Integrating HPF in a Skeleton Based Parallel Language*

C. Gennaro
Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche (C.N.R.)
Via Alfieri 1, Pisa, 56010 Italy
gennaro@iei.pi.cnr.it

S. Orlando
Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino 155, Venezia Mestre, 30172 Italy
orlando@unive.it

R. Perego
Istituto CNUCE
Consiglio Nazionale delle Ricerche (C.N.R.)
Via Alfieri 1, Pisa, 56010 Italy
Raffaele.Perego@cnuce.cnr.it

Abstract

Although HPF *allows programmers to express data–parallel computations in a portable, high–level way, it is widely accepted that many important parallel applications cannot be efficiently implemented following a pure data–parallel paradigm. For these applications, rather than having a single data–parallel program, it is more profitable to subdivide the whole computation into several data-parallel pieces, where the various pieces run concurrently and co–operate, thus exploiting task parallelism. This paper discusses the integration of* HPF *with* SkIE*, a skeleton based coordination language implemented on top of MPI (Message Passing Interface), which permits to describe complex computational parallel structures. We show how* HPF *can be used inside common forms of parallelism, e.g. pipeline and processor farms, and we present experimental results regarding a sample application.*

1 Introduction

It is widely accepted that many important parallel applications cannot be efficiently implemented following a pure data-parallel paradigm [5]. Although the data-parallel programming model provided by HPF [9] hides low-level programming details and let programmers to concentrate on the high-level exploitation of data parallelism, many important parallel applications do not fit into a pure data-parallel model and can be more efficiently implemented by exploiting both task and data parallelism. The advantage of exploiting both forms of parallelism is twofold. On the one hand, the exploitation of parallelism at different levels may significantly increase the scalability of applications which may exploit only a limited amount of data parallelism [8]. On the other hand, the capability of integrating task and data parallelism into a single framework allows the number of addressable applications to be enlarged. Task parallelism is in fact often needed to reflect the natural structure of an application. For example, applications of computer vision, image and signal processing, can be naturally structured as pipelines of data–parallel tasks where stages which deal with external devices may be run on a few processors to better match the available I/O bandwidth, while the remaining processors can be more efficiently used to run computation intensive parts of the application.

In this paper the problem of integrating HPF with a language based on skeletons (SkIE) is addressed. A skeleton is a high-order parallel form, capturing a common and widespread used type of parallelism. In particular, each skeleton describes a computational structure for which it is possible find an efficient parallel implementation. This methodology imposes to use only skeletons to exploit parallelism in a program and complex parallel computations can be described nesting skeletons.

In SkIE (Skeleton based Integrated Environment), when parallelizing a sequential application, the user can reuse large chunks of sequential code written in the most common sequential languages (e.g., C, C++, F77, F90, Java) encapsulating them in modules which can then be composed to develop a larger application. In this paper we enhance the expressivity of SkIE offering a tool for integrating data parallel parts developed using HPF.

This paper is organized as follow. Section 2 gives an overview of SkIE. Section 3 illustrates ADAPTOR the public

* This work has been partially founded by HPCC/SEA contract number EU1063.

domain HPF compilation system used for our integration. Section 4 presents the usage of HPF in SkIE and discusses the problems and relative solutions about its integration. In Section 5 a case study is given. Section 7 contains conclusions.

2 The SkIE developing environment

SkIE is an integrated environment which allows the rapid development of complex applications on several parallel platforms [2, 3]. SkIE has been developed in the PQE2000 project [10, 11], a joint initiative of the main Italian research agencies (CNR, ENEA, INFN) and of Finmeccanica's QSW (Quadrics Supercomputers World Ltd) for development of innovative HPC general-purpose systems and their applications to industry, commerce and public services.

SkIE includes a coordination language SkIE–CL which is based on the state of the art skeleton methodology. The language includes a well defined set of skeletons. These skeletons can be nested, supplying the programmers with a powerful mechanism to specify parallel applications. In SkIE–CL when parallelizing a sequential application, the user can reuse large portions of sequential code written in the most common sequential languages (e.g., C, C++, F77, F90, Java) encapsulating them in modules which can then be composed to develop a larger application.

Sequential modules written in different languages can be mixed in the same parallel application and, in the current version, the user can also encapsulate parallel code using plain MPI and specialized libraries (such as standard numerical libraries [6]).

SkIE–CL helps the programmer in the design of the global structure of his application by providing a collection of optimized and ready-to-use typical skeletons which can be instantiated and combined to define parallel applications. Examples of common skeletons provided are the processor farm, in which a pool of worker processes computes a pool of independent tasks, and the pipeline, which exploits parallelism in different phases of the computation. In SkIE these patterns can be freely composed to build more complex structures and SkIE–CL automatically generates an optimized implementation of the compositions of the skeletons provided. This means that when using a SkIE skeleton, the support not only generates the code needed for parallel interaction automatically, but also optimizes the resources allocated to each skeleton, decides the best granularity of computation and locates inefficiencies in the global structure.

3 The HPF compiler ADAPTOR 6.0

ADAPTOR (Automatic DAta Parallelism TranslatOR) is a public domain HPF compilation system developed at the SCAI institute (GMD) during the last years [4]. The tool transforms data parallel programs written in HPF (i.e., a Fortran language with array extensions, parallel loops, and layout directives) into programs with explicit message passing. Beside some restrictions, it supports the new standard HPF 2.0 as well as many of the approved extensions within this new standard. The ADAPTOR package consists of:

- the source to source transformation tool *fadapt*,
- the distributed array library *DALIB* . This is the HPF runtime system that handles descriptors for arrays, sections and distributions, and implements communication routines,
- the compiler driver *gmdhpf*.

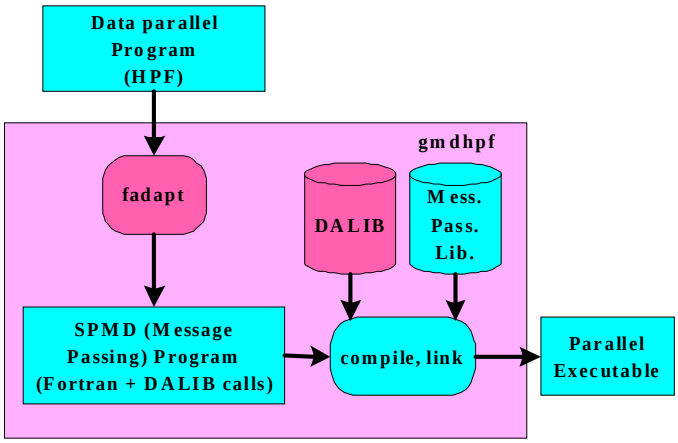

Figure 1. Overview of the ADAPTOR tool.

Figure 1 shows how the different components work together. The compiler driver gmdhpf invokes the source-to-source translation fadapt that generates an SPMD program with message passing from the HPF program. Afterwards, it invokes a native FORTRAN 77 or Fortran 90 compiler to compile the generated code. Finally, the compiled codes are linked with the DALIB runtime system and the utilized message passing library.

The ADAPTOR source–to–source translator fadapt, the DALIB runtime system, and the compile driver gmdhpf are available as C programs.

For the generated code to be compiled, linked and run, the following additional components are required:

- A MPI or PVM message-passing subsystem, for some systems (IBM SP, Intel Paragon, CM 5 and Meiko CS–2) the native message passing subsystem can be used directly.
- A FORTRAN 77 compiler.

- A parallel programming environment in which parallel programs can be loaded and executed on a parallel machine.

4 Integrating HPF in SkIE

In this Section we present the approach used for integrating HPF in the coordinator language SkIE–CL. As an example of our approach and without loss of generality, a pipeline consisting of three stages is presented. The first and the last stage of the pipeline are simple C language sequential tasks while the second one is a data–parallel HPF task compiled with ADAPTOR.

4.1 A simple pipeline of three stages

Suppose we have the three stages pipeline shown in Figure 2, where in particular:

- the first and the third stages of the pipeline are *sequential tasks* written in C language, while the second is a data–parallel task written in HPF.
- Each task (either sequential or data-parallel) is assigned to a disjoint set of processors.
- The tasks communicate by means of MPI using the SkIE global communicator SKIE_COMM_WORLD (i.e, a global communicator defined by the communication layer of SkIE–CL that, at the program startup, is a copy of MPI_COMM_WORLD).

The first stage (init) generates a stream of matrices of doubles, the stage HPF (StageHpf) receives a matrix X, performs some computation on X and outputs a new matrix Y. Finally, the last stage (stop) performs some final computation on the received matrix Y.

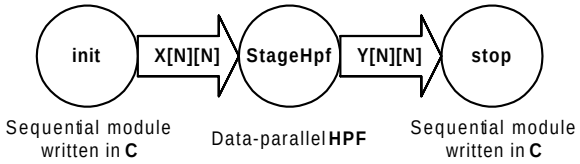

Figure 2. Example of pipeline composed of three stages, where the second one is a data–parallel HPF.

4.2 The SkIE implementation of the pipeline

The code of the simple pipeline described above would be (for further details see [11, 1]):

```
pipe main in() out()
  init        in()       out(stream of dou-
ble x[N][N])
  StageHpf    in(x)      out(double y[N][N])
  stop        in(stream y) out()
end pipe

init in() out(stream of double x[N][N])
$c{
/*
    C code that initializes x
*/
}c$
end

stop in (stream of double y[N][N]) out()
$c{
/*
    C code that uses y
*/
}c$
end

StageHpf in(double x[N][N]) out(double y[N][N])
$hpf{
! HPF Preamble: setting the stage status

!skie$ %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

! HPF Body: code that uses x and produces y
}hpf$ end

pragma parallelism degree 4 in StageHpf;
```

The main pipe construct declares a pipeline of three stages which exchange data representing the stream, as illustrated in the previous section. The following three blocks of code define the stages belonging to the declared pipeline. The pair of tags: $c{ }c$ and $hpf{ }hpf$ declare that the code inside the block is written in C and HPF, respectively.

We focus on the module HPF that is the subject of this paper. It is important to note the special comment !skie$ inside the module HPF. This comment works as a separator: the section of code before and following the special comment are called *Preamble* and *Body*, respectively. The Preamble is executed once at the start–up of the program and comprises the local variables of the module, the HPF directives and a possible initialization code. The Body is the main part of the module and it is executed every time a new stream item arrives from the previous stage.

The pragma directive instructs the SkIE–CL compiler that the stage HPF must be executed on 4 processors.

4.3 The local communicators

We discuss now the problem of the use of MPI as low–level communication layer, both for skeleton inter–task communications of SkIE–CL and for HPF (i.e., the communications performed inside the run–time ADAPTOR).

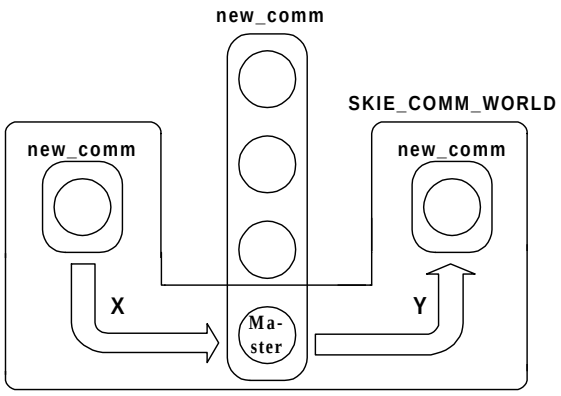

Figure 3. Communicator splitting.

Our solution is the following: we split the global communicator `MPI_COMM_WORLD` into as many sub–communicators as the number of stages of the pipeline. Each of these sub–communicators, that we refer as `new_comm`, comprises the virtual processors that belong to each stage of the pipeline. It is worth observing that, in the case of sequential tasks (in our case, the first and the last stage), the local communicators will comprise only a single virtual processor.

The advantage of using local communicators is that we need only a slight modification of the run–time of `ADAPTOR`: all the communications of the data–parallel generated by `ADAPTOR` that uses the global communicator `MPI_COMM_WORLD` must now refer the local communicator `new_comm`.

In order to simplify the integration of `HPF` and also modify as less as possible SkIE–CL, we use the communicator `SKIE_COMM_WORLD` for grouping the processors that are involved in the stream communications. Figure 3 shows how the different communicators work. As can be seen, only one of the processors of the data–parallel `HPF` is included in `SKIE_COMM_WORLD`. We refer to this processor as *Master*, that is the manager of the communications from/to the other stages of the pipeline. The processor of the stage `HPF` besides the Master, called *Slaves*, will be "hidden" from the perspective of the SkIE–CL communication layer.

4.4 The structure of the stage HPF.

As explained in Section 4.2, the variables defined in the Preamble will be instanced at the start–up of the program and will live for its total execution time (By default, using different language from `HPF`, the local variables are not static). This behavior can be obtained encapsulating the code of the stage `HPF` in a special template written in `HPF`. The structure of this template is shown in Figure 4.

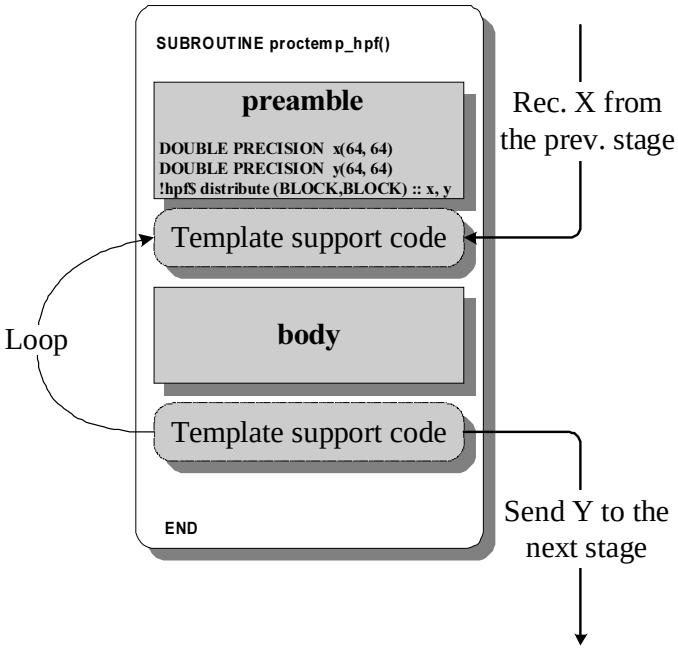

Figure 4. The structure of the `HPF` template.

The template corresponds to an `HPF` subroutine, `proc_temp_hpf`, which is called by the task loader of SkIE–CL at the program start–up and is executed by all of the processors of the stage `HPF` in a SPMD fashion. The template starts with the Preamble code and proceeds with a support code that manages the communications and executes the Body part of the `HPF` module. The first part of this support code, receives the matrix X (in general, it receives all the input data of the stage `HPF`). After that, the Body code is executed. Finally, the second part of the support code is executed. This part sends the results (the matrix Y) to the next stage of the pipeline. The template starts again receiving a new matrix X, and so on, until the pipeline stream finishes.

The communications from/to the stage `HPF` are obtained through two external procedures written in `C`. Moreover, because only the Master processor must be involved in the communications we need a mechanism that permits the distribution/gather of the received/sent arrays. The semantic of `HPF EXTRINSIC SERIAL` procedures ensure the required behavior. More specifically, if the external communicating procedures are defined as `EXTRINSIC SERIAL`, we have that:

- Only one processor, among the ones involved in the execution of the stage `HPF` (i.e., the Master) executes the `EXTRINSIC SERIAL` subroutines used for communicating. The Slave processors synchronize with the Master waiting for its return.

- A new copy of each parameter of the invoked subroutine is allocated in the Master processor.
- The input/output parameters are conveniently updated on the basis of type of their interface declaration: input, output or input–output (`INTENT(IN,OUT,INOUT)`).
- If a replicated variable (e.g., in the case of scalar variables) is passed to the subroutine as `INTENT(IN)`, the copy of the parameter will involve only the Master processor. On the contrary, if the parameter is declared as `INTENT(INOUT)`, the Master processor will broadcast it to the Slave processors at the subroutine return, in order to update the replicated variables.
- If a parameter of the subroutine is distributed, either if it is declared as input or is declared as output, it is necessary to gather or to distribute portion of the array that are allocated in the distinct processors of the stage with respect the local copy of the Master.

In order to better understand how the different communication mechanisms work together, in Figure 5 we show the distinct communications that characterize the receive phase of the `HPF` stage of the pipeline.

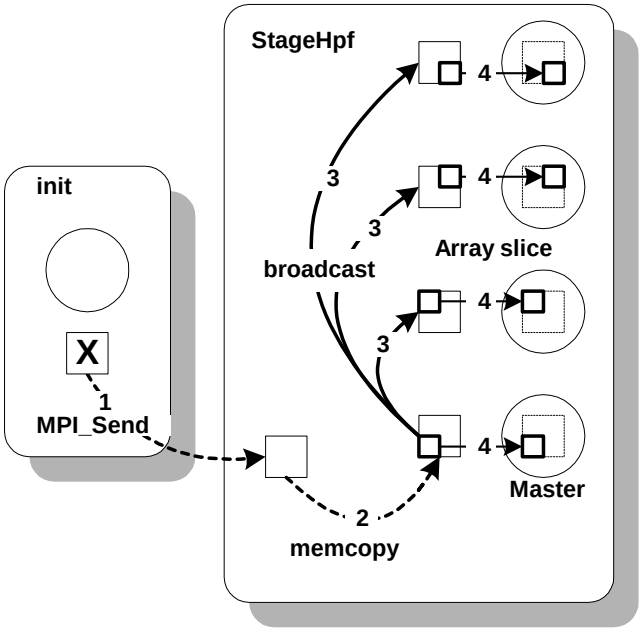

Figure 5. Receive phase of the array X.

The dashed arrows represent the operations performed by the SkIE–CL support system, while the solid arrows the operations relative the run–time of `ADAPTOR` (the numeric labels identify the items of the following list):

1. The first stage of the pipeline sends the matrix X to the stage `HPF`. The matrix is received by the support code (written in `C`) of the template above described.
2. The array is transposed during its copying. This is necessary for translating the array from the SkIE–CL standard which stores multidimensional arrays in row-major order (in the same way as `C` language does) to the `HPF` standard which stores multidimensional arrays in column-major order.
3. The Master processor broadcasts the matrix X to the Slave processors.
4. Array slice: because the matrix X is declared as distributed, each processor must copy its section of the array X in a local slice copy.

5 A case study

In this section a case study is given that demonstrates how the proposed tool can be applied when integrating a data–parallel program written in `HPF` in a SkIE program. The case study is a classical FFT 3–D transform which is probably the application most widely used to demonstrate the usefulness of exploiting a mixture of both task and data parallelism [5, 7]. FFT transformations are commonly used in the field of signal and image processing applications, which generally require the FFT to be applied in real–time to a stream of frames acquired from an external device.

The SkIE code implementing a 3-D FFT has the following structure (We omit the code of the part of the program written in C in the interest of space):

```
pipe main

matrix_generator
in()
out(stream of double xr[SIZE][SIZE][SIZE],
          stream of double xi[SIZE][SIZE][SIZE])

fft3d
in(xr,xi)
out(double yr[SIZE][SIZE][SIZE],
      double yi[SIZE][SIZE][SIZE])

matrix_printer     in(yr, yi) out()

end pipe

pragma parallelism degree 8 in fft3d;

fft3d    in (double xr[SIZE][SIZE][SIZE],
 double xi[SIZE][SIZE][SIZE])
         out(double yr[SIZE][SIZE][SIZE],
 double yi[SIZE][SIZE][SIZE])

$hpf{
  integer i1, i2, i3
```

```
!hpf$   distribute xr(BLOCK,*,*)
!hpf$   distribute xi(BLOCK,*,*)

!skie$ %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

!hpf$   independent
 do i1=1,SIZE
  do i2=1,SIZE
   call fft_slice(xr(i1,i2,:),xi(i1,i2,:),SIZE,1)
  end do
 end do

 forall (i1=1:SIZE)
  xr(i1,:,:) = transpose(xr(i1,:,:))
  xi(i1,:,:) = transpose(xi(i1,:,:))
 end forall

!hpf$   independent
do i1=1,SIZE
 do i2=1,SIZE
  call fft_slice(xr(i1,i2,:),xi(i1,i2,:),SIZE,1)
 end do
end do

forall (i2=1:SIZE)
 xr(:,i2,:) = transpose(xr(:,i2,:))
 xi(:,i2,:) = transpose(xi(:,i2,:))
end forall

!hpf$   independent
do i1=1,SIZE
 do i2=1,SIZE
  call fft_slice(xr(i1,i2,:),xi(i1,i2,:),SIZE,1)
 end do
end do

yr = xr
yi = xi

}hpf$
end
```

The `pipe` directive defines a pipeline of three stages. The first stage, named `matrix_generator`, generates a stream of $SIZE \times SIZE \times SIZE$ 3-D matrices (that we call *cubes*) to be transformed. The second stage is the `HPF` stage that performs the FFT3D transform. The last stage, named `matrix_printer`, prints the results of the FFT3D transform. The matrices `xr` and `xi` are the respective real and imaginary parts of the cubes to be transformed. Similarly, `yr` and `yi` represent the real and imaginary part of the transformed 3D matrix, respectively.

The stage `HPF` performs $SIZE$ independent 1–D FFT transform (i.e., the pair of do–loop with `fft_slice`) over the three spatial coordinates x, y and z on the input cube. Actually, each do–loop executes the transformation over the same spatial direction, but because between a do–loop and the successive we properly transpose the cube, we obtain the transformation according the three spatial coordinates. The cube transpositions are performed by forall–loops that transpose the matrices belonging first to the plane y, z and then to the plane x, z. Note that the cubes are distributed over the first spatial dimension. This allows the 1–D FFTs to be applied in parallel to each memory-contiguous column of the cubes. No communications are generated by the `HPF` compiler within the three parallel do–loops, while cube transpositions involve all–to–all communications.

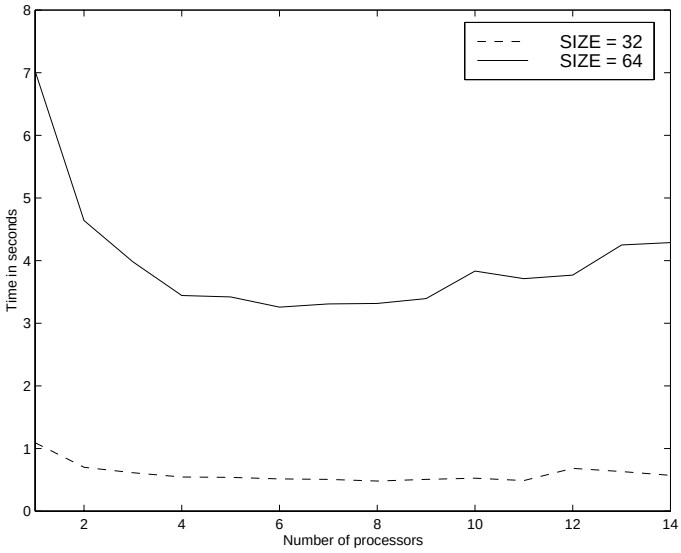

Figure 6. Execution time for SkIE FFT3D as a function of the number of processors of the stage `HPF`, for different problem size. Single task.

Figure 6 shows the total execution time of the program FFT3D on the QSW QM1 machine, as a function of the number of processors of the stage `HPF`. Throughout this and next experiments, we do not consider the time spent by `matrix_printer`. The solid and the dashed curves denote the execution time for stream of one cube of $SIZE = 64$ and $SIZE = 32$, respectively. In this experiment, we evaluate the performance of the simple pipeline structure of Figure 2, when executing just one task. In this way, we can evaluate how the FFT3D stage scales and the cost of the pipeline communications. In both cases the program does not practically scale when the number of used processors is greater that 4. This is due to the communication cost among the stages of the pipeline and the communication for the cube transpositions (the for–all loops). In order to have an estimate of the overhead for sending/receiving the matrices to/from the `HPF` stage (including also the overhead of the template `HPF` described in Section 4.4), we compare the execution time of the SkIE version with the one exhibited by a version entirely written in `HPF`. To compare the results, the same `HPF` compiler (Adaptor), the same data layouts and the same parallelization strategy (except the task

parallelism, of course) were used for both the pure HPF and SkIE implementations of the sample applications.

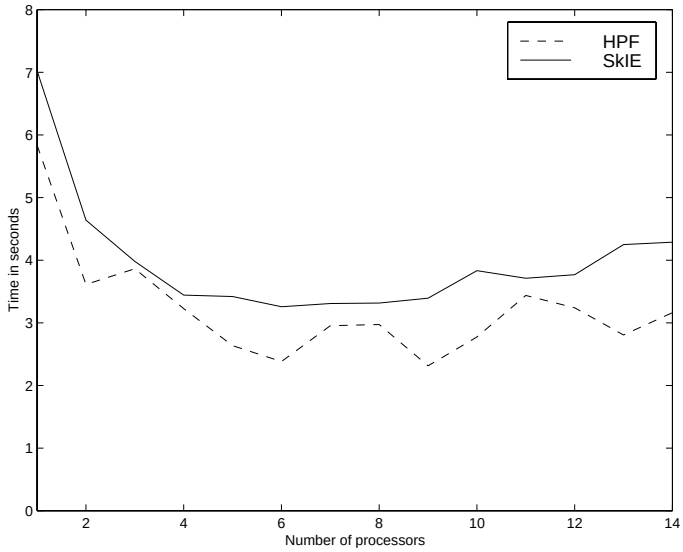

Figure 7. SkIE–CL **fft3d vs pure** HPF **fft3d. Problem size:** $64 \times 64 \times 64$**. Single task.**

Figure 7 makes this comparison for the worst case $SIZE = 64$, we can see that the overhead introduced by SkIE–CL is acceptable (The time spent by the matrix generation is about 1 second. This time is also present in the SkIE implementation because, in this case, the number of matrices elaborated is 1).

6 Exploiting task parallelism

The benefits of using a task parallelism developing environment as SkIE with respect a pure data–parallel one, can be observed when the data length stream is greater than one. In fact, as we shall see, SkIE–CL gives better performance because it is possible to overlap the elaboration of the i–th item of the stream with the elaboration of the $(i-1)$-th item of the stream, in a pipeline fashion. For this reason, it is convenient to split the stage HPF into three new stages each one performing a transformation over one of the three spatial coordinates.

Figure 8 shows the program structure. The first stage matrix_generator generates a stream of 100 cubes of complex numbers (the pair of matrix xr and xi) and the following three stages HPF, called fft3d1, fft3d2 and fft3d3, perform the three spatial FFT transformations.

Another technique of optimization is to replace the FFT3D sub–pipe with a processor farm construct, i.e., using a replication technique (see Figure 9). Suppose a set of p processors is available for executing the HPF FFT3D, it is

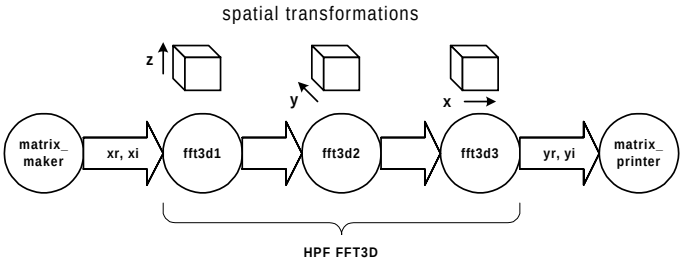

Figure 8. Structure of FFT3D pipeline implementation.

profitable to divide the p processors in two or more disjoint groups and process alternate cubes on these disjoint groups of processors. Clearly, this is feasible only if we are not interested in the order of transform results.

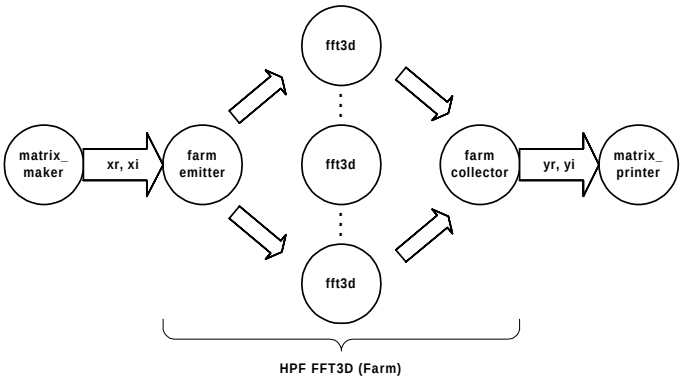

Figure 9. Structure of FFT3D farm implementation.

In Figure 10, we compare the behavior of the two different SkIE implementations with the pure HPF implementation. Due to the different implementation strategies not all processor allocations are convenient or feasible when using SkIE. In the case of pipeline implementation, it is convenient to distribute the available processors among the HPF stages as uniformly as possible. Instead, in case of farm implementation, we need two further processors, one for emitter and one for the collector (see Figure 9). Moreover, because we have chosen to allocate two processors[1] for each one of the w workers (i.e., the replicated modules) of the farm, the total number of processor is given by $2 + 2 * w$.

From these results we can anyway see that the SkIE im-

[1] We have conducted several experiments to determine that the optimal number of processors to assign to the workers is two.

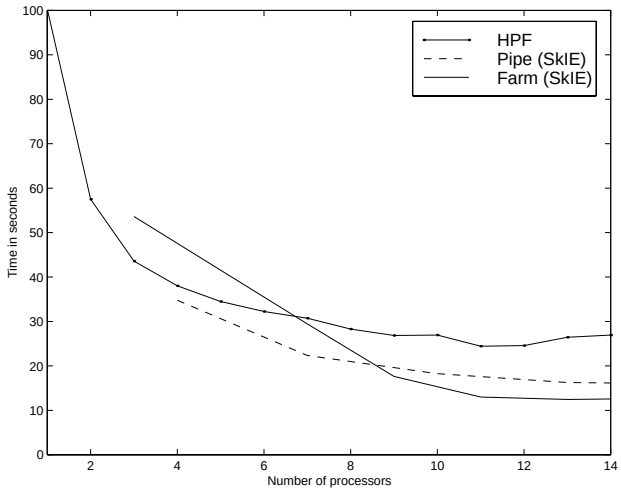

Figure 10. Comparison of execution times obtained with pipeline, farm parallelism and pure `HPF` implementation of the FFT3D. Stream length = 100, $SIZE = 32$.

plementation provides the best performance when the number of available processor is greater than 4 and that, in particular, the farm solution scales better than the others.

7 Conclusions

In this paper we have discussed the integration of `HPF` with SkIE, a skeleton coordination language implemented on top of MPI. The main idea behind our approach is to allow programmers to coordinate `HPF` tasks with modules written in different languages as C, C++, Java, according to specific paradigms for task parallelism. These paradigms, for example pipelines and processors farms, are the most commonly encountered in parallel applications, and, more importantly, can be associated with simple analytic performance models that, on the basis of profiling information, can be used to automatically solve the problem of optimal resource allocation. According to this approach, programmers have only to specify the `HPF` source code of each task, and the high-level constructs which specify the coordination among the various tasks. The associated compiler, depending on the specific paradigm, generates the suitable "system" code for carrying out task creation and interaction, and integrates the `HPF` user-provided code with the compiler-generated SkIE calls for intertask communications. For example, if the paradigm is a pipeline, it generates the code to process streams of data, while, for a processor farm paradigm, it produces the code which dynamically schedules incoming stream elements in order to balance the workload.

Finally, we have presented some encouraging performance studies, conducted on a QSW QM1 machine. For the experiments we used a real sample application. We structured this application in order to exploit a mixture of task and data parallelism, and we compared two different implementations with a pure data parallel one. We have presented a classical 3–D fast Fourier transform, which was structured as a three–stage pipeline and as a processor farms. We observed that the mixed task/data–parallel versions of such application always achieved performance improvements over the pure data parallel counterparts. These improvements ranged from a few per cent up to 200%.

References

[1] ENEA: PQE2000 Project. `www.pqe2000.enea.it/home/pqe1/Docum_a.html`.

[2] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: an heterogeneous HPC environment. *Parallel Computing*, 25:1827–1852, 1999.

[3] B. Bacci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Skeletons and transformations in an integrated parallel programming environment. September 1999. To be presented at *PaCT'99*, St. Petersburg, September 1999.

[4] T. Brandes. ADAPTOR Programmer's Guide Version 7.0. Internal Report Adaptor 3, GMD-SCAI, Sankt Augustin, Germany, Nov. 99. URL: `www.gmd.de/SCAI/lab/adaptor/adp_docs.html`.

[5] P. Dinda, T. Gross, D. O'Halloron, E. Segall, E. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU task parallel program suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March 1994.

[6] J. Dongarra, J. D. Croz, J. Hammarling, and R. J. Hanson. An extended set of fortran basic algebra subprograms. *ACM Trans. Math. Softw.*, 14:1–17, 1988.

[7] I. Foster, D. R. Kohr, Jr., R. Krishnaiyer, and A. Choudhary. A Library-Based Approach to Task Parallelism in a Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 45(2):148–158, Sept. 1997.

[8] T. Gross, D. O'Halloron, E. Stichnoth, and J. Subhlok. Exploiting task and data parallelism on a multicomputer. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, May 1993.

[9] C. Koebel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[10] M. Vanneschi. Heterogeneous HPC Environments. In *Proc. 4th Int. Conf. Euro-Par'98*, pages 21–34, Southampton, UK, Sept. 1998. LNCS 1470 Spinger-Verlag.

[11] M. Vanneschi. PQE2000: HPC Tools for industrial applications. *IEEE Concurrency*, 6(4), October-December 1998.