EXPERIENCES WITH ASPECT-BASED PARALLELIZATION OF SCIENTIFIC CODE

Manuel Díaz, Sergio Romero, Bartolomé Rubio, Enrique Soler and José M. Troya
Department of Languages and Computer Science
University of Málaga
29071, Málaga, SPAIN
E-mail: mdr@lcc.uma.es

KEYWORDS
Languages, Object-Oriented Programming & Design.

ABSTRACT

This paper discusses the use of Aspect-Oriented Programming (AOP) to support the parallelization of scientific code. The idea is to develop parallelism concerns in separate aspects so that the weaving process can inject the code structures which allow the sequential scientific core to be executed in parallel. A series of advantages can initially be derived from this aspect-based approach. As parallelism is modularized into separate units, the code-tangling level is reduced. The parallel version is developed by reusing the code pieces which implement the numerical computation sequentially. Moreover, different aspects can be written to adapt the application to different high-performance environments. This paper describes some experiences with the code parallelization using aspects. Specifically, the integration of both task and data parallelism in the context of two parallel scenarios (i.e. multithreading and message-passing) is addressed in a case study. The work is an attempt to assess the benefits and limitations of applying AOP for these purposes.

INTRODUCTION

Scientific software frequently exploits parallelism for achieving high-performance when tackling large-scale and realistic engineering problems. Implementations are typically based on parallel programming libraries (e.g. PThreads, MPI, PVM) which provide the developer with a set of primitives to code parallelism concerns. However, code-tangling problems often arise in these applications as the statements describing the numerical computation are mixed with those expressing parallelism.

Aspect-Oriented Programming (Kiczales et al. 1997) helps the developer to achieve the separation of concerns, especially in those situations when such concerns cut across multiple parts (clases, components, modules) of a system. The modular management of cross-cutting concerns leads to a simpler application code which is easier to develop and maintain and has a greater potential for reuse. A well-modularized cross-cutting concern is called an aspect.

The management of high-performance concerns using AOP is emerging as a promising line of research. In (Harbulot and Gurd 2004), the separation of the parallel structure cross-cutting Java scientific applications is addressed using the general-purpose language AspectJ (Kiczales et al. 2001). The work in (Harbulot and Gurd 2006) describes an extension of the AspectJ join point model aimed at allowing loops to be parallelized without re-factoring the base-code. A methodology for the modular development of parallel programs which is based on the composition of multiple fine-grain aspects such as concurrency, partition, distribution and so on, is discussed in (Sobral 2006). The proposal in (Díaz et al. 2005) is focused on a component framework for the efficient development of high-performance applications. Specific concerns which affect a set of scientific components, such as the communication scheme underlying the application, are modeled into a special type of entities called aspect components.

This paper focuses on the idea that scientific programs can (possibly) be written sequentially in a way that enables parallelism to be added later, for instance, once the code has been tested and debugged. Although the parallelization of a scientific program may require many changes of diverse nature in different parts of its code, we can consider the effects of these changes as the result of weaving what we have called parallelization aspects into the sequential scientific core (i.e. the part in charge of the numerical computation). The bodies of these aspects will implement additional functionalities such as distribution, communication and synchronization, in order to enable the sequential core to be executed in parallel.

We can anticipate a series of advantages derived from this aspect-based parallelization approach:

- Parallel statements don't obscure the mathematical model, as the former are isolated into aspects, which results in a reduction of code-tangling.
- The parallel application is set-up by weaving aspects into the sequential core, promoting core reuse.
- The applied parallelism model can be replaced simply selecting different aspects to be woven.

Proceedings 21st European Conference on Modelling and Simulation
Ivan Zelinka, Zuzana Oplatková, Alessandra Orsoni ©ECMS 2007
ISBN 978-0-9553018-2-7 / ISBN 978-0-9553018-3-4 (CD)

Despite the potential benefits, this approach may also lead to some serious shortcomings. The characteristics of the aspect-oriented language used determine the type of interaction between aspects and base-code. For instance, in a general-purpose language such as AspectJ, the interactions are mainly based on intercepting method calls (pointcuts plus advice code), augmenting data structures (introductions), and so on. However, the parallelization of code following these mechanisms can be very difficult unless the programs are written assuming (explicitly) the fact that the code pieces which will be parallelized must be accessed and managed through valid join points. This leads us to consider new sequential designs which include sets of interfaces and/or classes aimed at allowing aspects to inject parallelism. In some sense, the parallelization concern must be somehow considered from the initial design of the code. Obviously, this represents an extra effort in the development of the sequential scientific core.

This work is an attempt to assess the feasibility and suitability of the approach. The purpose is to describe a real experience which allows us to determine the overall advantages and limitations of using aspects for the code parallelization. At first, we describe a sequential program which computes the 2D-Fast Fourier Transform (Briham 1988) of a collection of matrices. Then, aspect weaving is used to integrate both task and data parallelism into the scientific code. This is carried out in the context of two different parallel scenarios: multithreading with shared memory, and message-passing based on MPI.

The implementation is based on the aspect-oriented language AspectC++ (Spinczyk et al. 2002). In AspectC++, almost all the language elements are efficiently implemented at compile-time, which makes the tool more suitable for the development of high-performance concerns than other approaches using Java-based dynamic weaving such as AspectJ.

The paper is structured as follows. Section 2 provides an overview of AspectC++. The case study is presented in section 3, where the sequential solution is described. Sections 4 and 5 provide an in-depth description of the parallelization using multithread programming and MPI, respectively. The paper finishes with some conclusions.

ASPECTC++ FUNDAMENTALS

AspectC++ is a general purpose language extension to C++ for the support of Aspect-Oriented Programming. An aspect can be understood as a modularized unit that implements a cross-cutting concern. The points at which an aspect can interfere with the base-code are called join points.

A pointcut identifies a set of join points. Pointcuts are described by means of pointcut expressions, which can refer to combinations of static program entities, such as classes, functions or namespaces, and other points in the control flow of the program. For instance, the expression:

```
execution("void Dialog::set%(...)")
```

refers to the execution of any method of `Dialog` having both a name beginning with `set` and `void` as return type. In the match expression, `%` is used as a wildcard symbol and `...` represents any sequence of arguments. A pointcut declaration allows a pointcut to be named so that it can be reused in different parts of the program.

The advice is the mechanism which defines the way the aspect affects the base-code. An advice declaration indicates the block of code to be executed when specific join points are reached. The advice code can be executed before, after, or both before and after (i.e. around) the join point. For instance, the following advice can be used to trace the execution of "setter" methods in `Dialog`:

```
advice execution("void Dialog::set%(...)") && that(d)
  : before(Dialog &d) {
  cout << "Dialog:" << d.name
       << " will be modified." << endl;
}
```

The code in the example above is triggered just before the method execution. The function `that()` binds a variable to the object on which the method is invoked (i.e. the object referred to by `this`). Other pointcut functions can be used to access information such as the arguments of a function and its return value. Furthermore, the object `tjp` (of class `JoinPoint`) allows the programmer to retrieve context information from within the advice code.

A different type of advice is that represented by introductions, which are used to augment data structures. For instance, the following code uses a slice element to add data members and methods to the class `Dialog`. A method which is defined this way can access even private data members:

```
advice "Dialog" : slice class {
  Time creation;
  int isExpired() {
    return ((Time::now() - creation) > 3600) ? 1:0;
  }
};
```

The aspect is the language construction in which all these elements are combined for the implementation of modularized cross-cutting concerns. In terms of syntaxis, an aspect is very similar to a C++ class definition. In this sense, aspects can have data members and methods, and can inherit from classes and even other aspects. The code below shows an aspect which implements an "expiration" concern for the class `Dialog`. The functionality consists of preventing the modification of any dialog window which has already

expired. In an around advice, the original join point code can be executed by calling `tjp->proceed()`:

```
aspect Expiration {
  pointcut resetTime() = construction("Dialog") ||
    execution("void Dialog::show(...)");

  advice resetTime() && that(d) : after(Dialog &d) {
    d.creation = Time::now();
  }
  advice execution("void Dialog::set%(...)")&&that(d)
    : around(Dialog &d) {
    if ( !d.isExpired() )
      tjp->proceed();
  }
};
```

CASE STUDY: THE SEQUENTIAL 2D-FFT

The Fast Fourier Transform (FFT) is used to produce frequency analysis of discrete signals in a wide range of application domains: image analysis, signal processing, speech recognition, astronomy, etc. The processing of a vector of N elements (i.e. complex numbers) involves O(N logN) operations. The FFT of a matrix (called 2D-FFT) consists of using FFT to transform each column, and then uses the result to transform each row (again using FFT).

This section describes a C++ code for the sequential processing of a stream of complex matrices using the 2D-FFT. The problem has been broken down into four main activities (sensor, FFT on columns, FFT on rows, writer) which are coupled following a pipeline scheme, so that each activity consumes the result of the previous one. The class diagram is shown in figure 1.

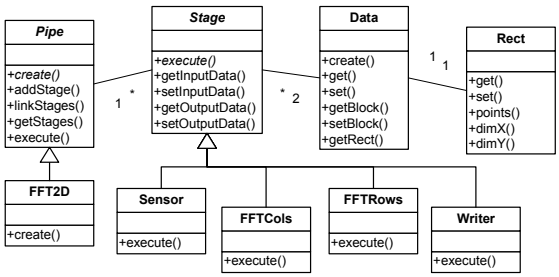

Figure 1: 2D-FFT class diagram

The class `FFT2D` models the problem as a sequential pipeline. The activities inherit from `Stage`. `Data` and `Rect` represent, respectively, complex matrices and rectangular regions. The implementation of some of the methods is described below:

```
int main(int argc, char **argv) {
  Pipe *pipe = new FFT2D;
  pipe->create();
  pipe->execute(NUM_MATS, pipe->getStages());
  delete pipe;
}

void FFT2D::create() {
  Stage *s1 = new Sensor(1);
  Stage *s2 = new FFTCols(2);
  ...
  Rect r(1,DIMX,1,DIMY);
  linkStages(s1,s2,&r);
  ...
  addStage(s1);
  ...
}
```

```
void Pipe::linkStages(Stage *s1,Stage *s2,Rect *r) {
  Data *d = new Data(r);
  s1->setOutputData(d);
  s2->setInputData(d);
  ...
}

void Pipe::execute(int iters,vector<Stage *> *stgs) {
  ...
  for (int i=0; i<iters; i++)
    for (int s=0; s<stgs->size(); s++) {
      stg = (*stgs)[s];
      stg->execute(&(stg->rect));
    }
}

void FFTCols::execute(Rect *r) {
  ...
  if ( iter == 0 )
    buffer = new Data(r);

  getInputData()->getBlock(buffer);  //input to buffer

  for (int x = r->x0; x<= r->x1; x++) {
    ...  // FFT on column x using buffer
  }
  getOutputData()->setBlock(buffer); //buffer tooutput
}
```

Each stage is associated with two `Data` objects, one of them being used as input and the other as output. Each time the stage is executed, the input is consulted so that its value can be used in the computation. The connection between two stages is implemented by sharing the same `Data` instance. This way, the i^{th} stage writes the result in the object which is read by the $(i+1)^{th}$ stage. In the class `FFTCols`, the creation of the intermediate data buffer is carried out only in the first iteration. The computation of the one-dimensional FFT entails a series of swapping and floating-point operations on the elements of the vector.

Compared to a classical implementation, the solution presented here offers some additional elements which are atypical in sequential scientific programs. They have been considered specifically with the aim of facilitating the parallelism integration by means of an AO language. Let us enumerate some of the elements:

- The degree of decomposition in terms of methods, interfaces and classes is high, therefore providing a rich set of join points to be intercepted.
- Activities are represented by different classes in the system, making the potential use of active elements easier (threads, distributed objects, etc.).
- Stages have to be linked explicitly and the linkage is based on sharing an object of class `Data`. This can be used as the basis for communicating active elements.
- Additional arguments in method calls are especially useful in two cases: for indicating the list of stages in the computation, and for setting the stage iteration range through an object of class `Rect`. The arguments can be altered by the aspect code.

Regarding the parallel execution, the solution can exploit both task and data parallelism. On the one hand, the pipeline can run its stages concurrently, which means that up to four matrices can be processed simultaneously. On the other hand, the stages

computing the FFT, which are the ones with higher computational cost, can divide the matrix into several blocks for an "embarrassingly" parallel computation. The following aspect establishes the degree of parallelism by setting both the type of data distribution and the number of blocks for every stage. This aspect declaration can be reused regardless of the parallelism implementation:

```
aspect Parallelism {
  advice "Stage" : slice class {
    int blocks;        // Number of blocks
    int distribution;  // Type of data distribution
  };

  advice construction("Stage") && that(s)
    : after(Stage &s){
    switch (s.id) {
      case 1:s.blocks=1; break;
      case 2:s.blocks=4;s.distribution=BY_COLS; break;
      case 3:s.blocks=4;s.distribution=BY_ROWS; break;
      case 4:s.blocks=1; break;
    }
  }
};
```

MULTITHREAD-BASED DESIGN

The execution of multiple threads is a way to exploit parallel hardware, as the operating system is able to run each thread on a different processor. Thread interactions can be based on variables which are allocated to a global memory space. Synchronization mechanisms are required to ensure the consistency of the shared data.

We have based the multithread implementation on the Adaptive Communication Environment ACE (Schmidt 2002), an object-oriented framework for the efficient development of concurrent communication software. The portability of ACE enables the same multithread code to be run on top of different interfaces such as POSIX PThreads, Solaris threads and Win32 threads.

The application described here uses groups of threads to execute the stages in parallel (one group per stage). The group size is indicated by the attribute `blocks` introduced in `Stage` by the aspect `Parallelism`, as stated at the end of the previous section.

Stage Connection

In the sequential version, the stages were connected by sharing a single instance of `Data`. This is a shortcoming in the parallel version since a stage running slower than the others will block the preceding ones, which will not be able to write in their output `Data` objects. A way to increase the performance is to link the stages using a buffer which supports the storage of multiple data elements. Figure 2 illustrates the idea. Four threads are running the code of `FFTCols`. Each time a result is produced, it is placed into a FIFO buffer from which the next stage (`FFTRows`, using four threads as well) retrieves its input data.

A `Fifo` class with the typical `get()/put()` methods is used for the implementation of the stage connections. As the `Fifo` objects will be accessed by different threads, synchronization mechanisms have to be considered. In our approach, the synchronization concern is developed in a separate aspect to be woven into the class `Fifo`:

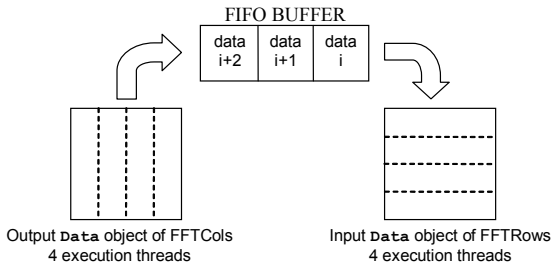

Figure 2: Stage connection scheme

```
aspect SynchFifo {
  advice "Fifo" : slice class {
    ACE_Thread_Mutex *mutex;
    ACE_Condition<ACE_Thread_Mutex> *notEmpty;
    ACE_Condition<ACE_Thread_Mutex> *notFull;
  };

  advice execution("void Fifo::get(...)") && that(f) :
    around(Fifo &f) {
    f.mutex->acquire();   // mutex acquired
    while (f.size() == 0)
      f.notEmpty->wait(); // Blocks if buffer is empty
    tjp->proceed();       // Element removal
    f.notFull->signal();  // The buffer has free space
    f.mutex->release();   // mutex released
  }
  ...  // advice on Fifo::put()
};
```

The second advice introduces code before and after the `get()` operation in order to control the removal of data. The body of the advice is accessed in mutual exclusion. If the `Fifo` buffer is empty, the thread is blocked in the `notEmpty` condition variable, waiting for any other thread to insert new data. Otherwise, a call to `tjp->proceed()` executes the code of `get()`. Then, we are sure that the buffer is not full, and so the variable `notFull` can be signaled, which will possibly wake up other threads which may be blocked in this variable because they can not complete a `put()` operation.

In order to integrate the new connection mechanism, the following advice is used to intercept the method `linkStages()`. Instead of sharing a single `Data` instance, adjacent stages will share a synchronized `Fifo` object.

```
advice "Stage" : slice class {
  Fifo *inputFifo;
  Fifo *outputFifo;

  ACE_Barrier *inputBarrier;
  ACE_Barrier *outputBarrier;
};

advice execution("void Pipe::linkStages(...)") &&
  args(s1,s2,r) && that(p)
  : after(Stage *s1,Stage *s2,Rect *r,Pipe &p) {
  ...
  Fifo *f = new Fifo(MAX_FIFO_SIZE);
  s1->outputFifo = f;
  s2->inputFifo = f;   // s1 and s2 share a FIFO
  ...
}
```

Thread Creation

The creation of threads is considered in the following advice, which affects the method `execute()` of `Pipe`:

```
advice execution("void Pipe::execute(...)") && that(p)
  && !cflow(execution("void *work(...)"))
  :  around(Pipe &p) {
  ...
  Vector<Rect> vr;

  for (int i=0; i<p.getStages()->size(); i++) {
    Stage *s = *(p.getStages())[i]; // For each stage
    s->rect.distribute(s->distribution,s->blocks,&vr);

    for (int j=0; j<s->blocks; j++) { //For each block
      arg = new ThrArg(j,iters,&p,s,&(vr[j]));
      ACE_Thread_Manager::instance()->spawn(
        (ACE_THR_FUNC)work,arg,
        THR_NEW_LWP|THR_JOINABLE);  // Thread launched
    }
  }
  ACE_Thread_Manager::instance()->wait(); //Wait
}
```

There is no call to `tjp->proceed()`, which means that the code join point will not be executed in this context. For each stage, the iteration range represented by the attribute `rect` is partitioned. Then, the group of threads is launched. The number of threads to be started is indicated in the attribute `blocks`. Every thread receives information through an object of type `ThrArg`, which includes the block identifier, the number of matrices to process, and references to the pipeline, the stage and the new iteration range.

Parallel Execution

The function `work()` executed by the threads saves the `ThrArg` variable and executes the pipeline using, this time, a stage list of only one element. The `ACE_TSS` template implements thread specific storage, which allows each thread to manage its own copy of `arg`:

```
ACE_TSS<ThrArg> arg;
static void *work(void *argument) {
  arg->set((ThrArg *)argument);
  arg->pipeline->execute(arg->iters,&(arg->stages));
}
```

Finally, data have to be moved from the `Fifo` buffers to the input and output `Data` objects each time a stage is executed. This involves the synchronization of threads:

```
pointcut invocations() =
  execution("void Sensor::execute(...)")  ||
  execution("void FFTCols::execute(...)") ||
  execution("void FFTRows::execute(...)") ||
  execution("void Writer::execute(...)");

advice invocations() && that(s) : around(Stage &s) {
  if ((s.getInputData()!= NULL) && (arg->id == 0))
    s.inputFifo->get(s.getInputData());
  s.inputBarrier->wait();   // Waits to complete input

  Rect **pr = (Rect **)tjp->arg(0);
  *pr = &(arg->rect);       // Iteration range changed

  tjp->proceed();          // Stage execution
  s.outputBarrier->wait(); // Waits to end computation

  if ((s.getOutputData()!= NULL) && (arg->id == 0))
    s.outputFifo->put(s.getOutputData());
}
```

The pointcut `invocations()` refers to the execution of any of the stages. When the advice code is triggered, the thread with `id` zero retrieves a new matrix from the input `Fifo` buffer and updates the input `Data` object. The other threads assigned to the stage are blocked until this operation is complete. Before the call to `tjp->proceed()`, the `Rect` argument, which denotes the iteration range, is replaced with the value stored in the variable `arg`. The threads wait until the code of the stage is executed. Then all the parts of the output `Data` object has been correctly updated with new values. The thread with `id` zero is allowed to insert the result into the output `Fifo` buffer.

MPI-BASED DESIGN

MPI is a collection of routines widely used in the development of parallel programs on architectures with distributed memory including computer networks. The programming model consists of a set of processes communicating by means of message-passing.

The aspect presented in this section will transform the sequential program into an SPMD application using MPI.

Application Set-up

The following advice ensures that the message-passing environment is initialized and terminated correctly:

```
advice execution("int main(...)") && args(ac,av)
  : around(int ac, char **av) {
  MPI_Init(&ac,&av);
  tjp->proceed();
  MPI_Finalize();
}
```

Unlike the code in section 4, the group of processes of an MPI application is created in a static way when the application is started. A specific computation has to be assigned to each process in the group. So, correspondence between the process rank and the pair (stage, iteration range) is required. The association in the opposite direction is also needed. The aspect defines the methods `mpiToProblem()` and `problemToMpi()` in order to carry out these mappings. Their codes are quite simple as they use the information set by the aspect `Parallelism`.

In the multithread version, the threads accessed data using shared memory, so the parallelization aspect was mainly focused on the creation and synchronization of threads. Owing to the distributed nature of MPI, the data flow in the pipeline has to be implemented by means of message-passing. The parallelization aspect in this section will be mainly focused on communications.

Data distribution must be taken into account in order to implement efficient point-to-point communications on the stage interactions. Figure 3 depicts this scenario. The stage on the left uses four processes, each one computing the FFT (on columns). When the result is passed to the next stage, data is partitioned to send remote processes the exact data pieces they require. In

this figure, process 3 has to send specific data to processes 5, 6, 7 and 8. As can be noted, the information about the data distribution and the number of blocks of the stages is essential. New attributes and methods are introduced in some classes:

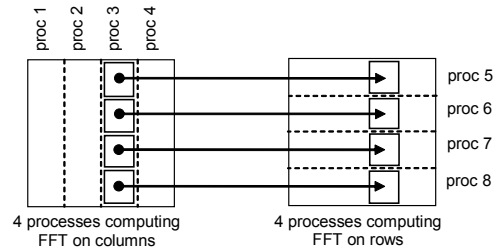

Figure 3: Communication between stages

```
advice "Stage" : slice class {
  vector<Data *> inputPartition;
  vector<Data *> outputPartition;
};

advice "Data" : slice class {
  int remote;         // Remote process
  Stage *preceding;   // Preceding stage
  Stage *next;        // Next stage

  void send(int p) { // Sends data to process p
    MPI_Send((void*)ptr,rect.points()*sizeof(complex),
      MPI_BYTE,p,MYTAG,MPI_COMM_WORLD);
  }
  void recv(int p) { // Receives data from process p
    MPI_Status status;
    MPI_Recv((void*)ptr,rect.points()*sizeof(complex),
      MPI_BYTE,p,MYTAG,MPI_COMM_WORLD,&status);
  }
};
advice execution("void Pipe::linkStages(...)") &&
  args(s1,s2,r) : after(Stage *s1,Stage *s2,Rect *r) {
  s1->getOutputData()->preceding = s1;
  s2->getInputData()->next = s2;
}
```

The meaning of the code is simple. The class `Stage` is augmented with two attributes which represent two data partitions used to receive (send) data from (to) other processes. For instance, the list `outputPartition` of the process 3 (again in figure 3) will contain four Data elements, each one being used to send a specific data piece to other process. The class `Data` is also augmented with pointers to the adjacent stages and operations to carry out the communication using MPI primitives.

The aspect declares some data members useful for the computation, such as `myrank`, `myid`, `myblock`, `mystg`, and `mystglist`. They are initialized in the following advice, which also includes the code needed to set-up the two data partition lists of the stage pointed to by `mystg` (this code has been omitted):

```
advice execution("void FFT2D::create(...)") && that(p)
  : after(Pipe &p) {
  ...
  vector<Rect> vr;
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);  // My rank
  mpiToProblem(p.getStages(),myrank,&myid,&myblock);

  mystg = p.getStageById(myid); // This is my stage
  mystglist.push_back(mystg); // List with one stage

  mystg->rect.distribute(mystg->distribution,
    mystg->blocks,&vr);
  myrect.set(&(vr[myblock])); //My iteration sub-range
  ...// inputPartition and outputPartition are filled
}
```

Parallel Execution

Finally, the following advices ensure that the parallel program will be executed correctly:

```
advice execution("void Pipe::execute(...)") :before {
  vector<Stage *> **stglist =
    (vector<Stage *> **)(tjp->arg(1));
  *stglist = &mystglist;    // Replaces the stage list
}

advice invocations() && that(stg) : around(Stage &stg)
{
  Data *d;
  // Receives data from other processes
  for (i=0; i<stg.inputPartition.size(); i++) {
    d = stg.inputPartition[i];
    d->recv(d->remote);
    stg.getInputData()->setBlock(d);
  }

  Rect **pr = (Rect **)(tjp->arg(0));
  *pr = &(myrect);  // Iteration range is changed

  tjp->proceed();   // Stage code execution

  // Sends the result to other processes
  for (i=0; i<stg.outputPartition.size(); i++) {
    d = stg.outputPartition[i];
    stg.getOutputData()->getBlock(d);
    d->send(d->remote);
  }
}
```

The first advice is needed to replace the original list of stages, which is the second argument of `execute()`, with another list containing one stage only. This way, the active element (i.e. the process) will address the execution of a single stage, instead of the complete pipeline.

The second advice uses the pointcut `invocations()` which was described in the previous section. The advice code is triggered when `execute()` is called on any of the subclasses of `Stage`. First, the process has to receive new data. More specifically, every object of `inputPartition` receives data from the corresponding remote process. These values are used to update the input `Data` object. Before the stage code is executed, the iteration range is restricted using the variable `myrect`. When the result is computed, it is sent to the processes associated with the next stage. This is done using the elements of the attribute `outputPartition`.

CONCLUSIONS

The parallelization of sequential scientific programs may entail significant changes in the code for tackling the creation, communication and synchronization of the computational tasks. In addition, the algorithm itself may be sufficiently different in the parallel version. Once these changes are applied, the resulting code probably suffers from code-tangling problems because the parallelism concerns obscure the numerical computation. Therefore, the applications become more difficult to maintain. This paper discuses the use of AOP to improve the management of the code parallelization. The benefits of the approach, as mentioned in section 1, can be expressed in terms of code modularization and reuse. It can be very difficult

to determine a theoretical result on the advantages and disadvantages of this approach. The code parallelization is traditionally done ad-hoc, so the process may vary significantly from one application to another. Thus, our conclusions will be based on the case study presented in this paper.

We consider the results to be quite promising. The complexity of the case study is moderated as it deals with the efficient integration of both task and data parallelism. The scientific core that calculates the 2D-FFT sequentially was successfully adapted to two distinct parallel scenarios (multithreading and MPI) which are characterized by quite different programming models. The only mechanism used in the code parallelization was aspect weaving (based on AspectC++, in this case). As a result, the new parallel applications have the parallelism concerns modularized into one or several aspects, and thereby code-tangling is reduced. Moreover, the same scientific core can be reused in different applications which can exploit very different parallelism models.

Regarding the development of the sequential core, the mechanisms that allowed the core to be parallelized using aspects were not difficult to implement. Basically, we have considered a good decomposition in terms of classes and interfaces, and we have included some elements, such as the iteration range of the stage, as additional arguments in some methods. Although this paper is focused on a particular case study, we are currently applying this approach to other numerical applications with more complex parallelism patterns successfully.

REFERENCES

Briham, E.O. 1988. *The Fast Fourier Transform and Its Applications*. Prentice-Hall International.

Díaz, M. et al. 2005. "An Aspect-Oriented Framework for Scientific Component Development". In *Proc. of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing PDP05* (Lugano, Switzerland). IEEE. 290-296.

Harbulot, B. and Gurd, J. 2004. "Using AspectJ to Separate Concerns in Parallel Scientific Java Code". In *Proc. of the 3rd International Conference on Aspect-Oriented Software Development AOSD04* (Lancaster, UK). ACM. 122-131.

Harbulot, B. and Gurd, J. 2006. "A Join Point for Loops in AspectJ". In *Proc. of the 5th International Conference on Aspect-Oriented Software Development AOSD06* (Bonn, Germany). ACM. 63-74.

Kiczales, G. et al. 1997. "Aspect-Oriented Programming". In *Proc. of the European Conference on Object-Oriented Programming ECOOP97* (Jyvskyl, Finland). Springer-Verlag. 220-242.

Kiczales, G. et al. 2001. "An Overview of AspectJ". In *Proc. of the European Conference on Object-Oriented Programming ECOOP01* (Budapest, Hungary). Springer-Verlag. 327-353.

Schmidt, D. and Huston, S. 2002. *C++ Network Programming: Mastering Complexity with ACE and Patterns*. Addison-Wesley.

Sobral, J.L. 2006. "Incrementally Developing Parallel Applications with AspectJ". In *Procs. of the 20th International Parallel & Distributed Processing Symposium IPDPS06* (Rodhes, Greece). IEEE.

Spinczyk, O. et al. 2002. "AspectC++: An Aspect-Oriented Extension to C++". In *Proc. of the 40th International Conference on Technology of Object-Oriented Languages and Systems TOOLS02* (Sydney, Australia). Australian Computer Society. 53-60.

AUTHOR BIOGRAPHIES

MANUEL DÍAZ. He received his M.S. and Ph.D. degree in Computer Science from the University of Málaga in 1990 and 1995, respectively. At present he is an Associate Professor in the Department of "Lenguajes y Ciencias de la Computación" (LCC) of the University of Málaga. He has worked in the areas of distributed and parallel programming and in real-time systems.

SERGIO ROMERO. He received his degree in Computer Science from the University of Málaga in 2003. At present he is a PhD student in the University of Málaga, where he works on the use of high-level software technologies, e.g. component- and aspect-based approaches, for the development of parallel and distributed numerical applications.

BARTOLOMÉ RUBIO. He received his M.S. and Ph.D. degree in Computer Science from the University of Málaga in 1990 and 1998, respectively. At present he is an Associate Professor in the Department of LCC of the University of Málaga. He has worked in the areas of distributed and parallel programming and coordination models and languages.

ENRIQUE SOLER. He received his M.S. and Ph.D. degree in Computer Science from the University of Málaga in 1990 and 2001, respectively. At present he is an Associate Professor in the Department of LCC of the University of Málaga. He has worked in the areas of distributed and parallel programming, especially in the context of high-performance scientific computing.

JOSÉ M. TROYA. He received his M.S. and Ph.D. degrees in Computer Science from the University Complutense of Madrid in 1975 and 1980, respectively. He has been a Full Professor at the Department of LCC of the University of Málaga. He has worked on parallel algorithms for optimization problems and on software engineering for distributed systems. He has been the head of the Software Engineering Group since its foundation in 1990.