_jj;
    }
}
```

In this nested control structure the size of data structures (notably, parts of columns) varies at every computation step, though according to a statically recognizable and predictable pattern. The literature contains several alternative versions of the basic sequential algorithm for Cholesky factorization. Some block-based versions have been studied in order to optimize the locality and reuse properties of matrix parts accessed during the various computation steps. A is represented as composed of smaller square blocks. This can improve the locality and reuse exploitation: though paid in terms of a larger number of elementary operations on matrix blocks, these properties are the key for potential optimizations of memory hierarchy structures, especially in architectures where caching is not primitive. The matrix representation can be the following:

$$\begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} * \begin{pmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{pmatrix}$$

where

$$A_{11} = L_{11} * L_{11}^T$$

$$A_{21} = L_{21} * L_{11}^T$$

$$A_{21}^T = L_{11} * L_{21}^T$$

$$A_{11} = L_{21} * L_{21}^T + L_{22} * L_{22}^T$$

Denoting $chol$ the application of the Cholesky basic algorithm:

$$L_{11} = chol(A_{11})$$

and iteratively

$$L_{21} = A_{11}/L_{11}^T$$

$$L_{22} * L_{22}^T = A_{22} - L_{21} * L_{21}^T$$

$$A_{22} = L_{22} * L_{22}^T$$

$$L_{22} = chol(A_{22})$$

An algorithmic pseudocode for the block method applied to the Cholesky factorization is (B denotes the number of blocks composing the original matrix):

```
for(k = 0 to B) do {
    L_kk = chol(A_kk);
```

```
    L_kk^T = transpose(L_kk);
    L_kk^-T = invert(L_kk^T);
    for (i = k + 1; i < B; i++) {
        L_ik = A_ik * L_kk^-T
    }
    for (j = k+1; j < B; j++) {
        L_jk^T = transpose(L_jk);
        for (i = j; j < B; i++) {
            A_ij = A_ij - L_ik * L_jk^T
        }
    }
}
```

Here we can statically recognize different patterns for data dependencies with respect to other versions not operating on blocks.
According to the application environment, it is possible that the application operates on cubes that are produced in storage subsystems accessible through I/O and/or files. Owing to the order of magnitudes of the calculation times, the I/O latency for a cube transfer can be overlapped with the internal calculation of previous cubes [2, 3].

7.1.2 Data-Parallel Cholesky Factorization

The block Cholesky algorithm can be expressed using several methods, but two of them are the most used: the left-looking method and the right-looking method. Both methods use the same kernel subroutine to do the numerical work. The differences are mainly in the memory access pattern and in cache data locality exploitation. From the parallelization point of view, the right-looking version expresses more parallelism as the algorithm explores the data dependency graph breadth-first, whereas the left-looking version is less parallel but more cache-oblivious.

In the following we outline in detail the communication pattern of the block-based right-looking version for the Cholesky factorization.

Suppose an $N \times N$ input matrix A. By choosing a block size of m, the input matrix can be split in a set of $\frac{N}{m} \times \frac{N}{m}$ blocks. Exploiting maximum parallelism in the computation of the Cholesky algorithm, each block can be computed by a distinct virtual processor, i.e., a concurrent entity which can be executed on an abstract machine. This results in a set of virtual processors (VPs), which cooperate in getting the factorization done by using explicit messages to resolve data dependencies (for the sake of simplicity we assume a message-passing abstract machine). Without any loss of generality, we also assume that the input matrix is distributed onto the set of VPs row-wise, i.e., the VP_{ij} owns the matrix block data L_{ij} in his local virtual memory.

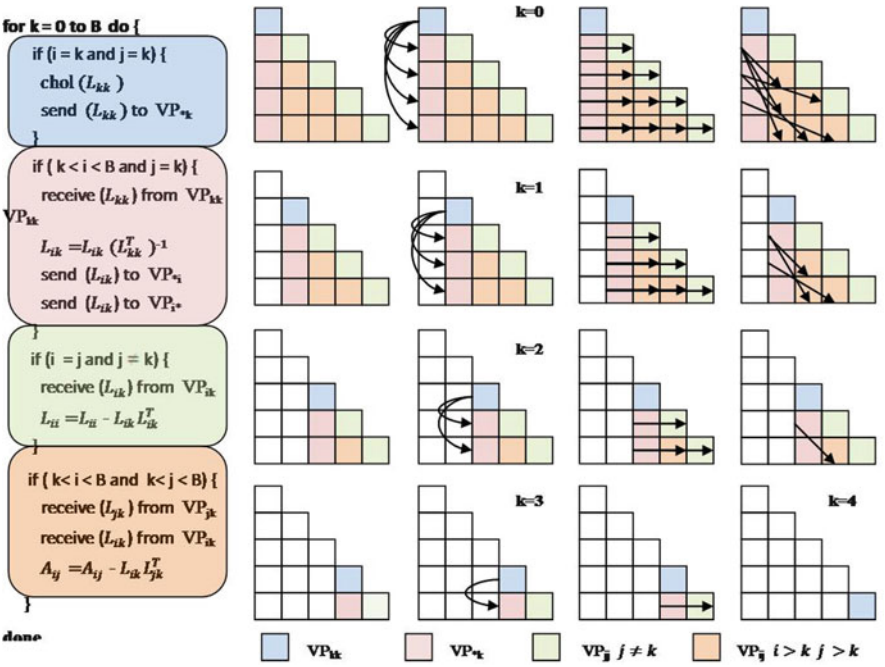

Fig. 7.2 The pseudo code of the generic VP ij (*left*)

The pseudocode of the generic abstract executor VP_{ij}, and the communications among VPs at each steps for the case $N = 5$, is sketched in the following figure ($B = \frac{N}{m}$).

The abbreviated notation VP_{*k} informally stands for "all valid VPs in the column k"; in the same way VP_{k*} stands for "all valid VPs in the row k".

At each step of the main **for** loop, it is possible to identify four sets of distinct VPs, each one with different data dependencies (i.e., activation condition). In Fig. 7.2 we represent different sets with different colors.

At the k-th step, first the VP_{kk} is executed by using only the local block data values, and then the resulting block L_{kk} is sent to all VPs in the k-th column; thus, upon receiving the block, each VP_{*k} can be executed in parallel. The resulting L_{ik} block produced by the VP_{ik} is then sent in parallel to all VPs in the sets VP_{*i} and VP_{i*}, i.e., to all VPs whose row index and column index is equal to the current VP row index i. The VPs in the right-side submatrix of size $[N - (k + 1)]$, upon receiving the blocks, update their local value A_{ij}. It is worth noting that no explicit barrier is needed to synchronize different sets of VPs during external loop iterations.

As the k index approaches B, the number of computing VPs decreases and the communications stencil changes his shape (thus changing the number and the size of the communications). A critical aspect to take into account is the mapping of the

VPs onto the physical processors (generally a subset of the number of VPs) or, in other words, the data distribution of the input matrix [4, 5].

7.2 Related Tool-Chain

The demonstrator leverages a tool-chain which includes:

- A front-end tool: SpearDE by Thales Research & Technology, which offers a graphical interface for describing computation kernels interacting in a data-flow model. It also allows to graphically model the target platform and the mapping of the computation kernels onto this platform.
- A back-end tool: Par4All by SILKAN, which is a source-to-source parallelizing compiler able to provide automatic parallelization of loops.
- A MCAPI layer by CEA which provides a number of low-level tools and runtimes for thread management, memory allocation, and communication.

The tool-chain used for the demonstrator is sketched in Fig. 7.3. It consists in coupling SpearDE [6] (front-end tool), Par4All [7] (back-end tool) and a MCAPI runtime. SpearDE is a graphical model-based design environment which provides the user with both domain-specific application interfaces and heterogeneous execution platform description interface in order to help the implementation of data-streaming applications on parallel machines. Par4All is a source-to-source parallelizing tool generating tasks or parallel programs for various targets from sequential code (C, Fortran, Scilab, Matlab). The MCAPI runtime is built on top of STHORM.

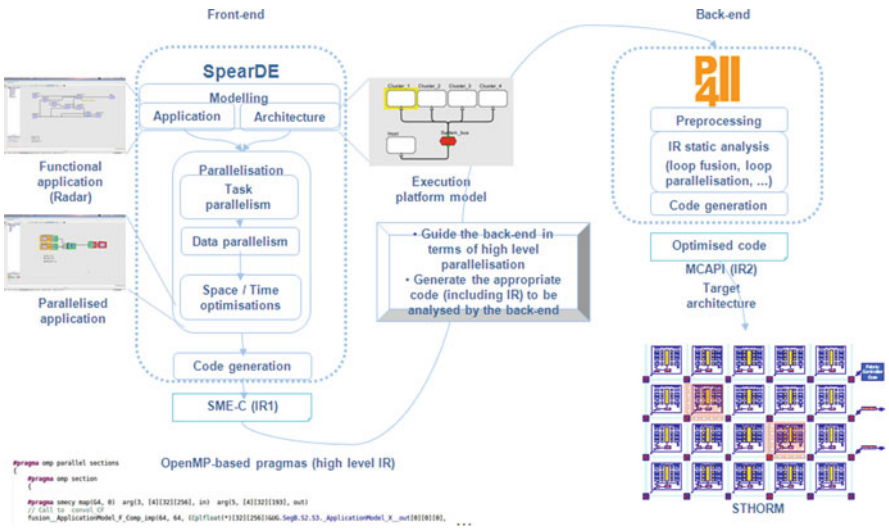

Fig. 7.3 Tool-chain used for the demonstrator

SpearDE allows rapid prototyping on the STHORM platform having the possibility to generate the SMECY IR1 (called SME-C). SpearDE is suited for regular data-streaming applications. It easily handles multidimensional arrays to partition and distribute them onto the STHORM memories. Starting from IR1, the Par4All tool is used to find parallelism in the computational kernels by discovering parallel loop nests through data dependency analysis. This is done by relying on the PIPS open source project [8] which is able to analyze the effects of the program operations by using an abstract interpretation. Parallel loop nests can be replaced by a corresponding kernel call on the low-level platform. The MCAPI layer provides thread management, memory allocation and communication primitives.

The main design steps in this tool-chain are the following ones:

Modeling Both the application (RT-STAP) and the execution platform (STHORM) have to be graphically modeled. The real-time STAP application will be built from scratch here but SpearDE also accepts C99-based code with a set of coding rules.

Parallelization First it consists in allocating computing tasks to hardware resources. Then data parallelism is pointed out. Finally communication tasks will be inserted where needed and scheduling may be refined.

Code generation All the previous parallelization steps are exploited to generate the appropriate IR1 code. This guides Par4All to generate an efficient IR2 code through several passes.

Execution The IR2 primitives rely on the MCAPI layer to ensure an efficient execution on STHORM.

7.2.1 Application and Execution Platform Modeling

The application model of the demonstrator is shown in Fig. 7.4. The block Cholesky algorithm (green dashed lines around the *Chol* module) has been decomposed into several nodes. Each node in the graph matches a computing task (one step of the Cholesky algorithm) that includes a (parallel) loop nest; it executes a so-called elementary task that is usually iterated by several static affine nested loops. The elementary task represents a basic operation (e.g., matrix multiply or inversion, convolution) or a function already optimized or inherently sequential. The *Cov* module (in blue) and the *Wei* module (in orange) were not decomposed in multiple elementary nodes. The first and last tasks are artificial tasks that, respectively, generate inputs and test results.

The SpearDE model of the STHORM platform is shown in Fig. 7.5. The platform model gives a hierarchical representation of the target system which is not the exact representation of the physical platform (the 16 cores of each cluster are not detailed because the generated code will be executed at cluster level but the

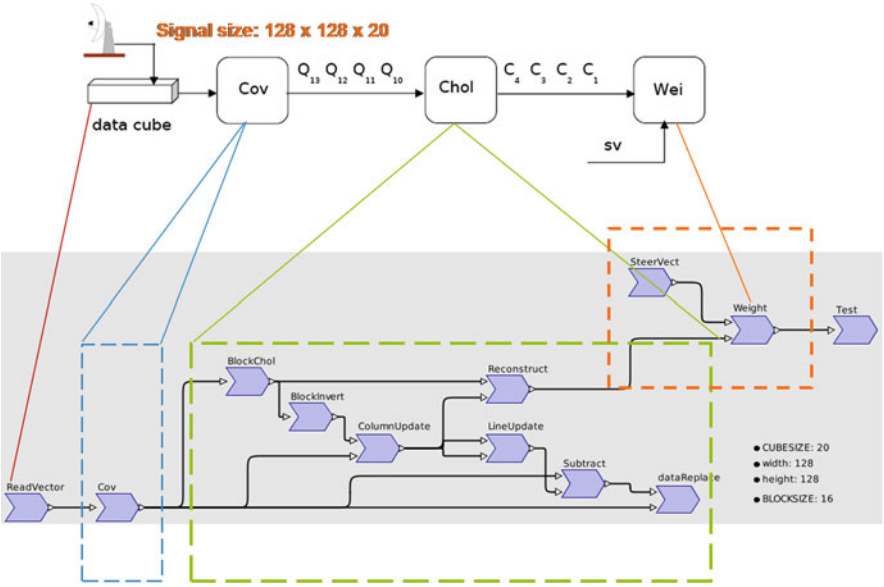

Fig. 7.4 SpearDE application model

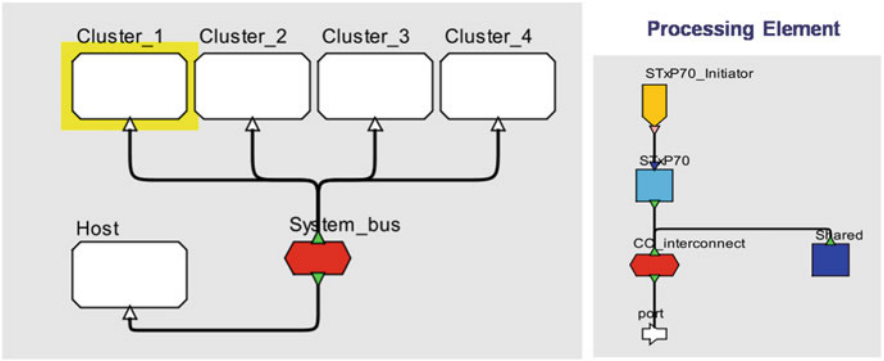

Fig. 7.5 SpearDE model of the STHORM platform

number of cores is known under the form of an attribute) but is detailed enough to allow automatic generation of communication nodes in case of distributed memory data accesses or when data reorganization is needed between two existing nodes. SpearDE relies on the described topology to compute communications between the different memories.

Due to STHORM cluster memory constraints and considering the proposed parallelization in SpearDE, the maximum input size for the demonstrator input data cube is $128 \times 128 \times 20$ complex float numbers (total size of 2.5 MB). The block size

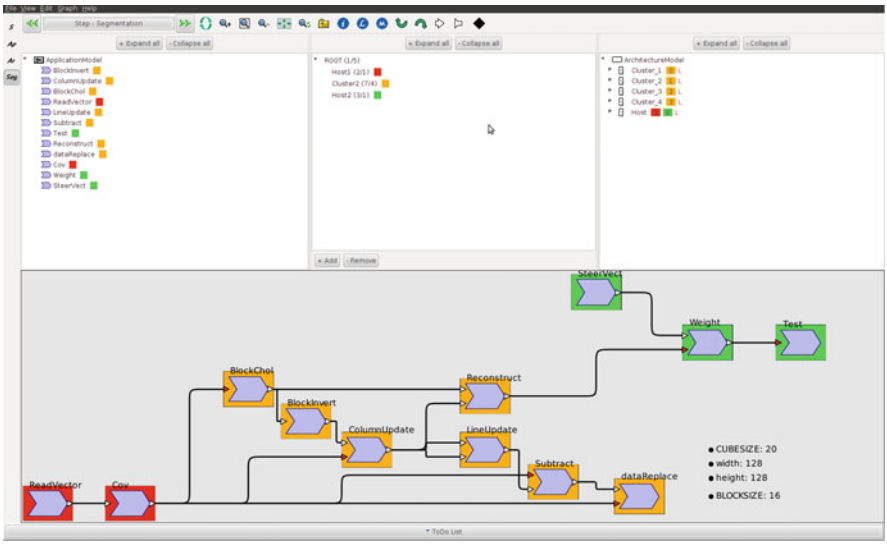

Fig. 7.6 Mapping of different application modules in SpearDE

is set to 16×16 elements in order to have a good balance between parallelism and computation granularity.

7.2.2 Parallelisation on the STHORM Platform

Considering the STHORM platform which includes four clusters, each one with 16 processing elements and 256 KB of local shared memory, the following mapping of modules has been applied:

- The *Cov* module is mapped on the host processor (the red coloured modules at the bottom of Fig. 7.6).
- All modules composing the block Cholesky factorization (*Chol*) are replicated on each cluster of the STHORM platform in order to be able to compute four matrices at a time (all orange-colored modules).
- The *Wei* module is mapped on the host processor working in pipeline with the 4 *Chol* modules (the green-colored modules).

This allocation is quickly done thanks to a graphical interface (through drag and drops). At the top of Fig. 7.6, computing tasks (flattened view) are listed on the left-hand side and hardware resources on the right-hand side. An association (depicted in the center by a color) between a subset of computing tasks and a part of available computing resources means that these tasks will be executed by the selected hardware resources (same color).

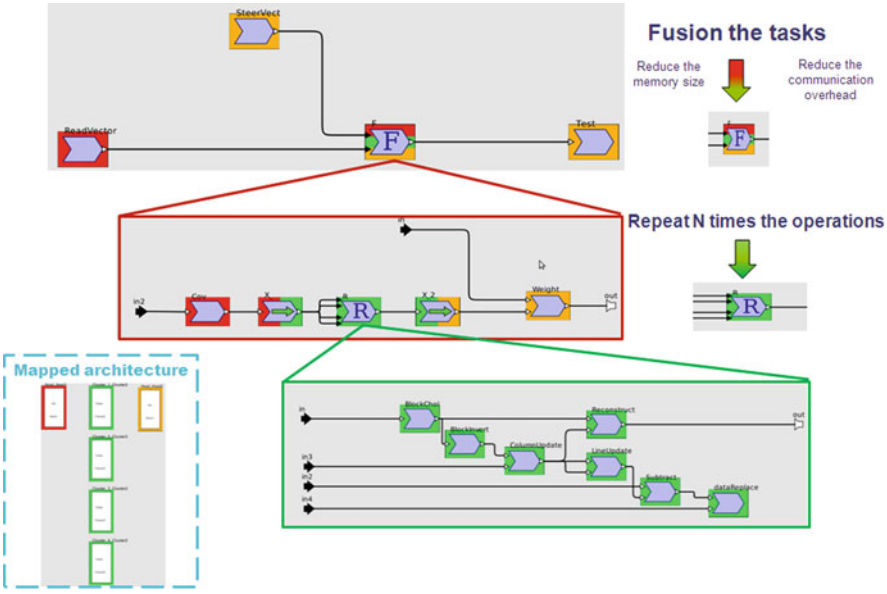

Fig. 7.7 Hierarchical view of the parallelized application under SpearDE

Once this task parallelism decided, the user has to repeat the *Chol* tasks 16 times to cover the whole matrix (through a push-button mechanism again). The next step consists in operating a task fusion around these tasks (including the communication tasks) to implement a round-robin distribution. This task fusion optimizes the memory occupancy (depending on the life duration of arrays) that fits the STHORM memory sizes. Communications between the Host and the STHORM are automatically inserted by pressing a button. This is done in order to send input data to the accelerator and to receive results back from STHORM.

The resulting parallelized application is shown in Fig. 7.7.

7.2.3 IR Code Generation

7.2.3.1 SpearDE/IR1 and Par4All/IR2

SpearDE translates the results of these parallelization steps into an IR1 code. The related Fig. 7.8 shows:

- Parallel loops (`#pragma omp parallel`) with other OpenMP directives (e.g., `schedule(static,1)` means 1 thread per processor with a static scheduling)
- Some SMECY-specific mapping directives (`#pragma smecy map`)

- The round-robin distribution (`num_threads(4) schedule(static,1)`)
- Some communication primitives (`#pragma smecy communication`) that make this IR1 executable.

All these informations enable to guide Par4All in producing an efficient IR2 code. As Par4All is a source-to-source compiler, it refines this SME-C code through several passes: it uses SMECY-specific macros (see below `SMECY_*`)

Listing 7.1 SMECY-specific macros generated by Par4All

```
void smecy_func_fusion_F_F2_fft_CF_6() {
 SMECY_set(fusion_F_F2_fft_CF,6,STHORM,1,0);
 SMECY_send_arg_vector(fusion_F_F2_fft_CF,1,Cplfloat,((Cplfloat (*)[32UL])(UG -> SegClusters.S2.S3.
 F_X_4_out[0]
 + 0)),193 * 32,STHORM,1,0);
 SMECY_prepare_get_arg_vector(fusion_F_F2_fft_CF,2,Cplfloat,((Cplfloat (*)[32UL])(UG ->
        SegClusters.S2.S3.F_Filt_Dop_out[0] + 0)),193 * 32,STHORM,1,0);
 SMECY_launch(fusion_F_F2_fft_CF,2,STHORM,1,0);
 SMECY_get_arg_vector(fusion_F_F2_fft_CF,2,Cplfloat,((Cplfloat (*)[32UL])(UG -> SegClusters.S2.S3.
  F_Filt_Dop_out[0] + 0)),193 * 32,STHORM,1,0);
 SMECY_cleanup_send_arg_vector(fusion_F_F2_fft_CF,1,Cplfloat,((Cplfloat (*)[32UL])(UG -> SegClusters.S2.S3.
  F_X_4_out[0] + 0)),193 * 32,STHORM,1,0);
 SMECY_accelerator_end(fusion_F_F2_fft_CF,6,STHORM,1,0);
 // Call to fusion_F_F2_fft_CF }
```

that are translated into SMECY MCAPI primitives (see `SMECY_MCAPI_*`).

```
// Call to Vect2Cov
#pragma smecy map(Host)  arg(3, [128][400], in)  arg(5, [21][128][128], out)
Vect2Cov(128, 400, (Cplfloat(*)[400])&UG->SegHost.S1.ReadVector_out[0][0], 21, (Cplfloat(*)[128][128])&UG->SegHost.S1.Vect2Cov_out[0][0][0], 200, 10);

int i_F_0;
#pragma omp parallel for private (i_F_0,idxTime) num_threads(4) schedule(static,1)
for (i_F_0 = 0; i_F_0 < 21; i_F_0++) {

    threadprivate_i_F_0 = i_F_0;
    #pragma omp critical
    {
    // Call to BlockCov
    #pragma smecy map(Host)  arg(4, [128][128], in)  arg(5, [16][16][8][8], out)
    BlockCov(128, 8, 16, (Cplfloat(*)[128])&UG->SegHost.S1.Vect2Cov_out[0 + i_F_0*1][0][0], (Cplfloat(*)[16][8][8])&UG->SegHost.S1.S2.F_BlockCov_out[0][0][0][0]);

    // Communication from x86.External_DDR to Cluster.Shared
    #pragma smecy communication src(External_DDR,Host) dst(Shared,STHORM,(threadprivate_i_F_0)%4,0)
    memcpy(&UG->SegClusters[(threadprivate_i_F_0)%4].S3.F_X_2_out[0][0][0][0], &UG->SegHost.S1.S2.F_BlockCov_out[0][0][0][0], 16*16*8*8*sizeof(Cplfloat));

    }
    int i_F_R_0;
    idxTime = 0;
    for (i_F_R_0 = 0; i_F_R_0 < 16; i_F_R_0++) {

        // Call to ComputeCholesky
        #pragma smecy map(STHORM,(threadprivate_i_F_0)%4,0)  arg(2, [8][8], in)  arg(3, [8][8], out)
        ComputeCholesky(8, (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.F_X_2_out[0][0][0][0], (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockChol_out[0][0]);

        // Call to Invert
        #pragma smecy map(STHORM,(threadprivate_i_F_0)%4,0)  arg(2, [8][8], in)  arg(3, [8][8], out)
        Invert(8, (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockChol_out[0][0], (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockInvert_out[0][0]);

        // Call to fusion_F_R_F_ColumnUpdate
        fusion_F_R_F_ColumnUpdate(8, (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockInvert_out[0][0], (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4].S3.F_X_2_

        // Call to fusion_F_R_F3_LineUpdate
        fusion_F_R_F3_LineUpdate(8, (Cplfloat(*)[8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_ColumnUpdate_out[0][0][0], (Cplfloat(*)[8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F

        // Call to Reconstruct2
        #pragma smecy map(STHORM,(threadprivate_i_F_0)%4,0)  arg(3, [8][8], in)  arg(4, [16][8][8], in)  arg(5, [128][128], out)
        Reconstruct2(8, 128, (Cplfloat(*)[8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_BlockChol_out[0][0], (Cplfloat(*)[8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_ColumnUpdate_

        // Call to fusion_F_R_F4_Subtract
        fusion_F_R_F4_Subtract(8, (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4].S3.F_X_2_out[0][0][0][0], (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_Li

        // Call to fusion_F_R_F2_DataReplace
        fusion_F_R_F2_DataReplace(8, (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4].S3.S4.F_R_Subtract_out3[1][1][0][0], (Cplfloat(*)[16][8][8])&UG->SegClusters[(i_F_0)%4


        idxTime++;
    }
    #pragma omp critical
    {
    // Communication from Cluster.Shared to x86.External_DDR
    #pragma smecy communication src(Shared,STHORM,(threadprivate_i_F_0)%4,0) dst(External_DDR,Host)
    memcpy(&UG->SegHost.S1.S2.F_X__out[0][0], &UG->SegClusters[(threadprivate_i_F_0)%4].S3.F_Reconstruct_out3[0][0], 128*128*sizeof(Cplfloat));

    // Call to ApplyWeights
    #pragma smecy map(Host)  arg(4, [128], in)  arg(5, [128][128], in)  arg(6, [128], out)
    ApplyWeights(128, 128, 128, (Cplfloat(*))&UG->SegHost.S1.SteerVect_out[0], (Cplfloat(*)[128])&UG->SegHost.S1.S2.F_X__out[0][0], (Cplfloat(*))&UG->SegHost.S1.Weights_out[0

    }

}
// Call to TestWeights
#pragma smecy map(Host)  arg(3, [21][128], in)
TestWeights(21, 128, (Cplfloat(*)[128])&UG->SegHost.S1.Weights_out[0][0], "./ref/weights_128x400x200_10_ref.txt");
```

Fig. 7.8 SpearDE IR1 piece of code

Listing 7.2 SMECY MCAPI primitives generated by Par4All

```
void smecy_func_fusion_F_F2_fft_CF_6 (void)   { {
 mcapi_status_t SMECY_MCAPI_status;
 size_t P4A_received_size;
 Cplfloat *p4a_STHORM_1_0_fusion_F_F2_fft_CF_1_msg;
 mcapi_pktchan_recv (P4A_receive, (void **)&p4a_STHORM_1_0_fusion_F_F2_fft_CF_1_msg,
       &P4A_received_size, &SMECY_MCAPI_status);
 SMECY_MCAPI_check_status (SMECY_MCAPI_status,"accel_smecy_gencode.c", __func__, 345);
 Cplfloat *p4a_STHORM_1_0_fusion_F_F2_fft_CF_1 = p4a_STHORM_1_0_fusion_F_F2_fft_CF_1_msg;
 Cplfloat p4a_STHORM_1_0_fusion_F_F2_fft_CF_2[193 * 32];
 fusion_F_F2_fft_CF (p4a_STHORM_1_0_fusion_F_F2_fft_CF_1, p4a_STHORM_1_0_fusion_F_F2_fft_CF_2);
 mcapi_pktchan_send (P4A_transmit, p4a_STHORM_1_0_fusion_F_F2_fft_CF_2, 193 * 32 * sizeof (Cplfloat)
       , &SMECY_MCAPI_status);
 SMECY_MCAPI_check_status (SMECY_MCAPI_status,"accel_smecy_gencode.c", __func__, 348);
 mcapi_pktchan_release (p4a_STHORM_1_0_fusion_F_F2_fft_CF_1_msg, &SMECY_MCAPI_status);
 SMECY_MCAPI_check_status (SMECY_MCAPI_status,"accel_smecy_gencode.c", __func__, 349); } }
```

The final MCAPI primitives then run on the GEPOP Posix simulator in this case.

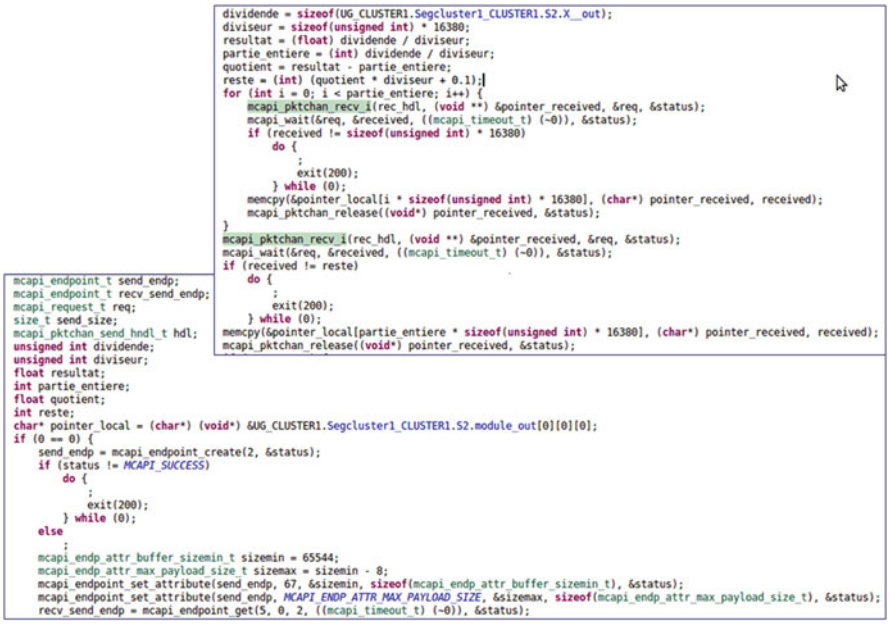

Fig. 7.9 SpearDE IR2 piece of code

7.2.3.2 SpearDE/IR2

Note that SpearDE is also able to generate the MCAPI IR2 from the same (graphical) parallelized application as seen in Fig. 7.7. Moreover SpearDE automatically manages the different communication protocols between communication ports of the host and the clusters (for instance no more than 16 ports per node are allowed by the MCAPI implementation). SpearDE also allows computation / communication overlapping (for latency optimization purpose).

Figure 7.9 shows the kind of MCAPI code that has been validated under the GEPOP Posix simulator.

7.3 Conclusion

This semiautomatic approach (through SME-C) brings some significant advantages:

- The user masters the parallelisation at high level.
- The parallelized application can be functionally tested at early stage since SME-C is executable (SME-C relies on pragmas that are close to standards like OpenMP).
- The user can rely on the design space exploration facilities for rapid prototyping purpose.
- A lot of parallelization information eases the back-end tool role (e.g., no need to analyze loops that are already declared as parallel).

Hence SME-C (generated by the front-end tool) provides the back-end tool with several hints that guide its work to generate an efficient low-level code on the target.

Acknowledgements This work also relies on the following Embedded Systems Lab's members: Teodora Petrisor (application modeling), Remi Barrere (tool enhancements, IR1 code generation), Paul Brelet (IR2 code generation) and Eric Lenormand (mapping). Claudia Cantini wishes to thank Prof. Marco Vanneschi and the Parallel Computing Laboratory of the Computer Science Department of the University of Pisa.

References

1. K. Cain, C. Torres, and R. Williams, "Rt-stap: Real-time space-time adaptive processing benchmark," 1997.
2. Wikipedia, "Cholesky decomposition," http://en.wikipedia.org/wiki/Choleskydecomposition, 2012. [Online]. Available: http://en.wikipedia.org/wiki/Choleskydecomposition
3. J. Choi, J. Dongarra, L. Ostrouchov, A. Petitet, D. Walker, and R. Whaley, "The design and implementation of the scalapack lu, qr and cholesky factorization routines," 1994.
4. E. Rothberg and R. Schreiber, "Improved load distribution in parallel sparse cholesky factorization," in *In Proceedings of Supercomputing '94*, 1994, pp. 783–792.
5. E. Rothberg and A. Gupta, "An efficient block-oriented approach to parallel sparse cholesky factorization," pp. 1413–1439, 1994.
6. E. Lenormand and G. Edelin, "An industrial perspective: a pragmatic high-end signal processing design environment at thales," in *In proceedings of the Workshop on Systems, Architectures, Modeling and Simulation SAMOS*, 2003, pp. 52–57.
7. SILKAN, http://www.par4all.org/, 2012. [Online]. Available: http://www.par4all.org/
8. MINES-ParisTech, "PIPS," http://pips4u.org, 1989–2009, open source, under GPLv3.